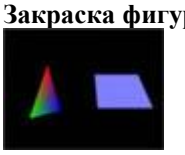
Народный учебник по OpenGL

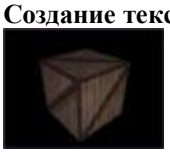
http://pmg.org.ru/nehe/index.html

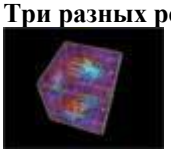
Уроки по OpenGL с сайта NeHe

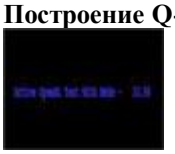
Иркутск 2005

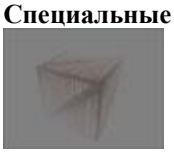
Уроки по OpenGL с сайта NeHe

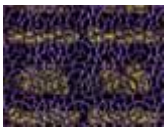












СТР. 101

Урок 20. Маскирование.
Использование маски изображения для создания действительной прозрачности при выводе текстур.

СТР. 109

Урок 21. Линии, сглаживание, синхронизация, ортографическая проекция и звуки.
Довольно большой урок, в котором приведен пример небольшой двухмерной игры. Все что нужно для 2D: по-пиксельный 2D экран, по-пиксельный вывод.

СТР. 134

Урок 22. Наложение микрорельефа методом тиснения, мультитекстурирование и использование расширений OpenGL.
Вроде бы обычный куб, но с микрорельефом. Возможно это Вам покажет не таким простым делом.

СТР. 157

Урок 23. Квадратирование со сферическим наложением в OpenGL.
Формирование и наложение текстур окружения на квадратичные объекты. Для эмуляции отражений от металлических и зеркальных поверхностей.

СТР. 161

Урок 24. Лексемы, Расширения, Вырезка и Загрузка TGA
В этом уроке вы поймете как вывести список доступных расширений вашей видеокарты с прокруткой в окне. Плюс загрузка и работа с TGA.

СТР. 173

Урок 25. Морфинг и загрузка объектов из файла.
Ясно и просто о морфинге, который позволяет перевоплотить сферу в тор, а тор в трубку.

СТР. 184

Урок 26. Реалистичное отражение с использование буфера шаблона и отсечения.
Буфер шаблона и смешивание в борьбе за получение реалистичных теней.

СТР. 195

Урок 27. Тени.
Мир без теней плоский. Взяв на вооружение буфер трафарета и бесконечность можно отбросить неплохую тень.

СТР. 203

Урок 28. Фрагменты поверхностей Безье.
Надоели плоские поверхности? Ощутите приятную выпуклость кривых поверхностей Безье!

СТР. 211

Урок 29. Блиттер-функция и чтение не обработанных текстур.
Самостоятельное смешивание изображений поможет Вам в создании процедурных текстур.

СТР. 222

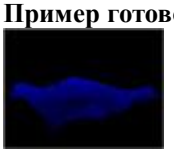
Урок 30. Определение столкновений и моделирование законов физики.
Как найти столкновения между плоскостью, цилиндром, и сферой. Как иммитировать физику, взрывы.

СТР. 235

Урок 31. Визуализация моделей Milkshape 3D
Описание как вывести произвольные 3D модели сохраненные в простом формате.

СТР. 242

Урок 32. Выбор, альфа смешивание, альфа тест, сортировка.
Пример готовой небольшой игры, где активно стреляют, и по этому необходимо использовать ВЫБОР.



СТР. 262

Урок 34. Построение красивых ландшафтов с помощью карты высот.
Скрещивая квадраты и картинки можно получить холмы, равнины и овраги. Просто и надежно.

СТР. 269

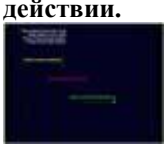
Урок 35. Проигрывание AVI файлов в OpenGL.
В этом уроке детально препарирован способ, как внутренности AVI-файла запихнуть в текстуру.

СТР. 281

Урок 36. Радиальное размытие и текстурный рендеринг
Очень интересный способ извлечения из OpenGL не свойственных ему возможностей!

СТР. 289

Урок 37. Мультипликационное закрашивание.
Добро пожаловать в мир мультиков. Один из интересных приемов не фотореалистической визуализации в действии.



СТР. 296

Урок 39. Введение в физический симулятор.
Симуляция поведения массы в условиях воздействия на нее сил гравитации, пружины и просто движущейся с постоянной скоростью.

СТР. 303

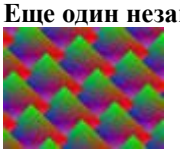
Урок 40. Моделирование движений веревки.
Описание того, как помахать пружинистой тяжеленной веревкой и потаскать ее по столу.

СТР. 312

Урок 41. Объемный туман и загрузка изображений через интерфейс IPicture.
Любые картинки от IPicture, клевый туман от glFog.

СТР. 322

Урок 43. FreeType шрифты в OpenGL.
Еще один независимый способ получить красивые буквы на экране, используя двухбайтные растры.



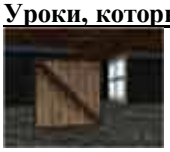
СТР. 332

Урок 46. Полноэкранное сглаживание.
Аппаратное сглаживание поможет Вам истребить зазубрины и артефакты по всему экрану легко и просто.

СТР. 337

Урок 48. Вращение объектов с помощью класса ArcBall.
Кручение и верчение объемов используя готовую функциональность класса ArcBall.

<u>Уроки, которые не опубликованы на сайте NeHe, но которые базируются на его уроках.</u>



СТР. 340

Урок X1. Улучшенная обработка ввода с использованием DirectInput и Windows.
Вы думали, что DirectInput и OpenGL две вещи несовместные, ан, нет, даже очень дружат.

СТР. 348

Урок X2. Отсечение по пирамиде видимости в OpenGL.
Простой и понятный способ выбросить за борт лишние объекты.

Урок 1. Инициализация в Windows

Setting Up An OpenGL Window

Я начинаю это пособие непосредственного с кода, разбитого на секции, каждая из которых будет подробно комментироваться. Первое, что вы должны сделать - это создать проект в Visual C++. Если это для вас затруднительно, то вам стоит для начала изучить C++, а уже затем переходить к OpenGL.

После того как вы создадите новое приложение в Visual C++ (причем Win32 приложение, а не консольное), Вам надо будет добавить для сборки проекта библиотеки OpenGL. В меню Project/setting, выберите закладку LINK. В строке "Object/Library Modules" добавьте "OpenGL32.lib GLu32.lib GLaux.lib". Затем кликните по OK. Теперь все готово для создания программы с OpenGL.

Первые четыре строки, которые вы введете, сообщают компилятору какие библиотечные файлы использовать. Они должны выглядят так:

```
#include <windows.h>                // Заголовочный файл для Windows
#include <gl\gl.h>                  // Заголовочный файл для OpenGL32 библиотеки
#include <gl\glu.h>                 // Заголовочный файл для GLu32 библиотеки
#include <gl\glaux.h>               // Заголовочный файл для GLaux библиотеки
```

Далее, необходимо инициализировать все переменные, которые будут использованы в вашей программе. Эта программа будет создавать пустое OpenGL окно, поэтому мы не будем нуждаться в большом количестве переменных. То немногое, что мы устанавливаем - очень важно, и будет использоваться в каждой программе с OpenGL, которую вы напишите с использованием этого кода.

Первые две строки устанавливают Контексты Рендеринга, которые связывает вызовы OpenGL с окном Windows. Контекст Рендеринга OpenGL определен как hRC. Для того чтобы рисовать в окне, вам необходимо создать Контекст Устройства Windows, который определен как hDC. DC соединяет окно с GDI. RC соединяет OpenGL с DC.

```
static HGLRC hRC;           // Постоянный контекст рендеринга
static HDC hDC;             // Приватный контекст устройства GDI
```

Последняя переменная, в которой мы будем нуждаться, это массив, который мы используем для отслеживания нажатия клавиш на клавиатуре. Есть много путей следить за нажатиями на клавиатуре, но я использую этот путь. При этом можно отслеживать нажатие более чем одной клавиши одновременно.

```
BOOL    keys[256];          // Массив для процедуры обработки клавиатуры
```

В следующей секции кода будут произведены все настройки для OpenGL. Мы установим цвет для очистки экрана, включим глубину буфера, разрешим плавное сглаживание цветов, и что наиболее важно, мы установим рендеринг на экран в перспективе, используя ширину и высоту окна. Эта процедура не должна быть вызвана до тех пор, пока OpenGL окно будет сделано.

```
GLvoid InitGL(GLsizei Width, GLsizei Height)    // Вызвать после создания окна GL
{
```

В следующей строке устанавливается цвет, которым будет очищен экран. Для тех, кто не знает, как устроены цвета, я постараюсь кратко объяснять. Все значения могут быть в диапазоне от 0.0f до 1.0f, при этом 0.0 самый темный, а 1.0 самый светлый. Первое число в glClearColor - это интенсивность красного, второе – зеленного, третье – синего. Наибольшее значение – 1.0f, является самым ярким значением данного цвета. Последнее число - для альфа значения. Когда начинается очистка экрана, я никогда не волнуюсь о четвертом числе. Пока оно будет 0.0f. Как его использовать, я объясню в другом уроке.

Поэтому, если вы вызвали glClearColor(0.0f,0.0f,1.0f,0.0f) вы произведете очистку экрана, с последующим закрашиванием его в ярко синий цвет. Если вы вызвали glClearColor(0.5f,0.0f,0.0f,0.0f) экран будет заполнен умеренно красным цветом. Не очень ярким (1.0f) и не темным (0.0f), а именно умеренно красным. Для того чтобы сделать белый фон, вы должны установить все цвета в (1.0f). Черный - как можно ниже (0.0f).

```
glClearColor(0.0f, 0.0f, 0.0f, 0.0f);   // Очистка экрана в черный цвет
```

Следующие три строки создают Буфер Глубины. Думайте о буфере глубины как о слоях на экране. Буфер глубины указывает, как далеко объекты находятся от экрана. Мы не будем реально использовать буфер глубины в этой

программе, но любая программа с OpenGL, которая рисует на экране в 3D будет его использовать. Он позволяет сортировать объекты для отрисовки, поэтому квадрат расположенный под кругом не изображен будет поверх круга. Буфер глубины очень важная часть OpenGL.

```
    glClearDepth(1.0);                  // Разрешить очистку буфера глубины
    glDepthFunc(GL_LESS);               // Тип теста глубины
    glEnable(GL_DEPTH_TEST);            // разрешить тест глубины
```

Следующие пять строк разрешают плавное цветовое сглаживание (которое я буду объяснять позднее) и установку экрана для перспективного просмотра. Отдаленные предметы на экране кажутся меньшими, чем ближние. Это придает сцене реалистичный вид. Перспектива вычисляется с углом просмотра 45 градусов на основе ширины и высоты окна. 0.1f, 100.0f глубина экрана.

glMatrixMode(GL_PROJECTION) сообщает о том, что следующие команды будут воздействовать на матрицу проекции. glLoadIdentity() – это функция работает подобно сбросу. Раз сцена сброшена, перспектива вычисляется для сцены. glMatrixMode(GL_MODELVIEW) сообщает, что любые новые трансформации будут воздействовать на матрицу просмотра модели. Не волнуйтесь, если вы что-то не понимаете этот материал, я буду обучать всему этому в дальнейших уроках. Только запомните, что НАДО сделать, если вы хотите красивую перспективную сцену.

```
    glShadeModel(GL_SMOOTH);        // разрешить плавное цветовое сглаживание
    glMatrixMode(GL_PROJECTION);    // Выбор матрицы проекции
    glLoadIdentity();               // Сброс матрицы проекции
    gluPerspective(45.0f,(GLfloat)Width/(GLfloat)Height,0.1f,100.0f);
                    // Вычислить соотношение геометрических размеров для окна
    glMatrixMode(GL_MODELVIEW);     // Выбор матрицы просмотра модели
}
```

Следующая секция кода очень простая, по сравнению с предыдущим кодом. Это функция масштабирования сцены, вызываемая OpenGL всякий раз, когда вы изменяете размер окна (допустим, что вы используете окна чаще, чем полноэкранный режим, который мы не рассматриваем). Даже если вы не делаете изменение размеров окна (например, если вы находитесь в полноэкранном режиме), эта процедура все равно должна быть вызвана хоть один раз, обычно во время запуска программы. Замечу, что сцена масштабируется, основываясь на ширине и высоте окна, которое отображается.

```
GLvoid ReSizeGLScene(GLsizei Width, GLsizei Height)
{
    if (Height==0)          // Предотвращение деления на ноль, если окно слишком мало
        Height=1;

    glViewport(0, 0, Width, Height);
        // Сброс текущей области вывода и перспективных преобразований

    glMatrixMode(GL_PROJECTION);            // Выбор матрицы проекций
    glLoadIdentity();                       // Сброс матрицы проекции

    gluPerspective(45.0f,(GLfloat)Width/(GLfloat)Height,0.1f,100.0f);
        // Вычисление соотношения геометрических размеров для окна
    glMatrixMode(GL_MODELVIEW);     // Выбор матрицы просмотра модели
}
```

В этой секции содержится весь код для рисования. Все, что вы планируете для отрисовки на экране, будет содержатся в этой секции кода. В каждом уроке, после этого будет добавлять код в эту секцию программы. Если вы уже понимаете OpenGL, вы можете попробовать добавить в код простейшие формы на OpenGL, ниже вызова glLoadIdentity(). Если вы новичок в OpenGL, подождите до следующего моего урока. Сейчас все что мы сделаем, это очистка экрана цветом, который мы определили выше, очистка буфера глубины и сброс сцены.

```
GLvoid DrawGLScene(GLvoid)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        // очистка Экрана и буфера глубины
    glLoadIdentity();
        // Сброс просмотра
}
```

Следующая секция кода наиболее важная секция в этой программе. Это установка окна Windows, установка формата пикселя, обработка при изменении размеров, при нажатии на клавиатуру, и закрытие программы.

Первые четыре строки делают следующее: переменная hWnd – является указателем на окно. Переменная message – сообщения, передаваемые вашей программе системой. Переменные wParam и lParam содержат информацию, которая посылается вместе с сообщением, например такая как ширина и высота окна.

```
LRESULT CALLBACK WndProc(       HWND    hWnd,
                        UINT    message,
                        WPARAM  wParam,
                        LPARAM  lParam)
```

Код между скобками устанавливает формат пикселей. Я предпочитаю не использовать режим индексации цвета. Если вы не знаете, что это означает, не заботьтесь об этом. Формат описания пикселя описывает, как OpenGL будет выводить в окно. Большинство кода игнорируется, но зачастую это необходимо. Я буду помещать короткий комментарий для каждой строки. Знак вопроса означает, что я не уверен, что это строка кода делает (я только человек!).

```
{
    RECT    Screen;         // используется позднее для размеров окна
    GLuint  PixelFormat;
    static  PIXELFORMATDESCRIPTOR pfd=
    {
        sizeof(PIXELFORMATDESCRIPTOR),  // Размер этой структуры
    1,                          // Номер версии (?)
    PFD_DRAW_TO_WINDOW |            // Формат для Окна
    PFD_SUPPORT_OPENGL |            // Формат для OpenGL
    PFD_DOUBLEBUFFER,               // Формат для двойного буфера
    PFD_TYPE_RGBA,                  // Требуется RGBA формат
    16,                         // Выбор 16 бит глубины цвета
    0, 0, 0, 0, 0, 0,                   // Игнорирование цветовых битов (?)
    0,                          // нет буфера прозрачности
    0,                          // Сдвиговый бит игнорируется (?)
    0,                          // Нет буфера аккумуляции
    0, 0, 0, 0,                         // Биты аккумуляции игнорируются (?)
    16,                         // 16 битный Z-буфер (буфер глубины)
    0,                          // Нет буфера траффарета
    0,                          // Нет вспомогательных буферов (?)
    PFD_MAIN_PLANE,                 // Главный слой рисования
    0,                          // Резерв (?)
    0, 0, 0                         // Маски слоя игнорируются (?)
    };
```

Эта секция кода обрабатывает системные сообщения. Они генерируются, когда вы выходите из программы, нажимаете на клавиши, передвигаете окно, и так далее, каждая секция "case" обрабатывает свой тип сообщения. Если вы что вставите в эту секцию, не ожидайте, что ваш код будет работать должным образом, или вообще работать.

```
    switch (message)        // Тип сообщения
    {
```

WM_CREATE сообщает программе, что оно должно быть создано. Вначале мы запросим DC (контекст устройства) для вашего окна. Помните, без него мы не можем рисовать в окно. Затем мы запрашиваем формат пикселя. Компьютер будет выбирать формат, который совпадает или наиболее близок к формату, который мы запрашиваем. Я не делаю здесь множества проверок на ошибки, чтобы сократить код, но это неправильно. Если что-то не работает, я просто добавляю необходимый код. Возможно, вы захотите посмотреть, как работают другие форматы пикселей.

```
    case WM_CREATE:
        hDC = GetDC(hWnd);      // Получить контекст устройства для окна
        PixelFormat = ChoosePixelFormat(hDC, &pfd);
            // Найти ближайшее совпадение для нашего формата пикселов
```

Если подходящий формат пикселя не может быть найден, будет выведено сообщение об ошибке с соответствующем уведомлением. Оно будет ждать, когда вы нажмете на OK, до выхода из программы.

```
if (!PixelFormat)
	{
		MessageBox(0,"Can't Find A Suitable
		PixelFormat.","Error",MB_OK|MB_ICONERROR);
		PostQuitMessage(0);
		// Это сообщение говорит, что программа должна завершится
	break;  // Предтовращение повтора кода
	}
```

Если подходящий формат найден, компьютер будет пытаться установить формат пикселя для контекста устройства. Если формат пикселя не может быть установлен по какой-то причине, выскочит сообщение об ошибке, что формат пикселя не найден, и будет ожидать, пока Вы не нажмете кнопку OK, до выхода из программы.

```
if(!SetPixelFormat(hDC,PixelFormat,&pfd))
	{
		MessageBox(0,"Can't Set The
		PixelFormat.","Error",MB_OK|MB_ICONERROR);
		PostQuitMessage(0);
		break;
	}
```

Если код сделан, как показано выше, будет создан DC (контекст устройства), и установлен подходящий формат пикселя. Сейчас мы создадим Контекст Рендеринга, для этого OpenGL использует DC. wglCreateContext будет захватывать Контекст Рендеринга и сохранять его в переменной hRC. Если по какой-то причине Контекст Рендеринга не доступен, выскочит сообщение об ошибке. Нажмите OK для вызова программы.

```
hRC = wglCreateContext(hDC);
if(!hRC)
	{
	MessageBox(0,"Can't Create A GL Rendering
	Context.","Error",MB_OK|MB_ICONERROR);
	PostQuitMessage(0);
	break;
	}
```

Сейчас мы имеем Контекст Рендеринга, и нам необходимо сделать его активным, для того чтобы OpenGL мог рисовать в окно. Снова, если по не которой причине это не может быть сделано, выскочит сообщение об ошибке. Кликните OK в окошке ошибки для выхода из программы.

```
if(!wglMakeCurrent(hDC, hRC))
	{
	MessageBox(0,"Can't activate GLRC.","Error",MB_OK|MB_ICONERROR);
	PostQuitMessage(0);
	break;
	}
```

Если все прошло удачно, то у нас есть все для того, чтобы создать область рисования OpenGL. GetClientRect возвратит нам ширину и высоту окна. Мы запомним ширину справа, и высоту снизу. После того как мы получили ширину и высоту, инициализируем экран OpenGL. Мы делаем это при помощи вызова InitGL, передавая в параметрах право и низ (ширину и высоту).

```
	GetClientRect(hWnd, &Screen);
	InitGL(Screen.right, Screen.bottom);
	break;
```

WM_DESTROY и WM_CLOSE очень похожи. Программа будет посылать это сообщение каждый раз, когда вы выходите из программы, нажав ALT-F4, или если вы послали PostQuitMessage(0) также как мы делали, когда происходила ошибка.

ChangeDisplaySettings(NULL,0) будет переключать разрешение рабочего стола обратно, делая его таким, каким мы переключались из него в полноэкранный режим. ReleaseDC(hWnd,hDC) уничтожает контекст устройства окна. По существу это уничтожает окно OpenGL.

```
        case WM_DESTROY:
        case WM_CLOSE:
        ChangeDisplaySettings(NULL, 0);

        wglMakeCurrent(hDC,NULL);
        wglDeleteContext(hRC);
        ReleaseDC(hWnd,hDC);

        PostQuitMessage(0);
        break;
```

WM_KEYDOWN вызывается всякий раз при нажатии клавиши. Клавиша, которая была нажата, сохраняется в переменной wParam. Итак, что же делает следующий код... Скажем, я нажал 'A'. Буква фактически – это число, которое ее представляет. Поэтому в ячейку, которая представляет 'A' заносится TRUE. Позднее, в коде, если я проверю состояние ячейки и увижу TRUE, то я знаю, что клавиша 'A' действительно в этот момент нажата.

```
        case WM_KEYDOWN:
        keys[wParam] = TRUE;
        break;
```

WM_KEYUP вызывается всякий раз, когда клавиша отпускается. Клавиша, которая отжата, также сохраняется в переменной wParam. Поэтому, когда я отпускаю клавишу 'A', это делает ячейку для клавиши 'A' равной FALSE. Когда я проверю ячейку, для того чтобы увидеть нажата ли клавиша 'A', она вернет FALSE, что означает "нет, она не нажата".

```
        case WM_KEYUP:
        keys[wParam] = FALSE;
        break;
```

И последнее, что я сделаю - обработаю изменение размеров окна. Возможно, кажется, что бесмыслено добавлять этот код, когда программа запущена в полноэкранном режиме, но без этого кода, экран OpenGL не появится. Поверьте, мне это очень важно.

Всякий раз сообщение WM_SIZE посылается Windows с двумя параметрами - новая ширина, и новая высота экрана. Эти параметры сохранены в LOWORD(lParam) и HIWORD(lParam). Поэтому вызов ReSizeGLScene изменяет размеры экрана. Это передает высоту и ширину в эту секцию кода.

```
        case WM_SIZE:
        ReSizeGLScene(LOWORD(lParam),HIWORD(lParam));
        break;
```

Затем, дадим Windows обработать все сообщения, которые мы не обрабатываем и завершим процедуру.

```
    default:
    return (DefWindowProc(hWnd, message, wParam, lParam));
    }
    return (0);
}
```

Это то место, где начинается программа, где создается окно, где делается практически все, кроме рисования. Мы начинаем с создания окна.

```
int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,
        LPSTR lpCmdLine,int nCmdShow)
{
    MSG         msg;    // Структура сообщения Windows
    WNDCLASS    wc; // Структура класса Windows для установки типа окна
    HWND        hWnd;   // Сохранение дискриптора окна
```

Флаги стиля CS_HREDRAW и CS_VREDRAW принуждают перерисовать окно всякий раз, когда оно перемещается. CS_OWNDC создает скрытый DC для окна. Это означает, что DC не используется совместно нескольким приложениями. WndProc - процедура, которая перехватывает сообщения для программы. hIcon установлен равным нулю, это означает, что мы не хотим ICON в окне, и для мыши используем стандартный указатель. Фоновый цвет не имеет значения (мы установим его в GL). Мы не хотим меню в этом окне, поэтому мы используем установку его в NULL, и имя класса – это любое имя которое вы хотите.

```
wc.style            = CS_HREDRAW | CS_VREDRAW | CS_OWNDC;
wc.lpfnWndProc      = (WNDPROC) WndProc;
wc.cbClsExtra       = 0;
wc.cbWndExtra       = 0;
wc.hInstance        = hInstance;
wc.hIcon            = NULL;
wc.hCursor          = LoadCursor(NULL, IDC_ARROW);
wc.hbrBackground    = NULL;
wc.lpszMenuName     = NULL;
wc.lpszClassName    = "OpenGL WinClass";
```

Сейчас мы регистрируем класс. Если произошла ошибка, появится соответствующее сообщение. Кликните на OK на окно с сообщение об ошибке и будете выкинуты из программы.

```
if(!RegisterClass(&wc))
{
MessageBox(0,"Failed To Register The Window
Class.","Error",MB_OK|MB_ICONERROR);
return FALSE;
}
```

Сейчас мы сделаем окно. Не смотря на то, что мы делаем окно здесь, это не вызовет OpenGL до тех пор, пока сообщение WM_CREATE не послано. Флаги WS_CLIPCHILDREN и WS_CLIPSIBLINGS требуются для OpenGL. Очень важно, чтобы вы добавили их здесь. Я люблю использовать всплывающее окно, оно хорошо работает в полноэкранном режиме.

```
hWnd = CreateWindow(
"OpenGL WinClass",
"Jeff Molofee's GL Code Tutorial ... NeHe '99", // Заголовок вверху окна

WS_POPUP |
WS_CLIPCHILDREN |
WS_CLIPSIBLINGS,

0, 0,               // Позиция окна на экране
640, 480,           // Ширина и высота окна

NULL,
NULL,
hInstance,
NULL);
```

Далее - обычная проверка на ошибки. Если окно не было создано по какой-то причине, сообщение об ошибке выскочит на экран. Давите OK и завершайте программу.

```
if(!hWnd)
{
MessageBox(0,"Window Creation Error.","Error",MB_OK|MB_ICONERROR);
    return FALSE;
}
```

Следующая секция кода вызывает у многих людей массу проблем … переход в полноэкранный режим. Здесь важна одна вещь, которую вы должны запомнить, когда переключаетесь в полноэкранный режим - сделать ширину и высоту в полноэкранном режиме необходимо туже самую, что и ширина и высота, которую вы сделали в своем окне.

Я не устанавливаю глубину цвета, когда я переключаю полноэкранный режим. Всякий раз, когда я пробовал переключать глубину цвета, я получал сверхъестественные запросы от Windows чтобы сделать перезагрузку компьютера для переключения нового режима цвета. Я не уверен, надо ли удовлетворять это сообщение, но я решил оставлять компьютер с той глубиной цвета, которая была до запуска GL программы.

Важно отметить, что этот код не будет скомпилирован на Си. Это файл должен быть сохранен как .CPP файл.

```
DEVMODE dmScreenSettings;                   // Режим работы

memset(&dmScreenSettings, 0, sizeof(DEVMODE));          // Очистка для хранения установок
dmScreenSettings.dmSize = sizeof(DEVMODE);              // Размер структуры Devmode
dmScreenSettings.dmPelsWidth    = 640;                  // Ширина экрана
dmScreenSettings.dmPelsHeight   = 480;                  // Высота экрана
dmScreenSettings.dmFields       = DM_PELSWIDTH | DM_PELSHEIGHT; // Режим Пиксела
ChangeDisplaySettings(&dmScreenSettings, CDS_FULLSCREEN);
    // Переключение в полный экран
```

ShowWindow название этой функции говорит само за себя - она показывает окно, которое вы создали на экране. Я люблю это делать, после того как я переключусь в полноэкранный режим, хотя я не уверен, что это имеет значения. UpdateWindow обновляет окно, SetFocus делает окно активным, и вызывает wglMakeCurrent(hDC,hRC) чтобы убедиться, что Контекст рендеринга не освобожден.

```
    ShowWindow(hWnd, SW_SHOW);
    UpdateWindow(hWnd);
    SetFocus(hWnd);
```

Теперь мы создадим бесконечный цикл. Есть только один момент выхода из цикла, - когда нажали ESC. При этом программе будет отправлено сообщение о выходе, и она прервется.

```
    while (1)
    {
        // Обработка всех сообщений
        while (PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE))
        {
            if (GetMessage(&msg, NULL, 0, 0))
            {
                TranslateMessage(&msg);
                DispatchMessage(&msg);
            }
            else
            {
                return TRUE;
            }
        }
```

DrawGLScene вызывает ту часть программы, которая фактически рисует объекты OpenGL. В этой программе мы оставим эту часть секции пустой, все что будет сделано - очистка экрана черным цветом. В следующих уроках я покажу, как применять OpenGL для рисования.

SwapBuffers(hDC) очень важная команда. Мы имеем окно с установленной двойной буферизацией. Это означает, что изображение рисуется на скрытом окне (называемым буфером). Затем, мы говорим компьютеру переключить буфера, скрытый буфер копируется на экран. При этом получается плавная анимация без рывков, и зритель не замечает отрисовку объектов.

```
        DrawGLScene();                  // Нарисовать сцену
        SwapBuffers(hDC);               // Переключить буфер экрана
        if (keys[VK_ESCAPE]) SendMessage(hWnd,WM_CLOSE,0,0);    // Если ESC - выйти
        }
}
```

В этом уроке я попытался объяснить как можно больше деталей каждого шага запутанной установки, и создания ваших собственных полноэкранных OpenGL программ, которые будут завершаться при нажатии ESC. Я потратил 3

дня и 13 часов для написания этого урока. Если вы имеете любые комментарии или вопросы, пожалуйста, пошлите их мне по электронной почте. Если вы ощущаете, что я некорректно комментировал что-то или что код должен быть лучше в некоторых секциях по некоторым причинам, пожалуйста, дайте мне знать. Я хочу сделать уроки по OpenGL хорошими насколько смогу. Я заинтересован в обратной связи.

Примечание переводчика: исходные коды для этого урока и других уроков, есть на сайте NeHe. Чтобы найти их, Вам надо перейти на оригинальный урок по ссылке, которая расположена вначале текста любого переведенного урока. Затем, надо перейти в конец оригинального (англоязычного) урока и там найти ссылки на нужный архив с нужным исходным кодом, для своего компилятора, или платформы, или языка (например, Delphi). Поместить все исходные коды для уроков NeHe на этом сайте, просто невозможно!

Еще одно примечание переводчика: в связи с тем, что NeHe иногда изменяет уроки, но при этом нет возможности понять когда он это сделал (нет даты последнего изменения урока), то бывает так, что текст переведенных уроков незначительно отличается от текущих оригинальных уроков. Не пугайтесь этого, смысл урока при этом значительно не меняется.

Урок 2. Отображение полигонов

Your First Polygon

В предыдущем уроке было рассмотрено создание OpenGL окна. Теперь мы изучим создание таких фигур как треугольники и квадраты, при помощи GL_TRIANGLES и GL_QUADS.

Для создания приложения мы будем использовать код предыдущего примера, только добавим код в функцию DrawGLScene. Все ее изменения приводятся ниже. Если вы планируете менять предыдущий урок, просто замените функцию DrawGLScene следующим кодом, или просто добавьте те строки, которые там отсутствуют.

```
GLvoid DrawGLScene(GLvoid)
{
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);     // Очистка экрана
                                        // и буфера глубины
      glLoadIdentity();                         // Сброс просмотра
```

Вызов функции gLoadIdentity() устанавливает начало системы координат в центр экрана, причем ось X идет слева направо, ось Y вверх и вниз, а ось Z к и от наблюдателя. Центр OpenGL экрана находится в точке 0, 0, 0. Координаты, расположенные слева, снизу и вглубь от него, имеют отрицательное значение, расположенные справа, сверху и по направлению к наблюдателю – положительное.

Функция glTranslate(x, y, z) перемещает оси координат на указанные значения. Следующая строчка кода перемещает ось X на 1.5 единиц и ось Z на 6 единиц. Следует заметить, что перевод осей координат осуществляется не относительно центра экрана, а от их текущего расположения.

```
      glTranslatef(-1.5f,0.0f,-6.0f);         // Сдвинемся влево на 1.5 единицы и
                                        // в экран на 6.0
```

Теперь, когда мы переместились в левую часть экрана и установили более удобный вид (в глубину на 6 единиц), отобразим геометрическую фигуру. Функция glBegin(GL_TRIANGLES) означает, что мы начинаем рисовать треугольник, и далее следует перечисление его вершин. После указания всех вершин, производится вызов функции glEnd(). Обычно треугольники на большинстве видеокарт отображаются наиболее быстро, но если есть желание отобразить иную фигуру, вместо GL_TRIANGLE используется другое значение, например: для создания четырехугольника указывается GL_QUADS, если вершин больше чем 4, то GL_POLYGONS.

В нашем примере рисуем только один треугольник. Если есть желание изобразить еще один треугольник, добавьте часть кода, следующую за описанием первого треугольника. В это случае все шесть строк кода следует размещать между glBegin(GL_TRIANGLES) и glEnd(). Нет нужды выделять каждый треугольник этими командами, после обработки трех вершин OpenGL сам перейдет к созданию новой фигуры. Это же относится и к четырехугольникам. Полигоны (GL_POLYGONS) в свою очередь могут иметь любое количество вершин, и поэтому нет разницы, сколько описателей располагалось между строками glBegin(GL_POLYGONS) и glEnd().

Первая строка после glBegin описывает первую вершину нашего полигона. Функция glVertex3f() получает в качестве параметров ее X, Y и Z координаты. Первая вершина треугольника смещена только от оси Y на 1, таким образом, мы

расположим ее точно в центре и она будет самой верхней. Следующая вершина будет располагаться на оси X слева от центра и на оси Y вниз от центра. Эта вершина будет расположена внизу слева. Третья вершина будет справа и снизу от центра. Функция glEnd() указывает OpenGL, что вершин больше не будет. Результатом всего этого будет залитый цветом по умолчанию треугольник.

```
glBegin(GL_TRIANGLES);
    glVertex3f( 0.0f, 1.0f, 0.0f);  // Вверх
    glVertex3f(-1.0f,-1.0f, 0.0f);  // Слева снизу
    glVertex3f( 1.0f,-1.0f, 0.0f);  // Справа снизу
glEnd();
```

Теперь у нас есть треугольник, изображенный в правой части экрана. Нам нужно переместиться в левую часть, для этого снова используем функцию glTranslate(). Так как мы в прошлый раз перемещались влево на 1.5 единицы, необходимо переместиться на 3.0 единицы вправо (1.5 единицы – это будет центр, плюс еще 1.5 единицы для правого края).

```
glTranslatef(3.0f,0.0f,0.0f);          // Сдвинем вправо на 3 единицы
```

Здесь мы изобразим квадрат. Так как он является четырехсторонним полигоном, мы будем использовать GL_QUADS. Создание квадрата напоминает создание треугольника, правда указывать нужно четыре вершины. Они будут идти в следующем порядке – левая вверху, правая вверху, правая снизу и левая снизу.

```
glBegin(GL_QUADS);
    glVertex3f(-1.0f, 1.0f, 0.0f);  // Слева вверху
    glVertex3f( 1.0f, 1.0f, 0.0f);  // Справа вверху
    glVertex3f( 1.0f,-1.0f, 0.0f);  // Справа внизу
    glVertex3f(-1.0f,-1.0f, 0.0f);  // Слева внизу
glEnd();
}
```

В этом примере я попытался описать как можно больше деталей, указать каждый шаг создания полигонов на экране с использованием OpenGL. Если у вас есть, какие либо замечания, вопросы или комментарии, я жду ваших писем. Если вы нашли неправильно описание или ошибки в коде, пожалуйста, сообщите мне об этом, мне очень хотелось бы, чтобы эти описания были лучше.

Урок 3. Отображение цветов

Colors

В предыдущем уроке я показал, как можно отобразить на экране треугольник и квадрат. В этом уроке я научу вас отображать эти же фигуры, но в разных цветах. Квадрат мы зальем одним цветом, а вот треугольник будет залит тремя разными цветами (по одному на каждую вершину) с гладкими переходами.

Вы можете использовать код из предыдущего урока, изменив лишь процедуру DrawGLScene(). Я переписал ее содержимое, и, если вы планируете изменять предыдущий урок, вам нужно полностью ее заменить, или добавить те строки, которые там отсутствуют.

```
GLvoid DrawGLScene(GLvoid)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    glTranslatef(-1.5f,0.0f,-6.0f);
    glBegin(GL_TRIANGLES);
```

Если вы еще помните предыдущий урок, в этой секции мы рисовали треугольник в левой части экрана. Следующие строки кода используют команду glColor3f(r, g, b). Три ее параметра указывают насыщенность цвета красной, синей и зеленой составляющей. Каждый из них может принимать значение от 0.0f до 1.0f.

Мы установим красный цвет (полный красный, без зеленого и синего). Следующие строки кода указывают, что первая вершина (верхняя) будет иметь красный цвет. Все что мы будем рисовать дальше красным, до тех пор, пока мы его не сменим цвет.

```
        glColor3f(1.0f,0.0f,0.0f);      // Красный цвет
        glVertex3f( 0.0f, 1.0f, 0.0f);
```

Мы отобразили первую вершину, установив для нее красный цвет. Теперь добавим следующую вершину (левую нижнею), но установим для нее уже зеленый цвет.

```
        glColor3f(0.0f,1.0f,0.0f);      // Зеленный цвет
        glVertex3f(-1.0f,-1.0f, 0.0f);
```

Пришло время третьей и последней вершины (правый нижний угол). Перед тем как отобразить ее, мы установим синий цвет. После выполнения команды glEnd() треугольник будет залит указанными цветами. Так как мы указали для каждой вершины свой цвет, каждая часть фигуры будет залита по-разному. При переходе в другую вершину заливка будет плавно изменять свой цвет на цвет вершины. В середине треугольника все три цвета будут слиты в один.

```
        glColor3f(0.0f,0.0f,1.0f);      // Синий цвет
        glVertex3f( 1.0f,-1.0f, 0.0f);
    glEnd();
    glTranslatef(3.0f,0.0f,0.0f);
```

Теперь мы отобразим квадрат, но зальем его одним цветом. Очень важно помнить, что если вы установили какой-либо цвет, все примитивы в дальнейшем будет отображаться именно им. Каждый последующий проект, который вы будете создавать, так или иначе, будет использовать цвета. Если вы, например, создает сцену, где все фигуры текстурированы, цвет будет использоваться для тона текстур, и т.д.

Так как мы рисуем наш квадрат в одном цвете (для примера – в синем), для начала установим этот цвет, а затем отобразим саму фигуру. Синий цвет будет использоваться OpenGL для каждой вершины, так как мы не меняем его. В итоге мы получим синий квадрат.

```
    glColor3f(0.5f,0.5f,1.0f);      // Установим синий цвет только один раз
    glBegin(GL_QUADS);
        glVertex3f(-1.0f, 1.0f, 0.0f);
        glVertex3f( 1.0f, 1.0f, 0.0f);
        glVertex3f( 1.0f,-1.0f, 0.0f);
        glVertex3f(-1.0f,-1.0f, 0.0f);
    glEnd();
}
```

В этом примере я пытался, как можно детальнее описать процедуру установки цветов для вершин, показать различие между однотонной заливкой и разноцветной сглаженной. Попробуйте, для тренировки, изменять составляющие красного, зеленого и синего компонентов цвета. Посмотрите, какие будут результаты ваших экспериментов. Если у вас есть, какие либо замечания, вопросы или комментарии, я жду ваших писем. Если вы нашли неправильно описание или ошибки в коде, пожалуйста, сообщите мне об этом, мне очень хотелось бы, чтобы эти описания были лучше.

Урок 4. Вращение полигонов

Rotation

В прошлом уроке я научил Вас как закрашивать треугольники и четырехугольники. В этом уроке я научу Вас как вращать эти закрашенные объекты вдоль их осей.

Мы будем использовать код из последнего урока, добавляя новые строчки кода. Я перепишу целую секцию кода ниже, чтобы вы могли понять, что добавлено, а что заменено.
Вначале мы добавим две переменные для хранения угла вращения каждого объекта. Мы сделаем это вначале программы. Посмотрите ниже, я добавил две строки после объявления переменной BOOL keys[256]. В этих строках объявляются две переменные с плавающей запятой, которые мы можем использовать для очень точного поворота объектов. Числа с плавающей запятой учитывают значения меньше единицы. Вместо использования 1, 2, 3 для угла,

мы можем использовать 1.1, 1.7, 2.3 или даже 1.015 для точности. Вы увидете, что числа с плавающей запятой неотемлимая часть программирования на OpenGL.

```
#include <windows.h>      // Заголовочный файл для Windows
#include <gl\gl.h>        // Заголовочный файл для OpenGL32 библиотеки
#include <gl\glu.h>       // Заголовочный файл для GLu32 библиотеки
#include <gl\glaux.h>     // Заголовочный файл для GLaux библиотеки

static HGLRC hRC;      // Постоянный контекст рендеринга
static HDC hDC;        // Приватный контекст устройства GDI

BOOL    keys[256];     // Массив для процедуры обработки клавиатуры

GLfloat rtri;          // Угол для треугольник
GLfloat rquad;         // Угол для четырехугольника
```

Необходимо модифицировать код в DrawGLScene(). Я буду переписывать всю процедуру. Это будет сделать легко для Вас, так как Вы увидите какие изменения я сделал. Я объясню почему некоторые строки были модифицированы, и какие линии добавлены. Следующая секция кода, такая же как в последнем уроке.

```
GLvoid DrawGLScene(GLvoid)
{
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);     // Очистка экрана
                                        //      и буфера глубины
glLoadIdentity();                        // Сброс просмотра
glTranslatef(-1.5f,0.0f,-6.0f);                 // Сдвиг в глубь экрана и влево
```

Следующая строка новая. glRotatef(Angle,Xtrue,Ytrue,Ztrue) отвечает за вращения объекта вдоль оси. Вы многое получите от использования этой команды. Угол некоторое число (обычно переменная), которое задает насколько Вы хотите повернуть объект. Xtrue, Ytrue и Ztrue или 0.0f или 1.0f. Если один из параметров равен 1.0f, OpenGL будет вращать объект вдоль соответствующей оси. Поэтому если Вы имеете glRotatef(10.0f,0.0f,1.0f,0.0f), объект будет поварачиваться на 10 градусов по оси Y. Если glRotatef(5.0f,1.0f,0.0f,1.0f), объект будет поварачиваться на 5 градусов по обеим осям X и Z.

Чтобы лучше понять вращения по осям X, Y и Z я объясню это на примерах.

Ось X - предположим Вы работаете за токарным станком. Заготовка перемещается слева направо (также как ось X в OpenGL). Заготовка вращается вокруг оси X. Также мы вращаем что-то вокруг оси X в OpenGL.

Ось Y- Представьте что Вы стоите посреди поля. Огромный торнадо приближается к Вам. Центр торнадо перемещается от земли в небо (верх и низ, подобно оси Y в OpenGL). Предметы захваченные торнадо кружаться вдоль оси Y (центр торнадо) слева направо или справо налево. Когда вы вращаете что-то вокруг оси Y в OpenGL, это что-то будет вращаться также.

Ось Z - Вы смотрите на переднюю часть вентилятора. Передняя часть вентилятора ближе к Вам, а дальняя часть дальше от Вас (также как ось Z в OpenGL). Лопасти вентилятора вращаются вдоль оси Z (центр вентилятора) по часовой или против часовой стрелки. Когда Вы вращаете что-то вокруг оси Z в OpenGL, это что-то будет вращаться также.

В следующей строке кода, если rtri равно 7, мы будем вращать на 7 градусов по оси Y (слева направо). Вы можете поэксперементировать с кодом. Изменяйте от 0.0f до 1.0f, и от 1.0f до 0.0f вращение треугольника по осям X и Y одновременно.

```
    glRotatef(rtri,0.0f,1.0f,0.0f);         // Вращение треугольника по оси Y
```

Следующая секция кода не изменена. Здесь будет нарисован закрашенный сглаженный треугольник. Треугольник будет нарисован с левой стороны экрана, и будет вращаться по оси Y слева направо.

```
    glBegin(GL_TRIANGLES);                  // Начало рисования треугольника
        glColor3f(1.0f,0.0f,0.0f);      // Верхняя точка - красная
        glVertex3f( 0.0f, 1.0f, 0.0f);  // Первая точка
        glColor3f(0.0f,1.0f,0.0f);      // Левая точка - зеленная
        glVertex3f(-1.0f,-1.0f, 0.0f);  // Вторая
```

```
        glColor3f(0.0f,0.0f,1.0f);      // Правая - синия
        glVertex3f( 1.0f,-1.0f, 0.0f);  // Третья
    glEnd();                            // Конец рисования
```

Посмотрите на код ниже, там мы добавим вызов glLoadIdentity(). Мы сделаем это для инициализации просмотра. Что будет если мы не сбросим просмотр? Если мы сдвинули объект после вращения, Вы получите очень неожиданные результаты. Поскольку оси вращаются, они будут указывать не в тех направлениях, о каких Вы думаете. Поэтому если мы сдвинулись влево по оси X (для установки треугольника), мы можем переместить квадрат в глубь экрана или вперед, в зависимости от того как много мы вращали по оси Y. Попробуйте убрать glLoadIdentity() и вы поймете о чем я говорю. Квадрат будет вращаться вокруг своей оси X, но и вокруг центра координат синхронно вращению треугольника.
Так как сцена сброшена, поэтому X идет слева направо, Y сверху вниз, Z от нас и далее. Теперь мы перемещаем. Как Вы видите мы сдвигаем на 1.5 вправо, вместо 3.0, как мы делали в последнем уроке. Когда мы сбрасываем экран, наш фокус перемещается в центр экрана, это означает, что мы находимся не 1.5 единицы слева, мы вернулись в 0.0. Поэтому мы не должны сдвигаться на 3.0 единицы вправо (если бы не было сброса), мы должны только сдвинуться от центра вправо на 1.5 единицы.

После того как мы сдвинулись в новое место на правой стороне экрана, мы вращаем квадрат по оси X. Квадрат будет вращаться верх и вниз.

```
    glLoadIdentity();
    glTranslatef(1.5f,0.0f,-6.0f);          // Сдвиг вправо на 1.5
    glRotatef(rquad,1.0f,0.0f,0.0f);        // Вращение по оси X
```

Эта секция кода завершение предыдущей. Рисуем синий квадрат из одного четырехугольника. Квадрат будет справа на экране и там же будет вращаться.

```
    glColor3f(0.5f,0.5f,1.0f);              // Синий цвет
    glBegin(GL_QUADS);                      // Начнем
        glVertex3f(-1.0f, 1.0f, 0.0f);  // Верх лево
        glVertex3f( 1.0f, 1.0f, 0.0f);  // Верх право
        glVertex3f( 1.0f,-1.0f, 0.0f);  // Низ право
        glVertex3f(-1.0f,-1.0f, 0.0f);  // Низ лево
    glEnd();                            // Окончим
```

Следующие две строки новые. Думайте о rtri и rquad как о контейнерах. Вначале нашей программы мы сделали контейнеры (GLfloat rtri и GLfloat rquad). Когда мы построили контейнеры они были пусты. В первой строке ниже ДОБАВЛЯЕМ 0.2 в контейнер. Если мы проверим значение контейнера rtri после этой секции кода, мы увидим что оно увеличилось на 0.2. Контейнер rquad уменьшиться на 0.15. Если мы проверим значение контейнера rquad после этой секции кода, мы увидим что оно уменьшилось на 0.15. Отрицательные значения вращения приводят к тому, что объект вращается в противоположную сторону. Как если бы значения были положительные.

Попробуйте изменить + на - в строке ниже и объект будет вращаться в другом направлении. Попробуйте изменить значение с 0.2 до 1.0. С увеличением значения объект будет вращаться быстрее. С уменьшением значения будет вращаться медленее.

```
    rtri+=0.2f;             // Увеличение переменной вращения для треугольника
    rquad-=0.15f;           // Уменьшение переменной вращения для квадрата
}
```

В этом уроке я попробывал рассказать как можно детальнее о том как вращаются объекты вокруг осей. Поиграйте с этим кодом, попробуйте вращать объекты по оси Z, X & Y, или по всем трем ;). Если у Вас есть комментарии или вопросы пожалуйста вышлите мне письмо. Если Вы нашли ошибки или улучшения дайте мне об этом знать. Я хочу сделать уроки по OpenGL хорошими насколько смогу. Я заинтересован в обратной связи.

Урок 5. Создание фигур в 3D

Solid Objects

Продолжая последний урок, мы сделаем объект, как ИСТИННЫЙ трехмерный объект, а не 2D объекты в 3D мире. Мы будем делать это добавлением с левой, задней и правой сторон треугольника, и с левой, правой, верхней и нижней сторон квадрата. Сделав это, мы превращаем треугольник в пирамиду с четырьмя гранями и квадрат в куб.

Мы будем смешивать цвета на пирамиде, создавая сглаженный закрашенный объект, а для квадрата мы назначим каждой грани свой цвет.

```
GLvoid DrawGLScene(GLvoid)
{
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);// Очистка экрана и буфера глубины
glLoadIdentity();                       // Сброс просмотра
glTranslatef(-1.5f,0.0f,-6.0f);             // Сдвиг влево и вглубь экрана
glRotatef(rtri,0.0f,1.0f,0.0f);             // Вращение пирамиды по оси Y
glBegin(GL_TRIANGLES);                      // Начало рисования пирамиды
```

Некоторые из Вас взяли код из последнего урока и сделали свои собственные 3D объекты. Вот один вопрос, который вы задали : "как сделать, чтобы мои объекты не вращались по своим осям? Потому что кажется, что они вращаются на весь экран". Чтобы объект вращался вокруг оси, он должен быть разработан для вращения ВОКРУГ оси. Вы должны помнить, что центр любого объекта должен быть в 0 для X, 0 для Y, 0 для Z.

Следующий код создаст пирамиду вокруг центральной оси. Верх пирамиды на единицу выше центра, низ пирамиды на единицу ниже центра. Верхняя точка как раз в середине (ноль), а нижние точки одна слева от центра, а одна справа от центра.

Отмечу, что все треугольники рисуются с вращением против часовой стрелки. Это важно, и будет объяснено в следующих уроках, сейчас, только запомните, что отличной привычкой будет делать объекты или по часовой или против часовой стрелки, и вы не должны смешивать эти два метода, если на то нет веской причины.

Мы начинаем рисовать с Передней Грани. Поскольку во все грани входит верхняя точка, мы будем делать эту точку красной во всех треугольниках. Цвет нижних двух точек треугольника будет другим. Передняя грань будет зеленной в левой точке и синей в правой точке. Треугольник с правой стороны будет синим в левой точке и зеленным в правой точке. При помощи чередования двух нижних цветов на каждой грани, мы сделаем общие закрашенные точки снизу на каждой грани.

```
    glColor3f(1.0f,0.0f,0.0f);              // Красный
    glVertex3f( 0.0f, 1.0f, 0.0f);          // Верх треугольника (Передняя)
    glColor3f(0.0f,1.0f,0.0f);              // Зеленный
    glVertex3f(-1.0f,-1.0f, 1.0f);          // Левая точка
    glColor3f(0.0f,0.0f,1.0f);              // Синий
    glVertex3f( 1.0f,-1.0f, 1.0f);          // Правая точка
```

Сейчас мы нарисуем правую грань. Отметим, что две нижних точки нарисованы на единицу справа от центра, верхняя точка нарисована на единицу выше оси Y, и справа от середины оси X. Поэтому эта грань имеет наклон от центральной точки сверху вниз с правой стороны.

Замечу, что левая точка нарисована синим цветом в этот раз. Так как она нарисована синей, это будет тот же самый цвет, какой у точки правого нижнего угла лицевой грани. Градиент синего цвета идет от одного угла вдоль лицевой и правой граней пирамиды.

Замечу, что все четыре грани включены внутрь тех же самых glBegin(GL_TRIANGLES) и glEnd(), словно одна сторона. Поскольку мы делаем целый объект из треугольников, OpenGL знает, что каждые три точки мы рисуем как три точки одного треугольника. Треугольник рисуется из трех точек, если больше трех точек, то OpenGL поймет, что надо рисовать другой треугольник. Если вы выведете четыре точки вместо трех, OpenGL будет рисовать первые три точки и примет четвертую точку как начальную точку нового треугольника. Но не будет нарисован Четырехугольник. Поэтому проверьте, что вы не добавили любые дополнительные точки нечаяно.

```
    glColor3f(1.0f,0.0f,0.0f);              // Красная
    glVertex3f( 0.0f, 1.0f, 0.0f);          // Верх треугольника (Правая)
    glColor3f(0.0f,0.0f,1.0f);              // Синия
    glVertex3f( 1.0f,-1.0f, 1.0f);          // Лево треугольника (Правая)
    glColor3f(0.0f,1.0f,0.0f);              // Зеленная
    glVertex3f( 1.0f,-1.0f, -1.0f);         // Право треугольника (Правая)
```

Сейчас задняя сторона. Снова переключим цвета. Левая точка – зеленного цвета, поскольку этот угол так же и зеленный угол правой грани.

```
    glColor3f(1.0f,0.0f,0.0f);              // Красный
    glVertex3f( 0.0f, 1.0f, 0.0f);          // Низ треугольника (Сзади)
    glColor3f(0.0f,1.0f,0.0f);              // Зеленный
    glVertex3f( 1.0f,-1.0f, -1.0f);         // Лево треугольника (Сзади)
    glColor3f(0.0f,0.0f,1.0f);              // Синий
    glVertex3f(-1.0f,-1.0f, -1.0f);         // Право треугольника (Сзади)
```

В завершении мы рисуем левую грань. Цвета переключаются в последний раз. Левая точка синего цвета, и смешивается с правой точкой на задней грани. Правая точка зеленного цвета, и смешивается с левой точкой на передней грани.

Мы закончили рисовать пирамиду. Поскольку пирамида только крутиться вдоль оси Y, мы не когда не увидим низ, поэтому нет необходимости выводить низ пирамиды. Если вы чувствуете потребность в экспериментах, попробуйте добавить низ используя четырехугольник, тогда вращайте по оси X для того чтобы увидеть низ, если конечно вы сделали все корректно. Проверьте, что цвет каждого угла в четырехугольнике совпадает с цветом, который использован в каждом из четырех углов пирамиды.

```
    glColor3f(1.0f,0.0f,0.0f);              // Красный
    glVertex3f( 0.0f, 1.0f, 0.0f);          // Верх треугольника (Лево)
    glColor3f(0.0f,0.0f,1.0f);              // Синий
    glVertex3f(-1.0f,-1.0f,-1.0f);          // Лево треугольника (Лево)
    glColor3f(0.0f,1.0f,0.0f);              // Зеленный
    glVertex3f(-1.0f,-1.0f, 1.0f);          // Право треугольника (Лево)
glEnd();                                    // Кончили рисовать пирамиду
```

Сейчас мы будем рисовать куб. Чтобы сделать это надо шесть квадратов. Все квадраты рисуются против часовой стрелке. Примем, что первая точка справа вверху, вторая точка слева вверху, третья точка слева внизу, и последняя слева внизу. Когда мы рисуем заднюю грань, то будет казаться, что мы рисуем по часовой стрелке, но вы помните, что если мы были позади куба и смотрели на него, то левая сторона экрана, фактически была бы с правой стороны квадрата, и правая сторона экрана была бы фактически с левой стороны квадрата.

Замечу, что мы сдвинули куб немного вглубь экрана в этом уроке. Поэтому размер куба будет казаться меньше размера пирамиды. Если мы переместили бы куб на 6 единиц к экрану, то куб будет казаться больше чем пирамида, и часть куба будет за пределами экрана. Вы можете поиграться с этим настройками, и сдвинув куб дальше от экрана он будет казаться меньше, а придвинув к экрану он будет казаться больше. Это происходит из-за переспективы. Объекты на расстоянии кажутся меньше :).

```
glLoadIdentity();
glTranslatef(1.5f,0.0f,-7.0f);          // Сдвинуть вправо и вглубь экрана
glRotatef(rquad,1.0f,1.0f,1.0f);        // Вращение куба по X, Y & Z
glBegin(GL_QUADS);                      // Рисуем куб
```

Мы начнем рисовать куб сверху. Мы сдвигаемся на одну единицу от центра куба. Отметим, что по оси Y всегда единица. Затем мы рисуем квадрат на Z плоскости. Мы начинаем рисовать с правой точки вверху экрана. Правая верхняя точка должна быть на одну единицу справа, и на одну единицу вглубь экрана. Вторая точка будет на одну единицу влево и на единицу вглубь экрана. Сейчас мы нарисуем ту часть квадрата, которая ближе к зрителю. Поэтому для того чтобы сделать это, вместо смещения вглубь экрана, мы сдвигаемся на одну единицу к экрану. Улавливаете смысл?

```
    glColor3f(0.0f,1.0f,0.0f);              // Синий
    glVertex3f( 1.0f, 1.0f,-1.0f);          // Право верх квадрата (Верх)
    glVertex3f(-1.0f, 1.0f,-1.0f);          // Лево верх
    glVertex3f(-1.0f, 1.0f, 1.0f);          // Лево низ
    glVertex3f( 1.0f, 1.0f, 1.0f);          // Право низ
```

Нижняя часть квадрата рисуется таким же образом, как и верхняя, но поскольку это низ, сдвигаемся вниз на одну единицу от центра куба. Замечу, что ось Y всегда минус единица. Если мы окажемся под кубом, и взглянем на квадрат, который снизу, вы заметите, что правый верхний угол – это угол ближний к зрителю. Поэтому вместо того чтобы рисовать дальше от зрителя в начале, мы рисуем ближе к зрителю, тогда левая сторона ближе к зрителю. И затем мы движемся вглубь экрана, для того чтобы нарисовать дальние две точки.

Если Вы действительно не заботитесь о порядке рисования полигонов (по часовой или против), вы должны скопировать код для верхнего квадрата, сдвинуть вниз по оси Y на единицу, и это будет работать, но игнорируя

порядок рисования квадрата можно получить неверный результат, если вы захотите сделать, например, наложение текстуры.

```
glColor3f(1.0f,0.5f,0.0f);              // Оранжевый
glVertex3f( 1.0f,-1.0f, 1.0f);          // Верх право квадрата (Низ)
glVertex3f(-1.0f,-1.0f, 1.0f);          // Верх лево
glVertex3f(-1.0f,-1.0f,-1.0f);          // Низ лево
glVertex3f( 1.0f,-1.0f,-1.0f);          // Низ право
```

Сейчас мы рисуем передний квадрат. Мы сдвигаемся на одну единицу ближе к экрану, и дальше от центра для того чтобы нарисовать переднею грань. Заметим, что ось Z всегда равна единице. В гранях пирамиды ось Z не всегда единица. Вверху, ось Z равна нулю. Если Вы попробуете установить ось Z равной нулю в привиденом ниже коде, вы увидите, что угол, который вы изменили наклонен к экрану. Но это не то, что мы хотим сейчас сделать ;).

```
glColor3f(1.0f,0.0f,0.0f);              // Красный
glVertex3f( 1.0f, 1.0f, 1.0f);          // Верх право квадрата (Перед)
glVertex3f(-1.0f, 1.0f, 1.0f);          // Верх лево
glVertex3f(-1.0f,-1.0f, 1.0f);          // Низ лево
glVertex3f( 1.0f,-1.0f, 1.0f);          // Низ право
```

Задняя грань квадрата такая же ка передняя грань, но сдвинута вглубь экрана. Отметим, что ось Z всегда минус один во всех точках.

```
glColor3f(1.0f,1.0f,0.0f);              // Желтый
glVertex3f( 1.0f,-1.0f,-1.0f);          // Верх право квадрата (Зад)
glVertex3f(-1.0f,-1.0f,-1.0f);          // Верх лево
glVertex3f(-1.0f, 1.0f,-1.0f);          // Низ лево
glVertex3f( 1.0f, 1.0f,-1.0f);          // Низ право
```

Сейчас нам осталось нарисовать только два квадрата. Как вы уже успели заметить одна ось всегда имеет тоже самое значение у всех точек квадрата. В этом случае ось X всегда равна минус один. Поскольку мы рисуем левую грань.

```
glColor3f(0.0f,0.0f,1.0f);              // Синий
glVertex3f(-1.0f, 1.0f, 1.0f);          // Верх право квадрата (Лево)
glVertex3f(-1.0f, 1.0f,-1.0f);          // Верх лево
glVertex3f(-1.0f,-1.0f,-1.0f);          // Низ лево
glVertex3f(-1.0f,-1.0f, 1.0f);          // Низ право
```

И последняя грань завершит куб. Для нее ось X всегда равна единице. Рисуем против часовой стрелки. Если вы хотите, то вы можете не рисовать эту грань и получите коробку ;).

Если вы хотите поэксперементировать, вы всегда можете изменить цвет каждой точки в кубе для того чтобы сделать градиент, как в пирамиде. Вы можете посмотреть пример интерполяционной заливки куба скопировав первую демонстрацию Evil с моего сайта. Запустите ее и нажмите TAB. Вы увидите чудесный цветной куб, с изменяющимися цветами вдоль всех граней.

```
glColor3f(1.0f,0.0f,1.0f);              // Фиолетовый
glVertex3f( 1.0f, 1.0f,-1.0f);          // Верх право квадрата (Право)
glVertex3f( 1.0f, 1.0f, 1.0f);          // Верх лево
glVertex3f( 1.0f,-1.0f, 1.0f);          // Низ лево
glVertex3f( 1.0f,-1.0f,-1.0f);          // Низ право
glEnd();                            // Закончили квадраты

rtri+=0.2f;             // Увеличим переменную вращения для треугольника
rquad-=0.15f;           // Уменьшим переменную вращения для квадрата
}
```

В конце этого урока, вы должны лучше понимать как объекты создаются в 3D пространстве. Вы должны представлять экран OpenGL, как гиганская милиметровка, с множеством прозрачных слоев за ней. Это похоже на гигантский куб сделаный из точек. Некоторые точки двигаются слева направо, некоторые двигаются верх и вниз, и некоторые точки двигаются взад и вперед в кубе. Если вы может представить глубину экрана, вы не будете иметь проблем с разработкой новых 3D объектов.

Если вы с трудом понимаете 3D пространство, то это не бесполезно. Это может быть сложным только вначале. Объекты подобные кубу хороший пример для обучения. Если вы заметили задняя грань рисуется также как передняя грань, только дальше от экрана. Поиграйте с этим кодом, и если вы не можете понять это, спросите меня, и я вам отвечу.

Урок 6. Наложение текстуры

Texture Mapping

Из наложения текстуры можно извлечь много полезного. Предположим, что вы хотите, чтобы ракета пролетела через экран. До этого урока мы попытались бы сделать ракету из полигонов и фантастических цветов. С помощью наложения текстуры, мы можем получить реальную картинку ракеты и заставить ее летать по экрану. Как вы думаете, что будет выглядеть лучше? Фотография или объект сделанный их треугольников и четырехугольников? Используя наложение текстуры, и выглядеть будет лучше, и ваша программа будет работать быстрее. Ракета с наложением текстуры - это всего лишь четырехугольник, движущийся по экрану. Ракета сделанная из полигонов может иметь сотни или тысячи полигонов. Отрисовка простого четырехугольника будет отнимать меньше процессорного времени.

Давайте начнем с добавления четырех новых строк в начало кода первого урока. Первые три строки задают четыре вещественных переменных - xrot, yrot и zrot. Эти переменные будут использованы для вращения куба по осям x, y, z. В четвертой строке резервируется место для одной текстуры. Если вы хотите загрузить более чем одну текстуру, измените, число один на число текстур, которые вы хотите загрузить.

```
#include <windows.h>// Заголовочный файл для Windows
#include <gl\gl.h>// Заголовочный файл для OpenGL32 библиотеки
#include <gl\glu.h>// Заголовочный файл для GLu32 библиотеки
#include <gl\glaux.h>// Заголовочный файл для GLaux библиотеки

static HGLRC hRC;// Постоянный контекст рендеринга
static HDC hDC;// Приватный контекст устройства GDI

BOOLkeys[256];// Массив для процедуры обработки клавиатуры

GLfloatxrot;// Вращение X
GLfloatyrot;// Y
GLfloatzrot;// Z

GLuinttexture[1];// Место для одной текстуры
```

Теперь сразу же после этого кода, до InitGL, мы добавим следующую секцию кода. Этот код загружает файл картинки, и конвертирует его в текстуру. Прежде чем я начну объяснять этот код, я сделаю нескольких ВАЖНЫХ замечаний, которые вы должны знать об изображениях, которые вы используете как текстуры. Такое изображение ДОЛЖНО иметь высоту и ширину кратной двум. При этом высота и ширина изображения должна быть не меньше чем 64 пикселя, и по причинам совместимости, не более 256 пикселов. Если изображение, которое вы используете не 64, 128 или 256 пикселов в ширину и высоту, измените его размер в программе для рисования. Имеются возможность обойти эти ограничения, но мы пока будем придерживаться стандартных размеров текстуры.

AUX_RGBImageRec *texture1 задает указатель на структуру для хранения первой картинки, которую мы загрузим и используем как текстуру. Структура содержит красную, зеленную и синею компоненты цвета, которые используются при создании изображения. Обычно так размещается в памяти загруженная картинка. Структура AUX_RGBImageRec определена в библиотеке glAux, и делает возможной загрузку картинки в память. В следующей строке происходит непосредственная загрузка. Файл картинки "NeHe.bmp" из каталога "Data" будет загружен и сохранен в структуре texture1, которую мы задали выше с помощью AUX_RGBImageRec.

```
// Загрузка картинки и конвертирование в текстуру
GLvoid LoadGLTextures()
{
// Загрузка картинки
AUX_RGBImageRec *texture1;
texture1 = auxDIBImageLoad("Data/NeHe.bmp");
```

Сейчас мы загрузили изображение как данные компонент цветов красного, зеленного и синего, далее мы построим текстуру используя эти данные. Вызовом glGenTextures(1, &texture[0]) мы скажем OpenGL, что мы хотим построить текстуру в нулевом элементе массива texture[]. Помните, в начале урока мы зарезервировали место для одной текстуры с помощью GLuint texture[1]. Хотя вы, возможно, подумали, что мы сохраним текстуру в &texture[1], но это не так. Первая действительная область для сохранения имеет номер 0. Если вы хотите две текстуры, надо задать GLuint texture[2] и вторая текстура будет сохранена в texture[1].

Во второй строке вызов glBindTexture(GL_TEXTURE_2D, texture[0]) говорит OpenGL, что texture[0] (первая текстура) будет 2D текстурой. 2D текстуры имееют и высоту (по оси Y) и ширину (по оси X). Основная задача glGenTexture указать OpenGL на доступную память. В этом случае мы говорим OpenGL, что память доступна в &texture[0]. Затем мы создаем текстуру, и она будет сохранена в этой памяти. Далее, если мы привязываемся к памяти, в которой уже находиться текстура, мы говорим OpenGL захватить данные текстуры из этой области памяти. Обычно это указатель на доступную память, или память, в которой содержиться текстура.

```
// Создание текстуры
glGenTextures(1, &texture[0]);
glBindTexture(GL_TEXTURE_2D, texture[0]);
```

В следующих двух строках мы сообщим OpenGL какой тип фильтрации надо использовать, когда изображение больше на экране, чем оригинальная текстура (GL_TEXTURE_MAG_FILTER), или когда оно меньше на экране, чем текстура (GL_TEXTURE_MIN_FILTER). я обычно использую GL_LINEAR для обоих случаев. При этом текстура выглядит сглаженной на расстоянии, и вблизи. Использование GL_LINEAR требует много работы для процессора/видеокарты, поэтому если ваша система медленная, вы можете захотеть использовать GL_NEAREST. Текстура, которая фильтруется с GL_NEAREST состоит из хорошо видимых цветовых прямоугольников, когда она вблизи. Вы можете попробовать комбинировать оба способа. Сделайте одну фильтрацию вблизи, а другую вдали.

```
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
```

В завершении мы создаем фактическую текстуру. В следующей строке мы говорим OpenGL, что текстура будет двухмерной (GL_TEXTURE_2D). Ноль задает уровень детализации, это обычно ноль. Три - число компонент цветовых данных, так как изображение сделано из трех цветовых компонент (красный, зеленный, синий). texture1->sizeX - это ширина текстуры, автоматически. Если вы знаете ширину, вы можете указать ее тут, но проще дать компьютеру сделать это за вас. texture1->sizeY - высота текстуры. Ноль - это бордюр. Он обычно остается нулем. GL_RGB сообщает OpenGL, что данные изображения представлены в порядке следования красных, зеленных и голубых компонент цвета. GL_UNSIGNED_BYTE означает, что данные из которых состоит изображение имеют размер байта и все числа без знака, и в конце texture1->data сообщает OpenGL, где брать сами данные. В этом случае указатель на данные в записи texture1.

```
glTexImage2D(GL_TEXTURE_2D, 0, 3, texture1->sizeX, texture1->sizeY, 0,
GL_RGB, GL_UNSIGNED_BYTE, texture1->data);
}
```

Мы добавим две строчки в код InitGL. я повторно привожу эту секцию кода, для того чтобы было легко увидеть строчки, которые я добавил, и где они идут в коде.

В первой строке происходит вызов процедуры LoadGLTextures(), которая загружает изображение и делает из него текстуру. Вторая строка glEnable(GL_TEXTURE_2D) разрешает наложение текстуры. Если вы не делаете доступной наложение текстуры, ваш объект будет закрашен сплошным белым цветом, который точно не очень хорош.

```
GLvoid InitGL(GLsizei Width, GLsizei Height)
{
LoadGLTextures();// Загрузка текстур
glEnable(GL_TEXTURE_2D);// Разрешение наложение текстуры
glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
glClearDepth(1.0);
glDepthFunc(GL_LESS);
glEnable(GL_DEPTH_TEST);
```

```
glShadeModel(GL_SMOOTH);

glMatrixMode(GL_PROJECTION);
glLoadIdentity();

gluPerspective(45.0f,(GLfloat)Width/(GLfloat)Height,0.1f,100.0f);

glMatrixMode(GL_MODELVIEW);
}
```

Сейчас мы нарисуем куб с текстурой. Мы можете заменить код DrawGLScene на код ниже, или вы можете добавить новый код в оригинальный код первого урока. Эта секция будет сильно прокомментирована, поэтому легка для понимания. Первые две строки кода glClear() и glLoadIdentity() взяты из оригинального кода первого урока. glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT) очищает экран цветом, который мы выбрали в InitGL(). В этом случае, экран будет очищен в синий цвет. Буфер глубины будет также очищен. Просмотр будет сброшен с помощью glLoadIdentity().

```
GLvoid DrawGLScene(GLvoid)
{
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glLoadIdentity();
glTranslatef(0.0f,0.0f,-5.0f);
```

Следующие три строки кода будут вращать куб по оси X, затем по оси Y, и в конце по оси Z. Насколько велико будет вращение (угол) по каждой оси будет зависеть от значения указанного в xrot, yrot и zrot.

```
glRotatef(xrot,1.0f,0.0f,0.0f);// Вращение по оси X
glRotatef(yrot,0.0f,1.0f,0.0f);// Вращение по оси Y
glRotatef(zrot,0.0f,0.0f,1.0f);// Вращение по оси Z
```

В следующей строке кода происходит выбор какую текстуру мы хотим использовать для наложения текстуры. Если вы хотите использовать более чем одну текстуру в вашей сцене, вы должны выбрать текстуру glBindTexture(GL_TEXTURE_2D, texture[номер текстуры для использования]). Вы должны затем нарисовать несколько четырехугольников используя эту текстуру. Каждый раз, когда Вы захотите сменить текстуру, Вы должны привязать новую текстуру. Одно замечание: вы НЕ должны связывать текстуру внутри glBegin() и glEnd().

```
glBindTexture(GL_TEXTURE_2D, texture[0]);
```

Для того чтобы правильно отобразить текстуру на четырехугольник, вы должны отобразить правый верхний угол текстуры на правую верхнею вершину четырехугольника. Левый верхний угол текстуры отображается в левую верхнею вершину четырехугольника, правый нижний угол текстуры отображается в правую нижнею вершину четырехугольника, и в завершении, левый нижний угол текстуры отображается в левую нижнею вершину четырехугольника. Если углы текстуры не совпадают с углами четырехугольника, изображение может быть сдвинуто вниз, в сторону, или вообще отсутствовать.
Первый аргумент glTexCoord2f - координата X. 0.0f - левая сторона текстуры. 0.5f - середина текстуры, и 1.0f - правая сторона текстуры. Втрое значение glTexCoord2f - это Y координата. 0.0f - низ текстуры. 0.5f - середина текстуры, и 1.0f - верх текстуры.

Теперь мы знаем, что левая верхняя координата текстуры 0.0f по X и 1.0f по Y, и левая верхняя вершина четырехугольника -1.0f по X, и 1.0f по Y. Теперь осталось сделать так, чтобы оставшиеся три координаты совпали с тремя углами четырехугольника.
Попробуйте поиграться со значениями x и y в glTexCoord2f. Изменение 1.0f на 0.5f будет только рисовать левую половину текстуры от 0.5f (середина) до 1.0f (право).

```
glBegin(GL_QUADS);

// Передняя грань
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f,  1.0f);// Низ лево
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f,  1.0f);// Низ право
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f,  1.0f,  1.0f);// Верх право
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f,  1.0f,  1.0f);// Верх лево
```

```
// Задняя грань
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);// Низ право
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f,  1.0f, -1.0f);// Верх право
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f,  1.0f, -1.0f);// Верх лево
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);// Низ лево

// Верхняя грань
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f,  1.0f, -1.0f);// Верх лево
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f,  1.0f,  1.0f);// Низ лево
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f,  1.0f,  1.0f);// Низ право
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f,  1.0f, -1.0f);// Верх право

// Нижняя грань
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, -1.0f, -1.0f);// Верх право
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, -1.0f, -1.0f);// Верх лево
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f,  1.0f);// Низ лево
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f,  1.0f);// Низ право

// Правая грань
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);// Низ право
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f,  1.0f, -1.0f);// Верх право
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f,  1.0f,  1.0f);// Верх лево
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f,  1.0f);// Низ лево

// Левая грань
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);// Низ лево
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f,  1.0f);// Низ право
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f,  1.0f,  1.0f);// Верх право
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f,  1.0f, -1.0f);// Верх лево

glEnd();
```

Сейчас мы увеличим значения xrot, yrot и zrot. Попробуйте изменить значения каждой переменной, замедляя или ускоряя вращение куба, или изменяя ' +' на '-' заставляя куб вращаться в другом направлении.

```
xrot+=0.3f;// Ось вращения X
yrot+=0.2f;// Ось вращения Y
zrot+=0.4f;// Ось вращения Z
}
```

Теперь Вы должны лучше понимать наложение текстуры. Вы научились накладывать текстуру на поверхность любого четырехугольника с изображением по вашему выбору. Как только вы лучше поймете наложение текстуры, попробуйте наложить на куб шесть разных текстур.

Наложение текстуры не трудно для понимания, если Вы понимаете координаты текстуры. Если Вы имеете проблемы с пониманием любой части этого урока, дайте мне знать. Или я изменю этот урок, и я отвечу Вам по почте. Развлекайтесь созданием наложения текстуры в Ваших сценах.

Урок 7. Режимы фильтрации текстур, освещение и обработка клавиатуры

Texture Filters, Lighting & Keyboard Control

В этом уроке я научу вас, как использовать три разных режима фильтрации текстур. Я научу вас, как перемещать объект, используя клавиши на клавиатуре, и я также преподам вам, как применить простое освещение в вашей OpenGL сцене. В этом уроке много материала, поэтому, если предыдущие уроки вам непонятны, вернитесь и посмотрите их вновь. Важно иметь хорошее понимание основ прежде, чем Вы перепрыгнете на этот код.

Мы собираемся снова изменить код первого урока. Как обычно, если много меняется, я привожу полную секцию кода, который был изменен. Мы начнем, добавив несколько новых переменных к программе.

```
#include <windows.h>  // Заголовочный файл для Windows
#include <stdio.h>    // Заголовочный файл для стандартного ввода/вывода (ДОБАВИЛИ)
#include <gl\gl.h>    // Заголовочный файл для библиотеки OpenGL32
#include <gl\glu.h>   // Заголовочный файл для для библиотеки GLu32
#include <gl\glaux.h> // Заголовочный файл для библиотеки GLaux

HDC    hDC=NULL;      // Служебный контекст GDI устройства
HGLRC  hRC=NULL;      // Постоянный контекст для визуализации
HWND   hWnd=NULL;     // Содержит дискриптор для окна
HINSTANCE hInstance;  // Содержит данные для нашей программы

bool keys[256];       // Массив, использующийся для сохранения состояния клавиатуры
bool active=TRUE;     // Флаг состояния активности приложения (по умолчанию: TRUE)
bool fullscreen=TRUE; // Флаг полноэкранного режима (по умолчанию: полноэкранное)
```

Строки ниже новые. Мы собираемся добавлять три логических переменных. Тип BOOL означает, что переменная может только быть ИСТИННА (TRUE) или ЛОЖЬ (FALSE). Мы создаем переменную называемую light, чтобы отслеживать, действительно ли освещение включено или выключено. Переменные lp и fp используются, для отслеживания нажатия клавиш 'L' и 'F'. Я объясню, почему нам нужны эти переменные позже. Пока, запомните, что они необходимы.

```
BOOL light;      // Свет ВКЛ / ВЫКЛ
BOOL lp;         // L нажата?
BOOL fp;         // F нажата?
```

Теперь нам нужны пять переменных, которые будут управлять следующими параметрами: углом по оси X (xrot), углом по оси Y (yrot), скоростью вращения ящика по оси X (xspeed), и скоростью вращения ящика по оси Y (yspeed). Мы также создадим переменную z, которая будет управлять, погружением ящика в экран (по оси Z).

```
GLfloat xrot;         // X вращение
GLfloat yrot;         // Y вращение
GLfloat xspeed;       // X скорость вращения
GLfloat yspeed;       // Y скорость вращения
GLfloat z=-5.0f;      // Сдвиг вглубь экрана
```

Теперь мы зададим массивы, которые мы будем использовать для освещения. Мы будем использовать два разных типа света. Первый тип света называется фоновым светом. Фоновый свет не имеет никакого определенного направления. Все объекты в вашей сцене будут освещены фоновым светом. Второй тип света - диффузный свет. Диффузный свет создается при помощи вашего источника света и отражается от поверхности объекта в вашей сцене. Любая поверхность объекта, на которую падает прямо свет, будет очень яркой и области, куда свет падает под углом, будут темнее. Это создает хороший эффект оттенения на сторонах нашей корзины.

Свет создается так же как цвет. Если первое число - 1.0f, а следующие два - 0.0f, мы получим ярко красный свет. Если третье число - 1.0f, и первые два - 0.0f, мы будем иметь ярко синий свет. Последнее число - альфа значение. Мы оставим его пока 1.0f.

Поэтому в строке ниже, мы зададим значение белого фонового света половиной интенсивности (0.5f). Поскольку все числа - 0.5f, мы получим свет средней яркости между черным (выключен свет) и белым (полная яркость). Смешанные в равных значениях красный, синий и зеленый дадут оттенки от черного (0.0f) до белого (1.0f). Без фонового света пятна, где нет диффузного света, будут очень темными.

```
GLfloat LightAmbient[]= { 0.5f, 0.5f, 0.5f, 1.0f }; // Значения фонового света ( НОВОЕ )
```

В следующей строке мы зададим значение для супер-яркой, полной интенсивности диффузного света. Все значения - 1.0f. Это означает, что свет настолько яркий, насколько мы можем получить его. Диффузный свет эти яркое пятно спереди ящика.

```
GLfloat LightDiffuse[]= { 1.0f, 1.0f, 1.0f, 1.0f }; // Значения диффузного света ( НОВОЕ )
```

Наконец мы зададим позицию света. Первые три числа совпадают с тремя первыми аргументами функции glTranslate. Первое число смещение влево или вправо по оси x, второе число - для перемещения вверх или вниз по оси y, и третье число для перемещения к или от экрана по оси z. Поскольку мы хотим чтобы наш свет, падал прямо на переднею часть ящика, мы не сдвигаемся влево или вправо, поэтому первое значение - 0.0f (нет движения по x), мы не хотим двигаться вверх или вниз, поэтому второе значение - 0.0f. Третье значение мы зададим так, чтобы свет был всегда перед ящиком. Поэтому мы поместим свет вне экрана по отношению к наблюдателю. Давайте примем, что стекло на вашем мониторе - 0.0f по оси z. Мы позиционируем свет в 2.0f по оси z. Если бы Вы могли бы фактически видеть свет, он бы плавал перед стеклом вашего монитора. Делая, таким образом, единственный способ, когда бы свет оказался позади ящика, был бы тот, если бы ящик был также перед стеклом вашего монитора. Конечно, если бы ящик был уже не позади стекла вашего монитора, Вы больше не видели ящик, поэтому тогда не имеет значения, где свет. Это имеет смысл?
Нет никакого простого реального способа разъяснить третий параметр. Вы должны понять, что -2.0f ближе к Вам чем -5.0f и что -100.0f ДАЛЕКО в экране. Как только Вы берете 0.0f, изображение становится настолько большим, это заполняет весь монитор. Как только Вы устанавливаете положительные значения, изображение больше не появляется на экране по той причине, что объект "пропал с экрана". Вот это я подразумеваю, когда я говорю вне экрана. Объект - все еще есть, но Вы уже не можете больше его видеть.
Оставьте последнее число в 1.0f. Это говорит OpenGL о том, что данные координаты - позиция источника света. Более подробно об этом я расскажу в одном из следующих уроков.

```
GLfloat LightPosition[]= { 0.0f, 0.0f, 2.0f, 1.0f };     // Позиция света ( НОВОЕ )
```

Переменная filter должна отслеживать, каким образом будут отображаться текстуры. Первая текстура (texture[0]) использует gl_nearest (без сглаживания). Вторая текстура (texture[1]) использует фильтрацию gl_linear, которая немного сглаживает изображение. Третья текстура (texture[2]) использует текстуры с мип-наложением (mipmapped, или множественное наложение), что повышает качество отображения. Переменная filter будет равняться 0, 1 или 2 в зависимости от текстуры, которую мы хотим использовать. Мы начинаем с первой текстуры.

Объявление GLuint texture[3] создает место для хранения трех разных текстур. Текстуры будут сохранены в texture[0], texture[1] и texture[2].

```
GLuint filter;          // Какой фильтр использовать
GLuint texture[3];     // Место для хранения 3 текстур
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);    // Декларация WndProc
```

Сейчас мы загрузим картинку (bitmap, или растровый (побитный) образ изображения), и создадим три различных текстуры из нее. В этом уроке используется библиотека glaux для загрузки картинки, поэтому проверьте, что Вы подключили библиотеку glaux прежде, чем Вы пробуете скомпилировать код. Я знаю, что Delphi, и Visual C++ имеют библиотеку glaux. Я не уверен, что она есть в других языках. Я собираюсь объяснить только новые строки кода, если Вы видите не прокомментированную строку, и Вы не понимаете, что она делает, посмотрите шестой урок. Там объясняется загрузка, и формирования образов текстуры из картинок очень подробно.
Сразу же после кода выше, и до ReSizeGLScene (), мы добавим следующую секцию кода. Это - тот же самый код, который мы использовали в уроке 6 для загрузки картинки. Ничего не изменилось. Если Вы не понимаете их, прочитайте шестой урок. Там этот код объяснен подробнее.

```
AUX_RGBImageRec *LoadBMP(char *Filename)     // Загрузка картинки
{
 FILE *File=NULL;          // Индекс файла
 if (!Filename)            // Проверка имени файла
 {
  return NULL;             // Если нет вернем NULL
 }
 File=fopen(Filename,"r"); // Проверим существует ли файл

 if (File)                 // Файл существует?
 {
  fclose(File);            // Закрыть файл
  return auxDIBImageLoad(Filename); // Загрузка картинки и вернем на нее указатель
 }
 return NULL;              // Если загрузка не удалась вернем NULL
}
```

В этой секции кода загружается картинка (вызов кода выше) и производится конвертирование ее в 3 текстуры. Переменная Status используется, чтобы следить, действительно ли текстура была загружена и создана.

```
int LoadGLTextures()                    // Загрузка картинки и конвертирование в текстуру
{
 int Status=FALSE;                      // Индикатор состояния
 AUX_RGBImageRec *TextureImage[1];      // Создать место для текстуры
 memset(TextureImage,0,sizeof(void *)*1); // Установить указатель в NULL
```

Теперь мы загружаем картинку и конвертируем ее в текстуру. Выражение TextureImage[0]=LoadBMP ("Data/Crate.bmp ") будет вызывать наш код LoadBMP(). Файл по имени Crate.bmp в каталоге Data будет загружен. Если все пройдет хорошо, данные изображения сохранены в TextureImage[0], Переменная Status установлена в TRUE, и мы начинаем строить нашу текстуру.

```
 // Загрузка картинки, проверка на ошибки, если картинка не найдена - выход
 if (TextureImage[0]=LoadBMP("Data/Crate.bmp"))
 {
  Status=TRUE;      // Установим Status в TRUE
```

Теперь, мы загрузили данные изображения в TextureImage [0], мы будем использовать эти данные для построения 3 текстур. Строка ниже сообщает OpenGL, что мы хотим построить три текстуры, и мы хотим, чтобы текстура была сохранена в texture[0], texture[1] и texture[2].

```
glGenTextures(3, &texture[0]);     // Создание трех текстур
```

В шестом уроке, мы использовали линейную фильтрацию образов текстур. Это способ фильтрации требует много мощности процессора, но текстуры при этом выглядят реалистичными. Первый тип текстуры, которую мы собираемся создать в этом уроке, использует GL_NEAREST. Этот тип текстуры не использует фильтрацию. Требуется очень мало мощности процессора, и качество плохое. Если вы когда-нибудь видели игру, где текстуры как будто состоят из блоков, они, вероятно, используют этот тип текстуры. Единственное применение этого типа текстуры для проектов, которые будут запускаться на медленных компьютерах.
Заметьте, что мы используем GL_NEAREST, и для MIN и для MAG. Вы можете смешивать использование GL_NEAREST с GL_LINEAR, и текстура будет смотреться немного лучше, но мы заинтересованы в быстродействии, поэтому мы будем использовать везде низкое качество. Фильтр MIN_FILTER используется, когда изображение рисуемого полигона меньше, чем первоначальный размер текстуры. Фильтр MAG_FILTER используется, когда изображение рисуемого полигона больше, чем первоначальный размер текстуры.

```
  // Создание текстуры с фильтром по соседним пикселям
  glBindTexture(GL_TEXTURE_2D, texture[0]);
  glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST); // ( НОВОЕ )
  glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST); // ( НОВОЕ )
  glTexImage2D(GL_TEXTURE_2D, 0, 3, TextureImage[0]->sizeX, TextureImage[0]->sizeY,
   0, GL_RGB, GL_UNSIGNED_BYTE, TextureImage[0]->data);
```

Следующая текстура, которую мы построим, имеет тот же самый тип текстуры, которую мы использовали в уроке шесть. Линейный режим фильтрации. Изменилось только то, что мы сохраняем эту текстуру в texture[1] вместо texture[0], потому что здесь она вторая текстура. Если бы мы сохранили ее в texture[0] как ранее, она затерла бы текстуру GL_NEAREST (первая текстура).

```
  // Создание текстуры с линейной фильтрацией
  glBindTexture(GL_TEXTURE_2D, texture[1]);
  glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
  glTexImage2D(GL_TEXTURE_2D, 0, 3, TextureImage[0]->sizeX, TextureImage[0]->sizeY,
   0, GL_RGB, GL_UNSIGNED_BYTE, TextureImage[0]->data);
```

Теперь новый способ создания текстуры. Мип-наложение! Вы, возможно, замечали, что, когда изображение на экране очень маленькое, пропадает множество мелких деталей. Орнаменты, которые выглядят вначале отлично, смотрятся отвратительно. Когда Вы говорите OpenGL построить текстуру с мип-наложением, OpenGL будет строить разного размера высококачественные текстуры. Когда Вы выводите текстуру с мип-наложением на экран, OpenGL будет выбирать НАИБОЛЕЕ ЛУЧШУЮ для отображения текстуру из построенных текстур (текстура с таким размером, как размер полигона на экране, т.е. наиболее детальная) и нарисует ее на экран вместо масштабирования первоначального изображения (что и вызывает потерю качества).
Я рассказал в уроке шесть про ограничение на размер текстур в OpenGL по ширине и высоте текстуры в 64,128,256, и т.д. Для gluBuild2DMipmaps Вы можете использовать картинки любых размеров при формировании текстуры с мип-наложением. OpenGL будет автоматически изменять размер ее задавая правильные ширину и высоту.

Поскольку это - текстура номер три, мы собираемся хранить эту текстуру в texture[2]. Поэтому, теперь мы имеем texture[0], которая не имеет никакой фильтрации, texture[1], которая использует линейную фильтрацию, и texture[2], которая использует текстуру с мин-наложением. Мы закончили построение текстур в этом уроке.

```
 // Создание Текстуры с Мип-Наложением
 glBindTexture(GL_TEXTURE_2D, texture[2]);
 glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
 glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR_MIPMAP_NEAREST); // ( НОВОЕ )
```

Следующая строка строит текстуру с мип-наложением. Мы создаем 2D текстуру, используя три цвета (красный, зеленый, синий) (red, green, blue). TextureImage[0]->sizeX - ширина картинки, TextureImage[0]->sizeY - высота картинки, GL_RGB означает, что мы используем цвета в порядке Красный, Зеленый, Синий (Red, Green, Blue). GL_UNSIGNED_BYTE означает что данные, из которых состоит текстура из байтов, и TextureImage[0]->data указатель на растр картинки, из которого мы строим текстуру.

```
 gluBuild2DMipmaps(GL_TEXTURE_2D, 3, TextureImage[0]->sizeX, TextureImage[0]->sizeY,
  GL_RGB, GL_UNSIGNED_BYTE, TextureImage[0]->data); // ( НОВОЕ )
}
```

Теперь мы можем освободить память, которую мы использовали для картинок. Мы проверяем, была ли картинка сохранена в TextureImage[0]. Если да, то мы проверяем, были ли данные сохранены. Если данные были сохранены, мы удаляем их. Тогда мы освобождаем структуру изображения, уверенные, что любая используемая память освобождена.

```
if (TextureImage[0])          // Если текстура существует
{
 if (TextureImage[0]->data)    // Если изображение текстуры существует
 {
  free(TextureImage[0]->data); // Освобождение памяти изображения текстуры
 }
free(TextureImage[0]);        // Освобождение памяти под структуру
}
```

Наконец мы возвращаем статус. Если все прошло OK, переменная Status будет TRUE. Если что-нибудь прошло не так, как надо, Status будет FALSE.

```
 return Status;        // Возвращаем статус
}
```

Теперь мы загружаем текстуры, и инициализируем параметры настройки OpenGL. В первой строке InitGL загружаются текстуры, используя код выше. После того, как текстуры созданы, мы разрешаем 2D наложение текстуры с помощью glEnable (GL_TEXTURE_2D). Режим закрашивания (shade) задается как сглаженное закрашивание. Цвет фона задан как черный, мы разрешаем тест глубины, затем мы разрешаем хорошие перспективные вычисления.

```
int InitGL(GLvoid)            // Все настройки для OpenGL делаются здесь
{
 if (!LoadGLTextures())       // Переход на процедуру загрузки текстуры
 {
  return FALSE;               // Если текстура не загружена возвращаем FALSE
 }

 glEnable(GL_TEXTURE_2D);     // Разрешить наложение текстуры
 glShadeModel(GL_SMOOTH);     // Разрешение сглаженного закрашивания
 glClearColor(0.0f, 0.0f, 0.0f, 0.5f); // Черный фон
 glClearDepth(1.0f);          // Установка буфера глубины
 glEnable(GL_DEPTH_TEST);     // Разрешить тест глубины
 glDepthFunc(GL_LEQUAL);      // Тип теста глубины
 glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // Улучшенные вычисления перспективы
```

Теперь мы задаем освещение. Строка ниже задает интенсивность фонового света, которое light1 будет давать. В начале этого урока мы задали интенсивность фонового света в LightAmbient. Значения, которые мы поместили в массив, будут использованы (фоновый свет половиной интенсивности).

```
glLightfv(GL_LIGHT1, GL_AMBIENT, LightAmbient);    // Установка Фонового Света
```

Затем мы задаем интенсивность диффузного света, который источник света номер один будет давать. Мы задали интенсивность диффузного света в LightDiffuse. Значения, которые мы поместили в этот массив, будут использованы (белый свет полной интенсивности).

```
glLightfv(GL_LIGHT1, GL_DIFFUSE, LightDiffuse);    // Установка Диффузного Света
```

Теперь мы задаем позицию источника света. Мы поместили позицию в LightPosition. Значения, которые мы поместили в этот массив, будут использованы (справо в центре передней грани, 0.0f по x, 0.0f по y, и 2 единицы вперед к наблюдателю {выходит за экран} по оси z).

```
 glLightfv(GL_LIGHT1, GL_POSITION, LightPosition);   // Позиция света
```

Наконец, мы разрешаем источник света номер один. Мы не разрешили GL_LIGHTING, поэтому Вы еще не увидите никакого освещения. Свет установлен, и позиционирован, и даже разрешен, но пока мы не разрешили GL_LIGHTING, освещение не будет работать.

```
glEnable(GL_LIGHT1); // Разрешение источника света номер один
 return TRUE;         // Инициализация прошла OK
}
```

В следующей секции кода, мы нарисуем текстуру, наложенную на куб. Я буду комментировать только несколько строк, потому что они новые. Если Вы не понимаете не прокомментированные строки, вернитесь к уроку номер шесть.

```
int DrawGLScene(GLvoid)        // Здесь мы делаем все рисование
{
 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);   // Очистка Экрана и Буфера Глубины
 glLoadIdentity();       // Сброс Просмотра
```

Следующих трех строках кода куб с наложенной текстурой позиционируется и вращается. Вызов glTranslatef (0.0f, 0.0f, z) перемещает куб на значение z по оси z (от наблюдателя или к наблюдателю). Вызов glRotatef (xrot, 1.0f, 0.0f, 0.0f) использует переменную xrot, чтобы вращать куб по оси X. Вызов glRotatef (yrot, 1.0f, 0.0f, 0.0f) использует переменную yrot, чтобы вращать куб по оси Y.

```
glTranslatef(0.0f,0.0f,z);      // Перенос В/Вне экрана по z
glRotatef(xrot,1.0f,0.0f,0.0f); // Вращение по оси X на xrot
glRotatef(yrot,0.0f,1.0f,0.0f); // Вращение по оси Y по yrot
```

Следующая строка подобна строке, которую мы использовали в уроке шесть, но вместо связывания к texture[0], мы привязываем texture[filter]. Если мы нажимаем клавишу 'F', значение в filter увеличится. Если значение больше чем два, переменная filter сбрасывается в ноль. Когда программа запускается, filter будет установлена в ноль. Это все равно, что glBindTexture (GL_TEXTURE_2D, texture[0]). Если мы нажимаем "F" еще раз, переменная filter будет равна единице, что фактически glBindTexture (GL_TEXTURE_2D, texture[1]). Используя, переменную filter мы можем выбирать любую из трех текстур, которые мы создали.

```
 glBindTexture(GL_TEXTURE_2D, texture[filter]);    // Выбор текстуры основываясь на filter
 glBegin(GL_QUADS);       // Начало рисования четырехугольников
```

glNormal3f новая функция в моих уроках. Нормаль - линия, берущая начало из середины полигона под 90 углом градусов. Когда Вы используете освещение, Вы должны задать нормали. Нормаль сообщает OpenGL, в каком направлении у полигона лицевая часть, какое направление верхнее. Если Вы не задали нормали, могут происходить сверхъестественные вещи. Грань, которая не освещена, будет освещена, неверная сторона полигона будет освещена, и т.д. Нормаль должна указывать вовне полигона.
Посмотрев на переднюю грань, Вы заметите, что нормаль имеет положительное направление по оси Z. Это означает, что нормаль указывает на наблюдателя. Это точно, то направление, которое мы хотим указать. На обратной грани, нормаль указывает от наблюдателя вглубь экрана. Снова это точно то, что мы хотим. Если куб повернут на 180

градусов или по оси X или по оси Y, передняя грань ящика будет лицом вглубь экрана, и задняя грань ящика будет лицом к наблюдателю. Независимо от того, какую грань видит наблюдатель, нормаль этой грани будет также направлена на наблюдателя. Поскольку свет - близко к наблюдателю всегда нормаль, указывающая на наблюдателя, также указывает на свет. Если это сделать, то грань будет освещена. Чем больше нормалей указывает на свет, тем более яркая будет грань. Если Вы переместились в центр куба, Вы заметите, что там темно. Нормали – указывают вовне, а не во внутрь, поэтому нет освещения внутри куба.

```
   // Передняя грань
  glNormal3f( 0.0f, 0.0f, 1.0f);     // Нормаль указывает на наблюдателя
  glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f,  1.0f); // Точка 1 (Перед)
  glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f,  1.0f); // Точка 2 (Перед)
  glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f,  1.0f,  1.0f); // Точка 3 (Перед)
  glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f,  1.0f,  1.0f); // Точка 4 (Перед)

  // Задняя грань
  glNormal3f( 0.0f, 0.0f,-1.0f);     // Нормаль указывает от наблюдателя
  glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f); // Точка 1 (Зад)
  glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f,  1.0f, -1.0f); // Точка 2 (Зад)
  glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f,  1.0f, -1.0f); // Точка 3 (Зад)
  glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f); // Точка 4 (Зад)

  // Верхняя грань
  glNormal3f( 0.0f, 1.0f, 0.0f);     // Нормаль указывает вверх
  glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f,  1.0f, -1.0f); // Точка 1 (Верх)
  glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f,  1.0f,  1.0f); // Точка 2 (Верх)
  glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f,  1.0f,  1.0f); // Точка 3 (Верх)
  glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f,  1.0f, -1.0f); // Точка 4 (Верх)

  // Нижняя грань
  glNormal3f( 0.0f,-1.0f, 0.0f);     // Нормаль указывает вниз
  glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, -1.0f, -1.0f); // Точка 1 (Низ)
  glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, -1.0f, -1.0f); // Точка 2 (Низ)
  glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f,  1.0f); // Точка 3 (Низ)
  glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f,  1.0f); // Точка 4 (Низ)


  // Правая грань
  glNormal3f( 1.0f, 0.0f, 0.0f);     // Нормаль указывает вправо
  glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f); // Точка 1 (Право)
  glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f,  1.0f, -1.0f); // Точка 2 (Право)
  glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f,  1.0f,  1.0f); // Точка 3 (Право)
  glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f,  1.0f); // Точка 4 (Право)

  // Левая грань
  glNormal3f(-1.0f, 0.0f, 0.0f);     // Нормаль указывает влево
  glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f); // Точка 1 (Лево)
  glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f,  1.0f); // Точка 2 (Лево)
  glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f,  1.0f,  1.0f); // Точка 3 (Лево)
  glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f,  1.0f, -1.0f); // Точка 4 (Лево)
 glEnd();        // Кончили рисовать четырехугольник
```

В следующих двух строках увеличиваются значения xrot и yrot на значения, сохраненные в xspeed, и yspeed. Если значение в xspeed или yspeed большое, xrot и yrot увеличивается быстро. Чем быстрее увеличение xrot, или yrot, тем быстрее куб вращается по соответствующей оси.

```
 xrot+=xspeed;        // Добавить в xspeed значение xrot
 yrot+=yspeed;        // Добавить в yspeed значение yrot
 return TRUE;         // Выйти
}
```

Теперь мы спускаемся до WinMain (). Здесь добавим код для включения или выключения освещения, вращения корзины, смены фильтра и перемещения ящика ближе или дальше от экрана. Ближе к концу WinMain () Вы увидите вызов SwapBuffers (hDC). Сразу после этой строки, добавьте следующий код.

Этот код отслеживает нажатие клавиши 'L'. В первой строке проверяется, нажата ли 'L'. Если 'L' нажата, но lp - не ложь, что значит клавиша 'L' уже была нажата, или она удерживается нажатой, то тогда ничего не происходит.

```
SwapBuffers(hDC);     // Переключение буферов (Двойная буферизация)
if (keys['L'] && !lp) // Клавиша 'L' нажата и не удерживается?
{
```

Если lp – ложь то, это означает, что клавиша 'L' не нажата, иначе, если она уже отпущена, lp - истина. Это гарантирует, что клавишу 'L' отпустят прежде, чем этот код выполнится снова. Если мы не будет проверять удержание клавиши, освещение будет мерцать, постоянно включаясь и выключаясь много раз, поскольку программа думала бы, что Вы нажимает клавишу 'L' снова и снова, при этом вызывая этот раздел кода.
Переменная lp будучи равной истине, сообщает, что 'L' отпущена, мы включаем или выключаем освещение.
Переменная light может только быть истина или ложь. Поэтому, если мы говорим light=!light, то мы фактически говорим, что свет, не равен свету. По-русски это будет значит, что, если light равена истине, то light не будет истина (ложь), и если light равена ложь, то light не будет ложь (истина). Поэтому, если переменная light была истина, она станет ложь, и если light была ложь, она станет истина.

```
lp=TRUE;      // lp присвоили TRUE
light=!light; // Переключение света TRUE/FALSE
```

Теперь мы проверим, какое значение light получилось в конце. Первая строка, переведенная на русский означает: если light равняется ложь. Поэтому, если Вы совместите все строки вместе, то они делают следующее: если light равняется ложь, то надо запретить освещение. Это выключает все освещение. Команда 'else' означает: если light не ложь. Поэтому, если light не была ложь то, она истинна, поэтому мы включаем освещение.

```
if (!light)               // Если не свет
{
 glDisable(GL_LIGHTING);  // Запрет освещения
}
else                      // В противном случае
{
 glEnable(GL_LIGHTING);   // Разрешить освещение
}
}
```

Следующая строка отслеживает отжатие клавиши 'L'. Если мы присвоили переменной lp значение ложь, то это, означает, что клавиша 'L' не нажата. Если мы не будем отслеживать отжатие клавиши, мы могли бы включить освещение, но поскольку компьютер считал бы, что 'L' нажата, поэтому он не позволит нам выключить освещение.

```
if (!keys['L']) // Клавиша 'L' Отжата?
{
 lp=FALSE;      // Если так, то lp равно FALSE
}
```

Теперь мы сделаем что-то подобное с клавишей 'F'. Если клавиша нажата, и она не удерживается, или она не была нажата до этого, тогда присвоим значение переменной fp равное истине, что значит клавиша 'F' нажата и удерживается. При этом увеличится значение переменной filter. Если filter больше чем 2 (т.е. texture[3], а такой текстуры не существует), мы сбрасываем значение переменной texture назад в ноль.

```
if (keys['F'] && !fp) // Клавиша 'F' нажата?
{
 fp=TRUE;             // fp равно TRUE
 filter+=1;           // значение filter увеличивается на один
 if (filter>2)        // Значение больше чем 2?
 {
  filter=0;           // Если так, то установим filter в 0
 }
}
if (!keys['F'])       // Клавиша 'F' отжата?
 {
  fp=FALSE;           // Если так, то fp равно FALSE
 }
```

В следующих четырех строках проверяем, нажали ли мы клавишу 'Page up'. Если так, то уменьшим значение переменной z. Если эта переменная уменьшается, куб будет двигаться вглубь экрана, поскольку мы используем glTranslatef (0.0f, 0.0f, z) в процедуре DrawGLScene.

```
    if (keys[VK_PRIOR])   // Клавиша 'Page Up' нажата?
    {
     z-=0.02f;            // Если так, то сдвинем вглубь экрана
    }
```

В этих четырех строках проверяется, нажали ли мы клавишу 'Page down'. Если так, то увеличим значение переменной z и сместим куб к наблюдателю, поскольку мы используем glTranslatef (0.0f, 0.0f, z) в процедуре DrawGLScene.

```
    if (keys[VK_NEXT])    // Клавиша 'Page Down' нажата?
    {
     z+=0.02f;            // Если так, то придвинем к наблюдателю
    }
```

Теперь все, что мы должны проверить - клавиши курсора. Нажимая влево или вправо, xspeed увеличивается или уменьшается. Нажимая вверх или вниз, yspeed увеличивается или уменьшается. Помните выше в этом уроке, я говорил, что, если значения xspeed или yspeed большие, куб вращается быстрее. Чем дольше Вы удерживаете одну из клавиш курсора, тем, быстрее куб будет вращаться в соответствующем направлении.

```
    if (keys[VK_UP])     // Клавиша стрелка вверх нажата?
    {
     xspeed-=0.01f;      // Если так, то уменьшим xspeed
    }
    if (keys[VK_DOWN])   // Клавиша стрелка вниз нажата?
    {
     xspeed+=0.01f;      // Если так, то увеличим xspeed
    }
    if (keys[VK_RIGHT])  // Клавиша стрелка вправо нажата?
    {
     yspeed+=0.01f;      // Если так, то увеличим yspeed
    }

    if (keys[VK_LEFT])   // Клавиша стрелка влево нажата?
    {
     yspeed-=0.01f;      // Если так, то уменьшим yspeed
    }
```

Как и во всех предыдущих уроках, удостоверитесь, что заголовок наверху окна правильный.

```
    if (keys[VK_F1])          // Клавиша 'F1' нажата?
    {
     keys[VK_F1]=FALSE;       // Если так, то сделаем Key FALSE
     KillGLWindow();          // Уничтожим наше текущее окно
     fullscreen=!fullscreen;  // Переключение между режимами Полноэкранный/Оконный
                              // Повторное создание нашего окна OpenGL
     if (!CreateGLWindow("Урок NeHe Текстуры, Свет & Обработка Клавиатуры",640,480,16,fullscreen))
     {
      return 0;               // Выход, если окно не создано
     }
    }
   }
  }
 }

 // Сброс
 KillGLWindow();              // Уничтожение окна
 return (msg.wParam);         // Выход из программы
}
```

Здесь появляется новая функция KillGLWindow, которая занимается тем, что корректно уничтожает окно и все что с ним связано. Раньше тоже самое мы делали при обработке сообщений WM_DESTROY и WM_CLOSE. Вот код KillGLWindow, который нужно разместить сразу за функцией InitGL.

```
GLvoid KillGLWindow(GLvoid)               // Правильное уничтожение окна
{
   if (fullscreen)                        // Полноэкранный режим?
   {
      ChangeDisplaySettings(NULL,0);    // Переход в режим разрешения рабочего стола
      ShowCursor(TRUE);// Показать указатель мыши
   }

   if (hRC)                               // Существует контекст рендеринга?
   {
      if (!wglMakeCurrent(NULL,NULL))   // Можно ли освободить DC и RC контексты?
      {
         MessageBox(NULL,"Release Of DC And RC Failed.","SHUTDOWN ERROR",
                                    MB_OK | MB_ICONINFORMATION);
      }
      if (!wglDeleteContext(hRC))         // Можно ли уничтожить RC?
      {
         MessageBox(NULL,"Release Rendering Context Failed.",
               "SHUTDOWN ERROR",MB_OK | MB_ICONINFORMATION);
      }
      hRC=NULL;                           // Установим RC в NULL
   }

   if (hDC && !ReleaseDC(hWnd,hDC))      // Можно ли уничтожить DC?
   {
      MessageBox(NULL,"Release Device Context Failed.","SHUTDOWN ERROR",
                                  MB_OK | MB_ICONINFORMATION);
      hDC=NULL;                           // Установим DC в NULL
   }
   if (hWnd && !DestroyWindow(hWnd))     // Можно ли уничтожить окно?
   {

      MessageBox(NULL,"Could Not Release hWnd.","SHUTDOWN ERROR",MB_OK |
                                       MB_ICONINFORMATION);
      hWnd=NULL;                          // Устввновим hWnd в NULL
   }
   if (!UnregisterClass("OpenGL",hInstance)) // Можно ли уничтожить класс?
   {
      MessageBox(NULL,"Could Not Unregister Class.","SHUTDOWN ERROR",MB_OK |
                                        MB_ICONINFORMATION);
      hInstance=NULL;                     // Устанавливаем hInstance в NULL
   }
}
```

После освоения этого урока Вы должны уметь создавать и оперировать объектами из четырехугольников с высококачественным, реалистичным наложением текстур. Вы должны понять преимущества каждого из трех фильтров, используемых в этом уроке. Нажимая определенные клавиши на клавиатуре Вы сможете взаимодействовать с объектом на экране, и наконец, Вы должны знать, как применить простое освещение на сцене, делая сцену более реалистичной.

Урок 8. Смешивание

Blending

Простая прозрачность

Большинство специальных эффектов в OpenGL основано на так называемом смешивании (blending). Смешивание используется, чтобы комбинировать цвет данного пикселя, который должен быть выведен с пикселем, уже

находящемся на экране. Как будут объединены цвета, будет зависеть от альфа-канала, и/или функции смешивания, которая задается. Альфа - 4-й компонент цвета, который обычно указывается в конце. В прошлом Вы использовали GL_RGB, чтобы определить цвет с тремя компонентами. GL_RGBA может использоваться, чтобы также определить альфа-канал. Теперь мы можем использовать glColor4f() вместо glColor3f().

Большинство людей думает об альфе, как о непрозрачности материала. Альфа - значение 0.0 подразумевает, что материал полностью прозрачен. Значение 1.0 означает полную непрозрачность.

Уравнение смешивания

Если Вы не дружите с математикой, и только хотите уметь просто использовать эффект прозрачности, то пропустите этот раздел. Если Вы хотите понять, как работает смешивание, этот раздел - для Вас.

(Rs Sr + Rd Dr, Gs Sg + Gd Dg, Bs Sb + Bd Db, As Sa + Ad Da)

OpenGL вычислит результат смешивания двух пикселей, основанный на уравнении выше. Буквы s и r в нижнем регистре задают пиксели источника и приемника. S и D компоненты – коэффициенты смешивания (R,G,B,A – составляющие цвета). Их значения указывают, как бы Вы хотели смешать пиксели. Обычные значения для S и D: (As, As, As, As) (так называемые альфа источника (source alpha)) для S и (1, 1, 1, 1) - (As, As, As, As) (один минус альфа источника) для D. Отсюда уравнение смешивания будет выглядеть следующим образом:

(Rs As + Rd (1 - As), Gs As + Gd (1 - As), Bs As + Bs (1 - As), As As + Ad (1 - As))

Это уравнение определит стиль прозрачности/полу прозрачности.

Смешивание в OpenGL

Мы включаем эффект смешивания так же как что-либо другое. Затем мы задаем уравнение и выключаем буфер глубины на запись при рисовании прозрачных объектов, поскольку мы хотим увидеть объекты позади нарисованных полупрозрачных фигур. Это не идеальный способ смешивания, но в нашем простом проекте он будет работать достаточно хорошо.

Комментарий Rui Martins: более правильный путь состоит в том, чтобы вывести все прозрачные (с альфой < 1.0) полигоны после того, как Вы вывели полную сцену, и выводить их в обратном порядке глубины (сначала – дальние).

Это оттого, что смешивание двух полигонов (1 и 2) в различном порядке дает различные результаты, то есть если полигон 1 - самый близкий к наблюдателю, то нужно вывести полигон 2 и затем 1. Если провести аналогию с реальностью, то свет, проходящий сквозь эти два полигона (которые являются прозрачными), должен пройти сначала 2-й полигон и затем полигон 1 до того как достигнет глаза наблюдателя.

Вы должны СОРТИРОВАТЬ ПРОЗРАЧНЫЕ ПОЛИГОНЫ ПО ГЛУБИНЕ и выводить их ПОСЛЕ ТОГО, КАК БЫЛА ОТРИСОВАНА ПОЛНАЯ СЦЕНА с ВЫКЛЮЧЕННЫМ БУФЕРОМ ГЛУБИНЫ, или получится неправильно. Я знаю, что это иногда трудно, но это - правильный способ делать это.

Мы будем использовать код от урока семь. Мы начинаем, прибавляя две новых переменных в начало кода. Я приведу повторно полный раздел кода для ясности.

```
#include <windows.h>    // Заголовочный файл для Windows
#include <stdio.h>      // Заголовочный файл для стандартного ввода/вывода
#include <gl\gl.h>      // Заголовочный файл для библиотеки OpenGL32
#include <gl\glu.h>     // Заголовочный файл для для библиотеки GLu32
#include <gl\glaux.h>   // Заголовочный файл для библиотеки GLaux

HDC         hDC=NULL;  // приватный контекст устройства GDI
HGLRC       hRC=NULL;  // постоянный контекст рендеринга
HWND        hWnd=NULL; // содержит дискриптор нашего окна
HINSTANCE   hInstance; // содержит экземпляр нашего приложения

bool        keys[256];      // массив для процедуры обработки клавиатуры
bool        active=TRUE;    // флаг активности окна, по умолчанию TRUE
bool        fullscreen=TRUE; // флаг полноэкранного режима, по умолчанию полный экран
```

```
bool  blend; // Смешивание НЕТ/ДА? (НОВОЕ)
bool  light; // Освещение Вкл./Выкл.
bool  lp;    // L Нажата?
bool  fp;    // F Нажата?
bool  bp;    // B Нажата? ( Новое )

GLfloat xrot;     // Вращение вдоль оси X
GLfloat yrot;     // Вращение вдоль оси Y
GLfloat xspeed;   // Скорость вращения вдоль оси X
GLfloat yspeed;   // Скорость вращения вдоль оси X
GLfloat z=-5.0f;  // Глубина в экран.

// Задаем параметры освещения
GLfloat LightAmbient[]= { 0.5f, 0.5f, 0.5f, 1.0f };
GLfloat LightDiffuse[]= { 1.0f, 1.0f, 1.0f, 1.0f };
GLfloat LightPosition[]={ 0.0f, 0.0f, 2.0f, 1.0f };

GLuint filter;     // Используемый фильтр для текстур
GLuint texture[3]; // Хранит 3 текстуры

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);// Объявление для WndProc
```

Спускаемся вниз до LoadGLTextures(). Найдите строку, которая говорит texture1 = auxDIBImageLoad ("Data/crate.bmp"), измените ее на строку ниже. Мы используем текстуру окрашенного стекла для этого урока вместо текстуры ящика.

```
    // Загрузка текстуры стекла (МОДИФИЦИРОВАННО)
    texture1 = auxDIBImageLoad("Data/glass.bmp");
```

Прибавьте следующие две строки где-нибудь в InitGL(). Первая строка задает яркость для отрисовки объекта, равную полной яркости с альфой 50 % (непрозрачность). Это означает, когда включается смешивание, объект будет на 50% прозрачный. Вторая строка задает тип смешивания.

Комментарий Rui Martins: альфа, равное 0.0 подразумевало бы, что материал полностью прозрачен. Значение 1.0 было бы сиречь полностью непрозрачно.

```
glColor4f(1.0f,1.0f,1.0f,0.5f);   // Полная яркость, 50% альфа (НОВОЕ)
glBlendFunc(GL_SRC_ALPHA,GL_ONE); // Функция смешивания для непрозрачности,
                                  // базирующаяся на значении альфы(НОВОЕ)
```

Найдите следующий раздел кода, он должен быть в самом конце урока семь.

```
    if (keys[VK_LEFT])        // Нажата левая стрелка?
    {
        yspeed-=0.01f;      // уменьшаем скорость
    }
```

Под вышеупомянутым кодом мы прибавляем следующие строки. Отслеживаем нажатие ‘B’. Если она была нажато, компьютер проверит, включено ли смешивание. Если смешивание задано, компьютер выключает его. И наоборот, если смешивание выключено, включает его.

```
   if (keys['B'] && !bp)
    {
        bp=TRUE;
        blend = !blend;                 // Инвертируем blend
        if(blend)                       // blend TRUE?
        {
            glEnable(GL_BLEND);         // Включаем смешивание
            glDisable(GL_DEPTH_TEST);   // Выключаем тест глубины
        }
        else
        {
```

```
            glDisable(GL_BLEND);        // Выключаем смешивание
            glEnable(GL_DEPTH_TEST);    // Включаем тест глубины
        }
    }
    if (!keys['B'])                 // ’B’ отжата?
        {
        bp=FALSE;                   // тогда bp возвращает ложь
        }
```

Но как мы можем определить цвет, если используем картинку текстуры? Просто, в режиме модулирования текстуры, каждый пиксель, который отображен из текстуры умножается на текущий цвет. Поэтому, если цвет для отображения (0.5, 0.6, 0.4), мы умножаем его на цвет и получаем (0.5, 0.6, 0.4, 0.2) (альфа принимается равной 1.0, если не задается явно).

Отлично! Смешивание, действительно просто сделать в OpenGL.

Примечание от 13.11.99

Я (NeHe) модифицировал код смешивания, поэтому выводимые объекты выглядят лучше. Использование альфа значений для источника и приемника приводит к появлению артефактов при смешивании. От того, что задние грани выглядят более темными, чем боковые грани. От этого объект будет выглядеть очень странно. Способ, с помощью которого я делаю смешивание, может быть не лучший, но он работает, и объекты выглядят хорошо, даже когда включено освещение. Спасибо Тому за начальный код, способ, с помощью которого он смешивал , был правильный способ для смешивания с альфа значениями, но не выглядел привлекательно и как ожидают этого люди ;).

Код был модифицирован еще раз, для того чтобы исключить проблему, которая возникала при использовании glDepthMask(). Это команда не эффективно разрешала и запрещала тест буфера глубины на некоторых видеокартах, поэтому я восстановил старый код с glEnable и glDisable теста глубины.

Альфа из картинки текстуры

Альфа значение, которое используется для прозрачности, может быть получено из картинки текстуры точно также как цвет, для того чтобы сделать это, вы должны извлечь альфа из изображения, которое Вы хотите загрузить, и затем использовать GL_RGBA для формата цвета при вызове glTexImage2D().

Вопросы?

Если у Вас есть вопросы, не стесняйтесь послать их по адресу stanis@cs.wisc.edu <mailto:stanis@cs.wisc.edu>.

Урок 9. Передвижение изображений в 3D

Moving Bitmaps In 3D Space

Добро пожаловать на 9-й урок. На данный момент вы должны уже хорошо понимать суть OpenGL. Вы уже научились всему, начиная от создания окна и установки контекста OpenGL, до текстурирования вращающихся объектов с использованием освещения и смешивания (blending). Этот урок будет первым из серии "продвинутых" уроков. Вы научитесь следующему: перемещать изображение (bitmap) по экрану в 3D, удаляя, черные пикселы (pixels) из изображения (используя смешивание), дополнять цветность в черно-белые текстуры и, наконец, узнаете, как создавать красивые цвета и простую анимацию путём смешивания различных цветных текстур вместе.

Мы изменим код из первого урока. И начнем с добавления некоторых новых переменных в начале программы. Я переписал начальную секцию программы, для того чтобы было легче определять, где произведены изменения.

```
#include <windows.h>                // Заголовочный файл для Windows
#include <stdio.h>                  // Заголовочный файл для стандартного ввода/вывода
#include <gl\gl.h>                  // Заголовочный файл для библиотеки OpenGL32
#include <gl\glu.h>                 // Заголовочный файл для для библиотеки GLu32
#include <gl\glaux.h>               // Заголовочный файл для библиотеки GLaux
```

```
HDC         hDC=NULL;           // Служебный контекст GDI устройства
HGLRC       hRC=NULL;           // Постоянный контекст для визуализации
HWND        hWnd=NULL;          // Содержит дискриптор для окна
HINSTANCE   hInstance;          // Содержит данные для нашей программы

bool    keys[256];      // Массив, использующийся для сохранения состояния клавиатуры
bool    active=TRUE;    // Флаг состояния активности приложения (по умолчанию: TRUE)
bool    fullscreen=TRUE;// Флаг полноэкранного режима (по умолчанию: полноэкранное)
```

Следующие строчки новые. twinkle и tp логические переменные, которые могут быть TRUE (истина) или FALSE (ложь). twinkle будет говорить о включении/выключении эффекта twinkle. tp используется для определения состояния клавиши 'T' (была ли нажата или нет). Если нажата, то tp=TRUE, иначе tp=FALSE.

```
BOOL    twinkle;            // Twinkling Stars (Вращающиеся звезды)
BOOL    tp;                 // 'T' клавиша нажата?
```

num переменная, хранит информацию о количестве звезд, которые мы рисуем на экране. Она определена как константа. Это значит, в коде программы мы не может поменять её значение. Причина, по которой мы определяем её как константу, в том, что мы не можем переопределить (увеличить/уменьшить) массив. Так или иначе, если мы задаём массив только на 50 звезд и хотим увеличить num до 51 звезды где-нибудь в программе, то массив не сможет увеличиться, и выдаст ошибку. Вы можете изменить num только в этой строчке программы. И не пытайтесь изменить значение num где-то в другом месте, если вы не хотите, чтобы случилось страшное :).

```
const   num=50;             // Количество рисуемых звезд
```

Сейчас мы создадим структуру (structure). Слово структура звучит глобально, но это не так на самом деле. Структура это совокупность простых данных (переменных, и т.д.) сгруппированых по какому-либо признаку в одну группу. Мы знаем, что мы будем хранить цепочку звезд. Вы увидите, что 7-ая строчка ниже это stars;. Мы знаем, что каждая звезда имеет 3 значения для цвета, и все эти значения целые числа: 3-я строчка: int r,g,b задаёт эти значения. Одно для красного (red) (r), одно для зелёного (green) (g), и одно для голубого (blue) (b). Мы знаем, что каждая звезда будет иметь разное расстояние от центра экрана, и расположена на одном из 360 углов от центра. Если вы посмотрите на 4-ую строчку ниже, вы увидите это. Мы создаём переменную типа число с плавающей точкой (floating point value) называется dist. Она означает расстояние. 5-ая строчка создаёт переменную того же типа с именем angle. Она будет отвечать за угол звезды.

И так мы имеем группу данных, которая содержит цвет, расстояние и угол звезды на экране. К сожалению, у нас больше чем одна звезда, и вместо того чтобы создавать 50 переменных для красного цвета, 50 переменных для зеленого цвета, 50 переменных для синего цвета, 50 переменных для расстояния, 50 переменных для угла мы просто создадим массив и назовем его star. Под каждым номером в массиве star содержится информация о нашей структуре stars. Это мы делаем в 8-ой строке ниже: stars star[num]. Тип элемента массива будет stars. stars это структура. И массив содержит всю информацию в структурах. Массив называется star. Количество элементов - [num]. И так как num=50, мы имеем массив с именем star. Наш массив содержит элементы типа структура stars. Намного проще, чем хранить каждую звезду в отдельных переменных. Что было бы большой глупостью и не позволило добавлять или уменьшать количество звезд с помощью переменной num.

(Прим. перев. - Такое ощущение, что объясняешь слону, как ходить :). Можно было одной строчкой это объяснить.)

```
typedef struct                  // Создаём структуру для звезд
{
    int r, g, b;                // Цвет звезды
    GLfloat dist;               // Расстояние от центра
    GLfloat angle;              // Текущий угол звезды
}
stars;                          // Имя структуры - Stars
stars star[num];                // Делаем массив 'star' длинной 'num',
                                // где элементом является структура 'stars'
```

Далее мы задаём переменную для хранения расстояния от наблюдателя до звезд (zoom), и какой будет начальный угол(tilt). Также мы делаем переменную spin, которая будет вращать звезды по оси z, и это будет выглядеть как вращение вокруг их текущей позиции.

loop это переменная, которую мы используем в программе для отрисовки всех 50-ти звезд, и texture[1] будет использоваться для хранения одной черно-белой текстуры, которую мы загружаем. Если вы хотите больше текстур, вы должны увеличить длину массива, с одного до нужной длины.

```
GLfloat zoom=-15.0f;                // Расстояние от наблюдателя до звезд
GLfloat tilt=90.0f;                 // Начальный угол
GLfloat spin;                       // Для вращения звезд

GLuint  loop;                       // Используется для циклов
GLuint  texture[1];                 // Массив для одной текстуры

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);   // Объявления для WndProc
```

Сразу же за переменными мы добавляем код для загрузки текстур. Я не буду объяснять этот код в деталях, это тот же код что был в 6, 7 и 8 уроке. Изображение, которое мы загружает называется star.bmp. Мы генерируем только одну текстуру используя glGenTextures(1, &texture[0]). Текстура будет иметь линейную фильтрацию (linear filtering).

```
AUX_RGBImageRec *LoadBMP(char *Filename)// Функция для загрузки bmp файлов
{
    FILE *File=NULL;                // Переменная для файла

    if (!Filename)                  // Нужно убедиться в правильности переданого имени
    {
        return NULL;                // Если неправильное имя, то возвращаем NULL
    }

    File=fopen(Filename,"r");       // Открываем и проверяем на наличие файла

    if (File)                       // Файл существует?
    {
        fclose(File);               // Если да, то закрываем файл
    // И загружаем его с помощью библиотеки AUX, возращая ссылку на изображение
        return auxDIBImageLoad(Filename);
    }
    // Если загрузить не удалось или файл не найден, то возвращаем NULL
    return NULL;
}
```

Эта секция кода загружает изображение (описанным выше кодом) и конвертирует в текстуру. Status хранит информацию об успехе операции.

```
int LoadGLTextures()    // Функция загрузки изображения и конвертирования в текстуру
{
    int Status=FALSE;               // Индикатор статуса

    AUX_RGBImageRec *TextureImage[1];// Создаём место для хранения текстуры

    memset(TextureImage,0,sizeof(void *)*1);// устанавливаем ссылку на NULL

    // Загружаем изображение, Проверяем на ошибки, Если файл не найден то выходим
    if (TextureImage[0]=LoadBMP("Data/Star.bmp"))
    {
        Status=TRUE;            // Ставим статус в TRUE

        glGenTextures(1, &texture[0]);  // Генерируем один индификатор текстуры

        // Создаём текстуру с линейной фильтрацией (Linear Filtered)
        glBindTexture(GL_TEXTURE_2D, texture[0]);
        glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
        glTexImage2D(GL_TEXTURE_2D, 0, 3, TextureImage[0]->sizeX,
    TextureImage[0]->sizeY, 0, GL_RGB, GL_UNSIGNED_BYTE, TextureImage[0]->data);
    }
```

```
    if (TextureImage[0])            // Если текстура существует
    {
        if (TextureImage[0]->data)   // Если изображение существует
        {
            // Освобождаем место выделенное под изображение
            free(TextureImage[0]->data);
        }

        free(TextureImage[0]);  // Освобождаем структуры изображения
    }

    return Status;                  // Возвращаем статус
}
```

Теперь мы настраиваем OpenGL для отрисовки того, что будет нужно. Нам не нужен Z-буфер (тест глубины) для нашего проекта, убедитесь что удалены строчки из первого урока: glDepthFunc(GL_LEQUAL); и glEnable(GL_DEPTH_TEST); иначе вы не получите нужного результата Мы используем текстурирование в этом коде, значит надо добавить все необходимые строки, которых нет в первом уроке. Включаем текстурирование и смешивание.

```
int InitGL(GLvoid)                  // Всё установки OpenGL будут здесь
{
    if (!LoadGLTextures())          // Загружаем текстуру
    {
        return FALSE;           // Если не загрузилась, то возвращаем FALSE
    }

    glEnable(GL_TEXTURE_2D);        // Включаем текстурирование
    // Включаем плавную ракраску (интерполирование по вершинам)
    glShadeModel(GL_SMOOTH);
    glClearColor(0.0f, 0.0f, 0.0f, 0.5f);   // Фоном будет черный цвет
    glClearDepth(1.0f);                 // Установки буфера глубины (Depth Buffer)
    // Максимальное качество перспективной коррекции
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
    // Устанавливаем функцию смешивания
    glBlendFunc(GL_SRC_ALPHA,GL_ONE);
    glEnable(GL_BLEND);                 // Включаем смешивание
```

Следующий фрагмент кода новый. Он устанавливает начальные углы, расстояние и цвет для каждой звезды. Заметьте, как просто менять информацию в структуре. Цикл по всем 50 звездам. Чтобы изменить угол star[1] все, что мы должны написать - это star[1].angle={некоторое значение}. Это просто!

```
    for (loop=0; loop<num; loop++)      // Делаем цикл и бежим по всем звездам
    {
        star[loop].angle=0.0f;  // Устанавливаем всё углы в 0
```

Я рассчитываю дистанцию взяв текущий номер звезды (это значение loop) и разделив на максимальное значение звезд. Потом я умножаю результат на 5.0f. Коротко, что это даёт - это отодвигает каждую звезду немного дальше, чем предыдущую. Когда loop равен 50 (последняя звезда), loop разделенный на num будет равен 1.0f. Я умножаю на 5 потому, что 1.0f*5.0f будет 5.0f. 5.0f это почти на границе экрана. Я не хочу, что бы звезды уходили за пределы экрана, так что 5.0f это идеально. Если вы установите zoom подальше в экран, вы должны использовать большее число, чем 5.0f, но ваши звезды должны быть немного меньше (из-за перспективы).

Заметьте, что цвета для каждой звезды задаются случайным образом от 0 до 255. Вы можете спросить, как мы может использовать эти числа, когда нормальное значение цвета от 0.0f до 1.0f. Отвечаю. Когда мы устанавливаем цвет, мы используем glColor4ub вместо glColor4f. ub значит Unsigned Byte (беззнаковый байт). И байт может иметь значения от 0 до 255. В этой программе легче использовать байты, чем работать с со случайными числами с плавающей запятой.

```
        // Вычисляем растояние до центра
        star[loop].dist=(float(loop)/num)*5.0f;
        // Присваиваем star[loop] случайное значение (красный).
        star[loop].r=rand()%256;
```

```
            // Присваиваем star[loop] случайное значение (зеленый)
            star[loop].g=rand()%256;
            // Присваиваем star[loop] случайное значение (голубой)
            star[loop].b=rand()%256;
        }
        return TRUE;                    // Инициализация прошла нормально.
}
```

Код функции Resize тот же самый, так что рассмотрим код отрисовки сцены. Если вы используете код из первого урока, то удалите весь код из DrawGLScene и просто скопируйте, то, что написано ниже. Здесь только две строки из первого урока, вообщем удалять немного придется.

```
int DrawGLScene(GLvoid)                 // Здесь мы всё рисуем
{
        // Очищаем буфер цвета и глубины
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        // Выбираем нашу текстуру
        glBindTexture(GL_TEXTURE_2D, texture[0]);

        for (loop=0; loop<num; loop++)              // Цикл по всем звездам
        {
            // Обнуляем видовую матрицу (Model Matrix) перед каждой звездой
            glLoadIdentity();
            // Переносим по оси z на 'zoom'
            glTranslatef(0.0f,0.0f,zoom);
            // Вращаем вокруг оси x на угол 'tilt'
            glRotatef(tilt,1.0f,0.0f,0.0f);
```

Теперь мы двигаем звезды! :) Звезда появляется в середине экрана. Первым делом мы вращаем сцену вокруг оси y. Если угол 90 градусов, то ось x будет лежать не слева направо, а наоборот и выходить за пределы экрана. В качестве примера: представьте, что вы стоите в центре комнаты. Теперь представьте, что слева на стене написано -x, впереди на стене написано -z, справа написано +x, в сзади написано +z. Если повернуться налево на 90 градусов, но не двигаться с места, то на стене впереди будет не -z, а -x. Все стены поменяются. -z будет справа, +z будет слева, -x впереди, и +x сзади. Проясняется? Вращая сцену, мы изменяем направления осей x и z.

Во второй строчке мы сдвигаем позицию по оси x. Обычно положительное значение по x двигает нас вправую сторону экрана (где обычно +x), но так как мы повернулись вокруг оси y, +x может быть хоть где. Если мы повернём на 180 градусов, +x будет с левой стороны экрана вместо правой. И так, когда мы двигаемся вперед по оси x, мы можем подвинуться влево, вправо, вперед и даже назад.

```
            // Поворачиваем на угол звезды вокруг оси y
            glRotatef(star[loop].angle,0.0f,1.0f,0.0f);
            // Двигаемся вперед по оси x
            glTranslatef(star[loop].dist,0.0f,0.0f);
```

Теперь немного хитрого кода. Звезда это всего лишь плоская текстура. И если мы нарисовали плоский квадрат в середине экрана с наложенной текстурой - это будет выглядеть отлично. Он будет, повернут к вам, как и должен быть. Но если повернёте его вокруг оси y на 90 градусов, текстура будет направлена вправо или влево экрана. Всё что вы увидите это тонкую линию. Нам не нужно чтобы это случилось. Мы хотим, чтобы звезды были направлены на наблюдателя всё время, не важно как они вращаются и двигаются по экрану.

Мы добиваемся этого путем отмены вращения, которое мы делаем, перед тем как нарисовать звезду. Мы отменяем вращение в обратном порядке. Выше мы повернули экран, когда сделали вращение на угол звезды. В обратном порядке, мы должны повернуть обратно звезду на текущий угол. Чтобы сделать это, мы используем обратное значение угла, и повернём звезду на этот угол. И так как мы повернули звезду на 10 градусов, то, поворачивая обратно на -10 градусов мы делаем звезду повернутой к наблюдателю по y. Первая строчка ниже отменяет вращение по оси y. Потом мы должны отменить вращение по оси x. Чтобы сделать это мы вращаем звезду на угол -tilt. После всех этих операций звезда полностью повернута к наблюдателю.

```
            glRotatef(-star[loop].angle,0.0f,1.0f,0.0f);
            // Отменяет текущий поворот звезды
            glRotatef(-tilt,1.0f,0.0f,0.0f);        // Отменяет поворот экрана
```

Если twinkle равно TRUE, мы рисуем не вращающуюся звезду на экране. Для того чтобы получить, различные цвета, мы берём максимальное количество звезд (num) и вычитаем текущий номер (loop), потом вычитаем единицу, так как наш цикл начинается с 0 и идет до num-1. Если результат будет 10, то мы используем цвет 10- ой звезды. Вследствие того, что цвет двух звезд обычно различный. Не совсем хороший способ, но зато эффективный. Последнее значение это альфа (alpha). Чем меньше значение, тем прозрачнее звезда (т.е. темнее, т.к. экран черный).
Если twinkle включен, то каждая звезда будет отрисована дважды. Программы будет работать медленнее в зависимости от типа вашего компьютера. Если twinkle включен, цвета двух звезд будут смешиваться вместе для создания реально красивых цветов. Так как это звезды не вращаются, то это проявлялось бы как будто звёзды анимировались, когда twinkling включен. (Если не понятно, то просто посмотрите).
Заметьте, как просто добавить цвет в текстуру. Даже если текстура черно-белая, она будет окрашена в цвет, который мы выбрали, перед тем как нарисовать текстуру. Также возьмите на заметку что мы используем байты для значений цвета, а не числа с плавающей запятой. Даже альфа тоже представляется байтом.

```
        if (twinkle)                        // Если Twinkling включен
        {       // Данный цвет использует байты
            glColor4ub(star[(num-loop)-1].r,star[(num-loop)-1].g,
            star[(num-loop)-1].b,255);
            glBegin(GL_QUADS);// Начинаем рисовать текстурированый квадрат
                glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f,-1.0f, 0.0f);
                glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f,-1.0f, 0.0f);
                glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 0.0f);
                glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 0.0f);
            glEnd();                        // Закончили рисовать
        }
```

Теперь мы рисуем основную звезду. Разница между кодом выше только в том, что звезда всегда рисуется, и вращается по оси z.

```
        glRotatef(spin,0.0f,0.0f,1.0f);// Поворачиваем звезду по оси z
        // Цвет использует байты
        glColor4ub(star[loop].r,star[loop].g,star[loop].b,255);
        glBegin(GL_QUADS);              // Начинаем рисовать текстурный квадрат
            glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f,-1.0f, 0.0f);
            glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f,-1.0f, 0.0f);
            glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 0.0f);
            glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 0.0f);
        glEnd();                            // Закончили рисовать
```

Здесь мы делаем все движения. Мы вращаем звезду увеличением значения spin. Потом мы меняем угол каждой звезды. Угол каждой звезды увеличивается на loop/num. Что ускоряет вращение звезды с отдалением от центра. Чем ближе к центру, тем медленнее вращается звезда. Наконец мы уменьшаем расстояние до центра для каждой звезды. Это создаёт эффект засасывания звезд в центр экрана.

```
        spin+=0.01f;                // Используется для вращения звезды
        star[loop].angle+=float(loop)/num;// Меняем угол звезды
        star[loop].dist-=0.01f; // Меняем растояние до центра
```

Нижеследующие строки проверяют видимость звезд. Попала звезда в центр экрана или нет. Если попала, то даём звезде новый цвет и двигаем на 5 единиц от центра. И так она может снова начать своё путешествие к центру как новая звезда.

```
        if (star[loop].dist<0.0f)    // Звезда в центре экрана?
        {
            star[loop].dist+=5.0f;  // Перемещаем на 5 единиц от центра
            // Новое значение красной компоненты цвета
            star[loop].r=rand()%256;
            // Новое значение зеленной компоненты цвета
            star[loop].g=rand()%256;
            // Новое значение синей компоненты цвета
            star[loop].b=rand()%256;
        }
    }
    return TRUE;                            // Всё ок
}
```

Теперь мы добавляем код для проверки нажатия клавиш. Идем ниже в WinMain(). Ищите строку с SwapBuffers(hDC). Мы добавляем код для нашей клавиши прямо под этой строкой.

Строкой ниже мы проверяем нажатие клавиши 'T'. Если нажата, и не была до этого нажата, то идем дальше. Если twinkle равно FALSE, то она станет TRUE. Если была TRUE, то станет FALSE. Одно нажатие 'T' установит tp равное TRUE. Это предотвращает постоянное выполнение этого кода, если клавиша 'T' удерживается.

```
SwapBuffers(hDC);               // Смена буфера (Double Buffering)
if (keys['T'] && !tp)           // Если 'T' нажата и tp равно FALSE
{
    tp=TRUE;                    // то делаем tp равным TRUE
    twinkle=!twinkle;           // Меняем значение twinkle на обратное
}
```

Код ниже проверяет "выключение" (повторное нажатие) клавиши 'T'. Если да, то делаем tp=FALSE. Нажатие 'T' делает ничего кроме установки tp равной FALSE, так что эта часть кода очень важная.

```
if (!keys['T'])                 // Клавиша 'T' была отключена
{
    tp=FALSE;                   // Делаем tp равное FALSE
}
```

Следующий код проверяет, нажаты ли клавиши 'стрелка вверх', 'стрелка вниз', 'page up', 'page down'.

```
if (keys[VK_UP])                // Стрелка вверх была нажата?
{
    tilt-=0.5f;                 // Вращаем экран вверх
}

if (keys[VK_DOWN])              // Стрелка вниз нажата?
{
    tilt+=0.5f;                 // Вращаем экран вниз
}

if (keys[VK_PRIOR])             // Page Up нажат?
{
    zoom-=0.2f;                 // Уменьшаем
}

if (keys[VK_NEXT])              // Page Down нажата?
{
    zoom+=0.2f;                 // Увеличиваем
}
```

Как и других предыдущих уроках, убедимся, что название окна корректно.

```
if (keys[VK_F1])                // Если F1 нажата?
{
    keys[VK_F1]=FALSE;          // Делаем клавишу равной FALSE
    KillGLWindow();             // Закрываем текущее окно
    fullscreen=!fullscreen;
// Переключаем режимы Fullscreen (полноэкранный) / Windowed (обычный)
    // Пересоздаём OpenGL окно
    if (!CreateGLWindow("NeHe's Textures,
     Lighting & Keyboard Tutorial",640,480,16,fullscreen))
    {
        return 0;               //Выходим если не получилось
    }
}
}
}
```

В этом уроке я попытался объяснить, как можно детальнее как загружать черно-белое изображение, удалять черный фон на изображении (используя смешивание), добавлять цвет в картинку, и двигать ее по экрану в трехмерном пространстве. Я также попытался показать, как сделать красивые цвета и анимацию путем наложения второй копии изображения поверх оригинала. Теперь вы имеете хорошее представление обо всем, что я хотел вам рассказать на данный момент. У вас теперь не должно возникнуть проблем с созданием своих собственных демок в 3D. Все начальные данные изложены тут.

Урок 10. Загрузка и перемещение в трехмерном мире

Loading And Moving Through A 3D World

Этот урок был написан человеком по имени Lionel Brits (Betelgeuse). Урок содержит только те части кода, которые нужно добавить. Но если просто добавите строки кода, описанные ниже, программа не будет работать. Если вы хотите знать, где должна идти каждая строка кода, скачайте исходник и просмотрите его так же, как вы прочитали этот урок.
Добро пожаловать в непопулярный Урок 10. Сейчас у вас есть вращающийся куб или цепочка звёзд и некоторая базовая ориентация в 3D программировании. Но стойте! Не убегайте сразу же и не начинайте создание Quake IV, пока что. Просто, вращающиеся кубики не доставят вам много удовольствий в десматче J. Вместо этого вам нужен большой, сложный и динамический 3D мир со свободным взглядом во все 6 сторон и с причудливыми эффектами такими, как зеркала, порталы, искривления и, конечно же, с высокой частотой кадров в секунду. В этом уроке представлена базовая «структура» 3D мира и, конечно же, способы перемещения по нему.

Структура данных

До сих пор было легко определять среду 3D мира, используя длинные комбинации чисел, но это становится чрезвычайно нелегко, когда сложность среды начинает возрастать. По этой причине мы должны организовать данные в более мощные структуры. Прежде всего обсудим понятие сектора. Каждый 3Д мир базируется на некотором количестве секторов. Сектор может быть комнатой, кубом или любым другим замкнутым пространством.

```
typedef struct tagSECTOR        // Создаём структуру нашего сектора
{
    int numtriangles;          // Кол-во треугольников в секторе
    TRIANGLE* triangle         // Ссылка на массив треугольников
} SECTOR;                      // Обзовём структуру словом SECTOR
```

Сектор содержит ряд многоугольников, однако мы будем использовать треугольники, потому что их проще всего запрограммировать.

```
typedef struct tagTRIANGLE      // Создаём стр-ру нашего треугольника
{
    VERTEX vertex[3];          // Массив трёх вершин
} TRIANGLE;                    // Обзовём это TRIANGLE
```

Треугольник, как и любой многоугольник, определяется вершинами. Вершина содержит реальные для использования OpenGL'ом данные. Мы определяем каждую точку треугольника её расположением в 3D пространстве (x, y, z) и координатами на текстуре (u, v).

```
typedef struct tagVERTEX        // Создаём стр-ру нашей вершины
{
    float x, y, z;    // 3D координаты
    float u, v;               // Координаты на текстуре
} VERTEX;                      // Обзовём это VERTEX
```

Загрузка файлов

Сохранение данных нашего мира внутри программы делает её слишком статичной и скучной. Загрузка миров с диска, тем не менее, даёт больше гибкости в тестировании различных миров, избавляя от перекомпиляции нашей программы. Другое преимущество в том, что пользователь может использовать новые уровни и модифицировать их, не задумываясь о коде нашей программы. Тип файла данных, который мы собираемся использовать будет текстовым. Это сделано для того, чтобы облегчить редактирование мира и уменьшить код программы. Оставим двоичные файлы на дальнейшее рассмотрение.

Естественный вопрос: как мы извлечем данные из нашего файла? Во-первых, мы создадим новую функцию SetupWorld(). Определим наш файл как filein и откроем его в режиме только чтение. А так же, когда закончим, мы должны не забыть закрыть наш файл. Давайте посмотрим на следующий код:

```
// Декларация выше: char* worldfile = "data\\world.txt";
void SetupWorld()                       // Установка нашего мира
{
    FILE *filein;                       // Файл для работы
    filein = fopen(worldfile, "rt");    // Открываем наш файл
          ...
    (считываем наши данные)
           ...
    fclose(filein);                     // Закрываем наш файл
    return;                             // Возвращаемся назад
}
```

Следующее, чему мы уделим внимание, будет собственно считывание каждой строки текста в переменную. Это можно выполнить очень многими способами. Одна проблема в том, что не все строки содержат значимую информацию. Пустые линии и комментарии не должны быть считаны. Теперь создадим функцию readstr(). Она будет считывать одну значимую строку в инициализированную строку. Вот этот код:

```
void readstr(FILE *f,char *string)  // Считать в строку
{
    do                          // Начинаем цикл
    {
        fgets(string, 255, f);  // Считываем одну линию
        // Проверяем её на условие повт. цикла
    } while ((string[0] == '/') || (string[0] == '\n'));
    return;                     // Возврат
}
```

Далее мы должны считать данные сектора. В этом уроке мы будем иметь дело только с одним сектором, но достаточно просто реализовать многосекторный движок. Давайте вернёмся к SetupWorld(). Наша программа должна знать сколько треугольников в секторе. В нашем файле данных мы обозначим количество треугольников следующим образом:

NUMPOLLIES n
Вот код для чтения количества треугольников:

```
int numtriangles;           // Кол-во треугольников в секторе
char oneline[255];          // Строка для сохранения данных
...
readstr(filein,oneline);    // Считать одну линию данных
// Считать кол-во треугольников
sscanf(oneline, "NUMPOLLIES %d\n", &numtriangles);
```

Остальная часть нашего процесса загрузки мира будет использовать тот же процесс. Далее мы инициализируем наш сектор и считываем в него некоторые данные:

```
// Декларация выше: SECTOR sector1;
char oneline[255];              // Строка для сохранения данных
int numtriangles;               // Кол-во треугольников в секторе
float x, y, z, u, v;            // 3D и текстурные координаты
...
// Выделяем память для  numtriangles и устанавливаем ссылку
sector1.triangle = new TRIANGLE[numtriangles];
// Определяем кол-во треугольников в Секторе 1
sector1.numtriangles = numtriangles;
// Цикл для всех треугольников
// За каждый шаг цикла – один треугольник в секторе
for (int triloop = 0; triloop < numtriangles; triloop++)
{
```

```
    // Цикл для всех вершин
// За каждый шаг цикла – одна вершина в треугольнике
    for (int vertloop = 0; vertloop < 3; vertloop++) {
        readstr(filein,oneline); // Считать строку для работы
        // Считать данные в соответствующие переменные вершин
        sscanf(oneline, "%f %f %f %f %f", &x, &y, &z, &u, &v);
        // Сохранить эти данные
        sector1.triangle[triloop].vertex[vertloop].x = x;
     // Сектор 1, Треугольник  triloop, Вершина vertloop, Значение x = x
        sector1.triangle[triloop].vertex[vertloop].y = y;
     // Сектор 1, Треугольник  triloop, Вершина vertloop, Значение y = y
        sector1.triangle[triloop].vertex[vertloop].z = z;
     // Сектор 1, Треугольник  triloop, Вершина vertloop, Значение z = z
       sector1.triangle[triloop].vertex[vertloop].u = u;
     // Сектор 1, Треугольник  triloop, Вершина vertloop, Значение u = u
        sector1.triangle[triloop].vertex[vertloop].v = v;
     // Сектор 1, Треугольник  triloop, Вершина vertloop, Значение y = v
    }
}
```

Каждый треугольник в нашем файле данных имеет следующую структуру:

```
X1 Y1 Z1 U1 V1
X2 Y2 Z2 U2 V2
X3 Y3 Z3 U3 V3
```

Отображение миров

Теперь, когда мы можем загружать наш сектор в память, нам нужно вывести его на экран. Итак, мы уже умеем делать немного простых вращений и проекций, но наша камера всегда находилась в начальном положении (0, 0, 0). Любой хороший 3D движок должен предоставлять пользователю возможность ходить и исследовать мир, так мы и сделаем. Одна из возможностей сделать это – перемещать камеру и перерисовывать 3D среду относительно её положения. Это медленно выполняется и тяжело запрограммировать. Поэтому мы будем делать так:

1) Вращать и проецировать позицию камеры следуя командам пользователя.
2) Вращать мир вокруг начала координат противоположно вращению камеры (это даёт иллюзию того, что повернулась камера).
3) Переместить мир способом, противоположным перемещению камеры (опять-таки, это даёт иллюзию того, что переместилась камера).

Это красиво и легко реализовывается. Давайте начнём с первого этапа (Вращение и проецирование камеры).

```
if (keys[VK_RIGHT])             // Была ли нажата правая стрелка?
{
    yrot -= 1.5f;               // Вращать сцену влево
}

if (keys[VK_LEFT])              // Была ли нажата левая стрелка?
{
    yrot += 1.5f;               // Вращать сцену вправо
}

if (keys[VK_UP])                // Была ли нажата стрелка вверх?
{
    // Переместиться по оси X вектора направления игрока
    xpos -= (float)sin(heading*piover180) * 0.05f;
    // Переместиться по оси Z вектора направления игрока
    zpos -= (float)cos(heading*piover180) * 0.05f;
    if (walkbiasangle >= 359.0f)// walkbiasangle>=359?
    {
        walkbiasangle = 0.0f; // Присвоить walkbiasangle  0
    }
```

```
    else                    // В противном случае
    {
    // Если walkbiasangle < 359 увеличить его на 10
        walkbiasangle+= 10;
    }
    // Иммитация походки человека
    walkbias = (float)sin(walkbiasangle * piover180)/20.0f;
}

if (keys[VK_DOWN])          // Была ли нажата стрелка вниз?
{
    // Переместиться по оси X вектора направления игрока
    xpos += (float)sin(heading*piover180) * 0.05f;
    // Переместиться по оси Z вектора направления игрока
    zpos += (float)cos(heading*piover180) * 0.05f;
    if (walkbiasangle <= 1.0f)    // walkbiasangle<=1?
    {
        walkbiasangle = 359.0f; // Присвоить walkbiasangle 359
    }
    else                    // В противном случае
    {
// Если walkbiasangle >1 уменьшить его на 10
walkbiasangle-= 10;
    }
    // Иммитация походки человека
    walkbias = (float)sin(walkbiasangle * piover180)/20.0f;
}
```

Это было довольно просто. Когда нажата стрелка влево или стрелка вправо, переменная вращения yrot увеличивает или уменьшает своё значение. Когда нажата стрелка вверх или стрелка вниз, новое положение камеры высчитывается с использованием синуса и косинуса (требуется немного тригонометрии J). Piover180 это просто коэффициент преобразования для перевода градусов в радианы.

Далее вы спросите меня: что такое walkbias (дословно: смещение походки)? Это слово, которое я изобрёл J. Оно представляет собой смещение, которое происходит, когда персона идёт (голова смещается вверх и вниз как буй). Это легко устанавливается изменением Y позиции камеры по синусоиде. Я решил использовать это, потому что простое перемещение вперёд и назад выглядит не реально.

Теперь когда эти переменные получили свои значения, мы можем сделать второй и третий шаги. Они будут сделаны в цикле отображения, так как наша программа на сложная, чтобы заслужить для этого отдельную функцию.

```
int DrawGLScene(GLvoid)         // Нарисовать сцену OpenGL
{
// Очистить сцену и буфер глубины
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();           // Сбросить текущую матрицу
    // Вещ. перем. для временных X, Y, Z, U и V
    GLfloat x_m, y_m, z_m, u_m, v_m;
    GLfloat xtrans = -xpos;     // Проекция игрока на ось X
    GLfloat ztrans = -zpos;     // Проекция игрока на ось Z
    // Для смещения изображения вверх и вниз
    GLfloat ytrans = -walkbias-0.25f;
    // 360 градусный угол для поворота игрока
    GLfloat sceneroty = 360.0f - yrot;
    int numtriangles;           // Количество треугольников
   glRotatef(lookupdown,1.0f,0,0);// Вращать вверх и вниз
// Вращать в соответствии с направлением взгляда игрока
    glRotatef(sceneroty,0,1.0f,0);
    // Проецировать сцену относительно игрока
    glTranslatef(xtrans, ytrans, ztrans);
    // Выбрать текстуру filter
    glBindTexture(GL_TEXTURE_2D, texture[filter]);
    // Получить кол-во треугольников Сектора 1
```

```
    numtriangles = sector1.numtriangles;
    // Процесс для каждого треугольника
    // Цикл по треугольникам
    for (int loop_m = 0; loop_m < numtriangles; loop_m++)
    {
        glBegin(GL_TRIANGLES);  // Начинаем рисовать треугольники
// Нормализованный указатель вперёд
                glNormal3f( 0.0f, 0.0f, 1.0f);
                x_m = sector1.triangle[loop_m].vertex[0].x;// X 1-ой точки
                y_m = sector1.triangle[loop_m].vertex[0].y;// Y 1-ой точки
                z_m = sector1.triangle[loop_m].vertex[0].z;// Z 1-ой точки
                // U текстурная координата
                u_m = sector1.triangle[loop_m].vertex[0].u;
                // V текстурная координата
                v_m = sector1.triangle[loop_m].vertex[0].v;
                glTexCoord2f(u_m,v_m); glVertex3f(x_m,y_m,z_m);
                                // Установить TexCoord и грань
            x_m = sector1.triangle[loop_m].vertex[1].x;// X 2-ой точки
            y_m = sector1.triangle[loop_m].vertex[1].y;// Y 2-ой точки
            z_m = sector1.triangle[loop_m].vertex[1].z;// Z 2-ой точки

              // U текстурная координата
                u_m = sector1.triangle[loop_m].vertex[1].u;
    // V текстурная координата
                v_m = sector1.triangle[loop_m].vertex[1].v;
                glTexCoord2f(u_m,v_m); glVertex3f(x_m,y_m,z_m);
                                // Установить TexCoord и грань
                x_m = sector1.triangle[loop_m].vertex[2].x;// X 3-ой точки
                y_m = sector1.triangle[loop_m].vertex[2].y;// Y 3-ой точки
                z_m = sector1.triangle[loop_m].vertex[2].z;// Z 3-ой точки
    // U текстурная координата
u_m = sector1.triangle[loop_m].vertex[2].u;
    // V текстурная координата
                v_m = sector1.triangle[loop_m].vertex[2].v;
                glTexCoord2f(u_m,v_m); glVertex3f(x_m,y_m,z_m);
                                // Установить TexCoord и грань
          glEnd();              // Заканчиваем рисовать треугольники
    }
    return TRUE;            // Возвращаемся
}
```

И вуаля! Мы только что нарисовали наш первый фрейм. Это не совсем Quake, но эй, мы не совсем Carmack или Abrash. Когда вы запустите программу, можете нажать F, B, PgUp и PgDown и увидеть дополнительные эффекты. PgUp/Down просто наклоняет камеру вверх и вниз (наподобие панорамирования из стороны в сторону). Текстура – это моя обработанная школьная фотография J, если конечно NeHe сохранит ее.

Теперь, вы наверное задумываетесь чем заняться дальше. Даже не думайте использовать этот код для полнофункционального 3D движка, так как код не был для этого предназначен. Вы наверное захотите более одного сектора в своей игре, особенно, если вы хотите использовать порталы. Вы также захотите использовать многоугольники с более чем тремя вершинами, опять-таки, особенно для движка с порталами. Моя текущая реализация этого кода позволяет загружать несколько секторов и производит удаление невидимых поверхностей (не рисуются многоугольники, не попадающие в камеру). Я напишу по этому поводу урок очень скоро, но это использует много математики, поэтому я собираюсь для начала написать урок по матрицам.

Урок 11. Эффект "флага" на OpenGL

OpenGL Flag Effect

Всем привет. Для тех, кто хочет узнать, чем же мы тут занимаемся: эту вещь можно увидеть в конце моего демо/хака "Worthless!". Меня зовут Боско (Bosco), и я сделаю все, что в моих силах, чтобы научить вас, парни, делать анимированную картинку с синусоидальной волной по ней. Этот урок основан на уроке №6 от NeHe, и вы должны, по крайней мере, знать и уметь делать то, что в нем описано. Вы должны скачать архив с исходным кодом, распаковать его куда-нибудь, взять картинку из папки data и поместить в подпапку data той папки, где находится ваш исходник :). Ну, или использовать свою текстуру, если она подходящего для OpenGL размера.

Перво-наперво откройте урок №6 для Visual C++ и добавьте этот #include сразу за другими. Данный #include позволит нам использовать различные библиотечные математические функции, как, например, синус и косинус.

```
#include <math.h>                       // для функции Sin()
```

Мы будем использовать массив точек (points) для хранения отдельных x, y и z - координат нашей сетки. Размер сетки 45x45, и она в свою очередь образует 44x44 квадрата. wiggle_count будет использоваться для определения того, насколько быстро "развевается" текстура. Каждые три кадра выглядят достаточно хорошо, и переменная hold будет содержать число с плавающей запятой для сглаживания волн. Эти строки можно добавить в начале программы, где-нибудь под последней строчкой с #include и перед строкой GLuint texture[1]:

```
float points[ 45 ][ 45 ][3]; // массив точек сетки нашей "волны"
int wiggle_count = 0;        // счетчик для контроля быстроты развевания флага
GLfloat hold;                // временно содержит число с плавающей запятой
```

Перейдем вниз, к функции LoadGLTextures(). Мы хотим использовать текстуру с именем Tim.bmp. Найдите LoadBMP("Data/NeHe.bmp") и поменяйте на LoadBMP("Data/Tim.bmp").

```
if (TextureImage[0]=LoadBMP("Data/Tim.bmp"))       // загружаем изображение
```

Теперь добавьте приведенный ниже код в конец функции InitGL() перед return TRUE:

```
glPolygonMode( GL_BACK, GL_FILL );       // нижняя (задняя) сторона заполнена
glPolygonMode( GL_FRONT, GL_LINE );      // верхняя (передняя) сторона прорисована линиями
```

Этот код просто означает, что мы хотим, чтобы полигоны нижней (задней) стороны были зарисованы полностью и чтобы полигоны верхней (передней) стороны были лишь очерчены.

Это мои личные предпочтения. Причина – ориентация полигона или направления граней. Загляните в Красную Книгу (по OpenGL), если хотите узнать больше. Пользуясь случаем, хочу сказать, что эта книга как раз из числа того, что помогает мне в изучении OpenGL, не считая сайта NeHe :). Спасибо NeHe. Купите книгу The Programmer's Guide to OpenGL издательства Addison-Wesley. Она – неисчерпаемый ресурс по части программирования на OpenGL. Ладно, вернемся к уроку.

Прямо под предыдущим кодом и выше return TRUE добавьте следующие строки:

```
// пройдемся по оси X
    for(int x=0; x<45; x++)
    {
        // пройдемся по оси Y
        for(int y=0; y<45; y++)
        {
            // применим волну к нашей сетке
            points[x][y][0]=float((x/5.0f)-4.5f);
            points[x][y][1]=float((y/5.0f)-4.5f);
            points[x][y][2]=float(sin((((x/5.0f)*40.0f)/360.0f)*3.141592654*2.0f));
        }
    }
```

Благодарю Грэма Гиббонса (Graham Gibbons) за замечание об целочисленном цикле для того чтобы избежать пиков на волне.
Два цикла вверху инициализируют точки на нашей сетке. Я инициализирую переменные в моем цикле, чтобы помнить, что они просто переменные цикла. Не уверен, что это хорошо. Мы используем целочисленные циклы, дабы предупредить глюки, которые появляются при вычислениях с плавающей запятой. Мы делим переменные x и y на 5

(т.е. 45 / 9 = 5) и отнимаем 4.5 от каждой из них, чтобы отцентрировать "волну". Того же эффекта можно достигнуть переносом (translate), но этот способ мне больше нравится.
Окончательное значение points[x][y][2] - это наше значение синуса. Функция sin() принимает параметр в радианах. Мы берем наши градусы, которые получились умножением float_x (x/5.0f) на 40.0f, и, чтобы конвертировать их в радианы делением, мы берем градусы, делим на 360.0f, умножаем на число пи (или на аппроксимацию) и на 2.0f.

Я заново перепишу функцию DrawGLScene, так что удалите ее текст и замените его текстом, приведенным ниже.

```
int DrawGLScene(GLvoid)                         // рисуем нашу сцену
{
    int x, y;                               // переменные циклов
    // для разбиения флага на маленькие квадраты
    float float_x, float_y, float_xb, float_yb;
```

Различные переменные используются для контроля в циклах. Посмотрите на код ниже – большинство из них служит лишь для контролирования циклов и хранения временных значений.

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // очистить экран и буфер глубины
glLoadIdentity();                               // сброс текущей матрицы
glTranslatef(0.0f,0.0f,-12.0f);                 // перенести 17 единиц в глубь экрана
glRotatef(xrot,1.0f,0.0f,0.0f);                 // вращение по оси X
glRotatef(yrot,0.0f,1.0f,0.0f);                 // вращение по оси Y
glRotatef(zrot,0.0f,0.0f,1.0f);                 // вращение по оси Z
glBindTexture(GL_TEXTURE_2D, texture[0]);           // выбрать нашу текстуру
```

Вы могли видеть это все раньше. Все тут так же, как в уроке №6, кроме того, что я просто отвожу сцену от камеры чуть дальше.

```
glBegin(GL_QUADS);              // начинаем рисовать квадраты
    for( x = 0; x < 44; x++ ) // пройдемся по оси X 0-44 (45 точек)
    {
        for( y = 0; y < 44; y++ ) // пройдемся по оси Y 0-44 (45 точек)
        {
```

Просто запускаем цикл рисования наших полигонов. Я здесь использую целые числа, чтобы избежать использования функции int(), которой я пользовался ранее для получения индекса ссылки массива как целое значение.

```
float_x = float(x)/44.0f;         // создать значение X как float
float_y = float(y)/44.0f;         // создать значение Y как float
float_xb = float(x+1)/44.0f;      // создать значение X как float плюс 0.0227f
float_yb = float(y+1)/44.0f;      // создать значение Y как float плюс 0.0227f
```

Мы используем четыре переменных выше для координат текстуры. Каждый наш полигон (квадрат в сетке) имеет секцию размером 1/44 x 1/44 с отображенной на нее текстурой. В начале циклов задается нижняя левая вершина секции, и потом мы просто добавляем к этой вершине соответственно 0 или 1 для получения трех остальных вершин (т.е. x+1 или y+1 будет правая верхняя вершина).

```
// первая координата текстуры (нижняя левая)
        glTexCoord2f( float_x, float_y);
        glVertex3f( points[x][y][0], points[x][y][1], points[x][y][2] );

    // вторая координата текстуры (верхняя левая)
        glTexCoord2f( float_x, float_yb );
        glVertex3f( points[x][y+1][0], points[x][y+1][1], points[x][y+1][2]);
// третья координата текстуры (верхняя правая)
        glTexCoord2f( float_xb, float_yb );
glVertex3f( points[x+1][y+1][0], points[x+1][y+1][1], points[x+1][y+1][2]);
         // четвертая координата текстуры (нижняя правая)
        glTexCoord2f( float_xb, float_y );
        glVertex3f( points[x+1][y][0], points[x+1][y][1], points[x+1][y][2]);
    }
}
glEnd();                            // закончили с квадратами
```

Строки выше просто делают OpenGL-вызовы для передачи всех данных, о которых мы говорили. Четыре отдельных вызова каждой функции glTexCoord2f() и glVertex3f(). Продолжим. Заметьте – квадраты рисуются по часовой стрелке. Это означает, что сторона, которую вы видите, вначале будет задней. Задняя заполнена. Передняя состоит из линий.

Если бы вы рисовали против часовой стрелки, то сторона, которую вы бы видели вначале, была бы передней. Значит, вы увидели бы текстуру, выглядящую как сетка, (ты написал: типа сетки) вместо заполненной текстурой поверхности.

```
  if( wiggle_count == 2 )      // для замедления волны (только каждый второй кадр)
      {
Теперь мы хотим повторять значения синуса, тем самым создавая "движение".
      for( y = 0; y < 45; y++ )          // пройдемся по оси Y
      {
// сохраним текущее значение одной точки левой стороны волны
            hold=points[0][y][2];
            for( x = 0; x < 44; x++)      // пройдемся по оси X
            {
                // текущее значение волны равно значению справа
                points[x][y][2] = points[x+1][y][2];
            }
      // последнее значение берется из дальнего левого сохраненного значения
            points[44][y][2]=hold;
      }
      wiggle_count = 0;                  // снова сбросить счетчик
}
wiggle_count++;                          // увеличить счетчик
```

Вот что мы делаем: сохраняем первое значение каждой линии, затем двигаем волну к левому, заставляя текстуру развеваться. Значение, которое мы сохранили, потом вставляется в конец, чтобы создать бесконечную волну, идущую по поверхности текстуры. Затем мы обнуляем счетчик wiggle_count для продолжения анимации.

Код вверху был изменен NeHe (в феврале 2000) для исправления недостатка волны. Волна теперь гладкая.

```
xrot+=0.3f;            // увеличить значение переменной вращения по X
yrot+=0.2f;            // увеличить значение переменной вращения по Y
zrot+=0.4f;            // увеличить значение переменной вращения по Z
return TRUE;             // возврат из функции
}
```

Обычные значения поворота у NeHe. :) Вот и все. Скомпилируйте, и у вас должна появиться миленькая вращающаяся "волна". Я не знаю, что еще сказать, фу-у-ух… Это было так ДОЛГО! Но я надеюсь, вы, парни, сможете последовать этому, либо приобрести что-то полезное для себя. Если у вас есть вопросы, если вы хотите, чтобы я что-нибудь подправил или сказать мне, как, все-таки, жутко я пишу программы :), то пришлите мне письмецо.

Это был экспромт, но он очень сэкономил время и силы. Он заставил меня ценить теперь взгляды NeHe гораздо больше. Спасибо всем.

© Bosco (bosco4@home.com)

Урок 12. Использование списков отображения

Display Lists

В этом уроке я научу вас, как использовать Списки Отображения (Display Lists). Использование списков отображения не только ускоряет ваш код, списки также позволяют уменьшить количество строк, которые необходимо написать для создания простой GL сцены.

Например. Скажем, вы создаете в игре астероиды. Каждый уровень начинается как минимум с двух астероидов. Итак, вы сидите с вашей графической бумагой (:))) и рисуете, как можно создать 3D астероиды. После того, как вы все нарисовали, вы создаете астероид в OpenGL, используя полигоны или четырехугольники. Скажем астероид - это октагон (восемь сторон). Если вы умны, вы создаете цикл, и рисуете астероид один раз в цикле. Вы сделаете это,

написав 18 или более небрежных строк кода, чтобы создать астероид. Создание астероида, перерисовка его на экран трудоемка для вашей системы. Когда вы возьмете более сложные объекты вы увидите, что я имел ввиду.

Ну и какое решение? Списки Отображения!!! Используя список отображения, вы создаете объект лишь один раз. Вы можете текстурировать его, раскрасить его, все что хотите. Вы даете списку название. Поскольку это астероид мы назовем список 'asteroid'. Теперь, всякий раз, когда я хочу нарисовать текстурированный/раскрашенный астероид на экране, все, что мне требуется сделать это вызов glCallList(asteroid). Предварительно созданный астероид немедленно появится на экране. Так как астероид уже создан в списке отображения, не нужно указывать OpenGL, как создать его. Он уже создан в памяти. Это снижает нагрузку на процессор и позволяет вашим программам выполняться быстрее.

Итак, вы готовы учиться? :). Мы назовем эту демонстрацию списков отображения Q- Bert. В этой демке мы создадим 15 кубиков на экране Q-Bert'a. Каждый кубик изготавливается из крышки (TOP) и коробки (BOX). Крышка будет отдельным списком отображения, так мы сможем закрасить ее в более темные цвета (как тень). Коробка - это куб без крышки :).

Этот код основан на уроке 6. Я перепишу большую часть программы - так будет легче видеть, где я внес изменения. Следующие строки - код, стандартно используемый во всех уроках.

```
#include <windows.h>      //Заголовочный файл для Windows
#include <stdio.h>        //Заголовочный файл стандартного ввода/вывода
#include <gl\gl.h>        //Заголовочный файл библиотеки OpenGL32
#include <gl\glu.h>       //Заголовочный файл библиотеки GLu32
#include <gl\glaux.h>     //Заголовочный файл библиотеки GLaux

HDC           hDC=NULL;       //Приватный контекст GDI устройства
HGLRC         hRC=NULL;       //Постоянный контекст отображения
HWND          hWnd=NULL;      //Содержит дискриптор нашего окна
HINSTANCE     hInstance;      //Содержит экземпляр приложения

bool    keys[256];            //Массив применяемый для подпрограммы клавиатуры
bool    active=TRUE;          //Флаг "Активное" окна устанавливается истинным (TRUE)
                              // по умолчанию.
bool    fullscreen=TRUE;      //Флаг "На полный экран" устанавливается в полноэкранный
                              // режим по умолчанию.
```

Теперь объявим наши переменные. Первым делом зарезервируем место для одной текстуры. Затем создадим две новых переменных для наших двух списков отображения. Эти переменные - указатели на списки отображения, хранящиеся в ОЗУ. Назовем их коробка (box) и крышка (top).

Затем заведем еще 2 переменные xloop и yloop, которые используем для задания позиций кубов на экране и 2 переменные xrot и yrot для вращения по осям x и y.

```
GLuint  texture[1];     // Память для одной текстуры
GLuint  box;            // Память для списка отображения box (коробка)
GLuint  top;            // Память для второго списка отображения top (крышка)
GLuint  xloop;  // Цикл для оси x
GLuint  yloop;  // Цикл для оси y
GLfloat xrot;   // Вращает куб на оси x
GLfloat yrot;   // Вращает куб на оси y
```

Далее создаем два цветовых массива. Первый, назовем его boxcol, хранит величины для следующих цветов: ярко-красного (Bright Red), оранжевого (Orange), желтого (Yellow), зеленого (Green) и голубого (Blue). Каждое значение внутри фигурных скобок представляет значения красного, зеленого и голубого цветов. Каждая группа внутри фигурных скобок - конкретный цвет.

Второй массив цветов мы создаем для следующих цветов: темно-красного (Dark Red), темно-оранжевого (Dark Orange), темно-желтого (Dark Yellow), темно-зеленого (Dark Green) и темно-голубого (Dark Blue). Темные цвета будем использовать для окрашивания крышки коробок. Мы хотим, чтобы крышка была темнее коробки.

```
static GLfloat boxcol[5][3]=    //Массив для цветов коробки
{     //Яркие: Красный, Оранжевый, Желтый, Зеленый, Голубой
{1.0f,0.0f,0.0f},{1.0f,0.5f,0.0f},{1.0f,1.0f,0.0f},{0.0f,1.0f,0.0f},{0.0f,1.0f,1.0f}
};
```

```
static GLfloat topcol[5][3]=    //Массив для цветов верха
{
//Темные: Красный, Оранжевый, Желтый, Зеленый, Голубой
{.5f,0.0f,0.0f},{0.5f,0.25f,0.0f},{0.5f,0.5f,0.0f},{0.0f,0.5f,0.0f},{0.0f,0.5f,0.5f}
};

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM); //Объявление для WndProc
```

Теперь мы создадим собственно список отображения. Если вы помните, весь код для создания коробки в первом списке, а для создания верха - в другом списке. Я попытаюсь разъяснить эту часть более детально.

```
GLvoid BuildLists()     //создаем список отображения
{
```

Мы начинаем, указывая OpenGL создать 2 списка. glGenLists(2) создает место для двух списков, и возвращает указатель на первый из них. 'box' будет хранить расположение первого списка. Всякий раз при вызове box будет нарисован первый список.

```
box=glGenLists(2);      //создаем два списка
```

Теперь мы начнем создавать первый список. Мы уже выделили память для двух списков, и мы знаем, что box указывает на место, где мы храним первый список. Все, что нам теперь надо сделать - это сказать OpenGL, где должен начаться список, какой тип списка сделать.

Мы используем для этого команду glNewList(). Укажите box в качестве первого параметра. Этим вы укажете OpenGL место хранения списка, на которое указывает переменная box. Второй параметр GL_COMPILE говорит OpenGL, что мы хотим предварительно создать список в памяти, таким образом, OpenGL не будет вычислять, как создать объект, всякий раз как мы хотим нарисовать его.

С флагом GL_COMPILE как в программирование. Если вы пишете программу и транслируете ее компилятором вы компилируете ее всякий раз, когда хотите запустить ее. Если же она уже скомпилирована в EXE файл, все, что вам нужно сделать - это запустить ее, кликнув на файле. Нет необходимости перекомпилировать ее заново. OpenGL компилирует список отображения только один раз, после этого он готов к применению, больше компилировать не надо. Поэтому мы получаем увеличение скорости при использовании списков отображения.

```
glNewList(box,GL_COMPILE);      // Новый откомпилированный список отображения box
```

В следующей части кода нарисуем коробку без верха. Она не окажется на экране. Она будет сохранена в списке отображения.

Вы можете написать любую команду, какую только захотите, между glNewList() и glEndList(). Вы можете задавать цвета, менять текстуры и т.д. Единственный тип кода, который вы не можете использовать - это код, который изменит список отображения "на лету". Однажды создав список отображения, вы не можете его изменить.

Если вы добавите строку glColor3ub(rand()%255,rand()%255,rand()%255) в нижеследующий код, вы можете подумать, что каждый раз, когда вы рисуете объект на экране, он будет другого цвета. Но так как список создается лишь однажды, цвет не будет изменяться всякий раз, как вы рисуете его на экране. Каким бы ни был цвет объекта, когда он создается первый раз, таким он и останется.

Если вы хотите изменить цвет списка отображения, вам необходимо изменить его ДО того как список отображения будет выведен на экран. Я больше расскажу об этом позже.

```
glBegin(GL_QUADS);      // Начинаем рисование четырехугольников (quads)
                        // Нижняя поверхность
glTexCoord2f(1.0f, 1.0f);
glVertex3f(-1.0f, -1.0f, -1.0f);        // Верхний правый угол текстуры и четырехугольник
glTexCoord2f(0.0f, 1.0f);
glVertex3f( 1.0f, -1.0f, -1.0f);        // Верхний левый угол текстуры и четырехугольник
glTexCoord2f(0.0f, 0.0f);
glVertex3f( 1.0f, -1.0f, 1.0f);         // Нижний левый угол текстуры и четырехугольник
glTexCoord2f(1.0f, 0.0f);
glVertex3f(-1.0f, -1.0f, 1.0f);         // Нижний правый угол текстуры и четырехугольник
                        // Передняя поверхность
```

```
glTexCoord2f(0.0f, 0.0f);
glVertex3f(-1.0f, -1.0f, 1.0f);        // Нижний левый угол текстуры и четырехугольник
glTexCoord2f(1.0f, 0.0f);
glVertex3f( 1.0f, -1.0f, 1.0f);        // Нижний правый угол текстуры и четырехугольник
glTexCoord2f(1.0f, 1.0f);
glVertex3f( 1.0f, 1.0f, 1.0f);        // Верхний правый угол текстуры и четырехугольник
glTexCoord2f(0.0f, 1.0f);
glVertex3f(-1.0f, 1.0f, 1.0f);        // Верхний левый угол текстуры и четырехугольник
                    // Задняя поверхность
glTexCoord2f(1.0f, 0.0f);
glVertex3f(-1.0f, -1.0f, -1.0f);        // Нижний правый угол текстуры и четырехугольник
glTexCoord2f(1.0f, 1.0f);
glVertex3f(-1.0f, 1.0f, -1.0f);        // Верхний правый угол текстуры и четырехугольник
glTexCoord2f(0.0f, 1.0f);
glVertex3f( 1.0f, 1.0f, -1.0f);        // Верхний левый угол текстуры и четырехугольник
glTexCoord2f(0.0f, 0.0f);
glVertex3f( 1.0f, -1.0f, -1.0f);        // Нижний левый угол текстуры и четырехугольник
                    // Правая поверхность
glTexCoord2f(1.0f, 0.0f);
glVertex3f( 1.0f, -1.0f, -1.0f);        // Нижний правый угол текстуры и четырехугольник
glTexCoord2f(1.0f, 1.0f);
glVertex3f( 1.0f, 1.0f, -1.0f);        // Верхний правый угол текстуры и четырехугольник
glTexCoord2f(0.0f, 1.0f);
glVertex3f( 1.0f, 1.0f, 1.0f);        // Верхний левый угол текстуры и четырехугольник
glTexCoord2f(0.0f, 0.0f);
glVertex3f( 1.0f, -1.0f, 1.0f);        // Нижний левый угол текстуры и четырехугольник
                    // Левая поверхность
glTexCoord2f(0.0f, 0.0f);
glVertex3f(-1.0f, -1.0f, -1.0f);        // Нижний левый угол текстуры и четырехугольник
glTexCoord2f(1.0f, 0.0f);
glVertex3f(-1.0f, -1.0f, 1.0f);        // Нижний правый угол текстуры и четырехугольник
glTexCoord2f(1.0f, 1.0f);
glVertex3f(-1.0f, 1.0f, 1.0f);        // Верхний правый угол текстуры и четырехугольник
glTexCoord2f(0.0f, 1.0f);
glVertex3f(-1.0f, 1.0f, -1.0f);        // Верхний левый угол текстуры и четырехугольник
glEnd();                    // Закончили рисование четырехугольников
```

При помощи команды glEndList() мы сказали OpenGL, что мы создали список. Все, что находится между glNewList() и glEndList - часть списка отображения, все, что находится до glNewList() или после glEndList() не является частью списка отображения.

```
glEndList(); // Закончили создание списка box
```

Теперь создадим наш второй список отображения. Чтобы найти, где список отображения хранится в памяти, мы возьмем значение старого списка отображения (box) и добавим к нему еще единицу. Нижеследующий код присваивает переменной 'top' местоположение второго списка отображения.

```
top=box+1; // Значение top это значение box + 1
```

Теперь, когда мы знаем, где хранится второй список отображения, мы можем создать его. Мы сделаем это, так же как и первый список, но в этот раз мы укажем OpenGL хранить этот список под названием 'top' в отличие от предыдущего 'box'.

```
glNewList(top,GL_COMPILE);// Новый откомпилированный список отображения 'top'
```

Следующая часть кода рисует верх коробки. Это просто четырехугольник нарисованный на плоскости z.

```
glBegin(GL_QUADS);      // Начинаем рисование четырехугольника
                    // Верхняя поверхность
glTexCoord2f(0.0f, 1.0f);
glVertex3f(-1.0f, 1.0f, -1.0f);        // Верхний левый угол текстуры и четырехугольник
glTexCoord2f(0.0f, 0.0f);
glVertex3f(-1.0f, 1.0f, 1.0f);        // Нижний левый угол текстуры и четырехугольник
```

```
glTexCoord2f(1.0f, 0.0f);
glVertex3f( 1.0f, 1.0f, 1.0f);        // Нижний правый угол текстуры и четырехугольник
glTexCoord2f(1.0f, 1.0f);
glVertex3f( 1.0f, 1.0f, -1.0f);       // Верхний правый угол текстуры и четырехугольник
glEnd();              // Заканчиваем рисование четырехугольника
```

Снова говорим OpenGL, что мы закончили создание списка отображения с помощью команды glEndList(). Вот. Мы успешно создали два списка отображения.

```
    glEndList();          // Закончили создание списка отображения 'top'
}
```

Код создания текстур тот же, который мы использовали в предыдущих уроках для загрузки и создания текстур. Мы хотим получить текстуру, которую сможем наложить на все шесть сторон каждого куба. Я, бесспорно, буду использовать мипмаппинг, чтобы сделать действительно сглаженные текстуры. Я ненавижу глядеть на пиксели :). Текстура для загрузки называется 'cube.bmp'. Она хранится в каталоге под названием 'data'. Найдите LoadBMP и измените эту строку таким образом, чтобы она выглядела как нижеследующая строка.

```
if (TextureImage[0]=LoadBMP("Data/Cube.bmp"))          // Загрузить картинку.
```

Код изменения размеров такой же, как в Уроке N6.

Только в коде инициализации есть несколько изменений. Я добавлю строку BuildList(). Это будет переход к части кода, в которой создаются списки отображения. Обратите внимание что BuildList() находится после LoagGLTextures(). Важно знать, что порядок должен быть именно таким. Сначала мы создаем текстуру, а затем создаем наши списки отображения, где уже созданные текстуру мы можем наложить на куб.

```
int InitGL(GLvoid)// Все настройки OpenGL начинаются здесь
{
    if (!LoadGLTextures())// Переход к процедуре загрузки текстуры
    {
        return FALSE;// Если текстура не загружена возращает FALSE
    }
    BuildLists();// Переход к коду, который создает наши списки отображения
    glEnable(GL_TEXTURE_2D);// Включение нанесения текстур
    glShadeModel(GL_SMOOTH);// Включение гладкой закраски (smooth shading)
    glClearColor(0.0f, 0.0f, 0.0f, 0.5f);// Черный фон
    glClearDepth(1.0f);// Установка буфера глубины
    glEnable(GL_DEPTH_TEST);// Включение проверки глубины
    glDepthFunc(GL_LEQUAL); // Тип выполняемой проверки глубины
```

Следующие три строки кода включают быстрое и простое освещение. Light0 предустановлен на большинстве видеоплат, и это спасет нас от трудностей установки источников освещения. После того как мы включили light0, мы включили освещение. Если light0 не работает на вашей видео плате (вы видите темноту), просто отключите освещение.

Еще одна строка GL_COLOR_MATERIAL позволяет нам добавить цвет к текстурам. Если мы, не включили закрашивание материала, текстуры всегда будут своего первоначального цвета. glColor3f(r,g,b) не будет действовать на расцвечивание. Поэтому важно включить это.

```
glEnable(GL_LIGHT0); // Быстрое простое освещение
                     // (устанавливает в качестве источника освещения Light0)
glEnable(GL_LIGHTING);  // Включает освещение
glEnable(GL_COLOR_MATERIAL);    // Включает раскрашивание материала
```

Наконец, мы устанавливаем коррекцию перспективы для улучшения внешнего вида и возвращаем TRUE, давая знать нашей программе, что инициализация прошла OK.

```
glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);      // Изящная коррекция перспективы
return TRUE;                                    // Инициализация прошла OK
}
```

Теперь часть программы для рисования. Как обычно, я немного шизею с математики. Нет SIN и COS, но это все еще непривычно :). Мы начнем, как обычно, с очистки экрана и глубины буфера.

Затем мы привяжем текстуру к кубу. Я мог добавить эту строку в коде списка отображения, но, оставляя ее вне списка, я могу изменить текстуру, когда захочу. Если бы я добавил строку glBindTexture(GL_TEXTURE_2D, texture[0]) внутри кода списка отображения, список был бы создан с этой текстурой постоянно нанесенной на него.

```
int DrawGLScene(GLvoid)                 // Здесь мы выполняем все рисование
{
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);     // Очищаем экран и буфер глубины

glBindTexture(GL_TEXTURE_2D, texture[0]);               // Выбираем текстуру
```

Теперь немного позабавимся. Мы имеем цикл по yloop. Этот цикл используется для позиционирования кубов по оси Y (вверх и вниз). Мы хотим иметь 5 строк кубиков вверх и вниз, итак мы создадим цикл от 1 до 6 (то есть 5).

```
    for (yloop=1;yloop<6;yloop++)                       // Цикл по оси Y
    {
```

Создадим другой цикл по xloop. Он используется для позиционирования кубиков по оси x (слева направо). Количество кубов зависит от того, в какой они рисуются строке. В верхней строчке xloop будет изменяться только от 0 до 1 (рисование одного кубика). В следующей строке xloop изменяется от 0 до 2 (рисование 2-х кубиков) и так далее.

```
        for (xloop=0;xloop< yloop;xloop++)              // Цикл по оси X
        {
```

Мы очищаем наше окно с помощью glLoadIdentity().

```
            glLoadIdentity();                   // Очистка вида
```

Следующая строка задает особую точку на экране. Это может сбить с толку, но не пугайтесь. На оси X происходит следующее:

Мы смещаемся вправо на 1.4 единицы так, чтобы центр пирамиды оказался в центре экрана. Умножим xloop на 2.8 и добавим к ней 1.4 (мы умножаем на 2.8, так как кубики стоят друг на друге, 2.8 это приближенная ширина куба, когда он повернут на 45 градусов, по диагонали куба). Наконец, вычитаем yloop*1.4. Это перемещение кубов влево в зависимости от того, в какой строке он находится. Если мы не сместим влево, пирамида начнется у левого края экрана (не будет выглядеть как пирамида).

На оси Y мы вычитаем yloop из 6, иначе пирамида будет нарисована сверху вниз. Результат умножим на 2.4. Иначе кубики будут находиться друг на друге. (2.4 примерная высота куба). Затем мы отнимем 7, чтобы пирамида начиналась от низа экрана и строилась вверх.

Наконец по оси Z мы переместимся на 20 единиц в глубь экрана. Так мы добьемся того, что бы пирамида хорошо вписалась в экран.

```
            // Размещение кубиков на экране
glTranslatef(1.4f+(float(xloop)*2.8f)-(float(yloop)*1.4f),
    ((6.0f-float(yloop))*2.4f)-7.0f,-20.0f);
```

Теперь мы повернемся вокруг оси X. Таким образом, мы наклоним кубики вперед на 45 градусов минус 2 умноженное на yloop. Режим перспективы наклоняет кубики автоматически, и я вычитаю, чтобы скомпенсировать наклон. Не лучший способ, но это работает.

Наконец добавим xrot. Это дает нам управлять углом с клавиатуры. (Получите удовольствие, играя с ним).

После этого повернем кубики на 45 градусов вокруг оси Y, и добавим yrot чтобы управлять с клавиатуры поворотом вокруг оси Y.

```
glRotatef(45.0f-(2.0f*yloop)+xrot,1.0f,0.0f,0.0f);      // Наклонять кубы вверх и вниз
glRotatef(45.0f+yrot,0.0f,1.0f,0.0f);                   // Вращать кубы вправо и влево
```

Далее мы выберем цвет коробки (яркий) перед тем как действительно нарисовать эту часть куба. Мы используем команду glColor3fv(), которая выполняет загрузку всех трех значений цвета (красный, зеленый, голубой). Аббревиатура 3fv означает 3 значения, числа с плавающей точкой, v указатель на массив. Цвет мы выбираем по переменной yloop-1, что дает нам разные цвета для каждого ряда кубов. Если мы используем xloop-1, мы получим различные цвета для каждого столбца.

```
glColor3fv(boxcol[yloop-1]);            // Выбор цвета коробки
```

Теперь, когда цвет установлен, все что надо сделать - нарисовать нашу коробку. В отличие от написания всего кода для рисования коробки, все, что нам нужно - вызвать наш список отображения. Мы сделаем это командой glCallList(box). box указывает OpenGL выбрать список отображения box. Список отображения box - это кубик без верха.

Коробка будет нарисована цветом, выбранным командой glColor3fv(), в месте, куда мы ее передвинули.

```
glCallList(box);                        // Рисуем коробку
```

Теперь выберем цвет крышки (темный) перед тем как нарисовать крышку коробки. Если вы действительно хотите сделать Q-Bert, вы измените этот цвет всякий раз, когда Q-Bert вскакивает на коробку. Цвет устанавливается в зависимости от ряда.

```
glColor3fv(topcol[yloop-1]);            // Выбор цвета верха
```

Наконец, осталось нарисовать список отображения top. Этим мы добавим более темного цвета крышку. Все. Очень легко!

```
                glCallList(top);                    // Рисуем крышку
            }
        }
        return TRUE;                                // Возвращаемся обратно.

}
```

Все необходимые изменения будут выполнены в WinMain(). Код будет добавлен сразу после строки SwapBuffers(hDC). Этим кодом мы проверим нажатие кнопок "стрелка влево", "вправо", "вверх", "вниз" и соответственно повернем кубы.

```
        SwapBuffers(hDC);       // Поменяем буферы (Двойная буферизация)
        if (keys[VK_LEFT])      // Была нажата стрелка влево?
        {
            yrot-=0.2f;     // Если так, то повернем кубы влево
        }
        if (keys[VK_RIGHT])     // Была нажата стрелка вправо?
        {
            yrot+=0.2f;     // Если так, то повернем кубы вправо
        }
        if (keys[VK_UP])        // Была нажата стрелка вверх?
        {
            xrot-=0.2f;     // Если так, то наклоним кубы вверх
        }
        if (keys[VK_DOWN])      // Была нажата стрелка вниз?
        {
            xrot+=0.2f;     // Если так, то наклоним кубы вниз
        }
```

Как во всех предыдущих уроках, убедимся, что заголовок на верху окна правильный.

```
        if (keys[VK_F1])                // Была нажата кнопка F1?
        {
            keys[VK_F1]=FALSE;// Если так - установим значение FALSE
            KillGLWindow(); // Закроем текущее окно OpenGL
            // Переключим режим "Полный экран"/"Оконный"
            fullscreen=!fullscreen;
```

```
                // Заново создадим наше окно OpenGL
                if (!CreateGLWindow("NeHe's Display List Tutorial",
                    640,480,16,fullscreen))
                {
                    return 0;// Выйти, если окно не было создано
                }
            }
        }
}
```

В результате этого урока вы усвоили, как работает список отображения, как его создавать, и как выводить его на экран. Списки отображения восхитительны. Не только потому, что они упрощают программирование сложных проектов, они также дают небольшое увеличение необходимое для поддержания высокой частоты обновления экрана.

Я надеюсь, вам понравился этот урок. Если у вас есть вопросы или вы чувствуете, что что-нибудь не ясно, пожалуйста, напишите мне и дайте мне знать.

Урок 13. Растровые шрифты

Bitmap Fonts

Добро пожаловать еще на один урок. На сей раз, я буду учить Вас, как использовать растровые шрифты. Вы можете сказать себе "разве так трудно вывести текст на экран". Если Вы когда-либо пробовали это, то вы знаете, что это не просто!
Уверен, Вы сможете запустить графический редактор, написать текст на изображении, загрузить изображение в вашу программу OpenGL, включить смешивание, затем отобразить текст на экран. Но это съедает время, конечный результат обычно выглядит расплывчатым или грубым в зависимости от типа фильтрации, который Вы использовали, и если ваше изображение имеет альфа канал, ваш текст будет в довершении прозрачным (смешанный с объектами на экране) после наложения на экран.
Если Вы когда-либо использовали Wordpad, Microsoft Word или другой текстовый процессор, Вы, возможно, заметили, сколько разных типов доступных шрифтов. В этом уроке Вы научитесь, как использовать те же самые шрифты в ваших собственных программах OpenGL. Фактически, любой шрифт, который Вы установлен на вашем компьютере, может использоваться в ваших примерах.
Растровые шрифты не только в 100 раз лучше выглядят, чем графические шрифты (текстуры). Вы можете изменять текст на лету. Не нужно делать текстуры для каждого слова или символа, которые Вы хочет вывести на экран. Только позиционируйте текст, и используйте мою удобную новую gl команду, чтобы отобразить текст на экране.
Я попробовал сделать команду настолько простой насколько это возможно. Все, что Вы должны сделать, так это, набрать glPrint ("Привет"). Это легко. Как бы то ни было, Вы можете сказать, что здесь довольно длинное введение, которое я счастлив, дать Вам в этом уроке. Мне потребуется около полутора часов, чтобы создать программу. Почему так долго? Поскольку нет почти никакой информации, доступной по использованию растровых шрифтов, если конечно Вы не наслаждаетесь кодом MFC. Чтобы сохранить по возможности полученный код как можно проще, я решил, что будет хорошо, если я напишу все это в простом для понимания коде Си :).

Небольшое примечание, этот код применим только в Windows. Он использует функции wgl Windows, для построения шрифтов. Очевидно, Apple имеет функции agl, которые должны делать то же самое, а X имеет glx. К сожалению, я не могу гарантировать, что этот код переносим. Если кто-нибудь имеет платформо-незавизимый код для вывода шрифтов на экран, пришлите мне его, и я напишу другой урок по шрифтам.

Мы начнем с такого же кода как в уроке 1. Мы будем добавлять заголовочный файл stdio.h для стандартных операций ввода/вывода; stdarg.h для разбора текста и конвертирования переменных в текст, и, наконец math.h, для того чтобы перемещать текст по экрану, используя SIN и COS.

```
#include <windows.h>  // Заголовочный файл для Windows
#include <stdio.h>    // Заголовочный файл для стандартной библиотеки ввода/вывода
#include <gl\gl.h>    // Заголовочный файл для библиотеки OpenGL32
#include <gl\glu.h>   // Заголовочный файл для библиотеки GLu32
#include <gl\glaux.h> // Заголовочный файл для библиотеки GLaux
#include <math.h>     // Заголовочный файл для математической библиотеки ( НОВОЕ )
#include <stdarg.h>   // Заголовочный файл для функций для работы с переменным
                      //  количеством аргументов ( НОВОЕ )
```

```
HDC       hDC=NULL;  // Приватный контекст устройства GDI
HGLRC     hRC=NULL;  // Постоянный контекст рендеринга
HWND      hWnd=NULL; // Сохраняет дескриптор окна
HINSTANCE  hInstance; // Сохраняет экземпляр приложения
```

Мы также собираемся добавить 3 новых переменных. В base будет сохранен номер первого списка отображения, который мы создаем. Каждому символу требуется собственный список отображения. Символ 'A' - 65 список отображения, 'B' - 66, 'C' - 67, и т.д. Поэтому 'A' будет сохранен в списке отображения base+65.

Затем мы добавляем два счетчика (cnt1 и cnt2). Эти счетчики будут изменяться с разной частотой, и используются для перемещения текста по экрану, используя SIN и COS. Это будет создавать эффект хаотичного движения строки текста по экрану. Мы будем также использовать эти счетчики, чтобы изменять цвет символов (но об этом чуть позже).

```
GLuint  base;      // База списка отображения для фонта
GLfloat  cnt1;     // Первый счетчик для передвижения и закрашивания текста
GLfloat  cnt2;     // Второй счетчик для передвижения и закрашивания текста

bool  keys[256];      // Массив для работы с клавиатурой
bool  active=TRUE;    // Флаг активации окна, по умолчанию = TRUE
bool  fullscreen=TRUE;// Флаг полноэкранного режима

LRESULT  CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);  // Объявление WndProc
```

В следующей секции кода происходит построение шрифта. Это наиболее трудная часть кода. Объявление 'HFONT font' задает шрифт в Windows.

Затем мы определяем base. Мы создаем группу из 96 списков отображения, используя glGenLists(96). После того, как списки отображения созданы, переменная base будет содержать номер первого списка.

```
GLvoid BuildFont(GLvoid)  // Построение нашего растрового шрифта
{
  HFONT  font;            // Идентификатор фонта
 base = glGenLists(96);  // Выделим место для 96 символов ( НОВОЕ )
```

Теперь позабавимся. Мы собираемся создать наш шрифт. Мы начинаем, задавая размер шрифта. Вы заметили, что это отрицательное число. Вставляя минус, мы сообщаем Windows, что надо найти нам шрифт, основанный на высоте СИМВОЛОВ. Если мы используем положительное число, мы выбираем шрифт, основанный на высоте ЯЧЕЙКИ.

```
font = CreateFont(  -24,        // Высота фонта ( НОВОЕ )
```

Затем мы определяем ширину ячейки. Вы увидите, что я установил ее в 0. При помощи установки значения в 0, Windows будет использовать значение по умолчанию. Вы можете поиграть с этим значением, если Вы хотите. Сделайте шрифт широким, и т.д.

```
        0,        // Ширина фонта
```

Угол отношения (Angle of Escapement) позволяет вращать шрифт. К сожалению, это - не очень полезная возможность. Исключая 0, 90, 180, и 270 градусов, у шрифта будет обрезаться то, что не попало внутрь невидимой квадратной границы. Угол наклона (Orientation Angle), цитируя справку MSDN, определяет угол, в десятых долях градуса, между базовой линией символа и осью X устройства. К сожалению, я не имею понятия о том, что это означает :(.

```
        0,        // Угол отношения
        0,        // Угол наклона
```

Ширина шрифта – отличный параметр. Вы можете использовать числа от 0 - 1000, или Вы можете использовать одно из предопределенных значений. FW_DONTCARE - 0, FW_NORMAL - 400, FW_BOLD - 700, и FW_BLACK - 900. Есть множество других предопределенные значений, но и эти 4 дают хорошее разнообразие. Чем выше значение, тем более толстый шрифт (более жирный).

```
        FW_BOLD,      // Ширина шрифта
```

Курсив, подчеркивание и перечеркивание может быть или TRUE или FALSE. Если подчеркивание TRUE, шрифт будет подчеркнут. Если FALSE то, нет. Довольно просто :).

```
    FALSE,      // Курсив
    FALSE,      // Подчеркивание
    FALSE,      // Перечеркивание
```

Идентификатор набора символов описывает тип набора символов, который Вы хотите использовать. Есть множество типов, и обо всех их не рассказать в этом уроке. CHINESEBIG5_CHARSET, GREEK_CHARSET, RUSSIAN_CHARSET, DEFAULT_CHARSET, и т.д. ANSI – тот набор, который я использую, хотя ЗНАЧЕНИЕ ПО УМОЛЧАНИЮ, вероятно, работало бы точно также.
Если Вы хотите использовать шрифт типа Webdings или Wingdings, Вы должны использовать SYMBOL_CHARSET вместо ANSI_CHARSET.

```
    ANSI_CHARSET,     // Идентификатор набора символов
```

Точность вывода очень важна. Этот параметр сообщает Windows какой из наборов символов использовать, если их доступно больше чем один. OUT_TT_PRECIS сообщает Windows что, если доступен больше чем один тип шрифта, то выбрать с тем же самым названием Truetype версию шрифта. Truetype шрифты всегда смотрят лучше, особенно когда Вы сделаете их большими по размеру. Вы можете также использовать OUT_TT_ONLY_PRECIS, при этом ВСЕГДА используется Truetype шрифт.

```
    OUT_TT_PRECIS,    // Точность вывода
```

Точность отсечения - тип отсечения, который применяется, когда вывод символов идет вне области отсечения. Об этом много говорить нечего, оставьте значение по умолчанию.

```
   CLIP_DEFAULT_PRECIS,   // Точность отсечения
```

Качество вывода - очень важный параметр. Вы можете выбрать PROOF, DRAFT, NONANTIALIASED, DEFAULT или ANTIALIASED. Всем известно, что при ANTIALIASED шрифты выглядят отлично :). Сглаживание (Antialiasing) шрифта – это тот же самый эффект, который Вы получаете, когда Вы включаете сглаживание шрифта в Windows. При этом буквы выглядят менее ступенчато.

```
    ANTIALIASED_QUALITY,   // Качество вывода
```

Затем идут настройка шага и семейство. Для настройки шага Вы можете выбрать DEFAULT_PITCH, FIXED_PITCH и VARIABLE_PITCH, а для настройки семейства, Вы можете выбрать FF_DECORATIVE, FF_MODERN, FF_ROMAN, FF_SCRIPT, FF_SWISS, FF_DONTCARE. Проиграйтесь с этими константами, чтобы выяснить, что они делают. Я оставил их по умолчанию.

```
    FF_DONTCARE|DEFAULT_PITCH,  // Семейство и шаг
```

Наконец... Фактическое название шрифта. Загрузите Microsoft Word или другой текстовый редактор. Щелчок по шрифту - выпадет вниз меню, и ищите шрифт, который Вам нравится. Чтобы его использовать, замените 'Courier New' на название шрифта, который Вы хотите использовать.

```
   "Courier New");     // Имя шрифта
```

Теперь мы выберем шрифт, привязав его к нашему DC, и построим 96 списков отображения, начиная с символа 32 (который является пробелом). Вы можете построить все 256 символов, если Вы хотите. Проверьте, что Вы удаляете все 256 списков отображения, когда Вы выходите из программы, и проверьте, что Вы задаете вместо 32 значение 0 и вместо 96 значение 255 в строке кода ниже.

```
  SelectObject(hDC, font);        // Выбрать шрифт, созданный нами ( НОВОЕ )
  wglUseFontBitmaps(hDC, 32, 96, base); // Построить 96 символов начиная с пробела ( НОВОЕ )
}
```

Следующий код очень прост. Он удаляет 96 списков отображения из памяти, начиная с первого списка, заданного 'base'. Я не уверен, что windows сделала бы это за Вас, поэтому лучше быть осмотрительным, чем потом жалеть :).

```
GLvoid KillFont(GLvoid)           // Удаление шрифта
```

```
{
  glDeleteLists(base, 96);        // Удаление всех 96 списков отображения ( НОВОЕ )
}
```

Теперь моя удобная первоклассная функция вывода текста GL. Вы вызываете этот раздел кода по команде glPrint ("здесь сообщение"). Текст находится в строке символов *fmt.

```
GLvoid glPrint(const char *fmt, ...)        // Заказная функция «Печати» GL
{
```

В первой строке кода ниже выделяется память для строки на 256 символов. text – это строка, которую мы хотим напечатать на экране. Во второй строке ниже создается указатель, который указывает на список параметров, которые мы передаем наряду со строкой. Если мы посылаем переменные вместе с текстом, она укажет на них.

```
  char    text[256];      // Место для нашей строки
  va_list    ap;          // Указатель на список аргументов
```

В следующих двух строках кода проверяется, если, что-нибудь для вывода? Если нет текста, fmt не будет равняться ничему (NULL), и ничего не будет выведено на экран.

```
  if (fmt == NULL)     // Если нет текста
    return;            // Ничего не делать

 va_start(ap, fmt);           // Разбор строки переменных
     vsprintf(text, fmt, ap); // И конвертирование символов в реальные коды
  va_end(ap);                 // Результат помещается в строку
```

Затем мы проталкиваем в стек GL_LIST_BIT, это защищает другие списки отображения, которые мы можем использовать в нашей программе, от влияния glListBase.

Команду glListBase(base-32) немного трудно объяснить. Скажем, что мы рисуем символ 'A', он задан номером 65. Без glListBase(base-32) OpenGL, не понял бы, где найти этот символ. Он стал бы искать этот символ в 65 списке отображения, но если бы база была равна 1000, 'A' был бы фактически сохранен в 1065 списке отображения. Поэтому при помощи, установки базовой отправной точки, OpenGL знает, где находится нужный список отображения. Причина, по которой мы вычитаем 32, состоит в том, что мы не сделали первые 32 списка отображения. Мы опустили их. Так что мы должны указать OpenGL про это, вычитая 32 из базового значения. Символы, кодируются, начиная с нуля, код символа пробела имеет значение 32, поэтому если, мы хотим вывести пробел, то мы должны иметь тридцать второй список отображения, а у нас он нулевой. Поэтому мы искусственно занижаем значение базы, с тем, чтобы OpenGL брал нужные списки. Я надеюсь, что это понятно.

```
  glPushAttrib(GL_LIST_BIT);      // Протолкнуть биты списка отображения ( НОВОЕ )
  glListBase(base - 32);          // Задать базу символа в 32 ( НОВОЕ )
```

Теперь, когда OpenGL знает, где находятся Символы, мы можем сообщить ему, что пора выводить текст на экран. glCallLists - очень интересная команда. Она может вывести больше, чем один список отображения на экран одновременно.

В строке ниже делает следующее. Сначала она сообщает OpenGL, что мы собираемся показывать на экране списки отображения. При вызове функции strlen(text) вычисляется, сколько символов мы собираемся отобразить на экране. Затем необходимо указать максимальное значение посылаемых символов. Мы не посылаем больше чем 255 символов. Так что мы можем использовать UNSIGNED_BYTE. (Вспомните, что байт - любое значение от 0 - 255). Наконец мы сообщаем, что надо вывести, передачей строки 'text'.

Возможно, Вы задаетесь вопросом, почему символы не наезжают друг на друга. Каждый список отображения для каждого символа знает, где правая сторона у символа. После того, как символ выведен, OpenGL переходит на правую сторону выведенного символа. Следующий символ или выведенный объект будут выведены, начиная с последнего места вывода GL, которое будет справа от последнего символа.

Наконец мы возвращаем настройки GL GL_LIST_BIT обратно, как было прежде, чем мы установили нашу базу, используя glListBase(base-32).

```
 glCallLists(strlen(text), GL_UNSIGNED_BYTE, text);// Текст списками отображения(НОВОЕ)
 glPopAttrib(); // Возврат битов списка отображения ( НОВОЕ )
}
```

В коде Init изменилось только одно: добавлена строчка BuildFont(). Она вызывает код выше, для построения шрифта, чтобы OpenGL мог использовать его позже.

```
int InitGL(GLvoid)           // Все начальные настройки OpenGL здесь
{
 glShadeModel(GL_SMOOTH);    // Разрешить плавное затенение
 glClearColor(0.0f, 0.0f, 0.0f, 0.5f); // Черный фон
 glClearDepth(1.0f);         // Установка буфера глубины
 glEnable(GL_DEPTH_TEST);    // Разрешение теста глубины
 glDepthFunc(GL_LEQUAL);     // Тип теста глубины
 // Действительно хорошие вычисления перспективы
 glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
 BuildFont();            // Построить шрифт
 return TRUE;            // Инициализация окончена
}
```

Теперь о коде для отрисовки. Вначале мы, очищаем экран и буфер глубины. Мы вызываем glLoadIdentity() чтобы все сбросить. Затем мы перемещаемся на одну единицу вглубь экрана. Если не сделать перемещения, текст не будет отображен. Растровые шрифты лучше применять с ортографической проекцией, а не с перспективной, но ортографическая проекция выглядит плохо, поэтому, когда работаем в этой проекции, перемещаем.

Вы можете заметить, что, если бы Вы переместили текст даже вглубь экрана, размер шрифта не уменьшиться, как бы Вы этого ожидали. Что реально происходит, когда Вы глубже перемещаете текст, то, что Вы имеете возможность контролировать, где текст находится на экране. Если Вы переместили на 1 единицу в экран, Вы можете расположить текст, где-нибудь от -0.5 до +0.5 по оси X. Если Вы переместите на 10 единиц в экран, то Вы должны располагать текст от -5 до +5. Это даст Вам возможность лучше контролировать точное позиционирование текста, не используя десятичные разряды. При этом размер текста не измениться. И даже с помощью glScalef(x,y,z). Если Вы хотите шрифт, больше или меньше, сделайте его большим или маленьким во время его создания!

```
int DrawGLScene(GLvoid) // Здесь мы будем рисовать все
{
 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Очистка экран и буфера глубины
 glLoadIdentity(); // Сброс просмотра
 glTranslatef(0.0f,0.0f,-1.0f); // Передвижение на одну единицу вглубь
```

Теперь мы воспользуемся нестандартными вычислениями, чтобы сделать цветовую пульсацию. Не волнуйтесь, если Вы не понимаете то, что я делаю. Я люблю пользоваться множеством переменных, и дурацкими уловками, чтобы достигнуть результата :).

На этот раз, я использую два счетчика, которые мы создали для перемещения текста по экрану, и для манипулирования красным, зеленым и синим цветом. Красный меняется от -1.0 до 1.0 используя COS и счетчик 1. Зеленый меняется от -1.0 до 1.0 используя SIN и счетчик 2. Синий меняется от 0.5 до 1.5 используя COS и счетчики 1 + 2. Тем самым синий никогда не будет равен 0, и текст не должен никогда полностью исчезнуть. Глупо, но это работает :).

```
 // Цветовая пульсация, основанная на положении текста
 glColor3f(1.0f*float(cos(cnt1)),1.0f*float(sin(cnt2)),1.0f-0.5f*float(cos(cnt1+cnt2)));
```

Теперь новая команда. glRasterPos2f(x,y) будет позиционировать растровый шрифт на экране. Центр экрана как прежде в 0,0. Заметьте, что нет координаты Z. Растровые шрифты используют только ось X (лево/право) и ось Y (вверх/вниз). Поскольку мы перемещаем на одну единицу в экран, левый край равен -0.5, и правый край равен +0.5. Вы увидите, что я перемещаю на 0.45 пикселей влево по оси X. Это устанавливает текст в центр экрана. Иначе он было бы правее на экране, потому что текст будет выводиться от центра направо.

Нестандартные вычисления делают в большой степени то же самое, как и при вычислении цвета. Происходит перемещение текста по оси X от -0.50 до -0.40 (вспомните, мы вычли справа от начала 0.45). При этом текст на экране будет всегда. Текст будет ритмично раскачиваться влево и вправо, используя COS и счетчик 1. Текст будет перемещаться от -0.35 до +0.35 по оси Y, используя SIN и счетчик 2.

```
  // Позиционирование текста на экране
  glRasterPos2f(-0.45f+0.05f*float(cos(cnt1)), 0.35f*float(sin(cnt2)));
```

Теперь моя любимая часть... Реальный вывод текста на экран. Я попробовал сделать его очень простым, и крайне дружелюбным способом. Вы увидите, что вывод текста выглядит как большинство команд OpenGL, при этом в комбинации с командой Print, сделанной на старый добрый манер :). Все, что Вам надо сделать, чтобы ввести текст на экран - glPrint ("{любой текст, который Вы хотите}"). Это очень просто. Текст будет выведен на экран точно в том месте, где Вы установили его.

Shawn T. прислал мне модифицированный код, который позволяет glPrint передавать переменные для вывода на экран. Например, Вы можете увеличить счетчик и отобразить результат на экране! Это работает примерно так... В линии ниже Вы увидите нормальный текст. Затем идет пробел, тире, пробел, затем "символ" (%7.2f). Посмотрев на %7.2f Вы можете сказать, что эта рогулька означает. Все очень просто. Символ % - подобен метке, которая говорит, не печатать 7.2f на экран, потому что здесь будет напечатано значение переменной. При этом 7 означает, что максимум 7 цифр будут отображены слева от десятичной точки. Затем десятичная точка, и справа после десятичной точки - 2. 2 означает, что только две цифры будут отображены справа от десятичной точки. Наконец, f. f означает, что число, которое мы хотим отобразить - число с плавающей запятой. Мы хотим вывести значение cnt1 на экран. Предположим, что cnt1 равен 300.12345f, окончательно мы бы увидели на экране 300.12. Цифры 3, 4, и 5 после десятичной точки были бы обрезаны, потому что мы хотим, чтобы появились только 2 цифры после десятичной точки.

Конечно, если Вы профессиональный программист на Си, то, это ненужный рассказ, но этот урок могут читать люди, которые и не использовали printf. Если Вы хотите больше узнать о маркерах, купите книгу, или посмотрите MSDN.

```
 glPrint("Active OpenGL Text With NeHe - %7.2f", cnt1);  // Печать текста GL на экран
```

И в завершении увеличим значение обоих счетчиков на разную величину, чтобы была цветовая пульсация и передвижение текста.

```
  cnt1+=0.051f;  // Увеличение первого счетчика
  cnt2+=0.005f;  // Увеличение второго счетчика
  return TRUE;   // Все отлично
}
```

Также необходимо добавить KillFont() в конец KillGLWindow() как, показано ниже. Важно добавить эту строку. При этом списки отображения очищаются прежде, чем мы выходим из нашей программы.

```
  if (!UnregisterClass("OpenGL",hInstance))    // Если класс не зарегистрирован
  {
    MessageBox(NULL,"Could Not Unregister Class.","SHUTDOWN ERROR",
MB_OK | MB_ICONINFORMATION);
    hInstance=NULL;          // Установить копию приложения в ноль
  }
 KillFont();            // Уничтожить шрифт
}
```

Вот и все... Все, что Вы должны знать, чтобы использовать растровые шрифты в ваших собственных проектах OpenGL. Я поискал в сети подобный материал, и ничего похожего не нашел. Возможно мой сайт первый раскрывает эту тему на простом понятном коде Си? Возможно. Получайте удовольствие от этого урока, и счастливого кодирования!

Урок 14. Векторные шрифты

Outline Fonts

Этот урок представляет собой продолжение предыдущего, 13-го урока, в котором я показал вам, как использовать растровые шрифты, и будет посвящен векторным шрифтам.
Путь, которым мы пойдем при построения векторного шрифта, подобен тому, как мы строили растровый шрифт в уроке 13. И все-таки:

Векторные шрифты во сто раз круче: вы можете менять их размер, перемещать по экрану в трехмерной проекции и они при этом могут быть объемными! Никаких плоских двумерных букв. В качестве трехмерного шрифта для OpenGL- приложения вы сможете использовать любой шрифт, имеющийся на вашем компьютере. И, наконец, - собственные нормали, помогающие так мило зажечь буквы, что просто приятно смотреть, как они светятся :).

Небольшое замечание - приведенный код предназначен для Windows. Для построения шрифтов в нем используются WGL-функции из набора Windows API. Очевидно, Apple имеет в AGL поддержку тех же вещей, и X - в GLX. К сожалению, я не могу гарантировать переносимость этого кода. Если кто-нибудь имеет независимый от платформы код для отрисовки шрифтов на экране, пришлите их мне и я напишу другое руководство.

Начнем со стандартного кода из урока номер 1. Добавим STDIO.H для стандартных операций ввода-вывода; STDARG.H - для разбора текста и конвертации в текст числовых переменных, и, MATH.H - для вызова функций SIN и COS, которые понадобятся, когда мы захотим вращать текст на экране :).

```
#include <windows.h> // заголовочный файл для Windows
#include <math.h>    // заголовочный файл для математической бибилиотеки Windows(добавлено)
#include <stdio.h>   // заголовочный файл для стандартного ввода/вывода(добавлено)
#include <stdarg.h>  // заголовочный файл для манипуляций
                     // с переменными аргументами (добавлено)
#include <gl\gl.h>   // заголовочный файл для библиотеки OpenGL32
#include <gl\glu.h>  // заголовочный файл для библиотеки GLu32
#include <gl\glaux.h>// заголовочный файл для библиотеки GLaux

HDC       hDC=NULL;  // Частный контекст устройства GDI
HGLRC     hRC=NULL;  // Контекст текущей визуализации
HWND      hWnd=NULL; // Декриптор нашего окна
HINSTANCE hInstance; // Копия нашего приложения
```

Теперь добавим 2 новые переменные. Переменная base будет содержать номер первого списка отображения ('display list' по-английски, представляет собой последовательность команд OpenGL, часто выполняемую и хранящуюся в специальном формате, оптимизированном для скорости выполнения - прим. перев.), который мы создадим. Каждому символу требуется свой список отображения. Например, символ 'A' имеет 65-й номер списка отображения, 'B' - 66, 'C' - 67, и т.д. То есть символ 'A' будет хранится в списке отображения base+65.

Кроме этой добавим переменную rot. Она будет использоваться при вычислениях для вращения текста на экране через функции синуса и косинуса, И еще она будет использована при пульсации цветов.

```
GLuint base;              // База отображаемого списка для набора символов (добавлено)
GLfloat   rot;            // Используется для вращения текста (добавлено)
bool   keys[256];         // Массив для манипуляций с клавиатурой
bool   active=TRUE;       // Флаг активности окна, по умолчанию=TRUE
bool   fullscreen=TRUE;   // Флаг полноэкранного режима, по умолчанию=TRUE
```

GLYPHMETRICSFLOAT gmf[256] будет содержать информацию о местоположении и ориентации каждого из 256 списков отображения нашего векторного шрифта. Чтобы получить доступ к нужной букве просто напишем gmf[num], где num - это номер списка отображения, соответствующий требуемой букве. Позже в программе я покажу вам, как узнать ширину каждого символа для того, чтобы вы смогли автоматически центрировать текст на экране. Имейте в виду, что каждый символ имеет свою ширину. Метрика шрифта (glyphmetrics) на порядок облегчит вам жизнь.

```
GLYPHMETRICSFLOAT gmf[256];  // Массив с информацией о нашем шрифте
LRESULT CALLBACK WndProc(
HWND, UINT, WPARAM, LPARAM); // Объявление оконной процедуры
```

Следующая часть программы создает шрифт так же, как и при построении растрового шрифта. Как и в уроке №13, это место в программе было для меня самым трудным для объяснения.

Переменная HFONT font будет содержать идентификатор шрифта Windows.

Далее заполним переменную base, создав набор из 256-ти списков отображения, вызвав функцию glGenLists(256). После этого переменная base будет содержать номер первого списка отображения.

```
GLvoid BuildFont(GLvoid)          // Строим растровый шрифт
{
  HFONT  font;                    // Идентификатор шрифта Windows
  base = glGenLists(256);         // массив для 256 букв
```

Теперь будет интереснее :). Делаем наш векторный шрифт. Во-первых, определим его размер. В приведенной ниже строке кода вы можете заметить отрицательное значение. Этим минусом мы говорим Windows, что наш шрифт основан на высоте СИМВОЛА. А положительное значение дало бы нам шрифт, основанный на высоте ЗНАКОМЕСТА.

```
font = CreateFont(      -12,        // высота шрифта
```

Определим ширину знакоместа. Вы видите, что ее значение установлено в ноль. Этим мы заставляем Windows взять значение по умолчанию. Если хотите, поиграйтесь с этим значением, сделайте, например, широкие буквы.

```
        0,          // ширина знакоместа
```

Угол отношения (Angle of Escapement) позволяет вращать шрифт. Угол наклона (Orientation Angle), если сослаться на 'MSDN help', определяет угол, в десятых долях градуса, между базовой линией символа и осью X экрана. Я, к сожалению, не имею идей насчет того, зачем это :(.

```
        0,          //Угол перехода
        0,          //Угол направления
```

Ширина шрифта - важный параметр. Может быть в пределах от 0 до 1000. Также можно использовать предопределенные значения: FW_DONTCARE = 0, FW_NORMAL = 400, FW_BOLD = 700 и FW_BLACK = 900. Таких значений много, но эти дают наилучший результат. Понятно, что чем выше значение, тем толще (жирнее) шрифт.

```
FW_BOLD,            //Ширина шрифта
```

Параметры Italic, Underline и Strikeout (наклонный, подчеркнутый и зачеркнутый) могут иметь значения TRUE или FALSE. Хотите, к примеру, чтобы шрифт был подчеркнутым, в параметре подчеркнутости поставьте значение TRUE, не хотите - FALSE. Просто :).

```
        FALSE,      // Курсив
        FALSE,      // Подчеркивание
        FALSE,      // Перечеркивание
```

Идентификатор набора символов определяет, соответственно, набор символов (кодировку), который мы хотим использовать. Их, конечно, очень много: CHINESEBIG5_CHARSET, GREEK_CHARSET, RUSSIAN_CHARSET, DEFAULT_CHARSET и т.д. Я, к примеру, использую ANSI. Хотя, DEFAULT тоже может подойти. Если интересуетесь такими шрифтами, как Webdings или Wingdings, вам надо использовать значение SYMBOL_CHARSET вместо ANSI_CHARSET.

```
        ANSI_CHARSET,       //Идентификатор кодировки
```

Точность вывода - тоже важный параметр. Он говорит Windows о том, какой тип символа использовать, если их более одного. Например, значение OUT_TT_PRECIS означает, что надо взять TRUETYPE - версию шрифта.Truetype - шрифт обычно смотрится лучше, чем другие, особенно когда буквы большие. Можно также использовать значение OUT_TT_ONLY_PRECIS, которое означает, что всегда следует брать, если возможно, шрифт TRUETYPE.

```
        OUT_TT_PRECIS,      // Точность вывода
```

Точность отсечения - этот параметр указывает вид отсечения шрифта при попадании букв вне определенной области. Сказать о нем больше нечего, просто оставьте его по умолчанию.

```
        CLIP_DEFAULT_PRECIS,        //Точность отсечения
```

Качество вывода - очень важный параметр. Можете поставить PROOF, DRAFT, NONANTIALIASED, DEFAULT или ANTIALIASED. Мы с вами знаем, что ANTIALIASED будет лучше смотреться :). Сглаживание (Antialiasing) шрифта - это эффект, позволяющий сгладить шрифт в Windows. Он делает вид букв менее ступенчатым.

```
        ANTIALIASED_QUALITY,// Качество вывода
```

Следующими идут значения семейства (Family) и шага (Pitch). Шаг может принимать значения DEFAULT_PITCH, FIXED_PITCH и VARIABLE_PITCH. Семейство может быть FF_DECORATIVE, FF_MODERN, FF_ROMAN, FF_SCRIPT, FF_SWISS, FF_DONTCARE. Поиграйте этими значениями, чтобы понять, что они делают. Здесь оба параметра установлены по умолчанию.

```
        FF_DONTCARE|DEFAULT_PITCH, // Семейство и Шаг
```

И последнее... Нам нужно настоящее имя используемого нами шрифта. Его можно увидеть, например в Word при выборе шрифта для текста. Зайдите в Word или другой текстовый редактор, выберите шрифт, который требуется, запомните, как он называется, и вставьте его название вместо значения 'Comic Sans MS' в приведенной ниже строке.

```
        "Comic Sans MS");         // Имя шрифта
```

Теперь выберем этот шрифт, связав его с нашим контекстом устройста (DC).

```
SelectObject(hDC, font);       //Выбрать шрифт, созданный нами
```

Подошло время добавить новые строки в программу. Мы построим наш векторный шрифт, используя новую команду - wglUseFontOutlines. Выберем контекст устройства (DC), начальный символ, количество создаваемых символов, и базовое значение списка отображения. Все очень похоже на то, как мы строили растровый шрифт.

```
wglUseFontOutlines( hDC,         // Выбрать текущий контекст устройства (DC)
        0,          // Стартовый символ
        255,         // Количество создаваемых списков отображения
        base,        // Стартовое значение списка отображения
```

Но это еще не все. Здесь мы установим уровень отклонения. Уже при значении 0.0f сглаживание будет видимо. После установки отклонения идет определение толщины шрифта (ширины по оси Z). Толщина, равная 0.0f, понятно, даст нам плоские буквы, то есть двумерные. А вот при значении 1.0f уже будет виден некоторый объем.

Параметр WGL_FONT_POLYGONS говорит OpenGL создать "твердый" шрифт при помощи полигонов. Если вместо него вы укажете WGL_FONT_LINES, шрифт будет каркасным, или "проволочным" (составленным из линий). Стоит заметить, если вы укажете значение GL_FONT_LINES, не будут сгенерированы нормали, что сделает невозможным свечение букв.

Последний параметр, gmf указывает на буфер адреса для данных списка отображения.

```
        0.0f,       //Отклонение от настоящего контура
        0.2f,       //Толщина шрифта по оси Z
        WGL_FONT_POLYGONS,       //Использовать полигоны, а не линии
        gmf),       //буфер адреса для данных списка отображения
        }
```

Следующий код очень простой. Он удаляет 256 списков отображения из памяти, начиная с первого списка, номер которого записан в base. Не уверен, что Windows сделает это за нас, поэтому лучше позаботиться самим, чем потом иметь глюки в системе.

```
GLvoid KillFont(GLvoid)                  // Удаление шрифта
        {
        glDeleteLists(base, 256); // Удаление всех 256 списков отображения
        }
```

Теперь функция вывода текста в OpenGL. Ее мы будем вызывать не иначе как glPrint("Выводимый текст"), точно также, как и в варианте с растровым шрифтом из урока N 13. Текст при этом помещается в символьную строку fmt.

```
GLvoid glPrint(const char *fmt, ...)     // Функция вывода текста в OpenGL
        {
```

В первой строке, приведенной ниже, объявляется и инициализируется переменная length. Она будет использована при вычислении того, какой из нашей строки выйдет текст. Во второй строке создается пустой массив для текстовой строки длиной в 256 символов. text - строка, из которой будет происходить вывод текста на экран. В третьей строке

описывается указатель на список аргументов, передаваемый со строкой в функцию. Если мы пошлем с текстом какие-нибудь переменные, этот указатель будет указывать на них.

```
float        length=0;     // Переменная для нахождения
                           // физической длины текста
 char         text[256];    // Здесь наша строка
 va_list             ap;    // Указатель на переменный список аргументов
```

Проверим, а может нет никакого текста :). Тогда выводить нечего, и - return.

```
 if (fmt == NULL)           // Если нет текста,
     return;              // ничего не делаем
```

В следующих трех строчках программа конвертирует любые символы в переданной строке в представляющие их числа. Проще говоря, мы можем использовать эту функцию, как C-функцию printf. Если в переданном постоянном аргументе мы послали программе строку форматирования, то в этих трех строках она будет читать переменный список аргументов-переменных и в соответствии со строкой форматирования преобразовать их в конечный понятный для программы текст, сохраняемый в переменной text. Подробности идут дальше.

```
 va_start(ap, fmt);         // Анализ строки на переменные
 vsprintf(text, fmt, ap);   // И конвертация символов в реальные коды
 va_end(ap);                // Результат сохраняется в text
```

Стоит сказать отдельное спасибо Джиму Вильямсу (Jim Williams) за предложенный ниже код. До этого я центрировал текст вручную, но его метод немного лучше :). Начнем с создания цикла, который проходит весь текст символ за символом. Длину текста вычисляем в выражении strlen(text). После обработки данных цикла (проверки условия завершения) идет его тело, где к значению переменной lenght добавляется физическая ширина каждого символа. После завершения цикла величина, находящаяся в lenght, будет равна ширине всей строки. Так, если мы напечатаем при помощи данной функции слово "hello", и по каким-то причинам каждый символ в этом слове будет иметь ширину ровно в 10 единиц, то, понятно, что в итоге у нас получится значение lenght=10+10+10+10+10=50 единиц.

Значение ширины каждого символа получается из выражения gmf[text[loop]].gmfCellIncX. Помните, что gmf хранит информацию о каждом списке отображения. Если loop будет равна 0, то text[loop] - это будет первый символ нашей с вами строки. Соответственно, при loop, равной 1, то text[loop] будет означать второй символ этой же строки. Ширину символа дает нам gmfCellIncX. На самом деле gmfCellIncX - это расстояние, на которое смещается вправо позиция графического вывода после отображения очередного символа для того, чтобы символы не наложились друг на друга. Так уж вышло, что это расстояние и наша ширина символа - одно и то же значение.

Высоту символа можно узнать при помощи команды gmfCellIncY. Это может пригодиться при выводе вертикального текста.

```
for (unsigned int loop=0;loop<(strlen(text));loop++)//Цикл поиска размера строки
        {
        length+=gmf[text[loop]].gmfCellIncX;
        // Увеличение размера на ширину символа
        }
```

Полученную величину размера строки преобразуем в отрицательную (потому что мы будем перемещаться влево от центра для центровки текста). Затем мы делим длину на 2: нам не нужно перемещать весь текст влево - только его половину.

```
glTranslatef(-length/2,0.0f,0.0f);       //Центровка на экране нашей строки
```

Дальше мы сохраним в стеке значение GL_LIST_BIT, это сохранит glListBase от воздействия любых других списков отображения, используемых в нашей программе.

Команда glListBase(base) говорит OpenGL, где искать для каждого символа соответствующий список отображения.

```
glPushAttrib(GL_LIST_BIT); // Сохраняет в стеке значения битов списка отображения
glListBase(base);          // Устанавливает базовый символ в 0
```

Теперь, когда OpenGL знает расположение всех символов, мы можем сказать ей написать текст на экране. glCallLists выводит всю строку текста на экран сразу, создавая для вас многочисленные списки отображения.

Строка, приведенная ниже, делает следующее. Сначала она сообщает OpenGL, где на экране будут находиться списки отображения. strlen(text) находит количество букв, которые мы хотим послать на экран. Далее ему потребуется знать, какой наибольший будет номер списка из подготавливаемых к посылке. Пока что мы не посылаем более 255 символов. Так что мы можем использовать значение GL_UNSIGNED_BYTE (байт позволяет хранить целые числа от 0 до 255). Наконец, мы ему скажем, что отображать при помощи передачи строки text.

На тот случай, если вас удивит, что буквы не сваливаются в кучу одна над другой. Каждый список отображения каждого символа знает, где находится правая сторона предыдущего символа. После того, как буква отобразилась на экране, OpenGL перемещает вывод к правой стороне нарисованной буквы. Следующая буква или объект рисования начнет рисоваться с последней позиции OpenGL после перемещения, находящейся справа от последней буквы.

Наконец, мы восстанавливаем из стека GL_LIST_BIT - установки OpenGL обратно по состоянию на тот момент. Как они были перед установкой базовых значений командой glCallLists(base).

```
    // Создает списки отображения текста
    glCallLists(strlen(text), GL_UNSIGNED_BYTE, text);
    glPopAttrib(); // Восстанавливает значение Display List Bits
            }
```

Участок программы, отвечающий за размеры в окне OpenGL, точно такой же, как и в уроке N 1, поэтому здесь мы его пропустим.

В конец функции InitGL добавилось несколько новых строк. Строка с выражением BuildFont() из 13 урока осталась прежней, вместе с новым кодом, который создает быстрое и черновое освещение. Источник света Light0 встроен в большинство видеокарт, поэтому достаточно приемлемое освещение сцены не потребует особых усилий с нашей стороны.

Еще я добавил команду glEnable(GL_Color_Material). Поскольку символы являются 3D-объектами, вам понадобится раскраска материалов (Material Coloring). В противном случае смена цвета с помощью glColor3f(r,g,b) не изменит цвет текста. Если у вас на экран выводятся кроме текста другие фигуры-объекты 3D-сцены, включайте раскраску материалов перед выводом текста и отключайте сразу после того, как текст будет нарисован, иначе будут раскрашены все объекты на экране.

```
int InitGL(GLvoid)                      // Здесь будут все настройки для OpenGL
{
    glShadeModel(GL_SMOOTH);            // Включить плавное затенение
    glClearColor(0.0f, 0.0f, 0.0f, 0.5f);   // Черный фон
    glClearDepth(1.0f);                 // Настройка буфера глубины
    glEnable(GL_DEPTH_TEST);            // Разрешить проверку глубины
    glDepthFunc(GL_LEQUAL);             // Тип проверки глубины
    // Действительно хорошие вычисления перспективы
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
    glEnable(GL_LIGHT0);        // Включить встроенное освещение (черновое) (новая)
    glEnable(GL_LIGHTING);              // Разрешить освещение                  (новая)
    glEnable(GL_COLOR_MATERIAL);        // Включить раскраску материалов (новая)

    BuildFont();                    // Построить шрифт (добавлена)
   return TRUE;                     // Инициализация прошла успешно
}
```

Теперь рисующий код. Начнем с очистки экрана и буфера глубины. Для полного сброса вызовем функцию glLoadIdentity(). Затем мы смещаемся на 10 единиц вглубь экрана. Векторный шрифт великолепно смотрится в режиме перспективы. Чем дальше в экран смещаемся, тем меньше шрифт. Чем ближе, тем шрифт больше.

Управлять векторными шрифтами можно также при помощи команды glScalef(x,y,z). Если захотите сделать буквы в два раза выше, дайте команду glScalef(1.0f,2.0f,1.0f). Значение 2.0f здесь относится к оси Y и сообщает OpenGL, что список отображения нужно нарисовать в двойной высоте. Если это значение поставить на место первого аргумента (X), то буквы будут в два раза шире. Ну, третий аргумент, естественно, касается оси Z.

```
int DrawGLScene(GLvoid)             // Здесь весь вывод на экран
{
   // Очистка экрана и буфера глубины
```

```
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();               // Сброс вида
    glTranslatef(0.0f,0.0f,-10.0f);   // Смещение на 10 единиц в экран
```

После сдвига вглубь экрана мы можем повращать текст. Следующие три строки поворачивают изображение по трем осям. Умножением переменной rot на различные значения я пытался добиться как можно более различающихся скоростей вращения.

```
    glRotatef(rot,1.0f,0.0f,0.0f);          // Поворот по оси X
    glRotatef(rot*1.5f,0.0f,1.0f,0.0f);     // Поворот по оси Y
    glRotatef(rot*1.4f,0.0f,0.0f,1.0f);     // Поворот по оси Z
```

Теперь займемся цветовым циклом. Как обычно, использую здесь переменную-счетчик (rot). Возгорание и затухание цветов получается при помощи функций SIN и COS. Я делил переменную rot на разные числа, так что бы каждый цвет не возрастал с такой же скоростью. Результат впечатляющий.

```
    // Цветовая пульсация основанная на вращении
    glColor3f(1.0f*float(cos(rot/20.0f)),1.0f*float(sin(rot/25.0f)),
        1.0f-0.5f*float(cos(rot/17.0f)));
```

Моя любимая часть... Запись текста на экран. Мною были использованы несколько команд, которые мы применяли также при выводе на экран растровых шрифтов. Сейчас вы уже знаете, как вывести текст в команде glPrint("Ваш текст"). Это так просто!

В коде, приведенном ниже, мы печатаем "NeHe", пробел, тире, пробел и число из переменной rot и разделенное на 50, чтобы немного его уменьшить. Если число больше 999.99, разряды слева игнорируются (так как в команде мы указали 3 разряда на целую часть числа и 2 - на дробную после запятой).

```
    glPrint("NeHe - %3.2f",rot/50);   // Печать текста на экране
```

Затем увеличиваем переменную rot для дальнейшей пульсации цвета и вращения текста.

```
    rot+=0.5f;                  // Увеличить переменную вращения
    return TRUE;                    // Все прошло успешно
        }
```

И последняя вещь, которую мы должны сделать, это добавить строку KillFont() в конец функции KillGLWindow(), так как я это сделал ниже. Очень важно это сделать. Она очистит все, что касалось шрифта, прежде чем мы выйдем из нашей программы.

```
if (!UnregisterClass("OpenGL",hInstance))// Если класс незарегистрирован
    {
    MessageBox(NULL,"Could Not Unregister Class.",
        "SHUTDOWN ERROR",MB_OK | MB_ICONINFORMATION);
    hInstance=NULL;                 // Установить копию приложения в ноль
    }
KillFont();                         // Уничтожить шрифт
```

Под конец урока вы уже должны уметь использовать векторные шрифты в ваших проектах, использующих OpenGL. Как и в случае с уроком N 13, я пытался найти в сети подобные руководства, но, к сожалению, ничего не нашел. Возможно мой сайт - первый в раскрытии данной темы в подробном рассмотрении для всех, понимающих язык Си? Читайте руководство и удачного вам программирования!

Урок 15. Текстурные шрифты

Texture Mapped Fonts

После публикации двух последних уроков о растровых и векторных шрифтах, я получил несколько писем от людей, которые задают вопрос: можно ли накладывать текстуру на шрифт. Вы можете использовать автоматическую генерацию координат текстуры. При этом будут генерироваться координаты текстуры для каждого полигона у шрифта.

Небольшое примечание, этот код применим только в Windows. Здесь используются функции wgl Windows для построения шрифта. Очевидно, Apple имеет поддержку agl, которая должна делать тоже самое, и X имеет glx. К сожалению, я не могу гарантировать, что этот код переносим. Если кто-нибудь имеет платформо-незавизимый код для вывода шрифтов на экран, пришлите мне его, и я напишу другой урок по шрифтам.

Мы будем использовать код от урока 14 для нашей демонстрационной программы текстурных шрифтов. Если код изменился в каком-либо разделе программы, я перепишу весь раздел кода, чтобы было проще видеть изменения, которые я сделал.

Следующий раздел кода такой же как в уроке 14, но на этот раз мы не включим в него stdarg.h.

```
#include <windows.h>  // Заголовочный файл для Windows
#include <stdio.h>    // Заголовочный файл для стандартной библиотеки ввода/вывода
#include <gl\gl.h>    // Заголовочный файл для библиотеки OpenGL32
#include <gl\glu.h>   // Заголовочный файл для библиотеки GLu32
#include <gl\glaux.h> // Заголовочный файл для библиотеки GLaux
#include <math.h>     // Заголовочный файл для математической библиотеки

HDC       hDC=NULL;  // Приватный контекст устройства GDI
HGLRC     hRC=NULL;  // Постоянный контекст рендеринга
HWND      hWnd=NULL; // Сохраняет дескриптор окна
HINSTANCE  hInstance; // Сохраняет экземпляр приложения

bool  keys[256];      // Массив для работы с клавиатурой
bool  active=TRUE;    // Флаг активации окна, по умолчанию = TRUE
bool  fullscreen=TRUE;// Флаг полноэкранного режима
```

Мы собираемся добавить одну новую переменную целого типа, которая называется texture[]. Она будет использоваться для хранения нашей текстуры. Последние три строки такие же, как в уроке 14 и не изменились и здесь.

```
GLuint  texture[1]; // Одна текстура ( НОВОЕ )
GLuint  base;       // База списка отображения для фонта
GLfloat   rot;      // Используется для вращения текста

LRESULT  CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM); // Объявление WndProc
```

Следующий раздел кода претерпел незначительные изменения. В этом уроке я собираюсь использовать wingdings шрифт, для того чтобы отобразить объект в виде черепа и двух скрещенных костей под ним (эмблема смерти). Если Вы захотите вывести текст вместо этого, Вы можете оставить код таким же, как это было в уроке 14, или замените шрифт на ваш собственный.

Может быть кто-то уже задавался вопросом, как использовать wingdings шрифт. Это тоже является причиной, по которой я не использую стандартный шрифт. wingdings – СПЕЦИАЛЬНЫЙ шрифт, и требует некоторых модификаций, чтобы заставить программу работать с ним. При этом надо не просто сообщить Windows, чтобы Вы будете использовать wingdings шрифт. Если Вы изменяете, название шрифта на wingdings, Вы увидите, что шрифт не будет выбран. Вы должны сообщить Windows, что шрифт является специальным шрифтом, и не стандартным символьным шрифтом. Но об этом позже.

```
GLvoid BuildFont(GLvoid)       // Построение шрифта
{
  GLYPHMETRICSFLOAT  gmf[256]; // Адрес буфера для хранения шрифта
  HFONT  font;                 // ID шрифта в Windows

  base = glGenLists(256);      // Храним 256 символов
  font = CreateFont(  -12,     // Высота фонта
        0,        // Ширина фонта
        0,        // Угол отношения
        0,        // Угол наклона
        FW_BOLD,  // Ширина шрифта
        FALSE,    // Курсив
        FALSE,    // Подчеркивание
        FALSE,    // Перечеркивание
```

Вот она волшебная строка! Вместо того чтобы использовать ANSI_CHARSET, как мы делали в уроке 14, мы используем SYMBOL_CHARSET. Это говорит Windows, что шрифт, который мы строим - не обычный шрифт, составленный из букв. Специальный шрифт обычно составлен из крошечных картинок (символов). Если Вы забудете изменить эту строку, wingdings, webdings и любой другой специальный шрифт, который Вы можете пробовать использовать, не будет работать.

```
    SYMBOL_CHARSET,      // Идентификатор набора символов ( Модифицировано )
```

Следующие строки не изменились.

```
    OUT_TT_PRECIS,       // Точность вывода
    CLIP_DEFAULT_PRECIS, // Точность отсечения
    ANTIALIASED_QUALITY, // Качество вывода
    FF_DONTCARE|DEFAULT_PITCH, // Семейство и шаг
```

Теперь, когда мы выбрали идентификатор набора символов, мы можем выбирать wingdings шрифт!

```
    "Wingdings");       // Имя шрифта ( Модифицировано )
 SelectObject(hDC, font);  // Выбрать шрифт, созданный нами
 wglUseFontOutlines(  hDC, // Выбрать текущий контекст устройства (DC)
    0,                // Стартовый символ
    255,              // Количество создаваемых списков отображения
    base,             // Стартовое значение списка отображения
```

Я устанавливаю большой уровень отклонения. При этом GL не будет точно отслеживать контур шрифта. Если Вы зададите отклонение равным 0.0f, Вы заметите проблемы с текстурированием на очень изогнутых поверхностях. Если Вы допустите некоторое отклонение, большинство проблем исчезнет.

```
    0.1f,       // Отклонение от истинного контура
```

Следующие три строки кода те же самые.

```
    0.2f,       // Толщина шрифта по оси Z
    WGL_FONT_POLYGONS, // Использовать полигоны, а не линии
    gmf);       // Буфер адреса для данных списка отображения
}
```

Перед ReSizeGLScene() мы собираемся добавить следующий раздел кода для загрузки нашей текстуры. Вы знаете этот код по прошлым урокам. Мы создаем память для хранения растрового изображения. Мы загружаем растровое изображение. Мы говорим OpenGL, сгенерировать 1 текстуру, и мы сохраняем эту текстуру в texture[0].

Я создаю мип-мап текстуру, так как она смотрится лучше. Имя текстуры - lights.bmp.

```
AUX_RGBImageRec *LoadBMP(char *Filename)     // Загрузка картинки
{
 FILE *File=NULL;         // Индекс файла

 if (!Filename)           // Проверка имени файла
 {
  return NULL;            // Если нет вернем NULL
 }

 File=fopen(Filename,"r"); // Проверим существует ли файл

 if (File)                // Файл существует?
 {
  fclose(File);           // Закрыть файл
  return auxDIBImageLoad(Filename); // Загрузка картинки и вернем на нее указатель
 }
 return NULL;             // Если загрузка не удалась вернем NULL
}
```

```
int LoadGLTextures()                    // Загрузка картинки и конвертирование в текстуру
{
 int Status=FALSE;                      // Индикатор состояния
AUX_RGBImageRec *TextureImage[1];       // Создать место для текстуры
memset(TextureImage,0,sizeof(void *)*1); // Установить указатель в NULL

 // Загрузка картинки, проверка на ошибки, если картинка не найдена - выход
 if (TextureImage[0]=LoadBMP("Data/Lights.bmp"))
 {
 Status=TRUE;       // Установим Status в TRUE
 glGenTextures(1, &texture[0]);     // Создание трех текстур
 // Создание текстуры с мип-мап наложением
 glBindTexture(GL_TEXTURE_2D, texture[0]);
 gluBuild2DMipmaps(GL_TEXTURE_2D, 3, TextureImage[0]->sizeX, TextureImage[0]->sizeY,
  GL_RGB, GL_UNSIGNED_BYTE, TextureImage[0]->data);
 glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
 glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR_MIPMAP_NEAREST);
 }
```

В следующих четырех строках кода автоматически генерируются текстурные координаты для любого объекта, который мы выводим на экран. Команда glTexGen чрезвычайно мощная и комплексная, и включает достаточно сложную математику. Но Вы должны только знать то, что GL_S и GL_T - координаты текстуры. По умолчанию они заданы так, чтобы брать текущее x положение на экране и текущее y положение на экране и из них вычислить вершину текстуру. Вы можете отметить, что объекты не текстурированны по z плоскости... только появляются полосы. Передние и задние грани текстурированны, однако, именно это и необходимо. X (GL_S) охватывает наложение текстуры слева направо, а Y (GL_T) охватывает наложение текстуры сверху и вниз.

GL_TEXTURE_GEN_MODE позволяет нам выбрать режим наложения текстуры, который мы хотим использовать по координатам текстуры S и T. Есть три возможности:

GL_EYE_LINEAR - текстура зафиксирована на экране. Она никогда не перемещается. Объект накладывается на любую часть текстуры, которую он захватывает.

GL_OBJECT_LINEAR – мы воспользуемся этим режимом. Текстура привязана к объекту, перемещающемуся по экрану.

GL_SPHERE_MAP – всегда в фаворе. Создает металлический отражающий тип объекта.

Важно обратить внимание на то, что я опускаю много кода. Мы также должны задать GL_OBJECT_PLANE, но значение по умолчанию то, которое мы хотим. Купите хорошую книгу, если Вы хотите изучить больше, или поищите в помощи MSDN на CD.

```
   // Текстуризация контура закрепленного за объектом
   glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
   // Текстуризация контура закрепленного за объектом
   glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
   glEnable(GL_TEXTURE_GEN_S);      // Автоматическая генерация
   glEnable(GL_TEXTURE_GEN_T);      // Автоматическая генерация
  }

  if (TextureImage[0])          // Если текстура существует
  {
   if (TextureImage[0]->data)    // Если изображение текстуры существует
   {
    free(TextureImage[0]->data); // Освобождение памяти изображения текстуры
   }

  free(TextureImage[0]);        // Освобождение памяти под структуру
  }
 return Status;        // Возвращаем статус
}
```

Есть несколько новых строк кода в конце InitGL(). Вызов BuildFont() был помещен ниже кода, загружающего нашу текстуру. Строка с glEnable(GL_COLOR_MATERIAL) была удалена. Если Вы хотите задать текстуре цвет, используйте glColor3f(r, г, b) и добавьте строку glEnable(GL_COLOR_MATERIAL) в конце этой секции кода.

```
int InitGL(GLvoid)        // Все начальные настройки OpenGL здесь
{
 if (!LoadGLTextures())  // Переход на процедуру загрузки текстуры
 {
  return FALSE;         // Если текстура не загружена возвращаем FALSE
 }
 BuildFont();           // Построить шрифт

 glShadeModel(GL_SMOOTH);    // Разрешить плавное затенение
 glClearColor(0.0f, 0.0f, 0.0f, 0.5f); // Черный фон
 glClearDepth(1.0f);         // Установка буфера глубины
 glEnable(GL_DEPTH_TEST);    // Разрешение теста глубины
glDepthFunc(GL_LEQUAL);     // Тип теста глубины
 glEnable(GL_LIGHT0);        // Быстрое простое освещение
                         // (устанавливает в качестве источника освещения Light0)
 glEnable(GL_LIGHTING);      // Включает освещение
 // Действительно хорошие вычисления перспективы
 glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
```

Разрешение наложения 2D текстуры, и выбор текстуры номер один. При этом будет отображена текстура номер один на любой 3D объект, который мы выводим на экран. Если Вы хотите большего контроля, Вы можете разрешать и запрещать наложение текстуры самостоятельно.

```
 glEnable(GL_TEXTURE_2D); // Разрешение наложения текстуры
 glBindTexture(GL_TEXTURE_2D, texture[0]); // Выбор текстуры
 return TRUE; // Инициализация окончена успешно
}
```

Код изменения размера не изменился, но код DrawGLScene изменился.

```
int DrawGLScene(GLvoid) // Здесь мы будем рисовать все
{
 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Очистка экран и буфера глубины
 glLoadIdentity(); // Сброс просмотра
```

Здесь наше первое изменение. Вместо того чтобы поместить объект в середину экрана, мы собираемся вращать его на экране, используя COS и SIN (это не сюрприз). Мы перемещаемся на 3 единицы в экран (-3.0f). По оси X, мы будем раскачиваться от -1.1 слева до +1.1 вправо. Мы будем использовать переменную rot для управления раскачиванием слева направо. Мы будем раскачивать от +0.8 верх до -0.8 вниз. Мы будем использовать переменную rot для этого раскачивания также (можно также задействовать и другие переменные).

```
 // Позиция текста
 glTranslatef(1.1f*float(cos(rot/16.0f)),0.8f*float(sin(rot/20.0f)),-3.0f);
```

Теперь сделаем вращения. Символ будет вращаться по осям X, Y и Z.

```
 glRotatef(rot,1.0f,0.0f,0.0f);        // Вращение по оси X
 glRotatef(rot*1.2f,0.0f,1.0f,0.0f);   // Вращение по оси Y
 glRotatef(rot*1.4f,0.0f,0.0f,1.0f);   // Вращение по оси Z
```

Мы смещаем символ по каждой оси немного налево, вниз, и вперед, чтобы центрировать его. Иначе, когда он вращается, он будет вращаться не вокруг собственного центра. (-0.35f, -0.35f, 0.1f) те числа, которые подходят. Я потратил некоторое время, прежде чем подобрал их, и они могут изменяться в зависимости от шрифта. Почему шрифты построены не вокруг центральной точки, я не знаю.

```
 glTranslatef(-0.35f,-0.35f,0.1f);      // Центр по осям X, Y, Z
```

Наконец мы выводим наш эмблемы смерти, затем увеличиваем переменную rot, поэтому наш символ вращается и перемещается по экрану. Если Вы не можете понять, почему я получаю череп из символа 'N', сделайте так: запустите Microsoft Word или Wordpad. Вызовите ниспадающие меню шрифтов. Выберите wingdings шрифт. Наберите в верхнем регистре 'N'. Появиться эмблема смерти.

```
  glPrint("N"); // Нарисуем символ эмблемы смерти
  rot+=0.1f;    // Увеличим переменную вращения
  return TRUE;  // Покидаем эту процедуру
}
```

Последнее, что надо сделать добавить KillFont() в конце KillGLWindow() точно так, как показано ниже. Важно добавить эту строку. Это почистит память прежде, чем мы выйдем из нашей программы.

```
  if (!UnregisterClass("OpenGL",hInstance))    // Если класс не зарегистрирован
  {
    MessageBox(NULL,"Could Not Unregister Class.","SHUTDOWN ERROR",MB_OK | MB_ICONINFORMATION);
    hInstance=NULL;          // Установить копию приложения в ноль
  }
  KillFont();            // Уничтожить шрифт
}
```

Даже притом, что я не вдавался в излишние подробности, Вы должны получить хорошее понимание о том, как заставить OpenGL генерировать координаты текстуры. У Вас не должно возникнуть никаких проблем при наложении текстур на шрифты, или даже на другие объекты. И, изменяя только две строки кода, Вы можете разрешить сферическое наложение, которое является действительно крутым эффектом

Урок 16 по OpenGL. Эффект тумана на OpenGL

Cool Looking Fog

Этот урок представляет Крис Алиотта (Chris Aliotta)...

Итак, Вы хотите добавить туман в Вашу программу на OpenGL? Что ж, в этом уроке я покажу как сделать именно это. Я первый раз пишу урок, и относительно недавно познакомился с программированием на OpenGL/C++, так что пожалуйста, если вы найдете здесь какие-нибудь ошибки, то пожалуйста, сообщите мне и не накидывайтесь все сразу. Эта программа основана на примере седьмого урока.

Подготовка данных:

Начнем с того, что подготовим все необходимые переменные, содержащие параметры затуманивания. Массив fogMode будет хранить три значения: GL_EXP, GL_EXP2 и GL_LINEAR, - это три типа тумана. Позже я объясню различия между ними. Объявим переменные в начале кода, после строки GLuint texture[3]. В переменной fogfilter будет храниться какой тип тумана мы будем использовать, fogColor будет содержать цвет, который мы хотим придать туману. Еще я добавил двоичную переменную gp в начало кода, чтобы можно было узнать нажата ли клавиша 'g' во время выполнения программы-примера.

```
bool   gp;                      // G Нажата? ( Новое )
GLuint filter;                  // Используемый фильтр для текстур
GLuint fogMode[]= { GL_EXP, GL_EXP2, GL_LINEAR }; // Хранит три типа тумана
GLuint fogfilter= 0;            // Тип используемого тумана
GLfloat fogColor[4]= {0.5f, 0.5f, 0.5f, 1.0f}; // Цвет тумана
```

Изменения в DrawGLScene
Теперь, когда мы задали переменные, передвинемся вниз к функции InitGL. Строка glClearColor() изменена так, чтобы заполнить экран цветом тумана для достижения лучшего эффекта. Требуется не так уж много кода, чтобы задействовать туман. Вообщем, Вам все это покажется очень простым.

```
glClearColor(0.5f,0.5f,0.5f,1.0f);      // Будем очищать экран, заполняя его цветом тумана. ( Изменено )
glEnable(GL_FOG);                       // Включает туман (GL_FOG)
glFogi(GL_FOG_MODE, fogMode[fogfilter]);// Выбираем тип тумана
glFogfv(GL_FOG_COLOR, fogColor);        // Устанавливаем цвет тумана
```

```
glFogf(GL_FOG_DENSITY, 0.35f);          // Насколько густым будет туман
glHint(GL_FOG_HINT, GL_DONT_CARE);      // Вспомогательная установка тумана
glFogf(GL_FOG_START, 1.0f);             // Глубина, с которой начинается туман
glFogf(GL_FOG_END, 5.0f);               // Глубина, где туман заканчивается.
```

Возьмем сначала первые три строчки этого кода. Первая строка glEnable(GL_FOG) во многом говорит сама за себя. Ее задача - инициализировать туман.
Вторая строка, glFogi(GL_FOG_MODE, fogMode[fogfilter]) устанавливает режим фильтрации тумана. Ранее мы объявили массив fogMode. Он содержал GL_EXP, GL_EXP2 и GL_LINEAR. Именно здесь эти переменные входят в игру. Вот что каждая из них значит:

· GL_EXP - Обычный туман, заполняющий весь экран. Во многом он напоминает туман отдаленно, но легко справляется со своей работой даже на старых PC.
· GL_EXP2 - Это следующий шаг после GL_EXP. Затуманит весь экран, за то придает больше глубины всей сцене.
· GL_LINEAR - Это лучший режим прорисовки тумана. Объекты выходят из тумана и исчезают в нем гораздо лучше.

Третья, glFogfv(GL_FOG_COLOR, fogcolor) задает цвет тумана. Раньше мы задали его как (0.5f,0.5f,0.5f,1.0f) через переменную fogcolor - получился приятный серый цвет.

Дальше, посмотрим на четыре последних строки. Строка glFogf(GL_FOG_DENSITY, 0.35f) устанавливает, насколько густым будет туман. Увеличьте число, и туман станет более густым, уменьшите - менее густым.

Eric Desrosiers добавляет небольшое объяснение glHint(GL_FOG_HINT, hintval):

hintval может быть: GL_DONT_CARE, GL_NICEST или GL_FASTEST

· GL_DONT_CARE - позволяет OpenGL выбрать формулу для расчета тумана (по вершинам или по пикселям).
· GL_NICEST - Создает туман по пикселям (хорошо смотрится).
· GL_FASTEST - Вычисляет туман по вершинам (быстрее, но не так красиво)) .

Следующая строка glFogf(GL_FOG_START, 1.0f) устанавливает насколько близко к экрану начинается затуманивание. Вы можете изменить это число на что угодно, в зависимости от того, где бы Вы хотели, чтобы начался туман. Следующая, похожая, строка glFogf(GL_FOG_END, 5.0f) сообщает программе OpenGL насколько глубоко в экран должен уходить туман.

События при нажатии клавиш

Сейчас, когда код прорисовки уже готов, мы добавим команды для клавиатуры, чтобы переключаться между разными способами затуманивания. Этот код идет в конце программы, вместе с обработкой нажатия клавиш.

```
if (keys['G'] && !gp)                   // Нажата ли клавиша "G"?
{
    gp=TRUE;                            // gp устанавливаем в TRUE
    fogfilter+=1;                       // Увеличиние fogfilter на 1
    if (fogfilter>2)                    // fogfilter больше 2 ... ?
    {
        fogfilter=0;                    // Если так, установить fogfilter в ноль
    }
    glFogi (GL_FOG_MODE, fogMode[fogfilter]); // Режим тумана
}
if (!keys['G'])                         // Клавиша "G" отпущена?
{
    gp=FALSE;                           // Если да, gp установить в FALSE
}
```

Вот и все! Мы закончили. В Ваших программах с OpenGL есть туман. Я бы даже сказал, что это было достаточно безболезненно. Если есть какие вопросы или комментарии, легко можете со мной связаться: chris@incinerated.com. Так же заходите ко мне на сайт: http://www.incinerated.com/ и http://www.incinerated.com/precursor.

Урок 17 по OpenGL. Двухмерные шрифты из текстур

2D Texture Font

Этот урок написан NeHe & Giuseppe D'Agat.

Я знаю, что все устали от шрифтов. Те уроки, которые уже были рассмотрены ранее, не только показывали текст, но они отображали 3-х мерный текст, текстурированный текст, и могли быть привязаны к переменным. Но что будет, если вы перенесете свой проект на машину, которая не поддерживает Bitmap или Outline шрифты?
Благодаря Guiseppe D'Agata у нас есть еще один урок со шрифтами. Вы спросите, что же еще осталось? Если вы помните, в первом уроке про шрифты, я упоминал об использовании текстур для рисования букв на экран. Обычно, когда вы используете текстуры для рисования текста на экране, вы загружаете свою любимую программу рисования, выбираете шрифт и набираете букву или фразу, которую хотите отобразить на экране. Дальше вы сохраняете изображение и загружаете его в свою программу, как текстуру. Это не очень эффективно для программ, которые используют большое количество текста, или текст, который непрерывно меняется!

Эта программа использует только одну текстуру для отображения любого из 256 различных символов на экран. Имейте в виду, что каждый символ всего лишь 16 пикселов в ширину и 16 в высоту. Если взять стандартную текстуру 256*256, легко заметить, что в ней можно разместить только 16 символов поперек и получится 16 строк. Если нужно более детальное объяснение то: текстура 256 пикселов в ширину, а символ 16 пикселов в ширину. 256 делим на 16, получаем 16 :)

Итак. Давайте сделаем демонстрационную программу 2-х мерных шрифтов. Эта программа дополняет код из первого урока. В первой части программы мы включим библиотеки math и stdio. Математическая библиотека нужна, чтобы двигать буквы по экрану, используя синус и косинус, а библиотека stdio нужна, чтобы убедиться в том, что файлы картинок, которые мы загружаем действительно существуют, перед тем как мы попытаемся сделать из них текстуры.

```
#include <windows.h>        // Заголовочный файл для Windows
#include <math.h>           // Заголовочный файл для математической
                            // библиотеки Windows  (Добавлено)
#include <stdio.h>          // Заголовочный файл для стандартной библиотеки
                            //ввода/вывода (Добавлено)
#include <gl\gl.h>          // Заголовочный файл для библиотеки OpenGL32
#include <gl\glu.h>         // Заголовочный файл для библиотеки GLu32
#include <gl\glaux.h>       // Заголовочный файл для библиотеки GLaux

HDC             hDC=NULL;   // Приватный контекст устройства GDI
HGLRC           hRC=NULL;   // Постоянный контекст визуализации
HWND            hWnd=NULL;  // Сохраняет дескриптор окна
HINSTANCE       hInstance;  // Сохраняет экземпляр приложения

bool    keys[256];          // Массив для работы с клавиатурой
bool    active=TRUE;        // Флаг активации окна, по умолчанию = TRUE
bool    fullscreen=TRUE;    // Флаг полноэкранного вывода
```

Сейчас мы добавим переменную base указывающую на наш список отображения. Так же мы добавим texture[2], для хранения 2-х текстур, используемых для создания простого 3-х мерного объекта.

Добавим также переменную loop, которую будем использовать для циклов. И, наконец, cnt1 и cnt2, которые будут использоваться для движения текста по экрану и для вращения простого 3-х мерного объекта.

```
GLuint  base;       // Основной список отображения для шрифта
GLuint  texture[2]; // Место для текстуры нашего шрифта
GLuint  loop;       // Общая переменная для циклов
GLfloat cnt1;       // Первый счетчик для движения и раскрашивания текста
GLfloat cnt2;       // Второй счетчик для движения и раскрашивания текста

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);   // Объявление WndProc
```

Теперь код для загрузки текстуры. Он точно такой же, как в предыдущих уроках по текстурированию.

```
AUX_RGBImageRec *LoadBMP(char *Filename) // Загрузка изображения
{
    FILE *File=NULL;                // Дескриптор файла
    if (!Filename)                  // Удостоверимся, что имя файла передано
    {
        return NULL;                // Если нет, возвратим NULL
    }
    File=fopen(Filename,"r");       // Проверка, существует ли файл
    if (File)                       // Существует?
    {
        fclose(File);               // Закрываем файл
        // Загружаем изображение и возвращаем указатель
        return auxDIBImageLoad(Filename);
    }
    return NULL;                    // Если загрузка не удалась, возвращаем NULL
}
```

Данный код тоже немного изменился, в отличие от кода в предыдущих уроках. Если вы не уверены в том, для чего каждая строка, вернитесь и просмотрите предыдущие примеры заново.
Отметим, что TextureImage[] будет хранить 2 записи о rgb изображении. Очень важно дважды проверить код, который работает с загрузкой и сохранением текстур. Одно неверное число может привести к зависанию!

```
int LoadGLTextures()                    // Загрузка и преобразование текстур
{
    int Status=FALSE;                   // Индикатор статуса
    AUX_RGBImageRec *TextureImage[2];   // Место хранения для текстур
```

Следующая строка самая важная. Если изменить 2 на любое другое число, точно возникнут проблемы. Проверьте дважды! Это число должно совпадать с тем, которое вы используете, когда определяете TextureImage[].

Две текстуры, которые мы загрузим, будут font.bmp (наш шрифт) и bumps.bmp. Вторая текстура может быть любой, какую вы захотите. Я не очень творческий человек, поэтому я решил воспользоваться простой текстурой.

```
memset(TextureImage,0,sizeof(void *)*2);        // Устанавливаем указатель в NULL
if ((TextureImage[0]=LoadBMP("Data/Font.bmp")) &&// Загружаем изображение шрифта
(TextureImage[1]=LoadBMP("Data/Bumps.bmp")))    // Загружаем текстуру
    {
        Status=TRUE;                    // Устанавливаем статус в TRUE
```

Другая важная строка, на которую нужно посмотреть дважды. Я не могу сказать, сколько писем я получил от людей, спрашивавших "почему я вижу только одну текстуру, или почему моя текстура вся белая!?!". Обычно проблема в этой строке. Если заменить 2 на 1, будет создана только одна текстура, а вторая будет в виде белой текстуры. Если заменить 2 на 3, то программа может зависнуть!

Вы должны вызывать glGenTextures() один раз. После вызова glGenTexture, необходимо сгенерировать все ваши текстуры. Я видел людей, которые вставляют вызов glGenTextures() перед созданием каждой текстуры. Обычно они ссылаются на то, что новая текстура перезаписывает все уже созданные текстуры. Было бы неплохо, сначала решить, сколько текстур необходимо сделать, а затем вызвать один раз glGenTextures(), а потом создать все текстуры. Не хорошо помещать вызов glGenTextures() в цикл без причины.

```
glGenTextures(2, &texture[0]);          // Создание 2-х текстур
for (loop=0; loop<2; loop++)            // Цикл для всех текстур
{           // Создание всех текстур
        glBindTexture(GL_TEXTURE_2D, texture[loop]);
        glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
        glTexImage2D(GL_TEXTURE_2D, 0, 3,
TextureImage[loop]->sizeX, TextureImage[loop]->sizeY, 0,
GL_RGB, GL_UNSIGNED_BYTE, TextureImage[loop]->data);
}
}
```

Следующие стоки кода проверяют, занимает ли загруженное нами rgb изображение для создания текстуры память. Если да, то высвобождаем ее. Заметьте, мы проверяем и освобождаем обе записи для изображений. Если мы используем три различные изображения для текстур, то необходимо проверить и освободить память из-под 3-х изображений.

```
for (loop=0; loop<2; loop++)
    {
    if (TextureImage[loop])             // Если текстура существует
        {
        if (TextureImage[loop]->data)   // Если изображение текстуры существует
        {
            // Освобождаем память от изображения текстуры
             free(TextureImage[loop]->data);
        }
        free(TextureImage[loop]);   // Освобождаем память от структуры изображения
        }
    }
return Status;                          // Возвращаем статус
}
```

Теперь создадим сам наш шрифт. Я пройду эту секцию с особо детальным описанием. Это не очень сложно, но там есть немного математики, которую нужно понять, а я знаю, что математика не всем нравится.

```
GLvoid BuildFont(GLvoid)                // Создаем список отображения нашего шрифта
{
```

Следующие две переменные будем использовать для сохранения позиции каждой буквы внутри текстуры шрифта. cx будет содержать позицию в текстуре по горизонтали, а cy содержать позицию по вертикали.

```
    float    cx;                    // Содержит X координату символа
    float    cy;                    // Содержит Y координату символа
```

Дальше мы скажем OpenGL, что хотим 256 списков отображения. Переменная base будет указывать на положение первого списка отображения. Второй будет base+1, третий base+2, и т.д. Вторая строка кода ниже, выбирает нашу текстуру шрифта (texture[0]).

```
    base=glGenLists(256);               // Создаем списки
    glBindTexture(GL_TEXTURE_2D, texture[0]);// Выбираем текстуру шрифта
```

Дальше мы начнем наш цикл. В цикле создадим 256 символов, сохраняя каждый символ в своем собственном списке отображения.

```
    for (loop=0; loop<256; loop++)      // Цикл по всем 256 спискам
    {
```

Первая строка ниже, может показаться загадочной. Символ % означает остаток от деления loop на 16. cx будет двигаться по текстуре шрифта слева направо. Позже вы заметите в коде, что мы вычитаем из 1 cy, чтобы двигаться сверху вниз, вместо того, чтобы двигаться снизу вверх. Символ % довольно трудно объяснить, но я попробую.

Все, о чем мы говорим, (loop % 16) /16.0f просто переводит результат в координаты текстуры. Поэтому, если loop было равно 16, cx будет равно остатку от деления 16 на 16, то есть 0. А cy равно 16/16, то есть 1. Поэтому мы двигаемся вниз на высоту одного символа, и совсем не двигаемся вправо. Теперь, если loop равно 17, cx будет равно 17/16, что равно 1.0625. Остаток .0625 или 1/16-ая. Это значит, что мы двигаемся на один символ вправо. cy все еще будет равно 1, потому что нам важны только знаки слева от десятичной запятой. 18/16 даст нам 2/16, двигая на 2 символа вправо, и все еще на один вниз. Если loop равно 32, cx будет опять 0, потому что остатка от деления нет, когда делим 32 на 16, но cy равно 2. Потому что число слева от десятичной запятой будет 2, двигая нас на 2 символа вниз от самого верха текстуры шрифта. Не так ли?

```
    cx=float(loop%16)/16.0f;        // X координата текущего символа
    cy=float(loop/16)/16.0f;        // Y координата текущего символа
```

Вау! Ок. Итак, теперь мы построим наш 2D шрифт, выбирая каждый символ из текстуры шрифта, в зависимости от значений cx и cy. В строках ниже мы добавим loop к значению base, если мы этого не сделаем, то каждая буква будет построена в первом дисплейном списке. Мы точно не хотим, чтобы это случилось, поэтому добавим loop к base и каждый следующий символ, который мы создаем, сохранится в следующем доступном списке отображения.

```
glNewList(base+loop,GL_COMPILE); // Начинаем делать список
```

Теперь, когда мы выбрали список отображения, который мы построили, мы создадим символ. Этого мы добьемся, создавая четырехугольник, и затем текстурируя его одним символом из текстуры шрифта.

```
glBegin(GL_QUADS);              // Используем четырехугольник, для каждого символа
```

cx и cy будут содержать очень маленькие значения от 0.0f до 1.0f. Оба они будут равны 0 в первой строчке кода ниже, а именно: glTexCoord2f(0.0f,1-0.0f-0.0625f). Помните, что 0.0625 это 1/16-ая нашей текстуры, или ширина/высота одного символа. Координаты текстуры ниже будут координатами левой нижней точки текстуры.

Заметьте, мы используем glVertex2i(x, y) вместо glVertex3f(x, y, z). Наш шрифт – это двумерный шрифт, поэтому нам не нужна координата z. Поскольку мы используем плоский экран (Ortho screen – ортографическая или параллельная проекция), нам не надо сдвигаться вглубь экрана. Мы должны сделать, чтобы нарисовать на плоском экране, это задать x и y координаты. Так как наш экран в пикселах от 0 до 639 и от 0 до 479, нам вообще не надо использовать плавающую точку или отрицательные значения :). Используя плоский экран, мы получаем (0, 0) в нижнем левом углу. (640, 480) будет в верхнем правом углу. 0 - левый край по оси x, 639 - правый край экрана по оси x. 0 – верхний край экрана по оси y и 479 – нижний край экрана на оси y. Проще говоря, мы избавились от отрицательных координат. Это тоже удобно для тех, кто не заботится о перспективе и предпочитает работать с пикселами больше, чем с экранными единицами.

```
glTexCoord2f(cx,1-cy-0.0625f);  // Точка в текстуре (Левая нижняя)
glVertex2i(0,0);                // Координаты вершины (Левая нижняя)
```

Следующая точка на текстуре будет 1/16-ая правее предыдущей точки (точнее ширина одного символа). Поэтому это будет нижняя правая точка текстуры.

```
// Точка на текстуре (Правая нижняя)
glTexCoord2f(cx+0.0625f,1-cy-0.0625f);
glVertex2i(16,0); // Координаты вершины (Правая нижняя)
```

Третья точка текстуры лежит в дальнем правом конце символа, но сдвинута вверх на 1/16-ую текстуры (точнее на высоту одного символа). Это будет верхняя правая точка отдельного символа.

```
glTexCoord2f(cx+0.0625f,1-cy);  // Точка текстуры (Верхняя правая)
glVertex2i(16,16);              // Координаты вершины (Верхняя правая)
```

Наконец, мы двигаемся влево, чтобы задать нашу последнюю точку в верхнем левом углу символа.

```
glTexCoord2f(cx,1-cy);  // Точка текстуры (Верхняя левая)
glVertex2i(0,16);       // Координаты вершины (Верхняя левая)
glEnd();                // Конец построения четырехугольника (Символа)
```

Наконец, мы перемещаемся на 10 пикселей вправо, двигаясь вправо по текстуре. Если мы не передвинемся, символы будут рисоваться поверх друг друга. Так как наш шрифт очень узкий, мы не будем двигаться на 16 пикселов вправо. Если мы это сделаем, то будет слишком большой пропуск между каждым символом. Двигаясь на 10 пикселей, мы уменьшаем расстояние между символами.

```
    glTranslated(10,0,0);           // Двигаемся вправо от символа
    glEndList();                    // Заканчиваем создавать список отображения
}                                   // Цикл для создания всех 256 символов
}
```

Следующая секция кода такая же как мы делали в предыдущих уроках, для освобождения списка отображения, перед выходом из программы. Все 256 экранных списков, начиная от base, будут удалены. (Это хорошо!)

```
GLvoid KillFont(GLvoid)              // Удаляем шрифт из памяти
{
    glDeleteLists(base,256);       // Удаляем все 256 списков отображения
}
```

Следующая секция кода содержит все рисование. Все довольно ново, поэтому я постараюсь объяснить каждую строчку особенно детально. Одно маленькое замечание: можно добавить переменные для поддержки размеров, пропусков, и кучу проверок для восстановления настроек которые были до того, как мы решили их напечатать.

glPrint() имеет четыре параметра. Первый это координата x на экране (позиция слева на право). Следующая это y координата на экране (сверху вниз... 0 внизу, большие значения наверху). Затем нашу строку string (текст, который мы хотим напечатать), и, наконец, переменную set. Если посмотреть на картинку, которую сделал Giuseppe D'Agata, можно заметить, что там два разных набора символов. Первый набор - обычные символы, а второй набор - наклонные. Если set = 0, то выбран первый набор. Если set = 1 или больше, то выбран второй набор символов.

```
GLvoid glPrint(GLint x, GLint y, char *string, int set) // Где печатать
{
```

Первое, что мы сделаем - это проверим, что set от 0 до 1. Если set больше 1, то присвоим ей значение 1.

```
    if (set>1)                  // Больше единицы?
    {
        set=1;                  // Сделаем Set равное единице
    }
```

Теперь выберем нашу текстуру со шрифтом. Мы делаем это только, если раньше была выбрана другая текстура, до того как мы решили печатать что-то на экране.

```
    glBindTexture(GL_TEXTURE_2D, texture[0]);       // Выбираем нашу текстуру шрифта
```

Теперь отключим проверку глубины. Причина того, почему я так делаю, в том, что смешивание работает приятнее. Если не отменить проверку глубины, то текст может проходить за каким-нибудь объектом, или смешивание может выглядеть неправильно. Если вы не хотите смешивать текст на экране (из-за смешивания, т.е прозрачности черный фон вокруг символов не виден) можете оставить проверку глубины.

```
    glDisable(GL_DEPTH_TEST);               // Отмена проверки глубины
```

Следующее несколько строк очень важны! Мы выбираем нашу матрицу проекции. Прямо после этого мы используем команду glPushMatrix(). glPushMatrix сохраняет текущую матрицу проекции. Похоже на кнопку "память" на калькуляторе.

```
    glMatrixMode(GL_PROJECTION);            // Выбираем матрицу проекции
    glPushMatrix();                         // Сохраняем матрицу проекции
```

Теперь, когда наша матрица сохранена, мы сбрасываем ее и устанавливаем плоский экран. Первое и третье число (0) задают нижний левый угол экрана. Мы можем сделать левую сторону экрана -640, если захотим, но зачем нам работать с отрицательными числами, если это не нужно. Второе и четвертое число задают верхний правый угол экрана. Неплохо установить эти значения равными текущему разрешению. Глубины нет, поэтому устанавливаем значения z в -1 и 1.

```
    glLoadIdentity();                       // Сбрасываем матрицу проекции
    glOrtho(0,640,0,480,-1,1);              // Устанавливаем плоский экран
```

Теперь выбираем нашу матрицу просмотра модели и сохраняем текущие установки, используя glPushMatrix(). Далее сбрасываем матрицу просмотра модели, так что можно работать, используя ортографическую проекцию.

```
    glMatrixMode(GL_MODELVIEW);             // Выбираем матрицу модели просмотра
    glPushMatrix();                         // Сохраняем матрицу модели просмотра
    glLoadIdentity();                       // Сбрасываем матрицу модели просмотра
```

Сохранив настройки перспективы и установив плоский экран, можно рисовать текст. Начнем с перемещения в позицию на экране, где мы хотим нарисовать текст. Используем glTranslated() вместо glTranslatef(), так как мы

работаем с пикселами, поэтому точки с дробными значениями не имеют смысла. В конце концов, нельзя использовать половину пиксела :).

```
    glTranslated(x,y,0);                 // Позиция текста (0,0 - Нижняя левая)
```

Строка ниже выбирает, каким набором символов мы хотим воспользоваться. Если мы хотим использовать второй набор символов, то добавляем 128 к base (128 - половина 256 символов). Добавляя 128, мы пропускаем первые 128 символов.

```
    glListBase(base-32+(128*set));       // Выбираем набор символов (0 или 1)
```

Сейчас, все что осталось - это нарисовать символы на экране. Делаем это так же как во всех других уроках со шрифтами. Используем glCallLists(). strlen(string) это длина строки (сколько символов мы хотим нарисовать), GL_BYTE означает то, что каждый символ представляется одним байтом (байт это любое значение от 0 до 255). Наконец, string содержит тот текст, который надо напечатать на экране.

```
    glCallLists(strlen(string),GL_BYTE,string); // Рисуем текст на экране
```

Все, что надо теперь сделать, это восстановить перспективу. Мы выбираем матрицу проектирования и используем glPopMatrix(), чтобы восстановить установки, сохраненные с помощью glPushMatrix(). Важно восстановить их в обратном порядке, в том в котором мы их сохраняли.

```
    glMatrixMode(GL_PROJECTION);         // Выбираем матрицу проекции
    glPopMatrix();                       // Восстанавливаем старую матрицу проекции
```

Теперь мы выбираем матрицу просмотра модели и делаем то же самое. Мы используем glPopMatrix(), чтобы восстановить нашу матрицу просмотра модели, на ту, которая была, прежде чем мы устанавливали плоский экран.

```
    glMatrixMode(GL_MODELVIEW);          // Выбираем матрицу просмотра модели
    glPopMatrix();                       // Восстанавливаем старую матрицу проекции
```

Наконец, разрешаем проверку глубины. Если мы не запрещали проверку глубины в коде раньше, то нам не нужна эта строка.

```
    glEnable(GL_DEPTH_TEST);             // Разрешаем тест глубины
}
```

В ReSizeGLScene() ничего менять не надо, так что переходим к InitGL().

```
int InitGL(GLvoid)                       // Все установки для OpenGL здесь
{
```

Переходим к коду построения текстур. Если построить текстуры не удалось по какой-либо причине, то возвращаем значение FALSE. Это позволит нашей программе узнать, что произошла ошибка, и программа изящно завершится.

```
    if (!LoadGLTextures()) // Переходим к загрузке текстуры
    {
        return FALSE;   // Если текстура не загрузилась - возвращаем FALSE
    }
```

Если ошибок не было, переходим к коду построения шрифта. Т.к. ничего не может случиться при построении шрифта, поэтому проверку ошибок не включаем.

```
    BuildFont();  // Создаем шрифт
```

Теперь делаем обычную настройку GL. Мы установим черный цвет фона для очистки, зададим значение глубины в 1.0. Выбираем режим проверки глубины вместе со смешиванием. Мы разрешаем сглаженное заполнение и, наконец, разрешаем 2-мерное текстурирование.

```
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f); // Очищаем фон черным цветом
    glClearDepth(1.0);                   // Очистка и сброс буфера глубины
    glDepthFunc(GL_LEQUAL);              // Тип теста глубины
    glBlendFunc(GL_SRC_ALPHA,GL_ONE);    // Выбор типа смешивания
```

```
    glShadeModel(GL_SMOOTH);            // Сглаженное заполнение
    glEnable(GL_TEXTURE_2D);            // 2-мерное текстурирование
    return TRUE;                        // Инициализация прошла успешно
}
```

Следующая секция кода создает сцену. Мы рисуем сначала 3-х мерный объект и потом текст, поэтому текст будет поверх 3-х мерного объекта, вместо того, чтобы объект закрывал текст сверху. Причина, по которой я добавил 3-х мерный объект, заключается в том, чтобы показать, что перспективная и ортографическая проекции могут быть использованы одновременно.

```
int DrawGLScene(GLvoid)                 // Здесь мы рисуем все объекты
{
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Очистка экрана и буфера глубины
glLoadIdentity();                       // Сброс матрицы просмотра модели
```

Мы выбрали нашу bumps.bmp текстуру, и сейчас можно построить маленький 3-х мерный объект. Мы сдвигаемся на 5 экранных единиц вглубь экрана, так чтобы можно было увидеть объект. Мы вращаем объект на 45 градусов вдоль оси z. Это повернет наш четырехугольник на 45 градусов по часовой стрелке, и он будет больше похож на ромб, чем на прямоугольник.

```
glBindTexture(GL_TEXTURE_2D, texture[1]);// Выбираем вторую текстуру
glTranslatef(0.0f,0.0f,-5.0f);          // Сдвигаемся на 5 единиц вглубь экрана
    glRotatef(45.0f,0.0f,0.0f,1.0f); // Поворачиваем на 45 градусов (по часовой стрелке)
```

После поворота на 45 градусов, мы вращаем объект вокруг осей x и y, с помощью переменной cnt1*30. Это заставляет наш объект вращаться вокруг своей оси, подобно алмазу.

```
    glRotatef(cnt1*30.0f,1.0f,1.0f,0.0f); // Вращение по X & Y на cnt1 (слева направо)
```

Отменяем смешивание (мы хотим, чтобы 3-х мерный объект был сплошным), и устанавливаем цвет на ярко белый. Затем рисуем один текстурированный четырехугольник.

```
    glDisable(GL_BLEND);            // Отменяем смешивание перед рисованием 3D
    glColor3f(1.0f,1.0f,1.0f);      // Ярко белый
    glBegin(GL_QUADS);              // Рисуем первый текстурированный прямоугольник
    glTexCoord2d(0.0f,0.0f);        // Первая точка на текстуре
        glVertex2f(-1.0f, 1.0f); // Первая вершина
         glTexCoord2d(1.0f,0.0f); // Вторая точка на текстуре
         glVertex2f( 1.0f, 1.0f); // Вторая вершина
        glTexCoord2d(1.0f,1.0f); // Третья точка на текстуре
         glVertex2f( 1.0f,-1.0f); // Третья вершина
         glTexCoord2d(0.0f,1.0f); // Четвертая точка на текстуре
         glVertex2f(-1.0f,-1.0f); // Четвертая вершина
    glEnd();                    // Заканчиваем рисование четырехугольника
```

Сразу, после того как мы нарисовали первый четырехугольник, мы поворачиваемся на 90 градусов по осям x и y. Затем рисуем другой четырехугольник. Второй четырехугольник проходит сквозь середину первого, в результате получается красивая фигура.

```
// Поворачиваемся по X и Y на 90 градусов (слева на право)
glRotatef(90.0f,1.0f,1.0f,0.0f);
glBegin(GL_QUADS);              // Рисуем второй текстурированный четырехугольник
         glTexCoord2d(0.0f,0.0f); // Первая точка на текстуре
         glVertex2f(-1.0f, 1.0f); // Первая вершина
         glTexCoord2d(1.0f,0.0f); // Вторая точка на текстуре
         glVertex2f( 1.0f, 1.0f); // Вторая вершина
         glTexCoord2d(1.0f,1.0f); // Третья точка на текстуре
         glVertex2f( 1.0f,-1.0f); // Третья вершина
         glTexCoord2d(0.0f,1.0f); // Четвертая точка на текстуре
         glVertex2f(-1.0f,-1.0f); // Четвертая вершина
    glEnd();                    // Заканчиваем рисовать четырехугольник
```

После того как нарисованы четырехугольники, разрешаем смешивание и рисуем текст.

```
    glEnable(GL_BLEND);         // Разрешаем смешивание
    glLoadIdentity();           // Сбрасываем просмотр
```

Мы используем такой же код для раскрашивания, как в предыдущих примерах с текстом. Цвет меняется постепенно по мере движения текста по экрану.

```
   // Изменение цвета основывается на положении текста
    glColor3f(1.0f*float(cos(cnt1)),1.0f*float(sin(cnt2)),
        1.0f-0.5f*float(cos(cnt1+cnt2)));
```

Затем мы рисуем текст. Мы все еще используем glPrint(). Первый параметр - это координата x. Второй - координата y. Третий параметр ("NeHe") - текст, который надо написать на экране, и последний это набор символов (0 - обычный, 1 - наклонный).

Как вы могли заметить, мы двигаем текст по экрану используя SIN и COS, используя счетчики cnt1 и cnt2. Если вы не понимаете, что делают SIN и COS, вернитесь и прочитайте тексты предыдущих уроков.

```
    // Печатаем GL текст на экране
    glPrint(int((280+250*cos(cnt1))),int(235+200*sin(cnt2)),"NeHe",0);
    glColor3f(1.0f*float(sin(cnt2)),
        1.0f-0.5f*float(cos(cnt1+cnt2)),1.0f*float(cos(cnt1)));
    // Печатаем GL текст на экране
    glPrint(int((280+230*cos(cnt2))),int(235+200*sin(cnt1)),"OpenGL",1);
```

Мы устанавливаем темно синий цвет и пишем имя автора внизу экрана. Затем мы пишем его имя на экране опять, используя ярко белые символы.

```
    glColor3f(0.0f,0.0f,1.0f);      // Устанавливаем синий цвет
    glPrint(int(240+200*cos((cnt2+cnt1)/5)),
        2,"Giuseppe D'Agata",0);  // Рисуем текст на экране
    glColor3f(1.0f,1.0f,1.0f);      // Устанавливаем белый цвет
    glPrint(int(242+200*cos((cnt2+cnt1)/5)),
        2,"Giuseppe D'Agata",0);  // Рисуем смещенный текст
```

Последнее, что мы сделаем - это добавим обоим счетчикам разные значения. Это заставит текст двигаться и вращаться как 3-х мерный объект.

```
    cnt1+=0.01f;            // Увеличим первый счетчик
    cnt2+=0.0081f;          // Увеличим второй счетчик
    return TRUE;            // Все прошло успешно
}
```

Код в KillGLWindow(), CreateGLWindow() и WndProc() не изменился, поэтому пропустим его.

```
int WINAPI WinMain( HINSTANCE       hInstance,          // Экземпляр
                    HINSTANCE       hPrevInstance,      // Предыдущий экземпляр
                    LPSTR           lpCmdLine,          // Параметры командной строки
                    int             nCmdShow)           // Стиль вывода окна
{
    MSG         msg;        // Структура сообщения
    BOOL        done=FALSE;     // Переменная для выхода из цикла

    // Спрашиваем у пользователя, какой режим он предпочитает

    if (MessageBox(NULL,"Would You Like To Run In Fullscreen Mode?",
        "Start FullScreen?",MB_YESNO|MB_ICONQUESTION)==IDNO) {
        fullscreen=FALSE;       // Режим окна
    }
```

Сменилось название окна.

```
    // Создаем окно OpenGL
    if (!CreateGLWindow(
        "NeHe & Giuseppe D'Agata's 2D Font Tutorial",640,480,16,fullscreen))
    {
        return 0;          // Окно не создалось - выходим
    }
    while(!done)             // Цикл пока done=FALSE
    {
        if (PeekMessage(&msg,NULL,0,0,PM_REMOVE)) // Пришло сообщение?
        {
            if (msg.message==WM_QUIT)          // Это сообщение о выходе?
            {
                done=TRUE; // Если да, то done=TRUE
            }
            else             // Если нет, то обрабатываем сообщение
            {
                TranslateMessage(&msg);// Переводим сообщение
                DispatchMessage(&msg); // Отсылаем сообщение
            }
        }
        else                 // Нет сообщений
        {
        // Рисуем сцену.  Ждем клавишу ESC или сообщение о выходе из DrawGLScene()
        // Активно?  Было сообщение о выходе?
            if ((active && !DrawGLScene()) || keys[VK_ESCAPE])
            {
                done=TRUE; // ESC или DrawGLScene сообщает о выходе
            }
            else             // Не время выходить, обновляем экран
            {
                SwapBuffers(hDC); // Меняем экраны (Двойная буферизация)
            }
        }
    }
    // Закрываем приложение
```

Последнее, что надо сделать, это добавить KillFont() в конец KillGLWindow(), как я показывал раньше. Важно добавить эту строчку. Она очищает память перед выходом из программы.

```
    if (!UnregisterClass("OpenGL",hInstance)) // Можем удалить регистрацию класса
    {
        MessageBox(NULL,"Could Not Unregister Class.","SHUTDOWN ERROR",
            MB_OK | MB_ICONINFORMATION);
        hInstance=NULL;       // Устанавливаем hInstance в NULL
    }
    KillFont();              // Уничтожаем шрифт
}
```

Я думаю, что могу официально заявить, что моя страничка теперь учит всем возможным способам, как писать текст на экране {усмешка}. В целом, я думаю это хороший урок. Код можно использовать на любом компьютере, на котором работает OpenGL, его легко использовать, и писать с помощью него текст на экране довольно просто.
Я бы хотел поблагодарить Giuseppe D'Agata за оригинальную версию этого урока. Я сильно его изменил и преобразовал в новый базовый код, но без его присланного кода, я, наверное, не смог бы написать этот урок. Его версия кода имела побольше опций, таких как пробелы между символами и т.д., но я дополнил его прикольным 3-х мерным объектом {усмешка}.

Я надеюсь, всем понравился этот урок. Если у вас есть вопросы, пишите Giuseppe D'Agata или мне.

Урок 18 по OpenGL. Квадратирование

Quadratics

Квадратирование (quadratic) - это способ отображения сложных объектов, обычно для рисования которых, нужно несколько циклов FOR и некоторые основы тригонометрии. (Прим. переводчика: квадратирование - представление сложных объектов с использованием четырехугольников).
Мы будем использовать код 7-ого урока. Мы добавим 7 переменных и изменим текстуру для разнообразия.

```
#include <windows.h>        // Заголовочный файл для Windows
#include <stdio.h>          // Заголовочный файл для стандартной библиотеки ввода/вывода
#include <gl\gl.h>          // Заголовочный файл для библиотеки OpenGL32
#include <gl\glu.h>         // Заголовочный файл для библиотеки GLu32
#include <gl\glaux.h>       // Заголовочный файл для библиотеки GLaux

HDC         hDC=NULL;     // Приватный контекст устройства GDI
HGLRC       hRC=NULL;     // Постоянный контекст рендеринга
HWND        hWnd=NULL;    // Сохраняет дескриптор окна
HINSTANCE   hInstance;    // Сохраняет экземпляр приложения

bool   keys[256];           // Массив для работы с клавиатурой
bool   active=TRUE;         // Флаг активации окна, по умолчанию = TRUE
bool   fullscreen=TRUE;     // Флаг полноэкранного вывода
bool   light;               // Освещение Вкл/Выкл
bool   lp;                  // L нажата?
bool   fp;                  // F нажата?
bool   sp;                  // Пробел нажат? ( НОВОЕ )
int    part1;               // Начало диска ( НОВОЕ )
int    part2;               // Конец диска  ( НОВОЕ )
int    p1=0;                // Приращение 1 ( НОВОЕ )
int    p2=1;                // Приращение 2 ( НОВОЕ )
GLfloat xrot;               // X вращение
GLfloat yrot;               // Y вращение
GLfloat xspeed;             // X скорость вращения
GLfloat yspeed;             // Y скорость вращения
GLfloat      z=-5.0f;       // Глубина экрана

GLUquadricObj *quadratic;    // Место для хранения объекта Quadratic ( НОВОЕ )

GLfloat LightAmbient[]=  { 0.5f, 0.5f, 0.5f, 1.0f };       // Фоновое значение света
GLfloat LightDiffuse[]=  { 1.0f, 1.0f, 1.0f, 1.0f };       // Значение рассеянного света
GLfloat LightPosition[]= { 0.0f, 0.0f, 2.0f, 1.0f };       // Позиция источника
GLuint filter;                          // Какой фильтр использовать
GLuint texture[3];                      // Место для 3-х текстур
GLuint object=0;                        // Какой объект рисовать ( НОВОЕ )

LRESULT      CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);       // Объявление WndProc
```

Ок. Теперь обратимся к InitGL(). Мы собираемся добавить 3 строчки кода, для инициализации нашего квадратичного объекта. Добавьте эти 3 строки после инициализации освещения (light1), но до строки return true. Первая строка инициализирует квадратичный объект и создает указатель на то место в памяти, где он будет содержаться. Если он не может быть создан, то будет возвращен 0. Вторая строка кода создает плавные нормали на квадратичном объекте, поэтому освещение будет выглядеть хорошо. Другое возможное значение - GL_NONE и GL_FLAT. Наконец, мы включим текстурирование на нашем квадратичном объекте.

```
    quadratic=gluNewQuadric();     // Создаем указатель на квадратичный объект ( НОВОЕ )
    gluQuadricNormals(quadratic, GLU_SMOOTH); // Создаем плавные нормали ( НОВОЕ )
     gluQuadricTexture(quadratic, GL_TRUE);    // Создаем координаты текстуры ( НОВОЕ )
```

Теперь я решил оставить куб в этом уроке, так, чтобы вы смогли увидеть, как текстура отображается на квадратичном объекте. Я решил поместить куб в отдельную функцию, поэтому, когда мы напишем функцию рисования, она станет намного проще. Все узнают этот код.

```
GLvoid glDrawCube()             // Рисование куба
{       glBegin(GL_QUADS);              // Начинаем рисовать четырехугольники
                        // Передняя сторона
      glNormal3f( 0.0f, 0.0f, 1.0f);   // Нормаль вперед
      glTexCoord2f(0.0f, 0.0f);
      glVertex3f(-1.0f, -1.0f,  1.0f); // Низ Лево на текстуре и четырехугольнике
      glTexCoord2f(1.0f, 0.0f);
      glVertex3f( 1.0f, -1.0f,  1.0f); // Низ Право на текстуре и четырехугольнике
      glTexCoord2f(1.0f, 1.0f);
      glVertex3f( 1.0f,  1.0f,  1.0f); // Верх Право на текстуре и четырехугольнике
      glTexCoord2f(0.0f, 1.0f);
      glVertex3f(-1.0f,  1.0f,  1.0f); // Верх Лево на текстуре и четырехугольнике
                        // Задняя сторона
      glNormal3f( 0.0f, 0.0f,-1.0f);   // Обратная нормаль
      glTexCoord2f(1.0f, 0.0f);
      glVertex3f(-1.0f, -1.0f, -1.0f); // Низ Право на текстуре и четырехугольнике
      glTexCoord2f(1.0f, 1.0f);
      glVertex3f(-1.0f,  1.0f, -1.0f); // Верх Право на текстуре и четырехугольнике
      glTexCoord2f(0.0f, 1.0f);
     glVertex3f( 1.0f,  1.0f, -1.0f); // Верх Лево на текстуре и четырехугольнике
      glTexCoord2f(0.0f, 0.0f);
      glVertex3f( 1.0f, -1.0f, -1.0f); // Низ Лево на текстуре и четырехугольнике
                        // Верхняя грань
      glNormal3f( 0.0f, 1.0f, 0.0f);   // Нормаль вверх
      glTexCoord2f(0.0f, 1.0f);
      glVertex3f(-1.0f,  1.0f, -1.0f); // Верх Лево на текстуре и четырехугольнике
      glTexCoord2f(0.0f, 0.0f);
      glVertex3f(-1.0f,  1.0f,  1.0f); // Низ Лево на текстуре и четырехугольнике
      glTexCoord2f(1.0f, 0.0f);
      glVertex3f( 1.0f,  1.0f,  1.0f); // Низ Право на текстуре и четырехугольнике
      glTexCoord2f(1.0f, 1.0f);
      glVertex3f( 1.0f,  1.0f, -1.0f); // Верх Право на текстуре и четырехугольнике
                        // Нижняя грань
      glNormal3f( 0.0f,-1.0f, 0.0f);   // Нормаль направлена вниз
      glTexCoord2f(1.0f, 1.0f);
      glVertex3f(-1.0f, -1.0f, -1.0f); // Верх Право на текстуре и четырехугольнике
      glTexCoord2f(0.0f, 1.0f);
      glVertex3f( 1.0f, -1.0f, -1.0f); // Верх Лево на текстуре и четырехугольнике
      glTexCoord2f(0.0f, 0.0f);
      glVertex3f( 1.0f, -1.0f,  1.0f); // Низ Лево на текстуре и четырехугольнике
      glTexCoord2f(1.0f, 0.0f);
      glVertex3f(-1.0f, -1.0f,  1.0f); // Низ Право на текстуре и четырехугольнике
                        // Правая грань
      glNormal3f( 1.0f, 0.0f, 0.0f);   // Нормаль направлена вправо
      glTexCoord2f(1.0f, 0.0f);
      glVertex3f( 1.0f, -1.0f, -1.0f); // Низ Право на текстуре и четырехугольнике
      glTexCoord2f(1.0f, 1.0f);
      glVertex3f( 1.0f,  1.0f, -1.0f); // Верх Право на текстуре и четырехугольнике
      glTexCoord2f(0.0f, 1.0f);
      glVertex3f( 1.0f,  1.0f,  1.0f); // Верх Лево на текстуре и четырехугольнике
      glTexCoord2f(0.0f, 0.0f);
      glVertex3f( 1.0f, -1.0f,  1.0f); // Низ Лево на текстуре и четырехугольнике
                        // Левая грань
      glNormal3f(-1.0f, 0.0f, 0.0f);   // Нормаль направлена влево
      glTexCoord2f(0.0f, 0.0f);   glVertex3f(-1.0f, -1.0f, -1.0f); // Низ Лево на текстуре и четырехугольнике
      glTexCoord2f(1.0f, 0.0f);
      glVertex3f(-1.0f, -1.0f,  1.0f); // Низ Право на текстуре и четырехугольнике
      glTexCoord2f(1.0f, 1.0f);
      glVertex3f(-1.0f,  1.0f,  1.0f); // Верх Право на текстуре и четырехугольнике
      glTexCoord2f(0.0f, 1.0f);
      glVertex3f(-1.0f,  1.0f, -1.0f); // Верх Лево на текстуре и четырехугольнике
      glEnd();                  // Заканчиваем рисование четырехугольника
}
```

Следующая функция - DrawGLScene. Я просто только написал case оператор для рисования разных объектов. Так же я использовал статическую переменную (локальная переменная, которая сохраняет свое значение каждый раз при вызове) для крутого эффекта, когда рисуем часть диска. Я собираюсь переписать всю функцию DrawGLScene для ясности.
Заметьте, что когда я говорю о параметрах, которые используются, я пропускаю первый параметр (quadratic). Этот параметр используется для всех объектов, которые мы рисуем, за исключением куба, поэтому я его пропускаю, когда говорю о параметрах.

```
int DrawGLScene(GLvoid)                 // Здесь мы все рисуем
{
    // Очистка видео буфера и буфера глубины
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();               // Сбрасываем вид
    glTranslatef(0.0f,0.0f,z);      // Перемещаемся вглубь экрана
    glRotatef(xrot,1.0f,0.0f,0.0f);  // Вращение по оси X
    glRotatef(yrot,0.0f,1.0f,0.0f);  // Вращение по оси Y
    glBindTexture(GL_TEXTURE_2D, texture[filter]); // Выбираем фильтрацию текстуре
    // Эта секция кода новая ( НОВОЕ )
    switch(object)                  // Проверяем, какой объект рисовать
    {
    case 0:                         // Рисуем первый объект
        glDrawCube();               // Рисуем наш куб
        break;                      // Закончили
```

Второй объект, который мы создадим, будет цилиндр. Первый параметр (1.0f) – радиус основания цилиндра (низ). Второй параметр (1.0f) - это радиус цилиндра сверху. Третий параметр (3.0f) - это высота цилиндра (какой он длины). Четвертый параметр (32) – это сколько делений будет "вокруг" оси Z, и, наконец, пятый (32) - количество делений "вдоль" оси Z. Большее количество делений приведет к увеличению детализации объекта. Увеличивая количество делений, вы добавляете больше полигонов в объект. В итоге вы должны будем пожертвовать скоростью ради качества. Самое сложное - найти золотую середину.

```
    case 1:                         // Рисуем второй объект
        glTranslatef(0.0f,0.0f,-1.5f); // Центр цилиндра
        gluCylinder(quadratic,1.0f,1.0f,3.0f,32,32); // Рисуем наш цилиндр
        break;                      // Закончили
```

Третий объект, который мы создадим, будет поверхность в виде CD диска. Первый параметр (0.5f) - внутренний радиус цилиндра. Его значение может быть нулевым, что будет означать, что внутри нет отверстия. Чем больше будет внутренний радиус - тем больше будет отверстие внутри диска. Второй параметр (1.5f) - внешний радиус. Это значение должно будь больше, чем внутренний радиус. Если сделать его значение чуть больше чем внутренний радиус, то получится тонкое кольцо. Если это значение будет намного больше, чем внутренний радиус, то получится толстое кольцо. Третий параметр (32) – количество кусочков, из которых состоит диск. Думайте об этих кусочках, как о частях пиццы. Чем больше кусочков, тем глаже будет внешняя сторона диска. И, наконец, четвертый параметр (32) - это число колец, которые составляют диск. Кольца похожи на треки на записи. Круги внутри кругов. Эти кольца делят диск со стороны внутреннего радиуса к внешнему радиусу, улучшая детализацию. Опять же, чем больше делений, тем медленнее это будет работать.

```
    case 2:                         // Рисуем третий объект
        gluDisk(quadratic,0.5f,1.5f,32,32); // Рисуем диск (в виде CD)
        break;                      // Закончили
```

Наш четвертый объект - объект, о котором я знаю то, что многие умерли, создавая его. Это сфера! Создать ее очень просто. Первый параметр - это радиус сферы. Если вы не очень знакомы с понятием радиус/диаметр и т.д., объясняю, радиус - это расстояние от центра объекта, до внешней стороны объекта. В нашем случае радиус равен 1.3f. Дальше идет количество разбиений "вокруг" оси Z (32), и количество разбиений "вдоль" оси Z (32). Большее количество придаст сфере большую гладкость. Для того, чтобы сфера была достаточно гладкой, обычно необходимо большое количество разбиений.

```
    case 3:                         // Рисуем четвертый объект
        gluSphere(quadratic,1.3f,32,32); // Рисуем сферу
        break;                      // Закончили
```

Чтобы создать наш пятый объект мы воспользуемся той же командой, что и для цилиндра. Если вы помните, когда мы создавали цилиндр, первые два параметра контролировали радиусы цилиндра сверху и снизу. Для того, чтобы сделать конус, имеет смысл сделать один из радиусов равный нулю. Это создаст точку на конце. Итак, в коде ниже мы делаем радиус на верхней стороне цилиндра равным нулю. Это создаст нашу точку, которая и сделает наш конус.

```
    case 4:                     // Рисуем пятый объект
        glTranslatef(0.0f,0.0f,-1.5f);  // Центр конуса
        // Конус с нижним радиусом .5 и высотой 2
        gluCylinder(quadratic,1.0f,0.0f,3.0f,32,32);
        break;                  // Закончили
```

Наш шестой объект создан с помощью gluParticalDisc. Объект, который мы создадим этой командой точно такой же диск, который был до этого, но у команды gluParticalDisc есть еще 2 новых параметра. Пятый параметр (part1) - это угол, с которого мы хотим начать рисование диска. Шестой параметр - это конечный угол (или угол развертки). Это угол, который мы проходим от начального. Мы будем увеличивать этот угол, что позволит постепенно рисовать диск на экране, по направлению часовой стрелки. Как только конечный угол достигнет 360 градусов, мы начнем увеличивать начальный угол. Это будет выглядеть, как будто диск начал стираться, затем мы все начнем сначала!

```
    case 5:                     // Рисуем шестой объект
        part1+=p1;              // Увеличиваем стартовый угол
        part2+=p2;              // Увеличиваем конечный угол

    if(part1>359)           // 360 градусов
        {
            p1=0;           // Хватит увеличивать начальный угол
            part1=0;        // Устанавливаем начальный угол в 0
            p2=1;           // Начинаем увеличивать конечный угол
            part2=0;        // Начиная с 0
        }
        if(part2>359)           // 360 градусов
        {
            p1=1;           // Начинаем увеличивать начальный угол
            p2=0;           // Перестаем увеличивать конечный угол
        }
                            // Диск, такой-же как в прошлый раз
        gluPartialDisk(quadratic,0.5f,1.5f,32,32,part1,part2-part1);
        break;              // Закончили
    };
    xrot+=xspeed;               // Увеличиваем угол поворота вокруг оси X
    yrot+=yspeed;               // Увеличиваем угол поворота вокруг оси Y
    return TRUE;                // Продолжаем
}
```

Теперь, в последней части, обработка клавиш. Просто добавим это, туда, где происходит проверка нажатия клавиш.

```
    if (keys[' '] && !sp)       // Нажата клавиша "пробел"?
    {
        sp=TRUE;                // Если так, то устанавливаем sp в TRUE
        object++;               // Цикл по объектам
        if(object>5)            // Номер объекта больше 5?
        object=0;           // Если да, то устанавливаем 0
    }
    if (!keys[' '])             // Клавиша "пробел" отпущена?
    {
        sp=FALSE;               // Если да, то устанавливаем sp в FALSE
    }
```

Это все! Теперь вы можете рисовать квадратичные объекты в OpenGL. С помощью морфинга и квадратичных объектов можно сделать достаточно впечатляющие вещи. Анимированный диск - это пример простого морфинга.

Все у кого есть время зайдите на мой сайт, TipTup.Com 2000. (http://www.tiptup.com)

Урок 19. Машина моделирования частиц с использованием полосок из треугольников

Particle Engine Using Triangle Strips

Добро пожаловать на урок 19. Вы многое узнали, и теперь слегка развлечься. Я познакомлю Вас только с одной новой командой в этом уроке... Полоски из треугольников (triangle strip). Это очень удобно, и поможет ускорить ваши программы, когда надо рисовать множество треугольников.
В этом уроке я обучу Вас, как сделать несложную машину моделирования частиц (Particle Engine). Если Вы поймете, как работает машина моделирования частиц, Вы сможете создавать эффекты огня, дыма, водных фонтанов и так далее, не правда ли хорошее лакомство!

Я должен, однако предупредить Вас! На сегодняшний день я не написал ни одной машины моделирования частиц. Я знал, что 'знаменитая' машина моделирования частиц очень сложный кусок кода. Я делал попытки ранее, но обычно отказывался от этого после того, как я понимал, что я не смогу управлять всеми точками без того, чтобы не сойти с ума.

Вы можете мне не поверить, если я Вам скажу, что этот урок был написан на 100% с нуля. Я не заимствовал других идей, и я не имел никакой дополнительной технической информации. Я начал думать о частицах, и внезапно моя голова, наполнилась идеями (мозг включился?). Вместо того чтобы думать о каждой частице, как о пикселе, который был должен следовать от точки 'A' до точки 'B', и делать это или то, я решил, что будет лучше думать о каждой частице как об индивидуальном объекте, реагирующему на окружающую среду вокруг ее. Я дал каждой частице жизнь, случайное старение, цвет, скорость, гравитационное влияние и другое.
Вскоре я имел готовый проект. Я взглянул на часы, видимо инопланетяне снова забирали меня. Прошло 4 часа! Я помню, что время от времени пил кофе и закрывал глаза, но 4 часа...?

Поэтому, и хотя эта программа, по-моему, мнению грандиозная, и работает точно, так как я бы хотел, но возможно это не самый правильный способ создать машину моделирования частиц. Я не считаю это очень важным, так как машина моделирования частиц работает хорошо, и я могу использовать ее в моих проектах! Если Вы хотите знать, как точно это делается, то Вам надо потратить множество часов, просматривая сеть, в поисках подходящей информации. Только одно предупреждение. Те фрагменты кода, которые Вы найдете, могут оказаться очень загадочными :).

Этот урок использует код урока 1. Есть, однако, много нового кода, поэтому я будут переписывать любой раздел кода, который содержит изменения (это будет проще для понимания).

Используя код урока 1, мы добавим 5 новых строк кода в начало нашей программы. Первая строка (stdio.h) позволит нам читать данные из файлов. Такую же строку, мы добавили и к другим урокам, которые использовали текстуры. Во второй строке задается, сколько мы будем создавать частиц, и отображать на экране. MAX_PARTICLES будет равно любому значению, которое мы зададим. В нашем случае 1000. В третьей строке будет переключаться 'режим радуги' (включен или выключен). Мы установим по умолчанию включенным этот режим. sp и rp - переменные, которые мы будем использовать, чтобы предотвратить автогенерацию повторений нажатия клавиш пробел или ввод (enter), когда они нажаты.

```
#include <windows.h>  // Заголовочный файл для Windows
#include <stdio.h>    // Заголовочный файл для стандартной библиотеки ввода/вывода(НОВОЕ)
#include <gl\gl.h>    // Заголовочный файл для библиотеки OpenGL32
#include <gl\glu.h>   // Заголовочный файл для библиотеки GLu32
#include <gl\glaux.h> // Заголовочный файл для библиотеки GLaux

#define MAX_PARTICLES 1000 // Число частиц для создания ( НОВОЕ )

HDC       hDC=NULL;  // Приватный контекст устройства GDI
HGLRC     hRC=NULL;  // Постоянный контекст рендеринга
HWND      hWnd=NULL; // Сохраняет дескриптор окна
HINSTANCE  hInstance; // Сохраняет экземпляр приложения

bool  keys[256];      // Массив для работы с клавиатурой
bool  active=TRUE;    // Флаг активации окна, по умолчанию = TRUE
bool  fullscreen=TRUE;// Флаг полноэкранного режима
bool rainbow=true; // Режим радуги? ( НОВОЕ )
bool sp; // Пробел нажат? ( НОВОЕ )
bool rp; // Ввод нажат? ( НОВОЕ)
```

В следующих 4 строках - разнообразные переменные. Переменная slowdown (торможение) контролирует, как быстро перемещаются частицы. Чем больше ее значение, тем медленнее они двигаются. Чем меньше ее значение, тем быстрее они двигаются. Если значение задано маленькое, частицы будут двигаться слишком быстро! Скорость, с которой частиц перемещаются, будет задавать их траекторию движения по экрану. Более медленные частицы не будут улетать далеко. Запомните это.

Переменные xspeed и yspeed позволяют нам контролировать направлением хвоста потока частиц. xspeed будет добавляться к текущей скорости частицы по оси X. Если у xspeed - положительное значение, то наша частица будет смещаться направо. Если у xspeed - отрицательное значение, то наша частица будет смещаться налево. Чем выше значение, тем больше это смещение в соответствующем направлении. yspeed работает также, но по оси Y. Причина, по которой я говорю 'БОЛЬШЕ' в заданном направлении означает, что есть и другие коэффициенты, воздействующие на направление траектории частицы. xspeed и yspeed позволяют перемещать частицу в том направлении, в котором мы хотим.

Последняя переменная zoom. Мы используем эту переменную для панорамирования внутрь и вне нашей сцены. В машине моделирования частиц, это позволяет увеличить размер просмотра, или резко его сократить.

```
float slowdown=2.0f; // Торможение частиц
float xspeed; // Основная скорость по X (с клавиатуры изменяется направление хвоста)
float yspeed; // Основная скорость по Y (с клавиатуры изменяется направление хвоста)
float zoom=-40.0f; // Масштаб пучка частиц
```

Теперь мы задаем еще одну переменную названную loop. Мы будем использовать ее для задания частиц и вывода частиц на экран. col будет использоваться для сохранения цвета, с каким созданы частицы. delay будет использоваться, чтобы циклически повторять цвета в режиме радуги.

Наконец, мы резервируем память для одной текстуры (текстура частицы). Я решил использовать текстуру вместо точек по нескольким причинам. Наиболее важная причина, что точки замедляют быстродействие и выглядят очень плохо. Во-вторых, текстуры - более крутой способ :). Вы можете использовать квадратную частицу, крошечное изображение вашего лица, изображение звезды, и т.д. Больше возможностей!

```
GLuint loop;       // Переменная цикла
GLuint col;        // Текущий выбранный цвет
GLuint delay;      // Задержка для эффекта радуги
GLuint texture[1]; // Память для нашей текстуры
```

Отлично, теперь интересный материал. В следующем разделе кода создается структура, которая описывает отдельную частицу. Это то место, где мы даем частице некоторые характеристики.

Мы начинаем с булевой переменной active. Если эта переменная ИСТИННА, то наша частица жива и летит. Если она равно ЛОЖЬ, то наша частица мертва, или мы выключили ее! В этой программе я не использую active, но ее удобно иметь на всякий случай (прим. переводчика: никогда неизвестно что будет потом, главное следовать определенным принципам).

Переменные life и fade управляют тем, как долго частица будет отображаться, и насколько яркой она будет, пока жива. Переменная life постепенно уменьшается на значение fade. В этой программе некоторые частицы будут гореть дольше, чем другие.

```
typedef struct // Структура частицы
{
  bool active; // Активность (Да/нет)
  float life;  // Жизнь
  float fade;  // Скорость угасания
```

Переменные r, g и b задают красную, зеленую и синюю яркости нашей частицы. Чем ближе r к 1.0f, тем более красной будет частица. Если все 3 переменных равны 1.0f, то это создаст белую частицу.

```
  float r; // Красное значение
  float g; // Зеленное значение
  float b; // Синие значение
```

Переменные x, y и z задают, где частица будет отображена на экране. x задает положение нашей частицы по оси X. y задает положение нашей частицы по оси Y, и, наконец, z задает положение нашей частицы по оси Z.

```
  float x; // X позиция
  float y; // Y позиция
  float z; // Z позиция
```

Следующие три переменные важны. Эти три переменные управляют тем, как быстро частица перемещается по заданной оси, и в каком направлении двигается. Если xi имеет отрицательное значение, то наша частица будет двигаться влево. Если положительное, то вправо. Если yi имеет отрицательное значение, то наша частица будет двигаться вниз. Если положительное, то вверх. Наконец, если zi имеет отрицательное значение, то частица будет двигаться вглубь экрана, и, если положительное, то вперед к зрителю.

```
  float xi; // X направление
  float yi; // Y направление
  float zi; // Z направление
```

Наконец, последние 3 переменные! О каждой из этих переменных можно думать как о гравитации. Если xg имеет положительное значение, то нашу частицу будет притягивать вправо. Если отрицательное, то нашу частицу будет притягивать влево. Поэтому, если наша частица перемещает влево (отрицательно) и мы применяем положительную гравитацию, то скорость в итоге замедлится настолько, что наша частица начнет перемещать в противоположном направлении. yg притягивает вверх или вниз, и zg притягивает вперед или назад от зрителя.

```
  float xg; // X гравитация
  float yg; // Y гравитация
  float zg; // Z гравитация
```

particles - название нашей структуры.

```
}
particles; // Структура Частиц
```

Затем мы создаем массив называемый particle. Этот массив имеет размер MAX_PARTICLES. В переводе на русский язык: мы создаем память для хранения 1000 (MAX_PARTICLES) частиц. Это зарезервированная память будет хранить информацию о каждой индивидуальной частице.

```
particles particle[MAX_PARTICLES]; // Массив частиц (Место для информации о частицах)
```

Мы сокращаем код программы, при помощи запоминания наших 12 разных цветов в массиве цвета. Для каждого цвета от 0 до 11 мы запоминаем красную, зеленую, и, наконец, синею яркость. В таблице цветов ниже запомнено 12 различных цветов, которые постепенно изменяются от красного до фиолетового цвета.

```
static GLfloat colors[12][3]= // Цветовая радуга
{
  {1.0f,0.5f,0.5f},{1.0f,0.75f,0.5f},{1.0f,1.0f,0.5f},{0.75f,1.0f,0.5f},
  {0.5f,1.0f,0.5f},{0.5f,1.0f,0.75f},{0.5f,1.0f,1.0f},{0.5f,0.75f,1.0f},
  {0.5f,0.5f,1.0f},{0.75f,0.5f,1.0f},{1.0f,0.5f,1.0f},{1.0f,0.5f,0.75f}
};

LRESULT  CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM); // Объявление WndProc
```

Код загрузки картинки не изменился.

```
AUX_RGBImageRec *LoadBMP(char *Filename)     // Загрузка картинки
{
 FILE *File=NULL;          // Индекс файла
 if (!Filename)            // Проверка имени файла
 {
  return NULL;             // Если нет вернем NULL
 }

 File=fopen(Filename,"r"); // Проверим существует ли файл

 if (File)                 // Файл существует?
 {
  fclose(File);            // Закрыть файл
```

```
  return auxDIBImageLoad(Filename); // Загрузка картинки и вернем на нее указатель
 }
 return NULL;              // Если загрузка не удалась вернем NULL
}
```

В этом разделе кода загружается картинка (вызов кода выше) и конвертирует его в текстуру. Status используется за тем, чтобы отследить, действительно ли текстура была загружена и создана.

```
int LoadGLTextures()                    // Загрузка картинки и конвертирование в текстуру
{
 int Status=FALSE;                      // Индикатор состояния

AUX_RGBImageRec *TextureImage[1];       // Создать место для текстуры
 memset(TextureImage,0,sizeof(void *)*1); // Установить указатель в NULL
```

Наша текстура загружается кодом, который будет загружать нашу картинку частицы и конвертировать ее в текстуру с линейным фильтром.

```
if (TextureImage[0]=LoadBMP("Data/Particle.bmp")) // Загрузка текстуры частицы
{
  Status=TRUE; // Задать статус в TRUE
  glGenTextures(1, &texture[0]); // Создать одну текстуру
  glBindTexture(GL_TEXTURE_2D, texture[0]);
  glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
  glTexImage2D(GL_TEXTURE_2D, 0, 3, TextureImage[0]->sizeX, TextureImage[0]->sizeY,
    0, GL_RGB,   GL_UNSIGNED_BYTE, TextureImage[0]->data);
}
if (TextureImage[0])            // Если текстура существует
{
 if (TextureImage[0]->data)     // Если изображение текстуры существует
 {
   free(TextureImage[0]->data); // Освобождение памяти изображения текстуры
 }
 free(TextureImage[0]);         // Освобождение памяти под структуру
}
 return Status;         // Возвращаем статус
}
```

Единственное изменение, которое я сделал в коде изменения размера, было увеличение области просмотра. Вместо 100.0f, мы можем теперь рассматривать частицы на 200.0f единиц в глубине экрана.

```
// Изменение размеров и инициализация окна GL
GLvoid ReSizeGLScene(GLsizei width, GLsizei height)
{
  if (height==0) // Предотвращение деления на ноль, если окно слишком мало
  {
    height=1; // Сделать высоту равной единице
  }

  //Сброс текущей области вывода и перспективных преобразований
  glViewport(0, 0, width, height);
  glMatrixMode(GL_PROJECTION); // Выбор матрицы проекций
  glLoadIdentity(); // Сброс матрицы проекции
  // Вычисление соотношения геометрических размеров для окна
  gluPerspective(45.0f,(GLfloat)width/(GLfloat)height,0.1f,200.0f); // ( МОДИФИЦИРОВАНО )
  glMatrixMode(GL_MODELVIEW); // Выбор матрицы просмотра модели
  glLoadIdentity(); // Сброс матрицы просмотра модели
}
```

Если Вы используете код урока 1, замените его на код ниже. Я добавил этот код для загрузки нашей текстуры и включения смешивания для наших частиц.

```
int InitGL(GLvoid)       // Все начальные настройки OpenGL здесь
{
 if (!LoadGLTextures()) // Переход на процедуру загрузки текстуры
 {
  return FALSE;        // Если текстура не загружена возвращаем FALSE
 }
```

Мы разрешаем плавное затенение, очищаем фон черным цветом, запрещаем тест глубины, разрешаем смешивание и наложение текстуры. После разрешения наложения текстуры мы выбираем нашу текстуру частицы.

```
 glShadeModel(GL_SMOOTH);    // Разрешить плавное затенение
 glClearColor(0.0f, 0.0f, 0.0f, 0.0f); // Черный фон
 glClearDepth(1.0f);         // Установка буфера глубины
 glDisable(GL_DEPTH_TEST);   // Запрещение теста глубины
 glEnable(GL_BLEND);         // Разрешаем смешивание
 glBlendFunc(GL_SRC_ALPHA,GL_ONE); // Тип смешивания

 // Улучшенные вычисления перспективы
 glHint(GL_PERSPECTIVE_CORRECTION_HINT,GL_NICEST);
 glHint(GL_POINT_SMOOTH_HINT,GL_NICEST);  // Улучшенные точечное смешение
 glEnable(GL_TEXTURE_2D);                 // Разрешение наложения текстуры
 glBindTexture(GL_TEXTURE_2D,texture[0]); // Выбор нашей текстуры
```

В коде ниже инициализируется каждая из частиц. Вначале мы активизируем каждую частицу. Если частица - не активна, то она не будет появляться на экране, независимо оттого, сколько жизни у нее осталось.

После того, как мы сделали частицу активной, мы даем ей жизнь. Я сомневаюсь, что тот способ, с помощью которого я задаю жизнь, и угасание частицы, это самый лучший способ, но повторюсь еще раз, что это отлично работает! Полная жизнь - 1.0f. Это также дает частице полную яркость.

```
for (loop=0;loop<MAX_PARTICLES;loop++) // Инициализация всех частиц
{
 particle[loop].active=true; // Сделать все частицы активными
 particle[loop].life=1.0f;   // Сделать все частицы с полной жизнью
```

Мы задаем, как быстро частица угасает, при помощи присвоения fade случайного значения. Переменная life будет уменьшаться на значение fade, каждый раз, после того как частица будет отображена. Случайное значение будет от 0 до 99. Его мы делим его на 1000, поэтому мы получим очень маленькое значение с плавающей запятой. В завершении мы добавим 0.003 к конечному результату так, чтобы скорость угасания никогда не равнялось 0.

```
 //Случайная скорость угасания
 particle[loop].fade=float(rand()%100)/1000.0f+0.003f;
```

Теперь, когда наша частица активна, и мы дали ей жизнь, пришло время задать ей цвет. Вначале, мы хотим, чтобы все частицы были разным цветом. Поэтому я, делаю каждую частицу одним из 12 цветов, которые мы поместили в нашу таблицу цветов вначале этой программы. Математика проста. Мы берем нашу переменную loop и умножаем ее на число цветов в нашей таблице цвета, и делим на максимальное число частиц (MAX_PARTICLES). Это препятствует тому, что заключительное значение цвета будет больше, чем наш максимальное число цветов (12).

Вот пример: 900 * (12/900) =12. 1000 * (12/1000) =12, и т.д.

```
 particle[loop].r=colors[loop*(12/MAX_PARTICLES)][0]; // Выбор красного цвета радуги
 particle[loop].g=colors[loop*(12/MAX_PARTICLES)][1]; // Выбор зеленного цвета радуги
 particle[loop].b=colors[loop*(12/MAX_PARTICLES)][2]; // Выбор синего цвета радуги
```

Теперь мы зададим направление, в котором каждая частица двигается, наряду со скоростью. Мы умножаем результат на 10.0f, чтобы создать впечатление взрыва, когда программа запускается.
Мы начинаем с положительного или отрицательного случайного значения. Это значение будет использоваться для перемещения частицы в случайном направлении со случайной скоростью.

```
 particle[loop].xi=float((rand()%50)-26.0f)*10.0f; // Случайная скорость по оси X
 particle[loop].yi=float((rand()%50)-25.0f)*10.0f; // Случайная скорость по оси Y
 particle[loop].zi=float((rand()%50)-25.0f)*10.0f; // Случайная скорость по оси Z
```

Наконец, мы задаем величину гравитации, которая воздействует на каждую частицу. В отличие от реальной гравитации, под действием которой все предметы падают вниз, наша гравитация сможет смещать частицы вниз, влево, вправо, вперед или назад (прим. переводчика: скорее всего это электромагнитное поле, а не гравитация). Вначале мы зададим гравитацию в полсилы, которая притягивает вниз. Чтобы сделать это, мы устанавливаем xg в 0.0f. Т.е. нет перемещения влево или вправо по оси X. Мы устанавливаем yg в -0.8f. Это создает притяжение вниз в полсилы. Если значение положительное, то притяжение вверх. Мы не хотим, чтобы частицы притягивались к нам или от нас, поэтому мы установим zg в 0.0f.

```
  particle[loop].xg=0.0f;  // Зададим горизонтальное притяжение в ноль
  particle[loop].yg=-0.8f; // Зададим вертикальное притяжение вниз
  particle[loop].zg=0.0f;  // зададим притяжение по оси Z в ноль
}
return TRUE; // Инициализация завершена OK
}
```

Теперь интересный материал. В следующем разделе кода мы выводим частицу, проверяем гравитацию, и т.д. Очень важно, чтобы Вы поняли, что происходит там, поэтому, пожалуйста, читайте тщательно :).

Мы сбрасываем матрицу просмотра модели только однажды. Мы позиционируем частицы, используя команду glVertex3f() вместо использования перемещения их, при таком способе вывода частиц мы не изменяем матрицу просмотра модели при выводе наших частиц.

```
int DrawGLScene(GLvoid) // Здесь мы все рисуем
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Очистка экрана и буфера глубины
  glLoadIdentity(); // Сброс матрицы просмотра модели
```

Мы начинаем наш вывод с цикла. Этот цикл обновит каждую из наших частиц.

```
  for (loop=0;loop<MAX_PARTICLES;loop++) // Цикл по всем частицам
  {
```

Вначале мы проверим, активна ли наша частица. Если она не активна, то ее не надо модифицировать. В этой программе они активны всегда. Но в вашей программе, Вы сможете захотеть сделать некоторые частицы неактивными.

```
    if (particle[loop].active) // Если частицы не активны
    {
```

Следующие три переменные x, y и z - временные переменные, которые мы будем использовать, чтобы запомнить позицию частицы по x, y и z. Отмечу, что мы добавляем zoom к позиции по z, так как наша сцена смещена в экран на значение zoom. particle[loop].x - это наша позиция по x для любой частицы, которую мы выводим в цикле. particle[loop].y - это наша позиция по y для нашей частицы, и particle[loop].z - это наша позиция по z.

```
      float x=particle[loop].x; // Захватим позицию X нашей частицы
      float y=particle[loop].y; // Захватим позицию H нашей частицы
      float z=particle[loop].z+zoom; // Позиция частицы по Z + Zoom
```

Теперь, когда мы имеем позицию частицы, мы можем закрасить частицу. particle[loop].r - это красная яркость частицы, particle[loop].g – это зеленая яркость, и particle[loop].b – это синяя яркость. Напомню, что я использую жизнь частицы (life) для альфа значения. По мере того, как частица умирает, она становится все более и более прозрачной, пока она, в конечном счете, не исчезнет. Именно поэтому, жизнь частиц никогда не должна быть больше чем 1.0f. Если Вы хотите, чтобы частицы горели более долго, пробуйте уменьшить скорость угасания так, чтобы частица не так быстро исчезла.

```
      // Вывод частицы, используя наши RGB значения, угасание частицы согласно её жизни
      glColor4f(particle[loop].r,particle[loop].g,particle[loop].b,particle[loop].life);
```

Мы задали позицию частицы и цвет. Все, что мы должны теперь сделать - вывести нашу частицу. Вместо использования текстурированного четырехугольника, я решил использовать текстурированную полоску из треугольников, чтобы немного ускорить программу. Некоторые 3D платы могут выводить треугольники намного быстрее чем, они могут выводить четырехугольники. Некоторые 3D платы конвертируют четырехугольник в два

треугольника за Вас, но некоторые платы этого не делают. Поэтому мы сделаем эту работу сами. Мы начинаемся с того, что сообщаем OpenGL, что мы хотим вывести полоску из треугольников.

```
glBegin(GL_TRIANGLE_STRIP); // Построение четырехугольника из треугольной полоски
```

Цитата непосредственно из красной книги (OpenGL Red Book): полоска из треугольников рисуется как ряд треугольников (трехсторонних полигонов) используя вершины V0, V1, V2, затем V2, V1, V3 (обратите внимание на порядок), затем V2, V3, V4, и так далее. Порядок должен гарантировать, что все треугольники будут выведены с той же самой ориентацией так, чтобы полоска могла правильно формировать часть поверхности. Соблюдение ориентации важно для некоторых операций, типа отсечения. Должны быть, по крайней мере, 3 точки, чтобы было что-то выведено.

Рисунок 1. Полоска из двух треугольников

Поэтому первый треугольник выведен, используя вершины 0, 1 и 2. Если Вы посмотрите на рисунок 1, Вы увидите, что точки вершин 0, 1 и 2 действительно составляют первый треугольник (верхняя правая, верхняя левая, нижняя правая). Второй треугольник выведен, используя вершины 2, 1 и 3. Снова, если Вы посмотрите на рисунок 1, вершины 2, 1 и 3 создают второй треугольник (нижняя правая, верхняя левая, нижняя правая). Заметьте, что оба треугольника выведены с тем же самым порядком обхода (против часовой стрелки). Я видел несколько сайтов, на которых заявлялось, что каждый второй треугольник должен быть в противоположном направлении. Это не так. OpenGL будет менять вершины, чтобы гарантировать, что все треугольники выведены тем же самым способом!

Есть две хорошие причины, для того чтобы использовать полоски из треугольников. Во-первых, после определения первых трех вершин начального треугольника, Вы должны только определять одну единственную точку для каждого другого дополнительного треугольника. Эта точка будет объединена с 2 предыдущим вершинами для создания треугольника. Во-вторых, сокращая количество данных, необходимых для создания треугольников ваша программа будет работать быстрее, и количество кода или данных, требуемых для вывода объекта резко сократиться.

Примечание: число треугольников, которые Вы видите на экране, будет равно числу вершин, которые Вы зададите минус 2. В коде ниже мы имеем 4 вершины, и мы видим два треугольника.

```
glTexCoord2d(1,1); glVertex3f(x+0.5f,y+0.5f,z); // Верхняя правая
glTexCoord2d(0,1); glVertex3f(x-0.5f,y+0.5f,z); // Верхняя левая
glTexCoord2d(1,0); glVertex3f(x+0.5f,y-0.5f,z); // Нижняя правая
glTexCoord2d(0,0); glVertex3f(x-0.5f,y-0.5f,z); // Нижняя левая
```

Наконец мы сообщаем OpenGL, что мы завершили вывод нашей полоски из треугольников.

```
glEnd(); // Завершение построения полоски треугольников
```

Теперь мы можем переместить частицу. Математически это может выглядеть несколько странно, но довольно просто. Сначала мы берем текущую позицию x частицы. Затем мы добавляем значение смещения частицы по x, деленной на slowdown/1000. Поэтому, если наша частица была в центре экрана на оси X (0), наша переменная смещения (xi) для оси X равна +10 (смещение вправо от нас) и slowdown было равно 1, мы сместимся направо на 10/(1*1000), или на 0.01f. Если мы увеличим slowdown на 2, мы сместимся только на 0.005f. Буду надеяться, что это поможет Вам понять, как работает замедление (slowdown).

Это также объясняет, почему умножение начальных значений на 10.0f заставляет пиксели перемещаться намного быстрее, создавая эффект взрыва.
Мы используем ту же самую формулу для осей y и z, для того чтобы переместить частицу по экрану.

```
// Передвижение по оси X на скорость по X
particle[loop].x+=particle[loop].xi/(slowdown*1000);
// Передвижение по оси Y на скорость по Y
```

```
particle[loop].y+=particle[loop].yi/(slowdown*1000);
// Передвижение по оси Z на скорость по Z
particle[loop].z+=particle[loop].zi/(slowdown*1000);
```

После того, как мы вычислили перемещение частицы, следующее, что мы должны сделать, это учесть гравитацию или сопротивление. В первой строке ниже, мы делаем это, при помощи добавления нашего сопротивления (xg) к скорости перемещения (xi).

Предположим, что скорость перемещения равна 10, а сопротивление равно 1. Первый раз, когда частица выводиться на экран, сопротивление воздействует на нее. Во второй раз, когда она выводится, сопротивление будет действовать, и скорость перемещения понизится от 10 до 9. Это заставит частицу немного замедлится. В третий раз, когда частица выводиться, сопротивление действует снова, и скорость перемещения понизится до 8. Если бы частица горела больше чем 10 перерисовок, то она будет в итоге перемещаться в противоположном направлении, потому что скорость перемещения станет отрицательным значением.

Сопротивление применяется к скорости перемещения по y и z, так же, как и по x.

```
particle[loop].xi+=particle[loop].xg; // Притяжение по X для этой записи
particle[loop].yi+=particle[loop].yg; // Притяжение по Y для этой записи
particle[loop].zi+=particle[loop].zg; // Притяжение по Z для этой записи
```

В следующей строке забирается часть жизни от частицы. Если бы мы не делали этого, то частица бы никогда не сгорела бы. Мы берем текущую жизнь частицы и вычитаем значение угасания для этой частицы. Каждая частица имеет свое значение угасания, поэтому они будут гореть с различными скоростями.

```
particle[loop].life-=particle[loop].fade; // Уменьшить жизнь частицы на ‘угасание’
```

Теперь мы проверим, жива ли частица, после того как мы изменили ее жизнь.

```
if (particle[loop].life<0.0f) // Если частица погасла
{
```

Если частица мертва (сгорела), мы оживим ее. Мы сделаем это, задав ей полную жизнь и новую скорость угасания.

```
particle[loop].life=1.0f; // Дать новую жизнь
// Случайное значение угасания
particle[loop].fade=float(rand()%100)/1000.0f+0.003f;
```

Мы также сделаем сброс позиций частицы в центр экрана. Мы делаем это, при помощи сброса позиций x, y и z частицы в ноль.

```
particle[loop].x=0.0f; // На центр оси X
particle[loop].y=0.0f; // На центр оси Y
particle[loop].z=0.0f; // На центр оси Z
```

После того, как частица была сброшена в центр экрана, мы задаем ей новую скорость перемещения / направления. Отмечу, что я увеличил максимальную и минимальную скорость, с которой частица может двигаться со случайного значения в диапазоне 50 до диапазона 60, но на этот раз, мы не собирается умножать скорость перемещения на 10. Мы не хотим взрыва на этот раз, мы хотим иметь более медленно перемещающиеся частицы.

Также заметьте, что я добавил xspeed к скорости перемещения по оси X, и yspeed к скорости перемещения по оси Y. Это позволит нам позже контролировать, в каком направлении двигаются частицы.

```
particle[loop].xi=xspeed+float((rand()%60)-32.0f);//Скорость и направление по оси X
particle[loop].yi=yspeed+float((rand()%60)-30.0f);//Скорость и направление по оси Y
particle[loop].zi=float((rand()%60)-30.0f);       //Скорость и направление по оси Z
```

Наконец мы назначаем частице новый цвет. В переменной col содержится число от 0 до 11 (12 цветов). Мы используем эту переменную для извлечения красной, зеленой и синей яркостей из нашей таблицы цветов, которую мы сделали в начале программы. В первой строке ниже задается красная яркость (r) согласно значению красного, сохраненного в colors[col][0]. Поэтому, если бы col равен 0, красная яркость равна 1.0f. Зеленые и синие значения получаются таким же способом.

Если Вы не поняли, как я получил значение 1.0f для красной яркости, если col - 0, я объясню это немного более подробно. Смотрите в начало программы. Найдите строку: static GLfloat colors[12][3]. Запомните, что есть 12 групп по 3 числа. Первые три числа - красная яркость. Второе значение - зеленая яркость, и третье значение - синяя яркость. [0], [1] и [2] ниже являются 1-ым, 2-ым и 3-ьим значениями, которые я только что упомянул. Если col равен 0, то мы хотим взглянуть на первую группу. 11 – последняя группа (12-ый цвет).

```
    particle[loop].r=colors[col][0]; // Выбор красного из таблицы цветов
    particle[loop].g=colors[col][1]; // Выбор зеленого из таблицы цветов
    particle[loop].b=colors[col][2]; // Выбор синего из таблицы цветов
  }
```

Строка ниже контролирует, насколько гравитация будет притягивать вверх. При помощи нажатия клавиши 8 на цифровой клавиатуре, мы увеличиваем переменную yg (y гравитация). Это вызовет притяжение вверх. Этот код расположен здесь, потому что это сделает нашу жизнь проще, гравитация будет назначена ко всем нашим частицам с помощью цикла. Если бы этот код был бы вне цикла, мы должны были бы создать другой цикл, чтобы проделать ту же самую работу, поэтому мы можем также сделать это прямо здесь.

```
  // Если клавиша 8 на цифровой клавиатуре нажата и гравитация меньше чем 1.5
  // тогда увеличим притяжение вверх
  if (keys[VK_NUMPAD8] && (particle[loop].yg<1.5f)) particle[loop].yg+=0.01f;
```

Эта строка создает точно противоположный эффект. При помощи нажатия 2 на цифровой клавиатуре мы уменьшаем yg, создавая более сильное притяжение вниз.

```
  // Если клавиша 2 на цифровой клавиатуре нажата и гравитация больше чем -1.5
  // тогда увеличим притяжение вниз
  if (keys[VK_NUMPAD2] && (particle[loop].yg>-1.5f)) particle[loop].yg-=0.01f;
```

Теперь мы модифицируем притяжение вправо. Если клавиша 6 на цифровой клавиатуре нажата, то мы увеличиваем притяжение вправо.

```
  // Если клавиша 6 на цифровой клавиатуре нажата и гравитация меньше чем 1.5
  // тогда увеличим притяжение вправо
  if (keys[VK_NUMPAD6] && (particle[loop].xg<1.5f)) particle[loop].xg+=0.01f;
```

Наконец, если клавиша 4 на цифровой клавиатуре нажата, то наша частица будет больше притягиваться влево. Эти клавиши позволяют получить некоторые действительно интересные результаты. Например, Вы сможете сделать поток частиц, стреляющих прямо в воздух. Добавляя немного притяжения вниз, Вы сможете превратить поток частиц в фонтан воды!

```
  // Если клавиша 4 на цифровой клавиатуре нажата и гравитация больше чем -1.5
  // тогда увеличим притяжение влево
  if (keys[VK_NUMPAD4] && (particle[loop].xg>-1.5f)) particle[loop].xg-=0.01f;
```

Я добавил этот небольшой код только для развлечения. Мой брат думает, что взрыв интересный эффект :). При помощи нажатия клавиши табуляции все частицы будут отброшены назад к центру экрана. Скорость перемещения частиц будет еще раз умножена на 10, создавая большой взрыв частиц. После того, как частицы взрыва постепенно исчезнут, появиться предыдущий столб частиц.

```
  if (keys[VK_TAB]) // Клавиша табуляции вызывает взрыв
  {
    particle[loop].x=0.0f; // Центр по оси X
    particle[loop].y=0.0f; // Центр по оси Y
    particle[loop].z=0.0f; // Центр по оси Z
    particle[loop].xi=float((rand()%50)-26.0f)*10.0f; // Случайная скорость по оси X
    particle[loop].yi=float((rand()%50)-25.0f)*10.0f; // Случайная скорость по оси Y
    particle[loop].zi=float((rand()%50)-25.0f)*10.0f; // Случайная скорость по оси Z
  }
 }
}
return TRUE; // Все OK
}
```

Код в KillGLWindow(), CreateGLWindow() и WndProc() не изменился, поэтому мы перейдем к WinMain(). Я повторю весь этот раздел кода, чтобы сделать просмотр кода проще.

```
int WINAPI WinMain(
        HINSTANCE hInstance,     // Экземпляр
         HINSTANCE hPrevInstance, // Предыдущий экземпляр
         LPSTR     lpCmdLine,     // Параметры командной строки
         int       nCmdShow)      // Показать состояние окна
{
 MSG  msg;        // Структура сообщения окна
 BOOL done=FALSE; // Булевская переменная выхода из цикла

  // Запросим пользователя какой режим отображения он предпочитает
 if (MessageBox(NULL,"Would You Like To Run In Fullscreen Mode?",
    "Start FullScreen?",MB_YESNO|MB_ICONQUESTION)==IDNO)
 {
  fullscreen=FALSE;              // Оконный режим
 }

 // Создадим наше окно OpenGL
 if (!CreateGLWindow("NeHe's Particle Tutorial",640,480,16,fullscreen))
 {
  return 0;                  // Выходим если окно не было создано
 }
```

Далее наше первое изменение в WinMain(). Я добавил код, который проверяет, в каком режиме пользователь решил запустить программу - в полноэкранном режиме или в окне. Если используется полноэкранный режим, я изменяю переменную slowdown на 1.0f вместо 2.0f. Вы можете опустить этот небольшой код, если Вы хотите. Я добавил этот код, чтобы ускорить полноэкранный режим на моем 3dfx (потому что при этом выполнение программы намного медленнее, чем в режиме окна по некоторым причинам).

```
 if (fullscreen)  // Полноэкранный режим ( ДОБАВЛЕНО )
 {
  slowdown=1.0f; // Скорость частиц (для 3dfx) ( ДОБАВЛЕНО )
 }

 while (!done) // Цикл, который продолжается пока done=FALSE
 {
  if (PeekMessage(&msg,NULL,0,0,PM_REMOVE)) // Есть ожидаемое сообщение?
  {
   if (msg.message==WM_QUIT) // Мы получили сообщение о выходе?
   {
    done=TRUE; // Если так done=TRUE
   }
   else // Если нет, продолжаем работать с сообщениями окна
   {
    TranslateMessage(&msg); // Переводим сообщение
    DispatchMessage(&msg);  // Отсылаем сообщение
   }
  }
  else // Если сообщений нет
  {
   // Рисуем сцену. Ожидаем нажатия кнопки ESC и сообщения о выходе от DrawGLScene()
   // Активно?  Было получено сообщение о выходе?
   if ((active && !DrawGLScene()) || keys[VK_ESCAPE])
   {
    done=TRUE; // ESC или DrawGLScene просигналили "выход"
   }
   else // Не время выходить, обновляем экран
   {
    SwapBuffers(hDC); // Переключаем буферы (Двойная буфферизация)
```

Я немного попотел со следующим куском кода. Обычно я не включаю все в одну строку, и это делает просмотр кода немного яснее :).

В строке ниже проверяется, нажата ли клавиша ‘+’ на цифровой клавиатуре. Если она нажата, и slowdown больше чем 1.0f, то мы уменьшаем slowdown на 0.01f. Это заставит частицы двигаться быстрее. Вспомните, что я говорил выше о торможении и как оно воздействует на скорость, с которой частица перемещается.

```
if (keys[VK_ADD] && (slowdown>1.0f)) slowdown-=0.01f;//Скорость частицы увеличилась
```

В этой строке проверяется, нажата ли клавиша ‘-‘ на цифровой клавиатуре. Если она нажата, и slowdown меньше чем 4.0f, то мы увеличиваем slowdown. Это заставляет частицы двигаться медленнее. Я выставил предел в 4.0f, потому что я не хочу, чтобы они двигались очень медленно. Вы можете изменить минимальные и максимальные скорости, на какие Вы хотите :).

```
if (keys[VK_SUBTRACT] && (slowdown<4.0f)) slowdown+=0.01f; // Торможение частиц
```

В строке ниже проверяется, нажата ли клавиша PAGE UP. Если она нажата, то переменная zoom увеличивается. Это заставит частицы двигаться ближе к нам.

```
if (keys[VK_PRIOR]) zoom+=0.1f; // Крупный план
```

В этой строке создается противоположный эффект. Нажимая клавишу Page down, zoom уменьшиться, и сцена сместиться глубже в экран. Это позволит нам увидеть больше частиц на экране, но при этом частицы будут меньше.

```
if (keys[VK_NEXT]) zoom-=0.1f; // Мелкий план
```

В следующей секции кода происходит проверка, была ли нажата клавиша Enter. Если она нажата в первый раз, и она не удерживается уже некоторое время, мы позволим компьютеру узнать, что она нажата, устанавливая rp в true. Тем самым мы переключим режим радуги. Если радуга была true, она станет false. Если она была false, то станет true. В последней строке проверяется, была ли клавиша Enter отпущена. Если это так, то rp устанавливается в false, сообщая компьютеру, что клавиша больше не нажата.

```
if (keys[VK_RETURN] && !rp) // нажата клавиша Enter
{
  rp=true; // Установка флага, что клавиша нажата
  rainbow=!rainbow; // Переключение режима радуги в Вкл/Выкл
}
if (!keys[VK_RETURN]) rp=false; // Если клавиша Enter не нажата – сбросить флаг
```

Код ниже немного запутанный. В первой строке идет проверка, нажата ли клавиша пробела и не удерживается ли она. Тут же проверяется, включен ли режим радуги, и если так, то проверяется значение переменной delay больше чем 25. delay - счетчик, который используется для создания эффекта радуги. Если Вы меняете цвет каждый кадр, то все частицы будут иметь разный цвет. При помощи создания задержки, группа частиц останется с одним цветом, прежде чем цвет будет изменен на другой.

Если клавиша пробел была нажата, или радуга включена, и задержка больше чем 25, цвет будет изменен!

```
if ((keys[' '] && !sp) || (rainbow && (delay>25))) // Пробел или режим радуги
{
```

Строка ниже была добавлена, для того чтобы режим радуги был выключен, если клавиша пробел была нажата. Если бы мы не выключили режим радуги, цвета продолжили бы циклически повторяться, пока клавиша Enter не была бы нажата снова. Это сделано, для того чтобы можно было просмотреть все цвета, нажимая пробел вместо Enter.

```
  if (keys[' ']) rainbow=false; // Если пробел нажат запрет режима радуги
```

Если клавиша пробел была нажата, или режим радуги включен, и задержка больше чем 25, мы позволим компьютеру узнать, что пробел было нажат, делая sp равной true. Затем мы зададим задержку равной 0, чтобы снова начать считать до 25. Наконец мы увеличим переменную col, чтобы цвет изменился на следующий цвет в таблице цветов.

```
  sp=true; // Установка флага нам скажет, что пробел нажат
  delay=0; // Сброс задержки циклической смены цветов радуги
  col++;   // Изменить цвет частицы
```

Если цвет больше чем 11, мы сбрасываем его обратно в ноль. Если бы мы не сбрасывали col в ноль, наша программа попробовала бы найти 13-ый цвет. А мы имеем только 12 цветов! Попытка получить информацию о цвете, который не существует, привела бы к краху нашей программы.

```
  if (col>11) col=0; // Если цвет выше, то сбросить его
}
```

Наконец, если клавиша пробел больше не нажата, мы позволяем компьютеру узнать это, устанавливая переменную sp в false.

```
if (!keys[' ']) sp=false; // Если клавиша пробел не нажата, то сбросим флаг
```

Теперь внесем немного управления нашими частицами. Помните, что мы создали 2 переменные в начале нашей программы? Одна называлась xspeed, и вторая называлась yspeed. Также Вы помните, что после того как частица сгорит, мы давали ей новую скорость перемещения и добавляли новую скорость или к xspeed или к yspeed. Делая это, мы можем повлиять, в каком направлении частицы будут двигаться, когда они впервые созданы.

Например. Пусть частица имеет скорость перемещения 5 по оси X и 0 по оси Y. Если мы уменьшим xspeed до -10, то скорость перемещения будет равна -10 (xspeed) +5 (начальная скорость). Поэтому вместо перемещения с темпом 10 вправо, частица будет перемещаться с темпом -5 влево Понятно?

Так или иначе. В строке ниже проверяем, нажата ли стрелка "вверх". Если это так, то yspeed будет увеличено. Это заставит частицы двигаться вверх. Частицы будут двигаться вверх с максимальной скоростью не больше чем 200. Если бы они двигались быстрее этого значения, то это выглядело бы не очень хорошо.

```
//Если нажата клавиша вверх и скорость по Y меньше чем 200, то увеличим скорость
if (keys[VK_UP] && (yspeed<200)) yspeed+=1.0f;
```

В этой строке проверяем, нажата ли клавиша стрелка "вниз". Если это так, то yspeed будет уменьшено. Это заставит частицу двигаться вниз. И снова, задан максимум скорости вниз не больше чем 200.

```
// Если стрелка вниз и скорость по Y больше чем –200, то увеличим скорость падения
if (keys[VK_DOWN] && (yspeed>-200)) yspeed-=1.0f;
```

Теперь мы проверим, нажата ли клавиша стрелка вправо. Если это так, то xspeed будет увеличено. Это заставит частицы двигаться вправо. Задан максимум скорости не больше чем 200.

```
// Если стрелка вправо и X скорость меньше чем 200, то увеличить скорость вправо
if (keys[VK_RIGHT] && (xspeed<200)) xspeed+=1.0f;
```

Наконец мы проверим, нажата ли клавиша стрелка влево. Если это так... то Вы уже поняли что... xspeed уменьшено, и частицы двигаются влево. Задан максимум скорости не больше чем 200.

```
// Если стрелка влево и X скорость больше чем –200, то увеличить скорость влево
if (keys[VK_LEFT] && (xspeed>-200)) xspeed-=1.0f;
```

И последнее, что мы должны сделать - увеличить переменную delay. Я уже говорил, что delay управляет скоростью смены цветов, когда Вы используете режим радуги.

```
delay++; // Увеличить счетчик задержки циклической смены цветов в режиме радуги
```

Так же как и во всех предыдущих уроках, проверьте, что заголовок сверху окна правильный.

```
if (keys[VK_F1])            // Была нажата кнопка F1?
{
  keys[VK_F1]=FALSE;        // Если так - установим значение FALSE
  KillGLWindow();           // Закроем текущее окно OpenGL
  fullscreen=!fullscreen;   // Переключим режим "Полный экран"/"Оконный"
  // Заново создадим наше окно OpenGL
  if (!CreateGLWindow("NeHe's Particle Tutorial",640,480,16,fullscreen))
  {
    return 0;               // Выйти, если окно не было создано
  }       }     }    }  }
```

```
  // Сброс
  KillGLWindow();                 // Закроем окно
  return (msg.wParam);              // Выйдем из программы
}
```

В этом уроке, я пробовал детально объяснять все шаги, которые требуются для создания простой, но впечатляющей системы моделирования частиц. Эта система моделирования частиц может использоваться в ваших собственных играх для создания эффектов типа огня, воды, снега, взрывов, падающих звезд, и так далее. Код может быть легко модифицирован для обработки большего количество параметров, и создания новых эффектов (например, фейерверк).

Благодарю Richard Nutman за предложение о том, что частицы можно позиционировать с помощью glVertex3f() вместо сброса матрицы модели просмотра и перепозиционирования каждой частицы с помощью glTranslatef(). Оба метода эффективны, но его метод уменьшил количество вычислений для вывода каждой частицы, что вызвало увеличение быстродействия программы.

Благодарю Antoine Valentim за предложение использовать полоски из треугольников для ускорения программы и введения новой команды в этом уроке. Замечания к этому уроку были великолепными, и я признателен за это!

Я надеюсь, что Вам понравился этот урок. Если Вы что-то не понимаете, или Вы нашли ошибку в этом уроке, пожалуйста, сообщите мне об этом. Я хочу сделать уроки лучше. Ваши замечания очень важны!

Урок 20. Маскирование

Masking

Добро пожаловать на урок номер 20. Растровый формат изображения поддерживается, наверное, на каждом компьютере, и, скорее всего во всех операционных системах. С ним не только легко работать, но и очень просто загружать и использовать как текстуру. До этого урока мы использовали смешивание, чтобы вывести текст на экран и другие изображения без стирания того, что под текстом или изображением. Это эффективно, но результат не всегда удовлетворительный.

В большинстве случаев текстура смешивается излишне или не достаточно хорошо. При разработке игры со спрайтами, Вы не хотите, чтобы сцена за вашим персонажем просвечивала через его тело. Когда вы выводите текст на экран, Вы хотите, чтобы текст был сплошным и легким для чтения.

В данном случае очень пригодиться маскирование. Маскирование – двух шаговый процесс. Вначале мы выводим черно-белое изображение нашей текстуры поверх сцены. Белое - прозрачная часть нашей текстуры. Черное - сплошная часть нашей текстуры. Мы будем использовать такой тип смешивания, при котором только черное будет появляться на сцене. Это похоже на форму для выпечки. Затем мы меняем режим смешивания, и отображаем нашу текстуру поверх, того, что вырезано черным. Опять же, из-за того режима смешивания, который мы используем, только те части нашей текстуры будут скопированы на экран, которые находятся сверху черной маски.

Я приведу весь код в этом уроке кроме тех разделов, которые не изменились. Итак, если Вы готовы научиться кое-чему новому, давайте начнем!

```
#include <windows.h>  // Заголовочный файл для Windows
#include <math.h>     // Заголовочный файл для математической библиотеки Windows
#include <stdio.h>    // Заголовочный файл для стандартной библиотеки ввода/вывода
#include <gl\gl.h>    // Заголовочный файл для библиотеки OpenGL32
#include <gl\glu.h>   // Заголовочный файл для библиотеки GLu32
#include <gl\glaux.h> // Заголовочный файл для библиотеки Glaux

HDC       hDC=NULL;  // Приватный контекст устройства GDI
HGLRC     hRC=NULL;  // Постоянный контекст визуализации
HWND      hWnd=NULL; // Сохраняет дескриптор окна
HINSTANCE  hInstance; // Сохраняет экземпляр приложения
```

Мы будем использовать 7 глобальных переменных в этой программе. masking - логическая переменная (ИСТИНА / ЛОЖЬ), которая будет отслеживать, действительно ли маскировка включена или выключена. mp используется, чтобы быть уверенным, что клавиша 'M' не нажата. sp используется, чтобы быть уверенным, что 'Пробел' не нажат, и переменная scene будет отслеживать, действительно ли мы рисуем первую или вторую сцену.

Мы выделяем память для 5 текстур, используя переменную texture[5]. loop - наш общий счетчик, мы будем использовать его несколько раз в нашей программе, чтобы инициализировать текстуры, и т.д. Наконец, мы имеем переменную roll. Мы будем использовать roll, чтобы сделать прокрутку текстуры по экрану. Создавая изящный эффект! Мы будем также использовать ее для вращения объекта в сцене 2.

```
bool  keys[256];      // Массив для работы с клавиатурой
bool  active=TRUE;    // Флаг активации окна, по умолчанию = TRUE
bool  fullscreen=TRUE;// Флаг полноэкранного режима
bool  masking=TRUE;   // Маскирование Вкл/Выкл
bool  mp;             // M нажата?
bool  sp;             // Пробел нажат?
bool  scene;          // Какая сцена выводиться

GLuint  texture[5];   // Память для пяти наших текстур
GLuint  loop;         // Общая переменная цикла
GLfloat  roll;        // Катание текстуры

LRESULT  CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM); // Объявление WndProc
```

Код загрузки картинки не изменился. Он тот же, какой был в уроке 6, и т.д.

В коде ниже мы резервируем память для 5 изображений. Мы очищаем место и загружаем все 5 картинок. Мы делаем цикл по всем изображениям и конвертируем их в текстуры для использования в нашей программе. Текстуры сохранены в texture[0-4].

```
int LoadGLTextures()                // Загрузка картинки и конвертирование в текстуру
{
 int Status=FALSE;                  // Индикатор состояния
 AUX_RGBImageRec *TextureImage[5]; // Создать место для текстуры
 memset(TextureImage,0,sizeof(void *)*5); // Установить указатель в NULL

 if ((TextureImage[0]=LoadBMP("Data/logo.bmp")) &&   // Текстура эмблемы
     (TextureImage[1]=LoadBMP("Data/mask1.bmp")) &&  // Первая маска
     (TextureImage[2]=LoadBMP("Data/image1.bmp")) && // Первое изображение
     (TextureImage[3]=LoadBMP("Data/mask2.bmp")) &&  // Вторая маска
     (TextureImage[4]=LoadBMP("Data/image2.bmp")))   // Второе изображение
 {
  Status=TRUE;                      // Задать статус в TRUE
  glGenTextures(5, &texture[0]);  // Создать пять текстур

  for (loop=0; loop<5; loop++)    // Цикл по всем пяти текстурам
  {
   glBindTexture(GL_TEXTURE_2D, texture[loop]);
   glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
   glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
   glTexImage2D(GL_TEXTURE_2D, 0, 3, TextureImage[loop]->sizeX, TextureImage[loop]->sizeY,
     0, GL_RGB, GL_UNSIGNED_BYTE, TextureImage[loop]->data);
  }
 }

 for (loop=0; loop<5; loop++)      // Цикл по всем пяти текстурам
 {
  if (TextureImage[loop])          // Если текстура существуют
  {
   if (TextureImage[loop]->data) // Если изображение текстуры существует
   {
    free(TextureImage[loop]->data); // Освободить память изображения
   }
```

```
    free(TextureImage[loop]);     // Освободить структуру изображения
   }
  }
  return Status;                  // Возвращаем статус
}
```

Код ReSizeGLScene() не изменился, и мы опустим его.

Код инициализации необходимая формальность. Мы загружаем наши текстуры, задаем цвет очистки, задаем и разрешаем тест глубины, включаем плавное закрашивание, и разрешаем наложение текстуры. У нас простая программа, поэтому нет необходимости в сложной инициализации :).

```
int InitGL(GLvoid)            // Все начальные настройки OpenGL здесь
{
  if (!LoadGLTextures())      // Переход на процедуру загрузки текстуры
  {
    return FALSE;             // Если текстура не загружена возвращаем FALSE
  }

  glClearColor(0.0f, 0.0f, 0.0f, 0.0f); // Черный фон
  glClearDepth(1.0);          // Установка буфера глубины
  glEnable(GL_DEPTH_TEST);    // Разрешение теста глубины
  glShadeModel(GL_SMOOTH);    // Разрешить плавное закрашивание
  glEnable(GL_TEXTURE_2D);    // Разрешение наложения текстуры
  return TRUE;                // Инициализация завершена OK
}
```

Теперь самое интересное. Наш код рисования! Мы начинаем как обычно. Мы очищаем фон и буфер глубины. Затем мы сбрасываем матрицу вида, и перемещаемся на 2 единицы вглубь экрана так, чтобы мы могли видеть нашу сцену.

```
int DrawGLScene(GLvoid)          // Здесь мы все рисуем
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);      // Очистка экрана и буфера глубины
  glLoadIdentity();               // Сброс матрицы вида
  glTranslatef(0.0f,0.0f,-2.0f); // Перемещение вглубь экрана на 2 единицы
```

Первая строка ниже выбирает текстуру 'эмблемы' сайта NeHe. Мы наложим текстуру на экран, используя четырехугольник. Мы задаем четыре текстурных координаты совместно с четырьмя вершинами.
Дополнение от Джонатана Роя: помните, что OpenGL - графическая система на основе вершин. Большинство параметров, которые Вы задаете, регистрируются как атрибуты отдельных вершин. Текстурные координаты - один из таких атрибутов. Вы просто задаете соответствующие текстурные координаты для каждой вершины многоугольника, и OpenGL автоматически заполняет поверхность между вершинами текстурой, в процессе известном как интерполяция. Интерполяция - стандартная геометрическая техника, которая позволяет OpenGL определить, как данный параметр изменяется между вершинами, зная только значение, которое параметр имеет в вершинах непосредственно.
Как и в предыдущих уроках, мы представим, что мы на внешней стороне четырехугольника и назначаем координаты текстуры следующим образом: (0.0, 0.0) нижний левый угол, (0.0, 1.0) верхний левый угол, (1.0, 0.0) нижний правый, и (1.0, 1.0) верхний правый. А теперь, с учетом этой настройки, Вы можете указать, какие координаты текстуры соответствуют точке в середине четырехугольника? Правильно, (0.5, 0.5). Но в коде Вы ни где не задавали эту координату, не так ли? Когда рисуется четырехугольник, OpenGL вычисляет это за Вас. И просто волшебство то, что он это делает безотносительно к позиции, размера, или ориентации многоугольника!
В этом уроке мы привнесем еще один интересный трюк, назначив текстурные координаты, которые будут отличаться от 0.0 и 1.0. Текстурные координаты должны быть нормализованы. Значение 0.0 отображается на одну грань текстуры, в тоже время как значение 1.0 отображает на противоположную грань, захватывая всю ширину или высоту изображения текстуры за одну единицу, независимо от размера многоугольника или размера изображения текстуры в пикселях (о чем мы не должны волноваться при выполнении наложения текстуры, и это делает жизнь в целом несколько проще). Значения большие, чем 1.0, будут просто заворачивать наложение с другой грани и повторять текстуру. Другими словами, например, текстурная координата (0.3, 0.5) отображает точно тот же самый пиксель в изображении текстуры, как и координата (1.3, 0.5), или как (12.3,-2.5). В этом уроке, мы добьемся мозаичного эффекта, задавая значение 3.0 вместо 1.0, повторяя текстуру девять раз (3x3 мозаика) по поверхности четырехугольника.

Дополнительно, мы используем переменную roll, чтобы заставить текстуру перемещаться (или скользить) по поверхности четырехугольника. Значение 0.0 для roll, которое добавлено к вертикальной координате текстуры, означает, что наложение текстуры на нижнею грань четырехугольника начинается на нижней грани изображения текстуры, как показано на рисунке слева. Когда roll равна 0.5, наложение на нижнею грань четырехугольника начинается с половины изображении (см. рисунок справа). Прокрутка текстуры может использоваться, чтобы создать отличные эффекты типа движущихся облаков, вращающихся слов вокруг объектов, и т.д.

```
  glBindTexture(GL_TEXTURE_2D, texture[0]); // Выбор текстуры эмблемы
 glBegin(GL_QUADS);       // Начало рисования текстурного четырехугольника
  glTexCoord2f(0.0f, -roll+0.0f); glVertex3f(-1.1f, -1.1f,  0.0f);  // Лево Низ
  glTexCoord2f(3.0f, -roll+0.0f); glVertex3f( 1.1f, -1.1f,  0.0f);  // Право Низ
  glTexCoord2f(3.0f, -roll+3.0f); glVertex3f( 1.1f,  1.1f,  0.0f);  // Право Верх
  glTexCoord2f(0.0f, -roll+3.0f); glVertex3f(-1.1f,  1.1f,  0.0f);  // Лево Верх
 glEnd();                 // Завершения рисования четырехугольника
```

Продолжим… Теперь мы разрешаем смешивание. Чтобы этот эффект работал мы также должны отключить тест глубины. Очень важно, чтобы Вы это сделали! Если Вы не отключили тест глубины, вероятно, вы ничего не увидите. Все Ваше изображение исчезнет!

```
 glEnable(GL_BLEND);       // Разрешение смешивания
 glDisable(GL_DEPTH_TEST); // Запрет теста глубины
```

Первое, что мы делаем после того, как мы разрешаем смешивание и отключаем тест глубины – проверка надо ли нам маскировать наше изображение или смешивать его на старый манер. Строка кода ниже проверяет истина ли маскировка. Если это так, то мы задаем смешивание таким образом, чтобы наша маска выводилась на экран правильным образом.

```
 if (masking)              // Маскировка разрешена?
{
```

Если маскировка ИСТИНА, что строка ниже задаст смешивание для нашей маски. Маска – это только черно-белая копия текстуры, которую мы хотим вывести на экран. Любая белая часть маски будет прозрачной. Любая черная часть маски будет непрозрачной.

Команда настройки смешивания ниже делает следующее: цвет адресата (цвет на экране) будет установлен в черный, если часть нашей маски, которая копируется на экран, черная. Это означает, что часть экрана, которая попадает под черную часть нашей маски, станет черной. Все, что было на экране под маской, будет очищено в черный цвет. Часть экрана, попавшего под белую маску не будет изменена.

```
  glBlendFunc(GL_DST_COLOR,GL_ZERO); // Смешивание цвета экрана с нулем (Черное)
 }
```

Теперь мы проверим, какую сцену надо вывести. Если scene ИСТИНА, то мы выведем вторую сцену. Если scene ЛОЖЬ, то мы выведем первую сцену.

```
 if (scene)                // Рисовать вторую сцену?
 {
```

Мы не хотим, чтобы объекты были слишком большими, поэтому мы перемещаемся еще на одну единицу в экран. Это уменьшит размер наших объектов.

После того, как мы переместились в экран, мы вращаемся от 0-360 градусов в зависимости от значения roll. Если roll - 0.0, мы будем вращать на 0 градусов. Если roll - 1.0, мы будем вращать на 360 градусов. Довольно быстрое вращение, но я не хочу создавать другую переменную только, для того чтобы вращать изображение в центре экрана. :)

```
  glTranslatef(0.0f,0.0f,-1.0f);      // Перемещение вглубь экрана на одну единицу
  glRotatef(roll*360,0.0f,0.0f,1.0f); // Вращение по оси Z на 360 градусов
```

Мы уже имеем прокрутку эмблемы на экране, и мы вращаем сцену по оси Z, при этом любые объекты, которые мы рисуем, вращаются против часовой стрелки, теперь все, что мы должны сделать – это проверить, включена ли маскировка. Если это так, то мы выведем нашу маску, затем наш объект. Если маскировка выключена, то мы выведем только наш объект.

```
  if (masking)              // Маскирование включено?
  {
```

Если маскировка ИСТИНА, то код ниже выведет нашу маску на экран. Наш режим смешивания уже задан ранее. Теперь все, что мы должны сделать – это вывести маску на экран. Мы выбираем маску 2 (потому что это вторая сцена). После того, как мы выбрали текстуру маски, мы накладываем текстуру на четырехугольник. Четырехугольник размером на 1.1 единицу влево и вправо так, чтобы он немного выходил за край экрана. Мы хотим показать только одну текстуру, поэтому координаты текстуры изменяются от 0.0 до 1.0.

После отрисовки нашей маски на экране будет находиться сплошная черная копия нашей завершающей текстуры. Это похоже на то, что формочка для выпечки, которая по форме совпадает с нашей завершающей текстурой, вырезала на экране пустое черное место.

```
    glBindTexture(GL_TEXTURE_2D, texture[3]); // Выбор второй маски текстуры
    glBegin(GL_QUADS);  // Начало рисования текстурного четырехугольника
      glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.1f, -1.1f,  0.0f);  // Низ Лево
      glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.1f, -1.1f,  0.0f);  // Низ Право
      glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.1f,  1.1f,  0.0f);  // Верх Право
      glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.1f,  1.1f,  0.0f);  // верх Лево
    glEnd();            // Конец рисования четырехугольника
  }
```

Теперь, когда мы вывели нашу маску на экран, пришло время снова изменить режим смешивания. На сей раз, мы собираемся, указать OpenGL, что надо копировать на экран любую часть нашей цветной текстуры, которая НЕ черная. Поскольку завершающая текстура - точная копия маски, но с цветом, выводятся на экран только те части нашей текстуры, которые попадают сверху черной части маски. Поскольку маска черная, ничто с экрана не будет просвечивать через нашу текстуру. И с нами остается сплошная текстура, плавающая сверху по экрану.

Заметьте, что мы выбираем второе изображение после выбора завершающего режима смешивания. При этом выбирается наше цветное изображение (изображение, на котором основана вторая маска). Также заметьте, что мы выводим это изображения с правого верхнего угла маски. Те же самые текстурные координаты, те же самые вершины.

Если мы не выведем маску, наше изображение будет скопировано на экран, но оно смешает с тем, что было на экране.

```
  glBlendFunc(GL_ONE, GL_ONE);  // Копирование цветного изображения 2 на экран
  glBindTexture(GL_TEXTURE_2D, texture[4]); // Выбор второго изображения текстуры
  glBegin(GL_QUADS);            // Начало рисования текстурного четырехугольника
    glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.1f, -1.1f,  0.0f);  // Низ Лево
    glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.1f, -1.1f,  0.0f);  // Низ Право
    glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.1f,  1.1f,  0.0f);  // Верх Право
    glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.1f,  1.1f,  0.0f);  // Верх Лево
  glEnd();                    // Завершение рисования четырехугольника
}
```

Если scene была ЛОЖЬ, мы выведем первую сцену (моя любимая).

```
else                // Иначе
{
```

Вначале мы проверяем, включена ли маскировка, точно так же как в коде выше.

```
  if (masking)              // Вкл. маскировка?
  {
```

Если masking ИСТИНА, то мы выводим нашу маску 1 на экран (маска для сцены 1). Заметьте, что текстура прокручивается справа налево (roll добавляется к горизонтальной координате текстуры). Мы хотим, чтобы эта текстура заполнила весь экран, именно поэтому мы и не перемещаемся глубже в экран.

```
    glBindTexture(GL_TEXTURE_2D, texture[1]); // Выбор первой маски текстуры
    glBegin(GL_QUADS);  // Начало рисования текстурного четырехугольника
      glTexCoord2f(roll+0.0f, 0.0f); glVertex3f(-1.1f, -1.1f,  0.0f);  // Низ Лево
```

```
      glTexCoord2f(roll+4.0f, 0.0f); glVertex3f( 1.1f, -1.1f,  0.0f);  // Низ Право
      glTexCoord2f(roll+4.0f, 4.0f); glVertex3f( 1.1f,  1.1f,  0.0f);  // Верх Право
      glTexCoord2f(roll+0.0f, 4.0f); glVertex3f(-1.1f,  1.1f,  0.0f);  // Верх Лево
    glEnd();            // Конец рисования четырехугольника
  }
```

Снова мы разрешаем смешивание и выбираем нашу текстуру для сцены 1. Мы накладываем эту текстуру поверх маски. Заметьте, что мы прокручиваем эту текстуру таким же образом, иначе маска и завершающие изображение не совместились.

```
  glBlendFunc(GL_ONE, GL_ONE); // Копирование цветного изображения 1 на экран
  glBindTexture(GL_TEXTURE_2D, texture[2]); // Выбор первого изображения текстуры
  glBegin(GL_QUADS);    // Начало рисования текстурного четырехугольника
    glTexCoord2f(roll+0.0f, 0.0f); glVertex3f(-1.1f, -1.1f,  0.0f);  // Низ Лево
    glTexCoord2f(roll+4.0f, 0.0f); glVertex3f( 1.1f, -1.1f,  0.0f);  // Низ Право
    glTexCoord2f(roll+4.0f, 4.0f); glVertex3f( 1.1f,  1.1f,  0.0f);  // Верх Право
    glTexCoord2f(roll+0.0f, 4.0f); glVertex3f(-1.1f,  1.1f,  0.0f);  // Верх Лево
  glEnd();              // Конец рисования четырехугольника
 }
```

Затем мы разрешаем тест глубины, и отключаем смешивание. Это предотвращает странные вещи, происходящие от случая к случаю в остальной части нашей программы. :)

```
 glEnable(GL_DEPTH_TEST); // Разрешение теста глубины
 glDisable(GL_BLEND);     // Запрещение смешивания
```

В завершении надо увеличить значение roll. Если roll больше, чем 1.0, мы вычитаем 1.0. Это предотвращает появление больших значений roll.

```
 roll+=0.002f;           // Увеличим прокрутку нашей текстуры
 if (roll>1.0f)          // Roll больше чем
 {
  roll-=1.0f;            // Вычтем 1 из Roll
 }

 return TRUE;            // Все OK
}
```

Код KillGLWindow(), CreateGLWindow() и WndProc() не изменился, поэтому мы опустим его.

Первое что изменилось в WinMain() - заголовок окна. Теперь название "Урок Маскирования NeHe". Вы можете изменить это название на такое, какое Вы захотите. :)

```
int WINAPI WinMain(
        HINSTANCE hInstance,     // Экземпляр
        HINSTANCE hPrevInstance, // Предыдущий экземпляр
        LPSTR     lpCmdLine,     // Параметры командной строки
        int       nCmdShow)      // Показать состояние окна
{
 MSG  msg;       // Структура сообщения окна
 BOOL done=FALSE; // Булевская переменная выхода из цикла

 // Запросим пользователя какой режим отображения он предпочитает
 if (MessageBox(NULL,"Would You Like To Run In Fullscreen Mode?",
     "Start FullScreen?",MB_YESNO|MB_ICONQUESTION)==IDNO)
 {
  fullscreen=FALSE;             // Оконный режим
 }

 // Создадим наше окно OpenGL
 if (!CreateGLWindow("NeHe's Masking Tutorial",640,480,16,fullscreen))
 {    return 0;                 // Выходим если окно не было создано
 }
```

```
while (!done) // Цикл, который продолжается пока done=FALSE
{
 if (PeekMessage(&msg,NULL,0,0,PM_REMOVE)) // Есть ожидаемое сообщение?
 {
  if (msg.message==WM_QUIT) // Мы получили сообщение о выходе?
  {
   done=TRUE; // Если так done=TRUE
  }
  else // Если нет, продолжаем работать с сообщениями окна
  {
   TranslateMessage(&msg); // Переводим сообщение
   DispatchMessage(&msg);  // Отсылаем сообщение
  }
 }    else // Если сообщений нет    {
  // Рисуем сцену. Ожидаем нажатия кнопки ESC и сообщения о выходе от DrawGLScene()
  // Активно?  Было получено сообщение о выходе?
  if ((active && !DrawGLScene()) || keys[VK_ESCAPE])
  {
   done=TRUE; // ESC или DrawGLScene просигналили "выход"
  }       else // Не время выходить, обновляем экран      {
   SwapBuffers(hDC); // Переключаем буферы (Двойная буферизация)
```

Теперь наш простой обработчик клавиатуры. Мы проверяем, нажат ли пробел. Если это так, то мы устанавливаем переменную sp в ИСТИНА. Если sp ИСТИНА, код ниже не будет выполняться второй раз, пока пробел не был отпущен. Это блокирует быстрое переключение сцен в нашей программе. После того, как мы устанавливаем sp в ИСТИНА, мы переключаем сцену. Если scene была ИСТИНА, она станет ЛОЖЬ, если она была ЛОЖЬ, то станет ИСТИНА. В нашем коде рисования выше, если scene ЛОЖЬ, первая сцена будет выведена. Если scene ИСТИНА, то вторая сцена будет выведена.

```
if (keys[' '] && !sp) // Пробел нажат?
{
  sp=TRUE;            // Сообщим программе, что пробел нажат
  scene=!scene;       // Переключение сцен
}
```

Код ниже проверяет, отпустили ли мы пробел (если НЕ ' '). Если пробел был отпущен, мы устанавливаем sp в ЛОЖЬ, сообщая нашей программе, что пробел не нажат. Задав sp в ЛОЖЬ, код выше снова проверит, был ли нажат пробел, и если так, то все повториться.

```
if (!keys[' '])    // Пробел отжат?
{          sp=FALSE;        // Сообщим программе, что пробел отжат
}
```

В следующем разделе кода проверяется нажатие клавиши 'M'. Если она нажата, мы устанавливаем mp в ИСТИНА, указывая программе не проверять это условие в дальнейшем, пока клавиша не отпущена, и мы переключаем masking с ИСТИНА на ЛОЖЬ или с ЛОЖЬ на ИСТИНА. Если masking ИСТИНА, то в коде рисования будет подключена маскировка. Если masking ЛОЖЬ, то маскировка будет отключена. Если маскировка выключена, то объект будет смешан с содержимым экрана, используя смешивание на старый манер, который мы использовали до сих пор.

```
if (keys['M'] && !mp) // M нажата?
{   mp=TRUE;            // Сообщим программе, что M нажата
  masking=!masking;   // Переключение режима маскирования Выкл/Вкл
}
```

Последняя небольшая часть кода проверяет, отпущена ли "M". Если это так, то mp присваивается ЛОЖЬ, давая знать программе, что мы больше не нажимаем клавишу 'M'. Как только клавиша 'M' была отпущена, мы можем нажать ее еще раз, чтобы переключить включение или отключение маскировки.

```
if (!keys['M']) // M отжата?
{          mp=FALSE;     // Сообщим программе, что M отжата
}
```

Как и в других уроках, удостоверитесь, что заголовок наверху окна правильный.

```
      if (keys[VK_F1])            // Была нажата кнопка F1?
      {
        keys[VK_F1]=FALSE;        // Если так - установим значение FALSE
        KillGLWindow();           // Закроем текущее окно OpenGL
        fullscreen=!fullscreen;   // Переключим режим "Полный экран"/"Оконный"
        // Заново создадим наше окно OpenGL
        if (!CreateGLWindow("NeHe's Masking Tutorial",640,480,16,fullscreen))
        {
          return 0;               // Выйти, если окно не было создано
        }       }     }    } }

   // Сброс
  KillGLWindow();                 // Закроем окно
  return (msg.wParam);            // Выйдем из программы
}
```

Создание маски не сложно, и не требует много времени. Лучший способ сделать маску из готового изображения, состоит в том, чтобы загрузить ваше изображение в графический редактор или программу просмотра изображений, такую как infranview, и перевести изображение в серую шкалу. После того, как Вы сделали это, увеличивайте контрастность так, чтобы серые пиксели стали черными. Вы можете также уменьшить яркость, и т.д. Важно, что белый цвет это ярко белый, и черный это чисто черный. Если Вы имеете любые серые пиксели в вашей маске, то эта часть изображения будет прозрачной. Наиболее надежный способ удостовериться, что ваша маска точная копия вашего изображения, снять копию изображения с черным. Также очень важно, что ваше изображение имеет ЧЕРНЫЙ цвет, и маска имеет БЕЛЫЙ цвет! Если Вы создали маску и заметили квадратную форму вокруг вашей текстуры, или ваш белый - не достаточно яркий (255 или FFFFFF) или ваш черный - не точно черный (0 или 000000). Ниже Вы приведен пример маски и изображения, которое накладывается поверх маски. Изображение может иметь любой цвет, который Вы хотите, но фон должен быть черный. Маска должна иметь белый фон и черную копию вашего изображения.

Это - маска - > . Это - изображение - > .

(Прим.переводчика: Вы можете в изображении назначить любой цвет фоновым, важно, чтобы он стал белым в маске, а все остальные цвета перешли в черный цвет. Можно воспользоваться для выделения прозрачного цвета (или группы цветов) инструментов Select/Color Range в AdobePhotoshop, а затем залить выделенную область в белый цвет (тем самым вы создадите прозрачные области в маски), а затем инвертировать эту область и залить ее черным цветом (тем самым вы создадите непрозрачные области в маске).

Эрик Десросиерс подсказал, что Вы можете также проверять значение каждого пикселя в вашем рисунке, во время его загрузки. Если Вы хотите иметь прозрачный пиксель, Вы можете присвоить ему альфа значение 0. Для всех других цветов Вы можете присвоить им альфа значение 255. Этот метод будет также работать, но требует дополнительного кодирования. Текущий урок прост и требует очень немного дополнительного кода. Я не отвергаю другие методы, но когда я писал обучающую программу, я пробовал сделать код простым, понятым и удобным. Я только хотел сказать, что есть всегда другие способы сделать эту работу. Спасибо за замечание Эрик.

В этом уроке я показал Вам простой, но эффективный способ рисования частей текстуры на экран без использования альфа канала. Стандартное смешивание обычно выглядит плохо (текстуры или прозрачные, или они не прозрачные), и текстурирование с альфа каналом требует, чтобы ваши изображения имели альфа канал. С растровыми изображениями удобно работать, но они не поддерживают альфа канала, эта программа показывает нам, как обойти ограничения растровых изображений, демонстрируя крутой способ создавать эффекты типа штампа (effect overlay).

Благодарю Роба Санте за идею и за пример кода. Я никогда не слышал об этом небольшом трюке, пока он не указал на него. Он хотел, чтобы я подчеркнул, что, хотя эта уловка и работает, но для нее требуется два прохода, и это снижается производительность. Он рекомендует, чтобы Вы использовали текстуры с альфа каналом для сложных сцен.

Я надеюсь, что Вам понравится этот урок. Если Вы что-то не понимаете, или Вы нашли ошибку в уроке, пожалуйста, сообщите мне. Я хочу сделать уроки лучше. Ваши замечания также очень важны!

Урок 21. Линии, сглаживание, синхронизация, ортографическая проекция и звуки.

Lines, Antialiasing, Timing, Ortho View And Simple Sounds

Добро пожаловать на 21-ый урок по OpenGL! Темы, затронутые в этом уроке довольно не простые. Я знаю, что многие из Вас уже устали от изучения основ: 3D объекты, мультитекстурирование и другие базовые темы. Мне жаль сообщить тем, кто устал, что я хочу сохранить постепенный темп обучения. Потому что, однажды сделав слишком большой шаг вперед, можно будет потерять интерес части читателей. Поэтому я предпочел бы продолжать, двигаясь вперед не спеша.

И все же, если я потерял кое-кого из Вас :), то я хочу рассказать Вам немного об этом уроке. До сих пор во всех моих уроках использовались многоугольники, четырехугольники и треугольники. Поэтому я решил, что будет интересно создать урок о линиях. После нескольких часов создания урока о линиях, я решил не продолжать его. Урок вышел отличным, но он был СКУЧНЫМ! Линии, конечно, это замечательно, но надо сделать что-то невероятное, чтобы линии стали интересными. Тогда я просмотрел ваши письма на форуме, и запомнил несколько ваших просьб. Из них было несколько вопросов, которые подошли больше чем другие. Итак... Я решил написать мультиурок :).

В этом уроке Вы научитесь: выводить линии, делать сглаживание, оперировать ортографической проекцией, осуществлять синхронизацию времени, выводить простейшие звуковые эффекты, и реализовывать простую игровую логику. Буду надеяться, что в этом уроке есть все, чтобы сделать каждого счастливым :). Я потратил 2 дня, чтобы создать программу урока, и почти 2 недели, чтобы написать текст урока (и потребовалось 3 месяца, чтобы перевести это урок). Я надеюсь, что Вы будете наслаждаться моим трудом!

По окончанию этого урока у Вас будет простая готовая игра типа 'amidar'. Ваша миссия состоит в том, чтобы закрасить сетку и не попасться при этом в “лапы” плохими парнями. Игра имеет уровни, стадии, жизни, звук, и секретный предмет, чтобы помочь Вам переходить с уровня на уровень, когда ситуация становится критичной. Хотя эта игра прекрасно работает на Pentium 166 с Voodoo 2, более быстрый процессор рекомендуется, если Вы хотите иметь более плавную анимацию.

Я использовал код урока 1 в качестве отправной точки при написании этого урока. Мы начинаем с того, что включаем необходимые заголовочные файлы. Файл stdio.h используется для операций с файлами, и мы включаем stdarg.h для того чтобы мы могли вывести переменные на экран, типа счета и текущей стадии.

```
// Этот код сделан Jeff Molofee в 2000
// Если вы сочли этот код полезным, то дайте мне знать.

#include <windows.h> // заголовочный файл для Windows
#include <stdio.h>   // заголовочный файл для стандартного ввода/вывода
#include <stdarg.h>  // заголовочный файл для манипуляций с переменными аргументами
#include <gl\gl.h>   // заголовочный файл для библиотеки OpenGL32
#include <gl\glu.h>  // заголовочный файл для библиотеки GLu32
#include <gl\glaux.h>// заголовочный файл для библиотеки GLaux

HDC       hDC=NULL;  // Частный контекст устройства GDI
HGLRC     hRC=NULL;  // Контекст текущей визуализации
HWND      hWnd=NULL; // Декриптор нашего окна
HINSTANCE hInstance; // Копия нашего приложения
```

Теперь мы задаем наши булевские переменные. Переменная vline отслеживает все 121 вертикальную линию, которые составляют нашу игровую сетку. 11 линий вдоль и 11 вверх и вниз. Переменная hline отслеживает все 121 горизонтальную линию, которые составляют игровую сетку. Мы используем переменную ap для отслеживания, действительно ли клавиша “A” нажата.

Значение переменной filled равно ЛОЖЬ, пока сетка не закрашена и равно ИСТИНА, когда она закрашена. Назначение переменной gameover довольно очевидно. Если gameover равно ИСТИНА, то игра закончена, иначе Вы все еще играете. Переменная anti отслеживает сглаживание (antialiasing). Если anti равно ИСТИНА, сглаживание объектов ВКЛЮЧЕНО. Иначе оно выключено. Переменные active и fullscreen отслеживают, была ли программа свернута или нет, и запущена программа в полноэкранном режиме или оконном режиме.

```
bool    vline[11][10];    // Отслеживает вертикальные линии
bool    hline[10][11];    // Отслеживает горизонтальные линии
bool    ap;               // Клавиша 'A' нажата?
bool    filled;           // Сетка закрашена?
```

```
bool    gameover;         // Игра окончена?
bool    anti=TRUE;        // Сглаживание?
bool    keys[256];        // Массив для манипуляций с клавиатурой
bool    active=TRUE;      // Флаг активности окна, по умолчанию=TRUE
bool    fullscreen=TRUE;  // Флаг полноэкранного режима, по умолчанию=TRUE
```

Теперь мы определяем наши целые переменные. Переменные loop1 и loop2 будут использоваться для разных целей, например: для проверки точек на нашей сетке, для проверки попадания противника в нас и для случайного размещения объектов на сетке. Вы увидите loop1/loop2 в действии позже. Переменная-счетчик delay используется, чтобы замедлить перемещение плохих парней. Если delay больше чем некоторое значение, враги двигаются, и delay сбрасывается в ноль.

Переменная adjust - особенная переменная! В нашей программе есть таймер, но он используется только для проверки, если ваш компьютер слишком быстр. Если это так, то delay создана, чтобы замедлить компьютер. На моей плате GeForce, программа выполняется безумно гладко, и очень быстро. После проверки этой программы на моем PIII/450 с Voodoo 3500TV, я заметил, что она выполняется чрезвычайно медленно. Проблема состоит в том, что мой код синхронизации, только замедляет игру. Но не ускоряет ее. Поэтому я ввел новую переменную, называемую adjust (коррекция). Переменная adjust может принимать любое значение от 0 до 5. Объекты в игре перемещаются с различными скоростями в зависимости от значения adjust. Маленькие значения задают более гладкое перемещение, чем выше значение, тем они быстрее двигаются (граница после значений выше, чем 3). Это был единственно действительно простой способ сделать игру выполняемой на медленных системах. На одну вещь обратите внимание, независимо от того, как быстро объекты перемещаются, быстродействие игры никогда не будет больше чем, я ее назначил. Так присваивание переменной adjust значения равного 3, безопасно для быстрых и медленных систем.

Переменной lives присвоено значение 5, поэтому Вы начинаете игру с 5 жизнями. Переменная level - внутренняя переменная. В игре она используется, для того чтобы отслеживать уровень сложности. Это не тот уровень, который Вы увидите на экране. Переменной level2 присваивается то же самое значение, как и level, но ее значение необратимо увеличивается в зависимости от вашего навыка. Если Вы прошли уровень 3, переменная level замрет на значении 3. Переменная level - внутренняя переменная, характеризующая сложность игры. Переменная stage отслеживает текущую стадию игры.

```
int    loop1;           // Общая переменная 1
int    loop2;           // Общая переменная 2
int    delay;           // Задержка для Противника
int    adjust=3;        // Настройка скорости для медленных видеокарт
int    lives=5;         // Жизни для игрока
int    level=1;         // Внутренний уровень игры
int    level2=level;    // Уровень игры для отображения
int    stage=1;         // Стадия игры
```

Теперь мы создадим структуру для слежения за объектами в нашей игре. Мы имеем точное положение по X (fx) и точное положение по Y (fy). Эти переменные будут передвигать игрока и противников по сетки сразу на несколько пикселей. Они служат для создания плавного перемещения объекта.

Затем мы имеем x и y. Эти переменные будут отслеживать, в каком узле сетки находится наш игрок. Есть 11 точек слева направо и 11 точек сверху вниз. Поэтому переменные x и y могут принимать любое значение от 0 до 10. Именно поэтому мы нуждаемся в точных значениях. Если бы мы стали перемещать игрока с одного из 11 узлов по горизонтали, или с одного из 11 узлов по вертикали на другой соседний узел, то наш игрок быстро бы прыгал по экрану, а не плавно двигался между ними.

Последняя переменная spin будет использоваться для вращения объектов относительно оси Z.

```
struct    object          // Структура для игрока
{
  int  fx, fy;            // Точная позиция для передвижения
  int  x, y;              // Текущая позиция игрока
  float  spin;            // Направление вращения
};
```

Теперь, когда мы создали структуру, которая может использоваться для нашего игрока, противников и даже специальных предметов. Мы можем создавать новые структуры, которые используют свойства структуры, которую мы только, что определили.

В первой строке ниже создается структура для нашего игрока. По существу мы даем нашему игроку структуру со значениями fx, fy, x, y и spin. Добавив эту строку, мы сможем обратиться к позиции игрока x при помощи записи player.x. Мы можем изменять вращение игрока, добавляя число к player.spin.

Вторая строка немного отличается. Поскольку мы можем иметь до 9 противников на экране одновременно, мы должны создать вышеупомянутые переменные для каждого противника. Мы делаем для этого массив из 9 противников. Позиция x первого противника будет enemy[0].x. Позиция второго противника будет enemy[1].x, и т.д.

Последняя строка создает структуру для нашего специального элемента. Специальный элемент - песочные часы, которые будут появляться на экране время от времени. Мы должны следить за значениями x и y песочных часов, но так как песочные часы не двигаются, мы не должны следить за точными позициями. Вместо этого мы будем использовать точные переменные (fx и fy) для других целей.

```
struct  object  player;              // Информация о игроке
struct  object  enemy[9];             // Информация о противнике
struct  object  hourglass;            // Информация о песочных часах
```

Теперь мы создаем структуру таймера. Мы создаем структуру так, чтобы было проще следить за переменными таймера и так, чтобы было проще сообщить, что переменная является переменной таймера.

Вначале мы создаем целое число размером 64 бита, которое называется frequency (частота). В эту переменную будет помещено значение частоты таймера. Когда я вначале написал эту программу, я забыл включить эту переменную. Я не понимал то, что частота на одной машине не может соответствовать частоте на другой. Моя большая ошибка! Код выполнился прекрасно на 3 системах в моем доме, но когда я проверил его на машине моих друзей, игра работала ОЧЕНЬ быстро. Частота – говорит о том, как быстро часы обновляются. Хорошая вещь для слежки :).

Переменная resolution (точность таймера) отслеживает число вызовов таймера, которые требуются прежде, чем мы получим 1 миллисекунду времени.

В переменных mm_timer_start и mm_timer_elapsed содержатся значения, с которого таймер был запущен, и время, которое прошло с запуска таймера. Эти две переменные используются только, если компьютер не имеет высокоточного таймера. В этом случае мы используем менее точный мультимедийный таймер, который все же не так плох, в случае не критичной ко времени игры, такой как наша.

Переменная performance_timer может быть равной или ИСТИНА или ЛОЖЬ. Если программа находит высокоточный таймер, переменная performance_timer будет равна ИСТИНА, и синхронизация использует высокоточный таймер (намного более точный, чем мультимедийный таймер). Если высокоточный таймер не найден, переменная performance_timer будет равна ЛОЖЬ и мультимедийный таймер используется для синхронизации.

Последние 2 переменные - целые переменные по 64 бита, которые содержат время запуска высокоточного таймера и время, которое прошло с момента запуска высокоточного таймера.

Название этой структуры - " timer", как Вы можете увидеть внизу структуры. Если мы хотим знать частоту таймера, мы можем теперь проверить timer.frequency. Отлично!

```
struct                  // Создание структуры для информации о таймере
{
  __int64       frequency;              // Частота таймера
  float         resolution;             // Точность таймера
  unsigned long mm_timer_start;         // Стартовое значение мультимедийного таймера
  unsigned long mm_timer_elapsed;       // Прошедшее время мультимедийного таймера
  bool          performance_timer;      // Использовать высокоточный таймер?
  __int64       performance_timer_start;  // Стартовое значение высокоточного таймера
  __int64       performance_timer_elapsed; // Прошедшее время высокоточного таймера
} timer;                // Структура по имени таймер
```

Следующая строка кода - наша таблица скорости. Объекты в игре будут двигаться с разной скоростью в зависимости от значения adjust (коррекции). Если adjust - 0, объекты будут перемещаться на один пиксель одновременно. Если значение adjust - 5, объекты переместят на 20 пикселей одновременно. Таким образом, увеличивая значение adjust, скорость объектов увеличится, делая выполнение игры более быстрой на медленных компьютерах. Однако при больших значениях adjust игра будет выглядеть более дерганой.

Массив steps[] - только таблица для поиска. Если adjust равно 3, мы ищем число в позиции 3 массива steps[]. В позиции 0 хранится значение 1, в позиции 1 хранится значение 2, в позиции 2 хранится значение 4, и в позиции 3 хранится значение 5. Если adjust равно 3, наши объекты перемещались бы на 5 пикселей одновременно. Понятен смысл?

```
int    steps[6]={ 1, 2, 4, 5, 10, 20 }; // Значения шагов для работы
                                        // на медленных видеокартах
```

Затем мы создаем память для двух текстур. Мы загрузим фон сцены, и картинку для шрифта. Затем мы определяем переменную base, для списка отображения шрифта точно так же как, мы делали в других уроках со шрифтами. Наконец мы объявляем WndProc().

```
GLuint    texture[2];          // Память для текстур
GLuint    base;                // База для списка отображения шрифта

LRESULT  CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM); // Объявление для WndProc
```

Теперь интересный материал :). В следующем разделе кода мы инициализируем наш таймер. Вначале проверим, доступен ли высокоточный таймер (очень точный таймер). Если нет высокоточного таймера, будем использовать мультимедийный таймер. Этот код должен быть переносим, как следует из того, что я говорил.

Вначале сбросим все переменные структуры таймера в ноль. Это присвоит всем переменным в нашей структуре таймера значение ноль. После этого, мы проверим наличие высокоточного таймера. Здесь знак '!' означает НЕТ. Если таймер есть, то частота будет сохранена в timer.frequency.

Если нет высокоточного таймера, код между скобками будет выполнен. В первой строке переменной performance_timer присваивается ЛОЖЬ. Это говорит нашей программе, что нет никакого высокоточного счетчика. Во второй строке мы получаем стартовое значение для мультимедийного таймера от timeGetTime(). Мы задаем timer.resolution в 0.001f, и timer.frequency к 1000. Поскольку еще не прошло время, мы присваиваем прошедшему времени время запуска.

```
void TimerInit(void) // Инициализация нашего таймера (Начали)
{
 memset(&timer, 0, sizeof(timer)); // Очистка нашей структуры
 // Проверим доступность высокоточного таймера
 // Если доступен, то частота таймера будет задана
 if (!QueryPerformanceFrequency((LARGE_INTEGER *) &timer.frequency))
 {
  // Нет высокоточного таймера
  timer.performance_timer  = FALSE;       // Установим флаг высокоточного таймера в ЛОЖЬ
  timer.mm_timer_start  = timeGetTime();  // Текущее время из timeGetTime()
  timer.resolution  = 1.0f/1000.0f;       // Точность равна 0.001f
  timer.frequency    = 1000;              // Частота равна 1000
  timer.mm_timer_elapsed  = timer.mm_timer_start; // Прошедшее время равно текущему
 }
```

Если есть высокоточный таймер, следующий код будет выполнен вместо этого. В первой строке захватывается значение запуска высокоточного таймера, и помещается в performance_timer_start. Затем мы присваиваем переменной performance_timer значение ИСТИНА так, чтобы наша программа знала, что есть доступный высокоточный таймер. После этого, мы, вычисляем точность таймера, используя частоту, которую мы получили, когда проверяли наличие высокоточного таймера в коде выше. Мы делим единицу на эту частоту, чтобы получить точность. Последнее что мы сделаем, будет присвоение прошедшему времени значения стартового времени.

Заметьте вместо совместного использования переменных для высокоточного и мультимедийного таймера и переменных времени, я решил сделать разные переменные. В любо случае это будет работать прекрасно.

```
 else  {
  // Высокоточный таймер доступен, используем его вместо мультимедийного таймера
  // Взять текущее время и сохранить его в performance_timer_start
  QueryPerformanceCounter((LARGE_INTEGER *) &timer.performance_timer_start);
  timer.performance_timer    = TRUE;        // Установить флаг наличия таймера в TRUE
  // Вычислить точность таймера, используя частоту
  timer.resolution    = (float) (((double)1.0f)/((double)timer.frequency));
```

```
    // Присвоить прошедшему времени текущее время
    timer.performance_timer_elapsed  = timer.performance_timer_start;
  }
}
```

Раздел кода выше инициализирует таймер. Код ниже читает таймер и возвращает время, которое прошло в миллисекундах.

Вначале определим переменную в 64 бита под именем time. Мы будем использовать эту переменную, чтобы получить текущее время. Следующая строка проверяет, доступен ли высокоточный таймер. Если performance_timer равен ИСТИНА, то код после условия выполнится.

Первая строка кода внутри скобок будет захватывать значение таймера, и сохранять его в переменной, которую мы создали и назвали time. Вторая строка берет время, которое мы только что захватили (time) и вычитает из него время запуска, которое мы получили, когда запустили таймер. Поэтому наш таймер будет считать, начиная с нуля. Затем мы умножаем результаты на точность, чтобы выяснить, сколько секунд прошло. В конце мы умножает результат на 1000, чтобы выяснить, сколько прошло миллисекунд. После того, как вычисление сделано, результат будет возвращен обратно в тот раздел кода, который вызывал эту процедуру. Результат будет в формате с плавающей запятой для повышения точности.

Если мы не используем высокоточный таймер, код после инструкции else будет выполнен. Там в значительной степени делается тоже самое. Мы захватываем текущее время с помощью timeGetTime() и вычитаем из него наше значение при запуске. Мы умножаем на точность и затем на 1000, чтобы преобразовать результат из секунд в миллисекунды.

```
float TimerGetTime()          // Взять время в миллисекундах
{
  __int64 time;               // time содержит 64 бита
 if (timer.performance_timer) // Есть высокоточный таймер?
  {
    // Захват текущего значения высокоточного таймера
    QueryPerformanceCounter((LARGE_INTEGER *) &time);
    // Вернем текущее время минус начальное время, умноженное на точность и 1000 (для миллисекунд)
    return ( (float) ( time - timer.performance_timer_start) * timer.resolution)*1000.0f;
  }
  else
  {
    // Вернем текущее время минус начальное время, умноженное на точность и 1000 (для миллисекунд)
    return( (float) ( timeGetTime() - timer.mm_timer_start) * timer.resolution)*1000.0f;
  }
}
```

В следующей секции кода производится сброс структуры player с установкой позиции игрока в левом верхнем углу экрана, и задается противникам случайные начальные точки.

Левый верхний угол экрана – это 0 по оси X и 0 по оси Y. Поэтому, устанавливая player.x в 0, мы помещаем игрока на левый край экрана. Устанавливая player.y в 0, мы помещаем нашего игрока на верхний край экрана.

Точные позиции должны быть равны текущей позиции игрока, иначе наш игрок начал бы двигаться из случайной позиции, а не с левого верхнего угла экрана.

```
void ResetObjects(void)      // Сброс Игрока и Противников
{
  player.x=0;                // Сброс позиции игрока X на левый край экрана
  player.y=0;                // Сброс позиции игрока Y на верх экрана
  player.fx=0;               // Установим точную позиции X
  player.fy=0;               // Установим точную позиции Y
```

Далее мы даем противникам случайное начальное размещение. Количество противников, выведенное на экран, будет равно текущему значению уровня, умноженному на текущую стадию. Помните, что максимальное значение уровня может равняться трем, и максимальное число стадий в уровне тоже трем. Так что мы можем иметь максимум 9 противников.

Чтобы быть уверенными, что мы даем всем видимым противникам новую позицию, мы организуем цикл по всем видимым противникам (стадия, умноженная на уровень). Мы устанавливаем для каждого противника позицию x равную 5 плюс случайное значение от 0 до 5 (максимальное случайное значение может быть всегда число, которое Вы зададите минус 1). Поэтому враги могут появляться на сетке, где-то от 5 до 10. Затем мы даем врагу случайное значение по оси Y от 0 до 10.

Мы не хотим, чтобы враг двигался из старой позиции к новой случайной позиции, поэтому мы должны быть уверены, что точные значения по x (fx) и y (fy) равны значениям по x и y, умноженные на ширину и высоту каждой ячейки на экране. Каждая ячейка имеет ширину 60 и высоту 40.

```
  for (loop1=0; loop1<(stage*level); loop1++) // Цикл по всем противникам
  {
   enemy[loop1].x=5+rand()%6;              // Выбор случайной позиции X
   enemy[loop1].y=rand()%11;               // Выбор случайной позиции Y
   enemy[loop1].fx=enemy[loop1].x*60;      // Установка точной X
   enemy[loop1].fy=enemy[loop1].y*40;      // Установка точной Y
  }
}
```

Код AUX_RGBImageRec не изменился, поэтому я опускаю его. В LoadGLTextures() мы загрузим две наши текстуры. Сначала картинку шрифта (Font.bmp) и затем фоновое изображение (Image.bmp). Мы конвертируем оба изображения в текстуры, которые мы можем использовать в нашей игре. После того, как мы построили текстуры, мы очищаем память, удаляя растровую информацию. Здесь нет ничего нового. Если Вы читали другие уроки, Вы не должны иметь никаких проблем, в понимании кода.

```
int LoadGLTextures()                    // Загрузка растра и конвертирование его в текстуры
{
 int Status=FALSE;                      // Индикатор статуса
 AUX_RGBImageRec *TextureImage[2];      // Память для текстур
 memset(TextureImage,0,sizeof(void *)*2); // Указатель в NULL
if   ((TextureImage[0]=LoadBMP("Data/Font.bmp")) &&  // Загрузка фонта
    (TextureImage[1]=LoadBMP("Data/Image.bmp")))     // Загрузка фона
 {
  Status=TRUE;                          // Установка статуса в TRUE

  glGenTextures(2, &texture[0]);        // Создание текстуры
   for (loop1=0; loop1<2; loop1++)      // Цикл из 2 текстур

   {
    glBindTexture(GL_TEXTURE_2D, texture[loop1]);
    glTexImage2D(GL_TEXTURE_2D, 0, 3, TextureImage[loop1]->sizeX, TextureImage[loop1]->sizeY,
      0, GL_RGB, GL_UNSIGNED_BYTE, TextureImage[loop1]->data);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
   }

   for (loop1=0; loop1<2; loop1++)      // Цикл из 2 текстур
   {
    if (TextureImage[loop1])            // Если текстура существует
    {
     if (TextureImage[loop1]->data)     // Если изображение текстуры существует
     {
      free(TextureImage[loop1]->data); // Освободить память текстуры
     }
     free(TextureImage[loop1]);         // Освободить структуру изображения
    }
   }
 }
 return Status;                         // Возврат статуса
}
```

Код ниже создает список отображения шрифта. Я уже делал урок со шрифтом из текстуры. Весь код, делит изображение Font.bmp на 16 x 16 ячеек (256 символов). Каждая ячейка размером 16x16 станет символом. Поскольку я задал ось Y направленную вверх, поэтому, чтобы происходил сдвиг вниз, а не вверх, необходимо вычесть наши значения по оси Y из значения 1.0f. Иначе символы будут инвертированны :). Если Вы не понимаете то, что происходит, возвратитесь, и читайте урок по шрифтам из текстур.

```
GLvoid BuildFont(GLvoid)                    // Создаем список отображения нашего шрифта
{
  base=glGenLists(256);                     // Создаем списки
  glBindTexture(GL_TEXTURE_2D, texture[0]);  // Выбираем текстуру шрифта
  for (loop1=0; loop1<256; loop1++)         // Цикл по всем 256 спискам
  {
    float cx=float(loop1%16)/16.0f;         // X координата текущего символа
    float cy=float(loop1/16)/16.0f;         // Y координата текущего символа
    glNewList(base+loop1,GL_COMPILE);             // Начинаем делать список
      glBegin(GL_QUADS);        // Используем четырехугольник, для каждого символа
        glTexCoord2f(cx,1.0f-cy-0.0625f);         // Точка в текстуре (Левая нижняя)
        glVertex2d(0,16);       // Координаты вершины (Левая нижняя)
        glTexCoord2f(cx+0.0625f,1.0f-cy-0.0625f);  // Точка на текстуре (Правая нижняя)
        glVertex2i(16,16);      // Координаты вершины (Правая нижняя)
        glTexCoord2f(cx+0.0625f,1.0f-cy);         // Точка текстуры (Верхняя правая)
        glVertex2i(16,0);       // Координаты вершины (Верхняя правая)
        glTexCoord2f(cx,1.0f-cy);                 // Точка текстуры (Верхняя левая)
        glVertex2i(0,0);        // Координаты вершины (Верхняя левая)
      glEnd();                  // Конец построения четырехугольника (Символа)
      glTranslated(15,0,0);     // Двигаемся вправо от символа
    glEndList();                // Заканчиваем создавать список отображения
  }                             // Цикл для создания всех 256 символов
}
```

Это - хорошая идея уничтожить список отображения шрифта, когда Вы поработали с ним, поэтому я добавил следующий раздел кода. Снова, ничего нового.

```
GLvoid KillFont(GLvoid)             // Удаляем шрифт из памяти
{
  glDeleteLists(base,256);          // Удаляем все 256 списков отображения
}
```

Код glPrint() изменился не значительно. Единственное отличие от урока об текстурных шрифтах то, что я добавил возможность печатать значение переменных. Единственная причина, по которой я привожу этот раздел кода это та, что Вы можете при этом увидеть изменения. Вызов функции печати позиционирует текст в позиции x и y, которые Вы задаете. Вы можете выбрать один из 2 наборов символов, и значение переменных будет выведено на экран. Это позволит нам отображать текущий уровень и стадию.

Заметьте, что я разрешаю наложение текстуры, сбрасываю матрицу вид, и затем уставливаю в необходимую x / y позицию. Также заметьте, что, если выбран набор символов 0, шрифт укрупнен по ширине в полтора раз, и в два раза по высоте от первоначального размера. Я сделал это для того чтобы написать заголовок игры большими буквами. После того, как текст выведен, я отключаю наложение текстуры.

```
GLvoid glPrint(GLint x, GLint y, int set, const char *fmt, ...) // Печать
{
  char    text[256];          // Место для строки
  va_list    ap;              // Ссылка на список аргументов

  if (fmt == NULL)            // Если нет текста
    return;                   // то выходим

  va_start(ap, fmt);          // Разбор строки из значений
     vsprintf(text, fmt, ap);// и конвертирование символов в фактические числа
  va_end(ap);                 // Результат в текст
```

```
 if (set>1)                 // Если выбран не верный набор символов?
 {
  set=1;                    // Если так, то выбрать набор 1 (Курсив)
 }

 glEnable(GL_TEXTURE_2D);    // Разрешить наложение текстуры
 glLoadIdentity();           // Сбросить матрицу просмотра вида
 glTranslated(x,y,0);        // Позиция текста (0,0 – Низ Лево)
 glListBase(base-32+(128*set)); // Выбор набора символов (0 или 1)

 if (set==0)                 // Если 0 используем укрупненный фонт
 {
  glScalef(1.5f,2.0f,1.0f); // Ширина и Высота укрупненного шрифта
 }

 glCallLists(strlen(text),GL_UNSIGNED_BYTE, text); // Вывод текста на экран
 glDisable(GL_TEXTURE_2D);   // Запрет наложения текстуры
}
```

Код изменения размеров НОВЫЙ :). Вместо использования перспективной проекции я использую ортографическую проекцию для этого урока. Это означает, что объекты не уменьшаются, когда они удаляются от наблюдателя. Ось Z не используется в этом уроке.

Вначале зададим область просмотра. Мы делаем это таким же образом, которым бы мы делали перспективную проекцию. Мы задаем область просмотра, которая равна размеру нашего окна.

Затем мы выбираем матрицу проецирования и сбросим ее.

Сразу же после того, как мы сбрасываем матрицу проецирования, мы устанавливаем ортогональное проецирование. Я объясню эту команду подробнее.

Первый параметр (0.0f) - значение, которое мы хотим иметь на крайней левой стороне экрана. Вы хотели бы узнать, как использовать реальные значения пикселей, а не трехмерные координаты. Поэтому вместо использования отрицательного числа для левого края, я задал значение 0. Второй параметр - значение для крайней правой стороны экрана. Если наше окно - 640x480, значение по ширине будет 640. Поэтому крайняя правая сторона экрана равна 640. Поэтому наш экран по оси X изменяется от 0 до 640.

Третий параметр (высота) обычно был равен отрицательному числу, задающему нижний край экрана по оси Y. Но так как мы хотим использовать реальные значения пикселей, мы, не хотим иметь отрицательное число. Вместо этого мы сделаем низ экрана, равным высоте нашего окна. Если наше окно - 640x480, высота будет равна 480. Поэтому низ нашего экрана будет 480. Четвертый параметр обычно был равен положительному числу, задающему верхний край нашего экрана. Мы хотим, чтобы верхний край экрана был равным 0 (добрые старые координаты экрана), поэтому мы задаем четвертый параметр равным 0. При этом мы получим изменение от 0 до 480 по оси Y.

Последние два параметра - для оси Z. Мы не заботимся об оси Z, поэтому мы зададим диапазон от -1.0f до 1.0f. Нам будет достаточно того, что мы можем увидеть в 0.0f по оси Z.

После того, как мы задали ортографическую проекцию, мы выбираем матрицу просмотра вида (информация об объектах... расположение, и т.д) и сбрасываем ее.

```
GLvoid ReSizeGLScene(GLsizei width, GLsizei height) // Масштабирование и инициализация окна GL
{
 if (height==0)              // Предотвращение деления на ноль
 {
  height=1;                  // Сделать высоту равной 1
 }
 glViewport(0,0,width,height);  // Сброс текущей области просмотра
 glMatrixMode(GL_PROJECTION);   // Выбор матрицы проектирования
 glLoadIdentity();              // Сброс матрицы проектирования
 glOrtho(0.0f,width,height,0.0f,-1.0f,1.0f); // Создание ортог. вида 640x480 (0,0 – верх лево)
 glMatrixMode(GL_MODELVIEW);    // Выбор матрицы просмотра вида
glLoadIdentity();              // Сброс матрицы просмотра вида
}
```

Код инициализации содержит несколько новых команд. Вначале загружаем наши текстуры. Если они не загрузились, программа прекратит работу с сообщением об ошибке. После того, как мы создали текстуры, мы создаем наш шрифт. Я не делаю проверок на ошибки, если Вам надо вставьте его самостоятельно.

После того, как шрифт создан, мы задаем настройки. Мы разрешаем плавное сглаживание, задаем черный цвет очистки экрана и значение 1.0f для очистки буфера глубины. После этого, следует новая строка кода.

Функция glHint() сообщает OpenGL о настройках вывода. В этом случае мы сообщаем OpenGL, что мы хотим, чтобы сглаживание линии было наилучшим (самым хорошим), насколько это возможно под OpenGL. Эта команда разрешает сглаживание (anti-aliasing).

В конце мы разрешаем смешивание, и выбирает режим смешивания, который делает сглаживание линий возможными. Смешивание требуется, если Вы хотите, чтобы линии аккуратно смешались с фоном. Отключите смешивание, если Вы хотите увидеть, как все будет плохо смотреться без него.

Важно отметить, что иногда кажется, что сглаживание не работает. Объекты в этой игре небольшие, поэтому Вы можете и не заметить сглаживание вначале. Посмотрите по внимательнее. Заметьте, как зазубренные линии на противниках сглаживаются, когда сглаживание включено. Игрок и песочные часы также должны смотреться лучше.

```
int InitGL(GLvoid)            // Все настройки для OpenGL делаются здесь
{
  if (!LoadGLTextures())      // Переход на процедуру загрузки текстур
  {
    return FALSE;             // Если текстура не загружена, вернем ЛОЖЬ
  }

  BuildFont();                // Построение шрифта
  glShadeModel(GL_SMOOTH);    // Разрешить плавное сглаживание
  glClearColor(0.0f, 0.0f, 0.0f, 0.5f);   // Черный фон
  glClearDepth(1.0f);         // Настройка буфера глубины
  glHint(GL_LINE_SMOOTH_HINT, GL_NICEST); // Сглаживание линий
  glEnable(GL_BLEND);         // Разрешить смешивание
  glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA); // Тип смешивания
  return TRUE;                // Инициализация окончена успешна
}
```

Теперь кода рисования. Это именно то место, где творится волшебство :).

Мы очищаем экран (черным) вместе с буфером глубины. Затем мы выбираем текстуру шрифта (texture[0]). Мы хотим вывести фиолетовым цветом слова "GRID CRAZY" (сумасшедшая сетка), поэтому мы задаем красный и синий полной интенсивности, а зеленый половиной интенсивности. После того, как мы выбрали цвет, мы вызываем glPrint(). Мы помещаем слова "GRID CRAZY" с 207 по оси X (в центре экрана) и с 24 по оси Y (вверху экрана). Мы используем наш увеличенный шрифт, при помощи выбора шрифта 0.

После того, как мы вывели "GRID CRAZY" на экран, мы изменяем цвет на желтый (полная интенсивность красного, полная интенсивность зеленого). Мы пишем на экране "Level:" (уровень) и значение переменной level2. Помните, что level2 может быть больше чем 3. level2 хранит значение уровня, которое игрок видит на экране. Выражение %2i означает, что мы не хотим больше чем двух цифр на экране для представления уровня. Спецификатор “i” означает, что значение - целое число.

После того, как мы вывели информацию об уровне на экран, мы выводим информацию о стадии под ней, используя тот же самый цвет.

```
int DrawGLScene(GLvoid)               //Здесь мы будем рисовать
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Очистка экрана и буфера глубины
  glBindTexture(GL_TEXTURE_2D, texture[0]); // Выбор текстуры нашего шрифта
  glColor3f(1.0f,0.5f,1.0f);          // Установить фиолетовый цвет
  glPrint(207,24,0,"GRID CRAZY");     // Написать GRID CRAZY на экране
  glColor3f(1.0f,1.0f,0.0f);          // Установить желтый цвет
  glPrint(20,20,1,"Level:%2i",level2);  // Вывод состояние текущего уровня
  glPrint(20,40,1,"Stage:%2i",stage);   // Вывод состояние стадии
```

Теперь мы проверим, закончена ли игра. Если игра закончена, переменная gameover будет равна ИСТИНА. Если игра закончена, мы используем glColor3ub(r, г, b) чтобы выбрать случайный цвет. Отмечу, что мы использует 3ub вместо 3f. Используя 3ub, мы можем использовать целочисленные значения от 0 до 255 для задания цветов. Плюс в том, что при этом проще получить случайное значение от 0 до 255, чем получить случайное значение от 0.0f до 1.0f.

Как только случайный цвет был выбран, мы пишем слова "GAME OVER" (игра окончена) справа от заголовка игры. Справа под "GAME OVER" мы пишем "PRESS SPACE" (нажмите пробел). Это визуальное сообщение игроку, которое позволит ему узнать, что жизней больше нет и нажать пробел для перезапуска игры.

```
if (gameover)                   // Игра окончена?
{
  glColor3ub(rand()%255,rand()%255,rand()%255); // Выбор случайного цвета
  glPrint(472,20,1,"GAME OVER");   // Вывод GAME OVER на экран
  glPrint(456,40,1,"PRESS SPACE"); // Вывод PRESS SPACE на экран
}
```

Если игрока еще имеются жизни, мы выводим анимированные изображения символа игрока справа от заголовка игры. Чтобы сделать это, мы создаем цикл, который начинается от 0 до текущего количества жизней игрока минус один. Я вычитаю один, потому что текущая жизнь это то изображение, которым Вы управляете.

Внутри цикла, мы сбрасываем область просмотра. После того, как область просмотра была сброшена, мы передвигаемся на 490 пикселей вправо плюс значение loop1 умноженное 40.0f. Это позволит, выводить каждую из анимированных жизней игрока сдвинутую друг относительно друга на 40 пикселей. Первое анимированное изображение будет выведено в 490 + (0*40) (= 490), второе анимированное изображение будет выведено в 490 + (1*40) (= 530), и т.д.

После того, как мы сдвинули точку, мы выводим анимированное изображение и вращаем его против часовой стрелки в зависимости от значения в player.spin. Это заставляет анимированные изображения жизней вращаться в другую сторону относительно изображения активного игрока.

Затем выбираем зеленый цвет, и рисуем изображение. Способ вывода линий очень похож на рисование четырехугольника или многоугольника. Вы начинаете с glBegin(GL_LINES), сообщая OpenGL, что мы хотим вывести линию. Линии имеют 2 вершины. Мы используем glVertex2d, чтобы задать нашу первую точку. Функции glVertex2d не требуется указывать значение z. Эта функция хорошо нам подходит, так как мы не заботимся о значении z. Первая точка нарисована на 5 пикселей слева от текущего значения x и на 5 пикселей выше от текущего значения y. Это даст нам левую верхнюю точку. Вторая точка нашей первой линии рисуется на 5 пикселей справа от нашего текущего положения по x, и на 5 пикселей вниз. Это даст нам правую нижнюю точку. При этом будет нарисована линия от левой верхней точки до правой нижней точки. Наша вторая линия будет нарисована от правой верхней точки до левой нижней точки. При этом будет нарисован зеленый символ "X" на экране.

После того, как мы вывели зеленый символ “X”, мы делаем вращение против часовой стрелки (по оси z) еще больше, но на этот раз с половиной скорости. Затем мы выбираем более темный оттенок зеленого (0.75f) и рисуем другой символ “X” размером 7 вместо 5. При этом будет выведен большой / темный символ “X” сверху первого зеленного символа “X”. Поскольку более темный символ “X” вращается медленнее, то возникнет иллюзия наличия сверху яркого символа “X” вращающихся усиков (смешок).

```
for (loop1=0; loop1<lives-1; loop1++)          // Цикл по всем жизням минус текущая жизнь
{
  glLoadIdentity();             // Сброс вида
  glTranslatef(490+(loop1*40.0f),40.0f,0.0f);  // Перенос вправо от нашего заголовка
  glRotatef(-player.spin,0.0f,0.0f,1.0f);      // Вращение против часовой стрелки
  glColor3f(0.0f,1.0f,0.0f);    // Цвет игрока зеленный
  glBegin(GL_LINES);            // Рисуем игрока с помощью линий
    glVertex2d(-5,-5);          // Лево верх игрока
    glVertex2d( 5, 5);          // Низ право
    glVertex2d( 5,-5);          // Верх право
    glVertex2d(-5, 5);          // Низ лево
  glEnd();                    // Закончили рисовать игрока

  glRotatef(-player.spin*0.5f,0.0f,0.0f,1.0f); // Вращение против часовой стрелки
  glColor3f(0.0f,0.75f,0.0f);   // Установка темно-зеленного
```

```
  glBegin(GL_LINES);            // Рисуем игрока с помощью линий
    glVertex2d(-7, 0);          // Влево от центра игрока
    glVertex2d( 7, 0);          // Вправо от центра
    glVertex2d( 0,-7);          // Вверх от центра
    glVertex2d( 0, 7);          // Вниз от центра
  glEnd();                    // Закончили рисовать игрока
 }
```

Теперь мы выводим сетку. Мы задаем значение переменной filled равной ИСТИНА. Это сообщит нашей программе, что сетка была полностью выведена (Вы увидите позже, зачем мы это делаем).

Затем мы устанавливаем ширину линии равной 2.0f. Это делает линии более толстыми, делая визуализацию сетки более четкой.

Затем мы отключаем сглаживание. Причина, по которой мы отключаем сглаживание, состоит в том, что это великолепная функция, но она съедает центральный процессор на завтрак. Если Вы не имеете убийственную быструю графическую карту, то Вы заметите значительное падение производительности, если Вы оставите включенным сглаживание. Пробуйте это, если Вы хотите :).

Вид сброшен, и мы начинаем два цикла. Переменная loop1 будет путешествовать слева направо. Переменная loop2 будет путешествовать сверху донизу.

Мы задаем синий цвет линии, затем мы проверяем, пройдена ли игроком эта горизонтальная линия, если это так, то мы задаем белый цвет. Значение hline[loop1][loop2] было бы равно ИСТИННА, если линия была пройдена, и ЛОЖЬ, если игрок не пробегал через нее.

После того, как мы задали синий или белый цвета, мы выводим линию. Первое что надо проверить это то, что мы не ушли далеко вправо. Нам не надо выводить линии или делать проверки о прохождении линии, когда loop1 больше, чем 9.

Если переменная loop1 имеет правильное значение, мы проверяем, пройдена ли горизонтальная линия. Если это не так, то переменная filled установлена в ЛОЖЬ, сообщая, что есть, по крайней мере, одна линия, которая не была пройдена.
Затем линия рисуется. Мы выводим нашу первую горизонтальную линию (слева направо), начиная от 20+(0*60) (= 20). Эта линия выводится до 80 + (0*60) (= 80). Заметьте, что линия выведена слева направо. Именно поэтому мы не хотим вывести 11 (0-10) линий. Потому что последняя линия началась бы с правого края экрана и кончилась бы на 80 пикселей за экраном.

```
 filled=TRUE;                    // Задать True до начала тестирования
 glLineWidth(2.0f);              // Задать ширину линий для ячеек 2.0f
 glDisable(GL_LINE_SMOOTH);      // Запретить сглаживание
 glLoadIdentity();               // Сброс текущей матрицы вида и модели
 for (loop1=0; loop1<11; loop1++)   // Цикл слева направо
 {
  for (loop2=0; loop2<11; loop2++) // Цикл сверху вниз
  {
   glColor3f(0.0f,0.5f,1.0f);     // Задать синий цвет линии
   if (hline[loop1][loop2])       // Прошли горизонтальную линию?
   {
    glColor3f(1.0f,1.0f,1.0f);   // Если да, цвет линии белый
   }
   if (loop1<10)                 // Не рисовать на правом краю
   {
    if (!hline[loop1][loop2])    // Если горизонтальную линию не прошли
    {
     filled=FALSE;               // filled равно False
    }
    glBegin(GL_LINES);           // Начало рисования горизонтального бордюра ячейки
      glVertex2d(20+(loop1*60),70+(loop2*40)); // Левая сторона горизонтальной линии
      glVertex2d(80+(loop1*60),70+(loop2*40)); // Правая сторона горизонтальной линии
   glEnd();                      // Конец рисования горизонтального бордюра ячейки
   }
```

Код ниже делает то же самое, но при этом проверяется, что линия не выводится за нижний край экрана также как за правый край. Этот код ответствен за рисование вертикальных линий.

```
glColor3f(0.0f,0.5f,1.0f);     // Задать синий цвет линии
if (vline[loop1][loop2])       // Прошли вертикальную линию?
{
 glColor3f(1.0f,1.0f,1.0f);   // Если да, цвет линии белый
}

if (loop2<10)                  // Не рисовать на нижнем краю
{
 if (!vline[loop1][loop2])    // Если вертикальную линию не прошли
 {
  filled=FALSE;               // filled равно False
 }
 glBegin(GL_LINES);           // Начало рисования вертикального бордюра ячейки
  glVertex2d(20+(loop1*60),70+(loop2*40));  // Верхняя сторона вертикальной линии
  glVertex2d(20+(loop1*60),110+(loop2*40)); // Нижняя сторона вертикальной линии
 glEnd();                     // Конец рисования вертикального бордюра ячейки
}
```

Теперь мы проверим, пройдены ли все 4 стороны ячейки. Каждая ячейка на экране занимает 1/100-ая часть картинки полноэкранного экрана. Поскольку каждая ячейка часть большой текстуры, мы должны вначале разрешить отображение текстуры. Мы не хотим, чтобы текстура была подкрашена в красный, зеленый или синий, поэтому мы устанавливаем ярко белый цвет. После того, как цвет задан, мы выбираем нашу текстуру сетки (texture[1]).

Затем мы проверяем наличие ячейки на экране. Вспомните, что наш цикл рисует 11 линий справа и налево, и 11 линий сверху и вниз. Но мы не имеем 11 ячеек по одной линии. Мы имеем 10 ячеек. Поэтому мы не должны проверять 11-ую позицию. Для этого надо проверить, что и loop1 и loop2, меньше чем 10. Это даст 10 ячеек от 0 - 9.

После того, как мы будем уверены, что мы не выходим за диапазон, мы можем начинать проверять границы. hline[loop1][loop2] - верх ячейки. hline[loop1][loop2+1] - низ ячейки. vline[loop1][loop2] - левая сторона ячейки, и vline[loop1+1][loop2] - правая сторона ячейки. Я надеюсь, что следующий рисунок вам поможет:

Все горизонтальные линии получаются от loop1 до loop1+1. Как Вы можете видеть, первая горизонтальная линия создается при loop2. Вторая горизонтальная линия создается при loop2+1. Вертикальные линии получаются от loop2 до loop2+1. Первая вертикальная линия создается при loop1, и вторая вертикальная линия создается при loop1+1.

Когда переменная loop1 увеличивается, правая сторона нашей старой ячейки становится левой стороной новой ячейки. Когда переменная loop2 увеличивается, низ старой ячейки становится вершиной новой ячейки.

Если все 4 бордюра ИСТИННЫ (это означает, что мы прошли через все) мы можем наложить текстуру на блок. Мы сделаем это тем же самым способом, с помощью которого мы разделили текстуру шрифта на отдельные символы. Мы делим, и loop1 и loop2 на 10, потому что мы хотим наложить текстуру на 10 ячеек слева направо, и на 10 ячеек сверху и вниз. Координаты текстуры меняются от 0.0f до 1.0f, и 1/10-ый от 1.0f будет 0.1f.

Поэтому для вычисления координат правого верхнего угла нашего блока мы делим значения цикла на 10 и добавляем 0.1f к x координате текстуры. Чтобы получить координаты левого верхнего угла блока, мы делим наше значение цикла на 10. Чтобы получить координаты левого нижнего угла блока, мы делим наше значение цикла на 10 и добавляем 0.1f к y координате текстуры. Наконец, чтобы получить координаты правого нижнего угла текстуры, мы делим значение цикла на 10 и добавляем 0.1f, и к x и к y координатам текстуры.

Небольшой пример: loop1=0 и loop2=0

· Правая X координата текстуры = loop1/10+0.1f = 0/10+0.1f = 0+0.1f = 0.1f
· Левая X координата текстуры = loop1/10 = 0/10 = 0.0f
· Верх Y координата текстуры = loop2/10 = 0/10 = 0.0f;
· Низ Y координата текстуры = loop2/10+0.1f = 0/10+0.1f = 0+0.1f = 0.1f;

loop1=1 и loop2=1

· Правая X координата текстуры = loop1/10+0.1f = 1/10+0.1f = 0.1f+0.1f = 0.2f
· Левая X координата текстуры = loop1/10 = 1/10 = 0.1f

· Верх Y координата текстуры = loop2/10 = 1/10 = 0.1f;
· Низ Y координата текстуры = loop2/10+0.1f = 1/10+0.1f = 0.1f+0.1f = 0.2f;

Буду надеяться, что это все имеет смысл. Если бы loop1 и loop2 были бы равны 9, мы закончили бы со значениями 0.9f и 1.0f. Поэтому, как вы можете видеть, наши координаты текстуры наложенной на 10 блоков меняются от наименьшего значения 0.0f до наибольшего значения 1.0f. Т.е. наложение всей текстуры на экран. После того, как мы наложили часть текстуры на экран, мы отключаем наложение текстуры. После того, как мы нарисовали все линии и заполнили все блоки, мы задаем ширину линий равной 1.0f.

```
    glEnable(GL_TEXTURE_2D);      // Разрешение наложение текстуры
    glColor3f(1.0f,1.0f,1.0f);    // Ярко белый свет
    glBindTexture(GL_TEXTURE_2D, texture[1]); // Выбор мозаичного изображения
    if ((loop1<10) && (loop2<10)) // Если в диапазоне, заполнить пройденные ячейки
    {
     // Все ли стороны ячейки пройдены?
     if (hline[loop1][loop2] && hline[loop1][loop2+1] &&
         vline[loop1][loop2] && vline[loop1+1][loop2])
     {
      glBegin(GL_QUADS);       // Нарисовать текстурированный четырехугольник
       glTexCoord2f(float(loop1/10.0f)+0.1f,1.0f-(float(loop2/10.0f)));
       glVertex2d(20+(loop1*60)+59,(70+loop2*40+1));  // Право верх
      glTexCoord2f(float(loop1/10.0f),1.0f-(float(loop2/10.0f)));
       glVertex2d(20+(loop1*60)+1,(70+loop2*40+1));   // Лево верх
       glTexCoord2f(float(loop1/10.0f),1.0f-(float(loop2/10.0f)+0.1f));
       glVertex2d(20+(loop1*60)+1,(70+loop2*40)+39);  // Лево низ
       glTexCoord2f(float(loop1/10.0f)+0.1f,1.0f-(float(loop2/10.0f)+0.1f));
       glVertex2d(20+(loop1*60)+59,(70+loop2*40)+39); // Право низ
      glEnd();                 // Закончить текстурирование ячейки
     }
    }
    glDisable(GL_TEXTURE_2D);    // Запрет наложения текстуры
  }
 }
 glLineWidth(1.0f);              // Ширина линий 1.0f
```

Код ниже проверяет, равно ли значение переменной anti ИСТИНА. Если это так, то мы разрешаем сглаживание линий.

```
 if (anti)                  // Anti TRUE?
 {
  glEnable(GL_LINE_SMOOTH); // Если так, то разрешить сглаживание
 }
```

Чтобы сделать игру немного проще я добавил специальный предмет. Этим предметом будут песочные часы. Когда Вы касаетесь песочных часов, противники замерзают на определенное количество времени. Следующий раздел кода ответственен за вывод песочных часов.

Для песочных часов мы используем x и y, чтобы позиционировать таймер, но в отличие от нашего игрока и противников, мы не используем fx и fy для точного позиционирования. Вместо этого мы будем использовать fx, чтобы следить, действительно ли часы отображаются. fx будет равно 0, если часы не видимы, и 1, если они видимы, и 2, если игрок коснулся часов. fy будет использоваться как счетчик, для отслеживания как давно видны или не видны часы.

Поэтому вначале мы проверяем, видны ли часы. Если нет, мы обходим код вывода часов. Если часы видны, мы сбрасываем матрицу вида модели, и позиционируем часы. Поскольку наша первая точка сетки находится на 20 пикселей слева, мы добавим 20 к hourglass.x умножим на 60. Мы умножаем hourglass.x на 60, потому что точки на нашей сетке слева направо отстоят друг от друга на 60 пикселей. Затем мы позиционируем песочные часы по оси Y. Мы добавляем 70 к hourglass.y умножаем на 40, потому что мы хотим начать рисовать на 70 пикселей вниз от верхнего края экрана. Каждая точка на нашей сетке сверху внизу отстоит друг от друга на 40 пикселей.

После того, как мы завершили позиционирование песочных часов, мы можем вращать их по оси Z. hourglass.spin используется, чтобы следить за вращением, так же как player.spin следит за вращением игрока. Прежде, чем мы начинаем выводить песочные часы, мы выбираем случайный цвет.

```
 if (hourglass.fx==1)            // Если fx=1 нарисовать песочные часы
 {
  glLoadIdentity();              // Сброс матрицы вида модели
  glTranslatef(20.0f+(hourglass.x*60),70.0f+(hourglass.y*40),0.0f); // Поместим часы
  glRotatef(hourglass.spin,0.0f,0.0f,1.0f);      // Вращаем по часовой стрелке
  glColor3ub(rand()%255,rand()%255,rand()%255);  // Зададим случайный цвет часов
```

Вызов функции glBegin(GL_LINES) сообщает OpenGL, что мы хотим нарисовать линии. Вначале мы смещаемся на 5 пикселей влево и вверх от нашего текущего положения. При этом мы получим левую верхнюю вершину наших песочных часов. OpenGL начнет рисовать линию от этого положения. Конец линии будет вправо и вниз на 5 пикселей от нашего первоначального положения. При этом наша линия, пройдет от левой верхней точки до правой нижней точки. Сразу же после этого мы выводим вторую линию, проходящую от правой верхней точки до левой нижней точки. Это даст нам символ 'X'. В конце мы соединяем две нижние точки вместе, и затем две верхние точки, чтобы создать объект типа песочных часов :).

```
  glBegin(GL_LINES);           // Начало рисования наших песочных часов линиями
    glVertex2d(-5,-5);         // Лево Верх песочных часов
    glVertex2d( 5, 5);         // Право Низ песочных часов
    glVertex2d( 5,-5);         // Право Верх песочных часов
    glVertex2d(-5, 5);         // Лево Низ песочных часов
    glVertex2d(-5, 5);         // Лево Низ песочных часов
    glVertex2d( 5, 5);         // Право Низ песочных часов
    glVertex2d(-5,-5);         // Лево Верх песочных часов
    glVertex2d( 5,-5);         // Право Верх песочных часов
  glEnd();                     // Конец рисования песочных часов
 }
```

Теперь мы рисуем нашего игрока. Мы сбрасываем матрицу вида модели, и позиционируем игрока на экране. Заметьте, что мы позиционируем игрока, используя fx и fy. Мы хотим, чтобы игрок двигался плавно, поэтому мы используем точное позиционирование. После позиционирования игрока, мы вращаем игрока относительно оси Z, используя player.spin. Мы задаем светло зеленный цвет и начинаем рисовать. Примерно так же как мы вывели песочные часы, мы выводим символ 'X'. Начинаем с левой верхней точки до правой нижней точки, затем с правой верхней точки до левой нижней точки.

```
 glLoadIdentity();             // Сброс матрицы вида модели
 glTranslatef(player.fx+20.0f,player.fy+70.0f,0.0f); // Перемещение игрока в точную позицию
 glRotatef(player.spin,0.0f,0.0f,1.0f);             // Вращение по часовой стрелки
 glColor3f(0.0f,1.0f,0.0f);    // Установить светло-зеленный цвет
 glBegin(GL_LINES);            // Начать рисование нашего игрока из линий
  glVertex2d(-5,-5);           // Лево Верх игрока
  glVertex2d( 5, 5);           // Право Низ игрока
  glVertex2d( 5,-5);           // Право Верх игрока
  glVertex2d(-5, 5);           // Лево Низ игрока
 glEnd();                      // Конец рисования игрока
```

Рисование не слишком разнообразных объектов может разочаровать. Я не хотел бы, чтобы игрок выглядел скучновато, поэтому я добавил следующий раздел кода, для того чтобы создать большое и быстро вращающиеся лезвие поверх игрока, которого мы только что нарисовали выше. Мы вращаем относительно оси Z лезвие на player.spin умножив его на 0.5f. Поскольку мы вращаем еще раз, будет казаться, что эта часть игрока перемещается немного быстрее, чем первая часть игрока.
После выполнения нового вращения, мы меняем цвет на более темный оттенок зеленного. Так, чтобы казалось, что игрок, сделан из различных цветов / частей. Затем мы выводим большой '+' сверху первой части игрока. Он будет больше, потому что мы используем -7 и +7 вместо -5 и +5. Также заметьте, что вместо рисования от одного угла до другого, я рисую эту часть игрока слева направо и сверху вниз.

```
 glRotatef(player.spin*0.5f,0.0f,0.0f,1.0f); // Вращаем по часовой
 glColor3f(0.0f,0.75f,0.0f);   // Задаем цвет игрока темно-зеленный
 glBegin(GL_LINES);            // Начало рисования нашего игрока используя линии
  glVertex2d(-7, 0);           // Влево от центра игрока
  glVertex2d( 7, 0);           // Вправо от центра игрока
  glVertex2d( 0,-7);           // Вверх от центра игрока
  glVertex2d( 0, 7);           // Вниз от центра игрока
 glEnd();                      // Конец рисования игрока
```

Теперь нам осталось вывести противников, и мы закончим рисование :). Вначале мы организуем цикл по числу всех противников, которые есть на текущем уровне. Мы вычисляем, сколько противников надо рисовать, умножив нашу текущую игровую стадию на внутренний игровой уровень. Вспомните, что каждый уровень имеет 3 стадии, и максимальное значение внутреннего уровня равно 3. Поэтому мы можем иметь максимум 9 противников.

Внутри цикла мы сбрасываем матрицу просмотра вида, и позиционируем текущего противника (enemy[loop1]). Мы позиционируем противника, используя его точные значения x и y (fx и fy). После позиционирования текущего противника мы задаем розовый цвет и начинаем рисование.

Первая линия пройдет от 0,-7 (7 пикселей верх от начального положения) к -7,0 (7 пикселей влево от начального положения). Вторая линия пройдет от -7,0 до 0,7 (7 пикселей вниз от начального положения). Третья линия пройдет от 0,7 до 7,0 (7 пикселей вправо от нашего начального положения), и последняя линия пройдет от 7,0 назад к началу первой линии (7 пикселей верх от начального положения). При этом на экране получится не вращающийся розовый алмаз.

```
  for (loop1=0; loop1<(stage*level); loop1++) // Цикл рисования противников
  {
    glLoadIdentity();             // Сброс матрицы просмотра вида
    glTranslatef(enemy[loop1].fx+20.0f,enemy[loop1].fy+70.0f,0.0f);
    glColor3f(1.0f,0.5f,0.5f);    // Сделать тело противника розовым
    glBegin(GL_LINES);            // Начало рисования противника
      glVertex2d( 0,-7);          // Верхняя точка тела
      glVertex2d(-7, 0);          // Левая точка тела
      glVertex2d(-7, 0);          // Левая точка тела
      glVertex2d( 0, 7);          // Нижняя точка тела
      glVertex2d( 0, 7);          // Нижняя точка тела
      glVertex2d( 7, 0);          // Правая точка тела
      glVertex2d( 7, 0);          // Правая точка тела
      glVertex2d( 0,-7);          // Верхняя точка тела
    glEnd();                      // Конец рисования противника
```

Мы не хотим, чтобы враги выглядели невзрачно, поэтому мы добавим темно красное вращающиеся лезвие ('X') сверху алмаза, который мы только что нарисовали. Мы вращаем его относительно оси Z на enemy[loop1].spin, и затем выводим 'X'. Мы начинаем с левого верхнего угла и рисуем линию к правому нижнему углу. Затем мы рисуем вторую линию с правого нижнего угла до левого нижнего угла. Эти две линии пересекают друг с другом, и при этом получается символ 'X' (или клинок... смешок).

```
    glRotatef(enemy[loop1].spin,0.0f,0.0f,1.0f); // Вращение клинка противника
    glColor3f(1.0f,0.0f,0.0f);    // Сделаем клинок противника красным
    glBegin(GL_LINES);            // Начало рисования клинка противника
      glVertex2d(-7,-7);          // Лево верх противника
      glVertex2d( 7, 7);          // Право низ противника
      glVertex2d(-7, 7);          // Лево низ противника
      glVertex2d( 7,-7);          // Право верх противника
    glEnd();                      // Конец рисования противника
  }
  return TRUE;                    // Все OK
}
```

Я добавил вызов функции KillFont() в конце KillGLWindow(). При этом мы будем уверены, что список отображения шрифта удален, когда окно будет уничтожено.

```
GLvoid KillGLWindow(GLvoid) // Корректное удаление окна
{
  if (fullscreen)           // Мы в полноэкранном режиме?
  {
    ChangeDisplaySettings(NULL,0);  // Если это так, то переключиться на рабочий стол
    ShowCursor(TRUE);       // Показать курсор мыши
  }
  if (hRC)                  // У нас есть контекст визуализации?
  {
    if (!wglMakeCurrent(NULL,NULL)) // Мы можем освободить контексты DC и RC?
    {
```

```
    MessageBox(NULL,"Release Of DC And RC Failed.","SHUTDOWN ERROR",
            MB_OK | MB_ICONINFORMATION);
   }

   if (!wglDeleteContext(hRC))     // Мы можем удалить RC?
   {
    MessageBox(NULL,"Release Rendering Context Failed.","SHUTDOWN ERROR",
            MB_OK | MB_ICONINFORMATION);
   }
   hRC=NULL;              // Задать RC в NULL
  }

  if (hDC && !ReleaseDC(hWnd,hDC))  // Мы можем освободить DC?
  {
   MessageBox(NULL,"Release Device Context Failed.","SHUTDOWN ERROR",
           MB_OK | MB_ICONINFORMATION);
   hDC=NULL;              // Задать DC в NULL
  }

  if (hWnd && !DestroyWindow(hWnd)) // Мы можем уничтожить окно?
  {
   MessageBox(NULL,"Could Not Release hWnd.","SHUTDOWN ERROR",
           MB_OK | MB_ICONINFORMATION);
   hWnd=NULL;              // Задать hWnd в NULL
  }

  if (!UnregisterClass("OpenGL",hInstance)) // Мы можем удалить регистрацию класса?
  {
   MessageBox(NULL,"Could Not Unregister Class.","SHUTDOWN ERROR",
           MB_OK | MB_ICONINFORMATION);
   hInstance=NULL;         // Задать hInstance в NULL
  }
  KillFont();              // Уничтожить фонт, который мы сделали
}
```

Код CreateGLWindow() и WndProc() не изменил, поэтому идите вниз пока не встретите следующий раздел кода.

```
int WINAPI WinMain(
        HINSTANCE hInstance,     // Экземпляр
        HINSTANCE hPrevInstance, // Предыдущий экземпляр
        LPSTR     lpCmdLine,     // Параметры командной строки
        int       nCmdShow)      // Показать состояние окна
{
  MSG  msg;        // Структура сообщения окна
  BOOL done=FALSE; // Булевская переменная выхода из цикла

  // Запросим пользователя какой режим отображения он предпочитает
  if (MessageBox(NULL,"Would You Like To Run In Fullscreen Mode?",
     "Start FullScreen?",MB_YESNO|MB_ICONQUESTION)==IDNO)
  {
   fullscreen=FALSE;              // Оконный режим
  }
```

Этот раздел кода не много изменился. Я изменил заголовок окна на " Урок по линиям NeHe", и я добавил вызов функции ResetObjects(). При этом игрок позиционируется в левой верхней точке сетки, и противникам задаются случайные начальные положения. Враги будут всегда стартовать, по крайней мере, на 5 ячеек от Вас. Функция TimerInit() корректно инициализирует таймер.

```
  // Создадим наше окно OpenGL
  if (!CreateGLWindow("NeHe's Line Tutorial",640,480,16,fullscreen))
  {
   return 0;              // Выходим если окно не было создано
  }
```

```
ResetObjects();          // Установка стартовых позиций Игрока / Противников
TimerInit();             // Инициализация таймера

while (!done) // Цикл, который продолжается пока done=FALSE
{
 if (PeekMessage(&msg,NULL,0,0,PM_REMOVE)) // Есть ожидаемое сообщение?
 {
  if (msg.message==WM_QUIT) // Мы получили сообщение о выходе?
  {
   done=TRUE; // Если так done=TRUE
  }
  else // Если нет, продолжаем работать с сообщениями окна
  {
   TranslateMessage(&msg); // Переводим сообщение
   DispatchMessage(&msg);  // Отсылаем сообщение
  }
 }
 else // Если сообщений нет
 {
```

Теперь сделаем работу по синхронизации. Вначале запомните, что перед выводом нашей сцены, мы запоминаем время в переменной с плавающей запятой, которая названа start. Затем мы выводим сцену и переключаем буфера.

Сразу же после того, как мы переключили буфера, мы делаем задержку. Мы делаем при помощи сравнения текущего значения таймера (TimerGetTime()) с нашим стартовым значением плюс шаг скорости игры умноженный на 2. Если текущее значение таймера меньше чем значение, которое мы хотим, мы повторяем цикл, пока текущее значение таймера не будет равно или большее чем значение, которое мы хотим. Это ДЕЙСТВИТЕЛЬНО замедляет быстрые системы.

Поскольку мы используем шаги скорости (с помощью adjust) программа будет всегда выполняться с той же самой скоростью. Например, если бы наш шаг скорости был бы 1, мы ждали бы, пока таймер не был равен или больше чем 2 (1*2). Но если мы увеличим шаг скорости до 2 (что вызовет перемещение игрока на удвоенное число пикселей одновременно), задержка увеличиться на 4 (2*2). Поэтому даже при том, что мы перемещаем в два раза быстрее, задержка также удвоится, поэтому игра будет выполняться с той же самой скоростью :).

Есть один прием, который многие делают – берут текущее время, и вычитают из него старое время, чтобы выяснить, сколько времени прошло. Затем они перемещают объекты на некоторое расстояние, основанное на значении времени, которое прошло. К сожалению, я не могу этого сделать в этой программе, потому что точное перемещение должно быть таким, чтобы игрок мог попасть на линии сетки. Если текущая точная позиция x была 59, и компьютер решил переместить игрока, на два пикселя, игрок никогда не попадет на вертикальную линию в позиции 60 на сетке.

```
    float start=TimerGetTime(); // Захват времени до начала рисования

    // Нарисовать сцену. Отследить нажатие на клавишу ESC и
    // приход сообщения о выходе из DrawGLScene()

    if ((active && !DrawGLScene()) || keys[VK_ESCAPE])  // Активно?  Выход принят?
    {
     done=TRUE;              // ESC или DrawGLScene сигнализирует о выходе
    }
    else                     // Не время выходить, надо обновить сцену
    {
     SwapBuffers(hDC);       // Переключить буфера (Двойная Буферизация)
    }

    // Отбросим циклы на быстрой системе
    while(TimerGetTime()<start+float(steps[adjust]*2.0f)) {}
```

Следующий код мало изменился. Я изменил заголовок окна на "Урок NeHe по линиям".

```
    if (keys[VK_F1])         // Была нажата кнопка F1?
    {
     keys[VK_F1]=FALSE;      // Если так - установим значение FALSE
```

```
  KillGLWindow();         // Закроем текущее окно OpenGL
  fullscreen=!fullscreen; // Переключим режим "Полный экран"/"Оконный"
  // Заново создадим наше окно OpenGL
  if (!CreateGLWindow("NeHe's Line Tutorial",640,480,16,fullscreen))
  {
    return 0;             // Выйти, если окно не было создано
  }
}
```

В этой секции кода проверяется, нажата ли клавиша и не удерживается ли она. Если 'A' нажата, ap станет ИСТИНА (сообщая нашей программе, что ‘A’ опущена вниз), и anti переключается из ИСТИНЫ в ЛОЖЬ или из ЛОЖИ в ИСТИНУ. Помните, что значение anti проверяется в коде рисования, чтобы узнать включено ли сглаживание или нет.

Если клавиша 'A' была отпущена (ЛОЖЬ) тогда значение ap будет ЛОЖЬ сообщая программе, что эта клавиша больше не удерживается.

```
if (keys['A'] && !ap) // Если клавиша 'A' нажата и не удерживается
{
  ap=TRUE;            // ap равно TRUE
  anti=!anti;         // Переключим сглаживание
}

if (!keys['A'])       // Если клавиша 'A' отпущена
{
  ap=FALSE;           // ap равно FALSE
}
```

Теперь как перемещать противников. Я стремился сделать этот раздел кода как можно проще. Здесь очень немного логики. В основном, враги следят за тем, где Вы находитесь, и они двигаются в этом направлении. Поскольку я проверяю фактические x и y позиции игроков и не проверяю точные значения, игрокам может казаться, что враги имеют некоторый интеллект. Враги могут видеть, что Вы наверху экрана. Но к тому времени, когда точные значения совпадут с верхом экрана, Вы можете уже быть в другом месте. Это заставляет их иногда двигаться мимо Вас, прежде чем они поймут, что Вы больше не там, где они думают. Они как будто глухи, но поскольку они иногда двигаются мимо Вас, Вы может оказаться в окружении.

Мы начнем с проверки того, что игра еще не закончена, и что окно (если в оконном режиме) является все еще активным. При помощи этой проверки active делаем так, чтобы враги не двигались, когда окно свернуто. Это даст Вам удобную паузу, когда Вы захотите перерваться :).

После того, как мы проверили то, что враги должны быть перемещены, мы запускаем цикл. В этом цикле мы проходим по всем видимым противникам. Снова мы вычисляем, сколько противников должно быть на экране, умножая текущую стадию на текущий внутренний уровень.

```
if (!gameover && active) // Если игра не окончена и программа активна – передвинуть объекты
{
  for (loop1=0; loop1<(stage*level); loop1++)  // Цикл по противникам
  {
```

Теперь мы перемещаем текущего противника (enemy[loop1]). Вначале мы проверяем меньше ли x позиция противника, чем x позиция игрока, и мы контролируем, что точная позиция y противника выровнена с горизонтальной линией. Мы не можем перемещать противника влево или вправо, если он не на горизонтальной линии. Если бы мы сделали, враг прошел бы через середину ячейки, сделав игру очень сложной :).

Если x позиция противника меньше, чем x позиция игрока, и точная позиция y противника выровнена с горизонтальной линией, мы передвигаем противника по x на одну клетку ближе к текущей позиции игрока.

Подобным образом мы делаем это, чтобы переместить противника, влево, вниз и вверх. При перемещении вверх или вниз, мы должны проконтролировать, что точная позиция x противника выровнена с вертикальной линией. Мы не хотим, чтобы враг срезал через верх или низ ячейки.

Примечание: изменение x и y позиций противников не перемещает противника на экране. Вспомните, что, когда мы рисовали противников, мы использовали точные позиции, чтобы разместить противников на экране. Изменение x и y позиций только сообщает нашей программе, где мы ХОТИМ, чтобы противники двигались.

```
if ((enemy[loop1].x<player.x) && (enemy[loop1].fy==enemy[loop1].y*40))
{
 enemy[loop1].x++;    // Сдвиг противника вправо
}

if ((enemy[loop1].x>player.x) && (enemy[loop1].fy==enemy[loop1].y*40))
{
 enemy[loop1].x--;    // Сдвиг противника влево
}

if ((enemy[loop1].y<player.y) && (enemy[loop1].fx==enemy[loop1].x*60))
{
 enemy[loop1].y++;    // Сдвиг противника вниз
}

if ((enemy[loop1].y>player.y) && (enemy[loop1].fx==enemy[loop1].x*60))
{
 enemy[loop1].y--;    // Сдвиг противника вверх
}
```

В этом коде фактически реализовано перемещение. Мы проверяем, больше ли значение переменной delay, чем 3 минус текущий внутренний уровень. Т.е., если наш текущий уровень равен 1 программа, сделает цикл 2 раза (3-1) прежде, чем враги фактически сдвинутся. На уровне 3 (самый высокий уровень) враги будут перемещаться с той же самой скоростью как игрок (без задержек). Мы также контролируем, что hourglass.fx не равен 2. Вспомните, если hourglass.fx равно 2, то это означает, что игрок коснулся песочных часов. Враги при этом не должны перемещаться.

Если delay больше, чем с 3-level, и игрок не коснулся песочных часов, мы перемещаем противников, изменения точные позиции противников (fx и fy). Вначале мы присваиваем delay снова 0 так, чтобы мы могли запустить счетчик delay снова. Затем мы запускаем цикл, который проходит по всем видимым противникам (stage * level).

```
// Если наша задержка истекла, и игрок не коснулся песочных часов
if (delay>(3-level) && (hourglass.fx!=2))
{
 delay=0;          // Сброс задержки
 for (loop2=0; loop2<(stage*level); loop2++)  // Цикл по всем противникам
 {
```

Для перемещения противников, мы проверяем, нужно ли текущего противника (enemy[loop2]) двигать в заданном направлении, чтобы установить противника в x и y позицию, которую мы хотим. В первой строке ниже мы проверяем, является ли точная позиция противника по оси X меньше, чем нужная позиции x умноженная на 60. Вспомните, что размер каждой клетки равен 60 пикселям по горизонтали. Если точная позиция x меньше, чем x позиция противника умноженная на 60, мы сдвигаем противника направо на steps[adjust] (скорость нашей игры зависит от значения adjust). Мы также вращаем противника по часовой стрелке, чтобы казалось, что он катится направо. Для этого мы увеличиваем enemy[loop2].spin на steps[adjust] (текущая скорость игры, которая зависит от adjust).

Затем мы проверяем, является ли значение fx противника больше, чем позиция x противника умноженная на 60 и если это так, мы перемещаем противника влево и вращаем противника влево.

То же самое мы делаем при перемещении противника вверх и вниз. Если позиция y противника меньше, чем позиция fy противника умноженная на 40 (40 пикселей размер ячейки по вертикали) мы увеличиваем fy, и вращаем противника, чтобы казалось, что он катится вниз. Наконец, если позиция y больше, чем позиция fy умноженная на 40, мы уменьшаем значение fy, чтобы переместить противника вверх. Снова, вращаем противника, чтобы казалось, что он катится вверх.

```
  // Точная позиция по оси X меньше чем назначенная позиция?
  if (enemy[loop2].fx<enemy[loop2].x*60)
  {
   enemy[loop2].fx+=steps[adjust];    // Увеличим точную позицию по оси X
   enemy[loop2].spin+=steps[adjust];  // Вращаем по часовой
  }
  // Точная позиция по оси X больше чем назначенная позиция?
```

```
      if (enemy[loop2].fx>enemy[loop2].x*60)
      {
        enemy[loop2].fx-=steps[adjust];    // Уменьшим точную позицию по оси X
        enemy[loop2].spin-=steps[adjust];  // Вращаем против часовой
      }

      // Точная позиция по оси Y меньше чем назначенная позиция?
      if (enemy[loop2].fy<enemy[loop2].y*40)
      {
        enemy[loop2].fy+=steps[adjust];    // Увеличим точную позицию по оси Y
        enemy[loop2].spin+=steps[adjust];  // Вращаем по часовой
      }

      // Точная позиция по оси Y больше чем назначенная позиция?
      if (enemy[loop2].fy>enemy[loop2].y*40)
      {
        enemy[loop2].fy-=steps[adjust];    // Уменьшим точную позицию по оси Y
        enemy[loop2].spin-=steps[adjust];  // Вращаем против часовой
      }
    }
  }
```

После перемещения противников мы проверяем, попал ли кто-нибудь из них в игрока. Для точности мы сравниваем точные позиции противников с точной позицией игрока. Если позиция противника fx равна точной позиция fx игрока, и позиция fy противника равна fy игрока, то игрок МЕРТВ :).

Если игрок мертв, то мы уменьшаем его количество жизней. Затем мы проверяем, что у игрока еще есть жизни. Это можно сделать сравнением lives с 0. Если lives равно нулю, то мы присваиваем gameover ИСТИНА.

Затем мы сбрасываем наши объекты, вызывая ResetObjects(), и проигрываем звук смерти.

Вывод звука новый материал в этом уроке. Я решил использовать наиболее простую процедуру вывода звука ... PlaySound(). PlaySound() имеет три параметра. В первом параметре мы передаем ей название файла, который мы хотим проиграть. В нашем случае мы хотим, чтобы проиграл звук из файла Die.WAV в каталоге Data. Второй параметр можно проигнорировать. Мы установим его в NULL. Третий параметр – флаг для проигрывания звука. Два наиболее часто используемых типа флага: SND_SYNC, который приостанавливает выполнение программы пока звук не проиграет, и SND_ASYNC, который запускает проигрывание звука, но не останавливает программу. Мы хотим иметь небольшую задержку после того, как игрок умер, поэтому мы используем SND_SYNC. Довольно просто!

Я забыл рассказать об одной вещи в начале программы: для того чтобы PlaySound() и таймер работали, Вы должны подключить файл winmm.lib в проект (в Visual C++ это делается в PROJECT / SETTINGS / LINK). winmm.lib – мультимедийная библиотека Windows. Если Вы не включите эту библиотеку, Вы получите сообщения об ошибках, когда Вы пробуете откомпилировать программу.

```
    // Кто-нибудь из противников сверху игрока?
    if ((enemy[loop1].fx==player.fx) && (enemy[loop1].fy==player.fy))
    {
      lives--;            // Уменьшим жизни

      if (lives==0)       // Нет больше жизней?
      {
        gameover=TRUE;    // gameover равно TRUE
      }

      ResetObjects();     // Сброс позиций игрока / противников
      PlaySound("Data/Die.wav", NULL, SND_SYNC);  // Играем звук смерти
    }
  }
```

Теперь мы можем переместить игрока. В первой строке кода ниже мы проверяем, нажата ли стрелка вправо, и player.x меньше, чем 10 (не хотим выйти из сетки), и player.fx равно player.x умноженное на 60, и player.fy равно player.y умноженное на 40, т.е. находится в месте пересечения X и Y линий сетки.

Если мы не проверим, что игрок был в месте пересечения, и разрешим игроку перемещать как угодно, то игрок срежет правый угол ячейки, точно так же как противники сделали бы, если бы мы не проверяли, что они выровнены с вертикальной или горизонтальной линией. Эта проверка также делается, для того чтобы проверить, что игрок закончил, передвигаться прежде, чем мы переместим его в новое местоположение.

Если игрок в месте пересечения сетки (где встречаются вертикальные и горизонтальные линии) и он не за правым краем, мы помечаем, что текущая горизонтальная линия пройдена. Затем мы увеличиваем значение player.x на единицу, что вызывает перемещение игрока на одну клетку вправо.

Далее мы делаем также при перемещении влево, вниз и вверх. Когда перемещаем влево, мы проверяем, что игрок не вышел за левый край сетки. Когда перемещаем вниз, мы проверяем, что игрок не покинул сетку снизу, и при перемещении вверх мы проверяем, что игрок не вылетел за верх сетки.

При перемещении влево и вправо мы помечаем горизонтальную линию (hline[][]) ИСТИНА, что означает, что она пройдена. При перемещении вверх и вниз мы помечаем вертикальную линию (vline[][]) ИСТИНА, что означает, что она пройдена.

```
if (keys[VK_RIGHT] && (player.x<10) && (player.fx==player.x*60) && (player.fy==player.y*40))
{
  // Пометить текущую горизонтальную границу как пройденную
  hline[player.x][player.y]=TRUE;
  player.x++;        // Переместить игрока вправо
}
if (keys[VK_LEFT] && (player.x>0) && (player.fx==player.x*60) && (player.fy==player.y*40))
{
  player.x--;        // Переместить игрока влево
  // Пометить текущую горизонтальную границу как пройденную
  hline[player.x][player.y]=TRUE;
}

if (keys[VK_DOWN] && (player.y<10) && (player.fx==player.x*60) && (player.fy==player.y*40))
{
  // Пометить текущую вертикальную границу как пройденную
  vline[player.x][player.y]=TRUE;
  player.y++;        // Переместить игрока вниз
}

if (keys[VK_UP] && (player.y>0) && (player.fx==player.x*60) && (player.fy==player.y*40))
{
  // Пометить текущую вертикальную границу как пройденную
  player.y--;        // Переместить игрока вверх
  vline[player.x][player.y]=TRUE;
}
```

Мы увеличиваем / уменьшаем точные fx и fy переменные игрока, так же как мы увеличиваем / уменьшаем точные fx и fy переменные противника.

Если значение fx игрока, меньше чем значение x игрока умноженное на 60, мы увеличиваем fx игрока, на шаг скорости нашей игры в зависимости от значения adjust.

Если значение fx игрока больше, чем x игрока умноженное на 60, мы уменьшаем fx игрока, на шаг скорости нашей игры в зависимости от значения adjust.

Если значение fy игрока, меньше чем y игрока умноженное на 40, мы увеличиваем fy игрока, на шаг скорости нашей игры в зависимости от значения adjust.

Если значение fy игрока, больше чем y игрока умноженное на 40, мы уменьшаем fy игрока, на шаг скорости нашей игры в зависимости от значения adjust.

```
if (player.fx<player.x*60)  // Точная позиция по оси X меньше чем назначенная позиция?
{
  player.fx+=steps[adjust]; // Увеличим точную позицию X
}
```

```
  if (player.fx>player.x*60)  // Точная позиция по оси X больше чем назначенная позиция?
  {
   player.fx-=steps[adjust]; // Уменьшим точную позицию X
  }

  if (player.fy<player.y*40)  // Точная позиция по оси Y меньше чем назначенная позиция?
  {
   player.fy+=steps[adjust]; // Увеличим точную позицию Y
  }

  if (player.fy>player.y*40)  // Точная позиция по оси Y больше чем назначенная позиция?
  {
   player.fy-=steps[adjust]; // Уменьшим точную позицию X
  }
 }
```

Если игра завершена, то будет запущен следующий небольшой раздел кода. Мы проверяем, нажатие клавиши пробел. Если это так, то мы присваиваем переменной gameover ЛОЖЬ (повторный запуск игры). Мы задаем значение переменной filled ИСТИНА. Это означает, что стадия окончена, вызывая сброс переменных игрока, вместе с противниками.

Мы задаем стартовый уровень равным 1, наряду с реальным отображенным уровнем (level2). Мы устанавливаем значение переменной stage равной 0. Мы делаем это, потому что после того, как компьютер видит, что сетка была заполнена, он будет думать, что Вы закончили стадию, и увеличит stage на 1. Поскольку мы устанавливаем stage в 0, то затем stage увеличивается, и станет равной 1 (точно, что мы хотим). Наконец мы устанавливаем lives обратно в 5.

```
 else               // Иначе
 {
  if (keys[' '])    // Если пробел нажат
  {
   gameover=FALSE; // gameover равно FALSE
   filled=TRUE;    // filled равно TRUE
   level=1;        // Стартовый уровень установим обратно в один
   level2=1;       // Отображаемый уровень также установим в один
   stage=0;        // Стадию игры установим в ноль
   lives=5;        // Количество жизней равно пяти
  }
 }
```

Код ниже проверяет, равен ли флаг filled ИСТИНА (означает, что сетка была заполнена). Переменная filled может быть установлена в ИСТИНУ одним из двух путей. Или сетка заполнена полностью и filled равно ИСТИНА, когда игра закончена, а пробел был нажат, чтобы перезапустить ее (код выше).

Если filled равно ИСТИНА, вначале мы проигрываем крутую мелодию завершения уровня. Я уже объяснял, как работает PlaySound(). На сей раз, мы будем проигрывать файл Complete.WAV из каталога DATA. Снова, мы используем SND_SYNC для реализации задержки перед запуском следующей стадии.

После того, как звук был проигран, мы увеличиваем stage на один, и проверяем, что stage не больше чем 3. Если stage больше чем 3, мы устанавливаем stage в 1, и увеличиваем внутренний уровень и видимый уровень на один.

Если внутренний уровень больше чем 3, мы устанавливаем внутренний уровень (level) равным 3, и увеличиваем lives на 1. Если Вы достаточно быстры, и закончили уровень с 3, Вы заслуживаете бесплатную жизнь :). После увеличения жизней мы проверяем, что игрок не имеет больше чем 5 жизней. Если жизней больше чем 5, мы сбрасываем число жизней до 5.

```
 if (filled)        // Если сетка заполнена?
 {
  PlaySound("Data/Complete.wav", NULL, SND_SYNC);  // Играем звук завершения уровня
  stage++;          // Увеличиваем Stage
  if (stage>3)      // Если Stage больше чем 3?
  {
   stage=1;         // Тогда Stage равно 1
```

```
    level++;        // Увеличим уровень
    level2++;       // Увеличим отображаемый уровень
    if (level>3)    // Если уровень больше чем 3?
    {
     level=3;      // Тогда Level равно 3
     lives++;      // Добавим игроку лишнюю жизнь
     if (lives>5)  // Если число жизней больше чем 5?
     {
      lives=5;    // Тогда установим Lives равной 5
     }
    }
   }
```

Затем мы сбрасываем все объекты (такие как игрок и враги). При этом игрока помещаем снова в левый верхний угол сетки, а противникам присваиваются случайные позиции на сетке.

Мы создаем два цикла (loop1 и loop2) для обхода сетки. В них мы присваиваем значения всем вертикальным и горизонтальным линиям в ЛОЖЬ. Если бы мы этого не делали, то, когда была бы запущенна следующая стадия, то игра бы думала, что сетка все еще заполнена.

Заметьте, что код, который мы используем, чтобы очистить сетку, похож на код, который мы используем, чтобы вывести сетку. Мы должны проверить, что линии не будут рисоваться за правым и нижним краем. Именно поэтому мы проверяем, что loop1 меньше чем 10 прежде, чем мы сбрасываем горизонтальные линии, и мы проверяем, что loop2 меньше чем 10 прежде, чем мы сбрасываем вертикальные линии.

```
   ResetObjects();                  // Сброс позиции Игрока / Противника

   for (loop1=0; loop1<11; loop1++)   // Цикл по X координатам сетки
   {
    for (loop2=0; loop2<11; loop2++) // Цикл по Y координатам сетки
    {
     if (loop1<10)                  // Если X координата меньше чем 10
     {
      hline[loop1][loop2]=FALSE;   // Задаем текущее горизонтальное значение в FALSE
     }
     if (loop2<10)                  // Если Y координата меньше чем 10
     {
      vline[loop1][loop2]=FALSE;   // Задаем текущее вертикальное значение в FALSE
     }
    }
   }
  }
```

Теперь мы проверяем, попал ли игрок в песочные часы. Если точная позиция fx игрока равна позиции x песочных часов умноженная на 60, и точная позиция fy игрока равна позиции y песочных часов умноженная на 40, и hourglass.fx равно 1 (т.е. песочные часы есть на экране), то тогда код ниже будет выполнен.

Первая строка кода - PlaySound("Data/freeze.wav",NULL, SND_ASYNC | SND_LOOP). В этой строке проигрывается файл freeze.WAV из каталога DATA. Обратите внимание на то, что мы на этот раз используем SND_ASYNC. Мы хотим, чтобы звук замораживания играл без остановки игры. Флаг SND_LOOP позволяет циклично повторять звук, пока мы не сообщим, что пора прекратить играть, или пока не будет запущен другой звук.

После того, как мы запустили проигрывание звука, мы задаем hourglass.fx в 2. Когда hourglass.fx равно 2, песочные часы исчезнут, враги замрут, и звук будет непрерывно играть.

Мы также устанавливаем hourglass.fy в 0. Переменная hourglass.fy - счетчик. Когда она достигнет некоторого значения, значение переменной hourglass.fx изменится.

```
  // Если игрок попал в песочные часы и они на экране
   if ((player.fx==hourglass.x*60) && (player.fy==hourglass.y*40) && (hourglass.fx==1))
   {
    // Играть звук замораживания
```

```
    PlaySound("Data/freeze.wav", NULL, SND_ASYNC | SND_LOOP);
    hourglass.fx=2;          // Задать hourglass fx значение 2
    hourglass.fy=0;          // Задать hourglass fy значение 0
  }
```

В этой небольшой части кода увеличивает значение вращения игрока наполовину скорости выполнения игры. Если player.spin больше чем 360.0f, мы вычитаем 360.0f из player.spin. Это предохраняет значение player.spin от переполнения.

```
  player.spin+=0.5f*steps[adjust]; // Вращение игрока по часовой
  if (player.spin>360.0f)          // Значение spin больше чем 360?
  {
    player.spin-=360;              // Тогда вычтем 360
  }
```

Код ниже уменьшает значение вращения песочных часов на 1/4 скорости выполнения игры. Если hourglass.spin меньше чем 0.0f, мы добавляем 360.0f. Мы не хотим, чтобы hourglass.spin принимало отрицательные значения.

```
  hourglass.spin-=0.25f*steps[adjust]; // Вращение часов против часовой
  if (hourglass.spin<0.0f)             // spin меньше чем 0?
  {
    hourglass.spin+=360.0f;            // Тогда добавим 360
  }
```

В первой строке ниже увеличивается счетчик песочных часов, как я говорил об этом. Переменная hourglass.fy увеличивается на скорость игры (она равна значению шага в зависимости от значения корректировки).

Во второй линии проверяется, равно ли hourglass.fx значению 0 (не видимы) и счетчик песочных часов (hourglass.fy) больше чем 6000 деленное на текущий внутренний уровень (level).

Если значение fx равно 0, и счетчик больше чем 6000 деленное на внутренний уровень, то мы проигрываем файл hourglass.WAV из каталога DATA. Мы не хотим, чтобы игра остановилась, поэтому мы используем SND_ASYNC. Мы не будем повторять звук на этот раз, поэтому после того как звук проиграл, он не будет играть снова.

После того, как мы проиграли звук, мы задаем песочным часам случайное положение по оси X. Мы добавляем единицу к случайному значению, для того чтобы песочные часы не появились на стартовой позиции игрока в верхнем углу сетки. Мы также задаем песочным часам случайное положение по оси Y. Мы устанавливаем hourglass.fx в 1, это заставит песочные часы появиться на экране в этом новом местоположении. Мы также сбрасываем hourglass.fy в ноль, поэтому можно запустить счетчик снова.

Это приведет к тому, что песочные часы появятся на экране после заданного времени.

```
  hourglass.fy+=steps[adjust];    // Увеличим hourglass fy
   // Если hourglass fx равно 0 и fy больше чем 6000 деленное на текущий уровень?
   if ((hourglass.fx==0) && (hourglass.fy>6000/level))
   {
     // Тогда играем звук песочных часов
     PlaySound("Data/hourglass.wav", NULL, SND_ASYNC);
     hourglass.x=rand()%10+1;      // Случайная позиция часов по X
     hourglass.y=rand()%11;        // Случайная позиция часов по Y
     hourglass.fx=1;               // Задать hourglass fx значение 1 (стадия часов)
     hourglass.fy=0;               // Задать hourglass fy значение 0 (счетчик)
   }
```

Если hourglass.fx равно нолю, и hourglass.fy больше чем 6000 деленное на текущий внутренний уровнь (level), мы сбрасываем hourglass.fx назад в 0, что приводит к тому, что песочные часы исчезают. Мы также устанавливаем hourglass.fy в 0, потому что можно начать счет снова.

Это приводит к тому, что песочные часы исчезнут, если Вы не получаете их после некоторого времени.

```
    // Если hourglass fx равно 1 и fy больше чем 6000 деленное на текущий уровень?

    if ((hourglass.fx==1) && (hourglass.fy>6000/level))
    {
      hourglass.fx=0;         // Тогда зададим fx равным 0 (Обратим часы в ноль)
      hourglass.fy=0;         // Задать fy равным 0 (Сброс счетчика)
    }
```

Теперь мы проверяем, окончилось ли время 'замораживания противников' после того, как игрок коснулся песочных часов.

Если hourglass.fx равняется 2, и hourglass.fy больше чем 500 плюс 500 умноженное на текущий внутренний уровень, мы прерываем звук заморозки, который беспрерывно проигрывается. Мы прерываем звук командой PlaySound(NULL, NULL, 0). Мы устанавливаем hourglass.fx снова в 0, и hourglass.fy в 0. После присваивания fx и fy к 0 происходит запуск цикла работы песочных часов снова. Значение fy будет равняться 6000 деленное на текущий внутренний уровень прежде, чем песочные часы появятся снова.

```
    // Переменная песочных часов fx равно 2 и переменная fy
    // больше чем 500 плюс 500 умноженное на текущий уровень?
    if ((hourglass.fx==2) && (hourglass.fy>500+(500*level)))
    {
      PlaySound(NULL, NULL, 0);// Тогда прерываем звук заморозки
      hourglass.fx=0;         // Все в ноль
      hourglass.fy=0;
    }
```

Последнее что надо сделать - увеличить переменную задержки. Если Вы помните, задержка используется, чтобы обновить передвижение и анимацию игрока. Если наша программа финишировала, нам надо уничтожить окно и произвести возврат на рабочий стол.

```
    delay++;        // Увеличение счетчика задержки противника
  }
 }

 // Shutdown
 KillGLWindow();      // Уничтожить окно
 return (msg.wParam); // Выход из программы
}
```

Я потратил много времени при написании этого урока. Вначале это был урок по линиям, а в дальнейшем он перерос в небольшую интересную игру. Буду надеяться, если Вы сможете использовать то, что Вы узнали в этом уроке в ваших проектах с OpenGL. Я знаю, что Вы часто просили рассказать об играх на основе мозаики (tile). Отлично Вы не сможете сделать что-то более мозаичное, чем это :). Я также получил много писем, в которых меня спрашивали, как сделать точное по пиксельное рисование. Я думаю, что охватил и это :). Наиболее важно, что этот урок не только преподает Вам новые сведения о OpenGL, но также рассказывает Вам, как использовать простые звуки, чтобы добавить немного возбуждения в ваши визуальные произведения искусства! Я надеюсь, что Вам понравился этот урок. Если Вы чувствуете, что я неправильно прокомментировал кое-что или что код мог быть лучше в некоторых разделах, пожалуйста, сообщите мне об этом. Я хочу сделать самые хорошие уроки по OpenGL, я могу и я заинтересованным в общении с вами.

Пожалуйста, обратите внимание, это был чрезвычайно большой проект. Я пробовал комментировать все настолько ясно, насколько это возможно, но облекать мысли в слова, не столь просто, как это кажется. Знать о том, почему это все работает, и пробовать это объяснить – это совершенно разные вещи :). Если Вы прочитали урок, и можете объяснить все лучше, или если Вы знаете, способы помочь мне, пожалуйста, пошлите мне свои предложения. Я хочу, чтобы этот урок был прост. Также обратите внимание, что этот урок не для новичка. Если Вы не читали предыдущие уроки, пожалуйста, не задавайте мне вопросов. Большое спасибо.

Урок 22. Наложение микрорельефа методом тиснения, мультитекстурирование и использование расширений OpenGL

Bump-Mapping, Multi-Texturing & Extensions

Этот урок, написанный Дженсом Шнайдером (Jens Schneider), основан на материале Урока 6, но содержит существенные изменения и дополнения. Здесь вы узнаете:
как управлять функциями мультитекстурирования видеокарты;
как выполнять "поддельное" наложение микрорельефа методом тиснения;
как, используя смешивание, отображать эффектно смотрящиеся логотипы, "летающие" по просчитанной сцене;
как просто и быстро выполнять преобразование матриц;
познакомитесь с основами техники многопроходной визуализации.

По меньшей мере, три из перечисленных пунктов могут быть отнесены к "продвинутым техникам текстурирования", поэтому для работы с ними нужно хорошо понимать основы функционирования конвейера визуализации в OpenGL. Требуется знать большинство команд, изученных в предыдущих уроках и быть достаточно близко знакомым с векторной математикой. Иногда будут попадаться блоки, озаглавленные "начало теории(…)" и оканчивающиеся фразой "конец теории(…)". В таких местах рассматриваются теоретические основы вопросов, указанных в скобках, и если вы их знаете, то можете пропустить. Если возникают проблемы с пониманием кода, лучше вернуться к теоретической части и попробовать разобраться. И еще: в уроке более 1200 строк кода, значительные фрагменты которого очевидны и скучны для тех, кто читал предыдущие главы. Поэтому я не стал комментировать каждую строку, пояснил только главное. Если встретите что-то вроде >…<, это значит, что строки кода были пропущены.

Итак:

```
#include <windows.h>        // Файл заголовков функций Windows
#include <stdio.h>          // Файл заголовков для библиотеки ввода-вывода
#include <gl\gl.h>          // Файл заголовков для библиотеки OpenGL32
#include <gl\glu.h>         // Файл заголовков для библиотеки GLu32
#include <gl\glaux.h>       // Файл заголовков для библиотеки GLaux
#include "glext.h"          // Файл заголовков для мультитекстурирования
#include <string.h>         // Файл заголовков для работы со строками
#include <math.h>           // Файл заголовков для математической библиотеки
```

Параметр GLfloat MAX_EMBOSS задает "интенсивность" эффекта рельефности. Увеличение этого числа значительно усиливает эффект, но приводит к снижению качества и появлению так называемых "артефактов" изображения по краям поверхностей.

```
// Коэффициент рельефности. Увеличьте, чтобы усилить эффект
#define MAX_EMBOSS (GLfloat)0.008f
```

Давайте подготовимся к использованию расширения GL_ARB_multitexture. Это просто.

В настоящее время подавляющая часть акселераторов имеет более одного блока текстурирования на чипе. Чтобы определить, верно ли это для используемой карточки, надо проверить ее на поддержку опции GL_ARB_multitexture, которая позволяет накладывать две или более текстур на примитив за один проход. Звучит не слишком впечатляюще, но на самом деле это мощный инструмент! Практически любая сцена выглядит гораздо красивее, если ее на объекты наложено несколько текстур. Обычно для этого требуется сделать несколько "проходов", состоящих из выбора текстуры и отрисовки геометрии; при увеличении числа таких операций работа серьезно тормозится. Однако не беспокойтесь, позже все прояснится.

Вернемся к коду: __ARB_ENABLE используется, чтобы при необходимости отключить мультитекстурирование. Если хотите видеть OpenGL-расширения, раскомментируйте строку #define EXT_INFO. Доступность расширений будет проверяться во время выполнения, чтобы сохранить переносимость кода, поэтому нужны будут несколько переменных строкового типа — они заданы двумя следующими строками. Кроме того, желательно различать доступность мультитекстурирования и его использование, то есть нужны еще два флага. Наконец, нужно знать, сколько блоков текстурирования доступно (хотя мы будем использовать всего два). По меньшей мере один такой блок обязательно присутствует на любой OpenGL-совместимой карте, так что переменную maxTexelUnits надо инициализировать единичкой.

```
#define __ARB_ENABLE true          // Используется, чтобы полностью отключить расширения
// #define EXT_INFO        // Раскомментируйте, чтобы увидеть при запуске доступные расширения
#define MAX_EXTENSION_SPACE 10240  // Символы строк-описателей расширений
#define MAX_EXTENSION_LENGTH 256   // Максимальное число символов в одной строке-описателе

bool multitextureSupported=false;  // Флаг, определяющий, поддерживается ли мультитекстурирование
bool useMultitexture=true;         // Использовать его, если оно доступно?
GLint maxTexelUnits=1;             // Число текстурных блоков. Как минимум 1 есть всегда
```

Следующие строки нужны, чтобы сопоставить расширениям соответствующие вызовы функций C++. Просто считайте, что PFN-и-как-там-дальше — предварительно определенный тип данных, нужный для описания вызовов функций. Мы не уверены, что к этим прототипам будут доступны функции, а потому установим их в NULL. Команды glMultiTexCoordifARB задают привязку к хорошо известным glTexCoordif, описывающим i-мерные текстурные координаты. Заметьте, что они могут полностью заменить команды, связанные с glTexCoordif. Мы пользуемся версиями с GLfloat, и нам нужны прототипы тех команд, которые оканчиваются на "f"; другие команды при этом также остаются доступны ("fv", "i" и т.д.). Два последних прототипа задают функцию выбора активного блока текстурирования (texture-unit), занятого привязкой текстур (glActiveTextureARB()), и функцию, определяющую, какой из текстурных блоков связан с командой выбора указателя на массив (glClientActiveTextureARB). К слову: ARB — это сокращение от "Architectural Review Board", "комитет по архитектуре". Расширения, содержащие в имени строку ARB, не требуются для реализации системы, соответствующей спецификации OpenGL, но ожидается, что такие расширения найдут широкую поддержку у производителей. Пока ARB-статус имеют только расширения, связанные с мультитекстурированием. Такая ситуация, скорее всего, указывает на то, что мультитекстурирование наносит страшный удар по производительности, когда дело касается некоторых продвинутых техник визуализации.

Пропущенные строки относятся к указателям на контекст GDI и прочему.

```
PFNGLMULTITEXCOORD1FARBPROC  glMultiTexCoord1fARB  = NULL;
PFNGLMULTITEXCOORD2FARBPROC  glMultiTexCoord2fARB  = NULL;
PFNGLMULTITEXCOORD3FARBPROC  glMultiTexCoord3fARB  = NULL;
PFNGLMULTITEXCOORD4FARBPROC  glMultiTexCoord4fARB  = NULL;
PFNGLACTIVETEXTUREARBPROC  glActiveTextureARB  = NULL;
PFNGLCLIENTACTIVETEXTUREARBPROC  glClientActiveTextureARB= NULL;
```

Создаем глобальные переменные:
filter задает используемый фильтр (см. Урок 06). Обычно берем GL_LINEAR, поэтому инициализируем переменную единичкой.
texture хранит текстуру, три копии, по одной на фильтр.
bump хранит карты микрорельефа.
invbump хранит инвертированную карту микрорельефа. Причина объясняется позже, в теоретическом разделе.
Переменные, относящиеся к логотипам, в имени которых есть слово "Logo" - хранят текстуры, добавляемые к сцене на последнем проходе.
Переменные, относящиеся к свету, в имени которых есть слово “Light” - хранят параметры источника света.

```
GLuint  filter=1;          // Какой фильтр использовать
GLuint  texture[3];        // Хранит 3 текстуры
GLuint  bump[3];           // Рельефы
GLuint  invbump[3];        // Инвертированные рельефы
GLuint  glLogo;            // Указатель на OpenGL-логотип
GLuint  multiLogo;         // Указатель на мультитекстурированный логотип
GLfloat LightAmbient[]  = { 0.2f, 0.2f, 0.2f};  // Фоновое освещение — 20% белого
GLfloat LightDiffuse[]  = { 1.0f, 1.0f, 1.0f};  // Рассеянный свет —  чисто белый
GLfloat LightPosition[]  = { 0.0f, 0.0f, 2.0f}; // Положение источника — перед экраном
GLfloat Gray[]    = { 0.5f, 0.5f, 0.5f, 1.0f};
```

Очередной фрагмент кода содержит числовое описание текстурированного куба, сделанного из GL_QUADS-ов. Каждые пять чисел представляют собой пару из двумерных текстурных и трехмерных вершинных координат. Это удобно для построения куба в цикле for…, учитывая, что нам потребуется сделать это несколько раз. Блок данных заканчивается прототипом функции WndProc(), хорошо известной из предыдущих уроков.

```
// Данные содержат грани куба в формате "2 текстурные координаты, 3 вершинные".
// Обратите внимание, что мозаичность куба минимальна.

GLfloat data[]= {
  // ЛИЦЕВАЯ ГРАНЬ
  0.0f, 0.0f,    -1.0f, -1.0f, +1.0f,
  1.0f, 0.0f,    +1.0f, -1.0f, +1.0f,
  1.0f, 1.0f,    +1.0f, +1.0f, +1.0f,
  0.0f, 1.0f,    -1.0f, +1.0f, +1.0f,
  // ЗАДНЯЯ ГРАНЬ
  1.0f, 0.0f,    -1.0f, -1.0f, -1.0f,
  1.0f, 1.0f,    -1.0f, +1.0f, -1.0f,
  0.0f, 1.0f,    +1.0f, +1.0f, -1.0f,
  0.0f, 0.0f,    +1.0f, -1.0f, -1.0f,
  // ВЕРХНЯЯ ГРАНЬ
  0.0f, 1.0f,    -1.0f, +1.0f, -1.0f,
  0.0f, 0.0f,    -1.0f, +1.0f, +1.0f,
  1.0f, 0.0f,    +1.0f, +1.0f, +1.0f,
  1.0f, 1.0f,    +1.0f, +1.0f, -1.0f,
  // НИЖНЯЯ ГРАНЬ
  1.0f, 1.0f,    -1.0f, -1.0f, -1.0f,
  0.0f, 1.0f,    +1.0f, -1.0f, -1.0f,
  0.0f, 0.0f,    +1.0f, -1.0f, +1.0f,
  1.0f, 0.0f,    -1.0f, -1.0f, +1.0f,
  // ПРАВАЯ ГРАНЬ
  1.0f, 0.0f,    +1.0f, -1.0f, -1.0f,
  1.0f, 1.0f,    +1.0f, +1.0f, -1.0f,
  0.0f, 1.0f,    +1.0f, +1.0f, +1.0f,
  0.0f, 0.0f,    +1.0f, -1.0f, +1.0f,
  // ЛЕВАЯ ГРАНЬ
  0.0f, 0.0f,    -1.0f, -1.0f, -1.0f,
  1.0f, 0.0f,    -1.0f, -1.0f, +1.0f,
  1.0f, 1.0f,    -1.0f, +1.0f, +1.0f,
  0.0f, 1.0f,    -1.0f, +1.0f, -1.0f
};

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);        // Объявление WndProc
```

В следующем блоке кода реализована проверка поддержки расширений во время выполнения.
Во-первых, предположим, что у нас есть длинная строка, содержащая список всех поддерживаемых расширений, представленных в виде подстрок, разделенных символом ‘\n’. Таким образом, надо провести поиск этого символа и начать сравнение string с search до достижения либо очередного ‘\n’, либо отличия в сравниваемых строках. В первом случае вернем в "найдено" значение true, во втором — возьмем следующую подстроку, и так до тех пор, пока не кончится string. Со string придется немного повозиться, поскольку она начинается не с символа ‘\n’.
Кстати: проверку доступности любого данного расширения во время выполнения программы надо выполнять ВСЕГДА!

```
bool isInString(char *string, const char *search) {
  int pos=0;
  int maxpos=strlen(search)-1;
  int len=strlen(string);
  char *other;
  for (int i=0; i<len; i++) {
    if ((i==0) || ((i>1) && string[i-1]=='\n')) { // Новые расширения начинаются здесь!
      other=&string[i];
      pos=0; // Начать новый поиск
      while (string[i]!='\n') { // Поиск по всей строке расширения
        if (string[i]==search[pos]) pos++; // Следующий символ
        if ((pos>maxpos) && string[i+1]=='\n') return true; // А вот и она!
        i++;
      }    }    }
  return false; // Простите, не нашли!
}
```

Теперь извлечем строку расширений и преобразуем ее в строки, разделенные символом ‘\n’, чтобы провести поиск. Если будет обнаружена строка ”GL_ARB_multitexture”, значит, эта опция поддерживается. Но чтобы ее использовать, нужно, во-первых, чтобы __ARB_ENABLE была установлена в true, а во-вторых, чтобы карточка поддерживала расширение GL_EXT_texture_env_combine, которое указывает, что аппаратура разрешает некоторые новые способы взаимодействия между своими текстурными блоками. Это необходимо, поскольку GL_ARB_multitexture обеспечивает лишь вывод обработанных данных последовательно с текстурного блока с меньшим номером на блок с большим, а поддержка GL_EXT_texture_env_combine означает возможность использования уравнений смешивания повышенной сложности, эффект от которых совсем другой. Если все необходимые расширения поддерживаются и мы не запретили их сами, определим количество доступных текстурных блоков. Это число будет храниться в maxTexelUnits. Затем установим связь между функциями и их именами, для этого воспользуемся вызовом wglGetProcAdress(), передавая ей в качестве параметра строку-имя искомой функции и проводя преобразование типа результата, чтобы гарантировать совпадение ожидаемого и полученного типов.

```
bool initMultitexture(void) {
  char *extensions;
  extensions=strdup((char *) glGetString(GL_EXTENSIONS)); // Получим строку расширений
  int len=strlen(extensions);
  for (int i=0; i<len; i++) // Разделим ее символами новой строки вместо пробелов
    if (extensions[i]==' ') extensions[i]='\n';

#ifdef EXT_INFO
  MessageBox(hWnd,extensions,"поддерживаются расширения GL:",MB_OK | MB_ICONINFORMATION);
#endif

  if (isInString(extensions,"GL_ARB_multitexture") // Мультитекстурирование поддерживается?
    && __ARB_ENABLE  // Проверим флаг
    // Поддерживается среда комбинирования текстур?
    && isInString(extensions,"GL_EXT_texture_env_combine"))
  {
    glGetIntegerv(GL_MAX_TEXTURE_UNITS_ARB,&maxTexelUnits);
    glMultiTexCoord1fARB = (PFNGLMULTITEXCOORD1FARBPROC) wglGetProcAddress("glMultiTexCoord1fARB");
    glMultiTexCoord2fARB = (PFNGLMULTITEXCOORD2FARBPROC) wglGetProcAddress("glMultiTexCoord2fARB");
    glMultiTexCoord3fARB = (PFNGLMULTITEXCOORD3FARBPROC) wglGetProcAddress("glMultiTexCoord3fARB");
    glMultiTexCoord4fARB = (PFNGLMULTITEXCOORD4FARBPROC) wglGetProcAddress("glMultiTexCoord4fARB");
    glActiveTextureARB   = (PFNGLACTIVETEXTUREARBPROC) wglGetProcAddress("glActiveTextureARB");
    glClientActiveTextureARB= (PFNGLCLIENTACTIVETEXTUREARBPROC)
wglGetProcAddress("glClientActiveTextureARB");

#ifdef EXT_INFO
    MessageBox(hWnd,"Будет использовано расширение GL_ARB_multitexture.",
     "опция поддерживается!",MB_OK | MB_ICONINFORMATION);
#endif
    return true;
  }
  useMultitexture=false;// Невозможно использовать то, что не поддерживается аппаратурой
  return false;
}
```

InitLights() инициализирует освещение OpenGL, будучи вызвана позже из InitGL().

```
void initLights(void) {
        // Загрузка параметров освещения в GL_LIGHT1
        glLightfv(GL_LIGHT1, GL_AMBIENT, LightAmbient);
        glLightfv(GL_LIGHT1, GL_DIFFUSE, LightDiffuse);
        glLightfv(GL_LIGHT1, GL_POSITION, LightPosition);
        glEnable(GL_LIGHT1);
}
```

Здесь грузится УЙМА текстур. Поскольку у функции auxDIBImageLoad() есть собственный обработчик ошибок, а LoadBMP() труднопредсказуема и требует блока try-catch, я отказался от нее. Но вернемся к процедуре загрузки. Сначала берем базовую картинку и создаем на ее основе три фильтрованных текстуры (в режимах GL_NEAREST, GL_LINEAR и GL_LINEAR_MIPMAP_NEAREST). Обратите внимание, для хранения растра используется лишь один экземпляр структуры данных, поскольку в один момент открытой нужна лишь одна картинка. Здесь применяется

новая структура данных, alpha — в ней содержится альфа-слой текстур. Такой подход позволяет хранить RGBA-изображения в виде двух картинок: основного 24-битного RGB растра и 8-битного альфа-канала в шкале серого. Чтобы индикатор состояния работал корректно, нужно удалять Image-блок после каждой загрузки и сбрасывать его в NULL.

Еще одна особенность: при задании типа текстуры используется GL_RGB8 вместо обычного "3". Это сделано для совместимости с будущими версиями OpenGL-ICD и рекомендуется к использованию вместо любого другого числа. Такие параметры я пометил оранжевым.

```
int LoadGLTextures() {          // Загрузка растра и преобразование в текстуры
  bool status=true;             // Индикатор состояния
  AUX_RGBImageRec *Image=NULL;   // Создадим место для хранения текстур
  char *alpha=NULL;

  // Загрузим базовый растр
  if (Image=auxDIBImageLoad("Data/Base.bmp")) {
    glGenTextures(3, texture);    // Создадим три текстуры

    // Создаем текстуру с фильтром по ближайшему
    glBindTexture(GL_TEXTURE_2D, texture[0]);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB8, Image->sizeX, Image->sizeY, 0,
      GL_RGB, GL_UNSIGNED_BYTE, Image->data);

    // Создаем текстуру с фильтром усреднения
    glBindTexture(GL_TEXTURE_2D, texture[1]);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB8, Image->sizeX, Image->sizeY, 0,
       GL_RGB, GL_UNSIGNED_BYTE, Image->data);

    // Создаем текстуру с мип-наложением
    glBindTexture(GL_TEXTURE_2D, texture[2]);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR_MIPMAP_NEAREST);
    gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGB8, Image->sizeX, Image->sizeY,
      GL_RGB, GL_UNSIGNED_BYTE, Image->data);
  }
  else status=false;

  if (Image) { // Если текстура существует
    if (Image->data) delete Image->data; // Если изображение существует
    delete Image;
    Image=NULL;
  }
```

Загрузим рельеф. По причинам, объясняемым ниже, текстура рельефа должна иметь 50% яркость, поэтому ее надо промасштабировать. Сделаем это через команды glPixelTransferf(), которые описывают попиксельное преобразование данных растра в текстуру. Если вы до сих пор не пользовались командами семейства glPixelTransfer(), рекомендую обратить на них пристальное внимание, поскольку они часто бывают очень удобны и полезны.

Теперь учтем, что нам не нужно, чтобы базовая картинка многократно повторялась в текстуре. Чтобы получить картинку единожды, растянутой в нужное количество раз, ее надо привязать к текстурным координатам с (s,t)=(0.0f, 0.0f) по (s,t)=(1.0f, 1.0f). Все остальные координаты привязываются к чистому черному цвету через вызовы glTexParameteri(), которые даже не требуют пояснений.

```
  // Загрузим рельефы
  if (Image=auxDIBImageLoad("Data/Bump.bmp")) {
    glPixelTransferf(GL_RED_SCALE,0.5f);   // Промасштабируем яркость до 50%,
    glPixelTransferf(GL_GREEN_SCALE,0.5f); // поскольку нам нужна половинная интенсивность
    glPixelTransferf(GL_BLUE_SCALE,0.5f);
```

```
  glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_CLAMP);  // Не укладывать паркетом
  glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_CLAMP);
  glGenTextures(3, bump);  // Создать три текстуры

  // Создать текстуру с фильтром по ближайшему
  >…<

  // Создать текстуру с фильтром усреднения
  >…<

  // Создать текстуру с мип-наложением
  >…<
```

С этой фразой вы уже знакомы: по причинам, объясненным ниже, нужно создать инвертированную карту рельефа с той же 50% яркостью. Для этого вычтем полученный ранее растр из чистого белого цвета {255, 255, 255}. Поскольку мы НЕ возвращали RGB-масштабирование на 100% уровень (я три часа разбирался, пока понял, что здесь скрывалась основная ошибка первой версии урока!), инверсный рельеф тоже получится 50% яркости.

```
  for (int i=0; i<3*Image->sizeX*Image->sizeY; i++) // Проинвертируем растр
    Image->data[i]=255-Image->data[i];
  glGenTextures(3, invbump); // Создадим три текстуры

  // с фильтром по ближайшему
  >…<

  // с фильтром усреднения
  >…<

  // с мип-наложением
  >…<
 }
 else status=false;
 if (Image) { // Если текстура существует
  if (Image->data) delete Image->data; // Если изображение текстуры существует
  delete Image;
  Image=NULL;
 }
```

Загрузка изображения логотипа очень проста, кроме, разве что, фрагмента рекомбинации RGB-A. Он, впрочем, тоже достаточно очевиден. Заметьте, что текстура строится на основе alpha-, а не Image-блока! Здесь применена только одна фильтрация.

```
 // Загрузка картинки логотипа
 if (Image=auxDIBImageLoad("Data/OpenGL_ALPHA.bmp")) {
  alpha=new char[4*Image->sizeX*Image->sizeY];
  // Выделим память для RGBA8-текстуры
  for (int a=0; a<Image->sizeX*Image->sizeY; a++)
   alpha[4*a+3]=Image->data[a*3]; // Берем красную величину как альфа-канал
  if (!(Image=auxDIBImageLoad("Data/OpenGL.bmp"))) status=false;
  for (a=0; a<Image->sizeX*Image->sizeY; a++) {
   alpha[4*a]=Image->data[a*3];       // R
   alpha[4*a+1]=Image->data[a*3+1];   // G
   alpha[4*a+2]=Image->data[a*3+2];   // B
  }

  glGenTextures(1, &glLogo);  // Создать одну текстуру
  // Создать RGBA8-текстуру с фильтром усреднения
  glBindTexture(GL_TEXTURE_2D, glLogo);
  glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, Image->sizeX, Image->sizeY, 0,
    GL_RGBA, GL_UNSIGNED_BYTE, alpha);
```

```
  delete alpha;
 }
 else status=false;

 if (Image) { // Если текстура существует
  if (Image->data) delete Image->data; // Если изображение текстуры существует
  delete Image;
  Image=NULL;
 }

 // Загрузим логотип "Extension Enabled"
 if (Image=auxDIBImageLoad("Data/multi_on_alpha.bmp")) {
  alpha=new char[4*Image->sizeX*Image->sizeY]; // Выделить память для RGBA8-текстуры
  >…<

  glGenTextures(1, &multiLogo); // Создать одну текстуру
  // Создать RGBA8-текстуру с фильтром усреднения
  >…<
  delete alpha;
 }
 else status=false;

 if (Image) { // Если текстура существует
  if (Image->data) delete Image->data; // Если изображение текстуры существует
  delete Image;
  Image=NULL;
 }
 return status;  // Вернем состояние
}
```

Далее идет практически единственная неизмененная функция ReSizeGLScene(), и ее я пропустил. За ней следует функция doCube(), рисующая куб с единичными нормалями. Она задействует только текстурный блок №0, потому что glTexCoord2f(s,t) делает то же самое, что и glMultiTexCoord2f(GL_TEXTURE0_ARB,s,t). Обратите внимание, что куб нельзя создать, используя чередующиеся массивы, но это тема для отдельного разговора. Кроме того, учтите, что куб НЕВОЗМОЖНО создать, пользуясь списками отображения. Видимо, точность внутреннего представления данных, используемая в этих списках, не соответствует точности, применяемой в GLfloat. Это ведет к неприятным эффектам, которые называются проблемами деколирования (когда источник света не влияет на закрашивание объекта), поэтому от списков я решил отказаться. Вообще, я полагаю, что надо либо делать всю геометрию, пользуясь списками, либо не применять их вообще. Смешивание разных подходов приводит к проблемам, которые где-нибудь да проявятся, даже если на вашей аппаратуре все пройдет успешно.

```
GLvoid ReSizeGLScene(GLsizei width, GLsizei height)
// Изменить размер и инициализировать окно GL
>…<

void doCube (void) {
 int i;
 glBegin(GL_QUADS);
  // Передняя грань
  glNormal3f( 0.0f, 0.0f, +1.0f);
  for (i=0; i<4; i++) {
   glTexCoord2f(data[5*i],data[5*i+1]);
   glVertex3f(data[5*i+2],data[5*i+3],data[5*i+4]);
  }
  // Задняя грань
  glNormal3f( 0.0f, 0.0f,-1.0f);
  for (i=4; i<8; i++) {
   glTexCoord2f(data[5*i],data[5*i+1]);
   glVertex3f(data[5*i+2],data[5*i+3],data[5*i+4]);
  }
```

```
    // Верхняя грань
    glNormal3f( 0.0f, 1.0f, 0.0f);
    for (i=8; i<12; i++) {
      glTexCoord2f(data[5*i],data[5*i+1]);
      glVertex3f(data[5*i+2],data[5*i+3],data[5*i+4]);
    }

    // Нижняя грань
    glNormal3f( 0.0f,-1.0f, 0.0f);
    for (i=12; i<16; i++) {
      glTexCoord2f(data[5*i],data[5*i+1]);
      glVertex3f(data[5*i+2],data[5*i+3],data[5*i+4]);
    }

    // Правая грань
    glNormal3f( 1.0f, 0.0f, 0.0f);
    for (i=16; i<20; i++) {
      glTexCoord2f(data[5*i],data[5*i+1]);
      glVertex3f(data[5*i+2],data[5*i+3],data[5*i+4]);
    }

    // Левая грань
    glNormal3f(-1.0f, 0.0f, 0.0f);
    for (i=20; i<24; i++) {
      glTexCoord2f(data[5*i],data[5*i+1]);
      glVertex3f(data[5*i+2],data[5*i+3],data[5*i+4]);
    }
  glEnd();
}
```

Время инициализировать OpenGL. Все как в Уроке 06, кроме вызова initLights() вместо прямой инициализации источников света в теле функции. Да, и еще одно: я выполняю здесь настройку мультитекстурирования.

```
int InitGL(GLvoid)                  // Все настройки OpenGL проходят здесь
{
  multitextureSupported=initMultitexture();
  if (!LoadGLTextures()) return false;  // Переход к процедуре загрузки текстур
  glEnable(GL_TEXTURE_2D);            // Включить привязку текстур
  glShadeModel(GL_SMOOTH);            // Включит сглаживание
  glClearColor(0.0f, 0.0f, 0.0f, 0.5f); // Черный фон
  glClearDepth(1.0f);                 // Установка буфера глубины
  glEnable(GL_DEPTH_TEST);            // Включить проверку глубины
  glDepthFunc(GL_LEQUAL);             // Тип проверки глубины
  glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // Наилучшая коррекция перспективы
  initLights();                       // Инициализация освещения OpenGL
  return true                         // Инициализация закончилась успешно
}
```

95% всей работы содержится здесь. Все, что упоминалось под грифом "по причинам, объясненным ниже", будет расписано в этом теоретическом блоке.
Начало теории (Наложение микрорельефа методом тиснения)

Если у вас установлен просмотрщик Powerpoint-презентаций, я настоятельно рекомендую скачать следующую презентацию:

"Emboss Bump Mapping" by Michael I. Gold, nVidia Corp. [.ppt, 309K]

Для тех, у кого нет просмотрщика, я попытался перевести презентацию в html-формат. Вот она:

Наложение микрорельефа методом тиснения

Майкл И. Голд

Корпорация Nvidia

Наложение микрорельефа (bump mapping)

Действительное наложение микрорельефа использует попиксельные вычисления освещенности
Вычисление освещенности в каждой точке базируется на возмущенном векторе нормали.
Вычисления весьма ресурсоемкие.
Более детальное описание читайте здесь: Blinn, J. : Simulation of Wrinkled Surfaces (Моделирование складчатых поверхностей), Computer Graphics. 12,3 (August 1978) 286-292.
Информация в сети: на http://www.objectecture.com/ лежит Cass Everitt's Orthogonal Illumination Thesis (Диссертация по ортогональному освещению Касса Эверитта).

Наложение микрорельефа методом тиснения (emboss bump mapping)

Микрорельеф, наложенный тиснением, похож на резьбу по материалу
Учитывается только рассеянный свет, нет зеркальной составляющей
Возможны артефакты изображения из-за недостаточного размера текстуры рельефа (в результате, например, движение приводит к сильному размытию — прим. Дженса)
Выполнение возможно на пользовательском оборудовании современного уровня (как показано — прим. Дженса)
Если рельеф выглядит хорошо, используйте его!

Расчет рассеяния света

C=(L*N) x Dl x Dm
L — вектор освещения
N — вектор нормали
Dl — цвет падающего света
Dm — цвет рассеяния материала
При наложении микрорельефа попиксельно меняется N
При наложении микрорельефа методом тиснения используется аппроксимация (L*N)

Аппроксимация коэффициента рассеяния L*N

В текстурной карте содержится поле высот

[0,1] — диапазон значений, принимаемых функцией рельефности
Первая производная определяет величину уклона m (материала) в данной точке (Заметьте, что m — чисто одномерная величина. Считайте, что m — это оценка grad(s,t) (градиента) в данной точке — прим. Дженса)
m увеличивает/уменьшает базовый коэффициент рассеяния Fd
(Fd+m) приближенно определяет (L*N) для каждого пикселя

Приближенное вычисление производной

Используется приближенное вычисление производной
Берется высота H0 в точке (s,t)
Определяется высота H1 в точке, слегка сдвинутой в направлении источника света, (s+ds,t+dt)
Исходная высота H0 вычитается из возмущенной высоты H1
Разница является оценкой мгновенного угла наклона m=H1-H0

Вычисление рельефа

1) Исходный рельеф (H0).
2) На исходный рельеф (H0) накладывается другой, (H1), слегка сдвинутый в направлении источника света.
3) Из второго вычитается первый (H0-H1). Появляются освещенные (B, bright) и затемненные (D, dark) участки.

Вычисление освещенности
Вычисляется цвет фрагмента Cf
Cf = (L*N) x Dl x Dm
(L*N) ~ (Fd + (H1-H0))

Dm x Dl закодировано в текстуре поверхности Ct.
Если хватит соображения, можно управлять Dl по отдельности (мы управляем им, пользуясь освещением OpenGL — прим. Дженса)
Cf = (Fd + (H0-H1)) x Ct

И все? Так просто!

Нет, мы еще не закончили. Мы должны:
Нарисовать текстуру (в любом графическом редакторе — прим. Дженса)
Вычислить сдвиги координат текстуры (ds,dt)
Вычислить коэффициент рассеяния Fd (управляется с помощью освещения в OpenGL — прим. Дженса)
Обе величины используют вектора нормали N и освещения L (в нашем случае явным образом вычисляются только (ds,dt) — прим. Дженса)
Теперь займемся математикой

Создание текстуры

Берегите текстуры!
В настоящее время аппаратура мультитекстурирования поддерживает максимум две текстуры! (Это утверждение устарело, но его надо иметь в виду, если хотите сохранить обратную совместимость — прим. Дженса)
Рельеф использует канал АЛЬФА (у нас это не так; но если на вашей машине карточка с чипом TNT, можете попробовать повторить предложенное здесь самостоятельно — прим. Дженса)
Максимальная высота = 1.0
Уровень нулевой высоты = 0.5
Максимальная глубина = 0.0
Цвета поверхности — каналы RGB
Внутренний формат должен быть GL_RGBA8 !!

Вычисление смещения текстур

Отображение вектора освещения в пространство нормали
Нужно получить систему координат нормали
Создадим систему координат из нормали к поверхности и вектора "вверх" (мы передаем направления texCoord генератору смещения в явном виде — прим. Дженса)
Нормаль — ось z
Перпендикулярно ей идет ось x
Направление "вверх", или ось y, получена как произведение x- и z-векторов
Построим матрицу 3x3 Mn из осей
Отобразим вектор освещения в пространстве нормали.(Mn называют также ортонормальным базисом. Можете рассматривать Mn*v как представление v в базисе, формирующем касательное пространство, а не обычное. Заметьте, что ортонормальный базис инвариантен к масштабированию, то есть при умножении векторов нормализация не теряется! — прим. Дженса)

Вычисление смещения текстур (продолжение)

Используем вектор освещения в пространстве нормали для смещения
L’ = Mn x L
Используем L’x, L’y для (ds,dt)
Используем L’z как коэффициент диффузного отражения (Совсем нет! Если вы не владелец TNT-карточки, используйте освещение OpenGL, потому что вам обязательно придется выполнять дополнительный проход — прим. Дженса)
Если вектор освещения близок к нормали, L’x и L’y малы.
Если вектор освещения близок к касательной, L’x и L’y значительны.
Что, если L’z меньше нуля?
Свет на стороне, обратной к нормали
Приравняем его вклад к нулю

Реализация на TNT
Вычисления векторов и координат текстур на хосте
Передаем коэффициент рассеяния как alpha
Можно использовать цвет вершины для передачи цвета диффузного рассеяния источника света
H0 и цвет поверхности берем из текстурного блока 0
H1 берем из текстурного блока 1 (та же самая текстура, но с другими координатами)

Используем расширение ARB_multitexture
Это расширение для комбайнов (точнее, речь идет о расширении NVIDIA_multitexture_combiners, поддерживаемом всеми карточками семейства TNT — прим. Дженса)

Реализация на TNT (продолжение)

Первичная установка комбайна 0:
(1-T0a) + T1a - 0.5 (T0a означает "текстурный блок 0, альфа-канал" — прим. Дженса)
(T1a-T0a) отображается в диапазон (-1,1), но аппаратура сжимает его до (0,1)
Смещение на 0.5 балансирует потерю от сжатия (подумайте о применении масштабирования с коэффициентом 0.5, ведь можно использовать разные карты рельефа — прим. Дженса)
Цвет диффузного рассеяния источника света можно регулировать с помощью T0c

Установка RGB комбайна 1:
(T0c * C0a + T0c * Fda - 0.5)*2
Смещение на 0.5 балансирует потерю от сжатия
Умножение на 2 осветляет изображение

Конец теории (Наложение микрорельефа методом тиснения)

Мы у себя делаем все не совсем так, как это предложено для TNT, поскольку хотим, чтобы наша программа работала на любом железе, однако здесь есть пара-тройка ценных идей. Во-первых, то, что на большинстве карточек наложение рельефа — многопроходная операция (хотя это не относится к семейству TNT, где рельефность можно реализовать за один двухтекстурный проход). Сейчас вы, наверное, оценили, какая отличная вещь — возможность мультитекстурирования. Теперь мы напишем 3-проходный немультитекстурный алгоритм, который можно (и мы это сделаем) реализовать за два мультитекстурных прохода.

Кроме того, вы, вероятно, поняли, что нам придется проводить умножения матриц на матрицы и матриц на вектора. Но об этом можно не беспокоиться: в OpenGL операция умножения матриц реализована (если точность правильная) и умножения матрицы на вектор реализована в функции VMatMult(M,v), где матрица M умножается на вектор v и результат сохраняется в v, то есть v:=M*v. Все передаваемые матрицы и вектора должны быть гомогенны (то бишь в одной системе координат — прим. перев.) и представлять собой матрицы 4x4 и четырехмерные вектора. Такие требования гарантируют быстрое и правильное умножение векторов и матриц по правилам OpenGL.

// Вычисляет v=vM, M — матрица 4x4 в порядке столбец-строка, v — четырехмерный вектор-строка (т.е. транспонированный)

```
void VMatMult(GLfloat *M, GLfloat *v) {
  GLfloat res[3];
  res[0]=M[ 0]*v[0]+M[ 1]*v[1]+M[ 2]*v[2]+M[ 3]*v[3];
  res[1]=M[ 4]*v[0]+M[ 5]*v[1]+M[ 6]*v[2]+M[ 7]*v[3];
  res[2]=M[ 8]*v[0]+M[ 9]*v[1]+M[10]*v[2]+M[11]*v[3];
  v[0]=res[0];
  v[1]=res[1];
  v[2]=res[2];
  v[3]=M[15];  // Гомогенные координаты
}
```

Начало теории (Алгоритмы наложения микрорельефа методом тиснения)

Сейчас мы обсудим два разных алгоритма. Первый я нашел несколько дней назад здесь:
http://www.nvidia.com/marketing/Developer/DevRel.nsf/TechnicalDemosFrame?OpenPage

Программа называется GL_BUMP и была написана Диего Тартара (Diego Tartara) в 1999 году. Диего создал очень симпатичный пример наложения микрорельефа, хотя и не лишенный некоторых недостатков.

Однако давайте взглянем на алгоритм:
Все вектора должны быть заданы ЛИБО в координатах объекта, ЛИБО в мировых координатах.
Вычисляется вектор v направления из текущей вершины к источнику света
v нормализуется
v проецируется на касательную плоскость (Касательная плоскость — такая, которая касается поверхности в данной точке. Для нас эта точка — текущая вершина.).
(s,t) сдвигается на величины соответственно x и y координат спроецированного вектора v.

Выглядит неплохо! В основном здесь повторен алгоритм, предложенный Майклом Голдом — мы рассмотрели его в предыдущем теоретическом блоке. Однако у нового варианта есть существенный недочет: Тартара берет проекцию только в плоскости xy! Для наших целей этого недостаточно, поскольку теряется необходимая z-компонента вектора v.

Диего выполняет освещение так же, как и мы: через встроенный в OpenGL механизм расчета. Поскольку мы не можем позволить себе комбинаторный метод, предложенный Голдом (наша программа должна работать на любом оборудовании, а не только на чипах TNT!), хранить коэффициент диффузного рассеяния в альфа-канале нельзя. Вспомним, что нас в любом случае будет 3 прохода немультитекстурного / 2 прохода мультитекстурного наложения. Почему бы не применить механизм освещения из OpenGL в последнем проходе, чтобы разобраться с окружающим освещением и цветами? Правда, это возможно (и красиво выглядит) только потому, что у нас нет геометрически сложных сцен — имейте это в виду. Если, не дай Бог, возникнет нужда просчитать рельеф нескольких тысяч треугольников, придется вам изобретать что-то новое.

Далее, Диего использует мультитекстурирование, которое как мы увидим впоследствии, далеко не так просто, как может показаться для данного случая.

Вернемся к нашей реализации. Она практически совпадает с рассмотренным алгоритмом, за исключением шага проецирования, где мы используем другой подход:
Мы применяем СИСТЕМУ КООРДИНАТ ОБЪЕКТА, то есть не используем в вычислениях матрицу вида модели (modelview). Из-за этого возникает неприятный побочный эффект: если куб приходится вращать, его система координат остается неизменной, в то время как мировая система (она же система координат наблюдателя) поворачивается. Однако положение источника света не должно изменяться, то есть мировые координаты источника должны оставаться постоянными. Чтобы скомпенсировать поворот, применим широко распространенный трюк: вместо пересчета каждой вершины куба в пространство мировых координат для последующего расчета рельефа, повернем источник в том же пространстве на величину, обратную повороту куба (используем инвертированную матрицу вида модели куба). Это делается очень быстро, поскольку раз мы знаем, как матрица вида модели была создана, то можем оперативно ее инвертировать. Позже мы вернемся к этому вопросу.
Вычислим текущую вершину "c" нашей поверхности (просто взяв ее из массива data).
Затем вычислим нормаль n длиной 1 (в военное время длина нормали может достигать четырех! :) — прим. перев.)
Обычно вектор нормали известен для каждой грани куба. Это важно, так как, получая нормализованные вектора, мы уменьшаем время расчета. Определим вектор освещения v от c к источнику света l.
Осталось рассчитать матрицу Mn для получения ортонормальной проекции. Получится f.
Вычислим сдвиг текстурных координат, умножив каждый из параметров s и t на v и MAX_EMBOSS: ds = s*v*MAX_EMBOSS, dt=t*v*MAX_EMBOSS. Обратите внимание: s, t и v — вектора, а MAX_EMBOSS — нет.
Во втором проходе добавим сдвиг к текстурным координатам.

Что в модели хорошего:
Она быстрая (вычисляется только один квадратный корень и пара умножений на вершину).
Она здорово выглядит.
Работает с любыми поверхностями, не только с плоскостями.
Работает на всех акселераторах.
glBegin/glEnd-совместима: не требует "запрещенных" GL-команд.

Какие недостатки:
Модель не вполне физически корректна.
Остаются мелкие артефакты.

На этом рисунке показано, где расположены вектора. t и s можно получить путем вычитания смежных векторов, но нужно следить за тем, чтобы они были верно направлены и нормализованы. Синей точкой помечена вершина, к которой проведена привязка texCoord2f(0.0f,0.0f).

Конец теории (Алгоритмы наложения микрорельефа методом тиснения)

Давайте сначала рассмотрим формирование сдвига текстурных координат. Функция называется SetUpBumps(), потому что именно этим она и занимается:

```
// Выполнение сдвига текстуры
// n : нормаль к поверхности. Должна иметь длину 1
// c : текущая вершина на поверхности (координаты местоположения)
// l : положение источника света
// s : направление s-координаты текстуры в пространстве объекта
```

```
//     (должна быть нормализована!)
// t : направление t-координаты текстуры в пространстве объекта
//     (должна быть нормализована!)

void SetUpBumps(GLfloat *n, GLfloat *c, GLfloat *l, GLfloat *s, GLfloat *t) {
  GLfloat v[3];               // Вектор от текущей точки к свету
  GLfloat lenQ;               // Используется для нормализации
  // Вычислим и нормализуем v
  v[0]=l[0]-c[0];
  v[1]=l[1]-c[1];
  v[2]=l[2]-c[2];
  lenQ=(GLfloat) sqrt(v[0]*v[0]+v[1]*v[1]+v[2]*v[2]);
  v[0]/=lenQ;
  v[1]/=lenQ;
  v[2]/=lenQ;
  // Получим величины проекции v вдоль каждой оси системы текстурных координат
  c[0]=(s[0]*v[0]+s[1]*v[1]+s[2]*v[2])*MAX_EMBOSS;
  c[1]=(t[0]*v[0]+t[1]*v[1]+t[2]*v[2])*MAX_EMBOSS;
}
```

Не так уж все и сложно, а? Но знание теории необходимо для понимания и управления эффектом (я даже сам разобрался в ЭТОМ, пока писал урок).

Мне нравится, чтобы во время работы презентационных программ по экрану летал логотип. У нас их целых два. Вызов doLogo() сбрасывает матрицу GL_MODELVIEW, поэтому он будет выполнен на последней стадии визуализации.

Функция отображает два логотипа: OpenGL и логотип мультитекстурного режима, если он включен. Логотипы содержат альфа-канал и, соответственно, полупрозрачны. Для реализации этого эффекта использованы GL_SRC_ALPHA и GL_ONE_MINUS_SRC_ALPHA, как рекомендовано OpenGL-документацией. Логотипы планарны, поэтому проводить z-сортировку нет необходимости. Числа, взятые для их высот, подобраны эмпирически (a.k.a. методом научного тыка) так, чтобы все помещалось в края экрана. Нужно включить смешивание и выключить освещение, чтобы избежать неприятных эффектов, а чтобы гарантировать размещение логотипов поверх сцены, достаточно сбросить матрицу GL_MODELVIEW и установить функцию глубины в GL_ALWAYS.

```
void doLogo(void) {
  // ВЫЗЫВАТЬ В ПОСЛЕДНЮЮ ОЧЕРЕДЬ!!! отображает два логотипа
  glDepthFunc(GL_ALWAYS);
  glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);
  glEnable(GL_BLEND);
  glDisable(GL_LIGHTING);
  glLoadIdentity();
  glBindTexture(GL_TEXTURE_2D,glLogo);
  glBegin(GL_QUADS);
    glTexCoord2f(0.0f,0.0f);  glVertex3f(0.23f, -0.4f,-1.0f);
    glTexCoord2f(1.0f,0.0f);  glVertex3f(0.53f, -0.4f,-1.0f);
    glTexCoord2f(1.0f,1.0f);  glVertex3f(0.53f, -0.25f,-1.0f);
    glTexCoord2f(0.0f,1.0f);  glVertex3f(0.23f, -0.25f,-1.0f);
  glEnd();

  if (useMultitexture) {
    glBindTexture(GL_TEXTURE_2D,multiLogo);
    glBegin(GL_QUADS);
      glTexCoord2f(0.0f,0.0f);  glVertex3f(-0.53f, -0.25f,-1.0f);
      glTexCoord2f(1.0f,0.0f);  glVertex3f(-0.33f, -0.25f,-1.0f);
      glTexCoord2f(1.0f,1.0f);  glVertex3f(-0.33f, -0.15f,-1.0f);
      glTexCoord2f(0.0f,1.0f);  glVertex3f(-0.53f, -0.15f,-1.0f);
    glEnd();
  }
}
```

Здесь начинается функция, реализующая наложение микрорельефа без использования мультитекстурирования. Это трехпроходная реализация. На первом шаге GL_MODELVIEW инвертируется путем применения к тождественной ей матрице всех шагов, применяемых позже к GL_MODELVIEW, но в обратном порядке и с инвертированными величинами. Такая матрица преобразования, будучи применена к объекту, "отменяет" воздействие GL_MODELVIEW. Мы получим ее от OpenGL вызовом glGetFloatv(). Напоминаю, что матрица должна быть массивом из 16 величин и что она транспонирована!

Кстати: если вы не уверены, в каком порядке была создана матрица вида модели, подумайте о возможности использования мировой системы координат, потому что классическая инверсия произвольной матрицы — вычислительно очень дорогостоящая операция. Впрочем, при обработке значительного числа вершин инверсия матрицы вида модели может быть более приемлемым выходом и, возможно, будет выполняться быстрее, чем расчет мировых координат для каждой вершины.

```
bool doMesh1TexelUnits(void) {
  GLfloat c[4]={0.0f,0.0f,0.0f,1.0f};       // Текущая вершина
  GLfloat n[4]={0.0f,0.0f,0.0f,1.0f};       // Нормаль к текущей поверхности
  GLfloat s[4]={0.0f,0.0f,0.0f,1.0f};       // s-вектор, нормализованный
  GLfloat t[4]={0.0f,0.0f,0.0f,1.0f};       // t-вектор, нормализованный
  GLfloat l[4];                             // Содержит координаты источника освещения,
                                    // который будет переведен в мировые координаты
  GLfloat Minv[16];                         // Инвертированная матрица вида модели
  int i;

  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Очистка экрана и буфера глубины

  // Инвертируем матрицу вида модели. Заменяет один Push/Pop и один glLoadIdentity();
  // Выполняется проведением всех преобразований в обратную сторону в обратном порядке
  glLoadIdentity();
  glRotatef(-yrot,0.0f,1.0f,0.0f);
  glRotatef(-xrot,1.0f,0.0f,0.0f);
  glTranslatef(0.0f,0.0f,-z);
  glGetFloatv(GL_MODELVIEW_MATRIX,Minv);
  glLoadIdentity();
  glTranslatef(0.0f,0.0f,z);
  glRotatef(xrot,1.0f,0.0f,0.0f);
  glRotatef(yrot,0.0f,1.0f,0.0f);

  // Преобразование положения источника в систему координат объекта:

  l[0]=LightPosition[0];
  l[1]=LightPosition[1];
  l[2]=LightPosition[2];
  l[3]=1.0f;                // Гомогенные координаты
  VMatMult(Minv,l);
```

На первом шаге надо:
Использовать текстуру рельефа
Отключить смешивание
Отключить освещение
Использовать несмещенные текстурные координаты
Построить геометрию
Будет визуализирован куб, содержащий только текстуру рельефа.

```
  glBindTexture(GL_TEXTURE_2D, bump[filter]);
  glDisable(GL_BLEND);
  glDisable(GL_LIGHTING);
  doCube();
```

На втором шаге надо:
Использовать инвертированную текстуру рельефа
Включить смешивание GL_ONE, GL_ONE
Освещение остается отключенным

Использовать смещенные координаты текстуры (это значит, что перед просчетом каждой грани куба придется вызывать SetUpBumps()).
Построить геометрию

Здесь будет визуализирован куб с корректно наложенной картой высот, но без цветов.

Можно было бы уменьшить время вычисления, повернув вектор освещения в обратную сторону. Однако этот способ не работает корректно, так что мы сделаем все просто: повернем каждую нормаль и центр так же, как делаем это с геометрией.

```
glBindTexture(GL_TEXTURE_2D,invbump[filter]);
glBlendFunc(GL_ONE,GL_ONE);
glDepthFunc(GL_LEQUAL);
glEnable(GL_BLEND);
glBegin(GL_QUADS);
  // Передняя грань
  n[0]=0.0f;
  n[1]=0.0f;
  n[2]=1.0f;
  s[0]=1.0f;
  s[1]=0.0f;
  s[2]=0.0f;
  t[0]=0.0f;
  t[1]=1.0f;
  t[2]=0.0f;
  for (i=0; i<4; i++) {
    c[0]=data[5*i+2];
    c[1]=data[5*i+3];
    c[2]=data[5*i+4];
    SetUpBumps(n,c,l,s,t);
    glTexCoord2f(data[5*i]+c[0], data[5*i+1]+c[1]);
    glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
  }
  // Задняя грань
  n[0]=0.0f;
  n[1]=0.0f;
  n[2]=-1.0f;
  s[0]=-1.0f;
  s[1]=0.0f;
  s[2]=0.0f;
  t[0]=0.0f;
  t[1]=1.0f;
  t[2]=0.0f;
  for (i=4; i<8; i++) {
    c[0]=data[5*i+2];
    c[1]=data[5*i+3];
    c[2]=data[5*i+4];
    SetUpBumps(n,c,l,s,t);
    glTexCoord2f(data[5*i]+c[0], data[5*i+1]+c[1]);
    glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
  }
  // Верхняя грань
  n[0]=0.0f;
  n[1]=1.0f;
  n[2]=0.0f;
  s[0]=1.0f;
  s[1]=0.0f;
  s[2]=0.0f;
  t[0]=0.0f;
  t[1]=0.0f;
  t[2]=-1.0f;
```

```
  for (i=8; i<12; i++) {
    c[0]=data[5*i+2];
    c[1]=data[5*i+3];
    c[2]=data[5*i+4];
    SetUpBumps(n,c,l,s,t);
    glTexCoord2f(data[5*i]+c[0], data[5*i+1]+c[1]);
    glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
  }
 // Нижняя грань
  n[0]=0.0f;
  n[1]=-1.0f;
  n[2]=0.0f;
  s[0]=-1.0f;
  s[1]=0.0f;
  s[2]=0.0f;
  t[0]=0.0f;
  t[1]=0.0f;
  t[2]=-1.0f;
  for (i=12; i<16; i++) {
    c[0]=data[5*i+2];
    c[1]=data[5*i+3];
    c[2]=data[5*i+4];
    SetUpBumps(n,c,l,s,t);
    glTexCoord2f(data[5*i]+c[0], data[5*i+1]+c[1]);
    glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
  }
  // Правая грань
  n[0]=1.0f;
  n[1]=0.0f;
  n[2]=0.0f;
  s[0]=0.0f;
  s[1]=0.0f;
  s[2]=-1.0f;
  t[0]=0.0f;
  t[1]=1.0f;
  t[2]=0.0f;
  for (i=16; i<20; i++) {
    c[0]=data[5*i+2];
    c[1]=data[5*i+3];
    c[2]=data[5*i+4];
    SetUpBumps(n,c,l,s,t);
    glTexCoord2f(data[5*i]+c[0], data[5*i+1]+c[1]);
    glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
  }
  // Левая грань
  n[0]=-1.0f;
  n[1]=0.0f;
  n[2]=0.0f;
  s[0]=0.0f;
  s[1]=0.0f;
  s[2]=1.0f;
  t[0]=0.0f;
  t[1]=1.0f;
  t[2]=0.0f;
  for (i=20; i<24; i++) {
    c[0]=data[5*i+2];
    c[1]=data[5*i+3];
    c[2]=data[5*i+4];
    SetUpBumps(n,c,l,s,t);
    glTexCoord2f(data[5*i]+c[0], data[5*i+1]+c[1]);
   glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
  }
glEnd();
```

На третьем шаге надо:
Использовать основную (цветную) текстуру
Включить смешивание GL_DST_COLOR, GL_SRC_COLOR
Уравнения смешивания фактически получает множитель 2: (Cdst*Csrc)+(Csrc*Cdst)=2(Csrc*Cdst)!
Включить освещение для расчета фонового и диффузного освещения
Сбросить матрицу GL_TEXTURE с целью вернуться к "нормальным" текстурным координатам
Построить геометрию

Это заключительная стадия расчета, с учетом освещения. Чтобы корректно переключаться между мультитекстурным и однотекстурным режимами, надо сначала выставить среду текстурирования в "нормальный" режим GL_MODULATE. Если захочется отказаться от наложения цветной текстуры, достаточно ограничиться первыми двумя проходами и пропустить третий.

```
 if (!emboss) {
   glTexEnvf (GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
   glBindTexture(GL_TEXTURE_2D,texture[filter]);
   glBlendFunc(GL_DST_COLOR,GL_SRC_COLOR);
   glEnable(GL_LIGHTING);
   doCube();
  }
```

На финальном шаге надо:
Обновить геометрию (особенно вращение)
Отобразить логотипы

```
 xrot+=xspeed;
  yrot+=yspeed;
  if (xrot>360.0f) xrot-=360.0f;
  if (xrot<0.0f) xrot+=360.0f;
  if (yrot>360.0f) yrot-=360.0f;
  if (yrot<0.0f) yrot+=360.0f;

  /* ПОСЛЕДНИЙ ПРОХОД: Даешь логотипы! */
  doLogo();
  return true;                // Продолжаем
}
```

Следующая новая функция выполнит всю задачу за 2 прохода с использованием мультитекстурирования. Будут задействованы два текстурных блока, большее их количество резко усложнит уравнения смешивания. Лучше уж заниматься оптимизацией под TNT. Обратите внимание, практически единственное отличие от doMesh1TexelUnits() в том, что для каждой вершины отсылается не один, а два набора текстурных координат.

```
bool doMesh2TexelUnits(void) {
  GLfloat c[4]={0.0f,0.0f,0.0f,1.0f};     // Здесь храним текущую вершину
  GLfloat n[4]={0.0f,0.0f,0.0f,1.0f};     // Вектор нормали к текущей поверхности
  GLfloat s[4]={0.0f,0.0f,0.0f,1.0f};     // s-вектор, нормализованный
  GLfloat t[4]={0.0f,0.0f,0.0f,1.0f};     // t-вектор, нормализованный
  GLfloat l[4];                           // Хранит координаты источника света,
                                 // для перевода в пространство координат объекта
  GLfloat Minv[16];                       // Инвертированная матрица вида модели
  int i;

  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Очистим экран и буфер глубины

  // Инвертируем матрицу вида модели. Заменяет один Push/Pop и один glLoadIdentity();
  // Выполняется проведением всех преобразований в обратную сторону в обратном порядке
  glLoadIdentity();
  glRotatef(-yrot,0.0f,1.0f,0.0f);
  glRotatef(-xrot,1.0f,0.0f,0.0f);
  glTranslatef(0.0f,0.0f,-z);
 glGetFloatv(GL_MODELVIEW_MATRIX,Minv);
```

```
  glLoadIdentity();
  glTranslatef(0.0f,0.0f,z);
  glRotatef(xrot,1.0f,0.0f,0.0f);
  glRotatef(yrot,0.0f,1.0f,0.0f);

  // Преобразуем координаты источника света в систему координат объекта
  l[0]=LightPosition[0];
  l[1]=LightPosition[1];
  l[2]=LightPosition[2];
  l[3]=1.0f;              // Гомогенные координаты
  VMatMult(Minv,l);
```

На первом шаге надо:
Отменить смешивание
Отменить освещение

Установить текстурный комбайн 0 на
Использование текстуры рельефа
Использование несмещенных координат текстуры
Выполнение операции GL_REPLACE, то есть простое отображение текстуры

Установить текстурный комбайн 1 на
Использование сдвинутых текстурных координат
Выполнение операции GL_ADD, эквивалента однотекстурного ONE-ONE-смешивания.
Будет рассчитан куб с наложенной картой эрозии поверхности.

```
  // ТЕКСТУРНЫЙ БЛОК #0
  glActiveTextureARB(GL_TEXTURE0_ARB);
  glEnable(GL_TEXTURE_2D);
  glBindTexture(GL_TEXTURE_2D, bump[filter]);
  glTexEnvf (GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE_EXT);
  glTexEnvf (GL_TEXTURE_ENV, GL_COMBINE_RGB_EXT, GL_REPLACE);

  // ТЕКСТУРНЫЙ БЛОК #1
  glActiveTextureARB(GL_TEXTURE1_ARB);
  glEnable(GL_TEXTURE_2D);
  glBindTexture(GL_TEXTURE_2D, invbump[filter]);
  glTexEnvf (GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE_EXT);
  glTexEnvf (GL_TEXTURE_ENV, GL_COMBINE_RGB_EXT, GL_ADD);

  // Общие флаги
  glDisable(GL_BLEND);
  glDisable(GL_LIGHTING);
```

Теперь визуализируем грани одну за одной, как это было сделано в doMesh1TexelUnits(). Единственное отличие — вместо glTexCoord2f() используется glMultiTexCoor2fARB(). Обратите внимание, надо прямо указывать, какой текстурный блок вы имеете в виду. Параметр должен иметь вид GL_TEXTUREi_ARB, где i лежит в диапазоне [0..31]. (Это что же за карточка с 32 текстурными блоками? И зачем она?)

```
  glBegin(GL_QUADS);
    // Передняя грань
    n[0]=0.0f;
    n[1]=0.0f;
    n[2]=1.0f;
    s[0]=1.0f;
    s[1]=0.0f;
    s[2]=0.0f;
    t[0]=0.0f;
    t[1]=1.0f;
    t[2]=0.0f;
    for (i=0; i<4; i++) {
      c[0]=data[5*i+2];
      c[1]=data[5*i+3];
```

```
  c[2]=data[5*i+4];
  SetUpBumps(n,c,l,s,t);
  glMultiTexCoord2fARB(GL_TEXTURE0_ARB,data[5*i], data[5*i+1]);
  glMultiTexCoord2fARB(GL_TEXTURE1_ARB,data[5*i]+c[0], data[5*i+1]+c[1]);
  glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
 }

// Задняя грань
 n[0]=0.0f;
 n[1]=0.0f;
 n[2]=-1.0f;
 s[0]=-1.0f;
 s[1]=0.0f;
 s[2]=0.0f;
 t[0]=0.0f;
 t[1]=1.0f;
 t[2]=0.0f;
 for (i=4; i<8; i++) {
  c[0]=data[5*i+2];
  c[1]=data[5*i+3];
  c[2]=data[5*i+4];
  SetUpBumps(n,c,l,s,t);
  glMultiTexCoord2fARB(GL_TEXTURE0_ARB,data[5*i], data[5*i+1]);
  glMultiTexCoord2fARB(GL_TEXTURE1_ARB,data[5*i]+c[0], data[5*i+1]+c[1]);
  glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
 }

 // Верхняя грань
 n[0]=0.0f;
 n[1]=1.0f;
 n[2]=0.0f;
 s[0]=1.0f;
 s[1]=0.0f;
 s[2]=0.0f;
 t[0]=0.0f;
 t[1]=0.0f;
 t[2]=-1.0f;
 for (i=8; i<12; i++) {
  c[0]=data[5*i+2];
  c[1]=data[5*i+3];
  c[2]=data[5*i+4];
  SetUpBumps(n,c,l,s,t);
  glMultiTexCoord2fARB(GL_TEXTURE0_ARB,data[5*i], data[5*i+1]);
  glMultiTexCoord2fARB(GL_TEXTURE1_ARB,data[5*i]+c[0], data[5*i+1]+c[1]);
  glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
 }

 // Нижняя грань
 n[0]=0.0f;
 n[1]=-1.0f;
 n[2]=0.0f;
 s[0]=-1.0f;
 s[1]=0.0f;
 s[2]=0.0f;
 t[0]=0.0f;
 t[1]=0.0f;
 t[2]=-1.0f;
 for (i=12; i<16; i++) {
  c[0]=data[5*i+2];
  c[1]=data[5*i+3];
  c[2]=data[5*i+4];
  SetUpBumps(n,c,l,s,t);
  glMultiTexCoord2fARB(GL_TEXTURE0_ARB,data[5*i], data[5*i+1]);
```

```
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB,data[5*i]+c[0], data[5*i+1]+c[1]);
    glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
   }

   // Правая грань
   n[0]=1.0f;
   n[1]=0.0f;
   n[2]=0.0f;
   s[0]=0.0f;
   s[1]=0.0f;
   s[2]=-1.0f;
   t[0]=0.0f;
   t[1]=1.0f;
   t[2]=0.0f;
   for (i=16; i<20; i++) {
    c[0]=data[5*i+2];
    c[1]=data[5*i+3];
    c[2]=data[5*i+4];
    SetUpBumps(n,c,l,s,t);
    glMultiTexCoord2fARB(GL_TEXTURE0_ARB,data[5*i], data[5*i+1]);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB,data[5*i]+c[0], data[5*i+1]+c[1]);
    glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
   }

   // Левая грань
   n[0]=-1.0f;
   n[1]=0.0f;
   n[2]=0.0f;
   s[0]=0.0f;
   s[1]=0.0f;
   s[2]=1.0f;
   t[0]=0.0f;
   t[1]=1.0f;
   t[2]=0.0f;
   for (i=20; i<24; i++) {
    c[0]=data[5*i+2];
    c[1]=data[5*i+3];
    c[2]=data[5*i+4];
    SetUpBumps(n,c,l,s,t);
    glMultiTexCoord2fARB(GL_TEXTURE0_ARB,data[5*i], data[5*i+1]);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB,data[5*i]+c[0], data[5*i+1]+c[1]);
    glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
   }
  glEnd();
```

На втором шаге надо:
Использовать основную текстуру
Включить освещение
Отменить смещение текстурных координат => сброс матрицы GL_TEXTURE
Текстурную среду вернуть в состояние GL_MODULATE, чтобы заработало освещение OpenGL (иначе не получится!)
Здесь уже будет полностью готов куб.

```
  glActiveTextureARB(GL_TEXTURE1_ARB);
  glDisable(GL_TEXTURE_2D);
  glActiveTextureARB(GL_TEXTURE0_ARB);
  if (!emboss) {
   glTexEnvf (GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
   glBindTexture(GL_TEXTURE_2D,texture[filter]);
   glBlendFunc(GL_DST_COLOR,GL_SRC_COLOR);
   glEnable(GL_BLEND);
   glEnable(GL_LIGHTING);
  doCube();
  }
```

На последнем шаге надо
Обновить геометрию (особенно вращение)
Отобразить логотипы

```
  xrot+=xspeed;
  yrot+=yspeed;
  if (xrot>360.0f) xrot-=360.0f;
  if (xrot<0.0f) xrot+=360.0f;
  if (yrot>360.0f) yrot-=360.0f;
  if (yrot<0.0f) yrot+=360.0f;

  /* ПОСЛЕДНИЙ ПРОХОД: да будут логотипы! */
  doLogo();
  return true;                // Продолжим
}
```

И, для сравнения, функция, рисующая куб без рельефа — почувствуйте разницу!

```
bool doMeshNoBumps(void) {
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);      // Очистить экран и буфер глубины
  glLoadIdentity();              // Сбросить вид
  glTranslatef(0.0f,0.0f,z);
  glRotatef(xrot,1.0f,0.0f,0.0f);
  glRotatef(yrot,0.0f,1.0f,0.0f);

  if (useMultitexture) {
    glActiveTextureARB(GL_TEXTURE1_ARB);
    glDisable(GL_TEXTURE_2D);
    glActiveTextureARB(GL_TEXTURE0_ARB);
  }

  glDisable(GL_BLEND);
  glBindTexture(GL_TEXTURE_2D,texture[filter]);
  glBlendFunc(GL_DST_COLOR,GL_SRC_COLOR);
  glEnable(GL_LIGHTING);
  doCube();

  xrot+=xspeed;
  yrot+=yspeed;
  if (xrot>360.0f) xrot-=360.0f;
  if (xrot<0.0f) xrot+=360.0f;
  if (yrot>360.0f) yrot-=360.0f;
  if (yrot<0.0f) yrot+=360.0f;

  /* ПОСЛЕДНИЙ ПРОХОД: логотипы */
  doLogo();
  return true;                // Продолжим
}
```

Все, что должна делать функция drawGLScene() — определить, какой из вариантов doMesh вызвать:

```
bool DrawGLScene(GLvoid)              // Здесь все рисуется
{
  if (bumps) {
    if (useMultitexture && maxTexelUnits>1)
      return doMesh2TexelUnits();
    else return doMesh1TexelUnits();
  }
  else return doMeshNoBumps();
}
```

Убиваем GLWindow. Функция не изменялась, а потому не приведена:

```
GLvoid KillGLWindow(GLvoid)          // Уничтожим окно корректно
>…<
```

Функция создает GLWindow; не изменена, поэтому пропущена:

```
BOOL CreateGLWindow(char* title, int width, int height, int bits, bool fullscreenflag)
>…<
```

Основной цикл обработки сообщений; не изменен, поэтому пропущен:

```
LRESULT CALLBACK WndProc(  HWND hWnd, // Указатель окна
      UINT uMsg,                 // Сообщение для этого окна
      WPARAM wParam,             // Дополнительная информация о сообщении
      LPARAM lParam)             // Дополнительная информация о сообщении
>…<
```

Основная функция окна. Здесь добавлена обработка различных дополнительных кнопок:
E: Переключает режимы чистого рельефа / рельефа с текстурой
M: Включает/отключает мультитекстурирование
B: Включает/отключает наложение микрорельефа. Опция является взаимоисключающей с настройками, управляемыми кнопкой E
F: Переключает способы фильтрации. Вы увидите, насколько режим GL_NEAREST не создан для рельефного текстурирования
Клавиши управления курсором: Вращение куба

```
int WINAPI WinMain(  HINSTANCE hInstance, // Экземпляр приложения
    HINSTANCE hPrevInstance,          // Предыдущий экземпляр
    LPSTR lpCmdLine,                  // Параметры командной строки
    int nCmdShow)                     // Показать состояние окна
{
 >…<

     if (keys['E'])
      {   keys['E']=false;
        emboss=!emboss;
      }

      if (keys['M'])
      {  keys['M']=false;
        useMultitexture=((!useMultitexture) && multitextureSupported);
      }

      if (keys['B'])
      {
        keys['B']=false;
        bumps=!bumps;
      }

      if (keys['F'])
      {
        keys['F']=false;
        filter++;
        filter%=3;
      }

      if (keys[VK_PRIOR])
      {
        z-=0.02f;
      }


     if (keys[VK_NEXT])
      {
```

```
        z+=0.02f;
       }

       if (keys[VK_UP])
       {
        xspeed-=0.01f;
       }

       if (keys[VK_DOWN])
       {
        xspeed+=0.01f;
       }

      if (keys[VK_RIGHT])
       {
        yspeed+=0.01f;
       }

      if (keys[VK_LEFT])
       {
        yspeed-=0.01f;
       }
     }
    }
  }
  // Выключаемся
  KillGLWindow();      // Убить окно
  return (msg.wParam); // Выйти из программы
}
```

Еще несколько слов о генерации текстур и наложении рельефа на объекты, прежде чем вы начнете ваять великие игры и поражаться, почему они идут так медленно и так ужасно выглядят:
Не стоит использовать текстуры рельефа размером 256x256, как в этом уроке. Все начинает сильно тормозить, поэтому такие размеры подходят только для демонстрационных целей (например, в уроках).
Куб, имеющий рельеф — редкая вещь. Повернутый рельефный куб — и того реже. Причина в том, что угол зрения сильно влияет на качество изображения, и чем он больше, тем хуже результат. Практически все многопроходные алгоритмы подвержены этому недостатку. Чтобы не применять текстуры высокого разрешения, увеличьте минимальный угол зрения до приемлемой величины или уменьшите диапазон углов и проводите предварительную фильтрацию текстур так, чтобы удовлетворить этому диапазону.
Сначала создайте основную текстуру. Рельеф можно сделать позже в любом редакторе, переведя картинку в шкалу серого.
Рельеф должен быть "острее" и контрастнее основной текстуры. Это можно сделать, применив фильтр "резкость" (sharpen) к основной текстуре. Поначалу может смотреться странно, но поверьте: чтобы получить первоклассный оптический эффект, нужно КАПИТАЛЬНО "заострить" текстуру.
Текстура рельефа должна быть отцентрована по 50% серому, (RGB=127,127,127), поскольку это эквивалентно отсутствию рельефа. Более яркие значения соответствуют выпуклостям, а менее яркие — провалам. Результат можно оценить, просмотрев текстуру в режиме гистограммы в каком-нибудь подходящем редакторе.
Текстура рельефа может быть в четверть размера основной текстуры, и это не приведет к катастрофическим последствиям, хотя разница, конечно, будет заметна.

Теперь у вас должно быть некоторое представление о вещах, обсужденных в этом уроке. Надеюсь, вам понравилось. Вопросы, пожелания, предложения, просьбы, жалобы? Почтуйте, потому что веб-странички у меня пока еще нет.

Это мой основной проект; надеюсь, вскоре последует продолжение.

Моя признательность:
Michael I. Gold за документацию по наложению микрорельефа
Diego Tartara за код этого примера
NVidia за размещение отличных примеров в Сети
И, конечно, NeHe за неоценимую помощь в освоении OpenGL.

Урок 23. Квадратирование со сферическим наложением в OpenGL

Sphere Mapping Quadrics In OpenGL

Сферическое наложение текстур окружения дает возможность быстро создавать отражения в металлических или зеркальных поверхностях в кадре. Этот метод не столь точен, как кубическая карта окружения, и уж тем более отличается от реальной жизни, зато он гораздо быстрее! За основу возьмем код Урока 18. Кроме того, мы не будем пользоваться текстурами оттуда, взамен них создадим две новые: сферическую карту и фоновое изображение.

Прежде чем мы начнем… "Красная книга" определяет сферическую карту как изображение сцены на металлическом шаре из бесконечно удаленной точки с бесконечным фокусным расстоянием. Конечно, в реальности это недостижимо. Лучший способ создать сферическую карту, не пользуясь линзой "рыбий глаз", какой я обнаружил, это поработать в Фотошопе.

Создание сферической карты в Фотошопе:

Прежде всего, нужно изображение, которое вы намерены использовать для сферического наложения. Откройте изображение в Фотошопе и выделите его целиком. Скопируйте, создайте новый файл (при создании его размеры будут предложены Фотошопом, они будут совпадать с размерами скопированного изображения) и вставьте туда содержимое буфера. Смысл операции состоит в том, чтобы получить возможность использовать все фильтры Фотошопа. Другой способ добиться этого — изменить текущий режим изображения на RGB через соответствующий пункт меню.

Затем изображение нужно промасштабировать так, чтобы его размеры были степенью двойки. Как вы помните, чтобы изображение можно было использовать в качестве текстуры, оно должно иметь размер 128x128, 256x256 точек и так далее. В меню "Изображение" выберите "Размер изображения", снимите галочку напротив опции сохранения пропорций и измените размеры так, чтобы подогнать их к размерам текстуры. Если исходное изображение, скажем, 100x90 точек, предпочтительно сделать его 128x128, а не 64x64 точки, чтобы максимально сохранить детали.

Наконец, из меню "Фильтры" надо выбрать "Искажения" и в них "Сферизацию". После применения этого фильтра центр изображения станет выпуклым, как шарик. В нормальной сферической карте изображение по мере приближения к краю должно темнеть и уходить в черноту, но сейчас это неважно. Сохраните полученный результат в формате BMP, и можно приступать к кодированию!

На этот раз мы не станем вводить никаких глобальных переменных, только модифицируем массив текстур так, чтобы он мог хранить их 6 штук.

```
GLuint  texture[6];               // Хранилище для 6 текстур ( ИЗМЕНЕНО )
```

Следующее, что я сделал — модифицировал функцию LoadGLTextures() так, чтобы она могла загружать 2 картинки и создавать 3 фильтра (похоже на то, как это было в уроках по обычному текстурированию). Там выполнялись два цикла, и в каждом создавалось три текстуры с использованием разных режимов фильтрации. Почти весь этот код переписан или изменен.

```
int LoadGLTextures()                // Загрузить картинки и создать текстуры
{
 int Status=FALSE;                  // Индикатор статуса
 AUX_RGBImageRec *TextureImage[2]; // Выделим место для хранения текстур
 memset(TextureImage,0,sizeof(void *)*2);         // Сбросим эти указатели
 // Загрузим картинку, проверим на ошибки, если картинка не найдена - выйдем
 if ((TextureImage[0]=LoadBMP("Data/BG.bmp")) &&  // Фоновая текстура
   (TextureImage[1]=LoadBMP("Data/Reflect.bmp"))) // Текстура отражения
                                    // (сферическая карта)
 {
  Status=TRUE;                      // Установить индикатор в TRUE
  glGenTextures(6, &texture[0]);  // Создадим три текстуры
  for (int loop=0; loop<=1; loop++)
  {
   // Создадим текстуры без фильтрации
   glBindTexture(GL_TEXTURE_2D, texture[loop]); // Текстуры 0 и 1
   glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST);
   glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST);
   glTexImage2D(GL_TEXTURE_2D, 0, 3, TextureImage[loop]->sizeX,
```

```
            TextureImage[loop]->sizeY, 0, GL_RGB, GL_UNSIGNED_BYTE,
            TextureImage[loop]->data);

    // Создадим линейно фильтрованные текстуры
    glBindTexture(GL_TEXTURE_2D, texture[loop+2]);    // Текстуры 2, 3 и 4
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
    glTexImage2D(GL_TEXTURE_2D, 0, 3, TextureImage[loop]->sizeX,
            TextureImage[loop]->sizeY, 0, GL_RGB, GL_UNSIGNED_BYTE,
            TextureImage[loop]->data);

    // мип-мап текстуры
    glBindTexture(GL_TEXTURE_2D, texture[loop+4]);    // Текстуры 4 и 5
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR_MIPMAP_NEAREST);
    gluBuild2DMipmaps(GL_TEXTURE_2D, 3, TextureImage[loop]->sizeX,
                TextureImage[loop]->sizeY, GL_RGB, GL_UNSIGNED_BYTE,
                TextureImage[loop]->data);
  }

  for (loop=0; loop<=1; loop++)
  {
      if (TextureImage[loop])                 // Если текстура существует
     {
         if (TextureImage[loop]->data)        // Если существует изображение текстуры
       {
           free(TextureImage[loop]->data);  // Освободим память изображения текстуры
       }
       free(TextureImage[loop]);               // Освободим память
                                    // структуры изображения
    }
   }
  }
  return Status; // Вернем статус
}
```

Теперь слегка изменим код рисования куба. Вместо 1.0 и -1.0 в качестве значений нормали используем 0.5 и -0.5. Так у нас появится возможность увеличивать и уменьшать карту отражений. Если значение нормали велико, то отраженное изображение станет больше и, возможно, на нем будут заметны квадратики. А если нормаль уменьшить до 0.5, изображение тоже уменьшится и качество картинки повысится. Если еще сильнее уменьшить значение нормали, то мы получим нежелательные результаты.

```
GLvoid glDrawCube()
{
  glBegin(GL_QUADS);
  // Передняя грань
  glNormal3f( 0.0f, 0.0f, 0.5f);          ( Изменено )
  glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f,  1.0f);
  glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f,  1.0f);
  glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f,  1.0f,  1.0f);
  glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f,  1.0f,  1.0f);
  // Задняя грань
  glNormal3f( 0.0f, 0.0f,-0.5f);          ( Изменено )
  glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
  glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f,  1.0f, -1.0f);
  glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f,  1.0f, -1.0f);
  glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
  // Верхняя грань
  glNormal3f( 0.0f, 0.5f, 0.0f);          ( Изменено )
  glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f,  1.0f, -1.0f);
  glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f,  1.0f,  1.0f);
  glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f,  1.0f,  1.0f);
  glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f,  1.0f, -1.0f);
```

```
    // Нижняя грань
    glNormal3f( 0.0f,-0.5f, 0.0f);          ( Изменено )
    glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
    glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
    glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f,  1.0f);
    glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f,  1.0f);
    // Правая грань
    glNormal3f( 0.5f, 0.0f, 0.0f);          ( Изменено )
    glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
    glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f,  1.0f, -1.0f);
    glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f,  1.0f,  1.0f);
    glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f,  1.0f);
    // Левая грань
    glNormal3f(-0.5f, 0.0f, 0.0f);          ( Изменено )
    glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
    glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f,  1.0f);
    glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f,  1.0f,  1.0f);
    glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f,  1.0f, -1.0f);
  glEnd();
}
```

Теперь в InitGL будут добавлены два новых вызова, устанавливающих режим генерации текстур для S и T для использования при сферическом текстурировании. Текстурные координаты S, T, R и Q определенным образом соответствуют координатам объекта x, y, z и w. Если вы применяете одномерную текстуру (1D), то будете использовать координату S. Если текстура двумерная, то кроме S применяется и координата T.

Следующий фрагмент кода заставляет OpenGL генерировать координаты S и T, основываясь на формуле сферического наложения. Координаты R и Q обычно игнорируются. Координата Q может быть использована в расширениях продвинутых техник текстурирования, а координата R, возможно, станет полезной, когда в библиотеку OpenGL будет добавлено 3D текстурирование. Сейчас же мы проигнорируем и R, и Q. Координата S идет горизонтально через плоскость нашего полигона, а координата T — вертикально.

```
  // Изменить для S режим генерации текстур на "сферическое наложение" ( Новое )
  glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
  // Изменить для T режим генерации текстур на "сферическое наложение" ( Новое )
  glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
```

Мы практически закончили! Остается настроить визуализацию. Я убрал несколько квадратичных объектов, потому что они плохо подходят для наложения текстур окружения. Сначала надо разрешить генерацию текстур. Затем выбрать текстуру, представляющую отражение, и нарисовать объект. После того, как объекты, для которых планируется сферическое текстурирование, будут отрисованы, генерацию текстур придется запретить, иначе сферически текстурированным окажется вообще все. Наложение текстур мы отключим перед тем, как начнем рисовать задний план (потому что не планируем сферически текстурировать и его). Вы увидите, что команды привязки текстур производят впечатление чрезвычайно сложных. На самом деле все, что мы делаем — это выбираем фильтр, который надо использовать при наложении сферической карты или фонового изображения.

```
int DrawGLScene(GLvoid)          // Здесь происходит все рисование
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Очистим экран и буфер глубины
  glLoadIdentity();              // Сбросим вид
  glTranslatef(0.0f,0.0f,z);
  glEnable(GL_TEXTURE_GEN_S);    // Включим генерацию координат текстуры для S ( НОВОЕ )
  glEnable(GL_TEXTURE_GEN_T);    // Включим генерацию координат текстуры для T ( НОВОЕ )
  // Выберем сферическое текстурирование ( ИЗМЕНЕНО )
  glBindTexture(GL_TEXTURE_2D, texture[filter+(filter+1)]);
  glPushMatrix();
  glRotatef(xrot,1.0f,0.0f,0.0f);
  glRotatef(yrot,0.0f,1.0f,0.0f);
  switch(object)
  {
  case 0:
    glDrawCube();
    break;
```

```
  case 1:
    glTranslatef(0.0f,0.0f,-1.5f);              // Отцентруем цилиндр
    gluCylinder(quadratic,1.0f,1.0f,3.0f,32,32); // Цилиндр радиусом 0.5 и высотой 2
    break;
  case 2:
    // Сфера радиусом 1, состоящая из 16 сегментов по долготе/широте
    gluSphere(quadratic,1.3f,32,32);
    break;
  case 3:
    glTranslatef(0.0f,0.0f,-1.5f);              // Отцентруем конус
    // Конус с радиусом основания 0.5 и высотой 2
    gluCylinder(quadratic,1.0f,0.0f,3.0f,32,32);
    break;
  };
  glPopMatrix();
  glDisable(GL_TEXTURE_GEN_S);       // Отключим генерацию текстурных координат ( НОВОЕ )
  glDisable(GL_TEXTURE_GEN_T);       // Отключим генерацию текстурных координат ( НОВОЕ )
  glBindTexture(GL_TEXTURE_2D, texture[filter*2]); // Выберем фоновую текстуру ( НОВОЕ )
  glPushMatrix();
    glTranslatef(0.0f, 0.0f, -24.0f);
    glBegin(GL_QUADS);
      glNormal3f( 0.0f, 0.0f, 1.0f);
      glTexCoord2f(0.0f, 0.0f); glVertex3f(-13.3f, -10.0f,  10.0f);
      glTexCoord2f(1.0f, 0.0f); glVertex3f( 13.3f, -10.0f,  10.0f);
      glTexCoord2f(1.0f, 1.0f); glVertex3f( 13.3f,  10.0f,  10.0f);
      glTexCoord2f(0.0f, 1.0f); glVertex3f(-13.3f,  10.0f,  10.0f);
    glEnd();

  glPopMatrix();

  xrot+=xspeed;
  yrot+=yspeed;
  return TRUE; // Продолжим
}
```

Последнее, что надо сделать — обновить процедуру обработки нажатия пробела, чтобы отразить изменения, внесенные нами в изображаемые квадратичные объекты (были удалены диски).

```
      if (keys[' '] && !sp)
      {
        sp=TRUE;
        object++;
        if(object>3)
          object=0;
       }
```

Мы закончили! Теперь, пользуясь наложением текстур окружения, вы можете делать разные впечатляющие вещи, например, почти точное отражение содержимого комнаты. Я планировал показать, как делать кубическое наложение, но моя видеокарточка не поддерживает этот режим. Может быть, через месяц или около того я куплю GeForce2 :). Кроме того, описанное текстурирование я изучал самостоятельно (в основном из-за того, что по этому вопросу практически нет информации), так что если в этом уроке есть какие-то неточности, сообщите о них либо мне по почте, либо сразу NeHe.

Спасибо. Удачи!

Урок 24. Лексемы, Расширения, Вырезка и Загрузка TGA
Tokens, Extensions, Scissor Testing And TGA Loading

Этот урок далеко не идеален, но Вы определенно узнаете кое-что новое. Я получил довольно много вопросов о расширениях и о том, как определить какие расширения поддерживаются конкретным типом видеокарты. Этот урок научит Вас определять, какие OpenGL расширения поддерживаются любой из 3D видео карт.

Также я научу Вас прокручивать часть экрана, не влияя при этом на остальную, используя вырезку. Вы также научитесь рисовать ломаные линии (GL_LINE_STRIP - прим. пер.), и, что самое важное, в этом уроке мы не будем использовать ни библиотеку AUX ни растровые изображения. Я покажу Вам, как использовать TGA-изображения в качестве текстур. С TGA изображениями не только просто работать, они поддерживают ALPHA-канал, который позволит Вам в будущих проектах использовать некоторые довольно интересные эффекты.

Первое, что Вы должны отметить в коде ниже - нет больше включения заголовочного файла библиотеки glaux (glaux.h). Важно отметить, что файл glaux.lib также можно не включать в проект. Мы не работаем с растровыми изображениями, так что нет смысла включать эти файлы в наш проект.

Используя glaux, я всегда получал от компилятора одно предупреждение (warning). Без glaux не будет ни предупреждений, ни сообщений об ошибках.

```
#include <windows.h>       // Заголовочный файл для Windows
#include <stdio.h>         // Заголовочный файл для стандартного ввода/вывода
#include <stdarg.h>        // Заголовочный файл для переменного числа параметров
#include <string.h>        // Заголовочный файл для работы с типом String
#include <gl\gl.h>         // Заголовочный файл для библиотеки OpenGL32
#include <gl\glu.h>        // Заголовочный файл для библиотеки GLu32

HDC       hDC=NULL;        // Частный контекст устройства
HGLRC     hRC=NULL;        // Постоянный контекст рендеринга
HWND      hWnd=NULL;       // Содержит дескриптор окна
HINSTANCE hInstance;       // Содержит экземпляр приложения

bool      keys[256];       // Массив для работы с клавиатурой
bool      active=TRUE;     // Флаг активности приложения
bool      fullscreen=TRUE; // Флаг полноэкранного режима
```

Первое, что мы должны сделать - добавить несколько переменных. Переменная scroll будет использоваться для прокрутки части экрана вверх и вниз. Переменная maxtokens будет хранить количество лексем (расширений), поддерживаемых данной видеокартой. base хранит базу списков отображения для шрифта. swidth и sheight будут использоваться для захвата текущих размеров окна. Мы используем эти две переменные позднее в коде для облегчения расчета координат вырезки.

```
int       scroll;          // Используется для прокручивания экрана
int       maxtokens;       // Количество поддерживаемых расширений
int       swidth;          // Ширина вырезки
int       sheight;         // Высота вырезки

Gluint    base;            // База списков отображения для шрифта
```

Создадим структуру для хранения информации о TGA изображении, которое мы загрузим. Первая переменная imageData будет содержать указатель на данные, создающие изображение. bpp содержит количество битов на пиксель (количество битов, необходимых для описания одного пикселя - прим. пер.), используемое в TGA файле (это значение может быть 24 или 32 в зависимости от того, используется ли альфа-канал). Третья переменная width будет хранить ширину TGA изображения. height хранит высоту изображения, и texID будет указывать на текстуру, как только она будет построена. Структуру назовем TextureImage.

В строке, следующей за объявлением структуры, резервируется память для хранения одной текстуры, которую мы будем использовать в нашей программе.

```
typedef   struct                    // Создать структуру
{
Glubyte  *imageData;                // Данные изображения (до 32 бит)
Gluint   bpp;                       // Глубина цвета в битах на пиксель
Gluint   width;                     // Ширина изображения
Gluint   height;                    // Высота изображения
Gluint   texID;                     // texID используется для выбора
                                    // текстуры
} TextureImage;                     // Имя структуры

TextureImage   textures[1];         // Память для хранения
                                    // одной текстуры

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);   // Объявление WndProc
```

Теперь позабавимся! Эта часть кода будет загружать TGA файл, и конвертировать его в текстуру, которая будет использоваться в нашей программе. Следует отметить, что этот код загружает только 24 или 32 битные несжатые TGA файлы. Я хорошо постарался, для того чтобы это код мог работать как с 24, так и с 32 битными файлами. :) Я никогда не говорил, что я гениален. Я хочу отметить, что этот код не весь был написан мною самостоятельно. Много хороших идей я извлек из чтения различных сайтов в сети. Я только собрал хорошие идеи и скомбинировал их в код, который хорошо работает с OpenGL. Ни легко, ни очень сложно!

Мы передаем два параметра в функцию ниже. Первый параметр (*texture) указывает на место в памяти, где можно сохранить текстуру. Второй параметр (*filename) - имя файла, который мы хотим загрузить.

Массив TGAheader[] содержит 12 байт. Эти байты мы будем сравнивать с первыми 12 байтами загружаемого TGA файла, для того чтобы убедиться в том, что это действительно TGA файл, а не файл изображения другого типа.

В TGAcompare[] будут помещены первые 12 байт загружаемого TGA файла. После этого произойдет сравнение байтов TGAcompare[] с байтами TGAheader[], для того чтобы убедиться в полном их соответствии.

header[] будет хранить 6 первых ВАЖНЫХ байт заголовка файла (ширину, высоту и количество битов на пиксель).

Переменная bytesPerPixel будет содержать результат деления количества битов на пиксель на 8, который будет являться уже количеством байтов на пиксель.

imageSize будет хранить количество байтов, которое требуется для создания изображения (ширина * высота * количество байтов на пиксель).

temp - временная переменная, будет использована для обмена байтов дальше в программе.

Последнюю переменную type я использую для выбора подходящих параметров построения текстуры, в зависимости от того, является TGA 24 или 32 битным. Если текстура 24-х битная, то мы должны использовать режим GL_RGB при построении текстуры. Если же текстура 32-х битная, то мы должны будем добавить alpha компоненту, т.е. использовать GL_RGBA (по умолчанию я предполагаю, что изображение 32-х битное, вот почему переменная type установлена в GL_RGBA).

```
bool LoadTGA(TextureImage *texture, char *filename)
    // Загружаем TGA файл в память
{
Glubyte  TGAheader[12]={0,0,2,0,0,0,0,0,0,0,0,0}; // Заголовок несжатого TGA файла
Glubyte  TGAcompare[12]; // Используется для сравнения заголовка TGA файла
Glubyte  header[6];      // Первые 6 полезных байт заголовка
Gluint   bytesPerPixel;  // Количество байтов на пиксель используемое в TGA файле
Gluint   imageSize;      // Количество байтов, необходимое для хранения изображения в памяти
Gluint   temp;           // Временная переменная
Gluint   type=GL_RGBA;   // Установим по умолчанию режим RBGA (32 BPP)
```

В первой строке кода ниже TGA файл открывается на чтение. file - указатель, который мы будем использовать для ссылки на данные в пределах файла. Команда fopen(filename, "rb") открывает файл filename, а параметр "rb" говорит нашей программе, что открыть файл нужно на чтение ([r]eading) как двоичный ([b]inary).

Инструкция if производит несколько действий. Во-первых, проверяется, не пуст ли файл. Если он пуст, то будет возвращено значение NULL, файл будет закрыт командой fclose(file) и функция вернет false.

Если файл не пуст, то мы попытаемся прочитать его первые 12 байтов в TGAcompare. Это сделает код в следующей строке: функция fread прочитает sizeof(TGAcompare) (12 байтов) из файла в TGAcompare. Затем мы проверим: соответствует ли количество байтов, прочитанных из файла, размеру TGAcompare, который должен быть равен 12 байтам. Если мы не смогли прочитать 12 байтов в TGAcompare, то файл будет закрыт, и функция возвратит false.

Если все прошло удачно, то мы сравниваем 12 байтов, которые мы прочитали в TGAcompare, с 12 байтами, которые хранятся в TGAheader.

Наконец, если все прошло великолепно, мы попытаемся прочитать следующие 6 байтов в header (важные байты). Если эти 6 байтов недоступны, файл будет закрыт и функция вернет false.

```
FILE *file = fopen(filename, "rb");         // Открытие TGA файла
if(file==NULL ||                            // Существует ли файл
  fread(TGAcompare,1,sizeof(TGAcompare),file)!=sizeof(TGAcompare) ||
  // Если прочитаны 12 байтов заголовка
  memcmp(TGAheader,TGAcompare,sizeof(TGAheader))!=0 || // Если заголовок правильный
  fread(header,1,sizeof(header),file)!=sizeof(header)) // Если прочитаны следующие 6 байтов
{
    if (file == NULL)                       // Если ошибка
        return false;                       // Возвращаем false
    else
    {
        fclose(file);                       // Если ошибка, закрываем файл
        return false;                       // Возвращаем false
    }
}
```

Если все прошло успешно, то теперь у нас достаточно информации для определения некоторых важных переменных. Первой переменной, которую мы определим, будет width. Мы хотим, чтобы width равнялась ширине TGA изображения. Эту ширину мы найдем, умножив значение, сохраненное в header[1], на 256. Затем мы добавим младший байт, сохраненный в header[0].

height вычисляется таким же путем, но вместо значений сохраненных в header[0] и header[1], мы используем значения, сохраненные в header[2] и header[3].

После того как мы вычислили ширину и высоту, мы должны проверить, что ширина и высота больше 0. Если ширина или высота меньше или равна нулю, файл будет закрыт и функция вернет false.

Также мы должны удостовериться, что TGA файл является 24 или 32 битным изображением. Это мы сделаем, проверив значение, сохраненное в header[4]. Если оно не равно ни 24, ни 32 (бит), то файл будет закрыт и функция вернет false.

В случае если бы Вы не осуществили проверку, возврат функцией значения false привел бы к аварийному завершению программы с сообщением "Initialization Failed". Убедитесь, что ваш TGA файл является несжатым 24 или 32 битным изображением!

```
// Определяем ширину TGA (старший байт*256+младший байт)
texture->width  = header[1] * 256 + header[0];
// Определяем высоту TGA (старший байт*256+младший байт)
texture->height = header[3] * 256 + header[2];

if(texture->width  <=0 ||           // Если ширина меньше или равна нулю
 texture->height <=0 ||             // Если высота меньше или равна нулю
(header[4]!=24 && header[4]!=32))  // Является ли TGA 24 или 32 битным?
{
  fclose(file);                     // Если где-то ошибка, закрываем файл
  return false;                     // Возвращаем false
}
```

Теперь, когда мы вычислили ширину и высоту изображения, нам необходимо вычислить количество битов на пиксель, байтов на пиксель и размер изображения (в памяти).

Значение, хранящееся в header[4] - это количество битов на пиксель. Поэтому установим bpp равным header[4].

Если Вам известно что-нибудь о битах и байтах, то Вы должны знать, что 8 битов составляют байт. Для того чтобы вычислить количество байтов на пиксель, используемое в TGA файле, все, что мы должны сделать - разделить количество битов на пиксель на 8. Если изображение 32-х битное, то bytesPerPixel будет равняться 4. Если изображение 24-х битное, то bytesPerPixel будет равняться 3.

Для вычисления размера изображения мы умножим width * height * bytesPerPixel. Результат сохраним в imageSize. Так, если изображение было 100x100x32, то его размер будет 100 * 100 * 32/8 = 10000 * 4 = 40000 байтов.

```
texture->bpp  = header[4];        // Получаем TGA бит/пиксель (24 or 32)
bytesPerPixel = texture->bpp/8;   // Делим на 8 для получения байт/пиксель
// Подсчитываем размер памяти для данных TGA
imageSize = texture->width*texture->height*bytesPerPixel;
```

Теперь, когда нам известен размер изображения в байтах, мы должны выделить память под него. Это мы сделаем в первой строке кода ниже. imageData будет указывать на область памяти достаточно большого размера, для того, чтобы поместить туда наше изображение. malloc(imagesize) выделит память (подготовит память для использования), основываясь на необходимом размере (imageSize).

Конструкция "if" осуществляет несколько действий. Первое - проверка того, что память выделена правильно. Если это не так, imageData будет равняться NULL, файл будет закрыт и функция вернет false.

Если память была выделена, мы попытаемся прочитать изображение из файла в память. Это осуществляет строка fread(texture->imageData, 1, imageSize, file). fread читает файл. imageData указывает на область памяти, куда мы хотим поместить прочитанные данные. 1 - это количество байтов, которое мы хотим прочитать (мы хотим читать по одному байту за раз). imageSize - общее количество байтов, которое мы хотим прочитать. Поскольку imageSize равняется общему размеру памяти, достаточному для сохранения изображения, то изображение будет прочитано полностью.

После чтения данных, мы должны проверить, что количество прочитанных данных совпадает со значением, хранящимся в imageSize. Если это не так, значит где-то произошла ошибка. Если были загружены какие-то данные, мы уничтожим их (освободим память, которую мы выделили). Файл будет закрыт и функция вернет false.

```
texture->imageData=(GLubyte *)malloc(imageSize); // Резервируем память для хранения данных TGA
if(texture->imageData==NULL ||           // Удалось ли выделить память?
 fread(texture->imageData, 1, imageSize, file)!=imageSize)
// Размер выделенной памяти равен  imageSize?
{
  if(texture->imageData!=NULL)       // Были ли загружены данные?
    free(texture->imageData);        // Если да, то освобождаем память
  fclose(file);                      // Закрываем файл
  return false;                      // Возвращаем false
}
```

Если данные были загружены правильно, то дела идут хорошо :). Все что мы должны теперь сделать - это обменять местами Красные (Red) и Синие (Blue) байты. Данные в TGA файле хранятся в виде BGR (blue, green, red). Если мы не обменяем красные байты с синими, то все, что в нашем изображении должно быть красным, станет синим и наоборот.

Во-первых, мы создадим цикл (по i) от 0 до imageSize. Благодаря этому, мы пройдемся по всем данным. Переменная цикла (i) на каждом шаге будет увеличиваться на 3 (0, 3, 6, 9, и т.д.) если TGA файл 24-х битный, и на 4 (0, 4, 8, 12, и т.д.) - если изображение 32-х битное. Дело в том, что значение i всегда должно указывать на первый байт ([b]lue байт) нашей группы, состоящей из 3-х или 4-х байтов (BGR или BGRA - прим. пер.).

Внутри цикла мы сохраняем [b]lue байт в переменной temp. Затем мы берем [r]ed байт, хранящийся в texture->imageData[i+2] (помните, что TGA хранит цвета как BGR[A]. B - i+0, G - i+1 и R - i+2) и помещаем его туда, где находился [b]lue байт.

Наконец, мы помещаем [b]lue байт, который мы сохранили в переменной temp, туда, где находился [r]ed байт.
Если все прошло успешно, то теперь TGA хранится в памяти, как пригодные данные для OpenGL текстуры.

```
 for(GLuint i=0; i<int(imageSize); i+=bytesPerPixel)  // Цикл по данным, описывающим изображение
 {                                                    // Обмена 1го и 3го байтов ('R'ed и 'B'lue)
  temp=texture->imageData[i];                         // Временно сохраняем значение imageData[i]
  texture->imageData[i] = texture->imageData[i + 2]; // Устанавливаем 1й байт в значение 3го байта
  texture->imageData[i + 2] = temp;                   // Устанавливаем 3й байт в значение,
                                                      // хранящееся в temp (значение 1го байта)
 }
 fclose (file);                                       // Закрываем файл
```

Теперь, когда у нас есть данные, пришло время сделать из них текстуру. Начнем с того, что сообщим OpenGL о нашем желании создать текстуру в памяти по адресу &texture[0].texID.

Очень важно, чтобы Вы поняли несколько вещей прежде чем мы двинемся дальше. В коде функции InitGL(), когда мы вызываем функцию LoadTGA() мы передаем ей два параметра. Первый параметр - это &textures[0]. В LoadTGA() мы не обращаемся к &textures[0], мы обращаемся к &texture[0](отсутсвует 's' в конце). Когда мы изменяем &texture[0], на самом деле изменяется &textures[0]. texture[0] отождествляется с textures[0]. Я надеюсь, что это понятно.

Таким образом, если мы хотим создать вторую текстуру, то мы должны передать в какестве параметра &textures[1]. В LoadTGA(), изменяя texture[0] мы будем изменять textures[1]. Если мы передадим &textures[2], texture[0] будет связан с &textures[2], и т.д.

Трудно объяснить, легко понять. Конечно, я не успокоюсь, пока не объясню это по-настоящему просто :). Вот бытовой пример. Допустим, что у меня есть коробка. Я назвал ее коробкой № 10. Я дал ее своему другу и попросил его заполнить ее. Моего друга мало заботит, какой у нее номер. Для него это просто коробка. Итак, он заполнил, как он считает "просто коробку" и возвратил мне ее. При этом для меня он заполнил коробку № 10. Он считает, что он заполнил обычную коробку. Если я дам ему другую коробку, названную коробкой № 11, и скажу «эй, можешь ли ты заполнить эту». Для него это опять всего лишь "коробка". Он заполнит и вернет мне ее полной. При этом для меня он заполнил коробку № 11.

Когда я передаю функции LoadTGA() параметр &textures[1], она воспринимает его как &texture[0]. Она заполняет его текстурой, и после завершения ее работы, у меня будет рабочая текстура textures[1]. Если я передам LoadTGA() &textures[2], она опять воспримет его как &texture[0]. Она заполнит его данными, И я останусь с рабочей текстурой textures[2]. В этом есть смысл :).

Во всяком случае... в коде! Мы говорим LoadTGA() построить нашу текстуру. Мы привязываем текстуру и говорим OpenGL, что она должна иметь линейный фильтр.

```
 // Строим текстуру из данных
 glGenTextures(1, &texture[0].texID);  // Сгенерировать OpenGL текстуру IDs
 glBindTexture(GL_TEXTURE_2D, texture[0].texID); // Привязать нашу текстуру
 // Линейная фильтрация
 glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
 // Линейная фильтрация
 glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

Теперь посмотрим, был ли TGA файл 24-х или 32-х битным. Если он был 24-х битным, то установим type в GL_RGB (отсутствует альфа-канал). Если бы мы этого не сделали, то OpenGL попытался бы построить текстуру с альфа-каналом. Так как информация об альфа отсутствует, то или произошел бы сбой программы или появилось бы сообщение об ошибке.

```
 if (texture[0].bpp==24)                  // Если TGA 24 битный
 {
   type=GL_RGB;                           // Установим 'type' в GL_RGB
 }
```

Теперь мы построим нашу текстуру, таким же путем как делали это всегда. Но вместо того, чтобы просто воспользоваться типом (GL_RGB или GL_RGBA), мы заменим его переменной type. Таким образом, если программа определит, что TGA был 24-х битным, то type будет GL_RGB. Если же TGA был 32-х битным, то type будет GL_RGBA.

После того как текстура будет построена, мы возвратим true. Это даст знать коду InitGL(), что все прошло успешно.

```
  glTexImage2D(GL_TEXTURE_2D, 0, type, texture[0].width, texture[0].height,
                        0, type, GL_UNSIGNED_BYTE, texture[0].imageData);
  return true;              // Построение текстуры прошло Ok, возвратим true
}
```

Код ниже является нашим стандартом построения шрифта из текстуры. Все Вы встречали этот код и раньше, если Вы прошли все уроки до этого. Здесь нет ничего нового, но я считаю, что должен включить этот код, для того, чтобы облегчить понимание программы.

Единственным отличием является то, что я привязываю текстуру textures[0].texID, которая указывает на текстуру шрифта. Я добавил только лишь .texID.

```
GLvoid BuildFont(Glvoid)                // Построение нашего шрифта
{
  base=glGenLists(256);                 // Создадим 256 списков отображения
                                // Выбираем нашу текстуру шрифта
  glBindTexture(GL_TEXTURE_2D, textures[0].texID);
  for (int loop1=0; loop1<256; loop1++)  // Цикл по всем 256 спискам
  {
    float cx=float(loop1%16)/16.0f;      // X позиция текущего символа
    float cy=float(loop1/16)/16.0f;      // Y позиция текущего символа
    glNewList(base+loop1,GL_COMPILE);    // Начало построение списка
    glBegin(GL_QUADS);   // Используем квадрат для каждого символа
      glTexCoord2f(cx,1.0f-cy-0.0625f);  // Коорд. текстуры (Низ Лево)
      glVertex2d(0,16);                  // Коорд. вершины  (Низ Лево)
      glTexCoord2f(cx+0.0625f,1.0f-cy-0.0625f);  // Коорд. текстуры (Низ Право)
      glVertex2i(16,16);                 // Коорд. вершины  (Низ Право)
      glTexCoord2f(cx+0.0625f,1.0f-cy-0.001f); // Коорд. текстуры (Верх Право)
      glVertex2i(16,0);                  // Коорд. вершины  (Верх Право)
      glTexCoord2f(cx,1.0f-cy-0.001f);   // Коорд. текстуры (Верх Лево)
      glVertex2i(0,0);                   // Коорд. вершины  (Верх Лево)
    glEnd();                // Конец построения квадрата (символа)
    glTranslated(14,0,0);   // Смещаемся в право от символа
    glEndList();            // Конец построения списка
  }                         // Цикл пока не будут построены все 256 списков
}
```

Функция KillFont не изменилась. Мы создали 256 списков отображения, поэтому мы должны будем уничтожить их, когда программа будет завершаться.

```
GLvoid KillFont(GLvoid)                 // Удаляем шрифт из памяти
{
    glDeleteLists(base,256);                // Удаляем все 256 списков
}
```

Код glPrint() немного изменился. Все буквы теперь растянуты по оси y, что делает их очень высокими. Остальную часть кода я объяснял в прошлых уроках. Растяжение выполняется командой glScalef(x,y,z). На оси x мы оставляем коэффициент равным 1.0, удваиваем размер (2.0) по оси y, и оставляем 1.0 по оси z.

```
GLvoid glPrint(GLint x, GLint y, int set, const char *fmt, ...)  // Здесь происходит печать
{
  char    text[1024];           // Содержит нашу строку
  va_list ap;                   // Указатель на список аргументов

  if (fmt == NULL)              // Если текста нет
    return;                     // Ничего не делаем
  va_start(ap, fmt);            // Разбор строки переменных
  vsprintf(text, fmt, ap);      // И конвертирование символов в реальные коды
  va_end(ap);                   // Результат помещаем в строку
  if (set>1)                    // Если выбран неправильный набор символов
  {
    set=1;                      // Если да, выбираем набор 1 (Italic)
  }
```

```
  glEnable(GL_TEXTURE_2D);      // Разрешаем двумерное текстурирование
  glLoadIdentity();             // Сбрасываем матрицу просмотра модели
  glTranslated(x,y,0);          // Позиционируем текст (0,0 - Верх Лево)
  glListBase(base-32+(128*set));// Выбираем набор шрифта (0 или 1)
  glScalef(1.0f,2.0f,1.0f);     // Делаем текст в 2 раза выше
  glCallLists(strlen(text),GL_UNSIGNED_BYTE, text);// Выводим текст на экран
  glDisable(GL_TEXTURE_2D);     // Запрещаем двумерное текстурирование
}
```

ReSizeGLScene() устанавливает ортографическую проекцию. Ничего нового. 0,1 - это верхний левый угол экрана. 639, 480 соответствует нижнему правому углу экрана. Это дает нам точные экранные координаты с разрешением 640x480. Заметьте, что мы устанавливаем значение swidth равным текущей ширине окна, а значение sheight равным текущей высоте окна. Всякий раз, при изменении размеров или перемещении окна, переменные sheight и swidth будут обновлены.

```
GLvoid ReSizeGLScene(GLsizei width, GLsizei height) // Изменение размеров и инициализация GL окна
{
  swidth=width;               // Устанавливаем ширину вырезки в ширину окна
  sheight=height;             // Устанавливаем высоту вырезки в высоту окна
  if (height==0)              // Предотвращаем деление на нуль
  {
    height=1;                 // Делаем высоту равной 1
  }
  glViewport(0,0,width,height);       // Сбрасываем область просмотра
  glMatrixMode(GL_PROJECTION);        // Выбираем матрицу проекции
  glLoadIdentity();                   // Сбрасываем матрицу проекции
  // Устанавливаем ортографическую проекцию 640x480  (0,0 - Верх Лево)
  glOrtho(0.0f,640,480,0.0f,-1.0f,1.0f);
  glMatrixMode(GL_MODELVIEW);         // Выбираем матрицу просмотра модели
  glLoadIdentity();                   // Сбрасываем матрицу просмотра модели
}
```

Код инициализации очень мал. Мы загружаем наш TGA файл. Заметьте, что первым параметром передается &textures[0]. Второй параметр - имя файла, который мы хотим загрузить. В нашем случае, мы хотим загрузить файл Font.TGA. Если LoadTGA() вернет false по какой-то причине, выражение if также вернет false, что приведет к завершению программы с сообщением "initialization failed".

Если Вы захотите загрузить вторую текстуру, то Вы должны будете использовать следующий код: if ((!LoadTGA(&textures[0],"image1.tga")) || (!LoadTGA(&textures[1],"image2.tga"))) { }

После того как мы загрузили TGA (создали нашу текстуру), мы строим наш шрифт, устанавливаем плавное сглаживание, делаем фоновый цвет черным, разрешаем очистку буфера глубины и выбираем нашу текстуру шрифта (привязываемся к ней).

Наконец, мы возвращаем true, и тем самым даем знать нашей программе, что инициализация прошла ok.

```
int InitGL(Glvoid)                        // Все настройки для OpenGL
{
  if (!LoadTGA(&textures[0],"Data/Font.TGA"))// Загружаем текстуру шрифта
  {
    return false;                         // Если ошибка, возвращаем false
  }
  BuildFont();                            // Строим шрифт

  glShadeModel(GL_SMOOTH);                // Разрешаем плавное сглаживание
  glClearColor(0.0f, 0.0f, 0.0f, 0.5f);   // Черный фон
  glClearDepth(1.0f);                     // Устанавливаем буфер глубины
  glBindTexture(GL_TEXTURE_2D, textures[0].texID); // Выбираем нашу текстуру шрифта
  return TRUE;                            // Инициализация прошла OK
}
```

Код отрисовки совершенно новый :). Мы начинаем с того, что создаем переменную token типа char. Token будет хранить текст, разобранный далее в коде.

У нас есть другая переменная, названная cnt. Я использую эту переменную, как для подсчета количества поддерживаемых расширений, так и для позиционирования текста на экране. cnt сбрасывается в нуль каждый раз, когда мы вызываем DrawGLScene.

Мы очищаем экран и буфер глубины, затем устанавливаем цвет в ярко-красный (красный полной интенсивности, 50% зеленый, 50% синий). С позиции 50 по оси x и 16 по оси y мы выводим слово "Renderer". Также мы выводим "Vendor" и "Version" вверху экрана. Причина, по которой каждое слово начинается не с 50 по оси x, в том, что я выравниваю все эти слова по их правому краю (все они выстраиваются по правой стороне).

```
int DrawGLScene(GLvoid)                 // Здесь происходит все рисование
{
  char *token;                          // Место для хранения расширений
  int  cnt=0;                           // Локальная переменная цикла

  // Очищаем экран и буфер глубины
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glColor3f(1.0f,0.5f,0.5f);            // Устанавливаем цвет в ярко-красный
  glPrint(50,16,1,"Renderer");          // Выводим имя производителя
  glPrint(80,48,1,"Vendor");            // Выводим имя поставщика
  glPrint(66,80,1,"Version");           // Выводим версию карты
```

Теперь, когда у нас есть текст на экране, мы изменяем цвет на оранжевый, и считываем из видеокарты имя производителя, имя поставщика и номер версии видеокарты. Мы делаем это, передавая GL_RENDERER, GL_VENDOR и GL_VERSION в glGetString(). glGetString вернет запрошенные имя производителя, имя поставщика и номер версии. Возвращаемая информация будет текстом, поэтому мы должны запросить информацию от glGetString как char. Это значит, что мы сообщаем программе, что мы хотим, чтобы возвращаемая информация была символами (текст). Если Вы не включите (char *), то Вы получите сообщение об ошибке. Мы печатаем в текстовом виде, поэтому нам необходимо возвратить текст. Мы получаем все три части информации и выводим их справа от предыдущего текста.

Информация, которую мы получим от glGetString(GL_RENDERER), будет выведена рядом с красным текстом "Renderer", а информация, которую мы получим от glGetString(GL_VENDOR), будет выведена справа от "Vendor", и т.д.

Мне бы хотелось объяснить процесс приведения типа более подробно, но у меня нет по-настоящему хорошего объяснения. Если кто-то может хорошо объяснить это, напишите мне, и я изменю мои пояснения.

После того, как мы поместили информацию о производителе, имя поставщика и номер версии на экран, мы изменяем цвет на ярко-синий и выводим "NeHe Productions" в низу экрана :). Конечно, Вы можете изменить этот текст как угодно.

```
  glColor3f(1.0f,0.7f,0.4f);            // Устанавливаем цвет в оранжевый
  glPrint(200,16,1,(char *)glGetString(GL_RENDERER));// Выводим имя производителя
  glPrint(200,48,1,(char *)glGetString(GL_VENDOR));  // Выводим имя поставщика
  glPrint(200,80,1,(char *)glGetString(GL_VERSION)); // Выводим версию
  glColor3f(0.5f,0.5f,1.0f);            // Устанавливаем цвет в ярко-голубой
  glPrint(192,432,1,"NeHe Productions"); // Печатаем NeHe Productions в низу экрана
```

Сейчас мы нарисуем красивую белую рамку вокруг экрана и вокруг текста. Мы начнем со сброса матрицы просмотра модели. Поскольку мы напечатали текст на экране и находимся не в точке 0,0 экрана, это лучший способ для возврата в 0,0.

Затем мы устанавливаем цвет в белый и начинаем рисовать наши рамки. Ломаная линия достаточно легка в использовании. Мы говорим OpenGL, что хотим нарисовать ломаную линию с помощью glBegin(GL_LINE_STRIP). Затем мы устанавливаем первую вершину. Наша первая вершина будет находиться на краю правой части экрана и на 63 пиксела вверх от нижнего края экрана (639 по оси x и 417 по оси y). После этого мы устанавливаем вторую вершину. Мы остаемся в том же месте по оси y (417), но по оси x сдвигаемся на левый край (0). Линия будет нарисована из правой части экрана (639,417) в левую часть(0,417).

У Вас должно быть, по крайней мере, две вершины для рисования линии (как подсказывает здравый смысл). Из левой части экрана мы перемещаемся вниз, вправо, и затем вверх (128 по оси y).

Затем мы начинаем другую ломаную линию, и рисуем вторую рамку вверху экрана. Если Вам нужно нарисовать много соединенных линий, то ломаная линия определенно позволит снизить количество кода, который был бы необходим для рисования регулярных линий(GL_LINES).

```
glLoadIdentity();               // Сбрасываем матрицу просмотра модели
glColor3f(1.0f,1.0f,1.0f);      // Устанавливаем цвет в белый
glBegin(GL_LINE_STRIP);         // Начало рисования ломаной линии
  glVertex2d(639,417);          // Верх Право нижней рамки
  glVertex2d(  0,417);          // Верх Лево нижней рамки
  glVertex2d(  0,480);          // Низ Лево нижней рамки
  glVertex2d(639,480);          // Низ Право нижней рамки
  glVertex2d(639,128);          // Вверх к Низу Права верхней рамки
glEnd();                        // Конец первой ломаной линии
glBegin(GL_LINE_STRIP);         // Начало рисования другой ломаной линии
  glVertex2d(  0,128);          // Низ Лево верхней рамки
  glVertex2d(639,128);          // Низ Право верхней рамки
  glVertex2d(639,  1);          // Верх Право верхней рамки
  glVertex2d(  0,  1);          // Верх Лево верхней рамки
glVertex2d(  0,417);            // Вниз к Верху Лева нижней рамки
glEnd();                        // Конец второй ломаной линии
```

А теперь кое-что новое. Чудесная GL команда, которая называется glScissor(x,y,w,h). Эта команда создает почти то, что можно называть окном. Когда разрешен GL_SCISSOR_TEST, то единственной частью экрана, которую Вы можете изменять, является та часть, которая находится внутри вырезанного окна. Первая строка ниже создает вырезанное окно, начиная с 1 по оси x и 13.5% (0.135...f) пути снизу экрана по оси y. Вырезанное окно будет иметь 638 пикселов в ширину(swidth-2) и 59.7%(0.597...f) экрана в высоту.

В следующей строке мы разрешаем вырезку. Что бы мы ни рисовали за пределами вырезанного окна, не появится. Вы можете нарисовать ОГРОМНЫЙ четырехугольник на экране с 0,0 до 639,480, но Вы увидите только ту часть, которая попала в вырезанное окно. Оставшаяся часть экрана не будет видна. Замечательная команда!

Третья строка кода создает переменную text, которая будет хранить символы, возвращаемые glGetString(GL_EXTENSIONS). malloc(strlen((char *)glGetString(GL_EXTENSIONS))+1) выделяет достаточно памяти, для хранения всей строки, которая будет возвращена, + 1 (таким образом, если строка содержит 50 символов, то text будет в состоянии хранить все 50 символов).

Следующая строка копирует информацию GL_EXTENSIONS в text. Если мы непосредственно модифицируем информацию GL_EXTENSIONS, то возникнут большие проблемы, поэтому вместо этого мы копируем информацию в text, и затем манипулируем информацией, сохраненной в text. По сути, мы просто берем копию и сохраняем ее в переменной text.

```
// Определяем область вырезки
glScissor(1,int(0.135416f*sheight),swidth-2,int(0.597916f*sheight));
glEnable(GL_SCISSOR_TEST);      // Разрешаем вырезку

// Выделяем память для строки расширений
char* text=(char*)malloc(strlen((char *)glGetString(GL_EXTENSIONS))+1);

// Получаем список расширений и сохраняем его в text
strcpy (text,(char *)glGetString(GL_EXTENSIONS));
```

Сейчас, немного нового. Давайте предположим, что после захвата информации о расширениях из видеокарты, в переменной text хранится следующая строка... "GL_ARB_multitexture GL_EXT_abgr GL_EXT_bgra". strtok(TextToAnalyze,TextToFind) будет сканировать переменную text пока не найдет в ней " "(пробел). Как только пробел будет найден, будет скопировано содержимое text ВПЛОТЬ ДО пробела в переменную token. В нашем случае, token будет равняться "GL_ARB_multitexture". Затем пробел заменится маркером. Подробнее об этом через минуту.

Далее мы создаем цикл, который остановится когда больше не останется информации в text. Если в text нет информации, то token будет равняться NULL и цикл остановится.

Мы увеличиваем переменную счетчик (cnt) на единицу, и затем проверяем больше ли значение cnt чем maxtokens. Если cnt больше чем maxtokens, то мы приравниваем maxtokens к cnt. Таким образом, если счетчик достиг 20-ти, то maxtokens будет также равен 20. Это необходимо для того, чтобы следить за максимальным значением счетчика.

```
// Разбиваем 'text' на слова, разделенные " " (пробелом)
token=strtok(text," ");
while(token!=NULL)              // Пока token не NULL
{
  cnt++;                // Увеличиваем счетчик
  if (cnt>maxtokens)        // Если 'maxtokens' меньше или равно 'cnt'
  {
     maxtokens=cnt; // Если да, то 'maxtokens' приравниваем к 'cnt'
  }
```

Итак, мы сохранили первое расширение из нашего списка расширений в переменную token. Следующее, что мы сделаем - установим цвет в ярко-зеленый. Затем мы напечатаем переменную cnt в левой части экрана. Заметьте, что мы печатаем с позиции 0 по оси x. Это могло бы удалить левую (белую) рамку, которую мы нарисовали, но так как включена вырезка, то пиксели, нарисованные в 0 по оси x, не будут изменены. Не получится нарисовать поверх рамки.

Переменная будет выведена левого края экрана (0 по оси x). По оси y мы начинаем рисовать с 96. Для того чтобы весь текст не выводился в одной и той же точке экрана, мы добавил (cnt*32) к 96. Так, если мы отображаем первое расширение, то cnt будет равно 1, и текст будет нарисован с 96+(32*1)(128) по оси y. Если мы отображаем второе расширение, то cnt будет равно 2, и текст будет нарисован с 96+(32*2)(160) по оси y.

Заметьте, что я всегда вычитаю scroll. Во время запуска программы scroll будет равняться 0. Так, наша первая строка текста рисуется с 96+(32*1)-0. Если Вы нажмете СТРЕЛКА ВНИЗ, то scroll увеличится на 2. Если scroll равняется 4, то текст будет нарисован с 96+(32*1)-4. Это значит, что текст будет нарисован с 124 вместо 128 по оси y, поскольку scroll равняется 4. Верх нашего вырезанного окна заканчивается в 128 по оси y. Любая часть текста, рисуемая в строках 124-127 по оси y, не появится на экране.

Тоже самое и с низом экрана. Если cnt равняется 11 и scroll равняется 0, то текст должен быть нарисован с 96+(32*11)-0 и в 448 по оси y. Поскольку вырезанное окно позволяет нам рисовать только до 416 по оси y, то текст не появится вообще.

Последнее, что нам нужно от прокручиваемого окна, это возможность посматривать 288/32 (9) строк текста. 288 - это высота нашего вырезанного окна. 32 - высота нашего текста. Изменяя значение scroll, мы можем двигать текст вверх или вниз (смещать текст).

Эффект подобен кинопроектору. Фильм прокручивается через линзу и все, что Вы видите - текущий кадр. Вы не видите область выше или ниже кадра. Линза выступает в качестве окна, аналогично окну, созданному при помощи вырезки.

После того, как мы нарисовали текущий счетчик (cnt) на экране, изменяем цвет на желтый, передвигаемся на 50 пикселов вправо по оси x, и выводим текст, хранящийся в переменной token на экран.

Используя наш пример выше, первая строка текста, которая будет отображена на экране, будет иметь вид:

1 GL_ARB_multitexture

```
  glColor3f(0.5f,1.0f,0.5f);      // Устанавливаем цвет в ярко-зеленый
  glPrint(0,96+(cnt*32)-scroll,0,"%i",cnt); // Печатаем текущий номер расширения
  glColor3f(1.0f,1.0f,0.5f);            // Устанавливаем цвет в желтый
  glPrint(50,96+(cnt*32)-scroll,0,token);     // Печатаем текущее расширение
```

После того, как мы отобразили значение token на экране, мы должны проверить переменную text: есть ли еще поддерживаемые расширения. Вместо использования token=strtok(text," "), как мы делали выше, мы замещаем text на NULL. Это сообщает команде strtok, что искать нужно от последнего маркера до СЛЕДУЮЩЕГО пробела в строке текста (text).
В нашем примере выше ("GL_ARB_multitexturemarkerGL_EXT_abgr GL_EXT_bgra") маркер будет находиться после текста "GL_ARB_multitexture". Строка кода ниже начнет поиск ОТ маркера до следующего пробела. Все, находящееся от маркера до следующего пробела, будет сохранено в token. В token будет помещено "GL_EXT_abgr", в text будет храниться GL_ARB_multitexturemarkerGL_EXT_abgrmarkerGL_EXT_bgra".

Когда у strtok() не останется текста для сохранения в token, token станет равным NULL и цикл остановится.

```
  token=strtok(NULL," ");              // Поиск следующего расширения
 }
```

После того, как все расширения будут разобраны из переменной text, мы можем запретить вырезку и освободить переменную text. Это освобождает память, которую мы использовали для хранения информации, полученной от glGetString(GL_EXTENSIONS).

При следующем вызове DrawGLScene(), будет выделена новая память. Свежая копия информации, которую вернет glGetStrings(GL_EXTENSIONS), будет скопирована с переменную text и весь процесс начнется заново.

```
 glDisable(GL_SCISSOR_TEST);              // Запрещаем вырезку
 free (text);                             // Освобождаем выделенную память
```

Первая строка ниже необязательна, но я подумал, что это хороший случай рассказать о ней, чтобы все знали, что она существует. Команда glFlush() в основном говорит OpenGL закончить то, что он делает. Если Вы когда-нибудь заметите мерцание в Вашей программе (исчезновение четырехугольников, и т.д.), то попытайтесь добавить команду flush в конец DrawGLScene.

Последнее, что мы делаем - возвращаем true, чтобы показать, что все прошло ok.

```
 glFlush();                               // Сброс конвейера рендеринга
 return TRUE:                             // Все прошло ok
}
```

Единственно, что стоит отметить в KillGLWindow() - в конец я добавил KillFont(). Таким образом, когда окно будет уничтожено, шрифт также будет уничтожен.

```
GLvoid KillGLWindow(GLvoid)               // Правильное уничтожение окна
{
  if (fullscreen)                         // Полноэкранный режим?
  {
    ChangeDisplaySettings(NULL,0);    // Переход в режим разрешения рабочего стола
    ShowCursor(TRUE);// Показать указатель мыши
  }

  if (hRC)                                // Существует контекст рендеринга?
  {
    if (!wglMakeCurrent(NULL,NULL))   // Можно ли освободить DC и RC контексты?
    {
      MessageBox(NULL,"Release Of DC And RC Failed.","SHUTDOWN ERROR",
                                MB_OK | MB_ICONINFORMATION);
    }
    if (!wglDeleteContext(hRC))         // Можно ли уничтожить RC?
    {
      MessageBox(NULL,"Release Rendering Context Failed.",
           "SHUTDOWN ERROR",MB_OK | MB_ICONINFORMATION);
    }
    hRC=NULL;                             // Установим RC в NULL
  }
  if (hDC && !ReleaseDC(hWnd,hDC))      // Можно ли уничтожить DC?
 {
    MessageBox(NULL,"Release Device Context Failed.","SHUTDOWN ERROR",
                              MB_OK | MB_ICONINFORMATION);
    hDC=NULL;                             // Установим DC в NULL
  }
  if (hWnd && !DestroyWindow(hWnd))     // Можно ли уничтожить окно?
  {
    MessageBox(NULL,"Could Not Release hWnd.","SHUTDOWN ERROR",MB_OK |
                                  MB_ICONINFORMATION);
    hWnd=NULL;                            // Устновим hWnd в NULL
  }
```

```
  if (!UnregisterClass("OpenGL",hInstance)) // Можно ли уничтожить класс?
  {
    MessageBox(NULL,"Could Not Unregister Class.","SHUTDOWN ERROR",MB_OK |
                                        MB_ICONINFORMATION);
    hInstance=NULL;                     // Устанавливаем hInstance в NULL
  }
  KillFont();                   // Уничтожаем шрифт
}
```

CreateGLWindow(), и WndProc() - те же.

Первое изменение в WinMain() - название, которое появляется вверху окна. Теперь оно будет "NeHe's Extensions, Scissoring, Token & TGA Loading Tutorial".

```
int WINAPI WinMain(HINSTANCE hInstance,      // Экземпляр
                   HINSTANCE hPrevInstance,  // Предыдущий экземпляр
                   LPSTR     lpCmdLine,      // Параметры командной строки
                   Int       nCmdShow)       // Состояние окна
{
 MSG msg;                       // Структура сообщения Windows
 BOOL done=FALSE;               // Логическая переменная выхода из цикла

 // Спрашиваем у юзера какой режим он предпочитает
 if (MessageBox(NULL,"Would You Like To Run In Fullscreen Mode?",
      "Start FullScreen?", MB_YESNO | MB_ICONQUESTION)==IDNO)
 {
  fullscreen=FALSE;             // Оконный режим
 }

 // Создание OpenGL окна
 if (!CreateGLWindow("NeHe's Token, Extensions, Scissoring & TGA Loading
                                  Tutorial",640,480,16,fullscreen))
 {
  return 0;                     // Выход, если окно не было создано
 }
 while(!done)                   // Цикл выполняется пока done=FALSE
 {
  if (PeekMessage(&msg,NULL,0,0,PM_REMOVE)) // Есть сообщение?
  {
   if (msg.message==WM_QUIT)    // Получили сообщение Quit?
   {
    done=TRUE;                  // Если да, то done=TRUE
   }
   else // Если нет, то обрабатываем оконные сообщения
   {
    DispatchMessage(&msg);      // Отправляем сообщение
   }
  }
          else                          // Если нет сообщений
  {
    // Рисуем сцену. Проверяем клавишу ESC и сообщение QUIT из DrawGLScene()
    // Активно?  Получили Quit сообщение?
    if ((active && !DrawGLScene()) || keys[VK_ESCAPE])
    {
     done=TRUE; // ESC или DrawGLScene сигнализирует о выходе
    }
    else                        // Не время выходить, обновляем экран
    {
     SwapBuffers(hDC);          // Меняем буфера (двойная буферизация)
     if (keys[VK_F1])           // Нажата клавиша F1?
     {
      keys[VK_F1]=FALSE;        // Если да, то установим в FALSE
      KillGLWindow();           // Уничтожаем наше текущее окно
      fullscreen=!fullscreen; // Полноэкран./окон. режим
```

```
      // Создаем наше OpenGL окно
      if (!CreateGLWindow("NeHe's Token, Extensions,
      Scissoring & TGA Loading Tutorial", 640,480,16,fullscreen))
      {
        return 0; // Выход если окно не было создано
      }
    }
```

Код ниже проверяет: если была нажата стрелка вверх и scroll больше 0, то уменьшаем scroll на 2. Это заставляет сдвинуться вниз текст на экране.

```
    if (keys[VK_UP] && (scroll>0))  // Нажата стрелка вверх?
    {
      scroll-=2; // Если да, то уменьшаем 'scroll', двигая экран вниз
    }
```

Если была нажата стрелка вниз, и scroll меньше чем (32*(maxtokens-9)), то scroll будет увеличена на 2, и текст на экране сдвинется вверх.

32 - это количество пикселей, занимаемое каждой строкой. Maxtokens - общее количество расширений, поддерживаемых Вашей видеокартой. Мы вычитаем 9, поскольку 9 строк могут одновременно показываться на экране. Если мы не вычтем 9, то сможем сдвигать за пределы списка, что приведет к тому, что список полностью сдвинется за пределы экрана. Попробуйте убрать -9, если Вы не понимаете, о чем я говорю.

```
    if (keys[VK_DOWN] && (scroll<32*(maxtokens-9))) // Нажата стрелка вниз?
    {
      scroll+=2; // Если да, то увеличиваем 'scroll', двигая экран вверх
    }
  }
 }
}

 // Завершение
 KillGLWindow(); // Уничтожаем окно
 return (msg.wParam); // Выходим из программы
}
```

Я надеюсь, что Вы нашли этот урок интересным. К концу этого урока Вы должны знать, как считывать имя производителя, поставщика и номер версии из Вашей видеокарты. Также Вы должны уметь определять, какие расширения поддерживаются любой видеокартой, которая поддерживает OpenGL. Также Вы должны знать, что такое вырезка и как можно использовать ее в своих OpenGL проектах, и, наконец, Вы должны знать, как загружать TGA изображения вместо BMP для использования их в качестве текстур.

Если у Вас возникли какие-либо проблемы с этим уроком, или Вы находите, что эта информация трудна для понимания, дайте мне знать. Я хочу сделать свои уроки лучше, насколько это возможно. Обратная связь очень важна!

Урок 25. Морфинг и загрузка объектов из файла.

Morphing & Loading Objects From A File

Добро пожаловать в еще один потрясающий урок! На этот раз мы сосредоточимся не на графике, а на эффекте, хотя результат все равно будет выглядеть очень круто. В этом уроке Вы научитесь осуществлять морфинг – плавное "превращение" одного объекта в другой. Подобный эффект я использую в демонстрации dolphin. Надо сделать несколько замечаний. Прежде всего, стоит отметить, что каждый объект должен состоять из одинакового количества точек. Очень редко удается получить три объекта, содержащих точно одно и тоже количество вершин, но, к счастью, в этом уроке у нас имеются три объекта с одинаковым количеством точек :). Не поймите меня неправильно, – Вы можете использовать объекты с разным количеством вершин, но тогда переход одного объекта в другой не будет выглядеть так четко и плавно.

Также Вы научитесь считывать объект из файла. Формат файла подобен формату, используемому в уроке 10, хотя код легко можно изменить для чтения .ASC файлов или других текстовых файлов. В общем, это действительно крутой эффект и действительно крутой урок. Итак, приступим!

Начинаем как обычно. Подключаем заголовочные файлы, в том числе необходимые для работы с математическими функциями и стандартной библиотекой ввода/вывода. Заметьте, что мы не подключаем библиотеку GLAUX. В этом уроке мы будем рисовать точки, а не текстуры. После того, как Вы поймете урок, Вы сможете поиграть с полигонами, линиями и текстурами!

```
#include <windows.h>       // Заголовочный файл для Windows
#include <math.h>          // Заголовочный файл для математической библиотеки
#include <stdio.h>         // Заголовочный файл для стандартного ввода/вывода
#include <gl\gl.h>         // Заголовочный файл для библиотеки OpenGL32
#include <gl\glu.h>        // Заголовочный файл для библиотеки GLu32

HDC       hDC=NULL;        // Контекст устройства
HGLRC     hRC=NULL;        // Контекст рендеринга
HWND      hWnd=NULL;       // Дескриптор окна
HINSTANCE hInstance;       // Экземпляр приложения

bool      keys[256];       // Массив для работы с клавиатурой
bool      active=TRUE;     // Флаг активности приложения
bool      fullscreen=TRUE; // Флаг полноэкранного режима
```

После установки всех стандартных переменных, мы добавим несколько новых. Переменные xrot, yrot и zrot будут хранить текущие значения углов вращения экранного объекта по осям x, y и z. Переменные xspeed, yspeed и zspeed будут контролировать скорость вращения объекта по соответствующим осям. Переменные cx, cy и cz определяют позицию рисования объекта на экране (cx – слева направо, cy – снизу вверх и cz – в экран и от него).

Переменную key я включил для того, чтобы убедиться в том, что пользователь не будет пытаться сделать морфинг первой формы на себя. Это было бы красивой бессмысленностью и вызвало бы задержку, за счет морфинга точек в позицию, в которой они уже находятся.

step – это переменная-счетчик, которая постепенно увеличивается до значения steps. Если Вы увеличите значение переменной steps, то морфинг объекта займет больше времени, зато морфинг будет осуществлен более плавно. Как только переменная step станет равной steps, мы поймем, что морфинг завершился.

Последняя переменная morph дает знать нашей программе, нужно ли осуществить морфинг точек или же не трогать их. Если она установлена в TRUE, то объект находится в процессе морфинга из одной фигуры в другую.

```
Lfloat xrot, yrot, zrot,       // углы вращения по X, Y и Z
      xspeed, yspeed, zspeed, // скорость вращения по X, Y и Z
      cx, cy, cz=-15;         // положение на X, Y и Z
int    key=1;                 // Используется для проверки морфинга
int    step=0, steps=200;     // Счетчик шага и максимальное число шагов
bool   morph=FALSE;           // По умолчанию morph равен False (морфинг выключен)
```

Здесь мы создаем структуру для хранения вершин. В ней будут храниться координаты x, y и z любой точки на экране. Переменные x, y и z – вещественные, поэтому мы можем позиционировать точку в любом месте экрана с большой точностью.

```
typedef struct                // Структура для вершины
{
      float x, y, z;          // X, Y и Z координаты
} VERTEX;                     // Назовем ее VERTEX
```

Итак, у нас есть структура для хранения вершин. Нам известно, что объект состоит из множества вершин, тогда давайте создадим структуру OBJECT. Первая переменная verts является целым числом, определяющим количество вершин необходимое для создания объекта. Таким образом, если наш объект состоит из 5 вершин, то verts будет равно 5. Мы установим значение позже в коде. Все что Вам нужно знать сейчас – это то, что verts следит за тем, сколько точек будет использовано для создания объекта.

Переменная points будет указывать на переменную типа VERTEX(значения x, y и z). Это даст нам возможность захватывать значения x, y и z координат любой точки, используя выражение points[{точка, к которой нужно осуществить доступ}].{x, y или z}.

Имя этой структуры… Вы угадали… OBJECT!

```
typedef struct        // Структура для объекта
{
  int     verts;      // Количество вершин в объекте
  VERTEX* points;     // Одна вершина
} OBJECT;             // Назовем ее OBJECT
```

Теперь, когда мы создали структуру VERTEX и структуру OBJECT, мы можем определить некоторые объекты.

Переменная maxver будет хранить самое большое количество вершин, из всех используемых в объектах. Если один объект содержит всего 5 точек, другой – 20, а последний объект – 15, то значение maxver будет равно самому большему из них. Таким образом, значение maxver будет равно 20.

После того, как мы определили переменную maxver, мы можем определить объекты. morph1, morph2, morph3, morph4 и helper – все определены как переменные типа OBJECT. *sour и *dest – определены как переменные типа OBJECT* (указатели на объект). Объект состоит из вершин (VERTEX). Первые четыре переменных morph{номер} будут хранить объекты, которые мы и будем подвергать морфингу. helper будет использоваться для отслеживания изменений морфинга объекта. *sour будет указывать на объект-источник, а *dest будет указывать на объект, в который мы хотим осуществить морфинг (объект-назначение).

```
int    maxver;                  // Хранит максимум числа вершин объектов
OBJECT morph1,morph2,morph3,morph4, // Наши объекты для морфинга (morph1, 2, 3 и 4)
       helper,*sour,*dest;      // Вспомогательный объект, Источник и Назначение
```

Так же, как всегда, объявляем WndProc().

```
LRESULTCALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);  // Объявление WndProc
```

В коде ниже происходит выделение памяти для каждого объекта, основанное на числе вершин которое мы передаем параметром n. *k указывает на объект, для которого мы хотим выделить память.

В строке кода расположенной между скобками выделяется память для переменной points объекта k. Размер выделенной памяти будет равен размеру структуры VERTEX (3 переменных типа float) умноженному на количество вершин (n). Так, если бы было 10 вершин (n=10), то мы выделили бы место для 30 вещественных значений (3 переменных типа float * 10 вершин).

```
void objallocate(OBJECT *k, int n) // Выделение памяти для каждого объекта
{                                  // И определение точек
    k->points=(VERTEX*)malloc(sizeof(VERTEX)*n); // points = размер(VERTEX)* число вершин
}                                  // (3 точки для каждой вершины)
```

Следующий код удаляет объект и освобождает память, используемую объектом. Объект задается параметром k. Функция free() говорит нашей программе освободить память, занимаемую вершинами, которые были использованы для создания нашего объекта (k).

```
void objfree(OBJECT *k)                 // Удаление объекта (Освобождение памяти)
{
    free(k->points);                    // Освобождаем память, занимаемую points
}
```

Код ниже считывает строку из текстового файла. Указатель на нашу файловую структуру задается параметром *f. Считанная строка будет помещена в переменную string.

Начинаем с создания do-while цикла. Функция fgets() прочитает 255 символов из нашего файла f и сохранит их в переменной *string. Если считанная строка пуста (перевод строки \n), то цикл будет продолжать искать строку с текстом. Оператор while() проверяет наличие пустых строк, и, в случае успеха, начинает цикл заново.

После того, как строка будет прочитана, функция возвратит управление.

```
void readstr(FILE *f, char *string)  // Считывает строку из файла (f)
{
    do                               // Повторять
    {
        fgets(string, 255, f);     // Считывание 255 символов из файла f в переменную string
```

```
    } while ((string[0] == '/') || (string[0] == '\n')); // Пока строка пуста
    return;                         // Возврат
}
```

Здесь мы загружаем объект из файла. *name указывает на имя файла. *k указывает на объект, в который мы хотим загрузить данные.

Мы начинаем с переменной целого типа ver. В ver будет храниться количество вершин, используемое для построения объекта.

В переменных rx, ry и rz будут храниться значения x, y и z каждой вершины.

Переменная filein является указателем на нашу файловую структуру, массив oneline[] будет использоваться для хранения 255 текстовых символов.

Мы открываем файл на чтение как текстовый (значение символа CTRL-Z означает конец строки). Затем мы считываем строку, используя readstr(filein, oneline). Строка текста будет сохранена в массиве oneline.

После того, как строка считана, мы сканируем ее (oneline) пока не найдем фразу "Vertices: {какое-то число}{\n}". Если фраза найдена, то число сохраняется в переменной ver. Это число является количеством вершин, используемых для построения объекта. Если Вы посмотрите в текстовые файлы описания объектов, то Вы увидите, что первой строкой является: Vertices: {какое-то число}.

Теперь, когда нам известно количество используемых вершин, мы сохраняем его в переменных verts объектов. Объекты могут иметь различное количество вершин, поэтому их значения verts могут отличаться.

Последнее, что мы делаем в этой секции кода – это выделение памяти для объекта. Это делается вызовом objallocate({имя объекта}, {количество вершин}).

```
void objload(char *name, OBJECT *k)  // Загружает объект из файла (name)
{
    int       ver;                // Будет хранить количество вершин
    float     rx, ry, rz;           // Будут хранить x, y и z координаты вершины
    FILE  *filein;                  // Файл для работы
    char    oneline[255];           // Хранит одну строку текста (до 255 символов)

    filein = fopen(name, "rt");     // Открываем файл на чтение (CTRL-Z означает конец файла)
    readstr(filein, oneline);       // Считываем одну строку текста из файла
                                    // Сканируем текст на "Vertices: ".
                                    // Число вершин сохраняем в ver
    sscanf(oneline, "Vertices: %d\n", &ver);
    k->verts=ver;                   // Устанавливаем переменные verts объектов
                                    // равными значению ver
    objallocate(k, ver);            // Выделяем память для хранения объектов
```

Мы знаем, из скольких вершин состоит объект. Память выделена. Все, что нам осталось сделать – это считать вершины. Мы создаем цикл по всем вершинам, используя переменную i.

Далее, мы считываем строку текста. Это будет первая строка корректного текста, следующая за строкой "Vertices: {какое-то число}". В конечном итоге мы считываем строку с вещественными значениями координат x, y и z.

Строка анализируется функцией sscanf() и три вещественных значения извлекаются и сохраняются в переменных rx, ry и rz.

```
for (int i=0; i<ver; i++)       // Цикл по всем вершинам
{
    readstr(filein, oneline);   // Считывание следующей строки
                                // Поиск 3 вещественных чисел и сохранение их в rx, ry и rz
    sscanf(oneline, "%f %f %f", &rx, &ry, &rz);
```

Следующие три строки кода сложно объяснить, если Вы не понимаете что такое структуры, и т.п., но я попытаюсь :) Строку k->points[i].x=rx можно разобрать следующим образом:

rx – это значение координаты x одной из вершин.

points[i].x – это координата x вершины points[i].

Если i = 0, то мы устанавливаем значение координаты x вершины 1, если i = 1, то мы устанавливаем значение координаты x вершины 2, и т. д.

points[i] является частью нашего объекта (представленного как k).

Таким образом, если i = 0, то мы говорим: координата x вершины 1 (points[0].x) нашего объекта (k) равняется значению координаты x, считанному нами из файла (rx).

Оставшиеся две строки устанавливают значения координат y и z каждой вершины нашего объекта.

Цикл проходит по всем вершинам. Во избежание ошибки, в случае если вершин будет не достаточно, убедитесь в том, что текст в начале файла “Vertices: {какое-то число}” соответствует действительному числу вершин в файле. То есть, если верхняя строка файла говорит “Vertices: 10”, то за ней должно следовать описание 10 вершин (значения x, y и z).

После считывания всех вершин, мы закрываем файл и проверяем больше ли переменная ver, чем переменная maxver. Если ver больше maxver, то мы делаем maxver равной ver. Таким образом, если мы считали первый объект, состоящий из 20 вершин, то maxver станет равной 20. Далее, если мы считали следующий объект, состоящий из 40 вершин, то maxver станет равной 40. Таким образом, мы узнаем, сколько вершин содержит наш самый большой объект.

```
        k->points[i].x = rx;   // Устанавливаем значение points.x объекта (k) равным rx
        k->points[i].y = ry;   // Устанавливаем значение points.y объекта (k) равным ry
        k->points[i].z = rz;   // Устанавливаем значение points.z объекта (k) равным rz
    }
    fclose(filein);            // Закрываем файл
    if(ver > maxver) maxver=ver;// Если ver больше чем maxver, устанавливаем maxver равным ver
}                              // Следим за максимальным числом используемых вершин
```

Следующий кусок кода может показаться немного пугающим… это не так :). Я объясню его настолько подробно, что Вы будете смеяться, когда в следующий раз посмотрите на это.

Код ниже всего лишь вычисляет новую позицию для каждой точки, когда включен морфинг. Номер вычисляемой вершины храниться в i. Возвращаемый результат вычислений будет иметь тип VERTEX.

Мы создаем переменную a типа VERTEX, что позволяет нам работать с a как с совокупностью значений x, y и z.

Давайте посмотрим на первую строку. Значению x вершины a присваивается разность значений x вершин sour->points[i].x объекта-ИСТОЧНИКА и dest->points[i].x объекта-НАЗНАЧЕНИЯ, деленная на steps.

Давайте для примера возьмем какие-нибудь числа. Пусть значение x нашего объекта-источника равняется 20, а значение x объекта-назначения равняется 40. Мы знаем, что steps равно 200! Тогда a.x=(40-20)/200=(20)/200=0.1

Это значит, что для перемещения из точки 40 в точку 20 за 200 шагов, мы должны перемещаться на 0.1 единиц за один раз. Для доказательства этого умножьте 0.1 на 200, и Вы получите 20. 40-20=20 :)

То же самое мы делаем для вычисления количества единиц, на которые нужно перемещаться по осям y и z, для каждой точки. Если Вы увеличите значение steps, то превращение будет выглядеть более красиво (плавно), но морфинг из одной позиции в другую займет больше времени.

```
VERTEX calculate(int i)    // Вычисление перемещения точек в процессе морфинга
{
    VERTEX a;              // Временная вершина a
                           // a.x равно x Источника - x Назначения делить на Steps
    a.x=(sour->points[i].x-dest->points[i].x)/steps;
                           // a.y равно y Источника - y Назначения делить на Steps
    a.y=(sour->points[i].y-dest->points[i].y)/steps;
```

```
                    // a.z равно z Источника - z Назначения делить на Steps
    a.z=(sour->points[i].z-dest->points[i].z)/steps;
    return a;           // Возвращаем результат
}                       // Возвращаем вычисленные точки
```

Код функции ReSizeGLScene() не изменился, поэтому мы пропускаем его.

```
GLvoid ReSizeGLScene(GLsizei width, GLsizei height) // Изменение размеров и инициализация GL окна
```

В коде ниже мы устанавливаем смешивание для эффекта прозрачности. Это позволит нам создать красиво смотрящиеся следы от перемещающихся точек.

```
int InitGL(GLvoid)                  // Здесь задаются все установки для OpenGL
{
    glBlendFunc(GL_SRC_ALPHA, GL_ONE);    // Устанавливаем функцию смешивания
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f); // Очищаем фон в черный цвет
    glClearDepth(1.0);                  // Очищаем буфер глубины
    glDepthFunc(GL_LESS);               // Устанавливаем тип теста глубины
    glEnable(GL_DEPTH_TEST);            // Разрешаем тест глубины
    glShadeModel(GL_SMOOTH);            // Разрешаем плавное цветовое сглаживание
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // Улучшенные вычисления перспективы
```

Для начала мы устанавливаем переменную maxver в 0. Мы не считывали объекты, поэтому нам неизвестно какое будет максимальное количество вершин.

Затем мы загружаем три объекта. Первый объект – сфера. Данные для сферы находятся в файле sphere.txt. Данные будут загружены в объект, который называется morph1. Также мы загружаем тор и трубку в объекты morph2 и morph3.

```
    maxver=0;                       // Устанавливаем максимум вершин в 0 по умолчанию
    objload("data/sphere.txt", &morph1); // Загружаем первый объект в morph1 из файла sphere.txt
    objload("data/torus.txt", &morph2);  // Загружаем второй объект в morph2 из файла torus.txt
    objload("data/tube.txt", &morph3);   // Загружаем третий объект в morph3 из файла tube.txt
```

Четвертый объект не считывается из файла. Это множество точек, произвольно разбросанных по экрану. Поскольку данные не считываются из файла, постольку мы должны вручную выделить память, используя вызов objallocate(&morph4, 486). Цифра 486 означает то, что мы хотим выделить достаточное количество памяти для хранения 486 вершин (то же количество вершин, из которого состоят остальные три объекта).

После выделения памяти мы создаем цикл, который назначает случайные значения x, y и z каждой вершине. Случайное значение является вещественным числом из интервала от -7 до 7. (14000/1000=14... минус 7 даст нам максимальное значение +7... если случайным числом будет 0, то мы получим минимум 0-7 или -7).

```
objallocate(&morph4, 486);     // Резервируем память для 486 вершин четвертого объекта (morph4)
for(int i=0; i<486; i++)       // Цикл по всем 486 вершинам
{
    // Точка x объекта morph4 принимает случайное значение от -7 до 7
    morph4.points[i].x=((float)(rand()%14000)/1000)-7;
    // Точка y объекта morph4 принимает случайное значение от -7 до 7
    morph4.points[i].y=((float)(rand()%14000)/1000)-7;
    // Точка z объекта morph4 принимает случайное значение от -7 до 7
    morph4.points[i].z=((float)(rand()%14000)/1000)-7;
}
```

Затем мы загружаем sphere.txt во вспомогательный объект helper. Мы никогда не будем модифицировать объектные данные в morph{1/2/3/4} непосредственно. Мы будем модифицировать данные, хранящиеся в helper, для превращения его в одну из 4 фигур. Мы начинаем с отображения morph1 (сфера), поэтому во вспомогательный объект helper мы также поместили сферу.

После того, как все объекты загружены, мы устанавливаем объекты источник и назначение (sour и dest) равными объекту morph1, который является сферой. Таким образом, все будет начинаться со сферы.

```
    // Загружаем sphere.txt в helper (используется как отправная точка)
    objload("data/sphere.txt",&helper);
    sour=dest=&morph1;  // Источник и Направление приравниваются к первому объекту (morph1)
    return TRUE;        // Инициализация прошла успешно
}
```

А теперь позабавимся! Фактический код рисования :)

Начинаем как обычно. Очищаем экран и буфер глубины, сбрасываем матрицу просмотра модели. Затем мы позиционируем объект на экране, используя значения, хранящиеся в переменных cx, cy и cz.

Осуществляем вращение, используя переменные xrot, yrot и zrot.

Углы вращения увеличиваются за счет xpseed, yspeed и zspeed.

Наконец, создаем три временные переменные tx, ty и tz вместе с новой вершиной q.

```
void DrawGLScene(GLvoid)          // Здесь происходит все рисование
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Очищаем экран и буфер глубины
    glLoadIdentity();             // Сбрасываем просмотр
    glTranslatef(cx, cy, cz);     // Сдвигаем текущую позицию рисования
    glRotatef(xrot, 1, 0, 0);     // Вращаем по оси X на xrot
    glRotatef(yrot, 0, 1, 0);     // Вращаем по оси Y на yrot
    glRotatef(zrot, 0, 0, 1);     // Вращаем по оси Z на zrot

    // Увеличиваем xrot, yrot и zrot на xspeed, yspeed и zspeed
    xrot+=xspeed; yrot+=yspeed; zrot+=zspeed;

    GLfloat tx, ty, tz;           // Временные переменные X, Y и Z
    VERTEX q;                     // Хранит вычисленные значения для одной вершины
```

Здесь мы рисуем точки и, если морфинг включен, производим наши вычисления. Команда glBegin(GL_POINTS) говорит OpenGL, что каждая вершина, которую мы определили, будет нарисована на экране как точка.

Мы создаем цикл для прохода по всем вершинам. Вы можете использовать переменную maxver, но, поскольку все объекты состоят из одинакового количества вершин, мы будем использовать morph1.verts.

В цикле мы проверяем, равняется ли значение morph TRUE. Если да, то мы вычисляем перемещение для текущей точки (i). Результат будет помещен в q.x, q.y и q.z. Если нет, то q.x, q.y и q.z будут сброшены в 0 (предотвращение перемещения).

Точки объекта helper перемещаются на основе результатов, полученных из calculate(i) (вспомните, как выше мы вычислили, что точка должна перемещаться по 0.1 единиц, для того, чтобы переместиться из 40 в 20 за 200 шагов).

Мы корректируем значения каждой точки по осям x, y и z, вычитая количество единиц перемещения из объекта helper.

Новая точка объекта helper сохраняется в tx, ty и tz (t{x/y/z}=helper.points[i].{x/y/z}).

```
    glBegin(GL_POINTS);  // Начало рисования точек
        // Цикл по всем вершинам объекта morph1 (все объекты состоят из
        for(int i=0; i<morph1.verts; i++)
        {    // одинакового количества вершин, также можно использовать maxver)
            // Если morph равно True, вычисляем перемещение, иначе перемещение = 0
            if(morph) q=calculate(i); else q.x=q.y=q.z=0;
            // Вычитание q.x единиц из helper.points[i].x (перемещение по оси X)
            helper.points[i].x-=q.x;
            // Вычитание q.y единиц из helper.points[i].y (перемещение по оси Y)
            helper.points[i].y-=q.y;
            // Вычитание q.z единиц из helper.points[i].z (перемещение по оси Z)
            helper.points[i].z-=q.z;
            // Делаем временную переменную X равной вспомогательной X
```

```
            tx=helper.points[i].x;
            // Делаем временную переменную Y равной вспомогательной Y
            ty=helper.points[i].y;
            // Делаем временную переменную Z равной вспомогательной Z
            tz=helper.points[i].z;
```

Теперь, когда вычислена новая позиция, пришло время нарисовать наши точки. Устанавливаем ярко-голубой цвет и рисуем первую точку с помощью glVertex3f(tx, ty, tz). Эта команда нарисует точку на новой вычисленной позиции.

Затем мы делаем цвет более темным и перемещаемся на 2 единицы в вычисленном направлении, вместо одной. Это перемещает точку на новую вычисленную позицию, а затем перемещает ее опять в том же направлении. Таким образом, если она путешествовала влево на 0.1 единиц, то следующая точка окажется на 0.2 единиц. После вычисления 2 позиций вперед, мы рисуем вторую точку.

Наконец, мы устанавливаем темно-синий цвет и вычисляем будущее перемещение. На этот раз, используя наш пример, мы переместимся на 0.4 единиц влево вместо 0.1 или 0.2. Конечным результатом является небольшой хвост, движущийся за перемещающимися точками. С включенным смешиванием это дает очень крутой эффект!

glEnd() говорит OpenGL о том, что мы закончили рисовать точки.

```
            glColor3f(0, 1, 1);               // Установить цвет в ярко голубой
            glVertex3f(tx, ty, tz);           // Нарисовать точку
            glColor3f(0, 0.5f, 1);            // Темный цвет
            tx-=2*q.x; ty-=2*q.y; ty-=2*q.y;  // Вычислить на две позиции вперед
            glVertex3f(tx, ty, tz);           // Нарисовать вторую точку
            glColor3f(0, 0, 1);               // Очень темный цвет
            tx-=2*q.x; ty-=2*q.y; ty-=2*q.y;  // Вычислить еще на две позиции вперед
            glVertex3f(tx, ty, tz);           // Нарисовать третью точку
        }                           // Это создает призрачный хвост, когда точки двигаются
glEnd();                                // Закончим рисовать точки
```

Напоследок, мы проверяем, равняется ли morph значению TRUE и меньше ли step значения переменной steps (200). Если step меньше 200, то мы увеличиваем step на 1.

Если morph равно FALSE или step больше или равно steps (200), то morph устанавливается в FALSE, объект sour (источник) устанавливается равным объекту dest (назначение), и step сбрасывается в 0. Это говорит программе о том, что, либо морфинг не происходит, либо он уже завершился.

```
    // Если делаем морфинг и не прошли все 200 шагов, то увеличим счетчик
    // Иначе сделаем морфинг ложью, присвоим источник назначению и счетчик обратно в ноль
    if(morph && step<=steps) step++; else { morph=FALSE; sour=dest; step=0; }
}
```

Код функции KillGLWindow() не сильно изменился. Единственное существенное отличие заключается в том, что мы освобождаем все объекты из памяти, перед тем как убить окно. Это предотвращает утечку памяти, да и просто является хорошим стилем программирования ;)

```
Lvoid KillGLWindow(GLvoid) // Правильное завершение работы окна
{
    objfree(&morph1);     // Освободим память
    objfree(&morph2);
    objfree(&morph3);
    objfree(&morph4);
    objfree(&helper);
    if (fullscreen)       // Полноэкранный режим?
    {
        ChangeDisplaySettings(NULL, 0);  // Перейти обратно в режим рабочего стола
        ShowCursor(TRUE);                // Показать указатель мыши
    }
    if (hRC)                          // Есть контекст визуализации?
    {
        if (!wglMakeCurrent(NULL, NULL)) // Можем освободим контексты DC и RC?
        {
```

```
            MessageBox(NULL, "Release Of DC And RC Failed.", "SHUTDOWN ERROR", MB_OK |
                       MB_ICONINFORMATION);
        }
        if (!wglDeleteContext(hRC))
        {
            MessageBox(NULL, "Release Rendering Context Failed.", "SHUTDOWN ERROR", MB_OK |
                       MB_ICONINFORMATION);
        }
        hRC=NULL;                       // Установить RC в NULL
    }
    if (hDC && !ReleaseDC(hWnd, hDC))
    {
        MessageBox(NULL, "Release Device Context Failed.", "SHUTDOWN ERROR", MB_OK |
                   MB_ICONINFORMATION);
        hDC=NULL;
    }
    if (hWnd && !DestroyWindow(hWnd))     // Можем удалить окно?
    {
        MessageBox(NULL, "Could Not Release hWnd.", "SHUTDOWN ERROR", MB_OK |
                   MB_ICONINFORMATION);
        hWnd=NULL;                      // Установить hWnd в NULL
    }
    if (!UnregisterClass("OpenGL", hInstance))
    {
        MessageBox(NULL, "Could Not Unregister Class.", "SHUTDOWN ERROR", MB_OK |
                   MB_ICONINFORMATION);
        hInstance=NULL;
    }
}
```

Код функций CreateGLWindow() and WndProc() не изменился. Пропустим его.

```
BOOL CreateGLWindow()                   // Создает GL окно
LRESULT CALLBACK WndProc()              // Дескриптор этого окна
```

Некоторые изменения произошли в функции WinMain(). Во-первых, изменилась надпись в заголовке окна.

```
int WINAPI WinMain( HINSTANCE  hInstance,     // Экземпляр
                    HINSTANCE  hPrevInstance, // Предыдущий экземпляр
                    LPSTR      lpCmdLine,     // Параметры командной строки
                    int        nCmdShow)      // Состояние отображения окна
{
    MSG   msg;                          // Структура сообщений Windows
    BOOL done=FALSE;                    // Переменная для выхода из цикла

    // Спросить пользователя какой он предпочитает режим
    if (MessageBox(NULL,"Would You Like To Run In Fullscreen Mode?", "Start
    FullScreen?",MB_YESNO|MB_ICONQUESTION)==IDNO)
    {
        fullscreen=FALSE;               // Оконный режим
    }

    // Create Our OpenGL Window
    if (!CreateGLWindow(
        "Piotr Cieslak & NeHe's Morphing Points tutorial",640,480,16,fullscreen))
    {
        return 0;                       // Выходим если окно не было создано
    }
    while(!done)                        // Цикл, который продолжается пока done=FALSE
    {
```

```
    if (PeekMessage(&msg,NULL,0,0,PM_REMOVE)) // Есть ожидаемое сообщение?
    {
        if (msg.message==WM_QUIT)      // Мы получили сообщение о выходе?
        {
            done=TRUE;              // Если так done=TRUE
        }
        else                        // Если нет, продолжаем работать с сообщениями окна
        {
            TranslateMessage(&msg);   // Переводим сообщение
            DispatchMessage(&msg);    // Отсылаем сообщение
        }
    }
    else                            // Если сообщений нет
    {
       // Рисуем сцену. Ожидаем нажатия кнопки ESC
       // Активно?  Было получено сообщение о выходе?
        if (active && keys[VK_ESCAPE])
        {
            done=TRUE;              // ESC просигналил "выход"
        }
        else                        // Не время выходить, обновляем экран
        {

            // Нарисовать сцену (Не рисовать, когда неактивно 1% использования CPU)
            DrawGLScene();
            SwapBuffers(hDC);         // Переключаем буферы (Двойная буферизация)
```

Код ниже отслеживает нажатия клавиш. Оставшийся код довольно легок для понимания. Если нажата клавиша Page Up, то мы увеличиваем значение zspeed. Это приводит к тому, что объект начинает вращаться быстрее по оси Z в положительном направлении.

Если нажата клавиша Page Down, то мы уменьшаем значение zspeed. Это приводит к повышению скорости вращения объекта по оси Z в отрицательном направлении.

Если нажата клавиша Стрелка Вниз, то мы увеличиваем значение xspeed. Это приводит к повышению скорости вращения объекта по оси X в положительном направлении.

Если нажата клавиша Стрелка Вверх, то мы уменьшаем значение xspeed. Это приводит к повышению скорости вращения объекта по оси X в отрицательном направлении.

Если нажата клавиша Стрелка Вправо, то мы увеличиваем значение yspeed. Это приводит к повышению скорости вращения объекта по оси Y в положительном направлении.

Если нажата клавиша Стрелка Влево, то мы уменьшаем значение yspeed. Это приводит к повышению скорости вращения объекта по оси Y в отрицательном направлении.

```
    if (keys[VK_PRIOR]) // Page Up нажата?
    zspeed+=0.01f;      // Увеличиваем zspeed

    if (keys[VK_NEXT])  // Page Down нажата?
    zspeed-=0.01f;      // Уменьшаем zspeed

   if (keys[VK_DOWN])  // Стрелка Вниз нажата?
    xspeed+=0.01f;      // Увеличиваем xspeed

    if (keys[VK_UP])    // Стрелка Вверх нажата?
    xspeed-=0.01f;      // Уменьшаем xspeed

    if (keys[VK_RIGHT]) // Стрелка Вправо нажата?
   yspeed+=0.01f ;     // Увеличиваем yspeed

    if (keys[VK_LEFT])  // Стрелка Влево нажата?
    yspeed-=0.01f;      // Уменьшаем yspeed
```

Следующие клавиши физически перемещают объект. 'Q' перемещает его внутрь экрана, 'Z' перемещает его к зрителю, 'W' перемещает объект вверх, 'S' перемещает объект вниз, 'D' перемещает его вправо, и, наконец, 'A' перемещает его влево.

```
if (keys['Q'])      // Клавиша Q нажата и удерживается?
cz-=0.01f;          // Перемещение объекта прочь от зрителя

if (keys['Z'])      // Клавиша Z нажата и удерживается?
cz+=0.01f;          // Перемещение объекта к зрителю

if (keys['W'])      // Клавиша W нажата и удерживается?
cy+=0.01f;          // Перемещение объекта вверх

if (keys['S'])      // Клавиша S нажата и удерживается?
cy-=0.01f;          // Перемещение объекта вниз

if (keys['D'])      // Клавиша D нажата и удерживается?
cx+=0.01f;          // Перемещение объекта вправо

if (keys['A'])      // Клавиша A нажата и удерживается?
cx-=0.01f;          // Перемещение объекта влево
```

Здесь мы отслеживаем нажатие клавиш с 1 по 4. Если нажата клавиша 1, и переменная key не равна 1 (не является текущим объектом), и значение morph равно FALSE (в текущий момент не происходит морфинг), то мы устанавливаем key в 1, тем самым, сообщая нашей программе, что мы только что выбрали объект 1. Затем мы устанавливаем morph в TRUE, позволяя нашей программе начать морфинг, и, наконец, мы делаем объект-назначение (dest) равным объекту 1 (morph1).

Обработка нажатия клавиш 2, 3 и 4 аналогична. Если нажата клавиша 2, то мы делаем dest равной morph2 и устанавливаем key равной 2. Нажатие 3 устанавливает dest в morph3 и key в 3.

Устанавливая значение переменной key в значение только что нажатой нами клавиши, мы предотвращаем попытку пользователя сделать морфинг из сферы в сферу или из тора в тор!

```
if (keys['1'] && (key!=1) && !morph) // Если нажата 1, key не равно 1 и morph равен False
{
    key=1; // Устанавливаем key в 1 (для предотвращения нажатия 1 два раза подряд)
    morph=TRUE; // Устанавливаем morph в True (Начинаем процесс морфинга)
    dest=&morph1; // Устанавливаем объект-назначение в morph1
}

if (keys['2'] && (key!=2) && !morph) // Если нажата 2, key не равно 2 и morph равен False
{
    key=2; // Устанавливаем key в 2 (для предотвращения нажатия 2 два раза подряд)
    morph=TRUE; // Устанавливаем morph в True (Начинаем процесс морфинга)
    dest=&morph2; // Устанавливаем объект-назначение в morph2
}

if (keys['3'] && (key!=3) && !morph) // Если нажата 3, key не равно 3 и morph равен False
{
    key=3; // Устанавливаем key в 3 (для предотвращения нажатия 3 два раза подряд)
    morph=TRUE; // Устанавливаем morph в True (Начинаем процесс морфинга)
    dest=&morph3; // Устанавливаем объект-назначение в morph3
}

if (keys['4'] && (key!=4) && !morph) // Если нажата 4, key не равно 4 и morph равен False
{
    key=4; // Устанавливаем key в 4 (для предотвращения нажатия 4 два раза подряд)
    morph=TRUE; // Устанавливаем morph в True (Начинаем процесс морфинга)
    dest=&morph4; // Устанавливаем объект-назначение в morph4
}
```

Наконец, если нажата клавиша F1, то мы переключаемся из полноэкранного режима в оконный режим или наоборот!

```
          if (keys[VK_F1])       // Нажата клавиша F1?
          {
            keys[VK_F1]=FALSE;  // Если да, то устанавливаем ее в FALSE
            KillGLWindow();     // Убиваем наше текущее окно
            fullscreen=!fullscreen; // Переключаемся в Полноэкранный/Оконный режим
            // Регенерируем наше OpenGL окно
            if (!CreateGLWindow("Piotr Cieslak & NeHe's Morphing Points Tutorial",
                640, 480, 16, fullscreen))
            {
              return 0;        // Выход если окно не было создано
            }
          }
        }
      }
    }
    // Завершение
    KillGLWindow();               // Убиваем окно
    return (msg.wParam);           // Выходим из программы
}
```

Я надеюсь, что Вам понравился этот урок. Это не сложный урок, но Вы можете извлечь из кода много полезного! Анимация в моей демонстрации dolphin осуществляется по правилам, аналогичным изложенным в этом уроке. Играя с этим кодом, Вы можете добиться реально крутых результатов! Превращение точек в слова. Анимация и др. Возможно, что Вы захотите использовать многоугольники или линии вместо точек. Эффект может получиться очень впечатляющим!

Piotr обновил код. Я надеюсь, что после прочтения этого урока Вы стали больше понимать о сохранении и загрузке объектов из файла, о манипулировании данными для достижения крутых GL эффектов в Ваших собственных программах! Написание файла .html этого урока заняло 3 дня. Если Вы обнаружили какие-либо ошибки, пожалуйста, дайте мне знать. Большая часть этого урока создана глубокой ночью, поэтому возможны некоторые ошибки. Я хочу улучшить свои уроки настолько, насколько это возможно. Обратная связь очень важна!

RabidHaMsTeR создал демо "Morph" до того как написал этот урок, в котором лучше показана более усовершенствованная версия этого эффекта. Вы можете проверить это сами на http://homepage.ntlworld.com/fj.williams/PgSoftware.html.

Урок 26. Реалистичное отражение с использование буфера шаблона и отсечения.

Clipping & Reflections Using The Stencil Buffer

Добро пожаловать в следующий урок № 26, довольно интересный урок. Код для него написал Banu Cosmin. Автором, конечно же, являюсь я (NeHe). Здесь вы научитесь создавать исключительно реалистичные отражения безо всяких подделок. Отражаемые объекты не будут выглядеть как бы под полом или по другую сторону стены, нет. Отражение будет настоящим!

Очень важное замечание по теме этого занятия: поскольку Voodoo1, 2 и некоторые другие видеокарты не поддерживают буфер шаблона (stencil buffer, или буфер трафарета или стенсильный буфер), программа на них не заработает. Код, приведенный здесь, работает ТОЛЬКО на картах, поддерживающих буфер шаблона. Если вы не уверены, что ваша видеосистема его поддерживает, скачайте пример и попробуйте его запустить. Для программы не требуется мощный процессор и видеокарта. Даже на моем GeForce1 лишь временами я замечал некоторое замедление. Данная демонстрационная программа лучше всего работает в 32-битном цветовом режиме.

Поскольку видеокарты становятся все лучше, процессоры - быстрее, с моей точки зрения, поддержка буфера шаблона становится все более распространенной. Итак, если оборудование позволяет и вы готовы к отражению, приступим к занятию!

Первая часть программы довольно стандартна. Мы включаем необходимые заголовочные файлы, готовим наш контекст устройства (Device Context), контекст рендеринга (Rendering Context) и т.д.

```
#include <windows.h>   // заголовочный файл для Windows
#include <math.h>      // заголовочный файл для математической библиотеки Windows(добавлено)
#include <stdio.h>     // заголовочный файл для стандартного ввода/вывода(добавлено)
#include <stdarg.h>    // заголовочный файл для манипуляций с переменными аргументами(добавлено)
#include <gl\gl.h>     // заголовочный файл для библиотеки OpenGL32
#include <gl\glu.h>    // заголовочный файл для библиотеки GLu32
#include <gl\glaux.h>  // заголовочный файл для библиотеки GLaux

HDC    hDC=NULL;     // Частный контекст устройства GDI
HGLRC    hRC=NULL;   // Контекст текущей визуализации
HWND    hWnd=NULL;   // Дескриптор нашего окна
HINSTANCE hInstance; // Копия нашего приложения
```

Далее идут стандартные переменные: keys[] – массив для работы с последовательностями нажатий клавиш, active – показывает, активна программа или нет, и индикатор полноэкранного режима – fullscreen.

```
bool        keys[256];        // Массив для работы с клавиатурой
bool        active=TRUE;      // Флаг активности окна, TRUE по умолчанию
bool        fullscreen=TRUE;  // Флаг полноэкранного режима, TRUE по умолчанию
```

Далее мы настроим параметры нашего освещения. В LightAmb[] поместим настройки окружающего освещения. Возьмем его составляющие в пропорции 70% красного, 70% синего и 70% зеленого цвета, что даст нам белый свет с яркостью в 70%. LightDif[]-массив с настройками рассеянного освещения (это составляющая света, равномерно отражаемая поверхностью нашего объекта). В данном случае нам потребуется освещение максимальной интенсивности для наилучшего отражения. И массив LightPos[] используется для размещения источника освещения. Сместим его на 4 единицы вправо, 4 единицы вверх и на 6 единиц к наблюдателю. Для более актуального освещения источник размещается на переднем плане правого верхнего угла экрана.

```
//Параметры освещения
static Glfloat LightAmb[]={0.7f, 0.7f, 0.7f};  //Окружающий свет
static Glfloat LightDif[]={1.0f, 1.0f, 1.0f};  //Рассеянный свет
//Позиция источника освещения
static Glfloat LightPos[]={4.0f, 4.0f, 6.0f, 1.0f};
```

Настроим переменную q для нашего квадратичного объекта (квадратичным объект, по-видимому, называется по той причине, что его полигонами являются прямоугольники – прим.перев), xrot и yrot – для осуществления вращения. xrotspeed и yrotspeed – для управления скоростью вращения. Переменная zoom используется для приближения и отдаления нашей сцены (начальное значение = -7, при котором мы увидим полную сцену), и переменная height содержит значение высоты мяча над полом.

Затем выделим массив для трех текстур и определим WndProc().

```
GLUquadricObj  *q;     // Квадратичный объект для рисования сферы мяча
GLfloat xrot = 0.0f;   // Вращение по X
GLfloat yrot = 0.0f;   // Вращение по Y
GLfloat xrotspeed = 0.0f;// Скорость вращения по X
GLfloat yrotspeed = 0.0f;// Скорость вращения по Y
GLfloat zoom = -7.0f;  // Глубина сцены в экране
GLfloat height = 2.0f; // Высота мяча над полом
GLuint  texture[3];    // 3 Текстуры

    // Объявление WndProc
LRESULT  CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
```

Функции ReSizeGLScene() и LoadBMP() не меняются, так что я их обе пропускаю.

```
// Функция изменения размера и инициализации OpenGL-окна
GLvoid ReSizeGLScene(GLsizei width, GLsizei height)
// Функция загрузки растрового рисунка
AUX_RGBImageRec *LoadBMP(char *Filename)
```

Код, загружающий текстуру довольно стандартный. Вы могли пользоваться им не раз при изучении предыдущих статей. Мы создали массив для трех текстур, затем мы загружаем три рисунка и создаем три текстуры с линеарной фильтрацией из данных рисунков. Файлы с растровыми рисунками мы ищем в каталоге DATA.

```
int LoadGLTextures()    // Загрузка рисунков и создание текстур
{
   int Status=FALSE;                    // Индикатор статуса
   AUX_RGBImageRec *TextureImage[3];       // массив для текстур
   memset(TextureImage,0,sizeof(void *)*3); // Обнуление
   if ((TextureImage[0]=LoadBMP("Data/EnvWall.bmp")) && // Текстура пола
   (TextureImage[1]=LoadBMP("Data/Ball.bmp")) &&       // Текстура света
   (TextureImage[2]=LoadBMP("Data/EnvRoll.bmp")))      // Текстура стены
   {
    Status=TRUE;                    // Статус положительный
    glGenTextures(3, &texture[0]);    // Создание текстуры
    for (int loop=0; loop<3; loop++)  // Цикл для 3 текстур
    {
     glBindTexture(GL_TEXTURE_2D, texture[loop]);
     glTexImage2D(GL_TEXTURE_2D, 0, 3, TextureImage[loop]->sizeX,
       TextureImage[loop]->sizeY, 0, GL_RGB, GL_UNSIGNED_BYTE,
       TextureImage[loop]->data);
     glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
     glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
    }
    for (loop=0; loop<3; loop++)      // Цикл для 3 рисунков
    {
     if (TextureImage[loop])        // Если текстура существует
     {
      if (TextureImage[loop]->data) // Если рисунок есть
      {
       free(TextureImage[loop]->data);
      }
      free(TextureImage[loop]);     // Очистить память из-под него
     }
    }
   }
   return Status;                   // Вернуть статус
}
```

В функции инициализации представлена новая команда OpenGL – glClearStencil. Значение параметра = 0 для этой команды говорит о том, что очищать буфер шаблона не надо. С остальной частью функции вы уже, должно быть, знакомы. Мы загружаем наши текстуры и включаем плавное закрашивание. Цвет очистки экрана - синий, значение очистки буфера глубины = 1.0f. Значение очистки буфера шаблона = 0. Мы включаем проверку глубины и устанавливаем значение проверки глубины меньше или равной установленному значению. Коррекция перспективы выбрана наилучшего качества и включается 2D-текстурирование.

```
int InitGL(GLvoid)        // Инициализация OpenGL
{
 if (!LoadGLTextures())  // Если текстуры не загружены, выход
 {
  return FALSE;
 }
 glShadeModel(GL_SMOOTH);//Включаем плавное закрашивание
 glClearColor(0.2f, 0.5f, 1.0f, 1.0f);// Фон
 glClearDepth(1.0f);    // Значение для буфера глубины
 glClearStencil(0);     // Очистка буфера шаблона 0
 glEnable(GL_DEPTH_TEST);//Включить проверку глубины
 glDepthFunc(GL_LEQUAL); // Тип проверки глубины
 // Наилучшая коррекция перспективы
 glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
 glEnable(GL_TEXTURE_2D);//Включить рисование 2D-текстур
```

Теперь пора настроить источник света GL_LIGHT0. Первая нижеприведенная строка говорит OpenGL использовать массив LightAmp для окружающего света. Если вы помните начало программы, RGB-компоненты в этом массиве были все равны 0.7f , что означает 70% интенсивности белого света. Затем мы при помощи массива LightDif настраиваем рассеянный свет, и местоположение источника света – значениями x,y,z из массива LightPos.

После настройки света мы включаем его командой glEnable(GL_LIGHT0). Хотя источник включен, вы его не увидите, пока не включите освещение командой glEnable(GL_LIGHTING).

Примечание: если вы хотите отключить все источники света в сцене, примените команду glDisable(GL_LIGHTING). Для отключения определенного источника света надо использовать команду glEnable(GL_LIGHT{0-7}). Эти команды позволяют нам контролировать освещение в целом, а также источники по отдельности. Еще раз запомните, пока не отработает команда glEnable(GL_LIGHTING), ничего вы на своей 3D-сцене не увидите.

```
  // Фоновое освещение для источника LIGHT0
  glLightfv(GL_LIGHT0, GL_AMBIENT, LightAmb);
  // Рассеянное освещение для источника LIGHT0
  glLightfv(GL_LIGHT0, GL_DIFFUSE, LightDif);
  // Положение источника LIGHT0
  glLightfv(GL_LIGHT0, GL_POSITION, LightPos);
  // Включить Light0
  glEnable(GL_LIGHT0);
  // Включить освещение
  glEnable(GL_LIGHTING);
```

В первой строке мы создаем новый квадратичный объект. Затем мы говорим OpenGL о типе генерируемых нормалей для нашего квадратичного объекта - нормали сглаживания. Третья строка включает генерацию координат текстуры для квадратичного объекта. Без этих строк – второй и третьей закрашивание объекта будет плоским и невозможно будет наложить на него текстуру.

Четвертая и пятая строки говорят OpenGL использовать алгоритм сферического наложения (Sphere Mapping) для генерации координат для текстуры. Это дает нам доступ к сферической поверхности квадратичного объекта.

```
  q = gluNewQuadric();  // Создать квадратичный объект
  // тип генерируемых нормалей для него – «сглаженные»
  gluQuadricNormals(q, GL_SMOOTH);
  // Включить текстурные координаты для объекта
  gluQuadricTexture(q, GL_TRUE);
  // Настройка сферического наложения
  glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
  // Настройка отображения сферы
  glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
  return TRUE;  // Инициализация прошла успешно
}
```

Нижеприведенная функция рисует наш объект (в самом деле, неплохо смотрящийся пляжный мяч).

Устанавливаем цвет с максимальной интенсивностью белого и подключаем текстуру мяча (состоящую из последовательности красной, белой и синей полос).

После этого мы рисуем квадратичную сферу (Quadratic Sphere) с радиусом в 0.35f, 32 срезами (разбиениями вокруг оси Z) и 16 полосами (разбиениями вдоль оси Z) (вверх и вниз).

```
void DrawObject()  // Рисование мяча
{
  glColor3f(1.0f, 1.0f, 1.0f);// Цвет - белый
  glBindTexture(GL_TEXTURE_2D, texture[1]);// Выбор текстуры 2 (1)
  gluSphere(q, 0.35f, 32, 16);// Рисование первого мяча
```

Нарисовав первый мяч, выбираем новую текстуру (EnvRoll), устанавливаем значение прозрачности в 40% и разрешаем смешивание цветов, основанное на значении альфа (прозрачности). Команды glEnable(GL_TEXTURE_GEN_S) и glEnable(GL_TEXTURE_GEN_T) разрешают сферическое наложение.

После всего этого мы перерисовываем сферу, отключаем сферическое наложение и отключаем смешивание.

Конечный результат – это отражение, которое выглядит как блики на пляжном мяче. Так как мы включили сферическое наложение, текстура всегда повернута к зрителю, даже если мяч вращается. Поэтому мы применили смешивание, так что новая текстура не замещает старую (одна из форм мультитекстурирования).

```
 glBindTexture(GL_TEXTURE_2D, texture[2]);// Выбор текстуры 3 (2)
 glColor4f(1.0f, 1.0f, 1.0f, 0.4f);// Белый цвет с 40%-й прозрачностью
 glEnable(GL_BLEND);              // Включить смешивание
 // Режим смешивания
 glBlendFunc(GL_SRC_ALPHA, GL_ONE);
 // Разрешить сферическое наложение
 glEnable(GL_TEXTURE_GEN_S);
 // Разрешить сферическое наложение
 glEnable(GL_TEXTURE_GEN_T);
 // Нарисовать новую сферу при помощи новой текстуры
 gluSphere(q, 0.35f, 32, 16);
 // Текстура будет смешена с созданной для эффекта мультитекстурирования (Отражение)
 glDisable(GL_TEXTURE_GEN_S);      // Запретить сферическое наложение
 glDisable(GL_TEXTURE_GEN_T);      // Запретить сферическое наложение
 glDisable(GL_BLEND);              // Запретить смешивание
}
```

Следующая функция рисует пол, над которым парит наш мяч. Мы выбираем текстуру пола (EnvWall), и рисуем один текстурированный прямоугольник, расположенный вдоль оси Z.

```
void DrawFloor()    // Рисование пола
{
 glBindTexture(GL_TEXTURE_2D, texture[0]);// текстура 1 (0)
 glBegin(GL_QUADS);           // Начало рисования
   glNormal3f(0.0, 1.0, 0.0); // «Верхняя» нормаль
   glTexCoord2f(0.0f, 1.0f);  // Нижняя левая сторона текстуры
   glVertex3f(-2.0, 0.0, 2.0);//Нижний левый угол пола
   glTexCoord2f(0.0f, 0.0f);  //Верхняя левая сторона текстуры
   glVertex3f(-2.0, 0.0,-2.0);//Верхний левый угол пола
   glTexCoord2f(1.0f, 0.0f);  //Верхняя правая сторона текстуры
   glVertex3f( 2.0, 0.0,-2.0);//Верхний правый угол пола
   glTexCoord2f(1.0f, 1.0f);  //Нижняя правая сторона текстуры
   glVertex3f( 2.0, 0.0, 2.0);//Нижний правый угол пола
 glEnd();                     // конец рисования
}
```

Теперь одна забавная вещь. Здесь мы скомбинируем все наши объекты и изображения для создания сцены с отражениями.

Начинаем мы с того, что очищаем экран (GL_COLOR_BUFFER_BIT) синим цветом (задан ранее в программе). Также очищаются буфер глубины (GL_DEPTH_BUFFER_BIT) и буфер шаблона (GL_STENCIL_BUFFER_BIT). Убедитесь в том, что вы включили команду очистки буфера шаблона, так как это новая команда для вас и ее легко пропустить. Важно отметить, что при очистке буфера шаблона мы заполняем его нулевыми значениями.

После очистки экрана и буферов мы определяем наше уравнение для плоскости отсечения. Плоскость отсечения нужна для отсечения отражаемого изображения.

Выражение eqr[]={0.0f,-1.0f, 0.0f, 0.0f} будет использовано, когда мы будем рисовать отраженное изображение. Как вы можете видеть, Y-компонента имеет отрицательное значение. Это значит, что мы увидим пиксели рисунка, если они появятся ниже пола, то есть с отрицательным значением по Y-оси. Любой другой графический вывод выше пола не будет отображаться, пока действует это уравнение.

Больше об отсечении будет позже… читайте дальше.

```
int DrawGLScene(GLvoid)// Рисование сцены
{
  // Очистка экрана, буфера глубины и буфера шаблона
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
  // Уравнение плоскости отсечения для отсечения отраженных объектов
  double eqr[] = {0.0f,-1.0f, 0.0f, 0.0f};
```

Итак, мы очистили экран и определили плоскости отсечения. Теперь перейдем к изюминке нашего руководства!

Сначала сбросим матрицу модели. После чего все процессы рисования будут начинаться из центра экрана. Затем мы перемещаемся вниз на 0.6f единиц (для наклонной перспективы пола) и в экран на значение zoom. Для лучшего понимания, для чего мы перемещаемся вниз на 6.0f единиц, я приведу вам простой пример. Если вы смотрите на лист бумаги на уровне своих глаз, вы его едва видите – так он скорее похож на тонкую полоску. Если вы переместите лист немного вниз, он перестанет быть похожим на линию. Вы увидите большую площадь бумаги, так как ваши глаза будут обращены вниз на лист вместо прямого взгляда на его ребро.

```
  glLoadIdentity();// Сброс матрицы модели
  // Отдаление и подъем камеры над полом (на 0.6 единиц)
  glTranslatef(0.0f, -0.6f, zoom);
```

Далее мы настроим маску цвета – новую вещь в этом руководстве. Маска представлена 4 значениями : красный, зеленый, синий и альфа-значение(прозрачность). По умолчанию все значения устанавливаются в GL_TRUE.

Если бы значение красной компоненты в команде glColorMask({red},{green},{blue},{alpha}) было установлено в GL_TRUE и в то же самое время все другие значения были равны 0 (GL_FALSE), единственный цвет, который бы мы увидели на экране, это красный. Соответственно, если бы ситуация была обратной (красная компонента равна GL_FALSE, а все остальные равны GL_TRUE), то на экране мы бы увидели все цвета за исключением красного.

Нам не нужно что-либо выводить на экран в данный момент, поэтому установим все четыре значения в 0.

```
  glColorMask(0,0,0,0);    // Установить маску цвета
```

Именно сейчас пойдет речь об изюминке урока… Настроим буфер шаблона и проверку шаблона.

Сначала включается проверка шаблона. После того, как была включена проверка шаблона, мы можем изменять буфер шаблона.

Немного сложно объяснить работу команд, приводимых ниже, так что, пожалуйста, потерпите, а если у вас есть лучшее объяснение, пожалуйста, дайте мне знать. Строка glStencilFunc(GL_ALWAYS, 1, 1) сообщает OpenGL тип проверки, производимой для каждого пикселя выводимого на экран объекта (если пиксель не выводиться, то нет и проверки - прим.перев.).

Слово GL_ALWAYS говорит OpenGL, что тест работает все время. Второй параметр – это значение ссылки, которое мы проверяем в третьей строке, и третий параметр – это маска. Маска – это значение, поразрядно умножаемое операцией AND (логическое умножение из булевой алгебры – прим.перев.) на значение ссылки и сохраняемое в буфере шаблона в конце обработки. Значение маски равно 1, и значение ссылки тоже равно 1. Так что, если мы передадим OpenGL эти параметры, и тест пройдет успешно, то в буфере шаблона сохранится единица (так как 1(ссылка)&1(маска)=1).

Небольшое разъяснение: тест шаблона – это попиксельная проверка изображения объектов, выводимых на экран во время работы теста. Значение ссылки, обработанное операцией логического умножения AND на значение маски, сравнивается с текущим значением в буфере шаблона в соответствующем пикселе, также обработанным операцией AND на значение маски.

Третья строка проверяет три различных состояния, основываясь на функции проверки шаблона, которую мы решили использовать. Первые два параметра – GL_KEEP, а третий -GL_REPLACE.

Первый параметр говорит OpenGL что делать в случае если тест не прошел. Так как этот параметр у нас установлен в GL_KEEP, в случае неудачного завершения теста (что не может случится, так как у нас вид функции установлен в GL_ALWAYS), мы оставим состояние буфера в том виде, в котором оно было на это время.

Второй параметр сообщает OpenGL о том, что делать, если тест шаблона прошел успешно, а тест глубины – нет. Далее в программе мы так и так отключаем тест глубины, поэтому этот параметр может быть игнорирован.

И, наиболее важный, третий параметр. Он сообщает OpenGL о том, что надо делать в случае, если весь тест пройден успешно. В приведенном ниже участке программы мы говорим OpenGL, что надо заместить (GL_REPLACE) значение в буфере шаблона. Значение, которое мы помещаем в буфер шаблона, является результатом логического умножения (операция AND) нашего значения ссылки и маски, значение которой равно 1.

Следующее, что нам надо сделать после указания типа теста, это отключить тест глубины и перейти к рисованию пола.

А теперь расскажу обо всем этом попроще.

Мы указываем OpenGL, что не надо ничего не отображать на экране. Значит, во время рисования пола мы ничего не должны видеть на экране. При этом любая точка на экране в том месте, где должен нарисоваться наш объект (пол) будет проверена выбранным нами тестом шаблона. Сначала буфер шаблона пуст (нулевые значения). Нам требуется, чтобы в том, месте, где должен был бы появиться наш объект (пол), значение шаблона было равной единице. При этом нам незачем самим заботиться о проверке. В том месте, где пиксель должен был бы нарисоваться, буфер шаблона помечается единицей. Значение GL_ALWAYS это обеспечивает. Это также гарантируют значения ссылки и маски, установленные в 1. Во время данного процесса рисования на экран ничего не выводится, а функция шаблона проверяет каждый пиксель и устанавливает в нужном месте 1 вместо 0.

```
    // Использовать буфер шаблона для «пометки» пола
glEnable(GL_STENCIL_TEST);
// Всегда проходит, 1 битовая плоскость, маска = 1
glStencilFunc(GL_ALWAYS, 1, 1); // 1, где рисуется хоть какой-нибудь полигон
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);
glDisable(GL_DEPTH_TEST);// Отключить проверку глубины
DrawFloor();// Рисование пола (только в буфере шаблона)
```

Теперь у нас есть невидимая шаблонная маска пола. Пока действует проверка шаблона, пиксели будут появляться, только в тех местах, где в буфере шаблона будет установлена 1. И он у нас устанавливается в 1 в том месте, где выводился невидимый пол. Это значит, что мы увидим рисунок лишь в том месте, где невидимый пол установил 1 в буфер шаблона. Этот трюк заставляет появиться отражение лишь на полу и нигде более!

Итак, теперь мы уверены, что отражение мяча нарисуется только на полу. Значит, пора рисовать само отражение! Включаем проверку глубины и отображение всех трех составляющих цвета.

Взамен использования значения GL_ALWAYS в выборе шаблонной функции мы станем использовать значение GL_EQUAL. Значение ссылки и маски оставим равными 1. Для всех операций с шаблоном установим все параметры в GL_KEEP. Проще говоря, теперь любой объект может быть отображен на экране (поскольку цветовая маска установлена в истину для каждого цвета). Во время работы проверки шаблона, выводимые пиксели будут отображаться лишь в том месте, где буфер шаблона установлен в 1 (значение ссылки AND значение маски (1&1) равно 1, что эквивалентно (GL_EQUAL) значению буфера шаблона AND значение маски, что также равно 1). Если в том месте, где рисуется пиксель, буфер шаблона не равен 1, пиксель не отобразится. Значение GL_KEEP запрещает модифицировать буфер шаблона вне зависимости от результата проверки шаблона.

```
glEnable(GL_DEPTH_TEST); // Включить проверку глубины
glColorMask(1,1,1,1); // Маска цвета = TRUE, TRUE, TRUE, TRUE
glStencilFunc(GL_EQUAL, 1, 1); // Рисуем по шаблону (где шаблон=1)
// (то есть в том месте, где был нарисован пол)
// Не изменять буфер шаблона
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
```

Теперь подключим плоскость отсечения для отражения. Эта плоскость задается массивом eqr, и разрешает рисовать только те объекты, которые выводятся в пространстве от центра экрана (где находится наш пол) и ниже. Это способ для того, чтобы не дать отражению мяча появиться выше центра пола. Будет некрасиво, если он это сделает. Если вы еще не поняли, что я имею ввиду, уберите первую строку в приведенном ниже коде и переместите клавишами исходный мяч (не отраженный) через пол.

После подключения плоскости отсечения plane0(обычно применяют от 0 до 5 плоскостей отсечения), мы определяем ее, передав параметры из массива eqr.

Сохраняем матрицу (относительно ее позиционируются все объекты на экране) и применяем команду glScalef(1.0f,-1.0f,1.0f) для поворота всех вещей сверху вниз (придавая отражению реальный вид). Негативное значение для Y-параметра в этой команде заставляет OpenGL рисовать все в положении с обратной координатой Y (то есть, «вниз головой» - прим.перев.). Это похоже на переворачивание картинки сверху вниз. Объект с положительным значением по оси Y появляется внизу экрана, а не вверху. Если вы поворачиваете объект к себе, он поворачивается от вас (словом, представьте себя Алисой – прим.перев.). Любая вещь будет перевернута, пока не будет восстановлена матрица или не отработает та же команда с положительным значением Y-параметра (1) (glScalef({x},1.0f,{z}).

```
glEnable(GL_CLIP_PLANE0);// Включить плоскость отсечения для удаления
// артефактов(когда объект пересекает пол)
glClipPlane(GL_CLIP_PLANE0, eqr);// Уравнение для отраженных объектов
glPushMatrix();  // Сохранить матрицу в стеке
glScalef(1.0f, -1.0f, 1.0f);  // Перевернуть ось Y
```

Первая нижеприведенная строка перемещает наш источник света в позицию, заданную в массиве LightPos. Источник света должен освещать правую нижнюю часть отраженного мяча, имитируя почти реальный источник света. Позиция источника света также перевернута. При рисовании «настоящего» мяча (мяч над полом) свет освещает правую верхнюю часть экрана и создает блик на правой верхней стороне этого мяча. При рисовании отраженного мяча источник света будет расположен в правой нижней стороне экрана.

Затем мы перемещаемся вниз или вверх по оси Y на значение, определенное переменной height. Перемещение также переворачивается, так что если значение height =0.5f, позиция перемещения превратится в -5.0f. Мяч появится под полом вместо того, чтобы появиться над полом!

После перемещения нашего отраженного мяча, нам нужно повернуть его по осям X и Y на значения xrot и yrot соответственно. Запомните, что любые вращения по оси X также переворачиваются. Так, если верхний мяч поворачивается к вам по оси X, то отраженный мяч поворачивается от вас.

После перемещения и вращения мяча нарисуем его функцией DrawObject(), и восстановим матрицу из стека матриц, для восстановления ее состояния на момент до рисования мяча. Восстановленная матрица прекратит отражения по оси Y.

Затем отключаем плоскость отсечения (plan0), так как нам больше не надо ограничивать рисование нижней половиной экрана, и отключаем шаблонную проверку, так что теперь мы можем рисовать не только в тех точках экрана, где должен быть пол.

Заметьте, что мы рисуем отраженный мяч раньше пола.

```
  // Настройка источника света Light0
  glLightfv(GL_LIGHT0, GL_POSITION, LightPos);
  glTranslatef(0.0f, height, 0.0f);// Перемещение объекта
  // Вращение локальной координатной системы по X-оси
  glRotatef(xrot, 1.0f, 0.0f, 0.0f);
  // Вращение локальной координатной системы по Y-оси
  glRotatef(yrot, 0.0f, 1.0f, 0.0f);
  DrawObject();// Рисование мяча (для отражения)
glPopMatrix();    // Восстановить матрицу
glDisable(GL_CLIP_PLANE0);// Отключить плоскость отсечения
// Отключение проверки шаблона
glDisable(GL_STENCIL_TEST);
```

Начнем эту секцию с позиционирования источника света. Так как ось Y больше не перевернута, свет будет освещать верхнюю часть экрана, а не нижнюю.
Включаем смешивание цветов, отключаем освещение и устанавливаем компоненту прозрачности в 80% в команде glColor4f(1.0f,1.0f,1.0f,0.8f). Режим смешивания настраивается командой glBlendFunc(), и полупрозрачный пол рисуется поверх отраженного мяча.

Если бы мы сначала нарисовали пол, а затем – мяч (как нам подсказывает логика – прим.перев.), результат выглядел бы не очень хорошо. Нарисовав мяч, а затем – пол, вы увидите небольшой участок пола, смешанный с рисунком мяча. Когда я посмотрю в синее зеркало, я предположу, что отражение будет немного синим.

Нарисовав сначала пол, последующим отображением пола мы придадим отраженному изображению мяча легкую окраску пола.

```
  glLightfv(GL_LIGHT0, GL_POSITION, LightPos);// Положение источника
  // Включить смешивание (иначе не отразится мяч)
  glEnable(GL_BLEND);
  // В течение использования смешивания отключаем освещение
  glDisable(GL_LIGHTING);
  // Цвет белый, 80% прозрачности
  glColor4f(1.0f, 1.0f, 1.0f, 0.8f);
  // Смешивание, основанное на «Source Alpha And 1 Minus Dest Alpha»
  glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
  DrawFloor();// Нарисовать пол
```

Теперь нарисуем «настоящий» мяч (парящий над полом). При рисовании пола освещение было отключено, но теперь мы опять его включим.

Так как смешивание нам более не потребуется, мы его отключаем. Если мы этого не сделаем, изображение мяча смешается с изображением пола. Нам не нужно, чтобы мяч выглядел, как его отражение, поэтому мы и отключаем смешивание цветов.

Мы не будем отсекать «настоящий» мяч. Если мяч будет проходить через пол, мы должны видеть его выходящим из пола снизу. Если мы будем использовать отсечение, мяч снизу пола не появится. При возникновении необходимости запретить мячу появляться снизу пола вы можете применить значение плоскости отсечения, где будет указано положительное значение Y-координаты. При этом мяч будет виден, только когда он будет рисоваться в верхней части экрана, до той Y-координаты, которую вы укажете в выражении плоскости отсечения. В данном демонстрационном примере у нас нет необходимости этого делать, поэтому мяч будет виден по обе стороны пола.

Затем мы перемещаемся на позицию вывода, заданную в переменной heght. Только теперь ось Y не перевернута, поэтому мяч двигается в направлении, противоположном направлению движения отраженного мяча.

Мяч вращается, и, опять же, поскольку ось Y на данный момент не перевернута, мяч будет вращаться в направлении, обратном направлению вращения отраженного мяча. Если отраженный мяч вращается к вам, «реальный» мяч вращается от вас. Это дополняет иллюзию отражения.

После перемещения и поворота мы рисуем мяч.

```
  glEnable(GL_LIGHTING);// Включить освещение
  glDisable(GL_BLEND);  // Отключить смешивание
  glTranslatef(0.0f, height, 0.0f);// Перемещение мяча
  glRotatef(xrot, 1.0f, 0.0f, 0.0f);// Поворот по оси X
  glRotatef(yrot, 0.0f, 1.0f, 0.0f);// Поворот по оси Y
  DrawObject();         // Рисование объекта
```

Следующий код служит для поворота мяча по осям X и Y. Для поворота по оси X увеличивается переменная xrot на значение переменной xrotspeed. Для поворота по оси Y увеличивается переменная yrot на значение переменной yrotspeed. Если xrotspeed имеет слишком большое позитивное или негативное значение, мяч будет крутиться быстрее, чем, если бы xrotspeed было близко к нулю. То же касается и yrotspeed. Чем больше yrotspeed, тем быстрее мяч крутится по оси Y.

Перед тем, как вернуть TRUE, выполняется команда glFlush(). Эта команда указывает OpenGL выполнить все команды, переданные ему в конвейер, что помогает предотвратить мерцание на медленных видеокартах.

```
  xrot += xrotspeed; // Обновить угол вращения по X
  yrot += yrotspeed; // Обновить угол вращения по Y
  glFlush();   // Сброс конвейера OpenGL
  return TRUE; // Нормальное завершение
}
```

Следующий код обрабатывает нажатия клавиш. Первые 4 строки проверяют нажатие вами 4 клавиш (для вращения мяча вправо, влево, вниз, вверх).

Следующие 2 строки проверяют нажатие вами клавиш ‘A’ или ‘Z’. Клавиша ‘A’ предназначена для приближения сцены, клавиша ‘Z’ – для отдаления.

Клавиши ‘PAGE UP’ и ’ PAGE UP’ предназначены для вертикального перемещения мяча.

```
void ProcessKeyboard()  // Обработка клавиатуры
{
  if (keys[VK_RIGHT])  yrotspeed += 0.08f;// Вправо
  if (keys[VK_LEFT])  yrotspeed -= 0.08f; // Влево
  if (keys[VK_DOWN])  xrotspeed += 0.08f; // Вверх
  if (keys[VK_UP])  xrotspeed -= 0.08f;   // Вниз
  if (keys['A'])    zoom +=0.05f; // Приближение
  if (keys['Z'])    zoom -=0.05f; // Отдаление
  if (keys[VK_PRIOR])  height +=0.03f; // Подъем
  if (keys[VK_NEXT])  height -=0.03f;  // Спуск
}
```

Функция KillGLWindow() не меняется, поэтому пропущена.

```
GLvoid KillGLWindow(GLvoid)// Удаление окна
```

Также можно оставить и следующую функцию - CreateGLWindow(). Для большей уверенности я включил ее полностью, даже если поменялась всего одна строка в этой структуре:

```
  static  PIXELFORMATDESCRIPTOR pfd=
  // pfd говорит Windows о наших запросах для формата пикселя
  {
    sizeof(PIXELFORMATDESCRIPTOR),  // Размер структуры
    1,                  // Номер версии
    PFD_DRAW_TO_WINDOW |  // Формат должен поддерживать Window
    PFD_SUPPORT_OPENGL |  // Формат должен поддерживать OpenGL
    PFD_DOUBLEBUFFER,     // Нужна двойная буферизация
    PFD_TYPE_RGBA,        // Формат данных- RGBA
    bits,                 // Глубина цвета
    0, 0, 0, 0, 0, 0,     // Игнорируются биты цвета
    0,                  // Нет альфа-буфера
    0,                  // Игнорируется смещение бит
    0,                  // Нет аккумулирующего буфера
    0, 0, 0, 0,           // Игнорируются биты аккумуляции
    16,                 // 16-битный Z-буфер (глубины)
```

Только одно изменение в этой функции – в приведенной ниже строке. ОЧЕНЬ ВАЖНО: вы меняете значение с 0 на 1 или любое другое ненулевое значение. Во всех предыдущих уроках значение в строке ниже было равным 0. Для использования буфера шаблона это значение должно быть больше либо равным 1. Оно обозначает количество битовых планов буфера шаблона.

```
    1,                  // Использовать буфер шаблона (* ВАЖНО *)
    0,                  // Нет вспомогательного буфера
    PFD_MAIN_PLANE,       // Основной уровень рисования
    0,                  // Не используются
    0, 0, 0  ,            // Нет маски уровня
  };
```

WndProc() не изменилась, поэтому здесь не приводится.

Здесь тоже ничего нового. Типичный запуск WinMain().

Меняется только заголовок окна, в котором содержится информация о названии урока и его авторах. Обратите внимание, что вместо обычных параметров экрана 640, 480, 16 в команду создания окна передаются переменные resx, resy и resbpp соответственно.

```
 // Создание окна в Windows
 if (!CreateGLWindow("Banu Octavian & NeHe's Stencil & Reflection Tutorial",
     resx, resy, resbpp, fullscreen))
{    return 0;                // Выход, если не создалось
}
  while(!done)// Цикл, пока done=FALSE
{                        // Выборка сообщений
  if (PeekMessage(&msg,NULL,0,0,PM_REMOVE))
  {
    if (msg.message==WM_QUIT) // Выход?
    {       done=TRUE;    // Если да
    }
    else  // Если нет, обработка сообщения
    {
      TranslateMessage(&msg); // Транслировать сообщение
      DispatchMessage(&msg);
    }
  }    else          // Если нет сообщений
  {  // Отрисовка сцены.  Ожидание клавиши ESC или
              // сообщения о выходе от DrawGLScene()
    if (active)  // Программа активна?
    {
      if (keys[VK_ESCAPE])        // ESC нажата?
      {       done=TRUE;            // Если да, выход
      }
          else                  // Иначе - рисование
      {
        DrawGLScene();// Рисование сцены
        SwapBuffers(hDC);//Переключить буфера
```

Вместо проверки нажатия клавиш в WinMain(), мы переходим к нашей функции обработки клавиатуры ProcessKeyboard(). Запомните, что эта функция вызывается, только если программа активна!

```
        ProcessKeyboard();// Обработка нажатий клавиш
      }
    }
  }
 }    // Конец работы
 KillGLWindow();  // Удалить окно
 return (msg.wParam);// Выход из программы
}
```

Я надеюсь, что вам понравилось это руководство. Понимаю, что оно задаст вам немного работы. Это было одно из самых трудных руководств, написанных мною. Оно несложно для моего понимания того, что оно делает, и какие команды используются для создания эффектов, но когда вы попытаетесь объяснить это, понимая, что большинство программистов даже не слышали о буфере шаблона, это трудно.

Если вам что-то показалось лишним в программе или если вы обнаружили какие-либо ошибки в руководстве или программе, пожалуйста, дайте мне знать. Как обычно, я пытался сделать данное руководство наилучшим, насколько смог, и ваши отзывы будут наиболее ценны.

Урок 27. Тени

Shadows

Добро пожаловать на довольно сложный урок, посвященный отбрасыванию теней. Вы увидите невероятный эффект, который создает программа этого урока. Тени, которые вытягиваются, сгибаются и окутывают объекты и падают на стенки. Можно перемещаться по сцене в 3D, используя клавиатуру.
На этом уроке предполагается, что Вы имеет уже достаточно много знаний об OpenGL. Вы должны знать, как работает буфер трафарета, и основы инициализации OpenGL. Иначе освежите свои знания и почитайте более ранние уроки. Функции типа CreateGLWindow и WinMain не будут объясняться в этом уроке. Дополнительно, необходимо знать азы 3D, поэтому возьмите в руки хороший учебник по трехмерной графике! (Я использовал мои лекции по математике с первого курса университета - я знал, что они мне пригодятся! :)
Сначала мы определим значение БЕСКОНЕЧНОСТИ, которое задает, как далеко, простираются теневые объемы от многоугольников (это объясним позже). Если Вы используете большую или меньшую систему координат, скорректируете это значение соответственно.

```
// определим Of "БЕСКОНЕЧНОСТЬ" для вычисления вектора расширения теневого объема

#define INFINITY  100
```

Затем - определение структур объектов.

Структура Point3f содержит значения координат точки в 3D пространстве. Она может использоваться для вершин или векторов.

```
// Структура описания вершины в объекте
struct Point3f
{
  GLfloat x, y, z;
};
```

Структура Plane содержит 4 значения, которые необходимы для формирования уравнения плоскости. Эти плоскости задают лицевые стороны объектов (их грани).

```
// Структура описания плоскости, в формате: ax + by + cz + d = 0
struct Plane
{
  GLfloat a, b, c, d;
};
```

Структура Face содержит всю информацию, необходимую для задания треугольника для отбрасывания тени.
Индексы задают порядок задания вершин.
Нормали вершин (vertexIndices) используются, чтобы вычислить ориентацию лицевых граней в 3D, с помощью них можно определить, с какой стороны находится источник света с лицевой или с обратной, по отношению к грани.
Уравнение плоскости (planeEquation) описывает плоскость, в которой этот треугольник находится в 3D.
Индексы соседей (neighbourIndices) - индексы в массиве граней объекта. Это позволяет Вам определить, как грань соединена с другими гранями на каждой стороне треугольника.

Параметр видимости (visible) используется, чтобы определить, является ли грань "видимой" источнику света, который отбрасывает тени.

```
// Структура описания грани объекта
struct Face
{
  int vertexIndices[3]; // Индекс каждой вершины в объекте, которые задают треугольник грани
  Point3f normals[3];   // Нормаль каждой вершины
  Plane planeEquation;  // Уравнение плоскости с треугольником
  int neighbourIndices[3]; // Индекс каждой грани, которая является соседом в этом объекте
  bool visible;         // Свет видит эту грань?
};
```

Наконец, структура ShadowedObject содержит все вершины и грани объекта. Память для каждого из массивов выделяется динамически, когда объект загружается.

```
struct ShadowedObject
{
  int nVertices;
  Point3f *pVertices; // Динамически выделяется

  int nFaces;
  Face *pFaces;       // Динамически выделяется
};
```

Действия функции readObject понятны из ее названия. В ней происходит заполнение структуры значениями из файла, выделение памяти для вершин и граней. Индексы соседних граней инициализируются значением –1, что означает, что их нет. Они будут рассчитаны позже.

```
bool readObject( const char *filename, ShadowedObject& object )
{
  FILE *pInputFile;
  int i;

  pInputFile = fopen( filename, "r" );
  if ( pInputFile == NULL )
  {
    cerr << "Не могу открыть файл объекта: " << filename << endl;
    return false;
  }
  // Читать вершины
  fscanf( pInputFile, "%d", &object.nVertices );
  object.pVertices = new Point3f[object.nVertices];
  for ( i = 0; i < object.nVertices; i++ )
  {
    fscanf( pInputFile, "%f", &object.pVertices[i].x );
    fscanf( pInputFile, "%f", &object.pVertices[i].y );
    fscanf( pInputFile, "%f", &object.pVertices[i].z );
  }
  // Читать грани
  fscanf( pInputFile, "%d", &object.nFaces );
  object.pFaces = new Face[object.nFaces];
  for ( i = 0; i < object.nFaces; i++ )
  {
    int j;
    Face *pFace = &object.pFaces[i];

    for ( j = 0; j < 3; j++ )
      pFace->neighbourIndices[j] = -1; // Нет соседей

    for ( j = 0; j < 3; j++ )
    {
      fscanf( pInputFile, "%d", &pFace->vertexIndices[j] );
      pFace->vertexIndices[j]--; // В файле индексы начинают с 1, а в массиве с 0
    }
    for ( j = 0; j < 3; j++ )
    {
      fscanf( pInputFile, "%f", &pFace->normals[j].x );
      fscanf( pInputFile, "%f", &pFace->normals[j].y );
      fscanf( pInputFile, "%f", &pFace->normals[j].z );
    }
  }
  return true;
}
```

Аналогично, действия killObject очевидны – в ней удаляется выделенная память, и обнуляются указатели. Обратите внимание, что, строка добавлена в KillGLWindow для вызова этой функции с указанным объектом.

```
void killObject( ShadowedObject& object )
{
 delete[] object.pFaces;
 object.pFaces = NULL;
 object.nFaces = 0;
 delete[] object.pVertices;
 object.pVertices = NULL;
object.nVertices = 0;
}
```

Теперь интереснее, рассмотрим функцию setConnectivity. Эта функция вычисляет все соседние грани для всех граней в данном объекте. Вот ее псевдокод:

```
Для каждой грани (A) в объекте
 Для каждой стороны в грани A
  Если мы не знаем эти грани еще как соседи
   Для каждой грани (B) в объекте (не включая грань A)
    Для каждой стороны в B
     Если сторона в A совпадает со стороной B, тогда они соседи по этой стороне
      Задать свойство связанности каждой из граней A и B,
      Затем перейти на следующую сторону в A
```

Последние две строки связаны со следующим кодом. Получаем индексы двух смежных вершин грани A и грани B, которые находятся на одной стороне грани. Так как у грани 3 вершины, то (edgeA+1) %3 дает индекс следующей вершины. Затем надо проверить, совпадают ли вершины (порядок следования индексов вершин может быть различным).

```
 int vertA1 = pFaceA->vertexIndices[edgeA];
 int vertA2 = pFaceA->vertexIndices[( edgeA+1 )%3];
 int vertB1 = pFaceB->vertexIndices[edgeB];
 int vertB2 = pFaceB->vertexIndices[( edgeB+1 )%3];
 // Проверить если они соседи, т.е. одинаковые стороны
 if (( vertA1 == vertB1 && vertA2 == vertB2 ) || ( vertA1 == vertB2 && vertA2 == vertB1 ))
 {
  pFaceA->neighbourIndices[edgeA] = faceB;
  pFaceB->neighbourIndices[edgeB] = faceA;
  edgeFound = true;
  break;
 }
```

К счастью, следующая функция очень проста, в то время как Вы усиленно дышите. Функция drawObject визуализирует каждую грань объекта.

```
// Нарисовать объект, просто нарисовать каждую треугольную грань

void drawObject( const ShadowedObject& object )
{
 glBegin( GL_TRIANGLES );
 for ( int i = 0; i < object.nFaces; i++ )
 {
  const Face& face = object.pFaces[i];
  for ( int j = 0; j < 3; j++ )
  {
   const Point3f& vertex = object.pVertices[face.vertexIndices[j]];
   glNormal3f( face.normals[j].x, face.normals[j].y, face.normals[j].z );
   glVertex3f( vertex.x, vertex.y, vertex.z );
  }
 }
 glEnd();
}
```

Вычисление уравнения плоскости выглядит мерзко, но это только простая математическая формула, которую надо взять из учебника, который Вам необходим.

```
void calculatePlane( const ShadowedObject& object, Face& face )
{
 // Упростить обращение к вершинам объектов
 const Point3f& v1 = object.pVertices[face.vertexIndices[0]];
 const Point3f& v2 = object.pVertices[face.vertexIndices[1]];
 const Point3f& v3 = object.pVertices[face.vertexIndices[2]];

 face.planeEquation.a = v1.y*(v2.z-v3.z) + v2.y*(v3.z-v1.z) + v3.y*(v1.z-v2.z);
 face.planeEquation.b = v1.z*(v2.x-v3.x) + v2.z*(v3.x-v1.x) + v3.z*(v1.x-v2.x);
 face.planeEquation.c = v1.x*(v2.y-v3.y) + v2.x*(v3.y-v1.y) + v3.x*(v1.y-v2.y);
 face.planeEquation.d = -( v1.x*( v2.y*v3.z - v3.y*v2.z ) +
      v2.x*(v3.y*v1.z - v1.y*v3.z) +  v3.x*(v1.y*v2.z - v2.y*v1.z) );
}
```

Вы восстановили свое дыхание? Отлично! Потому что далее идет код отбрасывания тени! Функция castShadow делает все необходимые настройки OpenGL, и вызывает doShadowPass для двухпроходной визуализации.

Сначала, мы определяем, какая поверхность обращена к свету. Это можно понять при помощи определения, с какой стороны поверхности находится свет. Подставим позицию света в уравнение плоскости. Если мы получили значение больше чем 0, тогда совпадает с направлением нормали плоскости, и свет видим. Если нет, то свет не видим. (Снова, обратитесь к хорошему учебнику по математике, если Вам, что-то не понятно).

```
void castShadow( ShadowedObject& object, GLfloat *lightPosition )
{
 // Определим, какие плоскости видят свет.
 for ( int i = 0; i < object.nFaces; i++ )
 {
  const Plane& plane = object.pFaces[i].planeEquation;
  GLfloat side = plane.a*lightPosition[0]+
    plane.b*lightPosition[1]+
    plane.c*lightPosition[2]+
    plane.d;

  if ( side > 0 )
    object.pFaces[i].visible = true;
  else
    object.pFaces[i].visible = false;
 }
```

Следующий раздел задает настройки OpenGL для визуализации теней.

Сначала, мы помещаем в стек все атрибуты, которые будут изменены. Это делает восстановление их намного проще.

Освещение отключено, потому что мы не будем что-либо выводить в буфер цвета, только в буфер трафарета. По той же самой причине, маска цвета выключает все цветные компоненты (поэтому отрисовка полигонов не влияет на выходной буфер).

Хотя тест глубины используется, мы не хотим, чтобы тени проявлялись себя, как сплошные объекты в буфере глубины, поэтому задана такая маска глубины, чтобы избежать этого.

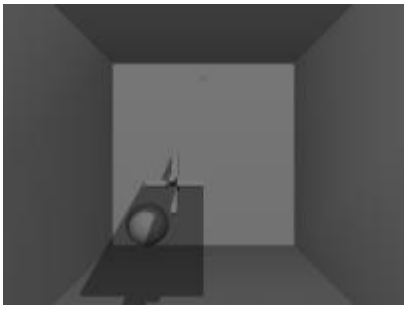
Буфер трафарета включен, так как именно в него и будут рисоваться тени.

```
 glPushAttrib( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT |
          GL_ENABLE_BIT | GL_POLYGON_BIT | GL_STENCIL_BUFFER_BIT );
 glDisable( GL_LIGHTING );    // Выключим свет
 glDepthMask( GL_FALSE );     // Выключим запись в буфер глубины
 glDepthFunc( GL_LEQUAL );
 glEnable( GL_STENCIL_TEST ); // Включим тест буфера трафарета
 glColorMask( GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE ); // Не рисовать в буфер цвета
 glStencilFunc( GL_ALWAYS, 1, 0xFFFFFFFFL );
```

Отлично, сейчас непосредственно возьмемся за визуализацию теней. Мы еще вернемся к этому моменту, когда будем рассматривать функцию doShadowPass. Тени визуализируются в два прохода, как Вы можете видеть, на первом проходе происходит инкремент буфера трафарета на лицевых гранях (отбрасывающих тень), на втором декремент буфера трафарета на невидимых гранях ("выключающий" тень между объектом и любыми другими поверхностями).

```
 // Первый проход увеличение значения трафарета в тени
 glFrontFace( GL_CCW );
 glStencilOp( GL_KEEP, GL_KEEP, GL_INCR );
 doShadowPass( object, lightPosition );
 // Второй проход уменьшение значения трафарета в тени
 glFrontFace( GL_CW );
 glStencilOp( GL_KEEP, GL_KEEP, GL_DECR );
 doShadowPass( object, lightPosition );
```

Чтобы понять как, работает второй проход, лучше всего закомментировать его и выполнить программу. Вот примерно, что Вы можете получить:



< не загружен =(>

Рисунок 1: Первый проход

Рисунок 2: Второй проход

В конце этой функции, выводится со смешением один прямоугольник на весь экран, для того чтобы отбросить тень. Чем более темным Вы сделаете этот прямоугольник, тем более темными будут тени. Поэтому, чтобы изменить свойства тени, измените значения цветовых компонент в glColor4f. При более высоких значениях альфа-канала тени будут более темными. Или Вы можете сделать их красными, зелеными, фиолетовыми...!

```
  glFrontFace( GL_CCW );
 // Включить визуализацию цветового буфера для всех компонент
 glColorMask( GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE );
   // Нарисовать теневой прямоугольник на весь экран
 glColor4f( 0.0f, 0.0f, 0.0f, 0.4f );
 glEnable( GL_BLEND );
 glBlendFunc( GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA );
 glStencilFunc( GL_NOTEQUAL, 0, 0xFFFFFFFFL );
 glStencilOp( GL_KEEP, GL_KEEP, GL_KEEP );
 glPushMatrix();
 glLoadIdentity();
 glBegin( GL_TRIANGLE_STRIP );
  glVertex3f(-0.1f, 0.1f,-0.10f);
   glVertex3f(-0.1f,-0.1f,-0.10f);
   glVertex3f( 0.1f, 0.1f,-0.10f);
   glVertex3f( 0.1f,-0.1f,-0.10f);
 glEnd();
 glPopMatrix();
 glPopAttrib();
}
```

Отлично, далее рассмотрим, как рисуются теневые четырехугольники. Как это работает? Что происходит, когда вы проходите каждую грань, и если она видимая, тогда вы проверяете все ее стороны. Если у стороны нет соседа, или соседняя грань не видима, тогда эта сторона отбрасывает тень. Если Вы как следует, обдумаете эти два варианта, тогда Вы поймете, что так оно и есть на самом деле. Рисуя четырехугольник (как два треугольника), который включает в себя точки стороны, и точки проекции стороны в “бесконечность”, Вы получаете тень, отбрасываемую этой стороной.

Здесь используется простой и прямолинейный подход для рисования в "бесконечность", каждый теневой многоугольник необходимо отсечь относительно всех многоугольников, с которыми он сталкивается. Это может вызвать "стресс" у видеокарты. В быстродействующей модификации этого алгоритма, Вы должны отсечь многоугольник по объектам позади него. Это намного сложнее и в этом алгоритме есть проблемы, но если вы хотите сделать это, то найдите статью на Gamasutra об этом.
Код реализации этого алгоритма более прост, чем рассуждения о нем. Вот начальный фрагмент кода, который надо повторить по всем объектам для получения тени. В конце этого фрагмента мы получаем номер края - j, и номер ее соседней грани - neighbourIndex.

```
void doShadowPass( ShadowedObject& object, GLfloat *lightPosition )
{
  for ( int i = 0; i < object.nFaces; i++ )
  {
    const Face& face = object.pFaces[i];
    if ( face.visible )
    {
      // Для каждого края
      for ( int j = 0; j < 3; j++ )
      {
        int neighbourIndex = face.neighbourIndices[j];
```

Затем, идет проверка, видима ли соседняя грань, если нет, то этот край отбрасывает тень.

```
        // Если нет соседа, или сосед не виден, тогда край отбрасывает тень
        if ( neighbourIndex == -1 || object.pFaces[neighbourIndex].visible == false )
        {
```

В следующем сегменте кода вычисляются две вершины текущего края - v1 и v2. Затем вычисляются вершины v3 и v4, которые получаются из проекции вдоль вектора между источником света и первым краем. Они масштабируются к БЕСКОНЕЧНОСТИ, которая была задана как очень большое значение.

```
          // Взять точки на стороне
          const Point3f& v1 = object.pVertices[face.vertexIndices[j]];
          const Point3f& v2 = object.pVertices[face.vertexIndices[( j+1 )%3]];

          // Вычислить две вершины на расстоянии
          Point3f v3, v4;
          v3.x = ( v1.x-lightPosition[0] )*INFINITY;
          v3.y = ( v1.y-lightPosition[1] )*INFINITY;
          v3.z = ( v1.z-lightPosition[2] )*INFINITY;
          v4.x = ( v2.x-lightPosition[0] )*INFINITY;
          v4.y = ( v2.y-lightPosition[1] )*INFINITY;
          v4.z = ( v2.z-lightPosition[2] )*INFINITY;
```

Я думаю, что Вы поймете следующий раздел, в нем только выводится четырехугольник, заданный теми четырьмя точками:

```
          // Нарисовать четырехугольник (как полоску из треугольников)
          glBegin( GL_TRIANGLE_STRIP );
            glVertex3f( v1.x, v1.y, v1.z );
            glVertex3f( v1.x+v3.x, v1.y+v3.y, v1.z+v3.z );
            glVertex3f( v2.x, v2.y, v2.z );
            glVertex3f( v2.x+v4.x, v2.y+v4.y, v2.z+v4.z );
          glEnd();
        }
      }
    }
  }
}
```

На этом, секция отбрасывания тени завершена. Но мы еще не закончили! Что там в drawGLScene? Начнем с простых вещей: очистка буферов, позиционирование источника света, и отрисовка сферы:

```
bool drawGLScene()
{
  GLmatrix16f Minv;
  GLvector4f wlp, lp;
  // Очистка буфера цвета, глубины и трафарета
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
  glLoadIdentity(); // Сброс матрицы вида модели
  glTranslatef(0.0f, 0.0f, -20.0f); // Наезд в экран на 20 единиц
  glLightfv(GL_LIGHT1, GL_POSITION, LightPos); // Позиция Света1
  glTranslatef(SpherePos[0], SpherePos[1], SpherePos[2]);// Позиция сферы
  gluSphere(q, 1.5f, 32, 16); // Отрисовать сферу
```

Затем, мы должны вычислить позицию источника света относительно локальной системы координат объекта. В комментариях каждый шаг объяснен подробнее. В Minv хранится матрица преобразования объекта, однако вращения и сдвиг сделаны с отрицательными аргументами, поэтому это фактически инверсия матрицы преобразования. Затем lp создается как копия позиции света, и умноженная на матрицу. Таким образом, lp - позиция света в системе координат объекта.

```
  glLoadIdentity(); // Сброс матрицы
  glRotatef(-yrot, 0.0f, 1.0f, 0.0f);        // Вращение на -yrot по оси Y
  glRotatef(-xrot, 1.0f, 0.0f, 0.0f);        // Вращение на -xrot по оси X
  glTranslatef(-ObjPos[0], -ObjPos[1], -ObjPos[2]); // Сдвинуть в противоположном направлении по всем осям исходя из
значений ObjPos[] (X, Y, Z)
  glGetFloatv(GL_MODELVIEW_MATRIX,Minv); // Получить матрицу
  lp[0] = LightPos[0];            // Сохранить позицию света X в lp[0]
  lp[1] = LightPos[1];            // Сохранить позицию света Y в lp[1]
  lp[2] = LightPos[2];            // Сохранить позицию света Z в lp[2]
  lp[3] = LightPos[3];            // Сохранить позицию света в lp[3]
  VMatMult(Minv, lp);             // Сохраним преобразованный вектор в массиве 'lp'
```

Теперь забабахаем работу по выводу комнаты, и объекта. Вызов castShadow выводит тень объекта.

```
  glLoadIdentity();          // Сброс матрицы
  glTranslatef(0.0f, 0.0f, -20.0f); // Наезд на 20 единиц в экран
  DrawGLRoom();              // Отрисовка комнаты
  glTranslatef(ObjPos[0], ObjPos[1], ObjPos[2]); // Позиция объекта
  glRotatef(xrot, 1.0f, 0.0f, 0.0f);         // Вращение по оси X на xrot
  glRotatef(yrot, 0.0f, 1.0f, 0.0f);         // Вращение по оси Y на yrot
  drawObject(obj);           // Процедура для рисования загруженного объекта
  castShadow(obj, lp);       // Процедура для отбрасывания тени используя силуэт объекта
```

Следующие несколько линии выводят небольшой оранжевый круг, обозначая место света:

```
  glColor4f(0.7f, 0.4f, 0.0f, 1.0f);// Оранжевый цвет
  glDisable(GL_LIGHTING);            // Отключить свет
  glDepthMask(GL_FALSE);             // Отключить маску глубины
  glTranslatef(lp[0], lp[1], lp[2]);// Перенос позиции света
                // Отмечу, что мы все еще в локальной системе координат
  gluSphere(q, 0.2f, 16, 8);         // Нарисовать маленькую оранжевую сферу
  glEnable(GL_LIGHTING);             // Разрешение света
  glDepthMask(GL_TRUE);              // Разрешение маски глубины
```

В конце меняем позицию объекта и выход.

```
  xrot += xspeed;              // Увеличим xrot на xspeed
  yrot += yspeed;              // Увеличим yrot на yspeed
  glFlush();                // Сброс OpenGL
  return TRUE;              // Скажем OK
}
```

Реализуем функцию DrawGLRoom, и в ней выводится пучок прямоугольников, чтобы на них падали тени:

```
void DrawGLRoom()               // Нарисовать комнату
{
  glBegin(GL_QUADS);            // Рисовать прямоугольники
    // Пол
    glNormal3f(0.0f, 1.0f, 0.0f);       // Нормаль вверх
    glVertex3f(-10.0f,-10.0f,-20.0f);     // Лево Назад
    glVertex3f(-10.0f,-10.0f, 20.0f);     // Лево Перед
    glVertex3f( 10.0f,-10.0f, 20.0f);     // Право Перед
    glVertex3f( 10.0f,-10.0f,-20.0f);     // Право Назад
    // Потолок
    glNormal3f(0.0f,-1.0f, 0.0f);       // Нормаль вниз
    glVertex3f(-10.0f, 10.0f, 20.0f);     // Лево Перед
    glVertex3f(-10.0f, 10.0f,-20.0f);     // Лево Назад
    glVertex3f( 10.0f, 10.0f,-20.0f);     // Назад Право
    glVertex3f( 10.0f, 10.0f, 20.0f);     // Право Верх
    // Передняя стена
    glNormal3f(0.0f, 0.0f, 1.0f);       // Нормаль вдаль от зрителя
    glVertex3f(-10.0f, 10.0f,-20.0f);     // Лево Верх
    glVertex3f(-10.0f,-10.0f,-20.0f);     // Лево Низ
    glVertex3f( 10.0f,-10.0f,-20.0f);     // Право Низ
    glVertex3f( 10.0f, 10.0f,-20.0f);     // Право Верх
    // Задняя стена
    glNormal3f(0.0f, 0.0f,-1.0f);       // Нормаль на зрителя
    glVertex3f( 10.0f, 10.0f, 20.0f);     // Право Верх
    glVertex3f( 10.0f,-10.0f, 20.0f);     // Право Низ
    glVertex3f(-10.0f,-10.0f, 20.0f);     // Лево Низ
    glVertex3f(-10.0f, 10.0f, 20.0f);     // Лево Верх
    // Левая стена
    glNormal3f(1.0f, 0.0f, 0.0f);       // Нормаль вправо
    glVertex3f(-10.0f, 10.0f, 20.0f);     // Верх Перед
    glVertex3f(-10.0f,-10.0f, 20.0f);     // Низ Перед
    glVertex3f(-10.0f,-10.0f,-20.0f);     // Назад Низ
    glVertex3f(-10.0f, 10.0f,-20.0f);     // Назад Перед
    // Правая стена
    glNormal3f(-1.0f, 0.0f, 0.0f);      // Нормаль влево
    glVertex3f( 10.0f, 10.0f,-20.0f);     // Назад Верх
    glVertex3f( 10.0f,-10.0f,-20.0f);     // Назад Низ
    glVertex3f( 10.0f,-10.0f, 20.0f);     // Низ Перед
    glVertex3f( 10.0f, 10.0f, 20.0f);     // Верх Перед
  glEnd();              // Конец рисования
}
```

И прежде, чем я забыл, вот код функции VMatMult, которая умножает вектор на матрицу:

```
void VMatMult(GLmatrix16f M, GLvector4f v)
{
  GLfloat res[4];           // Сохранить вычисленный результат
  res[0]=M[ 0]*v[0]+M[ 4]*v[1]+M[ 8]*v[2]+M[12]*v[3];
  res[1]=M[ 1]*v[0]+M[ 5]*v[1]+M[ 9]*v[2]+M[13]*v[3];
  res[2]=M[ 2]*v[0]+M[ 6]*v[1]+M[10]*v[2]+M[14]*v[3];
  res[3]=M[ 3]*v[0]+M[ 7]*v[1]+M[11]*v[2]+M[15]*v[3];
  v[0]=res[0];              // Результат сохранен обратно в v[]
  v[1]=res[1];
  v[2]=res[2];
  v[3]=res[3];              // Гомогенные координаты
}
```

Функция для загрузки объекта очень проста в ней только вызов readObject, и вычисление связанности и уравнений плоскости для каждой грани.

```
int InitGLObjects()          // Инициализация объектов
{
  if (!readObject("Data/Object2.txt", obj)) // Чтение Object2 в obj
  {
    return FALSE;            // Если сбой вернем False
  }
  setConnectivity(obj);      // Зададим связанность грань к грани

  for ( int i=0;i < obj.nFaces;i++)    // Цикл по всем граням
    calculatePlane(obj, obj.pFaces[i]);// Уравнение плоскости
  return TRUE;               // Вернем True
}
```

Наконец, вспомогательная функция KillGLObjects для уничтожения всех объектов.

```
void KillGLObjects()
{
  killObject( obj );
}
```

Все другие функции не требуют никакого дополнительного объяснения. Я не опустил стандартный код уроков NeHe, и все определения переменных и функцию обработки клавиатуры. Комментариев в них достаточно.

Я хотел бы обратить Ваше внимание на некоторое вещи:

Сфера не отбрасывает тень на стенку. В реальности, сфера должна также отбрасывать тень.

Если Вы наблюдаете, резкое падение чистоты смены кадров, пробуйте переключить в полноэкранный режим, или задать вашу цветовую глубину экрана в 32bpp.

Arseny L. пишет: Если Вы имеете проблемы с видеокартами TNT2 в оконном режиме, проверьте, что ваша цветовая глубина экрана не задана в 16bit. В 16bit цветном режиме, буфер трафарета эмулируется, что приводит к вялой производительности. Нет никаких проблем в 32bit режиме (я имею TNT2 Ultra, и я проверил это).

Я должен признаться, что это былf длинная задача написать этот урок. Это дает Вам точную оценку той работы, которую выполняет Джеф! Я надеюсь, что Вам понравился это урок, и Вы благодарите Banu, который написал первоначальный код! ЕСЛИ что-нибудь не понятно, то пишите письма - brettporter@yahoo.com

Урок 28. Фрагменты поверхностей Безье.

Bezier Patches

В этом уроке пойдет речь о поверхностях Безье, и я буду надеяться, что кто-то, прочитав этот урок, покажет нам более интересные варианты использования их, чем я. Здесь не пойдет речь о библиотеке фрагментов поверхностей Безье (Bezier patch, или патчи Безье, или лоскуты Безье), а скорее я попытаюсь ознакомить вас концепцией того, как реально эти кривые поверхности работают. Так как это, скорее всего не формальное изложение, то я иногда делаю небольшие отступления от формальной терминологии, для лучшего понимания сути дела. Я надеюсь, что это поможет. Если вы уже знакомы с Безье, и вы читаете эту статью, что бы посмотреть что я тут накрутил, то позор Вам! Но, если я действительно где-то ошибся, сообщите мне об этой ошибке или NeHe, в конце концов, никто не совершенен. И еще одно, код в уроке не оптимизирован, в противоположность моей обычной практики, это потому что я хочу дать всем возможность точно понять, как это работает. Отлично, хватит вводных слов, смотрите!

Математика – дьявольская музыка:: (предупреждаю, вероятно, это длинная секция)

Отлично, Вам будет очень трудно понять поверхности Безье без минимума математики, стоящей за этим, однако, если Вы не чувствуете в себе сил и желания читать эту секцию, или Вы уже знакомы с этой математикой, Вы можете опустить ее. Вначале мы займемся кривыми Безье, затем рассмотрим фрагменты Безье.

Даже, если Вы не знакомы ранее с таким термином, как кривые Безье, но работали в каком-нибудь редакторе графики (например, CorelDraw), то, скорее всего Вы знакомы с кривыми Безье, пусть даже и не под таким названием. Так как это основной метод рисования кривых линий. Обыкновенно для работы этого метода необходимо четыре точки, при чем две точки, изображают касательные, которые идут слева и справа. Вот как это выглядит:

Из разных форм представления кривой Безье – это наиболее простая (удлинять кривые можно присоединяя их друг к другу (много раз без вмешательства пользователя)). Эта кривая обычно задается только четырьмя точками: двумя концевыми контрольными точками и двумя средними контрольными точками. Для компьютера все точки идентичны, но для упрощения работы пользователя мы часто соединяем первые и последние две, соответственно, поскольку эти линии всегда будут касательными к конечной точке. Этот тип кривых задается параметрически и рисуется при помощи нахождения заданного числа равноотстоящих друг от друга точек на кривой и последующего их соединения прямыми линями. Таким образом, мы может контролировать разрешение фрагмента (patch) и скорость вычислений. Наиболее стандартный способ использовать это при тесселяции (разбиении), т.е. уменьшать количество разбиений поверхности при большом удалении от камеры и увеличивать количество разбиений при приближении к наблюдателю, чтобы изогнутая поверхность всегда была гладкой и при этом скорость вывода была наилучшей.

Кривые Безье базируются на простой функции, из которой получаются более сложные версии. Вот эта функция:

t + (1 - t) = 1

Не правда ли выглядит удивительно просто? Это действительно Безье, вернее наиболее простая кривая Безье, кривая первой степени. Как можно догадаться из последней фразы, кривые Безье – это полиномы, и как мы помним из алгебры, полином первой степени – это всего лишь прямая линия; что не очень интересно для нас. Хорошо, поскольку это простая функция истинна для всех значений t, мы можем взять квадрат, куб, любую степень этого выражения с обеих сторон, и это все еще будет истиной? Отлично, давайте попробуем кубическую степень.

(t + (1-t))^3 = 1^3
t^3 + 3*t^2*(1-t) + 3*t*(1-t)^2 + (1-t)^3 = 1

Это уравнение мы и будем использовать для вычисления наиболее общей кривой Безье третьей степени. Это наиболее общее уравнение по двум причинам: a) это полином с наиболее низкой степенью, который не обязательно должен лежать в плоскости (есть четыре контрольных точки) и b) касательные линии к сторонам не зависят одна от другой (в полиноме со второй степенью есть только три контрольных точки). Поэтому вы уже видите кривую Безье? Хе-хе, по мне так ни то ни другое, так как я должен добавить еще кое-что.

Отлично, так как с левой стороны стоит единица, то можно предположить, что когда мы сложим все компоненты, то они все еще будут равны единице. Это похоже на то, что можно описать, как много каждой контрольной точки будет использоваться для вычисления точки кривой? (Подсказка: скажите да ;)). Хорошо Вы правы! Когда мы хотим вычислить значение точки находящейся на кривой, мы просто умножаем каждую компоненту уравнения на свою контрольную точку (как вектор) и находим сумму. Вообще, мы работаем с 0 <= t <= 1, но это технически не обязательно. Непонятно? Вот эта функция.

P1*t^3 + P2*3*t^2*(1-t) + P3*3*t*(1-t)^2 + P4*(1-t)^3 = Pnew

Поскольку полиномы всегда непрерывны, это хороший способ сделать морфинг между 4 точками. Хотя фактически достижимы (интерполируются) только точки P1 и P4, когда t = 1 и t =0, соответственно. Точки P2 и P3 только аппроксимируются, т.е. кривая проходит рядом с ними, но не через них.

Теперь все прекрасно, но как я могу это использовать в 3D? Хорошо, довольно просто дальше сформировать фрагмент Безье. Для этого нам надо 16 контрольных точек (4*4), и две переменные t и v. Далее вы вычисляете точки v вдоль 4 параллельных кривых, затем используете эти 4 точки для создания новой кривой и вычисления t вдоль этой кривой. Вычисляя нужное количество этих точек, мы можем нарисовать треугольную полоску для соединения их, таким образом, рисуя фрагмент Безье. (Примечание переводчика. Так как каждая концевая точка четырехугольника фрагмента относится к двум сторонам, то всего для их представления надо 8 точек. Затем надо по две контрольные точки на каждую сторону для формирования кривых Безье на этих сторонах. Вначале надо вычислить кривую Безье на одной стороне четырехугольника, затем надо вычислить следующую параллельную ей кривую Безье и так далее. Конечно, для формирования полосок из треугольников необходимо иметь две кривые и рисовать полоски между ними.)

Хорошо, я думаю, пока хватит математики, давайте посмотрим код!

```
#include <windows.h>       // Заголовочный файл для Windows
#include <math.h>          // Заголовочный файл для математической библиотеки
#include <stdio.h>         // Заголовочный файл для стандартного ввода/вывода
#include <stdlib.h>        // Заголовочный файл для стандартной библиотеки
#include <gl\gl.h>         // Заголовочный файл для библиотеки OpenGL32
#include <gl\glu.h>        // Заголовочный файл для библиотеки GLu32
#include <gl\glaux.h>      // Заголовочный файл для GLaux библиотеки

typedef struct point_3d {  // Структура для 3D точки( НОВОЕ )
  double x, y, z;
} POINT_3D;

typedef struct bpatch {    // Структура для полинома фрагмента Безье 3 степени (НОВОЕ)
  POINT_3D  anchors[4][4]; // Сетка 4x4 анкерных (anchor) точек
  GLuint    dlBPatch;      // Список для фрагмента Безье
  GLuint    texture;       // Текстура для фрагмента
} BEZIER_PATCH;

HDC       hDC=NULL;        // Контекст устройства
HGLRC     hRC=NULL;        // Контекст визуализации
HWND      hWnd=NULL;       // Дескриптор окна
HINSTANCE hInstance;       // Экземпляр приложения

bool      keys[256];       // Массив для работы с клавиатурой
bool      active=TRUE;     // Флаг активности приложения
bool      fullscreen=TRUE; // Флаг полноэкранного режима

DEVMODE      DMsaved;      // Сохранить настройки предыдущего режима ( НОВОЕ )

GLfloat      rotz = 0.0f;  // Вращение по оси Z
BEZIER_PATCH    mybezier;  // Фрагмент Безье для использования ( НОВОЕ )
BOOL      showCPoints=TRUE;// Переключатель отображения контрольных точек сетки ( НОВОЕ )
int       divs = 7;         // Число интерполяции (Контроль разрешения полигона) ( НОВОЕ )

LRESULT  CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);      // Декларация для WndProc
```

Далее идут несколько функций для оперирования векторами. Если вы фанат C++, то вы может использовать вместо них какой-то класса 3D точки.

```
// Сложить 2 точки.
POINT_3D pointAdd(POINT_3D p, POINT_3D q) {
  p.x += q.x;    p.y += q.y;    p.z += q.z;
  return p;
}

// Умножение точки на константу
POINT_3D pointTimes(double c, POINT_3D p) {
  p.x *= c;  p.y *= c;  p.z *= c;
  return p;
}


// Функция для упрощения создания точки
POINT_3D makePoint(double a, double b, double c) {
  POINT_3D p;
  p.x = a;  p.y = b;  p.z = c;
  return p;
}
```

Затем идет функция для вычисления полиномов третьей степени (каждый член в уравнении кривой Безье, является одним из так называемых полиномов Берштейна), ей надо передать переменную u и массив из 4 точек p и вычислить точку на кривой. Изменяя u на одинаковые приращения между 0 и 1, мы получим хорошую аппроксимацию кривой.

```
// Вычисляем полином 3 степени на основании массива из 4 точек
// и переменной u, которая обычно изменяется от 0 до 1
POINT_3D Bernstein(float u, POINT_3D *p) {

  POINT_3D  a, b, c, d, r;

  a = pointTimes(pow(u,3), p[0]);
  b = pointTimes(3*pow(u,2)*(1-u), p[1]);
  c = pointTimes(3*u*pow((1-u),2), p[2]);
  d = pointTimes(pow((1-u),3), p[3]);
  r = pointAdd(pointAdd(a, b), pointAdd(c, d));
  return r;
}
```

Эта функция делает львиную долю работы, генерируя все полоски треугольников и сохраняя их в списке отображения. Мы это делаем, для того чтобы не перевычислять фрагмент каждый кадр. Между прочим, вы можете попробовать использовать урок о Морфинге для морфинга контрольных точек фрагмента. При этом получиться интересный эффект сглаженного морфинга с относительно небольшими затратами (вы только делает морфинг 16 точек, но вы должны перевычислить их). Массив "last" используется для сохранения предыдущей линии точек (поскольку для треугольных полосок необходимы обе строки). Также координаты текстуры вычисляются при помощи использования u и v значений в виде процентов (плоское наложение).

Одну вещь мы не делаем – вычисление нормалей для освещения. Когда вы начнете это делать, в общем, вы будете иметь два параметра. Первый параметр, который вы должны найти – это центр каждого треугольника, и затем использовать побитное исчисление и вычисление тангенса обоих осей x и y, затем сделать векторное произведение чтобы получить перпендикуляр к обоим осям, ЗАТЕМ нормализовать вектор и использовать его как нормаль. ИЛИ (это более быстрый путь) вы можете, для того чтобы получить хорошую аппроксимацию, использовать только нормаль треугольника (вычисленную на ваш любимый манер). Я предпочитаю последний способ, так как, по моему мнению, не стоит жертвовать скоростью взамен не большому улучшению реализма.

```
// Создание списков отображения на основе данных фрагмента
// и числе разбиений
GLuint genBezier(BEZIER_PATCH patch, int divs) {
  int    u = 0, v;
  float    py, px, pyold;
  GLuint    drawlist = glGenLists(1);  // Создать список отображения
  POINT_3D  temp[4];
  POINT_3D  *last = (POINT_3D*)malloc(sizeof(POINT_3D)*(divs+1));
       // Массив точек для отметки первой линии полигонов

  if (patch.dlBPatch != NULL)          // Удалить старые списки отображения
    glDeleteLists(patch.dlBPatch, 1);

  temp[0] = patch.anchors[0][3];       // Первая производная кривая (Вдоль оси X)
  temp[1] = patch.anchors[1][3];
  temp[2] = patch.anchors[2][3];
  temp[3] = patch.anchors[3][3];

  for (v=0;v<=divs;v++) {              // Создание первой линии точек
    px = ((float)v)/((float)divs);     // Процент вдоль оси Y
    // Используем 4 точки из производной кривой для вычисления точек вдоль кривой
    last[v] = Bernstein(px, temp);
  }

  glNewList(drawlist, GL_COMPILE);     // Начнем новый список отображения
  glBindTexture(GL_TEXTURE_2D, patch.texture); // Присоединим к текстуре
```

```
  for (u=1;u<=divs;u++) {
    py    = ((float)u)/((float)divs);          // Процент вдоль оси Y
    pyold = ((float)u-1.0f)/((float)divs);     // Процент вдоль старой оси Y
    temp[0] = Bernstein(py, patch.anchors[0]); // Вычислим новые точки Безье
    temp[1] = Bernstein(py, patch.anchors[1]);
    temp[2] = Bernstein(py, patch.anchors[2]);
    temp[3] = Bernstein(py, patch.anchors[3]);

    glBegin(GL_TRIANGLE_STRIP);        // Начнем новую полоску треугольников

    for (v=0;v<=divs;v++) {
      px = ((float)v)/((float)divs);   // Процент вдоль оси X
      glTexCoord2f(pyold, px);         // Применим старые координаты текстуры
      glVertex3d(last[v].x, last[v].y, last[v].z);  // Старая точка
      last[v] = Bernstein(px, temp);   // Генерируем новую точку
      glTexCoord2f(py, px);            // Применим новые координаты текстуры
      glVertex3d(last[v].x, last[v].y, last[v].z);  // Новая точка
    }

    glEnd();            // Конец полоски треугольников
  }

  glEndList();          // Конец списка
  free(last);           // Освободить старый массив вершин
  return drawlist;      // Вернуть список отображения
}
```

Далее зададим значения контрольных точек фрагмента, которые я подобрал, чтобы продемонстрировать эффект. Не стесняйтесь изменять эти значения и посмотреть, что же получится при этом.

```
void initBezier(void) {
  mybezier.anchors[0][0] = makePoint(-0.75,  -0.75,  -0.50);  // Вершины Безье
  mybezier.anchors[0][1] = makePoint(-0.25,  -0.75,   0.00);
  mybezier.anchors[0][2] = makePoint( 0.25,  -0.75,   0.00);
  mybezier.anchors[0][3] = makePoint( 0.75,  -0.75,  -0.50);
  mybezier.anchors[1][0] = makePoint(-0.75,  -0.25,  -0.75);
  mybezier.anchors[1][1] = makePoint(-0.25,  -0.25,   0.50);
  mybezier.anchors[1][2] = makePoint( 0.25,  -0.25,   0.50);
  mybezier.anchors[1][3] = makePoint( 0.75,  -0.25,  -0.75);
  mybezier.anchors[2][0] = makePoint(-0.75,   0.25,   0.00);
  mybezier.anchors[2][1] = makePoint(-0.25,   0.25,  -0.50);
  mybezier.anchors[2][2] = makePoint( 0.25,   0.25,  -0.50);
  mybezier.anchors[2][3] = makePoint( 0.75,   0.25,   0.00);
  mybezier.anchors[3][0] = makePoint(-0.75,   0.75,  -0.50);
  mybezier.anchors[3][1] = makePoint(-0.25,   0.75,  -1.00);
  mybezier.anchors[3][2] = makePoint( 0.25,   0.75,  -1.00);
  mybezier.anchors[3][3] = makePoint( 0.75,   0.75,  -0.50);
  mybezier.dlBPatch = NULL;
}
```

Это процедура загрузки одной картинки. Организовав цикл, Вы можете загрузить несколько картинок.

```
// Загрузить картинку и конвертировать ее в текстуру
BOOL LoadGLTexture(GLuint *texPntr, char* name)
{
  BOOL success = FALSE;
  AUX_RGBImageRec *TextureImage = NULL;
  glGenTextures(1, texPntr);          // Генерировать 1 текстуру

  FILE* test=NULL;
  TextureImage = NULL;

  test = fopen(name, "r");            // Существует ли файл?
```

```
  if (test != NULL) {                // Если да
    fclose(test);                    // Закрыть файл
    TextureImage = auxDIBImageLoad(name); // И загрузить текстуру
  }
  if (TextureImage != NULL) {        // Если загружена
    success = TRUE;

   // Обычная генерация текстура используя данные из картинки
    glBindTexture(GL_TEXTURE_2D, *texPntr);
    glTexImage2D(GL_TEXTURE_2D, 0, 3, TextureImage->sizeX, TextureImage->sizeY,
     0, GL_RGB, GL_UNSIGNED_BYTE, TextureImage->data);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
  }
  if (TextureImage->data)
    free(TextureImage->data);
  return success;
}
```

Инициализация фрагмента непосредственно здесь. Вы делаете это всякий раз, когда создаете фрагмент. Снова, это хорошее место использовать C++ (Безье класс).

```
int InitGL(GLvoid)               // Настройки OpenGL
{
  glEnable(GL_TEXTURE_2D);       // Разрешить наложение текстуры
  glShadeModel(GL_SMOOTH);       // Разрешить сглаживание
  glClearColor(0.05f, 0.05f, 0.05f, 0.5f); // Фон черный
  glClearDepth(1.0f);            // Настройки буфера глубины
  glEnable(GL_DEPTH_TEST);       // Разрешаем тест глубины
  glDepthFunc(GL_LEQUAL);        // Тип теста глубины
  glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // Улучшенные вычисления перспективы
  initBezier();                  // Инициализация контрольной сетки Безье ( НОВОЕ )
  LoadGLTexture(&(mybezier.texture), "./Data/NeHe.bmp"); // Загрузка текстуры ( НОВОЕ )
  mybezier.dlBPatch = genBezier(mybezier, divs); // Создание фрагмента ( НОВОЕ )
  return TRUE;
}
```

При отрисовке сцены, вначале отображаем список Безье. Затем (если контур включен) рисуются линии, соединяющие контрольные точки. Вы можете переключать этот режим при помощи ПРОБЕЛА.

```
int DrawGLScene(GLvoid)  {           // Здесь рисуем
  int i, j;
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);    // Очистка экрана и буфера глубины
  glLoadIdentity();              // Сброс текущей матрицы вида модели
  glTranslatef(0.0f,0.0f,-4.0f);         // Сдвиг налево на 1.5 единицы и вглубь экрана на 6.0
  glRotatef(-75.0f,1.0f,0.0f,0.0f);
  glRotatef(rotz,0.0f,0.0f,1.0f);        // Вращение по оси Z
  glCallList(mybezier.dlBPatch);// Вызов списка Безье
                          // Это необходимо только в том случае, когда фрагмент изменился
  if (showCPoints) {             // Если отрисовка сетки включена
    glDisable(GL_TEXTURE_2D);
    glColor3f(1.0f,0.0f,0.0f);
    for(i=0;i<4;i++) {           // Нарисовать горизонтальную линию
      glBegin(GL_LINE_STRIP);
      for(j=0;j<4;j++)
        glVertex3d(mybezier.anchors[i][j].x, mybezier.anchors[i][j].y, mybezier.anchors[i][j].z);
      glEnd();
    }
    for(i=0;i<4;i++) {           // Нарисовать вертикальную линию
      glBegin(GL_LINE_STRIP);
      for(j=0;j<4;j++)
        glVertex3d(mybezier.anchors[j][i].x, mybezier.anchors[j][i].y, mybezier.anchors[j][i].z);
      glEnd();
```

```
    }
    glColor3f(1.0f,1.0f,1.0f);
    glEnable(GL_TEXTURE_2D);
  }
 return TRUE;
}
```

В этой функции код модифицирован, чтобы сделать его более совместимым. Эти изменения не имеют прямого отношения к кривым Безье, но при этом решается проблема с возвратом разрешения экрана после работы в полноэкранном режиме, которая имеется с некоторыми видеокартами (включая мою, дерьмовую старую ATI Rage PRO, и некоторыми другими). Я надеюсь, вы будете использовать эти модификации, так как они позволят вашим крутым программам работать должным образом. Делая эти модификации, проверьте, что Dmsaved определена и инициализирована, как это отмечено в функции CreateGLWindow().

```
GLvoid KillGLWindow(GLvoid)    // Убить окно
{
  if (fullscreen)              // Мы в полноэкранном режиме?
  {
    if (!ChangeDisplaySettings(NULL,CDS_TEST)) {// Если это не работает ( НОВОЕ )
      // Сделать это все равно (чтобы получить значения из системного реестра) (НОВОЕ)
      ChangeDisplaySettings(NULL,CDS_RESET);
      ChangeDisplaySettings(&DMsaved,CDS_RESET);// Изменить его на сохраненные настройки (НОВОЕ)
    } else {
      ChangeDisplaySettings(NULL,CDS_RESET);    // Если это работает продолжаем (НОВОЕ)
    }
    ShowCursor(TRUE);          // Показать курсор мыши
  }
  if (hRC)                     // Мы имеем контекст визуализации?
  {
    if (!wglMakeCurrent(NULL,NULL)) // Можно освободить контексты DC и RC?
    {
      MessageBox(NULL,"Release Of DC And RC Failed.","SHUTDOWN ERROR",
                 MB_OK | MB_ICONINFORMATION);
    }
    if (!wglDeleteContext(hRC))     // Действительно ли мы можем удалить RC?
    {
      MessageBox(NULL,"Release Rendering Context Failed.","SHUTDOWN ERROR",
                 MB_OK | MB_ICONINFORMATION);
    }
    hRC=NULL;                  // Установить RC в NULL
  }
  if (hDC && !ReleaseDC(hWnd,hDC))  // Действительно ли мы можем удалить DC
  {
    MessageBox(NULL,"Release Device Context Failed.","SHUTDOWN ERROR",
               MB_OK | MB_ICONINFORMATION);
    hDC=NULL;                  // Установить DC в NULL
  }
  if (hWnd && !DestroyWindow(hWnd)) // Действительно ли мы можем удалить окно?
  {
    MessageBox(NULL,"Could Not Release hWnd.","SHUTDOWN ERROR",
               MB_OK | MB_ICONINFORMATION);
    hWnd=NULL;                 // Set hWnd To NULL
  }
  // Действительно ли мы можем отменить регистрацию класса
  if (!UnregisterClass("OpenGL",hInstance))
  {
    MessageBox(NULL,"Could Not Unregister Class.","SHUTDOWN ERROR",
               MB_OK | MB_ICONINFORMATION);
    hInstance=NULL;            // Установить hInstance в NULL
  }
}
```

В функции CreateGLWindow только добавлен вызов EnumDisplaySettings, чтобы сохранить параметры настройки дисплея.

```
BOOL CreateGLWindow(char* title, int width, int height, int bits, bool fullscreenflag)
{
  . . . Код вырезан, чтобы уменьшить размер урока . . .

  wc.lpszClassName  = "OpenGL"; // Имя класса
  // Сохранить текущие настройки дисплея (НОВОЕ)
  EnumDisplaySettings(NULL, ENUM_CURRENT_SETTINGS, &DMsaved);
  if (fullscreen)          // Попробовать перейти в полноэкранный режим?
  {

  . . . Код вырезан, чтобы уменьшить размер урока . . .

 return TRUE;              // Успех
}
```

Здесь добавлен код для вращения фрагмента, уменьшения/улучшения разрешения, и переключения режима контура фрагмента.

```
int WINAPI WinMain( HINSTANCE  hInstance,     // Экземпляр
                    HINSTANCE  hPrevInstance, // Предыдущий экземпляр
                    LPSTR      lpCmdLine,     // Параметры командной строки
                    int        nCmdShow)      // Состояние отображения окна
{
. . . Код вырезан, чтобы уменьшить размер урока . . .

      SwapBuffers(hDC);      // // Переключаем буферы (Двойная буферизация)
    }
    if (keys[VK_LEFT])  rotz -= 0.8f;    // Вращение влево ( НОВОЕ )
    if (keys[VK_RIGHT]) rotz += 0.8f;    // Вращение вправо
    if (keys[VK_UP]) {        // Увеличить разрешение
      divs++;
      mybezier.dlBPatch = genBezier(mybezier, divs);  // Обновить фрагмент
      keys[VK_UP] = FALSE;
    }
    if (keys[VK_DOWN] && divs > 1) {    // Уменьшить разрешения
      divs--;
      mybezier.dlBPatch = genBezier(mybezier, divs);  // Обновить фрагмент
      keys[VK_DOWN] = FALSE;
    }
    if (keys[VK_SPACE]) {        // ПРОБЕЛ переключает showCPoints
      showCPoints = !showCPoints;
      keys[VK_SPACE] = FALSE;
    }
    if (keys[VK_F1])        // Если F1 нажата?
    {

  . . . Код вырезан, чтобы уменьшить размер урока . . .

  return (msg.wParam);            // Выходим из программы
}
```

Надеюсь, что этот урок осветил эту тему, и теперь вы полюбили кривые Безье, так же как и я. Если Вам понравился этот урок, я напишу еще урок о NURBS кривых. Пожалуйста, свяжитесь со мной по электронной почте и сообщите, что Вы думаете о моем уроке.

Об авторе: Дэвиду Никделу 18 лет и он учится в Bartow Senior High School. На данный момент он изучает кривые поверхности в 3D графике и игру на OpenGL под названием Blazing Sands. Его хобби – это программирование и футбол. Если все будет удачно, то в следующем году он поступит в Georgia Tech.

Урок 29. Блиттер-функция и чтение не обработанных текстур.

Blitter Function, RAW Texture Loading

Этот урок вначале был написан Андреасом Лоффлером. Он также написал весь оригинальный код HTML для этого урока. Через несколько дней после этого Роб Флетчер выслал мне Irix версию этого урока. Он переписал большинство кода. Так что я портировал Irix/GLUT код Роба в Visual C++/Win32. Затем я модифицировал код цикла обработки сообщений, и код работы в полноэкранном режиме. Когда программа свернута, она не будет использовать процессор. Большинство проблем возникавших при переключениях полноэкранного режима исчезнет (таких как пропадание изображения на экране).

Так что урок Андреаса теперь лучше чем когда-либо. После того как был модифицирован код, так же изменился и код HTML. Огромное спасибо Андреасу за то, что он начал эту работу! Спасибо Робу за модификации!

Давайте начнем... Мы определяем структуру для хранения информации о режиме экрана DMsaved. Мы будем использовать эту структуру, для того чтобы сохранить информацию о видеорежиме, прежде чем мы переключимся в полноэкранный режим. Больше об этот позже! Заметьте, что мы определяем переменную для одной текстуры (texture [1]).

```
#include <windows.h> // заголовочный файл для Windows
#include <stdio.h>   // заголовочный файл для стандартного ввода/вывода
#include <gl\gl.h>   // заголовочный файл для библиотеки OpenGL32
#include <gl\glu.h>  // заголовочный файл для библиотеки GLu32

HDC       hDC=NULL;       // Контекст устройства
HGLRC     hRC=NULL;       // Контекст рендеринга
HWND      hWnd=NULL;      // Дескриптор окна
HINSTANCE hInstance;      // Экземпляр приложения

bool      keys[256];      // Массив для работы с клавиатурой
bool      active=TRUE;    // Флаг активности приложения
bool      fullscreen=TRUE; // Флаг полноэкранного режима

DEVMODE    DMsaved;       // Сохранить предыдущие настройки экрана (НОВОЕ)

GLfloat    xrot;              // X вращение
GLfloat    yrot;              // Y вращение
GLfloat    zrot;              // Z вращение
GLuint     texture[1];            // имя 1 текстуры
```

Теперь об интересном. Мы создаем структуру по имени TEXTURE_IMAGE. Структура содержит информацию о ширине изображения, высоте, и формате (число байт на пиксель). Поле структуры data - указатель на данные изображения.

```
typedef struct Texture_Image
{
  int width;              // Ширина
  int height;             // Высота
  int format;             // Байт на пиксель
  unsigned char *data;      // данные текстуры
} TEXTURE_IMAGE;
```

Мы тогда создаем указатель по имени P_TEXTURE_IMAGE на структуру TEXTURE_IMAGE. Переменные t1 и t2 имеют тип P_TEXTURE_IMAGE, где P_TEXTURE_IMAGE - переопределенный тип указателя на TEXTURE_IMAGE.

```
typedef TEXTURE_IMAGE *P_TEXTURE_IMAGE; // Указатель на структуру изображения

P_TEXTURE_IMAGE t1;             // Указатель на данные текстуры
P_TEXTURE_IMAGE t2;             // Указатель на данные текстуры

LRESULT  CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM); // Объявление WndProc
```

Код ниже выделяет память для текстуры. Когда мы вызовем этот код, мы передадим ему ширину, высоту и число байт на пиксель. Переменная ti – указатель на данные TEXTURE_IMAGE. Переменная c – указатель на данные типа unsigned char.

```
// Выделить память под структуру изображения и данных изображения
P_TEXTURE_IMAGE AllocateTextureBuffer( GLint w, GLint h, GLint f)
{
  P_TEXTURE_IMAGE ti=NULL;              // Указатель на структуру изображения
  unsigned char *c=NULL;               // Указатель на блок памяти для изображения
```

Далее, где мы распределяем память для нашей структуры изображения. Если все отлично, то ti укажет на распределенную память.

После распределения памяти, и проверки, мы можем заполнять структуру атрибутами изображения. Сначала мы задаем ширину (w), затем высоту (h) и наконец формат (f). Имейте в виду, что формат – число байт на пиксель.

```
  ti = (P_TEXTURE_IMAGE)malloc(sizeof(TEXTURE_IMAGE)); // Дайте пожалуйста одну структуру для изображения

  if( ti != NULL ) {
    ti->width  = w;   // Ширина
    ti->height = h;   // Высота
    ti->format = f;   // Формат
```

Теперь мы должны распределить память для данных изображения. Для вычисления размера нужного куска памяти надо просто умножить ширину изображения (w) на высоту изображения (h) и на формат (f – число байт на пиксель).

```
    c = (unsigned char *)malloc( w * h * f);
```

Делаем проверку, что все прошло удачно. Если значение в c - не равно NULL, мы присваиваем полю data значение переменной c.

Если память не выделена, то выскочит сообщение об ошибке и процедура вернет NULL.

```
    if ( c != NULL ) {
      ti->data = c;
    }
    else {
      MessageBox(NULL,"Не могу выделить память для буфера текстуры","BUFFER ERROR",MB_OK |
MB_ICONINFORMATION);
      return NULL;
    }
  }
```

Если невозможно выделить память для структуры изображения, то так же выскочит сообщение об ошибке и процедура вернет NULL.

Если нет проблем, мы возвращаем ti - указатель на готовую структуру изображения. Ух... надеюсь, что все это понятно и имеет смысл.

```
  else
  {
    MessageBox(NULL,"Не могу выделить память для структуры изображения","IMAGE STRUCTURE
ERROR",MB_OK | MB_ICONINFORMATION);
    return NULL;
  }
  return ti; // Вернем указатель на структуру изображения
}
```

Когда приходит время, то надо освободить память. Для этого служит код ниже, в котором происходит освобождение буфера текстуры и затем сброс памяти для структуры изображения. Переменная t - указатель на структуру данных TEXTURE_IMAGE, которую мы хотим освободить.

```
// Освободить память
void DeallocateTexture( P_TEXTURE_IMAGE t )
{
 if(t)
 {
  if(t->data)
  {
   free(t->data); // Освободить буфер
  }

  free(t); // Сброс текстуры
 }
}
```

Теперь мы будем читать наше RAW изображение (несжатые и не форматированные данные, так называемые “сырые” данные). Мы передаем имя файла и указатель на структуру изображения, в которую мы хотим загрузить изображение. Мы определяем несколько вспомогательных переменных, и затем вычисляем размер строки, умножая ширину нашего изображения на формат (число байт на пиксель). Если бы изображение было 256 пикселей в ширину и 4 байта на пиксель, то ширина строки была бы 1024 байта. Мы запоминаем ширину строки в stride.

Мы обнуляем указатель (p), и затем пытаемся открыть файл.

```
// Чтение RAW файла в готовый буфер.
// Поворот изображения сверху вниз. Возвращает 9, если сбой, или число прочитанных байт.
int ReadTextureData ( char *filename, P_TEXTURE_IMAGE buffer)
{
 FILE *f;
 int i,j,k,done=0;
 int stride = buffer->width * buffer->format; // Размер строки
 unsigned char *p = NULL;

 f = fopen(filename, "rb");  // Открыть файл
 if( f != NULL )             // Если файл существует
 {
```

Если файл существует, мы запускаем три цикла, чтобы прочитать нашу текстуру. Начинаем с нижней строки и сдвигаемся вверх по одной строке. Так как цикл начинается снизу, то зеркальный поворот будет выполнен правильно. RAW изображения сохраняются инвертированными вниз. Необходимо установить указатель в надлежащее место в буфере изображения. Каждый раз, когда мы сдвигаемся на следующую строку(i уменьшается) мы устанавливаем указатель на начало новой строки. Указатель data – показывает начало изображения, и для того чтобы перейти на следующую строку изображения, надо умножить номер текущей строки i на размер строки stride.

Во втором цикле j смещается слева (0) направо (ширина строки в пикселях, но не в байтах).

```
  for( i = buffer->height-1; i >= 0 ; i-- ) // Цикл по высоте (снизу вверх)
  {
   p = buffer->data + (i * stride );
   for ( j = 0; j < buffer->width ; j++ ) // Цикл по ширине
   {
```

В цикле по k читаем пиксель побайтно. Если формат (байт на пиксель) - 4,то k меняется от 0 до 2, где 2 равняется число байт на пиксель минус один (format -1). Так как большинство необработанных изображений не имеют альфа канала, а нам он нужен, то это та причина, по которой мы вычитаем единицу. Мы будем заполнять каждый 4-ый байт нашим значением альфа канала.

Заметьте, что в цикле, мы также увеличиваем указатель (p) и переменную done. Больше об этом позже.

В строке внутри цикла происходит чтение символа из нашего файла и запоминание его в буфере текстуры согласно текущего положения указателя. Если наше изображение имеет 4 байта на пиксель, первые 3 байта будут читаться из RAW файла (format-1), и 4-ый байт будет вручную установлен в 255. После того, как мы задаем 4-ый байт равным 255, мы сдвигаем указателя на один байт вперед, чтобы наш 4-ый байт не был перезаписан следующим байтом из файла.

После того, как все байты прочитались в пиксель, и прочитались все пиксели строке, и прочитались все строки, то все сделано! Мы можем закрывать файл.

```
      for ( k = 0 ; k < buffer->format-1 ; k++, p++, done++ )
      {
        *p = fgetc(f); // Чтение значения из файла и сохранение его в памяти
      }
      *p = 255; p++; // Установить 255 в альфа канал и увеличить указатель
    }
   }
   fclose(f); // Закрыть файл
  }
```

Если файл не открыт, то выскочит окно сообщения, о том, что файл не был открыт.

В конце мы возвращаем done. Если файл не был открыт, то done равно 0. Если все пошло удачно, то done равно числу прочитанных байтов из файла. Помните, мы увеличивались done, каждый раз, когда читали байт в цикле выше (цикл по k).

```
  else // Иначе
 {
    MessageBox(NULL,"Невозможно открыть файл изображения ","IMAGE ERROR",MB_OK |
MB_ICONINFORMATION);
  }
  return done; // Возвращает число прочитанных байт
}
```

This shouldn't need explaining. By now you should know how to build a texture. tex is the pointer to the TEXTURE_IMAGE structure that we want to use. We build a linear filtered texture. In this example, we're building mipmaps (smoother looking). We pass the width, height and data just like we would if we were using glaux, but this time we get the information from the selected TEXTURE_IMAGE structure.

Далее код не нуждается в пояснениях. Вы должны знать, как создать текстуру. Переменная tex - указатель на структуру TEXTURE_IMAGE, которую мы хотим использовать. Мы создаем текстуру с линейной фильтрацией. В этом примере, мы задаем мипмапы (для улучшения изображения). Мы передаем ширину, высоту и данные изображения точно так же как, если мы использовали glaux, но на сей раз, мы получаем информацию из структуры TEXTURE_IMAGE.

```
void BuildTexture (P_TEXTURE_IMAGE tex)
{
  glGenTextures(1, &texture[0]);
  glBindTexture(GL_TEXTURE_2D, texture[0]);
  glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
  gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGB, tex->width, tex->height, GL_RGBA, GL_UNSIGNED_BYTE, tex-
>data);
}
```

Теперь код блиттера (функция пересылки изображения). Код блиттера очень мощный. Он позволяет Вам копировать любую прямоугольную часть текстуру (src) в приемную текстуру (dst). Вы можете комбинировать так много текстур, как Вы хотите, Вы можете задать альфа значение для смешивания.

Переменная src - структура TEXTURE_IMAGE используется как исходное изображение. Переменная dst - структура TEXTURE_IMAGE используется как приемное изображение. Переменная src_xstart – задает начало копирования данных из исходного изображения по оси X. Переменная src_ystart - задает начало копирования данных из исходного изображения по оси Y. Переменная src_width – задает ширину в пикселях области, которую Вы хотите копировать из исходного изображения. Переменная src_height - высота в пикселях области, которую Вы хотите скопировать из исходного изображения. Переменные dst_xstart и dst_ystart – задают место, куда вы хотите скопировать пиксели из исходного изображения в приемное изображение. Если blend – равно 1, то два изображения будут смешаны. Значение переменной alpha задает степень прозрачности изображения. Если значение 0, то ничего не будет скопировано, если alpha равно 255, то исходное изображение полностью перекроет приемное.

Затем задаем все вспомогательные переменные циклов, наряду с указателями для нашего исходного изображения (s) и приемного изображения (d). Проверяем диапазон значения alpha. Если есть выход за диапазон, то исправляем его. Тоже и для blend.

```
void Blit( P_TEXTURE_IMAGE src, P_TEXTURE_IMAGE dst, int src_xstart, int src_ystart, int src_width, int src_height,
    int dst_xstart, int dst_ystart, int blend, int alpha)
{
 int i,j,k;
 unsigned char *s, *d; // Исходное и приемное

 // Ограничиваем Alpha
 if( alpha > 255 ) alpha = 255;
 if( alpha < 0 ) alpha = 0;

 // Проверка флага Blend
 if( blend < 0 ) blend = 0;
 if( blend > 1 ) blend = 1;
```

Теперь мы должны задать значение указателей. Приемный указатель - адрес приемных данных плюс смещение начальной строки в приемном изображении, которое равно произведению начальной строки изображения (dst_ystart) на ширину приемного изображения и на формат изображения.

Тоже делается и для исходного изображения. Исходный указатель - адрес исходных данных плюс смещение начальной строки в исходном изображении (src_ystart), которое равно произведению начальной строки изображения (src_ystart) на ширину исходного изображения и на формат изображения.

Цикл по i от 0 до src_height, копирует все строки из исходного изображения.

```
 d = dst->data + (dst_ystart * dst->width * dst->format);
 s = src->data + (src_ystart * src->width * src->format);

 for (i = 0 ; i < src_height ; i++ ) // Цикл по высоте
 {
```

Мы уже задали значения указателей источника и приемника на нужные строки в каждом изображении. Теперь мы должны двигаться в правильном направлении слева направо в каждом изображении прежде, чем мы запустим блиттинг данных. Мы увеличиваем указатель на исходное изображение (s) на значение src_xstart, которое является началом по оси X исходных данных изображения умноженное на число байт на пиксель. Это перемещает указатель источника (s) на начальный пиксель по оси X (слева направо) в исходном изображении.

Тоже и для приемного изображения. Мы увеличиваем указатель на приемное изображение (d) на значение dst_xstart, которое является началом по оси X приемных данных изображения умноженное на число байт на пиксель. Это перемещает указатель источника (d) на начальный пиксель по оси X (слева направо) в приемном изображении.

После того, как мы вычислили, где в памяти мы хотим захватить наши пиксели от (s) и куда мы хотим переместить их в (d), мы запускаем цикл по j. Мы будем использовать цикл по j, чтобы перемещаться слева направо через исходное изображение.

```
  s = s + (src_xstart * src->format); // Сдвинемся на начало исходных данных в строке
  d = d + (dst_xstart * dst->format); // Сдвинемся на начало приемных данных в строке
  for (j = 0 ; j < src_width ; j++ )  // Цикл по ширине
  {
```

Цикл по k используется для обработки всех байт в пикселе. Заметьте, что при увеличении k указатели для источника и приемника также увеличиваются.

Внутри цикла мы проверяем, включено ли смешение. Если blend равно 1, то значит надо выполнить смешение пикселей, и мы делаем несколько причудливых операций, чтобы вычислить цвет наших смешанных пикселей. Значение приемника (d) будет равно исходному значению (s), умноженному на альфа-значение плюс текущее значение приемника (d) умноженное на 255 минус альфа-значение. Оператор сдвига (>>8) сдвигает значение в диапазон 0-255.

Если смешивание выключено (0), то мы копируем данные из исходного изображения непосредственно в приемное изображение. При этом смешивание не делается, и альфа-значение игнорируется.

```
    for( k = 0 ; k < src->format ; k++, d++, s++)
    {
     if (blend) // Смешивание
      // Умножить данные Src haalpha плюс данные Dst *(255-alpha)
      // Сохранить значение в диапазоне 0-255 с помощью сдвига >> 8
      *d = ( (*s * alpha) + (*d * (255-alpha)) ) >> 8;
     else
      *d = *s; // нет смешения, просто копируем
    }
   }
   d = d + (dst->width - (src_width + dst_xstart))*dst->format; // Добавить конец строки
   s = s + (src->width - (src_width + src_xstart))*src->format; // Добавить конец строки
  }
}
```

Код InitGL() был изменен. Весь код ниже новый. Вначале выделяем память для изображения размером 256x256x4 байт. Переменная t1 укажет на выделенную память, если все пошло хорошо.

После выделения памяти для нашего изображения, мы пытаемся загрузить изображение. Мы передаем функции ReadTextureData() имя файла, который мы желаем открыть, вместе с указателем на нашу структуру изображения (t1).

Если изображение не загружено, выскочит окно сообщения на экран, чтобы дать понять пользователю, что была проблема, при загрузке текстуру.

Затем мы делаем тоже самое и для t2. Мы распределяем память, и пытаемся прочитать второе изображение. Если что-нибудь пойдет не так, как надо, выскочит окно сообщения.

```
int InitGL(GLvoid) // Вызов происходит после создания окна
{
 t1 = AllocateTextureBuffer( 256, 256, 4 ); // Взять структуру изображения
 if (ReadTextureData("Data/Monitor.raw",t1)==0) // Заполнить структуру данными
 { // Ничего не прочитали?
  MessageBox(NULL,"Не могу прочитать 'Monitor.raw'","TEXTURE ERROR",MB_OK | MB_ICONINFORMATION);
  return FALSE;
 }

 t2 = AllocateTextureBuffer( 256, 256, 4 ); // Второе изображение
 if (ReadTextureData("Data/GL.raw",t2)==0) // Заполнить структуру данными
 { // Ничего не прочитали?
  MessageBox(NULL," Не могу прочитать 'GL.raw'","TEXTURE ERROR",MB_OK | MB_ICONINFORMATION);
  return FALSE;
 }
```

Двигаемся дальше. Теперь пришло время использовать нашу функцию Blit(), чтобы объединить два изображения в одно.

Мы передаем Blit() два указателя t2 и t1, оба указатели на структуру TEXTURE_IMAGE (t2 - второе изображение, t1 - первое изображение).

Then we have to tell blit where to start grabbing data from on the source image. If you load the source image into Adobe Photoshop or any other program capable of loading .RAW images you will see that the entire image is blank except for the top right corner. The top right has a picture of the ball with GL written on it. The bottom left corner of the image is 0,0. The top right of the image is the width of the image-1 (255), the height of the image-1 (255). Knowing that we only want to copy 1/4 of the src image (top right), we tell Blit() to start grabbing from 127,127 (center of our source image).

Затем надо сообщить Blit(), где начать захватывать данные из исходного изображения. Если Вы загрузите исходное изображение в Adobe Photoshop, или любую другую программу, которая способна читать RAW изображения, то Вы увидите, что все изображение пустое, кроме правого верхнего угла. В нем есть изображение шара с надписью GL.

Левый угол изображения - 0,0. Правый верхний угол равен ширине изображения-1 (255) и высоте изображения-1 (255). Зная, что мы хотим скопировать 1/4 часть исходного изображения (право верх), мы сообщаем Blit() начать захват с 127,127 (центр нашего исходного изображения).

Затем мы сообщаем Blit(), сколько пикселей мы хотим копировать из нашей исходной точки вправо и вверх. Мы хотим захватить четвертую часть изображения. Наше изображение равно 256x256 пикселей, четверть его равна 128x128 пикселей. Вся исходная информация есть. Функция Blit() теперь знает, что надо скопировать от 127 по оси X до 127+128 (255) по оси X, и от 127 по оси Y до 127+128 (255) по оси Y.

Итак, Blit() знает, что копировать, и где получить данные для этого, но она не знает, куда поместить эти данные. Мы хотим вывести шар с GL, в середине нашего изображения монитора. Вы находите центр приемного изображения (256x256), который равен 128x128, и вычитаете половину ширины и высоты исходного изображения (128x128), что равно 64x64. Так (128-64) x (128-64) дает нам начальную точку 64,64.

И последнее, мы сообщаем нашей блиттер-функции, что мы хотим смешать два изображения (единица означает смешивание, а ноль означает не смешивание), и насколько смешать изображения. Если последнее значение - 0, то мы не смешиваем изображения. Если мы используем значение 127, то изображения смешиваются вместе в 50% пропорции, и если Вы используете 255, то изображение, которое Вы копируете, будет полностью прозрачно, и не будет видно вообще.

Пиксели копируются из изображения 2 (t2) в изображение 1 (t1). Смешанное изображение будет сохранено в t1.

```
  // Смешивание изображений: исходное, приемное изображение
  // исходные X & Y, исходные Width & Height, приемные X & Y, флаг смешивания
  // альфа-значение
  Blit(t2,t1,127,127,128,128,64,64,1,127); // блиттер
```

После того, как мы смешали два изображения (t1 и t2), мы создаем текстуру из объединенных изображений (t1). После того, как текстура была создана, мы можем освободить память из двух структур TEXTURE_IMAGE. Остальная часть кода уже стала привычным стандартом. Мы разрешаем наложение текстуры, тест глубины, и т.д.

```
  BuildTexture (t1);       // Загрузка текстуры в память
  DeallocateTexture( t1 ); // Очистка памяти
  DeallocateTexture( t2 );

  glEnable(GL_TEXTURE_2D); // Разрешить наложение текстуры
  glShadeModel(GL_SMOOTH); // Разрешить плавное сглаживание
  glClearColor(0.0f, 0.0f, 0.0f, 0.0f); // Фоновый цвет черный
  glClearDepth(1.0);       // Очистка буфера глубины
  glEnable(GL_DEPTH_TEST); // Тест глубины
  glDepthFunc(GL_LESS);    // Тп теста
  return TRUE;
}
```

Я не буду подробно пояснять код ниже. Мы сдвигаемся на 5 единиц в глубь экрана, выбираем текстуру, и рисуем текстурированный куб. Заметьте, что оба изображения теперь смешаны в одно изображение. Мы не выводим обе текстуры на кубе. Код блиттер делает это за нас.

```
GLvoid DrawGLScene(GLvoid)
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Очистка экрана и буфера глубины
  glLoadIdentity(); // Сброс просмотра
  glTranslatef(0.0f,0.0f,-5.0f);
  glRotatef(xrot,1.0f,0.0f,0.0f);
  glRotatef(yrot,0.0f,1.0f,0.0f);
  glRotatef(zrot,0.0f,0.0f,1.0f);
  glBindTexture(GL_TEXTURE_2D, texture[0]);
  glBegin(GL_QUADS);
    // Передняя грань
    glNormal3f( 0.0f, 0.0f, 1.0f);
    glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f,  1.0f,  1.0f);
    glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f,  1.0f,  1.0f);
```

```
    glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f,  1.0f);
    glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f,  1.0f);
    // Задняя грань
    glNormal3f( 0.0f, 0.0f,-1.0f);
    glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f,  1.0f, -1.0f);
    glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f,  1.0f, -1.0f);
    glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
    glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
    // Верхняя грань
    glNormal3f( 0.0f, 1.0f, 0.0f);
    glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f,  1.0f, -1.0f);
    glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f,  1.0f, -1.0f);
    glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f,  1.0f,  1.0f);
    glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f,  1.0f,  1.0f);
    // Нижняя грань
    glNormal3f( 0.0f,-1.0f, 0.0f);
    glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f,  1.0f);
    glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f,  1.0f);
    glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
    glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
    // Правая грань
    glNormal3f( 1.0f, 0.0f, 0.0f);
    glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
    glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f,  1.0f, -1.0f);
    glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f,  1.0f,  1.0f);
    glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f,  1.0f);
    // Левая грань
    glNormal3f(-1.0f, 0.0f, 0.0f);
    glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
    glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f,  1.0f);
    glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f,  1.0f,  1.0f);
    glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f,  1.0f, -1.0f);
  glEnd();

  xrot+=0.3f;
  yrot+=0.2f;
  zrot+=0.4f;
}
```

В код KillGLWindow() внесено ряд изменений. Во-первых, код переключения из полноэкранного режима назад к видеорежиму вашего рабочего стола теперь вначале KillGLWindow(). Если пользователь запустил программу в полноэкранном режиме, вначале, перед тем как уничтожить окно - пробуем переключиться назад в видеорежим рабочего стола. Если быстрый вариант не сработает, мы сбрасываем экран, используя информацию, сохраненную в DMsaved. Это должно восстановить оригинальный видеорежим.

```
GLvoid KillGLWindow(GLvoid) // Выполняет при уничтожении окна
{
  if (fullscreen)                // Мы в полноэкранном?
  {
    if (!ChangeDisplaySettings(NULL,CDS_TEST)) { // Если это не работает
      ChangeDisplaySettings(NULL,CDS_RESET); // Еще раз (чтобы взять значения из реестра)
      ChangeDisplaySettings(&DMsaved,CDS_RESET); // Изменить разрешение используя
                                        // сохраненные данные
    }
    else // все прошло
    {
      ChangeDisplaySettings(NULL,CDS_RESET); // Ничего не делать
    }

    ShowCursor(TRUE); // Показать курсор мыши
  }
```

```
  if (hRC)                          // Существует контекст рендеринга?
  {
     if (!wglMakeCurrent(NULL,NULL))   // Можно ли освободить DC и RC контексты?
     {
        MessageBox(NULL,"Release Of DC And RC Failed.","SHUTDOWN ERROR",
                                    MB_OK | MB_ICONINFORMATION);
  }
    if (!wglDeleteContext(hRC))         // Можно ли уничтожить RC?
  {
        MessageBox(NULL,"Release Rendering Context Failed.",
             "SHUTDOWN ERROR",MB_OK | MB_ICONINFORMATION);
  }
 hRC=NULL;                          // Установим RC в NULL
  }

  if (hDC && !ReleaseDC(hWnd,hDC))      // Можно ли уничтожить DC?
  {
     MessageBox(NULL,"Release Device Context Failed.","SHUTDOWN ERROR",
                                  MB_OK | MB_ICONINFORMATION);
  hDC=NULL;                         // Установим DC в NULL
  }
  if (hWnd && !DestroyWindow(hWnd))     // Можно ли уничтожить окно?
  {
     MessageBox(NULL,"Could Not Release hWnd.","SHUTDOWN ERROR",MB_OK |
                                     MB_ICONINFORMATION);
  hWnd=NULL;                         // Устновим hWnd в NULL
  }
  if (!UnregisterClass("OpenGL",hInstance)) // Можно ли уничтожить класс?
  {
     MessageBox(NULL,"Could Not Unregister Class.","SHUTDOWN ERROR",MB_OK |
                                       MB_ICONINFORMATION);
  hInstance=NULL;                    // Устанавливаем hInstance в NULL
  }
     KillFont();                    // Уничтожаем шрифт
}
```

Я сделал некоторые изменения и в CreateGLWindow. Эти изменения помогут устранить множество проблем, которые возникают при переключениях видеорежимов. Я включил первую часть CreateGLWindow(), чтобы вам было легче разобраться.

```
BOOL CreateGLWindow(char* title, int width, int height, int bits, bool fullscreenflag)
{
 GLuint    PixelFormat; // Для хранения результатов поиска
 WNDCLASS  wc;          // Структура класса окна
 DWORD    dwExStyle;    // Расширенный стиль окна
 DWORD    dwStyle;      // Стиль окна

 fullscreen=fullscreenflag; // Глобальный флаг

 hInstance    = GetModuleHandle(NULL);
 wc.style    = CS_HREDRAW | CS_VREDRAW | CS_OWNDC;
 wc.lpfnWndProc    = (WNDPROC) WndProc; // Обработчик
 wc.cbClsExtra    = 0;
 wc.cbWndExtra    = 0;
 wc.hInstance    = hInstance;
 wc.hIcon    = LoadIcon(NULL, IDI_WINLOGO);
 wc.hCursor    = LoadCursor(NULL, IDC_ARROW);
 wc.hbrBackground  = NULL;
 wc.lpszMenuName    = NULL;
 wc.lpszClassName  = "OpenGL";
```

The big change here is that we now save the current desktop resolution, bit depth, etc. before we switch to fullscreen mode. That way when we exit the program, we can set everything back exactly how it was. The first line below copies the display settings into the DMsaved Device Mode structure. Nothing else has changed, just one new line of code.

Здесь есть существенное изменение: мы сохраняем текущие настройки видеорежима прежде, чем переключить видеорежим. Поэтому мы можем восстановить все обратно, когда выходим из программы. В первой строке ниже копируются параметры настройки дисплея в структуру режима дисплея DMsaved. Далее все тоже самое.

```
  // Сохранить настройки видеорежима (НОВОЕ)
  EnumDisplaySettings(NULL, ENUM_CURRENT_SETTINGS, &DMsaved);

 if (fullscreen) // Попробуем перейти в полноэкранный режим?
  {
   DEVMODE dmScreenSettings; // Режим работы
   memset(&dmScreenSettings,0,sizeof(dmScreenSettings)); // Очистка для хранения установок
   dmScreenSettings.dmSize=sizeof(dmScreenSettings);      // Размер структуры Devmode
   dmScreenSettings.dmPelsWidth  = width;        // Ширина экрана
   dmScreenSettings.dmPelsHeight  = height;        // Высота экрана
   dmScreenSettings.dmBitsPerPel  = bits;          // Число бит на пиксель
   dmScreenSettings.dmFields=DM_BITSPERPEL|DM_PELSWIDTH|DM_PELSHEIGHT;

   // Попробовать переключить режим и получить результат
   // Замечание: при CDS_FULLSCREEN исчезает панель задач.
   if (ChangeDisplaySettings(&dmScreenSettings,CDS_FULLSCREEN)!=DISP_CHANGE_SUCCESSFUL)
   {
    // Если не прошло, то запросить разрешение на работу в видеорежиме рабочего стола.
    if (MessageBox(NULL,"The Requested Fullscreen Mode Is Not Supported By\nYour Video Card. Use Windowed Mode
Instead?","NeHe GL",MB_YESNO|MB_ICONEXCLAMATION)==IDYES)
    {
     fullscreen=FALSE; // Выбран оконный режим
    }
    else
    {
     // Сообщение о том, что работа программы завершается.
     MessageBox(NULL,"Program Will Now Close.","ERROR",MB_OK|MB_ICONSTOP);
     return FALSE;            // Return FALSE
    }
   }
  }
```

Функция WinMain() начинается так же. Спросим у пользователя разрешение на работу в полноэкранном режиме.

```
int WINAPI WinMain(
    HINSTANCE  hInstance,
    HINSTANCE  hPrevInstance,
    LPSTR      lpCmdLine,
    int        nCmdShow)
{
  MSG  msg; // Структура обработки сообщений
  BOOL  done=FALSE; // Переменная выхода из цикла

  // Спросить у пользователя какой режим он предпочитает
  if (MessageBox(NULL,"Would You Like To Run In Fullscreen Mode?",
     "Start FullScreen?",MB_YESNO|MB_ICONQUESTION)==IDNO)
  {
   fullscreen=FALSE; // Оконный режим
  }
  // Создать окно
  if (!CreateGLWindow("Andreas Luffler, Rob Fletcher & NeHe's Blitter & Raw Image Loading Tutorial", 640, 480, 32,
fullscreen))
  {
   return 0; // Выход, если окно не создано
  }
```

```
  while(!done) // Цикл пока done=FALSE
  {
   if (PeekMessage(&msg,NULL,0,0,PM_REMOVE)) // Ждем сообщений?
   {
    if (msg.message==WM_QUIT) // Получили сообщение о выходе?
    {
     done=TRUE; // тогда done=TRUE
    }
    else // Нет - обработать сообщений
    {
     TranslateMessage(&msg); // Транслировать
     DispatchMessage(&msg);  // Переслать
    }
   }
```

Я сделал некоторые изменения в коде ниже. Если программа - не активна (свернута), мы ждем первого сообщения программе с помощью команды WaitMessage(). Все останавливается, пока программа не получит сообщения (обычно это сообщение о развертывании окна). Спасибо Джиму Стронгу за это предложение.

```
   if (!active) // Программа не активна?
   {
    WaitMessage(); // Ожидать сообщения/Ничего не делать (НОВОЕ)
   }

   if (keys[VK_ESCAPE]) // Нажата Escape?
   {
    done=TRUE; // ESC сигнал на выход
   }

   if (keys[VK_F1]) // F1 нажата?
   {
    keys[VK_F1]=FALSE;
    KillGLWindow();
    fullscreen=!fullscreen;
    if (!CreateGLWindow("Andreas Luffler, Rob Fletcher & NeHe's Blitter & Raw Image Loading
Tutorial",640,480,16,fullscreen))
    {
     return 0;
    }
   }

   DrawGLScene();
   SwapBuffers(hDC);
  }

  // Выход
  KillGLWindow();
  return (msg.wParam);
}
```

Отлично! Теперь вы можете создавать крутые эффекты с применением смешивания. Используя текстурные буферы, как в этом уроке, вы можете создавать крутые эффекты, такие как плазма или вода. При объединении этих эффектов вместе Вы можете сделать почти фотореалистический ландшафт. Если что-то не работает в этом уроке, или у вас есть предложения, как улучшить урок, пошлите мне сообщение по электронной почте. Спасибо за то, что прочитали этот урок! Удачи Вам в создании ваших собственных интересных эффектов

Урок 30. Определение столкновений и моделирование законов физики

Collision Detection

Исходный код на котором основан этот урок, взят из моей старой конкурсной работы (ее можно найти на OGLchallenge.dhs.org). Тема называлась “Сумасшедшие столкновения” и моя статья (которая, кстати, заняла первое место :)) была названа Магической Комнатой. Она освещала определение столкновений, моделирование законов физики и эффекты.

Определение столкновений

Трудная тема, и честно говоря, я не знаю до сих пор простых подходов для ее решения. Для каждого приложения существуют различные способы нахождения и проверки столкновений. Конечно, существуют обобщенные физические законы и они могут работать с любыми видами объектов, но они очень медленные.

Мы собираемся исследовать алгоритмы, которые очень быстрые, легкие для понимания и до некоторой степени гибкие. К тому же важно и должно быть рассмотрено, что сделать, когда столкновение определено, и то, как тогда перемещать объекты, в соответствии с законами физики. Мы имеет много материала для рассмотрения. Давайте просмотрим, что мы собираемся изучить:

1) Определение столкновений
· Движущаяся сфера - Плоскость
· Движущаяся сфера - Цилиндр
· Движущаяся сфера - движущаяся сфера

2) Моделирование законов физики
· Реакция на столкновение
· Движение под действием гравитации с использованием уравнения Эйлера

3) Специальные эффекты
· Моделирование взрыва с использованием метода Fin-Tree Billboard
· Использование звуков с помощью The Windows Multimedia Library (только для Windows)

4) Разъяснение кода
· Код, разделен на 5 файлов

Lesson30.cpp : Основной код для этого учебника
Image.cpp, Image.h : Код загрузки текстур
Tmatrix.cpp, Tmatrix.h : Классы обработки вращения
Tray.cpp, Tray.h : Классы, обрабатывающие операции с лучами
Tvector.cpp, Tvector.h : Классы, обрабатывающие операции с векторами

В этом коде есть много удобного для дальнейшего использования! Классы операций с векторами, лучами и матрицами очень полезны. Я использую их до сих пор в собственных проектах.

1) Определение столкновений.

Для определения столкновений мы собираемся использовать алгоритмы метода трассировки лучей. Дадим сначала определение луча.

Луч задается с помощью вектора, он имеет начальную точку и вектор (обычно нормализованный), по направлению которого идет луч. По существу, луч исходит из начальной точки и движется по направлению направляющего вектора. Итак, наше уравнение луча:

PointOnRay = Raystart + t * Raydirection

t - точка, принимающая значения из [0, бесконечность).

При 0 мы получим начальную точку, используя другие значения, мы получаем соответствующие точки вдоль луча.

PointOnRay, Raystart, Raydirection - трехмерные вектора со значениями (x,y,z). Сейчас мы можем использовать это представление луча и вычислить пересечение с плоскостью или цилиндром.

Определение пересечения луча с плоскостью.

Плоскость представляется с помощью векторного представления таким образом:

Xn dot X = d

Xn, X - векторы, и d - значение с плавающей точкой.
Xn - ее нормаль.
X - точка на ее поверхности.
d - расстояние от центра системы координат до плоскости вдоль нормали.

По существу, плоскость обозначает половину пространства. Поэтому, все, что нам необходимо, чтобы определить плоскость, это 3D точка и нормаль в этой точке, которая является перпендикуляром к этой плоскости. Эти два вектора формируют плоскость, т.е. если мы возьмем для 3D точки вектор (0,0,0) и нормаль (0,1,0), мы по существу определяем плоскость через оси x и y. Поэтому, определения точки и нормали достаточно для вычисления векторного представления плоскости.

Согласно векторному уравнению плоскости, нормаль - Xn и 3D точка из которой исходит нормаль - X. Недостающие значение - d, которое легко вычисляется с помощью dot product (скалярного произведения).

(Замечание: Это векторное представление эквивалентно широко известной параметрической формуле плоскости Ax + By + Cz + D=0, для соответствия надо просто взять три значения нормали x,y,z как A,B,C и присвоить D=-d).

Вот два уравнения, которые мы пока что имеем:

PointOnRay = Raystart + t * Raydirection
Xn dot X = d

Если луч пересекает плоскость в некоторой точке, то тогда должна быть какая-то точка на луче, которая соответствует уравнению плоскости следующим образом:

Xn dot PointOnRay = d или (Xn dot Raystart) + t * (Xn dot Raydirection) = d

находя для t:

t = (d - Xn dot Raystart) / (Xn dot Raydirection)

заменяя d:

t= (Xn dot PointOnRay - Xn dot Raystart) / (Xn dot Raydirection)

сокращая его:

t= (Xn dot (PointOnRay - Raystart)) / (Xn dot Raydirection)

t представляет расстояние от начала луча до точки пересечения с плоскостью по направлению луча. Поэтому, подставляя t в уравнении луча, мы можем получить точку столкновения. Однако, существует несколько особых случаев. Если Xn dot Raydirection = 0, тогда эти два вектора перпендикулярны (луч идет паралельно плоскости), и столкновения не будет. Если t отрицателен, луч направлен в противоположную от плоскости сторону и не пересекает ее.

```
int TestIntersionPlane
(const Plane& plane,const TVector& position,
 const TVector& direction, double& lamda, TVector& pNormal)
{
    // Векторное произведение между нормалью плоскости и лучом
    double DotProduct=direction.dot(plane._Normal);
    double l2;

    // Определить, параллелен ли луч плоскости
    if ((DotProduct<ZERO)&&(DotProduct>-ZERO))
        return 0;
```

```
    // Определить расстояние до точки столкновения
    l2=(plane._Normal.dot(plane._Position-position))/DotProduct;

    if (l2<-ZERO)     // Определить, пересекает ли луч плоскость
        return 0;

    pNormal=plane._Normal;
    lamda=l2;
    return 1;
}
```

Код, приведенный выше, вычисляет и возвращает пересечение. Он возвращает 1, если пересечение есть, иначе 0. Параметры: плоскость (plane), начало (position) и направление вектора луча (direction), lamda - расстояние до точки столкновения, если оно есть, и вычисляется нормаль от точки столкновения (pNormal).

Пересечение луча с цилиндром

Вычисление пересечения между бесконечным цилиндром и лучом настолько сложено, что я не хочу объяснять его здесь. Этот способ требует больших математических расчетов и его просто объяснить, но моя главная цель дать вам инструменты, без излишней детализации (это не класс геометрии). Если кто-то интересуется теорией, на которой основан код, смотрите Graphic Gems II Book (pp 35, intersection of a with a cylinder). Цилиндр представляется как луч, с началом и направляющим вектором (здесь он совпадает с как осью), и радиус (радиус вокруг оси цилиндра). Соответственно функция:

```
int TestIntersionCylinder
(const Cylinder& cylinder, const TVector& position, const TVector& direction,
 double& lamda, TVector& pNormal, TVector& newposition)
```

Возвращает 1, если было обнаружено пересечение, иначе 0.

Параметры: структура, задающая цилиндр (смотрите в объяснении кода ниже), вектор начала и вектор направления луча. Значения, возвращаемые через параметры - расстояние, нормаль от точки пересечения и сама точка пересечения.

Столкновение сферы со сферой

Сфера задается с помощью ее центра и ее радиуса. Столкновение двух сфер определить легко. Находя расстояние между двумя центрами (метод dist класса TVector) мы можем это определить, пересекаются ли они, если расстояние меньше, чем сумма их радиусов.

Проблема лежит в определении, столкнуться ли две ДВИЖУЩИЕСЯ сферы. Ниже есть пример, где две сферы двигаются в течение временного шага из одной точки в другую. Их пути пересекаются, но этого недостаточно, чтобы подтвердить, что столкновение произошло (они могут пройти в различное время), да и точку столкновения определить невозможно.

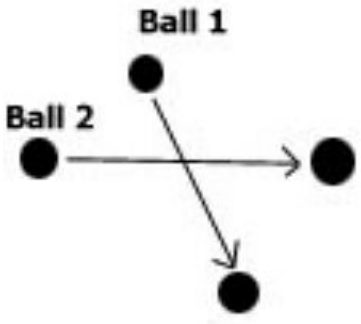


Рисунок 1

Предыдущие методы пересечения были решением уравнений объектов для определения пересечения. Когда используются сложные формы объектов или эти уравнения не применимы или не могут быть решены, должны быть использованы другие методы. Начальные и конечные точки, временной шаг, скорость (направление сферы + скорость) сферы и метод вычисления пересечения неподвижных сфер уже известны. Чтобы вычислить пересечение, временной шаг должен быть разрезан на более мелкие части. Затем, мы перемещаем сферы в соответствии к этим разрезанным временным шагам, используя ее скорость, и проверяем столкновение. Если в какой-либо точке обнаруживается столкновение (это означает, что сферы уже проникли друг в друга), то мы берем предыдущую позицию как точку пересечения (мы можем начать интерполяцию между этими точками, чтобы точно определить позицию пересечения, но это в основном не требуется).

Чем меньше временной шаг, чем больше частей мы используем, тем точнее метод. Например, допустим временной шаг равен 1 и количество частей - 3. Мы бы проверили два шара на столкновение во время 0, 0.33, 0.66, 1. Легко!!!!

Код, который это выполняет:

```
/*** Определить, какой из текущих шаров ***/
/*** пересекает другой в текущем временном шаге ***/
/*** Возвращает индекс двух пересекающихся шаров, точку и время пересечения ***/

int FindBallCol
(TVector& point, double& TimePoint, double Time2,
 int& BallNr1, int& BallNr2)
{
    TVector RelativeV;
    TRay rays;
    double MyTime=0.0, Add=Time2/150.0, Timedummy=10000, Timedummy2=-1;
    TVector posi;
    // Проверка всех шаров один относительно других за 150 маленьких шагов
    for (int i=0;i<NrOfBalls-1;i++)
    {
        for (int j=i+1;j<NrOfBalls;j++)
        {
            RelativeV=ArrayVel[i]-ArrayVel[j]; // Найти расстояние
            rays=TRay(OldPos[i],TVector::unit(RelativeV));
            MyTime=0.0;

            // Если расстояние между центрами больше чем 2*радиус
            if ( (rays.dist(OldPos[j])) > 40) continue;
            // Произошло пересечение
            // Цикл для точного определения точки пересечения
            while (MyTime<Time2)
            {
                MyTime+=Add;
                posi=OldPos[i]+RelativeV*MyTime;
                if (posi.dist(OldPos[j])<=40)
                {
                    point=posi;
                    if (Timedummy>(MyTime-Add)) Timedummy=MyTime-Add;
                    BallNr1=i;
                    BallNr2=j;
                    break;
                }
            }
        }
    }

    if (Timedummy!=10000)
    {
        TimePoint=Timedummy;
        return 1;
    }
    return 0;
}
```

Как использовать то, что мы только что изучили.

Поскольку сейчас мы можем определить точку пересечения между лучом и плоскостью/цилиндром, мы должны использовать это каким-нибудь образом для определения столкновения между сферой и одним из этих примитивов. Что мы могли сделать до этого, это определить точную точку столкновения между частицей (точкой) и плоскостью/цилиндром. Начало луча - расположение частицы, и направление луча - его вектор скорости (скорость и направление). Сделать это применительно к сферам довольно легко. Смотрите на рисунке 2a, как это может быть выполнено.

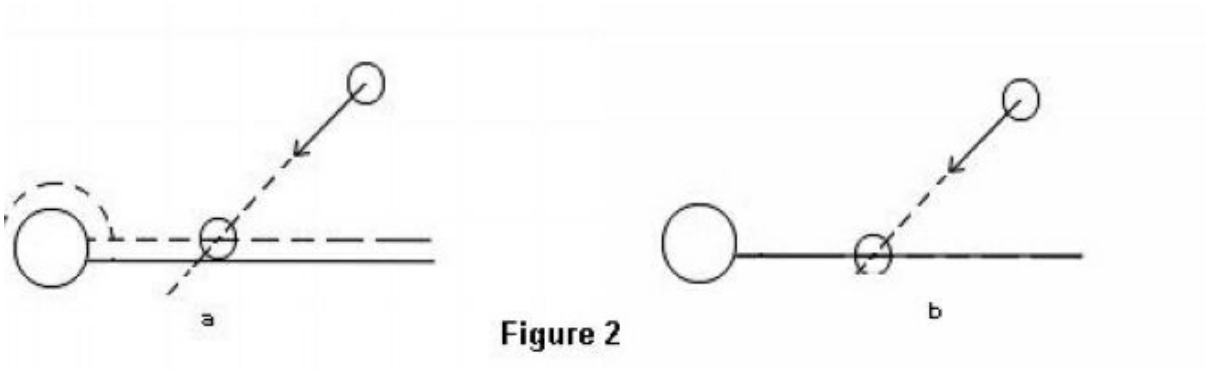


Рисунок 2a

Рисунок 2b

Каждая сфера имеет радиус, берем центр сферы как частицу (точка) и сдвигаем поверхность вдоль нормали каждой интересующей нас плоскости/цилиндра. На рисунке 2a эти новые примитивы изображены пунктирными линиями. Наши настоящие примитивы изображены непрерывными линиями, но тест на столкновение делается с помощью сдвинутых примитивов (представленных пунктирными линиями). Фактически, мы выполняем тест на пересечение с помощью небольшой плоскости сдвига и увеличенным радиусом цилиндра. Используя эту маленькую хитрость, шар не проникает в поверхность, если пересечение обнаружено с помощью его центра. Иначе мы получаем ситуацию как на рисунке 2b, где сфера проникает в поверхность. Это происходит, потому что мы определяем пересечение между его центром и примитивом, что означает, что мы не изменяли наш первоначальный код!

Определив, где будет столкновение, мы должны определить, будет ли пересечение в нашем текущем временном шаге. Временной шаг это время, в течение которого мы перемещаем сферу из ее текущей точки в соответствии с ее скоростью. Из-за того, что мы тестируем с помощью бесконечных лучей, всегда существует возможность того, что точка столкновения будет позади нового расположения сферы. Чтобы определить это, мы перемещаем сферу, вычисляем ее новое расположение и находим расстояние между начальной и конечной точкой. Из нашей процедуры определения столкновений мы также можем взять расстояния от начальной точки до точки столкновения. Если это расстояние меньше чем расстояние между начальной и конечной точкой, тогда столкновение есть. Чтобы вычислить точное время, мы решаем следующее простое уравнение. Представляем расстояние между начальной и конечной точкой как Dst, расстояние между начальной точкой и точкой столкновения - Dsc, и временной шаг - T. Время, когда происходит столкновение (Tc):

Tc= Dsc*T / Dst

Все это выполняется, конечно, если пересечение было определено. Возвращаемое время - часть от целого временного шага, если временной шаг был в 1 секунду, и мы обнаружили пересечение точно в середине расстояния, то вычисленное время столкновения будет 0.5 сек. Сейчас точка пересечения может быть вычислена только умножением Tc на текущую скорость и прибавлением к начальной точке.

Collision point= Start + Velocity*Tc

Это точка столкновения на расширенном примитиве, чтобы найти точку столкновения на настоящем примитиве мы добавляем к этой точке реверс нормали от этой точки (который также возвращается процедурой пересечения) с помощью радиуса сферы. Заметьте, что процедура пересечения цилиндра возвращает точку пересечения, если она существует, поэтому не нуждается в вычислении.

2) Моделирование законов физики

Реакция на столкновения

Определить, как отреагируют после удара неподвижные объекты, типа плоскостей, цилиндров также важно, как определить точку столкновения. Используя описанные алгоритмы и функции, можно обнаружить точную точку столкновения, нормаль от нее и время внутри временного шага, в течение которого происходит столкновение.

Чтобы определить, как отреагировать на столкновение, должны быть применены законы физики. Когда объект сталкивается с поверхностью, его направление меняется, т.е. он отскакивает. Угол нового направления (или вектор отражения) от нормали точки столкновения такой же, как у первоначального вектора. Рисунок 3 показывает столкновение со сферой.

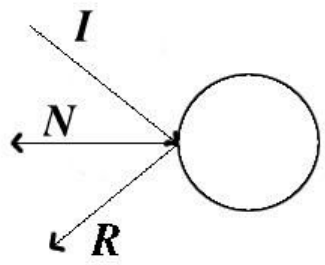


Рисунок 3

R - новый направляющий вектор
I - старый направляющий вектор, до столкновения
N - нормаль от точки столкновения

Новый вектор R вычисляется следующим образом:

R= 2*(-I dot N)*N + I

Есть ограничение: вектора I и N должны быть единичными векторами. Вектор скорости, который мы использовали в наших примерах, представляет скорость и направление. Вектор скорости не может быть включен в уравнение за место I, без преобразования. Скорость должна быть исключена. Скорость исключается нахождением величины вектора. Когда величина вектора найдена, вектор может быть преобразован в единичный и включен в уравнение, вычисляющее вектор отражения R. R показывает нам направление луча отражения, но для того, чтобы использовать как вектор скорости, необходимо к нему подключить скорость. Берем его, умножаем на величину первоначального луча, получаем правильный вектор скорости.

В примере эта процедура применяется для вычисления реакции на столкновение, когда шар сталкивается с плоскостью или цилиндром. Но она работает также для любых поверхностей, их форма не имеет значения. Пока точка столкновения и нормаль могут быть вычислены, метод вычисления реакции на столкновение всегда будет тот же самый. Код, который выполняет эти операции:

```
rt2=ArrayVel[BallNr].mag(); // Найти величину скорости
ArrayVel[BallNr].unit(); // Нормализовать его

// Вычислить отражение
ArrayVel[BallNr]=TVector::unit( (normal*(2*normal.dot(-ArrayVel[BallNr]))) + ArrayVel[BallNr] );
// Умножить на величину скорости для получения вектора скорости
ArrayVel[BallNr]=ArrayVel[BallNr]*rt2;
```

Когда сфера сталкивается с другой сферой

Определить реакцию на столкновение двух шаров намного труднее. Должны быть решены сложные уравнения динамики частиц, и поэтому я выдам только окончательное решение без каких-либо доказательств. Просто поверьте мне в этом :). Во время столкновения имеем ситуацию, как изображено на рисунке 4.

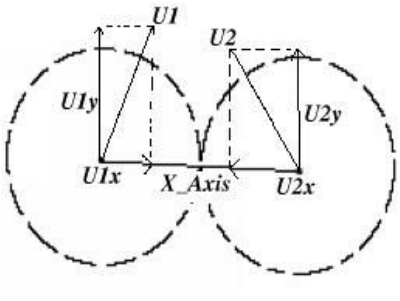


Рисунок 4

U1 и U2 векторы скорости двух сфер во время столкновения. Существует ось (X_Axis), вектор, которые соединяет центры двух сфер, и U1x, U2x проекции векторов скоростей U1,U2 ось (X_Axis).

U1y и U2y проекции векторов скорости U1,U2 на ось, перпендикулярную X_Axis. Чтобы найти эти вектора нужно просто произвести скалярное произведение. M1, M2 - массы двух сфер, соответственно. V1,V2 - новые скорости после столкновения, и V1x, V1y, V2x, V2y - проекции векторов скорости на X_Axis.

Более подробно:

a) Найти X_Axis

```
X_Axis = (center2 - center1);
Unify X_Axis, X_Axis.unit();
```

b) Найти проекции

```
U1x= X_Axis * (X_Axis dot U1)
U1y= U1 - U1x
U2x =-X_Axis * (-X_Axis dot U2)
U2y =U2 - U2x
```

c) Найти новые скорости

```
    (U1x * M1)+(U2x*M2)-(U1x-U2x)*M2
V1x= --------------------------------
             M1+M2
    (U1x * M1)+(U2x*M2)-(U2x-U1x)*M1
V2x= --------------------------------
             M1+M2
```

В нашем приложении мы установили M1=M2=1, поэтому уравнение получилось даже проще.

d) Найти окончательные скорости

```
V1y=U1y
V2y=U2y
V1=V1x+V1y
V2=V2x+V2y
```

Решение этих уравнений требует много работы, но раз они в той форме, как выше, они могут быть использованы совершенно легко. Код, который вычисляет действительную реакцию на столкновение:

```
TVector pb1,pb2,xaxis,U1x,U1y,U2x,U2y,V1x,V1y,V2x,V2y;
double a,b;
// Найти расположение первого шара
pb1=OldPos[BallColNr1]+ArrayVel[BallColNr1]*BallTime;
// Найти расположение второго шара
pb2=OldPos[BallColNr2]+ArrayVel[BallColNr2]*BallTime;
xaxis=(pb2-pb1).unit(); // Найти X-Axis
```

```
a=xaxis.dot(ArrayVel[BallColNr1]); // Найти проекцию
U1x=xaxis*a;      // Найти спроецированные вектора
U1y=ArrayVel[BallColNr1]-U1x;
xaxis=(pb1-pb2).unit(); // Сделать также, как выше
b=xaxis.dot(ArrayVel[BallColNr2]); // Найти проекцию
U2x=xaxis*b; // Векторы для другого шара
U2y=ArrayVel[BallColNr2]-U2x;
V1x=(U1x+U2x-(U1x-U2x))*0.5; // Сейчас найти новые скорости
V2x=(U1x+U2x-(U2x-U1x))*0.5;
V1y=U1y;
V2y=U2y;
for (j=0;j<NrOfBalls;j++) // Обновить все новые расположения
ArrayPos[j]=OldPos[j]+ArrayVel[j]*BallTime;
ArrayVel[BallColNr1]=V1x+V1y; // Установить новые вектора скорости
ArrayVel[BallColNr2]=V2x+V2y; // столкнувшимся шарам
```

Движение под действием гравитации, с использованием уравнения Эйлера

Чтобы изобразить реалистичное движение со столкновениями, определение точки столкновения и вычисления реакции не достаточно. Движение основывается на физических законах и тоже должно быть смоделировано.

Наиболее широко используемый метод для этого - использование уравнения Эйлера. Как показано, все вычисления должны быть выполнены с использованием временного шага. Это означает, что все моделирование происходит в некоторых временных шагах, в течение которых происходит движение, и выполняются тесты на столкновения и реакцию. Как пример, мы можем произвести моделирование в течение 2 секунд на каждом фрейме. Основываясь на уравнении Эйлера, скорость и расположение в каждом нового временном шаге вычисляется следующим образом:

```
Velocity_New = Velovity_Old + Acceleration*TimeStep
Position_New = Position_Old + Velocity_New*TimeStep
```

Сейчас объекты перемещаются и тестируются на столкновения, используя новую скорость. Ускорение для каждого объекта вычисляется делением силы, действующей на него, на его массу, в соответствии с эти уравнением:

```
Force = mass * acceleration
```

Много физических формул :)

Но, в нашем случае, на объекты действует только сила тяжести, которая может быть представлена сейчас как вектор, указывающий ускорение. В нашем случае, что-либо отрицательное в направлении Y, типа (0,-0.5,0). Это означает, что в начале каждого временного шага, мы вычисляем новую скорость каждой сферы и перемещаем их, тестируя на столкновение. Если во время временного шага происходит столкновение (скажем после 0.5 сек. с временным шагом равным 1 сек.) мы передвигаем объект в эту позицию, вычисляем отражение (новый вектор скорости) и перемещаем объект за оставшееся время (0.5 в нашем примере) снова тестируя на столкновения в течение этого времени. Эта процедура выполняется пока не завершится временной шаг.

Когда присутствует много движущихся объектов, каждый движущийся объект тестируется на пересечение с неподвижными объектами и ближайшее пересечение записывается. Далее выполняется тест на пересечение среди движущихся объектов для определения столкновений, в котором каждый объект тестируется с каждым другим. Обнаруженные пересечения сравниваются с пересечениями со статическим объектами, и берется наиболее близкое из них. Все моделирование обновляется в этой точке, (т.е., если ближайшее пересечение было после 0.5 сек., мы должны переместить все объекты на 0.5 сек.), для столкнувшихся объектов вычисляется вектор отражения, и цикл снова выполняется за оставшееся время.

3) Специальные эффекты
Взрывы

Каждый раз, когда происходит столкновение, в точке столкновения происходит взрыв. Хороший способ моделировать взрывы - произвести смешивание двух перпендикулярных друг другу полигонов с центрами в интересующей точке (в точке пересечения). Полигоны уменьшаются и исчезают со временем. Исчезновение выполняется изменением для вершин в течение времени значения alpha от 1 до 0. Так как возникает много полупрозрачных полигонов, то это может вызвать проблемы, и они могут перекрывать друг друга (как указано в Red Book в главе о прозрачности и смешивании) из-за Z-буфера, мы заимствуем технику, используемую в рендеринге частиц. Чтобы реализовать корректно этот трюк (как это описано в Red Book), мы должны отсортировать полигоны в

соответствии с их расположением по глубине, но с выключением записи в буфер глубины (не чтения). Заметьте, что число взрывов ограничено до 20 за один фрейм, если происходят дополнительные взрывы, буфер переполняется, и они сбрасываются. Код, который производит взрывы:

```
// Исполнение / смешивание взрывов
glEnable(GL_BLEND); // Включить смешивание
glDepthMask(GL_FALSE); // Отключить запись буфера глубины
glBindTexture(GL_TEXTURE_2D, texture[1]); // Подключение текстуры
for(i=0; i<20; i++) // Обновление и визуализация взрывов
{
    if(ExplosionArray[i]._Alpha>=0)
    {
        glPushMatrix();
        ExplosionArray[i]._Alpha-=0.01f; // Обновить альфу
        ExplosionArray[i]._Scale+=0.03f; // Обновить размер
        // Назначить прозрачным вершинам желтый цвет
        glColor4f(1,1,0,ExplosionArray[i]._Alpha); // Размер
        glScalef(ExplosionArray[i]._Scale,
            ExplosionArray[i]._Scale,ExplosionArray[i]._Scale);
        // Переместить в позицию с учетом масштабирования
        glTranslatef(
        (float)ExplosionArray[i]._Position.X()/ExplosionArray[i]._Scale,
        (float)ExplosionArray[i]._Position.Y()/ExplosionArray[i]._Scale,
        (float)ExplosionArray[i]._Position.Z()/ExplosionArray[i]._Scale);
        glCallList(dlist);// Вызвать список изображений
        glPopMatrix();
    }
}
```

Звук
Для звука была использована мультимедийная функция окошек PlaySound(). Это быстрый и отвратительный способ проигрывания звуковых файлов быстро и без хлопот.

4) Разъяснение кода
Поздравляю...
Если вы еще со мной, значит, вы успешно пережили теоретическую часть ;). Перед тем как позабавиться с демкой, необходимы некоторые разъяснения исходного кода. Основные действия и шаги моделирования следующие (в псевдокоде):

```
Цикл (ВременнойШаг!=0)
{
    Цикл по всем шарам
    {
        вычислить ближайшее столкновение с плоскостью
        вычислить ближайшее столкновение с цилиндром
        Сохранить и заменить, если это ближайшее пересечение
        по времени вычисленное до сих пор;
    }
    Проверить на столкновение среди движущихся шаров;
    Сохранить и заменить, если это ближайшее пересечение
    по времени, вычисленное до сих пор;
    If (Столкновение произошло)
    {
        Переместить все шары на время, равное времени столкновения;
        (Мы уже вычислили точку, нормаль и время столкновения.)
        Вычислить реакцию;
        ВременнойШаг -=ВремяСтолкновения;
    }
    else
        Переместить все шары на время, равное временному шагу
}
```

Настоящий код, выполняющий псевдокод выше - тяжелей для чтения, но, в сущности, точная реализация этого псевдокода.

```
// Пока не закончится временной шаг
while (RestTime>ZERO)
{
    lamda=10000; // Инициализировать очень большое значение
    // Для всех шаров найти ближайшее пересечение между шарами и плоскостями/цилиндрами
    for (int i=0;i<NrOfBalls;i++)
    {
        // Вычислить новое расположение и расстояние
        OldPos[i]=ArrayPos[i];
        TVector::unit(ArrayVel[i],uveloc);
        ArrayPos[i]=ArrayPos[i]+ArrayVel[i]*RestTime;
        rt2=OldPos[i].dist(ArrayPos[i]);
        // Проверить, произошло ли столкновение между шаром и всеми 5 плоскостями
        if (TestIntersionPlane(pl1,OldPos[i],uveloc,rt,norm))
        {
            // Найти время пересечения
            rt4=rt*RestTime/rt2;
            // Если оно меньше, чем уже сохраненное во временном шаге, заменить
            if (rt4<=lamda)
            {
                // Если время пересечения в текущем временном шаге
                if (rt4<=RestTime+ZERO)
                    if (! ((rt<=ZERO)&&(uveloc.dot(norm)>ZERO)) )
                    {
                        normal=norm;
                        point=OldPos[i]+uveloc*rt;
                        lamda=rt4;
                        BallNr=i;
                    }
            }
        }

        if (TestIntersionPlane(pl2,OldPos[i],uveloc,rt,norm))
        {
            // ...То же самое, что и выше
        }

        if (TestIntersionPlane(pl3,OldPos[i],uveloc,rt,norm))
        {
            // ...То же самое, что и выше
        }

        if (TestIntersionPlane(pl4,OldPos[i],uveloc,rt,norm))
        {
            // ...То же самое, что и выше
        }

        if (TestIntersionPlane(pl5,OldPos[i],uveloc,rt,norm))
        {
            // ...То же самое, что и выше
        }

        // Сейчас проверяем пересечения с 3 цилиндрами
        if (TestIntersionCylinder(cyl1,OldPos[i],uveloc,rt,norm,Nc))
        {
            rt4=rt*RestTime/rt2;
            if (rt4<=lamda)
            {
                if (rt4<=RestTime+ZERO)
                    if (! ((rt<=ZERO)&&(uveloc.dot(norm)>ZERO)) )
```

```
                {
                    normal=norm;
                    point=Nc;
                    lamda=rt4;
                    BallNr=i;
                }
            }
        }

        if (TestIntersionCylinder(cyl2,OldPos[i],uveloc,rt,norm,Nc))
        {
            // ...То же самое, что и выше
        }

        if (TestIntersionCylinder(cyl3,OldPos[i],uveloc,rt,norm,Nc))
        {
            // ...То же самое, что и выше
        }
}


// После того, как были проверены все шары на столкновение с плоскостями/цилиндрами
// Проверить между ними и записать наименьшее время столкновения
if (FindBallCol(Pos2,BallTime,RestTime,BallColNr1,BallColNr2))
{
    if (sounds)
        PlaySound("Explode.wav",NULL,SND_FILENAME|SND_ASYNC);

    if ( (lamda==10000) || (lamda>BallTime) )
    {
        RestTime=RestTime-BallTime;
        TVector pb1,pb2,xaxis,U1x,U1y,U2x,U2y,V1x,V1y,V2x,V2y;
        double a,b;
        .
        .
        Код опущен для экономии пространства
        Код описан в разделе Моделирование физических законов
        Столкновение между сферами
        .
        .
        //Обновить массив взрывов и вставить взрыв
        for(j=0;j<20;j++)
        {
            if (ExplosionArray[j]._Alpha<=0)
            {
                ExplosionArray[j]._Alpha=1;
                ExplosionArray[j]._Position=ArrayPos[BallColNr1];
                ExplosionArray[j]._Scale=1;
                break;
            }
        }
      continue;
    }
}

// Конец проверок
// Если столкновение произошло, произвести моделирование для точного временного шага
// и вычислить реакцию для столкнувшихся шаров
if (lamda!=10000)
{
    RestTime-=lamda;
    for (j=0;j<NrOfBalls;j++)
    ArrayPos[j]=OldPos[j]+ArrayVel[j]*lamda;
```

```
        rt2=ArrayVel[BallNr].mag();
        ArrayVel[BallNr].unit();
        ArrayVel[BallNr]=TVector::unit( (normal*(2*normal.dot(-ArrayVel[BallNr])))
                    + ArrayVel[BallNr] );
        ArrayVel[BallNr]=ArrayVel[BallNr]*rt2;

        // Обновить массив взрывов и вставить взрыв
        for(j=0;j<20;j++)
        {
            if (ExplosionArray[j]._Alpha<=0)
            {
                ExplosionArray[j]._Alpha=1;
                ExplosionArray[j]._Position=point;
                ExplosionArray[j]._Scale=1;
                break;
            }
        }
    }
    else RestTime=0;
}
```

Основные глобальные переменные, представляющие важность:

Представляет направление и расположение камеры. Камера перемещается, используя функцию LookAt. Как вы, возможно, заметите, в не hook моде (который я объясню позже), вся сцена вращается вокруг, camera_rotation - угол вращения.

```
TVector dir
Tvector pos(0,-50,1000);
float camera_rotation=0;
```

Представляет ускорение, приложенное к движущимся шарам. В приложении действует как гравитация.

```
TVector accel(0, -0.05, 0);
```

Массив, который содержит новые и старые расположения и векторы скорости каждого шара. Количество шаров жестко установлено равным 10.

```
TVector ArrayVel[10];
TVector ArrayPos[10];
TVector OldPos[10];
int NrOfBalls=3;
```

Временной шаг, который мы используем.

```
double Time=0.6;
```

Если 1, камера меняет вид и следует за шаром (шар с индексом 0 в массиве). Для того чтобы камера следовала за шаром, мы использовали его расположение и вектор скорости для расположения камеры точно за шаром, и установили ее вид вдоль вектора скорости шара.

```
int hook_toball1=0;
```

Структуры, содержащие данные по взрывам, плоскостям и цилиндрам.

```
struct Plane
struct Cylinder
struct Explosion
```

Взрывы, хранящиеся в массиве фиксированной длины.

```
Explosion ExplosionArray[20];
```

Основные интересующие функции:

Выполняет тест на пересечение с примитивами

```
Int TestIntersionPlane(….);
int TestIntersionCylinder(...);
```

Загружает текстуры из bmp файлов

```
void LoadGLTextures();
```

Код визуализации. Визуализация шаров, стен, колонн и взрывов.

```
void DrawGLScene();
```

Выполнение основной логики симуляции

```
void idle();
```

Инициализация OpenGL

```
void InitGL();
```

Поиск, если любой шар сталкивается с другим в текущее время

```
int FindBallCol(...);
```

Для большей информации смотрите исходный код. Я пытался прокомментировать его настолько хорошо, насколько смог. Сейчас, когда логика определения столкновения и реакции понята, исходный код должен стать ясным. Не стесняйтесь обращаться ко мне для получения большей информации.

Как я заявил в начале этого учебника, тема определения столкновений - очень сложная тема, чтобы ее охватить в одном учебнике. Вы многое изучите в этом учебнике, достаточное для создания своих собственных достаточно впечатляющих демок, но все еще есть много чего, что нужно изучить по этой теме. Сейчас, когда вы имеете основы, все другие исходники по определению столкновений и моделированию физических законов должны стать легче для понимания. С этими словами, я отправляю вас своей дорогой и желаю вам счастливых столкновений!!!

Немного информации о Dimitrios Christopoulos: в настоящее время он работает как программный инженер по виртуальной реальности в Foundation of the Hellenic World в Athens/Греция (www.fhw.gr). Хотя он родился в Германии, он учился в Греции в University of Patras на факультете Компьютерной инженерии и информатики. Он также имеет MSc степень в Университете Hull (UK) по Компьютерной Графике и Виртуальному окружению. Первые шаги по программированию игр он начинал на Basic на Commodore 64, и перешел на C/C++/Assembly на PC платформе, после того как стал студентом. В течение нескольких последних лет он перешел на OpenGL. Также смотри на его сайте http://members.xoom.com/D_Christop.

Урок 31. Визуализация моделей Milkshape 3D

Model Loading

В качестве источника этого проекта я взял PortaLib3D, библиотеку, которую я написал, чтобы тем, кто ей пользуется, было легко отображать модели, используя очень маленькую часть дополнительного кода. И хотя вы, конечно, можете доверить все библиотеке, вы должны понимать, что она делает, в этом вам и поможет данный урок.

Часть PortaLib3D включенная здесь содержит мое авторское право. Это не значит, что этот код не может быть использован вами - это значит, что если вы вырежете и вставите в свой проект часть кода, то вам придется сослаться на меня. Это все. Если вы сами разберете и переделаете код (то, что вы сделаете, если вы не используете библиотеку, и если вы не изучаете, что-то простое типа 'вырезать и вставить код'!), тогда вы освободитесь от обязательств. Давайте посмотрим, в коде нет ничего особенного! Ок, перейдем к кое-чему более интересному!

Основной OpenGL код.

Основной OpenGL код в файле Lesson31.cpp. Он, почти совпадает с уроком 6, с небольшими изменениями в секции загрузки текстур и рисования. Мы обсудим изменения позже.

Milkshape 3D

Модель, которую использованная в примере разработана в Milkshape 3D. Причина, по которой я использую ее в том, что этот пакет для моделирования чертовски хорош, и имеет свой собственный формат файлов, в котором легко разобраться и понять. В дальнейшем я планирую включить поддержку загрузки формата Anim8or, потому что он бесплатный и, конечно, загрузку 3DS.

Тем не менее, самое важное в формате файла, который здесь будет кратко обсуждаться, не просто загрузка модели. Вы должны создать свою собственную структуру, которая будет удобна для хранения данных, и потом читать файл в нее. Поэтому, в начале, обсудим структуры, необходимые для модели.

Структуры данных модели

Вот структуры данных модели представленные в классе Model в Model.h. Первое и самое важное, что нам надо - это вершины:

```
// Структура для вершины

struct Vertex
{
    char m_boneID;      // Для скелетной анимации
    float m_location[3];
};


// Используемые вершины
int m_numVertices;
Vertex *m_pVertices;
```

Сейчас вы можете не обращать на переменную m_boneID внимания - рассмотрим ее в следующих уроках! Массив m_location представляет собой координаты точек (X, Y, Z). Две переменные хранят количество вершин и сами вершины в динамическом массиве, который создается загрузчиком.

Дальше нам надо сгруппировать вершины в треугольники:

```
// Структура треугольника
struct Triangle
{
    float m_vertexNormals[3][3];
    float m_s[3], m_t[3];
    int m_vertexIndices[3];
};
```

```
// Используемые треугольники
int m_numTriangles;
Triangle *m_pTriangles;
```

Теперь 3 вершины составляют треугольник и хранятся в m_vertexIndices. Это смещения в массиве m_pVertices. При этом каждая вершина содержится в списке только один раз, что позволят сократить место в памяти (и в вычислениях, когда мы потом будем рассматривать анимацию). m_s и m_t - это координаты (s, t) в текстуре для каждой из 3-х вершин. Текстура используется только одна для данной сетки (которые будут описаны ниже). Наконец, у нас есть член m_vertexNormals, в котором хранится нормали к каждой из 3-х вершин. Каждая нормаль имеет 3 вещественные координаты, описывающие вектор.

Следующая структура, которую мы рассмотрим в модели, это сетка (mesh). Сетка - это группа треугольников, к которым применен одинаковый материал. Набор сеток составляет целую модель. Вот структура сетки:

```
// Сетка
struct Mesh
{
    int m_materialIndex;
    int m_numTriangles;
    int *m_pTriangleIndices;
};

// Используемые сетки
int m_numMeshes;
Mesh *m_pMeshes;
```

На этот раз у нас есть m_pTriangleIndices, в котором хранится треугольники в сетке, в точности так же, как треугольники хранят индексы своих вершин. Этот массив будет выделен динамически, потому что количество треугольников в сетке в начала не известно, и определяется из m_num_Triangles. Наконец, m_materialIndex - это индекс материала (текстура и коэффициент освещения) используемый для сетки. я покажу структуру материала ниже:

```
// Свойства материала
struct Material
{
    float m_ambient[4], m_diffuse[4], m_specular[4], m_emissive[4];
    float m_shininess;
    GLuint m_texture;
    char *m_pTextureFilename;
};


// Используемые материалы
int m_numMaterials;
Material *m_pMaterials;
```

Здесь есть все стандартные коэффициенты освещения в таком же формате, как и в OpenGL: окружающий, рассеивающий, отражающий, испускающий и блестящий. У нас так же есть объект текстуры m_texture и имя файла (динамически располагаемое) текстуры, которые могут быть выгружены, если контекст OpenGL упадет.

Код - загрузка модели

Теперь займемся загрузкой модели. Вы увидите, что это чистая виртуальная функция, названная loadModelData, которая в качестве параметра имеет имя файла модели. Все что мы сделаем - это создадим производный класс MilkshapeModel, который использует эту функцию, которая заполняет защищенные структуры данных, упомянутые выше. Теперь посмотрим на функцию:

```
bool MilkshapeModel::loadModelData( const char *filename )
{
    ifstream inputFile( filename, ios::in | ios::binary | ios::nocreate );
    if ( inputFile.fail())
        return false; // "Не можем открыть файл с моделью."
```

Для начала мы открыли файл. Это бинарный файл, поэтому используем ios::binary. Если файл не найден, функция возвратит false, что говорит об ошибке.

```
inputFile.seekg( 0, ios::end );
long fileSize = inputFile.tellg();
inputFile.seekg( 0, ios::beg );
```

Код дальше определяет размер файла в байтах.

```
byte *pBuffer = new byte[fileSize];
inputFile.read( pBuffer, fileSize );
inputFile.close();
```

Затем файл читается во временный буфер целиком.

```
const byte *pPtr = pBuffer;
MS3DHeader *pHeader = ( MS3DHeader* )pPtr;
pPtr += sizeof( MS3DHeader );

if ( strncmp( pHeader->m_ID, "MS3D000000", 10 ) != 0 )
    return false; // "Не настоящий Milkshape3D файл."

if ( pHeader->m_version < 3 || pHeader->m_version > 4 )
    return false; // "Не поддерживаемая версия.
            //   Поддерживается только Milkshape3D версии 1.3 и 1.4."
```

Теперь указатель pPtr будет указывать на текущую позицию. Сохраняем указатель на заголовок и устанавливаем pPtr на конец заголовка. Вы, наверное, заметили несколько структур MS3D, которые мы использовали. Они объявлены в начале MilkshapeModel.cpp и идут прямо из спецификации формата файла. Поля в заголовке проверяются, что бы убедиться, в правильности загружаемого файла.

```
int nVertices = *( word* )pPtr;
m_numVertices = nVertices;
m_pVertices = new Vertex[nVertices];
pPtr += sizeof( word );

int i;
for ( i = 0; i < nVertices; i++ )
{
    MS3DVertex *pVertex = ( MS3DVertex* )pPtr;
    m_pVertices[i].m_boneID = pVertex->m_boneID;
    memcpy( m_pVertices[i].m_location, pVertex->m_vertex, sizeof( float )*3 );
    pPtr += sizeof( MS3DVertex );
}
```

Текст выше читает каждую структуру вершины из файла. Начальная память для модели выделяется для вершин, а затем каждая вершина копируется, пока не будут обработаны все. В функции используются несколько вызовов memcpy которая просто копирует содержимое маленьких массивов. Член m_boneID пока по-прежнему игнорируется - он для скелетной анимации!

```
int nTriangles = *( word* )pPtr;
m_numTriangles = nTriangles;
m_pTriangles = new Triangle[nTriangles];
pPtr += sizeof( word );

for ( i = 0; i < nTriangles; i++ )
{
    MS3DTriangle *pTriangle = ( MS3DTriangle* )pPtr;
    int vertexIndices[3] = { pTriangle->m_vertexIndices[0],
            pTriangle->m_vertexIndices[1], pTriangle->m_vertexIndices[2] };
    float t[3] = { 1.0f-pTriangle->m_t[0], 1.0f-pTriangle->m_t[1],
```

```
                1.0f-pTriangle->m_t[2] };
        memcpy( m_pTriangles[i].m_vertexNormals, pTriangle->m_vertexNormals,
                sizeof( float )*3*3 );
        memcpy( m_pTriangles[i].m_s, pTriangle->m_s, sizeof( float )*3 );
        memcpy( m_pTriangles[i].m_t, t, sizeof( float )*3 );
        memcpy( m_pTriangles[i].m_vertexIndices, vertexIndices, sizeof( int )*3 );
        pPtr += sizeof( MS3DTriangle );
    }
```

Так же как и для вершин, эта часть функции сохраняет все треугольники модели. Пока что она включает просто копирование массивов из одной структуры в другую, и вы увидите разницу между массивом vertexIndeces и t-массивами. В файле номера вершин хранятся как массив переменных типа word, в модели это переменные типа int для согласованности и простоты (при этом противное приведение не нужно). Итак просто нужно привести 3 значения к типу int. Все значения t задаются как 1.0 - (оригинальное значение). Причина этого в том, что OpenGL использует левую нижнюю систему координат, тогда как Milkshape использует правую верхнюю систему координат (прим.: имеется в виду расположение точки центра системы координат и ориентация) для работы с текстурой. Это меняет направление оси y.

```
    int nGroups = *( word* )pPtr;
    m_numMeshes = nGroups;
    m_pMeshes = new Mesh[nGroups];
    pPtr += sizeof( word );
   for ( i = 0; i < nGroups; i++ )
    {
        pPtr += sizeof( byte );     // Флаги
        pPtr += 32;           // Имя

        word nTriangles = *( word* )pPtr;
        pPtr += sizeof( word );
        int *pTriangleIndices = new int[nTriangles];
        for ( int j = 0; j < nTriangles; j++ )
        {
            pTriangleIndices[j] = *( word* )pPtr;
            pPtr += sizeof( word );
        }

        char materialIndex = *( char* )pPtr;
        pPtr += sizeof( char );

        m_pMeshes[i].m_materialIndex = materialIndex;
        m_pMeshes[i].m_numTriangles = nTriangles;
        m_pMeshes[i].m_pTriangleIndices = pTriangleIndices;
    }
```

Текст выше загружает данные структуры сетки (в Milkshape3D они называется группами "groups"). Так как число треугольников меняется от сетки к сетке, нет никакой стандартной структуры чтения. Поэтому берется поле за полем. Память для индексов треугольников выделяется динамически внутри сетки и читается по очереди.

```
    int nMaterials = *( word* )pPtr;
    m_numMaterials = nMaterials;
    m_pMaterials = new Material[nMaterials];
    pPtr += sizeof( word );
    for ( i = 0; i < nMaterials; i++ )
    {
        MS3DMaterial *pMaterial = ( MS3DMaterial* )pPtr;
        memcpy( m_pMaterials[i].m_ambient, pMaterial->m_ambient, sizeof( float )*4 );
        memcpy( m_pMaterials[i].m_diffuse, pMaterial->m_diffuse, sizeof( float )*4 );
        memcpy( m_pMaterials[i].m_specular, pMaterial->m_specular,
                sizeof( float )*4 );
        memcpy( m_pMaterials[i].m_emissive, pMaterial->m_emissive,
```

```
                sizeof( float )*4 );
        m_pMaterials[i].m_shininess = pMaterial->m_shininess;
        m_pMaterials[i].m_pTextureFilename = new char[strlen(
                pMaterial->m_texture )+1];
        strcpy( m_pMaterials[i].m_pTextureFilename, pMaterial->m_texture );
        pPtr += sizeof( MS3DMaterial );
    }
    reloadTextures();
```

Наконец, из буфера берется информация о материале. Это происходит так же, как и раньше, копированием каждого коэффициента освещения в новую структуру. Так же выделяется новая память для названия файла, содержащего текстуру, и оно копируется в эту память. Последний вызов reloadTextures используется собственно для загрузки текстур и привязки ее к объекту текстуры OpenGL. Эта функция из базового класса Model описывается ниже.

```
    delete[] pBuffer;
    return true;
}
```

Последний фрагмент освобождает память временного буфера, когда вся информация уже скопирована и работа процедуры завершена успешно.

Итак, в данный момент, защищенные члены класса Model заполнены информацией о модели. Заметьте, что это только код для MilkshapeModel, потому что все это относилось к специфике Milkshape3D. Теперь, перед тем как можно будет нарисовать модель, необходимо загрузить текстуры для всех материалов. Это мы сделаем в следующем куске кода:

```
void Model::reloadTextures()
{
    for ( int i = 0; i < m_numMaterials; i++ )
        if ( strlen( m_pMaterials[i].m_pTextureFilename ) > 0 )
            m_pMaterials[i].m_texture = LoadGLTexture( m_pMaterials[i].m_pTextureFilename );
        else
            m_pMaterials[i].m_texture = 0;
}
```

Для каждого материала, текстура загружается, используя функцию из основных уроков NeHe (слегка измененных в отличие от предыдущих версий). Если имя файла с текстурой - пустая строка, то текстура не загружается, но взамен текстуре объекта присваивается 0, что означает, что нет никакой текстуры.

Код - рисование модели

Теперь можем начать код, рисующий модель! Теперь это совсем не сложно, когда у нас есть аккуратно расположенные структуры данных в памяти.

```
void Model::draw()
{
    GLboolean texEnabled = glIsEnabled( GL_TEXTURE_2D );
```

Эта часть сохраняет состояние отображения текстур в OpenGL, поэтому функция не нарушит его. Заметьте, что она не сохраняет так же свойства материала.

Теперь цикл рисования каждой сетки по отдельности:

```
    // Рисовать по группам
    for ( int i = 0; i < m_numMeshes; i++ )
    {
```

m_pMeshes[i] будет использован для ссылки на текущую сетку. Теперь, каждая сетка имеет свои свойства материала, поэтому мы устанавливаем соответствующее состояние OpenGL. Если, однако, materialIndex сетки равен -1, это значит, что материала для такой сетки нет, и она рисуется в стандартном виде OpenGL.

```
        int materialIndex = m_pMeshes[i].m_materialIndex;
        if ( materialIndex >= 0 )
        {
            glMaterialfv( GL_FRONT, GL_AMBIENT,
                          m_pMaterials[materialIndex].m_ambient );
            glMaterialfv( GL_FRONT, GL_DIFFUSE,
                          m_pMaterials[materialIndex].m_diffuse );
            glMaterialfv( GL_FRONT, GL_SPECULAR,
                          m_pMaterials[materialIndex].m_specular );
            glMaterialfv( GL_FRONT, GL_EMISSION,
                          m_pMaterials[materialIndex].m_emissive );
            glMaterialf( GL_FRONT, GL_SHININESS,
                          m_pMaterials[materialIndex].m_shininess );

            if ( m_pMaterials[materialIndex].m_texture > 0 )
            {
                glBindTexture( GL_TEXTURE_2D,
                          m_pMaterials[materialIndex].m_texture );
                glEnable( GL_TEXTURE_2D );
            }
            else
                glDisable( GL_TEXTURE_2D );
        }
        else
        {
            glDisable( GL_TEXTURE_2D );
        }
```

Свойства материала устанавливаются в соответствие со значением, сохраненным в модели. Заметим, что текстура используется и доступна, если ее индекс больше чем 0. Если поставить 0, то вы отказываетесь от текстуры, и текстура не используется. Так же текстура не используется, если для сетки вообще нет материала.

```
        glBegin( GL_TRIANGLES );
        {
            for ( int j = 0; j < m_pMeshes[i].m_numTriangles; j++ )
            {
                int triangleIndex = m_pMeshes[i].m_pTriangleIndices[j];
                const Triangle* pTri = &m_pTriangles[triangleIndex];

                for ( int k = 0; k < 3; k++ )
                {
                    int index = pTri->m_vertexIndices[k];
                    glNormal3fv( pTri->m_vertexNormals[k] );
                    glTexCoord2f( pTri->m_s[k], pTri->m_t[k] );
                    glVertex3fv( m_pVertices[index].m_location );
                }
            }
        }
        glEnd();
    }
```

Секция выше производит рисование треугольников модели. Она проходит цикл для каждого из треугольников сетки и потом рисует каждую из 3-х вершин, используя нормали и координаты текстуры. Помните, что каждый треугольник в сетке, как и все вершины, пронумерованы в общих массивах модели (они используют 2 индексные переменные). pTri - указывает на текущий треугольник в сетке и используется для упрощения кода следующего за ним.

```
    if ( texEnabled )
        glEnable( GL_TEXTURE_2D );
    else
        glDisable( GL_TEXTURE_2D );
}
```

Заключительный фрагмент кода устанавливает режим отображения текстур в свое первоначальное состояние.

Другой важный кусок кода в классе Model - это конструктор и деструктор. Они сами все объясняют. Конструктор устанавливает все члены в 0-ое значение (или NULL для указателей), и деструктор удаляет динамические массивы из памяти для всех структур модели. Вы должны заметить, что если вызываете функцию loadModelData дважды, для объекта Model, то можете потерять часть памяти. Будьте осторожны!

Последняя тема, которую я буду здесь обсуждать, это изменение, в основном коде отображения используя новый класс модели, и прямо отсюда я планирую начать свой будущий урок о скелетной анимации.

```
Model *pModel = NULL;  // Место для хранения данных модели
```

В начале кода lesson31.cpp была объявлена модель, но не инициализирована. Она создается в процедуре WinMain:

```
pModel = new MilkshapeModel();
if ( pModel->loadModelData( "data/model.ms3d" ) == false )
{
    MessageBox( NULL, "Couldn't load the model data/model.ms3d",
            "Error", MB_OK | MB_ICONERROR );
    return 0;  // Если модель не загружена, выходим
}
```

Модель создается здесь, и не в InitGL, потому что InitGL вызывается каждый раз, когда мы меняем графический режим (теряя контекст OpenGL). Но модель не должна перезагружаться, так как данные не меняются. Это не относиться к текстурам, которые присоединены к объектам текстур, когда мы загружаем объект. Поэтому добавлена следующая строка в InitGL:

```
pModel->reloadTextures();
```

Это место вызова LoadGLTextures, как и раньше. Если бы в сцене было несколько моделей, то эту функцию пришлось бы вызывать для каждой из них. Если все объекты стали вдруг белыми, то это значит, что с вашими текстурами что-то не так и они не правильно загрузились.

Наконец, вот новая функция DrawGLScene:

```
int DrawGLScene(GLvoid)                         // Здесь происходит все рисование
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Очищаем экран и буфер глубины
    glLoadIdentity();                           // Сбрасываем вид
    gluLookAt( 75, 75, 75, 0, 0, 0, 0, 1, 0 );
    glRotatef(yrot,0.0f,1.0f,0.0f);
    pModel->draw();

    yrot+=1.0f;
    return TRUE;                                // Продолжаем
}
```

Просто? Мы очистили цветовой буфер, установили единичную матрицу модели/вида, и потом устанавливается камера в режим gluLookAt. Если вы раньше не пользовались gluLookAt, то по существу она помещает камеру в позицию, описываемую 3-мя параметрами, перемещая центр сцены в позицию, описываемую 3-мя следующими параметрами, и последние 3 параметра описывают вектор направленный вверх. В данном случае мы смотрим из точки (75, 75, 75) в точку (0, 0, 0), так как модель расположена в центре системы координат, если вы не переместили ее до этого, и положительное направление оси Y смотрит вверх.

Функция должна быть вызвана вначале, но после сброса матрицы просмотра, таким образом, она работает.

Чтобы сделать сцену немного интереснее, постепенно вращаем ее вокруг оси y с помощью glRotatef.

Наконец, рисуем модель с помощью члена-функции рисования. Она рисуется в центре (она была создана в центре системы координат в Milkshape3D!), поэтому если вы хотите вращать ее, менять позицию или масштабировать, просто вызовите соответствующие функции GL перед рисованием. Вуаля! Чтобы проверить, сделайте свои собственные модели в Milkshape (или используйте функции импорта), и загрузите их, изменяя строки в функции WinMain. Или добавьте их в сцену и нарисуйте несколько объектов.

Что дальше?

В будущем уроке NeHe, я опишу, как расширить класс, чтобы объединить модель с анимационным скелетом. И если я вернусь к этому, то я напишу еще классы для загрузки, чтобы сделать программу более гибкой.

Шаг к скелетной анимации не такой уж и большой, как может показаться, хотя математика, привлекаемая для этого, достаточно хитра. Если вы что-то не понимаете в матрицах и векторах, пришло время прочитать что-нибудь по этой теме! Есть несколько источников в сети, которые вам в этом помогут.

Увидимся!

Информация о Brett Porter: родился в Австралии, был студентом University of Wollongong, недавно получил дипломы BCompSc и BMath. Он начал программировать на Бейсике в 12 лет на "клоне" Commandore 64, называемом VZ300, но скоро перешел на Паскаль, Intel ассемблер, C++ и ява. В течение последних нескольких лет его интересом стало 3-х мерное программирование, и его выбором в качестве API стал OpenGL. Для более подробной информации посетите его страницу http://rsn.gamedev.net/.

Продолжение этого урока Скелетной Анимацией можно найти на странице Brett'a.

Урок 32. Выбор, альфа смешивание, альфа тест, сортировка.

Picking, Alpha Blending, Alpha Testing, Sorting

Добро пожаловать на тридцать второй урок. Это, пожалуй, самый большой урок, который я написал. Более чем 1000 строк кода, и более чем 1540 строк текста. Это также первый урок, в котором использован новый базовый код "NeHeGL basecode". Этот урок отнял у меня много времени, но я думаю, он стоит этого. Вот некоторые из тем, которые рассматриваются в этом уроке: альфа смешивание, альфа тест, получение сообщений от мыши, одновременное использование ортографической и перспективной проекций, отображение собственного курсора, ручная сортировка объектов с учетом глубины, хранение кадров анимации в одной текстуре и, что, пожалуй, самое важное Вы овладеете ВЫБОРОМ (picking)!

В первоначальной версии этого урока на экране отображались три объекта, которые изменяли цвет, когда на них нажимали. Вам это интересно!?! Не возбуждает вообще! Как всегда, мои дорогие, я хотел впечатлить Вас крутым уроком высшего качества. Я хотел, чтобы урок вам понравился, набил ваши мозги ценной информацией и конечно... клево выглядел. Итак, после нескольких недель кодирования, урок сделан! Даже, если Вы не программируете, Вы можете просто наслаждаться результатом этого урока. Это полноценная игра! Вы будете стрелять по множеству целей, до тех пор, пока ваш боевой дух (morale) не упадет ниже предельной черты или ваши руки сведёт судорогой, и Вы больше не сможете щелкать по кнопке мыши.

Я уверен, что в мой адрес по поводу этого урока будет критика, но я очень счастлив, что создал этот урок! Такие малоприятные темы, как выбор и сортировка объектов по глубине, я превратил в забавные!

Несколько небольших замечаний о коде, я буду обсуждать только ту часть кода, которая находится в файле lesson32.cpp. Произошло несколько незначительных изменений в коде NeHeGL. Наиболее важное изменение то, что я добавил поддержку мыши в WindowProc(). Я также добавил int mouse_x, mouse_y, чтобы сохранить положение курсора мыши. В NeHeGL.h две следующие строки кода были добавлены: extern int mouse_x; & extern int mouse_y;.

Все текстуры, которые используются в этом уроке, были сделаны в Adobe Photoshop. Каждый TGA файл – это 32-битное изображение с альфа-каналом. Если Вы не знаете, как добавить альфа-канал к изображению, то тогда купите себе хорошую книгу, посмотрите в Интернет или прочитайте помощь по Adobe Photoshop. Весь процесс создания изображения очень похож на тот, с помощью которого я создавал маски в уроке маскирования. Загрузите ваш объект в Adobe Photoshop (или другую программу для работы с изображениями, которая поддерживает альфа-канал). Произведите выбор области объекта по его контуру, например, выбором цветового диапазона. Скопируйте эту область. Создайте новое изображение. Вставьте выбранную область в новое изображение. Сделаете инверсию изображения так, чтобы область, где находится ваше изображение, стала черной. Сделайте область вокруг изображения белой. Выберите все изображение, и скопируйте его. Возвратитесь к первоначальному изображению, и создайте альфа-канал. Вставьте черно-белую маску, которую Вы только, что создали в альфа-канал. Сохраните изображение как 32 битный TGA файл. Проверьте, что сохранилась прозрачность, и проверьте, что Вы сохранили несжатое изображение!

Поскольку буду я надеяться, что вам понравится этот урок. Мне интересно знать, что Вы думаете о нем. Если у вас есть вопросы, или вы нашли ошибки в уроке, то сообщите мне об этом. Я быстро пройду по части урока, поэтому, если Вы находите, что какая-то часть урока трудна для понимания, то сообщите мне об этом, и я должен буду пробовать объяснить вещи по-другому или более подробно!

```
#include <windows.h> // заголовочный файл для Windows
#include <stdio.h>   // заголовочный файл для стандартного ввода/вывода
#include <stdarg.h>  // заголовочный файл для манипуляций с переменными аргументами
#include <gl\gl.h>   // заголовочный файл для библиотеки OpenGL32
#include <gl\glu.h>  // заголовочный файл для библиотеки GLu32
#include <time.h>    // для генерации псевдослучайных чисел
#include "NeHeGL.h"  // заголовочный файл для NeHeGL
```

В уроке 1 я рассказал, как правильно подключить библиотеки OpenGL. В Visual C ++ надо выбрать Project / Settings / Link. И добавить в строчку "Object/library modules" следующую запись: OpenGL32.lib, GLu32.lib и glaux.lib. Если транслятор не сможет подключить нужную библиотеку, то это заставит его извергать ошибку за ошибкой. Именно то, что вы и не хотите! Это все обостряется, когда Вы включили библиотеки в режим отладки, и пробуете скомпилировать ваш в режиме без отладочной информации (release)... еще больше ошибок. Есть много людей, которые просматривают код. Большинство из них плохо знакомы с программированием. Они берут ваш код, и пробуют компилировать его. Они получают ошибки, удаляют код и ищут дальше.

Код ниже сообщит транслятору, что нужно прикомпоновать нужные библиотеки. Немного больше текста, но намного меньше головной боли, в конечном счете. В этом уроке, мы используем библиотеки OpenGL32, GLU32 и WinMM (для проигрывания звука). В этом уроке мы будем загружать TGA файлы, поэтому мы не нуждаемся в библиотеке glaux.

```
#pragma comment( lib, "opengl32.lib" )  // Найти OpenGL32.lib во время линкования
#pragma comment( lib, "glu32.lib" )     // Найти GLu32.lib во время линкования
#pragma comment( lib, "winmm.lib" )     // Найти WinMM во время линкования
```

В трех строках ниже происходит проверка определения компилятором CDS_FULLSCREEN (используется в переключении видеорежима в функции ChangeDisplaySettings). Если это не так, то мы вручную задаем CDS_FULLSCREEN значение 4. Некоторые компиляторы не определяют CDS_FULLSCREEN и возвратят сообщение об ошибки, если CDS_FULLSCREEN используется! Чтобы предотвратить сообщение об ошибке, мы проверяем, был ли CDS_FULLSCREEN определен и если нет, то мы вручную задаем его. Это сделает жизнь немного проще для каждого. (Примечание переводчика: здесь ошибка, это определение нужно перенести в файл NeHeGL.cpp, так как там используется функция ChangeDisplaySettings).

Затем мы объявляем DrawTargets, и задаем переменные для нашего окна и обработки клавиатуры. Если Вы не знаете, как надо определять переменные, то пролистайте MSDN глоссарий. Имейте в виду, я не преподаю C/C++, купите хорошую книгу по этому языку, если Вам нужна справка не по GL коду!

```
#ifndef   CDS_FULLSCREEN     // CDS_FULLSCREEN не определен
#define   CDS_FULLSCREEN 4   // компилятором. Определим его,
#endif                       // чтобы избежать ошибок

void DrawTargets();          // декларация

GL_Window*  g_window;
Keys*    g_keys;
```

В следующем разделе кода задаются наши пользовательские переменные. Переменная base будет использоваться для наших списков отображения для шрифта. Переменная roll будет использоваться, чтобы перемещения земли и создания иллюзии прокрутки облаков. Переменная level (уровень) используется по прямому назначению (мы начинаем с уровня 1). Переменная miss отслеживает, сколько объектов не было сбито. Она также используется, чтобы показать боевой дух игрока (если не было промахов, то это означает высокий боевой дух). Переменная kills следит за тем, сколько целей было поражено на каждом уровне. В переменной score сохраняется общее число попаданий в объекты, и переменная game будет использоваться, чтобы сообщить о конце игры!

Последняя строка определяет нашу функцию сравнения. Функция qsort требует последний параметр с типом (const *void, const *void)..

```
// Наши пользовательские переменные
GLuint   base;    // Список отображения для шрифта
GLfloat  roll;    // Прокрутка облаков
GLint    level=1; // Текущий уровень
GLint    miss;    // Пропущенные цели
GLint    kills;   // Счетчик поражений для уровня
GLint    score;   // Текущий счет
bool     game;    // Игра окончена?

typedef int (*compfn)(const void*, const void*); // Определение нашей функции сравнения
```

Теперь о нашей структуре для объектов. Эта структура содержит всю информацию об объекте. Направление, в котором он вращается, попадание в него, положение на экране, и т.д.

Краткое описание переменных... Переменная rot определяет направление, в котором мы хотим вращать объект. Переменная hit равна ЛОЖЬ, если в объект еще не попали. Если объект был поражен или вручную помечен как пораженный, значение hit будет ИСТИННА.
Переменная frame используется, чтобы циклически повторять кадры анимации взрыва объекта. Поскольку frame увеличивается, то и произойдет смена текстуры взрыва. Далее мы остановимся на этом подробнее.
Чтобы следить за тем, в каком направлении наш объект перемещается, мы имеем переменную называемую dir. Переменная dir может принимать одно из 4 значений: 0 - объект перемещается влево, 1 - объект перемещает вправо, 2 – объект перемещается вверх и, наконец, 3 - объект перемещается вниз.
Переменная texid может иметь любое значение от 0 до 4. Ноль задает текстуру BlueFace (голубая рожа), 1 - текстуру Bucket (ведро), 2 – текстуру Target (мишень), 3 – Coke (банка кока-колы), и 4 - текстура Vase (ваза). Позже в коде загрузке текстуры, Вы увидите, что первые 5 текстур – изображения целей.

Обе переменные x и y используются для позиционирования объекта на экране. Переменная x задает позицию объекта по оси X, а переменная y задает позицию объекта по оси Y.

Объекты вращаются относительно оси Z в зависимости от значения переменной spin. Позже в коде, мы будем увеличивать или уменьшать вращение, основываясь на направлении перемещения объекта.

Наконец, переменная distance содержит значение расстояния нашего объекта от экрана. Переменная distance - чрезвычайно важная переменная, мы будем использовать ее, чтобы вычислить левую и правую стороны экрана, и сортировать объекты, так что дальние объекты были выведены прежде, чем ближайшие объекты.

```
struct objects {
  GLuint  rot;       // Вращение (0-нет, 1-по часовой, 2-против)
  bool  hit;         // В объект попали?
  GLuint  frame;     // Текущий кадр взрыва
  GLuint  dir;       // Направление объекта (0-лево, 1-право, 2-вверх, 3-низ)
  GLuint  texid;     // ID текстуры объекта
  GLfloat  x;        // X позиция
  GLfloat y;         // Y позиция
  GLfloat  spin;     // Вращение
  GLfloat  distance; // Расстояние
};
```

Абсолютно не зачем пояснять код ниже. Мы загружаем изображения в формате TGA в этом уроке вместо изображений в формате BMP. Структура ниже используется, чтобы хранить данные изображения, также как информацию об изображении в формате TGA. Прочитайте урок по работе с TGA файлами, если Вы нуждаетесь в детальном объяснении кода ниже.

```
typedef struct         // Создаем структуру
{
  GLubyte  *imageData; // Данные изображения
  GLuint  bpp;         // Цветность изображения в битах на пиксель.
  GLuint  width;       // Ширина изображения
  GLuint  height;      // Высота изображения
  GLuint  texID;       // ID текстуры используемый для выбора текстуры
} TextureImage;        // Имя структуры
```

Следующий код задает место для хранения наших 10 текстур и 30 объектов. Если Вы хотите добавить еще объектов в игру, то проверьте, что Вы изменили значение от 30 на то число объектов, которое Вы хотите.

```
TextureImage textures[10]; // Место для хранения 10 текстур
objects  object[30];       // Место для хранения 30 объектов
```

Я не хочу ограничивать размер каждого объекта. Я хочу, чтобы ваза была более высокой, чем ведро, а ведро было более широкое, чем ваза. Чтобы упростить жизнь, я создаю структуру, в которой есть ширина объектов (w) и высота (h).

Затем я задаю ширину и высоту каждого объекта в последней строке кода. Чтобы получить ширину банки кока-колы (coke), я буду использовать вызов size[3].w. Для голубой рожи (Blueface) - 0, ведро (Bucket) - 1, мишень (Target) - 2, банка кока-колы (Coke) - 3 и ваза (Vase) - 4. Ширина задается w. Понятно?

```
struct dimensions {       // Размеры объекта
  GLfloat  w;             // Ширина объекта
  GLfloat h;              // Высота объекта
};

// Размеры для объектов:Blueface,     Bucket,     Target,     Coke,         Vase
dimensions size[5] = { {1.0f,1.0f}, {1.0f,1.0f}, {1.0f,1.0f}, {0.5f,1.0f}, {0.75f,1.5f} };
```

Следующий большой раздел кода загружает наше TGA изображение и конвертирует ее в текстуру. Это - тот же самый код, так который я использовал в уроке 25, если Вы нуждаетесь в детальном описании его, обратитесь к этому уроку.

Я использую изображения TGA, потому что у них есть возможность сохранять альфа-канал. Альфа-канал сообщает OpenGL, какие части изображения будут прозрачными, а какие части непрозрачны. Альфа-канал создается в программе редактирования изображений, и сохраняется вместе с изображением в файле TGA. OpenGL загружает изображение, и использует альфа-канал для задания величины прозрачности каждого пикселя в изображении.

```
bool LoadTGA(TextureImage *texture, char *filename)   // Загрузка TGA файла в память
{
  GLubyte    TGAheader[12]={0,0,2,0,0,0,0,0,0,0,0,0}; // Заголовок несжатого TGA
  GLubyte    TGAcompare[12]; // Используется для сравнения заголовка TGA
  GLubyte    header[6];      // Первые 6 полезных байт заголовка
  GLuint     bytesPerPixel;  // Число байт на пиксель в файле TGA
  GLuint     imageSize;      // Используется для сохранения размера изображения
  GLuint     temp;           // Временная переменная
  GLuint     type=GL_RGBA;   // Режим GL по-умолчанию RBGA (32 BPP)

  FILE *file = fopen(filename, "rb"); // Открыть TGA файл

  // Если файл существует и 12 байт прочитаны и заголовок совпал и прочитаны
  // следующие 6 байт, то все нормально, иначе не удача
  if( file==NULL ||
    fread(TGAcompare,1,sizeof(TGAcompare),file)!=sizeof(TGAcompare) ||
    memcmp(TGAheader,TGAcompare,sizeof(TGAheader))!=0 ||
    fread(header,1,sizeof(header),file)!=sizeof(header))
  {
    if (file == NULL)         // Файл не существует? *Добавлено Jim Strong*
      return FALSE;           // вернуть FALSE
    else                      // иначе
    {
      fclose(file);           // Закрыть файл
      return FALSE;           // вернуть FALSE
    }
  }

  texture->width  = header[1] * 256 + header[0]; // Определить ширину (highbyte*256+lowbyte)
  texture->height = header[3] * 256 + header[2]; // Определить высоту (highbyte*256+lowbyte)
```

```
  if( texture->width <=0 ||            // Ширина меньше или равна чем 0
      texture->height <=0 ||           // Высота меньше или равна чем 0
      (header[4]!=24 && header[4]!=32)) // TGA 24 или 32 бита?
  {
   fclose(file);             // Если не так, то закрыть файл
   return FALSE;             // вернуть FALSE
  }

  texture->bpp = header[4];             // Получить число бит на пиксель (24 или 32)
  bytesPerPixel = texture->bpp/8;       // Разделить на 8, чтобы получить байт на пиксель
                                // Вычислить размер памяти для данных TGA
  imageSize = texture->width*texture->height*bytesPerPixel;

  texture->imageData=(GLubyte *)malloc(imageSize); // Выделить память для данных TGA

  if( texture->imageData==NULL ||       // Память выделена?
      // Прочитано верное число байт?
     fread(texture->imageData, 1, imageSize, file)!=imageSize)
  {
   if(texture->imageData!=NULL) // Если память выделена
      free(texture->imageData); // Освободить память

   fclose(file);               // Закрыть файл
   return FALSE;               // вернуть FALSE
  }

 for(GLuint i=0; i<int(imageSize); i+=bytesPerPixel) // Цикл по данным
  {                 // Смена местами первого и третьего байтов ('R'ed и 'B'lue)
   temp=texture->imageData[i]; // Временно сохраняем значение
   texture->imageData[i] = texture->imageData[i + 2]; // Третий байт на место первого
   texture->imageData[i + 2] = temp; // Первый байт на место третьего
  }

  fclose (file); // Закрыть файл

  // Построить текстуру из данных
  glGenTextures(1, &texture[0].texID); // Генерировать ID текстуры OpenGL
  glBindTexture(GL_TEXTURE_2D, texture[0].texID); // Привязка текстуры
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);  // Линейная
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);  // фильтрация

  if (texture[0].bpp==24) // Если TGA 24 бита
  {
   type=GL_RGB;          // Тогда 'type' равен GL_RGB
  }

  glTexImage2D(GL_TEXTURE_2D, 0, type,
               texture[0].width, texture[0].height,
               0, type, GL_UNSIGNED_BYTE, texture[0].imageData);
  return true; // Текстура построена, вернуть True
}
```

Код вывода шрифта из 2D текстуры тот же самый, который я использовал в предыдущих уроках. Однако есть несколько небольших изменений. Первое, что Вы можете заметить это то, что мы генерируем только 95 списков отображения. Если Вы посмотрите на текстуру шрифта, Вы увидите, что там есть только 95 символов. Второе, что Вы можете заметить то, что мы делим на 16.0f для получения cx, и мы делим на 8.0f для cy. Это делается, потому что текстура шрифта широкая (256 пикселей), а высота равна половине ширины (128 пикселей). Поэтому, вычисляя cx, мы делим на 16.0f, и, вычисляя cy, мы делим на 8.0f.

Если Вы не понимаете код ниже, возвратитесь к уроку 17. Код вывода шрифта подробно объясняется в уроке 17!

```
GLvoid BuildFont(GLvoid)              // Построить наш список отображения для фонта
{
 base=glGenLists(95);                 // Создание 95 списков отображения
 glBindTexture(GL_TEXTURE_2D, textures[9].texID); // Привязка нашей текстуры фонта
 for (int loop=0; loop<95; loop++)   // Цикл по спискам
  {
   float cx=float(loop%16)/16.0f;    // X позиция текущего символа
   float cy=float(loop/16)/8.0f;     // Y позиция текущего символа

   glNewList(base+loop,GL_COMPILE);  // Начало построения списка
    glBegin(GL_QUADS);               // Использовать четырехугольник для символа
      // Координаты текстуры / вершин (Низ Лево)
      glTexCoord2f(cx,          1.0f-cy-0.120f); glVertex2i(0,0);
      // Координаты текстуры / вершин (Низ Право)
      glTexCoord2f(cx+0.0625f, 1.0f-cy-0.120f);  glVertex2i(16,0);
      // Координаты текстуры / вершин (Верх Право)
      glTexCoord2f(cx+0.0625f, 1.0f-cy);         glVertex2i(16,16);
      // Координаты текстуры / вершин (Низ Лево)
      glTexCoord2f(cx,         1.0f-cy);         glVertex2i(0,16);
    glEnd();                         // Конец построения нашего четырехугольника (символ)
    glTranslated(10,0,0);            // Сдвиг вправо на символ
   glEndList();                      // Завершение построения списка отображения
  }                                  // Цикл по всем символам
}
```

Код вывода строки такой же, как в уроке 17, но был изменен, чтобы дать возможность нам вывести счет, уровень и мораль на экран (переменные, которые непрерывно изменяются).

```
GLvoid glPrint(GLint x, GLint y, const char *string, ...) // Здесь печать
{
 char       text[256];            // Место для строки
 va_list    ap;                   // Указатель на список аргументов

 if (string == NULL)              // Если нет текста
  return;                         // то ничего не делаем

 va_start(ap, string);            // Разбор строки из переменных
    vsprintf(text, string, ap); // Конвертирование символов в числа
 va_end(ap);                      // Значения сохраняются в текст
 glBindTexture(GL_TEXTURE_2D, textures[9].texID);   // Выбор нашей текстуры шрифта
 glPushMatrix();                  // Сохранить матрицу вида модели
 glLoadIdentity();                // Сброс матрицы вида модели
 glTranslated(x,y,0);             // Позиционирование текста (0,0 – низ лево)
 glListBase(base-32);             // Выбор набора символов
 glCallLists(strlen(text), GL_UNSIGNED_BYTE, text); // Нарисовать текст списками
 glPopMatrix();                   // Восстановить матрицу
}
```

Этот код будет вызываться позже из функции qsort. Он нужен для сравнения расстояний в двух структурах и возвращает значение -1, если расстояние в первой структуре было меньше чем во второй, и значение 1, если расстояние в первой структуре больше чем во второй, и значение 0, если расстояние в обеих структурах равно.

```
int Compare(struct objects *elem1, struct objects *elem2) // Функция сравнения
{
  // Если расстояние в первой структуре меньше чем во второй
  if ( elem1->distance < elem2->distance)
    return -1; // Вернуть –1
  // Если расстояние в первой структуре больше чем во второй
  else if (elem1->distance > elem2->distance)
    return 1;  // Вернуть 1
  else         // Иначе (Если расстояние равно)
    return 0;  // Вернуть 0
}
```

В коде функции InitObject() мы инициализируем каждый объект. Мы начинаем, с установки rot в 1. При этом объект будет вращаться по часовой стрелке. Затем мы инициализируем анимацию взрыва для кадра 0 (мы не хотим, чтобы взрыв был показан с половины анимационной последовательности). Затем мы устанавливаем hit в ЛОЖЬ, что означает, что объект еще не был сбит или подвергся самоуничтожению. Для выбора текстурного объекта, переменной texid присваивается случайное значение от 0 до 4. Ноль - текстура blueface, и 4 - текстура vase. Это даст нам один из 5 случайных объектов.

Переменной distance будет присвоено случайное число от -0.0f до -40.0f (4000/100 равно 40). Когда мы будем рисовать объект, мы смещаем его на 10 единиц в экран. Поэтому, когда объект нарисован, он будет нарисован от -10.0f до -50.0f единиц в глубине экрана (и не близко и не далеко). Я делю случайное число на 100.0f, чтобы получить более точное значение с плавающей запятой.

После того как мы задали случайное значение дистанции до объекта, мы задаем объекту случайное значение по y. Мы не хотим, чтобы объект был ниже чем -1.5f, иначе он будет под землей, и мы не хотим, чтобы объект был выше, чем 3.0f. Поэтому диапазон наших случайных чисел не может быть больше чем 4.5f (-1.5f+4.5f=3.0f).

Для вычисления позиции x, мы используем довольно хитрый способ. Мы берем наше расстояние, и мы вычитаем 15.0f из него. Затем мы делим результат на 2 и вычитаем 5*level. Наконец, мы вычитаем случайное число от 0.0f до 5 умноженное на текущий уровень. Мы вычитаем 5*level и случайное число от 0.0f до 5*level так, чтобы наш объект появлялся дальше от экрана на более высоких уровнях. Если бы мы не делали этого, объекты появились бы один за другим, при этом попасть в них было бы труднее, чем в этом варианте.

Наконец мы выбираем случайное направление (dir) от 0 (слева) до 1 (направо).

Вот пример, чтобы вам было все это понятно. Скажем, что расстояние равно -30.0f, а текущий уровень равен 1:

```
object[num].x=((-30.0f-15.0f)/2.0f)-(5*1)-float(rand()%(5*1));
object[num].x=(-45.0f/2.0f)-5-float(rand()%5);
object[num].x=(-22.5f)-5-{давайте скажем 3.0f};
object[num].x=(-22.5f)-5-{3.0f};
object[num].x=-27.5f-{3.0f};
object[num].x=-30.5f;
```

Запомним, что мы сдвигаем на 10 единиц в экран прежде, чем мы выводим наши объекты, и расстояние в примере выше равно -30.0f. Можно с уверенностью сказать, что наше фактическое расстояние вглубь экрана будет равно -40.0f. Используя код перспективы из файла NeHeGL.cpp, с уверенностью можно будет предположить, что, если расстояние равно -40.0f, левый край экрана будет -20.0f, и правый край будет +20.0f. В коде выше наше значение x равно -22.5f (т.е. за левым краем экрана). Затем мы вычитаем 5 и наше случайное значение 3, которое гарантирует, что объект начнет перемещаться за экраном (с -30.5f) - это означает, что объект должен будет переместить приблизительно на 8 единиц вправо прежде, чем он появится на экране.

```
GLvoid InitObject(int num)     // Инициализация объекта
{
  object[num].rot=1;           // Вращение по часовой
  object[num].frame=0;         // Сброс кадра взрыва в ноль
  object[num].hit=FALSE;       // Сброс статуса попадания в объект
  object[num].texid=rand()%5;  // Назначение новой текстуры
  object[num].distance=-(float(rand()%4001)/100.0f);  // Случайная дистанция
  object[num].y=-1.5f+(float(rand()%451)/100.0f);     // Случайная Y позиция
  // Случайное начальная X позиция, основанная на расстоянии объекта
  // и случайном числе для задержки (Положительное значение)
  object[num].x=((object[num].distance-15.0f)/2.0f)-(5*level)-float(rand()%(5*level));
  object[num].dir=(rand()%2);  // Взять случайное направление
```

Теперь мы проверим, в каком направлении объект собирается лететь. Код ниже проверяет, перемещается ли объект влево. Если это так, то мы должны изменить вращение так, чтобы объект вращался против часовой стрелки. Мы делаем это, изменяя значение rot на 2.

Наше значение x по умолчанию будет отрицательным числом. Однако на правой стороне экрана будут положительные значения. Поэтому в завершении мы инвертируем знак текущего значения x. По-русски говоря, мы делаем положительное значение x вместо отрицательного значения.

```
 if (object[num].dir==0)          // Направление вправо
 {
  object[num].rot=2;              // Вращение против часовой стрелки
  object[num].x=-object[num].x;  // Начнем с левой стороны (Отрицательное значение)
 }
```

Теперь мы проверяем texid, чтобы выяснить какой случайный объект компьютер выбрал. Если texid равно 0, то компьютер выбрал объект blueface. Парни с голубыми рожами всегда катятся по земле. Чтобы быть уверенными, что они начинают свое путешествие на наземном уровне, мы принудительно устанавливаем значение y в -2.0f.

```
 if (object[num].texid==0)        // Голубая рожа
   object[num].y=-2.0f;           // Всегда катится по земле
```

Затем мы проверяем, равно ли texid 1. Если это так, то компьютер выбрал ведро. Ведро не путешествует слева направо, оно падает с неба. Поэтому вначале мы должны установить dir в 3. Это сообщит компьютеру, что наш ковш падает или перемещается вниз.

Наш код инициализации предполагает, что объект будет путешествовать слева направо. Поскольку ковш падает, мы должны дать ему новое случайное значение по x. Если бы мы этого не делали, ковш никогда не был бы видим. Он падал бы или за левым краем экрана или за правым краем экрана. Чтобы назначить новое значение, мы выбираем случайное значение, которое получается на основании расстояния от экрана. Вместо того чтобы вычитать 15, мы вычитаем только 10. Это дает нам более маленький диапазон, и сохраняет объект на экране. Назначив наше расстояние в -30.0f, мы будем иметь случайное значение от 0.0f до 40.0f. Если Вы спрашиваете себя, почему от 0.0f до 40.0f? Разве оно не должно быть от 0.0f до -40.0f? Ответ прост. Функция rand() всегда возвращает положительное число. Поэтому независимо от числа, мы получим обратно положительное значение. Так или иначе... вернемся. Поэтому мы имеем положительное число от 0.0f до 40.0f. Затем мы добавляем расстояние (отрицательное значение) минус 10.0f деленное на 2. Для примера ... пусть случайное значение 15, а расстояние равно -30.0f:

```
object[num].x=float(rand()%int(-30.0f-10.0f))+((-30.0f-10.0f)/2.0f);
object[num].x=float(rand()%int(-40.0f)+(-40.0f)/2.0f);
object[num].x=float(15 {пусть будет 15))+(-20.0f);
object[num].x=15.0f-20.0f;
object[num].x=-5.0f;
```

В завершении мы должны задать значение y. Мы хотим, чтобы ведро падало с неба. Мы не хотим, что бы оно проходило сквозь облака. Поэтому мы устанавливаем значение y в 4.5f. Немного ниже облаков.

```
 if (object[num].texid==1)        // Ведро
 {
  object[num].dir=3;              // Падает вниз
  object[num].x=float(rand()%int(object[num].distance-10.0f))+
                ((object[num].distance-10.0f)/2.0f);
  object[num].y=4.5f;             // Случайное X, начинаем с верха экрана
 }
```

Мы хотим, чтобы мишень вылетела из земли и поднялась в воздух. Мы проверяем объект действительно мишень (texid ранвно 2). Если это так, то мы задаем направление (dir) равным 2 (верх). Мы используем точно тот же самый код, как и выше, чтобы получить случайное положение по x.

Мы не хотим, чтобы мишень вылетала выше земли. Поэтому мы задаем начальное значение y равным -3.0f (под землей). Затем мы вычитаем случайное значение от 0.0f до 5 умноженное на текущий уровень. Мы делаем это затем, чтобы мишень НЕМЕДЛЕННО НЕ появилась. На более высоких уровнях мы хотим ввести задержку прежде, чем мишень появится. Без задержки, мишени бы вылетали одна за другой, отводя Вам, мало времени, чтобы сбить их.

```
 if (object[num].texid==2)        // Мишень
 {
  object[num].dir=2;              // Вверх
  // Случайное X, старт из под земли + случайное значение
  object[num].x=float(rand()%int(object[num].distance-10.0f))+
                ((object[num].distance-10.0f)/2.0f);
  object[num].y=-3.0f-float(rand()%(5*level));
 }
```

Все другие объекты путешествуют слева направо, поэтому нет смысла для них что-то менять. Они должны прекрасно работать со случайными значениями, которые уже были назначены.

Теперь интересное! "Для правильно работы техники использующей альфа смешивание прозрачные примитивы должны быть выведены от дальних к ближним и не должны пересекаться". Когда рисуются объекты с альфа смешиванием очень важно, чтобы дальние объекты были выведены вначале, а ближние объекты выведены последними.

Причина этого проста... Z буфер не допускает рисование OpenGL пикселей, которые уже сзади выведенных пикселей. Поэтому может так случится, что объекты, выведенные сзади прозрачных объектов, не обнаруживаются. Поэтому Вы можете увидеть квадратную форму вокруг перекрывающихся объектов... Не хорошо!

Мы уже знаем глубину каждого объекта. Поэтому после инициализации нового объекта, мы можем обойти эту проблему, сортируя объекты, используя функцию qsort (быстрая сортировка). При помощи сортировки объектов, мы можем убедиться, что первый выведенный объект – это тот объект, который дальше всего. Поэтому, когда мы выводим объекты, мы начинаем с первого объекта, дальние объекты будут выведены вначале. Ближние объекты будут видеть предварительно выведенные объекты позади них, и смешивание пройдет должным образом!

Как отмечено в комментариях ниже: я нашел этот код в MSDN после нескольких часов поиска в Интернете. Этот код работает хорошо и позволяет Вам сортировать структуры целиком. Функция qsort имеет 4 параметра. Первый параметр указывает на массив объектов (массив, который нужно сортировать). Второй параметр - число массивов, которые мы хотим сортировать... Конечно, мы хотим сортировать все объекты, которые в настоящее время отображаются (число объектов задается уровнем). Третий параметр определяет размер нашей структуры объектов, и четвертый параметр указывает на нашу функцию сравнения.

Есть, вероятно, более лучший способ сортировать структуры, но qsort () работает... Это быстро и удобно!

Важно обратить внимание на то, что, если Вы хотите использовать glAlphaFunc() и glEnable (GL_ALPHA_TEST), в сортировке нет необходимости. Однако, используя только альфа-тест (позволяет принять или отклонить фрагмент, основываясь на значение его альфа-канала, но не смешивает) Вы ограничены полностью прозрачным или полностью непрозрачным смешиванием, но при этом не возникает реального смешивания. С сортировкой и использованием Blendfunc() надо немного больше работы, но при этом учитываются полупрозрачные объекты.

```
  // Сортировка объектов по расстоянию:
  //  *** Код MSDN модифицирован в этом уроке ***
  //              Начальный адрес нашего массива объектов
  //              Число сортируемых элементов
  //              Размер каждого элемента
  //              Указатель на нашу функцию сравнения
  qsort((void *) &object, level, sizeof(struct objects), (compfn)Compare );
}
```

Код инициализации тот же самый, как и всегда. В первых двух строках сохраняется информация о нашем окне и нашем обработчике клавиатуры. Затем используем srand() чтобы сделать нашу игру более случайную, основываясь на времени. После этого мы загружаем наши TGA изображения и конвертируем их в текстуры, используя LoadTGA(). Первые 5 изображений - объекты, которые будут летать по экрану. Далее, загрузим текстуры Explode (взрыв) – анимация взрыва, ground (земля) и sky (небо) - фон сцены, crosshair (перекрестье) - курсор, который показывает текущее положение мыши на экране, и, наконец, font - шрифт для отображения счета, заголовка, и морали. Если какое-то из изображений не загрузиться, будет возвращено ЛОЖЬ, и программа закроется. Важно обратить внимание на то, что этот основной код не будет выводить сообщение о неудавшейся инициализации.

```
BOOL Initialize (GL_Window* window, Keys* keys) // Инициализация OpenGL
{
  g_window  = window;
  g_keys    = keys;

  srand( (unsigned)time( NULL ) );          // Привнесение случайности

  if ((!LoadTGA(&textures[0],"Data/BlueFace.tga")) ||// Загрузка текстуры BlueFace
    (!LoadTGA(&textures[1],"Data/Bucket.tga")) ||    // Загрузка текстуры Bucket
    (!LoadTGA(&textures[2],"Data/Target.tga")) ||    // Загрузка текстуры Target
    (!LoadTGA(&textures[3],"Data/Coke.tga")) ||      // Загрузка текстуры Coke
    (!LoadTGA(&textures[4],"Data/Vase.tga")) ||      // Загрузка текстуры Vase
```

```
    (!LoadTGA(&textures[5],"Data/Explode.tga")) ||   // Загрузка текстуры Explosion
    (!LoadTGA(&textures[6],"Data/Ground.tga")) ||    // Загрузка текстуры Ground
    (!LoadTGA(&textures[7],"Data/Sky.tga")) ||       // Загрузка текстуры Sky
    (!LoadTGA(&textures[8],"Data/Crosshair.tga")) || // Загрузка текстуры Crosshair
    (!LoadTGA(&textures[9],"Data/Font.tga")))        // Загрузка текстуры Font
  {
    return FALSE;              // Если не удачно, то вернем False
  }
```

Если все изображения были загружены и конвертированы в текстуры успешно, мы можем продолжать инициализацию. Текстура шрифта загружена, поэтому можно создать наш шрифт. Мы делаем это, вызывая BuildFont().

Затем мы настраиваем OpenGL. Цвет фона - черный, альфа в 0.0f. Буфер глубины включен и разрешен с тестом меньше или равно.

Функция glBlendFunc() - очень важная строка кода. Мы задаем функцию смешивания как (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA). При этом объект смешивается со всем, что на экране, используя альфа значения, которые есть в текстуре объекта. После настройки режима смешивания, мы разрешаем смешивание. Затем разрешаем 2D наложение текстуры, и, наконец, мы разрешаем GL_CULL_FACE. При этом удаляются задние грани каждого объекта (нет смысла выводить то, что мы не видим). Мы выводим все наши четырехугольники против часовой стрелки, поэтому будет выбрана правильная грань.

Ранее я говорил об использовании glAlphaFunc() вместо альфа смешивания. Если Вы хотите использовать альфа тест, закомментируйте 2 строки кода смешивания и уберите комментарий с 2 строк сразу за glEnable(GL_BLEND). Вы можете также закомментировать функцию qsort() в InitObject().

Программа запуститься, но текстуры неба при этом не будет. Причина этого в том, что текстура неба имеет альфа-значение равное 0.5f. Когда я говорил об альфа тесте раньше, я упомянул, что он работает только с альфа значениями 0 или 1. Вы должны будете изменить альфа канал для текстуры неба, если Вы хотите, чтобы оно появилось! Еще раз, если Вы хотите использовать альфа тест, Вы не должны сортировать объекты. Оба метода хороши! Ниже небольшая цитата с сайта SGI:

"Альфа тест отклоняет фрагменты вместо рисования их в буфер кадра. Поэтому сортировка примитивов не нужна (если другой какой-то режим подобно альфа смешиванию не разрешен). Недостаток альфа теста в том, что пиксели должны быть полностью непрозрачные или полностью прозрачные".

```
  BuildFont();                 // Построение списков отображения для шрифта

  glClearColor(0.0f, 0.0f, 0.0f, 0.0f); // Черный фон
  glClearDepth(1.0f);          // Настройка буфера глубины
  glDepthFunc(GL_LEQUAL);      // Тип теста глубины
  glEnable(GL_DEPTH_TEST);     // Разрешен тест глубины
  // Разрешено альфа-смешивание (запрет альфа теста)
  glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
  glEnable(GL_BLEND);          // Разрешено смешивание  (запрет альфа теста)
//  glAlphaFunc(GL_GREATER,0.1f);  // Настройка альфа теста (запрет смешивания)
//  glEnable(GL_ALPHA_TEST);     // Разрешен альфа тест   (запрет смешивания)
  glEnable(GL_TEXTURE_2D);     // Разрешено наложение текстуры
  glEnable(GL_CULL_FACE);      // Удалить заднюю грань
```

До этой части программы, ни один из объектов не был задан. Поэтому мы делаем цикл по всем тридцати объектам, вызывая InitObject() для каждого объекта.

```
  for (int loop=0; loop<30; loop++) // Цикл по 30 объектам
    InitObject(loop);          // Инициализация каждого объекта
  return TRUE;                 // Возврат TRUE (Инициализация успешна)
}
```

В нашем коде инициализации, мы вызывали BuildFont(), для создания наших 95 списков отображения. Следующая строка кода удаляет все 95 списков отображения перед выходом программы.

```
void Deinitialize (void)          // Любая пользовательская деинициализация здесь
{
  glDeleteLists(base,95);          // Уничтожить все 95 списков отображения фонта
}
```

Теперь немного хитрого кода ... Кода, который фактически делает выбор объектов. В первой строке кода ниже создается буфер, который мы можем использовать, чтобы хранить информацию о наших выбранных объектах. В переменной попадания (hits) будет храниться число обнаруженных попаданий в режиме выбора.

```
void Selection(void)              // Здесь происходит выбор
{
  GLuint  buffer[512];            // Настройка буфера выбора
  GLint  hits;                   // Число объектов, которые мы выбрали
```

Вначале, мы проверяем, окончена ли игра. Если это так, то нет никакого смысла выбирать, поэтому мы производим выход из функции. Если игра еще не окончена, мы проигрываем звук выстрела с использование функции Playsound(). Функция Selection() вызывается только тогда, когда кнопка мыши была нажата, и каждый раз, когда кнопка мыши нажата, мы хотим проиграть звук выстрела. Звук запускается в асинхронном режиме для того, чтобы не остановить программу, во время проигрывания звука.

```
  if (game)              // Игра окончена?
    return;              // Если так, то выбирать нечего

  PlaySound("data/shot.wav",NULL,SND_ASYNC); // Проигрывание звука выстрела
```

Теперь мы настроим область просмотра. Массив viewport[] хранит x, y, длину и ширину текущей области просмотра (окно OpenGL).

Функция glGetIntegerv(GL_VIEWPORT, viewport) возвращает границы текущей области просмотра и помещает viewport[]. Первоначально, границы равны размерам окна OpenGL. Вызов функции glSelectBuffer(512, buffer) сообщает OpenGL, что надо использовать buffer для буфера выбора.

```
  // Размер области просмотра. [0] - <x>, [1] - <y>, [2] - <length>, [3] - <width>
  GLint  viewport[4];

  // Помещаем в массив <viewport> размеры и положение на экране относительно окна
  glGetIntegerv(GL_VIEWPORT, viewport);
  glSelectBuffer(512, buffer); // Скажем OpenGL использовать этот массив для выбора
```

Следующий код очень важен. В первой строке OpenGL переключается в режим выбора. В режиме выбора, ничего не выводится на экран. Вместо этого, вся информация о визуализированных объектах будет сохранена в буфере выбора.

Далее мы инициализируем стек имен, вызывая glInitNames() и glPushName(0). Важно обратить внимание на то, что, если программа не в режиме выбора, запрос к glPushName() будет игнорирован. Конечно, мы находимся в режиме выбора, но это важно иметь в виду.

```
  // Переключить OpenGL в режим выбора. Ничего не будет нарисовано.
  // Идентификатор объекта и его размеры будут сохранены в буфере.
  (void) glRenderMode(GL_SELECT);

  glInitNames();               // Инициализация стека имен
  glPushName(0);               // Поместить 0 в стек (наименьший первый элемент)
```

После подготовки стека имен, мы должны ограничить область рисования только под нашим курсором. Чтобы это сделать, мы должны выбрать матрицу проецирования. После выбора матрицы проецирования мы помещаем ее в стек. Затем сбрасываем матрицу проецирования используя glLoadIdentity().

Мы ограничим рисование, используя gluPickMatrix() (задает область выбора). Первый параметр - наша текущая позиция мыши по оси X, второй параметр - текущая позиция мыши по оси Y, затем ширина и высота области выбора. Наконец, viewport[]. Массив viewport[] указывает текущие границы области просмотра. Переменные mouse_x и mouse_y будут в центре области выбора.

```
glMatrixMode(GL_PROJECTION);   // Выбор матрицы проецирования
glPushMatrix();                // Поместить матрицу проецирования
glLoadIdentity();              // Сброс матрицы

// Создание матрицы, которая будет задавать маленькую часть экрана под мышью.
gluPickMatrix((GLdouble) mouse_x, (GLdouble) (viewport[3]-mouse_y), 1.0f, 1.0f, viewport);
```

Вызов gluPerspective() умножает перспективную матрицу на матрицу выбора, которая ограничивает область рисования, которая задана gluPickMatrix().
Затем мы выбираем матрицу вида модели и выводим наши цели при помощи вызова DrawTargets(). Мы выводим цели в DrawTargets(), а не в Draw() потому что мы хотим, чтобы в режиме выбора были проверены попадания только в объекты (цели), а не с небом, землей или курсором.
После отрисовки наших целей, мы выбираем снова матрицу проецирования и возвращаем сохраненную матрицу из стека. Затем мы выбираем снова матрицу вида модели.
В последней строке кода ниже мы переключаемся обратно в режим визуализации так, чтобы объекты, которые мы выводим, появились на экране. Переменная hits будет содержать число объектов, которые были визуализированы в области просмотра, которая задана gluPickMatrix().

```
// Применить перспективную матрицу
gluPerspective(45.0f, (GLfloat) (viewport[2]-viewport[0])/(GLfloat) (viewport[3]-viewport[1]),
          0.1f, 100.0f);
glMatrixMode(GL_MODELVIEW);   // Выбор матрицы вида модели
DrawTargets();                // Визуализация целей в буфер выбора
glMatrixMode(GL_PROJECTION);  // Выбор матрицы проецирования
glPopMatrix();                // Получить матрицу проецирования
glMatrixMode(GL_MODELVIEW);   // Выбор матрицы вида модели
hits=glRenderMode(GL_RENDER); // Выбор режима визуализации, найти как много
```

Теперь мы проверяем больше ли нуля число зарегистрированных попаданий. Если это так, то мы присваиваем переменной choose имя первого объекта, нарисованного в области выбора. Переменной depth присваивается значение расстояния объекта от экрана (глубина).
Каждое попадание помещает 4 элемента в буфер. Первый элемент - число имен в стеке имен, когда произошло попадание. Второй элемент - значение минимума z всех вершин, которые пересекают видимую область во время попадания. Третий элемент - значение максимума z всех вершин, которые пересекают видимую область во время попадания, и последний элемент – содержимое стека имен во время попадания (имя объекта). В этом уроке нам необходимо только значение минимума z и имя объекта.

```
if (hits > 0)              // Если есть попадания
{
  int  choose = buffer[3];   // Сделать наш выбор первым объектом
  int  depth = buffer[1];    // Сохранить как далеко он
```

Затем мы делаем цикл по всем попаданиям, для того чтобы проверить, что ни один из объектов не ближе чем первый объект. Если бы мы этого не сделали, и два объекта будут накладываться один на другой, и первый объект будет позади другого объекта, то по щелчку мыши будет выбран первый объект, даже притом, что он был позади другого объекта. Когда Вы стреляете, то самый близкий объект должен быть выбран.

Поэтому, мы проверяем все попадания. Вспомните, что на каждый объект приходится 4 элемента в буфере, поэтому чтобы перейти на следующее попадание мы должны умножить текущее значение цикла на 4. Мы добавляем 1, чтобы получить глубину каждого объекта. Если глубина - меньше чем текущая глубина выбранного объекта, мы сохраняем имя более близкого объекта в choose, и мы сохраняем в depth глубину более близкого объекта. После завершения цикла по всем нашим попаданиям, в choose будет находиться имя самого близкого объекта, а в depth будет содержаться глубина его.

```
  for (int loop = 1; loop < hits; loop++) // Цикл по всем обнаруженным попаданиям
  {
    // Если этот объект ближе, чем выбранный
    if (buffer[loop*4+1] < GLuint(depth))
    {
      choose = buffer[loop*4+3];          // Выбрать ближний объект
      depth = buffer[loop*4+1];           // Сохранить как далеко он
    }
  }
```

Все, что мы должны сделать – пометить, что в этот объект попали. Мы проверяем, что объект уже не был помечен. Если он не был помечен, то мы отмечаем, что в него попали, задавая hit ИСТИНА. Мы увеличиваем счет игрока на единицу, и мы увеличиваемся счетчик поражений на 1.

```
    if (!object[choose].hit)          // Если в объект еще не попадали
    {
      object[choose].hit=TRUE;        // Пометить, что в объект попали
      score+=1;             // Увеличить счет
      kills+=1;             // Увеличить число поражений
```

Я использую kills, чтобы знать, сколько объектов были уничтожены на каждом уровне. Я хочу на каждом уровне иметь большее количество объектов (чтобы каждый следующий уровень проходился тяжелее). Поэтому я проверяю, если игрок уничтожил объектов, больше чем значение текущего уровня, умноженного на 5, то он переходит на следующий уровень. На уровне 1, игрок должен уничтожить 5 объектов (1*5). На уровне 2 игрок должен уничтожить 10 объектов (2*5). Уровень трудности прогрессивно повышается.

Поэтому, в первой строке кода проверки смотрим, выше ли число поражений, чем уровень, умноженный на 5. Если это так, то мы устанавливаем miss в 0. Это задает боевой дух игрока обратно в 10 из 10 (боевой дух равен с 10-miss). Затем устанавливаем число поражений в 0 (начинаем процесс подсчета снова).

Наконец, мы увеличиваем значение уровня 1 и проверяем, достигли ли мы последнего уровня. Я установил максимальный уровень в 30 по следующим двум причинам... Уровень 30 безумно труден. Я уверен, что никто не доиграет до этого уровня. Вторая причина... Выше, мы задаем 30 объектов. Если Вы хотите большее количество объектов, Вы должны увеличить значение соответственно.

Очень важно обратить внимание, на что Вы можете иметь максимум 64 объекта на экране (0-63). Если Вы пробуете визуализировать 65 или более объектов, выбор становится путанным, и не верным. Каждый из объектов, случайно может привести к остановке вашего компьютера. Это - физический предел в OpenGL (точно так же как 8 источников света).

Если вдруг Вы - бог, и Вы закончили уровень 30, level больше не будет увеличиваться, и ваш счет тоже. Ваш боевой дух также сбросится к 10, каждый раз Вы, когда вы заканчиваете 30-ый уровень.

```
    if (kills>level*5)   // Новый уровень?
    {
      miss=0;            // Сброс числа промахов
      kills=0;           // Сброс число поражений
      level+=1;          // Увеличение уровня
      if (level>30)      // Больше чем 30?
        level=30;        // Поэтому уровень30 (Вы бог?)
    }
  }
 }
}
```

В функции Update() проверяется нажатие клавиш, и обновляется положение объектов. Одна из приятных особенностей Update() – миллисекундный таймер. Вы можете использовать миллисекундный таймер, чтобы переместить объекты, основываясь на времени, которое прошло, с того момента, как Update() была вызвана последний раз. Важно обратить внимание на то, что движущийся объект будет перемещаться с той же самой скоростью на любом процессоре... НО есть и недостатки! Пусть имеется объект, который перемещается на 5 единиц за 10 секунд. На быстрой системе, компьютер будет перемещать объект на половину единицы за каждую секунду. На медленной системе, это произойдет через 2 секунды прежде, чем даже процедура обновления будет вызвана. Поэтому, когда объект двигается, будет казаться, что он дергается. Мультипликация не будет плавной на более медленной системе. (Примечание: это несколько преувеличенный пример... любой компьютер будет обновлять экран быстрее, чем раз в две секунды).

Так или иначе... возвращаемся... к коду. В коде ниже делается проверка, для того чтобы увидеть, нажата ли клавиша выхода из программы. Если это так, то мы выходим из приложения, вызывая функцию TerminateApplication(). В переменной g_window находится информация о нашем окне.

```
void Update(DWORD milliseconds)              // Обновление движения здесь
{
 if (g_keys->keyDown[VK_ESCAPE])          // ESC нажата?
 {
  TerminateApplication (g_window);        // Прервать программу
 }
```

Далее проверяем, нажата ли клавиша "пробел", и при этом игра окончена. Если оба условия истинны, мы инициализируем все 30 объектов (даем им новые направления, присваиваем текстуры, и т.д). Мы устанавливаем game в ЛОЖЬ, сообщая программе, что игра продолжается. Мы сбрасываем score в 0, level в 1, kills в 0, и, наконец, мы устанавливаем переменную miss в ноль. При этом перезапуск игры будет с первым уровнем, с полной моралью и счетом 0.

```
 if (g_keys->keyDown[' '] && game)         // Пробел нажат после того как окончена игра?
 {
  for (int loop=0; loop<30; loop++)        // Цикл по 30 объектам
   InitObject(loop);      // Инициализация каждого объекта

  game=FALSE;              // Установка game в False
  score=0;                 // Установка score в 0
  level=1;                 // Установка level в 1
  kills=0;                 // Установка Kills в 0
  miss=0;                  // Установка miss в 0
 }
```

В коде ниже проверяет нажатие клавиши F1. Если клавиша F1 нажата, то ToggleFullscreen переключит из оконного в полноэкранный режим или из полноэкранного режима в оконный режим.

```
 if (g_keys->keyDown[VK_F1])           // F1 нажата?
 {
  ToggleFullscreen (g_window);         // Переключение видеорежима
 }
```

Для создания иллюзии движущихся облаков и перемещения земли, мы уменьшаем roll на 0.00005f, умноженное на число прошедших миллисекунд. При этом облака будут перемещаться с той же самой скоростью на всех системах (быстро или медленно).

Затем создаем цикл по всем объектам на экране. На уровне 1 имеется один объект, на уровне 10 имеется 10 объектов, и т.д.

```
 roll-=milliseconds*0.00005f;            // Прокрутка облаков

 for (int loop=0; loop<level; loop++)    // По всем объектам
 {
```

Мы должны выяснить, каким способом объект должен вращаться. Мы делаем это при помощи проверки значения rot. Если rot равняется 1, мы должны вращать объект по часовой стрелке. Для того чтобы сделать это, мы уменьшаем значение spin. Мы уменьшаем spin на 0.2f, умноженное на значение loop плюс число прошедших миллисекунд. Используя миллисекунды, объекты будут вращать с той же самой скоростью на всех системах. Добавляя loop, мы заставляем каждый НОВЫЙ объект вращаться чуточку быстрее, чем последний объект. Поэтому второй объект будет вращаться быстрее, чем первый объект, и третий объект будет вращаться быстрее, чем второй объект.

```
  if (object[loop].rot==1)          // Вращение по часовой
   object[loop].spin-=0.2f*(float(loop+milliseconds));
```

Затем проверим, равняется ли rot двум. Если rot равняется двум, то мы должны вращать против часовой стрелки. Единственное отличие от кода выше то, что мы увеличиваем значение вращения вместо уменьшения его. Это заставляет объект вращаться в противоположном направлении.

```
  if (object[loop].rot==2)          // Вращение против часовой
   object[loop].spin+=0.2f*(float(loop+milliseconds));
```

Теперь код перемещения. Мы проверяем значение dir, если оно равно 1, мы увеличиваем x позицию объекта на значение 0.012f умноженное на миллисекунды. При этом объект перемещается вправо. Поскольку мы используем миллисекунды, объекты должны перемещаться с той же самой скоростью на всех системах.

```
if (object[loop].dir==1)          // Смещение вправо
  object[loop].x+=0.012f*float(milliseconds);
```

Если dir равняется 0, объект перемещается влево. Мы перемещаем объект, влево уменьшая x координату объекта на прошедшее время в миллисекундах, умноженное на заранее заданное значение 0.012f.

```
if (object[loop].dir==0)          // Направление налево
  object[loop].x-=0.012f*float(milliseconds);
```

На сей раз, мы проверяем, равняется ли dir 2. Если это так, то мы увеличиваем y координату объекта. При этом объект двигается вверх по экрану. Имейте в виду, что положительная ось Y направлена вверх экрана, а отрицательная ось Y - вниз. Поэтому увеличение y перемещает объект снизу вверх. Снова используется время.

```
if (object[loop].dir==2)          // Направление вверх
  object[loop].y+=0.012f*float(milliseconds);
```

И последнее, если dir равняется три, то мы хотим переместить объект вниз экрана. Мы делаем это, увеличивая y координату объекта на прошедшее время. Заметьте, что мы передвигаем объекты вниз медленнее чем, мы вверх. Когда объект падает, наша константа падения 0.0025f. Когда объект поднимается - 0.012f.

```
if (object[loop].dir==3)          // Направление вниз
  object[loop].y-=0.0025f*float(milliseconds);
```

После перемещения наших объектов мы должны проверить попадают ли они еще в поле зрения. Код ниже проверяет, где наш объект находится на экране. Мы можем, приближено вычислить, как далеко сдвинулся объект влево, взяв расстояние объекта от экрана минус 15.0f (чтобы быть уверенными, что это небольшой выход за экран) и разделив это на 2. Для тех, кто не знает... Если Вы сдвинулись на 20 единиц в экран, в зависимости от способа, которым Вы задаете перспективу, Вы имеете примерно 10 единиц в левой части экрана и 10 в правой части. Поэтому -20.0f (расстояние)-15.0f (дополнительно) =-35.0f... делим это на 2, и получаем -17.5f. Это примерно на 7.5 единиц за левым краем экрана. Что означает, что наш объект полностью вне поля зрения.

Так или иначе... После проверки, как далеко объект за левой стороной экрана, мы проверяем, двигается ли он влево (dir=0). Если он не перемещается влево, то нам не надо заботимся о том, что он за левым краем экрана!

Наконец, мы проверяем, было ли попадание в объект. Если объект за экраном, он перемещается влево, и в него не попали, то в этом случае игрок в него уже никогда не попадет. Поэтому мы увеличиваем значение miss. При этом понижаем боевой дух и увеличивает число не сбитых объектов. Мы задаем значение hit объекта в ИСТИНА, поэтому компьютер будет думать, что в этот объект попали. Это вынуждает объект самоуничтожится (позволить нам присвоить объекту новую текстуру, направления, вращение, и т.д).

```
// Если мы за левым краем, направление влево и в объект не попали
if ((object[loop].x<(object[loop].distance-15.0f)/2.0f) &&
   (object[loop].dir==0) && !object[loop].hit)
{
  miss+=1;              // Увеличиваем miss (Промазали по объекту)
  object[loop].hit=TRUE; // Задать hit в True вручную убив объект
}
```

Следующий код делает тоже самое как и код выше, но проверяет вышел ли объект за правый край экрана. Мы также проверяем перемещается ли объект вправо, а не в другом направлении. Если объект за экраном, мы увеличиваем значение miss и вынуждаем объект самоуничтожится, сообщая нашей программе, что в него попали.

```
// Если мы за правым краем, направление вправо и в объект не попали
if ((object[loop].x>-(object[loop].distance-15.0f)/2.0f) &&
   (object[loop].dir==1) && !object[loop].hit)
{
  miss+=1;              // Увеличиваем miss (Промазали по объекту)
  object[loop].hit=TRUE; // Задать hit в True вручную убив объект
}
```

Код падения довольно прост. Мы проверяем, попал ли объект в землю. Мы не хотим, чтобы объект провалился сквозь землю, которая расположена на отметке -3.0f. Вместо этого, мы проверяем, находится ли объект ниже -2.0f. Затем проверяем, что объект действительно падает (dir=3), и что в объект еще не попали. Если объект ниже -2.0f по оси Y, мы увеличиваем miss и задаем значение hit объекта в ИСТИНА (вынуждая объект самоуничтожиться, поскольку он упал на землю)... неплохой трюк.

```
    // Если мы за нижним краем, направление вниз и в объект не попали
    if ((object[loop].y<-2.0f) && (object[loop].dir==3) && !object[loop].hit)
    {
      miss+=1;              // Увеличиваем miss (Промазали по объекту)
      object[loop].hit=TRUE; // Задать hit в True вручную убив объект
    }
```

В отличие от предыдущего кода, этот код немного отличается. Мы не хотим, чтобы объект улетел за облака! Мы проверяем больше ли переменная y объекта, чем 4.5f (близко к облакам). Мы также проверяем, что объект двигается вверх (dir=2). Если значение переменной y объекта больше, чем 4.5f, вместо разрушения объекта, мы изменяем его направление. Таким образом, объект быстро упадет на землю (вспомните, что вверх быстрее, чем вниз) и как только он долетит до потолка, мы изменяем его направление, поэтому он начинает падать на землю.

Нет необходимости уничтожать объект, или увеличивать переменную miss. Если Вы промазали в объект, поскольку он улетел в небо, есть всегда шанс, чтобы попасть в него, поскольку он падает. Код падения обработает финальное разрушение объекта.

```
    // Если мы за верхним краем и направление вверх
    if ((object[loop].y>4.5f) && (object[loop].dir==2))
      object[loop].dir=3;          // Изменим на направление вверх
  }
}
```

Следующим идет код рисования объекта. Я хотел иметь наиболее быстрый и простой способ вывода игровых объектов, вместе с перекрестьем, и обойтись при этом небольшим кодом насколько это возможно. Функции Object надо указать 3 параметра. Вначале ширину, она задает насколько будет широк объект, когда он будет выведен. Затем высота, она задает насколько будет высоким объект, когда он будет выведен. Наконец, мы имеем texid. Переменная texid выбирает текстуру, которую мы хотим использовать. Если мы хотим вывести ведро, которое задается текстурой 1, мы передаем значение 1 для texid. Довольно просто!

Мы выбираем текстуру, и затем выводим четырехугольник. Мы используем стандартные текстурные координаты, поэтому вся текстура накладывается на лицевую часть четырехугольника. Четырехугольник выведен против часовой стрелки (что требуется для отсечения).

```
// Отрисовка объекта используя требуемые ширину, высоту и текстуру
void Object(float width,float height,GLuint texid)
{
  glBindTexture(GL_TEXTURE_2D, textures[texid].texID); // Выбор правильной текстуры
  glBegin(GL_QUADS); // Начала рисования четырехугольника
    glTexCoord2f(0.0f,0.0f); glVertex3f(-width,-height,0.0f);  // Лево Низ
    glTexCoord2f(1.0f,0.0f); glVertex3f( width,-height,0.0f);  // Право Низ
    glTexCoord2f(1.0f,1.0f); glVertex3f( width, height,0.0f);  // Право Верх
    glTexCoord2f(0.0f,1.0f); glVertex3f(-width, height,0.0f);  // Лево Верх
  glEnd();           // Конец рисования четырехугольника
}
```

Функции отрисовки взрыва Explosion надо указать один параметр. Переменная num – идентификатор объекта. Для создания корректной отрисовки взрыва, мы должны захватить часть текстуры взрыва, таким же способом, каким мы захватываем каждый символ из текстуры шрифта. В двух строках ниже вычисляются столбец (ex) и строка (ey) из одного числа (frame).

В первой строке берется текущий кадр и делится на 4. Деление на 4 должно замедлить мультипликацию. Выражение %4 сохраняет значение в диапазоне 0-3. Если бы значение будет выше, чем 3, то они вернутся и начнутся снова с 0. Если бы значение было 5, то оно станет 1. Значения от 0 до 9 становятся значениями 0,1,2,3,0,1,2,3,0. Мы делим конечный результат 4.0f, потому что координаты текстуры находятся в диапазоне от 0.0f до 1.0f. Наша текстура взрыва имеет 4 изображения взрыва слева направо и 4 сверху вниз.

Надеюсь, что вы не растерялись. Поэтому, если наше число до деления может только быть 0,1,2 или 3, наше число после деления на 4.0f может только быть 0.0f, 0.25f (1/4), 0.50f (2/4) или 0.75f (3/4). Это дает нам нашу слева направо координату текстуры (ex).

Затем мы вычисляем строку (ey). Мы берем текущий кадр объекта и делим его на 4, чтобы немного замедлить мультипликацию. Затем мы делим на 4 снова, чтобы исключить полную строку. Наконец мы делим на 4 последний раз, чтобы получить нашу вертикальную координату текстуры.

Небольшой пример. Если наш текущий кадр равен 16. Тогда ey = ((16/4)/4)/4 или 4/4/4 или 0.25f. Одна строка вниз. Если наш текущий кадр равен 60. Тогда ey = ((60/4)/4)/4 или 15/4/4 или 3/4 или 0.75f. Причина, по которой 15/4 - не равно 3.75 та, что мы работаем с целыми числами вплоть до того, как мы делаем последнее деление. Удерживая это в памяти, мы можем сказать, что значение ey может только быть одним из 4 значений: 0.0f, 0.25f, 0.50f или 0.75f. Принимая это мы остаемся в пределах нашей текстуры (предотвращая превышения текущего кадра выше значения 63).

Надеюсь, это имеет смысл... Это - простая, но наводящая ужас математика.

```
void Explosion(int num)             // Нарисовать анимированный взрыв для объекта "num"
{
  float ex = (float)((object[num].frame/4)%4)/4.0f; // Вычислить X (0.0f - 0.75f)
  float ey = (float)((object[num].frame/4)/4)/4.0f; // Вычислить Y (0.0f - 0.75f)
```

Теперь, когда мы вычислили координаты текстуры, все, что нам осталось нарисовать наш текстурированный четырехугольник. Координаты вершины фиксированы от -1.0f до 1.0f. Вы видите, что мы вычитаем ey из 1.0f. Если бы мы этого не делали, анимация была бы в обратном порядке ... Взрыв должен быть большим, а потом постепенно исчезать. Эффект должен быть правильным!

Вначале мы привязываем текстуру взрыва, до вывода текстурированного четырехугольника. Снова, четырехугольник выводится против часовой стрелки.

```
  glBindTexture(GL_TEXTURE_2D, textures[5].texID);  // Выбор текстуры взрыва
  glBegin(GL_QUADS);      // Нарисовать четырехугольник
    glTexCoord2f(ex      ,1.0f-(ey      )); glVertex3f(-1.0f,-1.0f,0.0f);  // Лево Низ
    glTexCoord2f(ex+0.25f,1.0f-(ey      )); glVertex3f( 1.0f,-1.0f,0.0f);  // Право Низ
    glTexCoord2f(ex+0.25f,1.0f-(ey+0.25f)); glVertex3f( 1.0f, 1.0f,0.0f);  // Право Верх
    glTexCoord2f(ex      ,1.0f-(ey+0.25f)); glVertex3f(-1.0f, 1.0f,0.0f);  // Лево Верх
  glEnd();                // Четырехугольник нарисован
```

Ранее я упомянул о том, что значение кадра не должно быть больше 63, иначе анимация повторится. Поэтому мы увеличиваем значение кадра, и проверяем его большее ли оно, чем 63. Если это так, то мы вызываем InitObject(num), который уничтожает объект и дает ему новые значения, чтобы создать полностью новый объект.

```
  object[num].frame+=1;          // Увеличим текущий кадр взрыва
  if (object[num].frame>63)      // Отрисованы все 16 кадров?
  {
    InitObject(num);             // Инициализируем объект (назначим новые значения)
  }
}
```

В этой часть кода выводятся все цели на экран. Мы начинаем со сброса матрицы вида модели. Затем мы сдвигаемся на 10 единиц в экран и запускаем цикл от 0 до текущего уровня игрока.

```
void DrawTargets(void)                  // Нарисуем цель
{
  glLoadIdentity();                     // Сброс матрицы вида модели
  glTranslatef(0.0f,0.0f,-10.0f);       // Переход на 10 единиц в экран
  for (int loop=0; loop<level; loop++)  // Цикл по объектам
  {
```

Первой строке кода происходит назначение имени (номера) на каждый объект. Первый выведенный объект будет 0. Второй объект будет 1, и т.д ... Если бы цикл продолжался до 29, последнему выведенному объекту дали бы имя 29. После назначения имени объекту, мы помещаем матрицу вида модели в стек. Важно обратить внимание на то, что вызовы glLoadName() игнорируются, если программа не в режиме выбора.

Затем мы перемещаемся на то место на экране, где мы хотим, чтобы наш объект был выведен. Мы используем object[loop].x, чтобы позиционировать объект по оси X, object[loop].y чтобы позиционировать объект по оси Y и object[loop].distance, чтобы позиционировать объект по оси Z (глубина). Мы уже переместились на 10 единиц в экран, поэтому фактическое расстояние, на которое объект будет выведен, равно object[loop].distance-10.0f.

```
    glLoadName(loop);           // Назначение имени объекта (ID)
    glPushMatrix();             // Поместить в стек матрицу
                                // Позиционирование объекта (x,y)
    glTranslatef(object[loop].x,object[loop].y,object[loop].distance);
```

До того как мы выведем объект, мы должны проверить попали ли в него или нет. Мы делаем это, проверяя равно ли значение object[loop].hit ИСТИНА. Если это так, то мы вызываем Explosion(loop), которая анимирует взрыв вместо фактического объекта. Если в объект не попали, мы вращаем объект по оси Z на object[loop].spin градусов до того как мы вызовем Object(…).
В функцию Object() передаются 3 параметра. Первый - ширина, второй - высота, и третий - номер текстуры. Чтобы получить ширину и высоту, мы используем массив size[object[loop].texid].w и size[object[loop].texid].h. Так мы найдем ширину и высоту из нашего заранее определенного в начале этой программы массива размеров объектов. Причина, по которой мы используем object[loop].texid состоит в том, что texid указывает на тип объекта, который мы выводим. Если texid равно 0, то это всегда голубая рожа... texid равно 3, то это всегда кока-кола, и т.д.

После отрисовки объекта, мы возвращаем матрицу из стека, сбрасывая вид, поэтому наш следующий объект будет выведен в нужном положении на экране.

```
    if (object[loop].hit)       // В объект попали
    {
      Explosion(loop);          // Нарисовать взрыв
    }
    else                        // Иначе
    {
                                // Вращать объект
      glRotatef(object[loop].spin,0.0f,0.0f,1.0f);
                                // нарисовать объект
      Object(size[object[loop].texid].w,size[object[loop].texid].h,object[loop].texid);
    }
    glPopMatrix();              // Вытолкнуть матрицу
  }
}
```

Далее то место, где происходит рисование. Мы начинаем с очистки экран, и сбрасывая матрицу вида модели.

```
void Draw(void)                 // Нарисовать нашу сцену
{                               // Очистка экрана и буфера глубины
  glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glLoadIdentity();             // Сброс матрицы просмотра вида
```

Затем мы помещаем матрицу вида модели в стек и выбираем текстуру неба (текстура 7). Небо сделано из 4 текстурированных четырехугольников. Первые 4 вершины рисуют полоску неба, в половину общей высоты неба начиная от земли. Текстура этого четырехугольника будет двигаться довольно медленно. Следующие 4 вершины выводят небо там же, но текстура неба будет двигаться быстрее. Две текстуры смешаются вместе в режиме альфа смешивания, чтобы создать красивый слоистый эффект.

```
  glPushMatrix();               // Запомнить матрицу
  glBindTexture(GL_TEXTURE_2D, textures[7].texID); // Выбор текстуры неба
  glBegin(GL_QUADS);            // Начало рисования четырехугольников
    glTexCoord2f(1.0f,roll/1.5f+1.0f); glVertex3f( 28.0f,+7.0f,-50.0f);  // Право Верх
    glTexCoord2f(0.0f,roll/1.5f+1.0f); glVertex3f(-28.0f,+7.0f,-50.0f);  // Лево Верх
    glTexCoord2f(0.0f,roll/1.5f+0.0f); glVertex3f(-28.0f,-3.0f,-50.0f);  // Лево Низ
    glTexCoord2f(1.0f,roll/1.5f+0.0f); glVertex3f( 28.0f,-3.0f,-50.0f);  // Право Низ
    glTexCoord2f(1.5f,roll+1.0f); glVertex3f( 28.0f,+7.0f,-50.0f);    // Право Верх
    glTexCoord2f(0.5f,roll+1.0f); glVertex3f(-28.0f,+7.0f,-50.0f);    // Лево Верх
    glTexCoord2f(0.5f,roll+0.0f); glVertex3f(-28.0f,-3.0f,-50.0f);    // Лево Низ
    glTexCoord2f(1.5f,roll+0.0f); glVertex3f( 28.0f,-3.0f,-50.0f);    // Право Низ
```

Чтобы создать иллюзию, что небо движется к зрителю, мы выводим еще два четырехугольника, но на сей раз мы рисуем их в верхней полоске неба. Первые 4 вершины выводят медленно движущиеся облака, и оставшиеся 4 выводит более быстро перемещающиеся облака. Эти два уровня смешаются вместе в режиме альфа смешивания, чтобы создать слоистый эффект. Второй уровень облаков сдвинут на 0.5f так, чтобы две текстуры не совпадали друг с другом. Тоже самое с двумя слоями облаков выше. Второй слой смещен на 0.5f.
После вывода всех 4 четырехугольников получится эффект движения неба, при котором будет казаться, что оно движется вначале вверх, а потом на зрителя. Я мог бы использовать текстурированную полусферу для неба, но мне было лень это делать, и полученный эффект все таки хороший.

```
  glTexCoord2f(1.0f,roll/1.5f+1.0f); glVertex3f( 28.0f,+7.0f,0.0f);  // Право Верх
  glTexCoord2f(0.0f,roll/1.5f+1.0f); glVertex3f(-28.0f,+7.0f,0.0f);  // Лево Верх
  glTexCoord2f(0.0f,roll/1.5f+0.0f); glVertex3f(-28.0f,+7.0f,-50.0f);  // Лево Низ
  glTexCoord2f(1.0f,roll/1.5f+0.0f); glVertex3f( 28.0f,+7.0f,-50.0f);  // Право Низ
  glTexCoord2f(1.5f,roll+1.0f); glVertex3f( 28.0f,+7.0f,0.0f);    // Право Верх
  glTexCoord2f(0.5f,roll+1.0f); glVertex3f(-28.0f,+7.0f,0.0f);    // Лево Верх
  glTexCoord2f(0.5f,roll+0.0f); glVertex3f(-28.0f,+7.0f,-50.0f);    // Лево Низ
  glTexCoord2f(1.5f,roll+0.0f); glVertex3f( 28.0f,+7.0f,-50.0f);    // Право Низ
 glEnd();               // Кончили рисовать
```

После отрисовки неба пришло время нарисовать землю. Мы выводим землю с того места, где начали выводить текстуру неба. Текстура земли движется с той же самой скоростью как быстро двигающиеся облака.
Текстура повторяется 7 раз слева направо и 4 раза от горизонта до зрителя для увеличения детализации и предотвращения появления на текстуре больших блочных пикселей (blocky). Это делается при помощи увеличения диапазона координат текстуры от 0.0f - 1.0f до 0.0f - 7.0f (слева направо) и 0.0f - 4.0f (вверх и вниз).

```
 glBindTexture(GL_TEXTURE_2D, textures[6].texID); // Выбор текстуры земли
 glBegin(GL_QUADS);      // Начало рисования четырехугольника
  glTexCoord2f(7.0f,4.0f-roll); glVertex3f( 27.0f,-3.0f,-50.0f);  // Право Верх
  glTexCoord2f(0.0f,4.0f-roll); glVertex3f(-27.0f,-3.0f,-50.0f);  // Лево Верх
  glTexCoord2f(0.0f,0.0f-roll); glVertex3f(-27.0f,-3.0f,0.0f);  // Лево Низ
  glTexCoord2f(7.0f,0.0f-roll); glVertex3f( 27.0f,-3.0f,0.0f);  // Право Низ
 glEnd();               // Кончили рисовать
```

После того как мы нарисовали небо и землю, мы переходим к разделу кода, который выводит все наши цели в DrawTargets().
После вывода целей, мы возвращаем матрицу вида модели (восстанавливая ее к предыдущему состоянию).

```
 DrawTargets();              // Нарисовать наши цели
 glPopMatrix();             // Возврат матрицы
```

Следующий код выводит перекрестье. Вначале мы получаем текущие размеры нашего окна. Мы делаем это каждый раз, так размеры окна могут быть изменены в оконном режиме. Функция GetClientRect возвращает размеры и сохраняет их в window. Затем мы выбираем нашу матрицу проецирования и помещаем ее в стек. Мы сбрасываем вид вызовом glLoadIdentity() и затем переключаем экран режим ортографической проекции вместо перспективной. Окно начинается с 0 до window.right слева направо, и от 0 до window.bottom с нижнего края экрана до верхнего.
Третий параметр glOrtho() задает значение нижнего края, но я поменял значения нижнего и верхнего края местами. Я сделал это, потому что перекрестье выводится в направлении против часовой стрелки. Если 0 будет сверху, а window.bottom в низу, обход вершин при отображении будет в противоположном направлении, и перекрестье с текстом не будут отображаться.
После настройки ортографического вида, мы выбираем матрицу вида модели, и устанавливаем перекрестье. Поскольку экран перевернут снизу верх, мы должны также инвертировать мышь. Иначе наше перекрестье двигалось бы вниз, если бы мы переместили бы мышь верх, и наше перекрестье двигалось бы верх, если бы мы переместили бы мышь вниз. Чтобы сделать это, мы вычитаем текущее значение mouse_y из значения нижнего края окна (window.bottom).

После переноса в текущую позицию мыши, мы выводим перекрестье. Мы делаем это, вызывая Object(). Вместо единиц, мы задаем ширину и высоту в пикселях. Перекрестье будет размером 16x16 пикселей, и используется текстура от восьмого объекта (текстура перекрестья).
Я решил использовать не стандартный курсор по двум причинам. Первая причина и наиболее важная состоит в том, что это выглядит привлекательно, и курсор можно изменить, используя любую программу рисования, которая поддерживает альфа канал. Во-вторых, некоторые видео платы не отображают курсор в полноэкранном режиме. Играть без курсора в полноэкранном режиме не просто :).

```
  // Перекрестье (Ортографический просмотр)
  RECT window;                  // Размеры окна
  GetClientRect (g_window->hWnd,&window); // Получить их
  glMatrixMode(GL_PROJECTION);          // Выбор матрицы проекции
  glPushMatrix();               // Сохранить матрицу
  glLoadIdentity();             // Сброс матрицы
  glOrtho(0,window.right,0,window.bottom,-1,1);     // Настройка ортографического экрана
  glMatrixMode(GL_MODELVIEW);    // Выбор матрицы вида модели
  glTranslated(mouse_x,window.bottom-mouse_y,0.0f); // Перенос в текущую позицию мыши
  Object(16,16,8);              // Нарисовать перекрестье
```

В этом разделе кода выводится заголовок сверху экрана, и отображает уровень и счет внизу слева и справа в углах экрана. Причина, по которой я поместил этот код здесь состоит в том, что проще точно позиционировать текст ортографическом режиме.

```
  // Счет и Заголовок игры
  glPrint(240,450,"NeHe Productions"); // Заголовок
  glPrint(10,10,"Level: %i",level);    // Уровень
  glPrint(250,10,"Score: %i",score);   // Счет
```

В этом разделе проверяется, пропустил ли игрок больше чем 10 объектов. Если так, то мы устанавливаем число промахов (miss) в 9, и мы задаем game в ИСТИНА. Если game равно ИСТИНА, то игра закончена!

```
 if (miss>9)              // Пропустил 10 объектов?
 {    miss=9;               // Предел равен 10
   game=TRUE;              // Конец игры
 }
```

В коде ниже, мы проверяем, является ли game ИСТИННА. Если game ИСТИННА, мы выводим сообщение - ИГРА ОКОНЧЕНА. Если game ЛОЖЬ, мы выводим боевой дух игрока (до 10). Боевой дух вычисляется, вычитая число промахов игрока (miss) из 10. Чем больше промахов, тем более низкий у него боевой дух.

```
  if (game)                // Игра окончена?
    glPrint(490,10,"GAME OVER"); // Сообщение о том, что игра окончена
  else
    glPrint(490,10,"Morale: %i/10",10-miss); // Вывод морали #/10
```

Последнее, что мы делаем выбор матрицы проецирования, восстановление нашей матрицы в предыдущее состояние, выбор матрицы вида модели и сброс буфера, чтобы быть уверенным, что все объекты выведены.

```
  glMatrixMode(GL_PROJECTION);  // Выбор матрицы проецирования
  glPopMatrix();                // Восстановление старой матрицы
  glMatrixMode(GL_MODELVIEW);   // Выбор матрицы вида модели
  glFlush();                    // Сброс конвейера GL
}
```

Этот урок появился после многих ночей, и многих многих часов кодирования и редактирования HTML. По окончанию этого урока Вы должны хорошо понимать, как работает выбор, сортировка, альфа смешивание и альфа тест. Выбор позволяет Вам создавать процедуры для позиционирования и выбора объектов. Любая игра – это иллюзия графического интерфейса. Лучшая возможность выбора заключается в том, что Вы не должны следить за тем, где ваши объекты. Вы назначаете имя и проверяете попадания. Это просто! Используя альфа смешивание и альфа тест Вы может делать ваши объекты полностью непрозрачными, или полностью прозрачными только в отдельных местах. Результаты великолепны, и Вы не должны волноваться о просвечивании через ваши объекты фона за ним, если Вы не хотите этого! Как всегда, я буду надеяться, что Вы получите наслаждение от этого урока, и надеюсь увидеть новые крутые игры, или проекты, основанные на коде этого урока. Если у Вас есть вопросы, или Вы нашли ошибки в уроке, пожалуйста, сообщите мне об этом ... Я всего лишь человек :).
Я мог потратить намного больше времени, добавив, например физику, или улучшив графику, или звук, и т.д. Но это только урок! И я не писал этот урок так, чтобы поразить вас. Я написал его, чтобы обучить Вас OpenGL с наименьшими отклонениями от темы насколько это возможно. Я надеюсь увидеть модификации кода. Если Вы добавите что-то в урок, то вышлите мне демонстрационную версию своей программы. Если это великолепная модификация, я помещу ее на страницах "Download". Если я получу достаточно модификаций, я могу изменить этот урок! Я должен здесь дать Вам точку опоры. Все остальное за Вами :) .

Урок 34. Построение красивых ландшафтов с помощью карты высот.

Beautiful Landscapes By Means Of Height Mapping

Добро пожаловать в очередной потрясающий урок. Код этого урока был написан Беном Хамфри (Ben Humphrey) и он основан на коде первого урока. К этому моменту вы должны быть уже гуру в OpenGL (усмешка) и перенос кода из этого урока в ваш базовый код должно быть проще простого!

Этот урок научит вас, как сделать круто выглядящий ландшафт из карты высот. Для тех из вас, кто не представляет что такое карта высот, я попытаюсь объяснить. Карта высот это просто… смещение от поверхности. Для тех, кто до сих пор ломает голову вопросом «о чем, черт побери, этот парень толкует!?!»… отмечу, что по-английски, карта высот представляет низкие и высокие точки для нашего ландшафта. Исключительно от вас зависит, какие значения элементов карты высот будут представлять низкие точки, а какие высокие. Важно заметить, что карты высот не обязательно могут быть картинками… Вы можете создать карту высот из любого типа данных. Например, вы можете использовать аудио-поток для визуального представления карты высот. Если еще ничего не прояснилось… продолжайте читать… все будет проясняться по мере изучения урока :)

```
#include <windows.h>  // Заголовочный файл для Windows
#include <stdio.h>    // Заголовочный файл для стандартного ввода-вывода (НОВОЕ)
#include <gl\gl.h>    // Заголовочный файл для OpenGL32 библиотеки
#include <gl\glu.h>   // Заголовочный файл для GLu32 библиотеки
#include <gl\glaux.h> // Заголовочный файл для GLaux библиотеки

#pragma comment(lib, "opengl32.lib")     // Ссылка на OpenGL32.lib
#pragma comment(lib, "glu32.lib")        // Ссылка на Glu32.lib
```

Мы начнем с объявления нескольких важных переменных. MAP_SIZE – это размер нашей карты. В этом уроке размер будет 1024x1024. STEP_SIZE – это размер каждого квадрата, используемого для построения ландшафта. Уменьшая значение этой переменной, мы увеличиваем гладкость ландшафта. Важно заметить, что чем меньше STEP_SIZE, тем больше времени потребуется для выполнения программы, особенно когда используются большие карты высот. HEIGHT_RATIO используется для масштабирования ландшафта по оси y. Если это значение невелико, горы будут более пологими, иначе – более отвесными.

В дальнейшем в исходном коде вы заметите переменную bRender. Если bRender истинно (по умолчанию), то рисуем заполненные полигоны, иначе – проволочные.

```
#define    MAP_SIZE  1024     // Размер карты вершин (НОВОЕ)
#define    STEP_SIZE  16      // Ширина и высота каждого квадрата (НОВОЕ)
// Коэффициент масштабирования по оси Y в соответствии с осями X и Z (НОВОЕ)
#define    HEIGHT_RATIO  1.5f


HDC    hDC=NULL;        // Приватный контекст устройства GDI
HGLRC    hRC=NULL;      // Постоянный контекст рендеринга
HWND    hWnd=NULL;      // Указатель на наше окно
HINSTANCE  hInstance;   // Указывает на дескриптор текущего приложения

bool    keys[256];       // Массив для процедуры обработки клавиатуры
bool    active=TRUE;     // Флаг активности окна, по умолчанию=TRUE
bool    fullscreen=TRUE;  // Флаг полноэкранного режима, по умолчанию=TRUE
bool    bRender = TRUE;   // Флаг режима отображения полигонов,
                  // по умолчанию=TRUE (НОВОЕ)
```

Здесь мы создаем массив (g_HeightMap[ ]) байтов для хранения нашей карты вершин. Мы будем считывать массив из .RAW файла, который содержит значения от 0 до 255. 255 будет значением, соответствующим самой высокой точке, а 0 – самой низкой. Мы также создаем переменную scaleValue для масштабирования сцены. Это дает возможность пользователю увеличивать и уменьшать сцену.

```
BYTE g_HeightMap[MAP_SIZE*MAP_SIZE];    // Содержит карту вершин (НОВОЕ)
float scaleValue = 0.15f;               // Величина масштабирования поверхности (НОВОЕ)

LRESULT  CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM); // Объявление для WndProc
```

Код в процедуре ReSizeGLScene() остался таким же, как и в первом уроке, за исключением дальней плоскости отсечения. Она изменилась со 100.0f до 500.0f

```
GLvoid ReSizeGLScene(GLsizei width, GLsizei height) // Масштабирование и инициализация окна OpenGL
{
... вырезано ...
}
```

Следующие строчки кода загружают данные из .RAW файла. Все достаточно просто! Мы открываем файл в режиме бинарного чтения.(Read/Binary) Затем делаем проверку на существование и открытие файла. Если не удалось открыть файл, то возникнет сообщение об ошибке.

```
// Чтение и сохранение .RAW файла в pHeightMap
void LoadRawFile(LPSTR strName, int nSize, BYTE *pHeightMap)
{
 FILE *pFile = NULL;
 // открытие файла в режиме бинарного чтения
 pFile = fopen( strName, "rb" );
 // Файл найден?
 if ( pFile == NULL )
 {
  // Выводим сообщение об ошибке и выходим из процедуры
  MessageBox(NULL, "Can't Find The Height Map!", "Error", MB_OK);
  return;
 }
```

Если мы дошли до этого места, значит, никаких проблем с открытием файла не возникло. Теперь можно считывать данные. Делаем это с помощью функции fread(). pHeightMap это место для хранения данных (указатель на массив g_Heightmap). Цифра 1 говорит о том, что мы будем считывать по байту за раз, nSize – это сколько байт нужно считать (размер карты в байтах – ширина карты * высоту карты). Наконец, pFile – это указатель на структуру файла. После чтения данных мы проверяем, возникли ли какие-либо ошибки. Сохраняем результат в result и потом проверяем его. Если произошла ошибка – выводим предупреждение.
И последнее что мы сделаем, это закроем файл с помощью fclose(pFile).

```
 // Загружаем .RAW файл в массив pHeightMap
 // Каждый раз читаем по одному байту, размер = ширина * высота
 fread( pHeightMap, 1, nSize, pFile );
 // Проверяем на наличие ошибки
 int result = ferror( pFile );
 // Если произошла ошибка
 if (result)
 {
  MessageBox(NULL, "Failed To Get Data!", "Error", MB_OK);
 }
 // Закрываем файл
 fclose(pFile);
}
```

Код инициализации довольно простой. Мы устанавливаем цвет, которым будет очищен экран, в черный, создаем буфер глубины, включаем сглаживание полигонов и т.д. После всего этого загружаем наш .RAW файл. Для этого передаем в качестве параметров имя файла ("Data/Terrain.raw"), размер .RAW файла (MAP_SIZE * MAP_SIZE) и, наконец, массив HeightMap (g_HeightMap) в функцию LoadRawFile(). Файл будет загружен, и наши данные сохранятся в массиве g_HeightMap.

```
int InitGL(GLvoid)          // Инициализация OpenGL
{
 glShadeModel(GL_SMOOTH);   // Включить сглаживание
 glClearColor(0.0f, 0.0f, 0.0f, 0.5f);  // Очистка экрана черным цветом
 glClearDepth(1.0f);        // Установка буфера глубины
 glEnable(GL_DEPTH_TEST);   // Включить буфер глубины
 glDepthFunc(GL_LEQUAL);    // Тип теста глубины
 glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);  // Улучшенные вычисления перспективы
```

```
  // Читаем данные из файла и сохраняем их в массиве g_HeightMap array.
  // Также передаем размер файла (1024).

  LoadRawFile("Data/Terrain.raw", MAP_SIZE * MAP_SIZE, g_HeightMap);  // ( НОВОЕ )
  return TRUE;                // Инициализация прошла успешно
}
```

Когда имеем дело с массивами, мы должны быть уверены, что не выходим за их пределы. Чтобы быть уверенным, что это не произойдет, будем использовать оператор %, который, в нашем случае, будет запрещать превышение переменными x и y величины MAX_SIZE – 1.
Убеждаемся, что pHeightMap указывает на верные данные, иначе возвращаем 0.
Если все прошло успешно, мы возвратим величину, хранящуюся в переменных x и y в нашей карте вершин. К этому моменту вы должны знать, что мы умножаем y на ширину карты MAP_SIZE, чтобы перемещаться по данным, хранящимся в карте вершин.

```
int Height(BYTE *pHeightMap, int X, int Y)      // Возвращает высоту из карты вершин (?)
{
  int x = X % MAP_SIZE;          // Проверка переменной x
  int y = Y % MAP_SIZE;          // Проверка переменной y
  if(!pHeightMap) return 0;      // Убедимся, что данные корректны
```

Так как имеем двумерный массив, то можем использовать уравнение: номер = (x + (y * ширинаМассива)), т.е. получаем pHeightMap[x][y], в противном случае: (y + (x * ширинаМассива)).

Теперь, когда имеем верный номер, возвратим значение высоты этого номера (x, y в нашем массиве)

```
  return pHeightMap[x + (y * MAP_SIZE)];      // Возвращаем значение высоты
}
```

Здесь будем выбирать цвет вершины, основываясь на значении номера высоты. Чтобы сделать потемнее, начнем с -0.15f. Будем получать коэффициент цвета от 0.0f до 1.0f путем деление на 256.0f. Если нет никаких входных данных, функция не возвратит никакого значения. Если все прошло успешно, будем устанавливать оттенок синего цвета, используя glColor3f(0.0f, fColor, 0.0f). Попробуйте поизменять fColor в красный и зеленый, чтобы изменить цвет ландшафта.

```
// Эта функция устанавливает значение цвета для конкретного номера, зависящего от номера высоты
void SetVertexColor(BYTE *pHeightMap, int x, int y)
{
  if(!pHeightMap) return;          // Данные корректны?
  float fColor = -0.15f + (Height(pHeightMap, x, y ) / 256.0f);

  // Присвоить оттенок синего цвета для конкретной точки
  glColor3f(0.0f, 0.0f, fColor );
}
```

Далее мы вырисовываем наш ландшафт. Переменные X и Y будут использоваться для перемещения по массиву карты высоты. Переменные x, y и z будут использоваться для визуализации квадратов, составляющих ландшафт
Как обычно, проверяем, содержит ли pHeightMap нужные нам данные. Если нет – ничего не делаем.

```
void RenderHeightMap(BYTE pHeightMap[]) // Визуализация карты высоты с помощью квадратов
{
  int X = 0, Y = 0;    // Создаем пару переменных для перемещения по массиву
  int x, y, z;         // И еще три для удобства чтения
  if(!pHeightMap) return;     // Данные корректны?
```

Здесь мы сможем изменять режим отображения (проволочный или сплошной). Если bRender = True, то рендерим полигоны, иначе – линии.

```
  if(bRender)            // Что хотим визуализировать?
    glBegin( GL_QUADS ); // Полигоны
  else
    glBegin( GL_LINES ); // Линии
```

Далее рисуем поверхность из карты вершин. Для этого пройдемся по массиву высот и, доставая значения вершин, будем рисовать наши точки. Если бы мы могли видеть, как это происходит, то вначале нарисовались бы столбцы (Y), а затем строки. Заметьте, что мы используем STEP_SIZE. Чем больше STEP_SIZE, тем менее гладко выглядит поверхность, и наоборот. Если принять STEP_SIZE = 1, то вершина будет создаваться для каждого пикселя из карты высот. Я выбрал STEP_SIZE = 16, как достаточно скромный размер. Намного меньшее значение было бы безрассудством, да и потребовалось бы гораздо больше процессорного времени. Естественно вы можете увеличить значение, когда включите источник света. Освещение спрячет шероховатость формы. Вместо освещения мы ассоциируем цвет с каждой точкой карты вершин, дабы облегчить урок. Чем выше полигон – тем ярче цвет.

```
for ( X = 0; X < MAP_SIZE; X += STEP_SIZE )
   for ( Y = 0; Y < MAP_SIZE; Y += STEP_SIZE )
   {
    // Получаем (X, Y, Z) координаты нижней левой вершины
    x = X;
    y = Height(pHeightMap, X, Y );
    z = Y;

    // Устанавливаем цвет конкретной точки
    SetVertexColor(pHeightMap, x, z);
    glVertex3i(x, y, z);      // Визуализация ее

    // Получаем (X, Y, Z) координаты верхней левой вершины
    x = X;
    y = Height(pHeightMap, X, Y + STEP_SIZE );
    z = Y + STEP_SIZE ;

    // Устанавливаем цвет конкретной точки
    SetVertexColor(pHeightMap, x, z);
    glVertex3i(x, y, z);      // Визуализация ее

    // Получаем (X, Y, Z) координаты верхней правой вершины
    x = X + STEP_SIZE;
    y = Height(pHeightMap, X + STEP_SIZE, Y + STEP_SIZE );
    z = Y + STEP_SIZE ;

    // Устанавливаем цвет конкретной точки
    SetVertexColor(pHeightMap, x, z);
    glVertex3i(x, y, z);      // Визуализация ее

    // Получаем (X, Y, Z) координаты нижней правой вершины
    x = X + STEP_SIZE;
    y = Height(pHeightMap, X + STEP_SIZE, Y );
    z = Y;

    // Устанавливаем цвет конкретной точки
    SetVertexColor(pHeightMap, x, z);
    glVertex3i(x, y, z);      // Визуализация ее
   }
  glEnd();
```

Как только все закончили, восстанавливаем цвет до ярко-белого с альфа-значением 1.0f. Если на экране будут присутствовать другие объекты, мы не хотим видеть их СИНИМИ :)

```
  glColor4f(1.0f, 1.0f, 1.0f, 1.0f);      // Сбрасываем цвет
}
```

Для тех, кто еще не использовал gluLookAt() поясню, что эта функция позиционирует камеру, вид и головной вектор. Мы отодвигаем камеру, чтобы получить хороший вид на ландшафт. Чтобы избежать больших значений, мы будем делить вершины ландшафта на константу масштабирования, как мы делаем это ниже.

Входные параметры функции gluLookAt() следующие: первые три значения указывают на положение камеры, т.е. 212, 60, 94 – это смещение по осям x, y и z соответственно от начала координат. Следующие 3 значения указывают точку визирования (направление камеры). В этом уроке во время просмотра демонстрационного примера вы заметите, что камера направлена немного влево. Также мы направляем камеру навстречу ландшафту. 186 меньше 212, что позволяет нам смотреть влево, 55 ниже, чем 60, что позволяет нам находиться выше ландшафта и смотреть на него с легким наклоном. 171 указывает как далеко от объектов находится камера. Последние три значения указывают OpenGL направление головного вектора. Наши горы растут вверх по оси y, поэтому мы ставим значение y в 1, а остальные в 0.

На первый взгляд gluLookAt кажется очень запутанной. После грубого объяснения этой функции вы возможно запутались. Мой лучший совет – поиграть со значениями. Изменить позицию камеры. Если вы измените y-позицию камеры, скажем в 120, вы увидите больше вершин ландшафта.

Я не уверен, поможет ли это, но я попытаюсь объяснить на пальцах :). Допустим, ваш рост равен шести футам и немного выше. Также предположим, что ваши глаза находятся на высоте шести футов (глаза представляют камеру - 6 футов это 6 делений по оси y). Если вы встанете напротив двух футовой стены (2 деления по оси y), вы будете смотреть ВНИЗ на стену и будете способны видеть ее верх. Если бы стена была высотой 8 футов, вы бы смотрели ВВЕРХ на стену и НЕ видели бы ее верх. Направление взгляда изменяется, когда вы смотрите вверх или вниз (если вы находитесь выше или ниже объекта, на который смотрите). Надеюсь, я немного прояснил ситуацию.

```
int DrawGLScene(GLvoid)      // Здесь содержится код рисования
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);  // Очистка экрана и буфера глубины
  glLoadIdentity();          // Сброс просмотра

  //      Положение   Вид    Вектор вертикали
  gluLookAt(212, 60, 194,  186, 55, 171,  0, 1, 0);  // Определяет вид и положение камеры
```

Следующая строчка кода позволит масштабировать поверхность. Мы можем изменять scaleValue нажатием клавиш ВВЕРХ и ВНИЗ. Заметьте, что мы умножаем Y scaleValue на HEIGHT_RATIO. Это сделано для того, чтобы можно было бы изменять высоту ландшафта.

```
  glScalef(scaleValue, scaleValue * HEIGHT_RATIO, scaleValue);
```

Если мы передадим g_HeightMap в качестве входного параметра в функцию RenderHeightMap(), эта функция визуализирует поверхность в квадратах. Если вы планируете использовать эту функцию в дальнейшем, было бы неплохо добавить в нее переменные (X, Y), чтобы позиционировать ландшафт именно там, где это нужно.

```
  RenderHeightMap(g_HeightMap);        // Визализация карты высот
  return TRUE;            // Идем дальше
}
```

Код функции The KillGLWindow() такой же, как в первом уроке.

```
GLvoid KillGLWindow(GLvoid)          // Уничтожение окна
{
…
}
```

Код функции CreateGLWindow() также не изменился с первого урока.

```
BOOL CreateGLWindow(char* title, int width, int height, int bits, bool fullscreenflag)
{
…
}
```

Единственное изменение в WndProc() – это добавление обработчика события M_LBUTTONDOWN. Он проверяет, нажата ли левая кнопка мыши. Если да, то меняется режим отображения поверхности с проволочной на полигональную и наоборот.

```
LRESULT CALLBACK WndProc(
      HWND  hWnd,      // Указатель на окно
      UINT  uMsg,      // Сообщение для этого окна
      WPARAM  wParam,  // Параметры сообщения
      LPARAM  lParam)  // Параметры сообщения
{
 switch (uMsg)          // Проверим сообщения окна
 {
  case WM_ACTIVATE:    // Наблюдаем за сообщением об активизации окна
  {
   if (!HIWORD(wParam)) // Проверим состояние минимизации
   {
    active=TRUE;     // Программа активна
   }
   else
   {
    active=FALSE;    // Программа больше не активна
   }
   return 0;          // Вернуться к циклу сообщений
  }
  case WM_SYSCOMMAND:  // Перехватаем системную команду
  {
   switch (wParam)    // Проверка выбора системы
   {
    case SC_SCREENSAVE:   // Пытается включиться скринсэйвер?
    case SC_MONITORPOWER: // Монитор пытается переключиться в режим сохранения энергии?
    return 0;        // Не давать этому случиться
   }
   break;             // Выход
  }
  case WM_CLOSE:       // Мы получили сообщение о закрытии программы?
  {
   PostQuitMessage(0);// Послать сообщение о выходе
   return 0;          // Возврат обратно
  }

  case WM_LBUTTONDOWN: // Нажата левая клавиша мыши?
  {
   bRender = !bRender;// Изменить тип визуализации
   return 0;          // Вернуться
  }

  case WM_KEYDOWN:     // Клавиша была нажата?
  {
   keys[wParam] = TRUE; // Если так – пометим это TRUE
   return 0;          // Вернуться
  }
    case WM_KEYUP:       // Клавиша была отпущена?
  {
   keys[wParam] = FALSE; // Если так – пометим это FALSE
   return 0;          // Вернуться
  }

  case WM_SIZE:        // Изменились окна OpenGL
  {
   ReSizeGLScene(LOWORD(lParam),HIWORD(lParam));  // LoWord=ширина, HiWord=высота
   return 0;          // Вернуться
  }
 }
 // Пересылаем все прочие сообщения в DefWindowProc
 return DefWindowProc(hWnd,uMsg,wParam,lParam);
}
```

Никаких принципиальных изменений в нижеследующем коде. Единственное, что претерпело изменение это заголовок окна. До проверки нажатий клавиш все осталось без изменений.

```
int WINAPI WinMain(
     HINSTANCE  hInstance,    // Экземпляр
     HINSTANCE  hPrevInstance,// Предыдущий экземпляр
     LPSTR    lpCmdLine,      // Параметры командной строки
     int    nCmdShow)         // Показать состояние окна
{
 MSG    msg;          // Структура сообщения окна
 BOOL  done=FALSE;    // Булевская переменная выхода из цикла

 // Запросим пользователя, какой режим отображения он предпочитает
 if (MessageBox(NULL,"Would You Like To Run In Fullscreen Mode?",
   "Start FullScreen?",MB_YESNO|MB_ICONQUESTION)==IDNO)
 {
  fullscreen=FALSE;          // Оконный режим
 }
  // Создадим наше окно OpenGL
 if (!CreateGLWindow("NeHe & Ben Humphrey's Height Map Tutorial",
     640, 480, 16, fullscreen))
 {
  return 0;                  // Выходим если окно не было создано
 }
 while(!done)                // Цикл, который продолжается пока done=FALSE
 {
  if (PeekMessage(&msg,NULL,0,0,PM_REMOVE))  // Есть ожидаемое сообщение?
  {
   if (msg.message==WM_QUIT)// Мы получили сообщение о выходе?
   {
    done=TRUE;              // Если так done=TRUE
   }
   else                     // Если нет, продолжаем работать с сообщениями окна
   {
    TranslateMessage(&msg);// Переводим сообщение
    DispatchMessage(&msg); // Отсылаем сообщение
   }
  }
  else                      // Если сообщений нет
  {
   // Рисуем сцену. Ожидаем нажатия кнопки ESC и сообщения о выходе от DrawGLScene()
   // Активно?  Было получено сообщение о выходе?
   if ((active && !DrawGLScene()) || keys[VK_ESCAPE])
   {
    done=TRUE;              // ESC или DrawGLScene просигналили "выход"
   }
   else if (active)         // Не время выходить, обновляем экран
   {
    SwapBuffers(hDC);       // Переключаем буферы (Двойная буферизация)
   }
    if (keys[VK_F1])        // Была нажата кнопка F1?
   {
    keys[VK_F1]=FALSE;      // Если так - установим значение FALSE
    KillGLWindow();         // Закроем текущее окно OpenGL
    fullscreen=!fullscreen;// Переключим режим "Полный экран"/"Оконный"
    // Заново создадим наше окно OpenGL
    if (!CreateGLWindow("NeHe & Ben Humphrey's Height Map Tutorial",
        640, 480, 16, fullscreen))
    {
     return 0;              // Выйти, если окно не было создано
    }
   }
```

Следующий код позволяет вам увеличивать и уменьшать scaleValue. Нажав клавишу «вверх» scaleValue, и ландшафт вместе с ней, увеличатся. Нажав клавишу «вниз» - уменьшится.

```
    if (keys[VK_UP])           // Нажата клавиша ВВЕРХ?
      scaleValue += 0.001f;  // Увеличить переменную масштабирования
     if (keys[VK_DOWN])       // Нажата клавиша ВНИЗ?
      scaleValue -= 0.001f;  // Уменьшить переменную масштабирования
    }
  }

  // Shutdown
  KillGLWindow();              // Закроем окно
  return (msg.wParam);         // Выйдем из программы
}
```

Это все, что нужно для создания красивых ландшафтов. Я надеюсь вы оценили работу Бена! Как обычно, если вы найдете ошибки в уроке, пожалуйста, напишите мне, и я попытаюсь исправить проблему / пересмотреть урок.

Как только вы поймете как программа работает, попробуйте поэкспериментировать с ней. Например вы можете попробовать добавить маленький мячик, катающийся по поверхности. Вы уже знаете высоту каждого кусочка поверхности, поэтому добавление мячика не должно составить проблем. Также можете попробовать следующее: создайте карту высот самостоятельно, сделайте скроллинг ландшафта, добавьте цвета для представления снежных вершин / воды / и т.д., добавьте текстуры, используйте эффект плазмы для постоянно изменяющегося ландшафта. Возможности безграничны :)

Надеюсь вы остались довольны уроком. Посетите сайт Бена: http://www.GameTutorials.com.

Урок 35. Проигрывание AVI файлов в OpenGL.

Playing AVI Files In OpenGL

Я хотел бы начать с того, что я очень горжусь своей обучающей программой. Когда Джонатан де Блок подкинул мне идею сделать AVI проигрыватель в OpenGL, я не имел понятия о том, как открыть видео-файл, не говоря уже о проигрывателе их. Я начал с того, что заглянул в мою коллекцию книг по программированию. Но не в одной из них не было информации об AVI. Тогда я начал читать все, что можно было прочитать об AVI в MSDN. В MSDN много полезной информации, но мне нужно было больше.

После нескольких часов поиска примеров проигрывания AVI, я нашел два сайта. Я не хочу сказать, что я великолепный следопыт, но я обычно всегда нахожу, что ищу. Я был неприятно поражен, когда я увидел, как немного примеров было в Web. Большинство файлов, которые я нашел, не компилировались. Часть примеров была сложна (по крайней мере, для меня), и они делали то, что надо, но исходники были написаны на VB, Delphi, и т.д. (но не VC++).

Вначале отметим статью, написанную Джонатаном Никсом и озаглавленную "Работа с AVI файлами" ("Working with AVI Files"). Вы можете найти ее по адресу: http://www.gamedev.net/reference/programming/features/avifile/

Я очень уважаю Джонатана за написание такой блестящей статьи про формат AVI. Хотя я решил написать код по-другому, отрывки из его кода и комментарии сделали процесс обучения намного проще! Вторым сайтом был "Краткий обзор по AVI" ("The AVI Overview") Джона Ф. МакГоуана. Я мог бы рассказывать и рассказывать об этой удивительной странице Джона, но проще, если вы сами посмотрите на нее. Вот адрес:

http://www.jmcgowan.com/avi.html

Его сайт в значительной степени покрывает все, что нужно знать об AVI формате. Спасибо Джону за создание такой замечательной и доступной всем страницы.

Последнее, что я хотел упомянуть то, что НЕ ОДИН отрывок кода не был скопирован или заимствован. Весь код к уроку был написан за три дня, используя информацию с указанных сайтов и статей. Я хотел бы обратить ваше внимание на то, что этот код не является ЛУЧШИМ способом проигрывания видео-файлов. Это может быть даже

неправильный способ проигрывания AVI файлов. Если вам не нравится мой код, или мой стиль программирования, или вы чувствуете, что я порчу программистское сообщество, выпуская этот урок, у вас есть несколько вариантов:

1) Поискать в сети альтернативные статьи.
2) Написать ваш собственный AVI проигрыватель.
3) Или написать ваш собственный урок!

Каждый, кто посетит мой сайт должен знать, что я средний программист со средними способностями (я упоминал об этом на многочисленных страницах этого сайта)! Я кодирую ради ЗАБАВЫ! Цель этого сайта в том, чтобы сделать жизнь проще для начинающих программистов, которые начинают изучение OpenGL. Эти уроки просто примеры того, как сделать тот или иной эффект. Не больше и не меньше!

Приступим…

Первое что вы заметите это то, что мы подключили библиотеку Видео для Windows. Большое спасибо Microsoft (я не могу поверить, что только сказал это!). Эта библиотека открывает и проигрывает AVI файлы. Всё, что вам надо знать это то, что вы ДОЛЖНЫ подключить файл vfw.h и прилинковать vfw32.lib, если вы хотите, чтобы этот код скомпилировался.

```
#include <windows.h>        // Заголовочный файл Windows
#include <gl\gl.h>          // Заголовочный файл библиотеки OpenGL32
#include <gl\glu.h>         // Заголовочный файл библиотеки Glu32
#include <vfw.h>            // Заголовочный файл для «Видео для Windows»
#include "NeHeGL.h"         // Заголовочный файл NeHeGL.h

#pragma comment( lib, "opengl32.lib" ) // Искать OpenGL32.lib при линковке
#pragma comment( lib, "glu32.lib" )    // Искать GLu32.lib при линковке
#pragma comment( lib, "vfw32.lib" )    // Искать VFW32.lib при линковке

#ifndef CDS_FULLSCREEN        // CDS_FULLSCREEN не определяется некоторыми
#define CDS_FULLSCREEN 4      // компиляторами. Определяем эту константу
#endif                        // Таким образом мы можем избежать ошибок

GL_Window*  g_window;
Keys*       g_keys;
```

Теперь мы объявим переменные. Переменная angle используется, для того чтобы вращать объекты, основываясь на прошедшем времени. Мы будем использовать эту переменную везде, где есть вращение для простоты.

Next – целая переменная которая будет использована для того чтобы узнать сколько времени прошло (в миллисекундах). Она будет использована для сохранения начальной частоты кадров. Мы поговорим об этом позже.

Переменная frame…Конечно это текущий кадр анимации, который мы хотим отобразить. Мы начинаем с 0 (первый кадр). Я думаю безопасно (для программы) предположить что видео, которое мы открыли ДОЛЖНО ИМЕТЬ хотя бы один кадр :).

Переменная effect текущий эффект видимый на экране (объекты: Куб, Сфера, Цилиндр, Ничто). Env - булевская переменная. Если она равна Истине, то тогда наложение окружения включено, если Ложь, то окружение не будет отображено. Если bg Истина, то вы будете видеть полноэкранное видео за объектом. Если Ложь, вы будете видеть только объект (никакого фона).
sp, ep and bp используются чтобы быть уверенным в том, что пользователь не удерживает нажатой соответствующую клавишу.

```
// Пользовательские переменные
float  angle;              // Для вращения
int    next;               // Для анимации
int    frame=0;            // Счётчик кадров
int    effect;             // Текущий эффект
bool   sp;                 // Пробел нажат?
bool   env=TRUE;           // Показ среды
(По умолчанию включен)
bool   ep;                 // 'E' нажато?
bool   bg=TRUE;            // Фон(по умолчанию включен)
bool   bp;                 // 'B' нажато?
```

В структуре psi будет сохранена информация о нашем AVI файле далее в коде. pavi – указатель на буфер, который получает новый дескриптор потока, как только AVI файл будет открыт. pgf - это указатель на объект GetFrame. bmih будет использована потом в коде для конвертирования кадра анимации в формат, который мы захотим (содержит заголовок растра, описывающий, что мы хотим). Lastframe будет содержать номер последнего файла AVI анимации. width и height будут содержать размеры видео потока и наконец… pdata будет указателем на содержимое изображения возвращенного после получения кадра анимации из AVI! Mpf будет использован для подсчёта, сколько миллисекунд каждый кадр отображается на экране. Мы поговорим об этом чуть позже.

```
AVISTREAMINFO    psi;      // Указатель на структуру содержащую информацию о потоке
PAVISTREAM       pavi;     // Дескриптор для открытия потока
PGETFRAME        pgf;      // Указатель на объект GetFrame
BITMAPINFOHEADER bmih;     // Заголовочная информация для DrawDibDraw декодирования
long             lastframe; // Последний кадр анимации
int              width;    // Ширина видео
int              height;   // Высота видео
char             *pdata;   // Указатель на данные текстуры
int              mpf;      // Сколько миллисекунд отображен кадр
```

В этом уроке мы создадим 2 разных квадратичных объекта (сферу и цилиндр) используя библиотеку GLU. Переменная quadratic - это указатель на наши квадратичные объекты.

hdd - это дескриптор контекста устройства DrawDib . hdc - дескриптор контекста устройства.

hBitmap - это дескриптор устройства вывода аппаратно-независимого растра (будет использован потом в процессе конвертирования растра).

data - указатель который укажет на данные нашего конвертированного изображения. Будет иметь смысл позже. Продолжайте читать :).

```
GLUquadricObj *quadratic; // Хранилище для наших квадратичных объектов

HDRAWDIB      hdd;        // Дескриптор для нашего рисунка
HBITMAP       hBitmap;    // Дескриптор устройства растра
HDC hdc = CreateCompatibleDC(0); // Создание совместимого контекста устройства
unsigned char* data = 0;         // Указатель на наше измененное в размерах изображение
```

Теперь немного ассемблера. Тем из вас, которые до этого никогда не пользовались ассемблером, не стоит бояться. Это может выглядеть загадочно, но это просто!

Пока я сочинял этот урок, я обнаружил весьма большую проблему. Первое видео, которое я получил, проигрывалось прекрасно, но при этом цвета отображались неверно. Везде, где должен быть красный цвет был синий, а где должен быть синий был красный. Я начал СХОДИТЬ С УМА. Я был убежден, что я сделал, где-то ошибку в коде. После просмотра всего кода, я не смог найти ошибку! Тогда я начал снова читать MSDN. Почему красные и голубые байты переставлены местами!? В MSDN говорилось, что 24 битные изображения в формате RGB!!! После более углубленного изучения я нашел, в чем состоит проблема. В Windows данные RGB (в картинках) фактически хранятся наоборот (BGR). В OpenGL, RGB это по настоящему… RGB.
После нескольких жалоб от фанатов Microsoft'а :) я решил добавить небольшое замечание! Я не ругаю Microsoft за то, что RGB данные сохраняются наоборот. Мне только очень непонятно, когда-то, что называется RGB в действительности является BGR!

Техническая заметка: есть "little endian" стандарт, а есть "big endian". Intel и аналоги Intel используют "little endian", где наименее значимый байт (LSB) идет первым. OpenGL пришел из SGI, где распространен «big endian», и OpenGL требует растровый формат в своем стандарте.

Замечательно! Так вот я тут с проигрывателем, который напоминает мне абсолютное дерьмо! Моим первым решением было поменять байты вручную в следующем цикле. Это работало, но очень медленно. Сытый по горло, я модифицировал код генерации текстур, чтобы он использовал GL_BGR_EXT вместо GL_RGB. Огромное увеличение скорости и цвета, все выглядело прекрасно! Так моя проблема была решена… или я так думал. Некоторые OpenGL драйверы имели проблемы с GL_BGR_EXT… . Назад к рисовальной доске :(.
После разговора с моим хорошим другом Максвелом Сэйлом, он посоветовал мне поменять местами байты, используя asm-код. Минуту позже был готов код, который я привел ниже! Может быть он не оптимизирован, но он работает и быстро работает!

Каждый кадр анимации сохраняется в буфере. Рисунок всегда будет иметь 256 пикселей в ширину, 256 пикселей в высоту и 1 байт на цвет (3 байта на пиксель). Код ниже будет проходить по буферу, и переставлять красные и синие байты. Красный хранится в ebx+0, а голубой в ebx+2. Мы двигаемся через буфер, обрабатывая по три байта за раз (пиксель состоит из трёх байтов). Мы будем делать это, пока все данные не будут переставлены.

У некоторых из вас были проблемы с использованием ASM кода, я полагаю, что я объясню, почему я использую его в своем уроке. Сначала я планировал использовать GL_BGR_EXT поскольку это работает. Но не на всех платах! Тогда я решил использовать метод перестановки с последнего урока (очень опрятный код перестановки методом XOR). Перестановка работала на всех машинах, но она не была быстрой. В прошлом уроке это хорошо работало. Но сейчас мы имеем дело с ВИДЕО В РЕАЛЬНОМ ВРЕМЕНИ. Вы хотите иметь самую быструю перестановку. Взвесив всё, ASM по моему мнению наилучший выбор!

Если у вас есть лучший путь чтобы сделать эту работу, пожалуйста… ИСПОЛЬЗУЙТЕ ЕГО! Я не говорю вам, как делать эти вещи. Я показываю, как я сделал это. Я также подробно объясняю свой код. Если вы хотите написать лучший код, то вы знаете, как устроен мой код, сделайте его проще и найдите альтернативный метод, если вы хотите написать ваш код!

```
void flipIt(void* buffer) // Функция меняющая красный и синий цвет
{
  void* b = buffer;     // Указатель на буфер
__asm                   // Начало asm кода
  {
    mov ecx, 256*256    // Установка счётчика (Размер блока памяти)
    mov ebx, b          // Указатель ebx на наши данные (b)
    label:              // Метка для цикла
      mov al,[ebx+0]    // Загружаем значение из ebx в регистр al
      mov ah,[ebx+2]    // Загружаем значение из ebx+2 в регистр ah
      mov [ebx+2],al    // Сохраняем данные в al из ebx+2
      mov [ebx+0],ah    // Сохраняем данные в ah из ebx

      add ebx,3         // Перемещаем указатель на три байта
      dec ecx           // Уменьшаем наш счётчик
  jnz label             // Если не равно нулю перемещаемся назад
  }
}
```

Код ниже открывает AVI файл в режиме чтения. SzFile - это название файла который мы хотим открыть. title[100] будет использован чтобы модифицировать заголовок окна (чтобы показать информацию об AVI файле).

Первое что нам надо сделать это вызвать AVIFileInit(). Она инициализирует библиотеку по работе с файлами AVI.

Есть много способов, чтобы открыть AVI файл. Я решил использовать функцию AVIStreamOpenFromFile(…). Она открывает единственный поток из AVI файла (AVI файлы могут содержать несколько потоков).

Параметры следующие: pavi - это указатель на буфер, который получает новый дескриптор потока. szFile - это имя файла, который мы желаем открыть (полный путь). Третий параметр - это тип потока. В нашей программе мы заинтересованы только в видео потоке (streamtypeVIDEO). Четвертый параметр обозначает номер потока (мы хотим только первый). OF_READ обозначает то, что мы хотим открыть файл ТОЛЬКО для чтения. Последний параметр - это указатель на класс идентификатора дескриптора, который мы хотим использовать. Если честно, то я не знаю для чего он. Позволим Windows выбрать его, послав NULL в последнем параметре.

Если возникнут, какие-то ошибки при открытии файла, то выскочит окно и даст вам знать о том, что поток не может быть открыт. Я не сделал так, чтобы при этой ошибке программа вызывала какую-то секцию кода. Если будет ошибка, программа будет пробовать проиграть файл. Добавление проверки потребовало бы усилий, а я очень ленивый :).

```
void OpenAVI(LPCSTR szFile)  // Вскрытие AVI файла (szFile)
{
 TCHAR  title[100];         // Будет содержать заголовок
 AVIFileInit();             // Открывает файл

 // Открытие AVI потока
 if (AVIStreamOpenFromFile(&pavi, szFile, streamtypeVIDEO, 0, OF_READ, NULL) !=0)
 {
  // Если ошибка
  MessageBox (HWND_DESKTOP, "Failed To Open The AVI Stream",
           "Error", MB_OK | MB_ICONEXCLAMATION);
 }
```

Если мы сделали это, то можно считать что файл был открыт и данные потока локализованы! После этого мы получаем немного информации от AVI файла с помощью AVIStreamInfo(…).

Ранее мы создали структуру psi, которая будет содержать информацию о нашем AVI потоке. Эта структура заполнится информацией с помощью первой строки кода ниже. Всё от ширины потока (в пикселях) до частоты кадров анимации сохранено в psi. Для тех, кто хочет добиться точной скорости воспроизведения сделайте, как я сказал. Более подробную информацию ищите об AVIStreamInfo в MSDN.

Мы можем вычислить ширину кадра отняв левую границу окна из правой. Это будет точная ширина в пикселях. Высота кадра получается, когда мы вычитаем верхнюю границу из нижней. Это даст нам высоту в пикселях.

Затем мы находим номер последнего кадра из AVI файла используя AVIStreamLength(…). Она возвращает число кадров анимации в AVI файле. Результат сохранен в lastframe.
Вычисление частоты кадров довольно просто. Кадры в секунду = psi.dwRate / psi.dwScale. Это значение совпадает со значением, которое можно получить в свойствах AVI-файла, если щелкнуть по нему правой кнопкой мыши в Проводнике. Так почему же мы используем mpf спросите вы? Когда я впервые написал этот код, я попробовал использовать этот метод чтобы выбрать правильный кадр анимации. Я столкнулся с проблемой… У меня есть файл face2.avi продолжительностью 3.36 секунды. Частота кадров 29.974 кадров в секунду. Видео имеет 91 кадр. Если вы умножите 3.36 на 29.974 вы получите 100 кадров. Очень странно!

Я решил переписать код немного по-другому. Вместо вычисления частоты кадров в секунду, я посчитал, как долго каждый кадр показывается на экране. AVIStreamSampleToTime() конвертирует позицию анимацию в «сколько миллисекунд требуется чтобы добраться до этой позиции».Таким образом мы вычисляем сколько миллисекунд имеет все видео с помощью получения времени (в миллисекундах) последнего кадра. Тогда мы делим результат на общее количество кадров анимации (lastframe). Это даёт нам время необходимое для показа одного кадра. Мы сохраняем полученный результат в переменной mpf (millisecond per frame – число миллисекунд на кадр). Вы также можете посчитать, сколько отводится миллисекунд на кадр посредством получения времени первого кадра анимации с помощью вот этого кода: AVIStreamSampleToTime(pavi,1). Простой и отлично работающий способ! Большое спасибо Альберту Чаулку за эту идею!
Причина, по которой я говорю приблизительное число миллисекунд на кадр та, что mpf целое и любое дробное значение будет округлено!

```
 AVIStreamInfo(pavi, &psi, sizeof(psi));   // Записываем информацию о потоке в psi
 width=psi.rcFrame.right-psi.rcFrame.left; // Ширина = правая граница минус левая
 height=psi.rcFrame.bottom-psi.rcFrame.top;// Высота равна верх минус низ
 lastframe=AVIStreamLength(pavi);          // Последний кадр потока
 // Вычисление приблизительных миллисекунд на кадр
 mpf=AVIStreamSampleToTime(pavi,lastframe)/lastframe;
```

Поскольку OpenGL требует, чтобы данные в текстуре были кратны двум, и потому что большинство видео размеров 160x120, 320x240 и некоторые другие странные размеры, нам нужен быстрый способ на лету изменить размеры видео в формат, который мы можем использовать как текстуру. Чтобы сделать это мы воспользуемся преимуществом Windows DIB функций.

Первую вещь, которую мы сделаем это опишем тип нужного нам изображения. Чтобы сделать это мы заполняем структур bmih типа BitmapInfoHeader нужными данными. Мы начнём с изменения размера структуры. Тогда мы установим bitplanes в 1. Три байта данных это 24 битовый цвет (RGB). Мы хотим изображение 256x256 пикселей и, наконец, мы хотим, чтобы данные возвращались как UNCOMPRESSED RGB (BI_RGB).

Функция CreateDIBSection создает изображение, в которое мы и запишем рисунок. Если всё прошло нормально hBitmap укажет нам на значения битов изображения. Hdc - это дескриптор контекста устройства (DC). Второй параметр - это указатель на структуру BitmapInfo. Это структура содержит информацию об изображении как было сказано выше. Третий параметр (DIB_RGB_COLORS) определяет, что данные в формате RGB.data - это указатель на переменную, которая получает указатель на DIB данные. Если мы укажем в качестве пятого параметра NULL, память будет выделена под наш рисунок. Наконец последний параметр может быть игнорирован (установлен в NULL).

Цитата из MSDN: Функция SelectObject выбирает объект для контекста устройства (DC).

Теперь мы создали DIB, который мы можем непосредственно выводить на экран. Вау :).

```
 bmih.biSize        = sizeof (BITMAPINFOHEADER); // Размер BitmapInfoHeader'a
 bmih.biPlanes      = 1;       // Размер
 bmih.biBitCount    = 24;      // Формат битов
 bmih.biWidth       = 256;     // Ширина(256 пикселов)
 bmih.biHeight      = 256;     // Высота(256 пикселов)
 bmih.biCompression = BI_RGB;  // Цветовой режим (RGB)

 hBitmap = CreateDIBSection (hdc, (BITMAPINFO*)(&bmih),
                        DIB_RGB_COLORS, (void**)(&data), NULL, NULL);
 SelectObject (hdc, hBitmap)   // Выбор hBitmap в наш контекст устройства (hdc)
```

Ещё несколько вещей мы должны сделать, чтобы быть готовыми к чтению кадров из AVI. Следующее что нам надо сделать, это подготовить нашу программу к извлечению кадров из файла с видео-фильмом. Для этого мы используем AVIStreamGetFrameOpen(…).

Вы можете передавать структуру подобно тому, как мы выше передавали второй параметр, чтобы получить заданный видео формат. К сожалению, единственное, что мы можем изменять при этом ширину и высоту возвращаемого изображения.

Если всё прошло хорошо, объект GETFRAME возвращен (он нужен нам, для того чтобы читать кадры). Если есть какие-нибудь проблемы, окно выскочит и сообщит вам об ошибке.

```
 pgf=AVIStreamGetFrameOpen(pavi, NULL); // Создание PGETFRAME с нужными нам параметрами
 if (pgf==NULL)
 {
  // Если ошибка
  MessageBox (HWND_DESKTOP, "Failed To Open The AVI Frame",
            "Error", MB_OK | MB_ICONEXCLAMATION);
 }
```

Код ниже выводит ширину, высоту и количество кадров в заголовок. Мы показываем заголовок с помощью функции SetWindowText(…). Запустите программу в оконном режиме, чтобы увидеть, что делает этот код.

```
 // Информация для заголовка (Ширина/Высота/Кол-во кадров)
 wsprintf (title, "NeHe's AVI Player: Width: %d, Height: %d, Frames: %d",
              width, height, lastframe);
 SetWindowText(g_window->hWnd, title);  // Изменение заголовка
}
```

Теперь интересное…мы захватываем кадр из AVI и конвертируем его к используемым размерам изображения и разрядности цвета. lpbi будет содержать информацию BitmapInfoHeader для кадра анимации. Во второй строчке кода мы выполняем сразу несколько вещей. Сначала мы захватываем кадр анимации. Кадр, который мы хотим, задан frame. Это считает кадр анимации и заполнит lpbi информацией для этого кадра.

Еще интересного ... нам необходим указатель на данные изображения. Чтобы сделать это мы должны опустить информацию заголовка (lpbi->biSize). Одну вещь я не делал пока не сел писать этот урок. Она состоит в том, что мы должны также опустить любую информацию о цвете. Чтобы сделать это мы должны сложить цвета, умноженные на размер RGBQUAD (biClrUsed*sizeof(RGBQUAD)). После выполнения ВСЕГО, что мы хотели :) мы оставлены один на один с указателем на данные (pdata).
Сейчас нам надо конвертировать кадр анимации к размеру используемой текстуры также, мы должны преобразовать данные в RGB формат. Чтобы сделать это мы используем DrawDibDraw(…).

Краткое замечание. Мы можем рисовать непосредственно в наш DIB. Это делает DrawDibDraw(…). Первый параметр - это дескриптор нашего DrawDib DC. Второй - это дескриптор на наш DC. Следующие параметры - это верхний левый угол (0,0) и правый нижний угол (256,256) результирующего прямоугольника.

lpbi – указатель на BitmapInfoHeader информацию для кадра который мы сейчас читаем. pdata – указатель на данные изображения для этого кадра.

Теперь у нас есть верхний левый угол (0,0) исходного изображения (текущий кадр) и правый нижний угол кадра (ширина и высота кадра). Последний параметр пусть будет нуль.

Таким образом, мы преобразуем изображение любого размера и разрядности цвета к 256*256*24.

```
void GrabAVIFrame(int frame) // Захват кадра
{
  LPBITMAPINFOHEADER lpbi;  // Содержит BitmapInfoHeader
  // Получение данных из потока
  lpbi = (LPBITMAPINFOHEADER)AVIStreamGetFrame(pgf, frame);
  // Указатель на данные возвращенные AVIStreamGetFrame
  // (Пропуск заголовка для получения указателя на данные)
  pdata=(char *)lpbi+lpbi->biSize+lpbi->biClrUsed * sizeof(RGBQUAD);
  // Преобразование информации в нужный нам формат
  DrawDibDraw (hdd, hdc, 0, 0, 256, 256, lpbi, pdata, 0, 0, width, height, 0);
```

Теперь у нас есть наш кадр анимации, но красные и голубые байты переставлены. Чтобы решить эту проблему мы переходим к нашему быстрому flipIt(…) коду. Помните, data – это указатель на переменную, которая получает указатель на расположения битовых значений DIB'a. Это означает то, что после того как мы вызовем DrawDibDraw, data укажет на наши изменённые (256*256*24) растровые данные.

Первоначально я создавал текстуру для каждого кадра анимации. Я получил несколько писем предлагающих мне использовать glTexSubImage2D(). После чтения «Красной книги по OpenGL», я наткнулся на следующую цитату: «Создание текстуры может быть в вычислительном отношении более дорогостоящим, чем изменить существующую. В OpenGL версии 1.1 есть подпрограммы для замены всей площади или части текстуры на новую информацию. Это может быть полезно для некоторых приложений, которые делаю анимацию в реальном времени, и захвата видео изображений в текстуры. Для этих приложений имеет смысл создавать одну текстуру и использовать glTexSubImage2D() чтобы потом неоднократно заменять данные текстуры новыми видео изображениями».

Лично я не замечал огромного увеличения скорости, но если у вас слабая видеокарта вы могли бы стать свидетелем этого. Параметры для glTexSubImage2D() следующие: наша цель - двухмерная текстура (GL_TEXTURE_2D). Уровень детализации (0), который используется для мипмэпинга. Смещения x(0) и y(0) которые показывают функции, где начать копирование (0,0 нижний левый угол текстуры). У нас есть размеры изображения, которое мы хотим копировать (256*256). GL_RGB - это формат данных. Мы копируем беззнаковые данные. Очень просто.

Заметка Кевина Рогерса: я только хотел указать на другую причину использования glTexSubImage2D. Это не только будет быстрее для большинства приложений OpenGL, но целевая область может быть по размеру не обязательно кратной степени 2. Это особенно удобно для воспроизведения, так как типичные размеры кадра редко кратные степени 2 (часто 320*200 или подобные). Это даёт вам достаточную гибкость, чтобы запустить видео поток в его первоначальном варианте, чем искажать и отсекать каждый кадр, для того чтобы приспособить его к вашим размерам.

Важно обратить ваше на то, что вы не можете обновлять текстуру, если вы до этого её не создали! Мы создаем текстуру в Initialize().

Я также хотел упомянуть… Если вы планируете использовать больше одной текстуры в вашем проекте удостоверьтесь, что вы связываете текстуру (glBindTexture()) . Если вы не свяжете текстуру она не будет обновлена!

```
  flipIt(data); // Перестановка красных и синих байтов
  // Обновление текстуры
  glTexSubImage2D (GL_TEXTURE_2D, 0, 0, 0, 256, 256, GL_RGB, GL_UNSIGNED_BYTE, data);
}
```

Следующая секция кода вызывается, когда программа завершается. Мы закрываем DC, и ресурсы освобождаются. Тогда мы разблокируем ресурсы AVI GetFrame. Наконец мы завершаем поток и закрываем файл.

```
void CloseAVI(void)             // Функция закрытия
{
  DeleteObject(hBitmap);        // Уничтожение устройства растра
  DrawDibClose(hdd);            // Закрытие контекста DrawDib устройства
  AVIStreamGetFrameClose(pgf);  // Закрытие объекта GetFrame
  AVIStreamRelease(pavi);       // Завершение потока
  AVIFileExit();                // Закрытие файла
}
```

Инициализация довольно проста. Мы устанавливаем угол в 0. Далее мы открываем DrawDib библиотеку (которая получает DC). Если все хорошо, hdd становится дескриптором на только что созданный контекст устройства.

Наш экран черный, включается тестирование глубины, и т.д..

Затем мы создаем новый квадратичный объект. quadratic - это указатель на наш новый объект. Мы устанавливаем сглаженные нормали, и включаем автогенерацию текстурных координат для нашего квадратичного объекта.

```
BOOL Initialize (GL_Window* window, Keys* keys) //Инициализация
{
  g_window  = window;
  g_keys    = keys;

  // Начало инициализации
  angle = 0.0f;                         // Установка угла в ноль
  hdd = DrawDibOpen();                  // Получение контекста устройства
  glClearColor (0.0f, 0.0f, 0.0f, 0.5f);    // Черный фон
  glClearDepth (1.0f);                  // Установка буфера глубины
  glDepthFunc (GL_LEQUAL);              // Тип тестирования глубины (Less или Equal)
  glEnable(GL_DEPTH_TEST);              // Включение теста глубины
  glShadeModel (GL_SMOOTH);             // Выбор гладкости
  // Очень аккуратная установка перспективы
  glHint (GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);

  quadratic=gluNewQuadric();            // Создание нового квадратичного объекта
  gluQuadricNormals(quadratic, GLU_SMOOTH); // Сглаженные нормали
  gluQuadricTexture(quadratic, GL_TRUE);    // Создание текстурных координат
```

В следующем кусочке кода, мы включаем отображение двухмерных текстур, и мы устанавливаем фильтры текстур в GL_NEAREST (быстро, но грубо) и мы устанавливаем сферическое наложение (чтобы создать эффект наложения окружения). Проиграйтесь с фильтрами. Если у вас мощный компьютер, попробуйте GL_LINEAR для более гладкой анимации.

После установки нашей текстуры и сферического наложения мы открываем .AVI файл. Файл называется face2.avi и он расположен в каталоге 'data'.

Последнее, что мы должны сделать – это создать нашу первоначальную текстуру. Мы должны сделать это чтобы использовать glTexSubImage2D() для модификации нашей текстуры в GrabAVIFrame().

```
  glEnable(GL_TEXTURE_2D); // Включение двухмерных текстур
  // Установка фильтра увеличения текстуры
  glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST);
  // Установка фильтра уменьшения текстуры
  glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST);
  // Включение автогенерации текстурных координат по координате S сферического наложения
  glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
  // Включение автогенерации текстурных координат по координате T сферического наложения
  glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
                        OpenAVI("data/face2.avi");          // Откроем видео-файл
  // Создание текстуры
  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 256, 256, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
 return TRUE; //Возвращение true (инициализация успешна)
}
```

При завершении мы вызываем CloseAVI(). Это корректно закроет AVI файл и все используемые ресурсы.

```
void Deinitialize (void) //Вся деинициализация здесь
{
  CloseAVI();           // Закрываем AVI
}
```

Далее мы проверяем клавиши и обновляем наше вращение (angle) относительно прошедшего времени. Сейчас я не буду подробно объяснять код. Мы проверяем, нажат ли пробел. Если это так, то мы выполняем следующий по списку эффект. У нас есть три эффекта (куб, сфера, цилиндр) и когда выбран четвёртый эффект (effect=3) ничего не рисуется…показывается лишь сцена! Когда выбран четвёртый эффект и нажат пробел, то мы возвращаемся к первому эффекту (effect = 0). Да, я знаю, я должен был назвать это ОБЬЕКТОМ :).
Затем мы проверяем, нажата ли клавиша ‘B’ если это так, то мы переключаем фон (bg) от включенного состояния в выключенное или наоборот.
Для отображения окружения мы проделаем то же самое. Мы проверяем, нажата ли ‘E’. Если это так, то мы переключаем env от TRUE к FALSE и наоборот. То есть, включено наложение окружения или нет.
Угол увеличивается на крошечную долю каждый раз при вызове Update(). Я делю время на 60.0f, чтобы немного замедлить скорость вращения.

```
void Update (DWORD milliseconds) // Движение обновляется тут
{
  if (g_keys->keyDown [VK_ESCAPE] == TRUE) // Если ESC нажат
  {
    TerminateApplication (g_window); // Завершение приложения
  }
  if (g_keys->keyDown [VK_F1] == TRUE) // Если F1 нажата
  {
    ToggleFullscreen (g_window); // Включение полноэкранного режима
  }
  if ((g_keys->keyDown [' ']) && !sp) // Пробел нажат и не удерживается
  {
    sp=TRUE;      // Установка sp в истину
    effect++;     // Изменение эффекта (увеличение effect)
    if (effect>3) // Превышен лимит?
      effect=0;   // Возвращаемся к нулю
  }
  if (!g_keys->keyDown[' ']) // Если пробел отпущен
    sp=FALSE; // Установка sp в False
  if ((g_keys->keyDown ['B']) && !bp) // ‘B’ нажат и не удерживается
  {
    bp=TRUE; // Установка bp в True
    bg=!bg;  // Включение фона Off/On
  }

if (!g_keys->keyDown['B']) // Если ‘B’ отпущен
   bp=FALSE; //Установка bp в False
  if ((g_keys->keyDown ['E']) && !ep) // Если ‘E’ нажат и не удерживается
  {
    ep=TRUE;  // Установка ep в True
    env=!env; // Включение отображения среды Off/On
  }
  if (!g_keys->keyDown['E']) // Если 'E' отпущен?
    ep=FALSE;               // Установка ep в False
  angle += (float)(milliseconds) / 60.0f; // Обновление angle на основе времени
```

В первоначальном варианте урока, все AVI файлы проигрывались с одинаковой скоростью. После этого программа была переписана, чтобы запустить видео с правильной скоростью. next - увеличивает число миллисекунд, которое прошло после вызова этой секции кода в последний раз. Если ранее в уроке мы вычисляли, как долго каждый кадр должен быть отображен (mpf). Чтобы вычислить текущий кадр, мы берём прошедшее время и делим его на mpf. После этого нам надо удостовериться в том, что номер текущего кадра анимации не больше общего числа кадров. Если это так, то анимация будет сброшена в нуль и начата заново.

Код ниже пропустит кадры, если ваш компьютер тормозит или другое приложение занимает процессор. Если вы хотите, чтобы каждый кадр был отображен независимо от того тормозит ли компьютер, вы можете проверить является ли next больше mpf и, если это так, то сбросьте next и увеличьте frame на единицу. Любой способ сработает, но код ниже больше подходит для более мощных машин.

Если вы чувствуете силы, попробуйте добавить перемотку, быструю перемотку, паузу или обратный ход проигрывания!

```
 next+= milliseconds;  // Увеличение next основанное на таймере (миллисекундах)
 frame=next/mpf;       // Вычисление текущего кадра
 if (frame>=lastframe) // Не пропустили ли мы последний кадр?
 {
   frame=0;            // Сбрасываем frame назад в нуль (начало анимации)
   next=0;             // Сбрасываем таймер анимации (next)
 }
```

Теперь код рисования. Мы очищаем буфер глубины и экрана. Затем мы получаем кадр анимации. Снова, я постараюсь сделать код простым!
Вы передаете требуемый кадр (frame) функции GrabAVIFrame(). Довольно просто! Конечно, если бы хотели воспроизводить многопотоковый AVI вы должны были бы передать текстурный идентификатор.

```
void Draw (void)        // Прорисовка сцены
{
 // Очистка экрана и буфера глубины
 glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

 GrabAVIFrame(frame);   // Захват кадра анимации
```

Код ниже проверяет, хотим ли мы видеть фоновое изображение. Если bg равен TRUE, мы сбрасываем матрицу и прорисовываем одну текстуру в форме квадрата (отображает кадр AVI) достаточно большую чтобы заполнить экран. Квадрат нарисован на 20 единиц вглубь экрана (-20), поэтому он всегда показывается позади объекта.

```
 if (bg)               // Фоновое изображение показывать?
 {
   glLoadIdentity();    // Сброс матрицы просмотра
   glBegin(GL_QUADS);   // Начало прорисовки фонового рисунка
     // Передняя грань
    glTexCoord2f(1.0f, 1.0f); glVertex3f( 11.0f,  8.3f, -20.0f);
    glTexCoord2f(0.0f, 1.0f); glVertex3f(-11.0f,  8.3f, -20.0f);
    glTexCoord2f(0.0f, 0.0f); glVertex3f(-11.0f, -8.3f, -20.0f);
   glTexCoord2f(1.0f, 0.0f); glVertex3f( 11.0f, -8.3f, -20.0f);
  glEnd();             // Конец рисования
 }
```

После прорисовки фона (или не прорисовки), мы сбрасываем матрицу (это переводит нас в центр экрана по всем трём измерениям). Далее мы сдвигаем матрицу на 10 единиц вглубь экрана (-10).
После этого мы проверяем, равна ли переменная env значению TRUE. Если это так, то мы включаем сферическое наложение для создания эффекта наложения окружения.

```
 glLoadIdentity ();     // Сброс матрицы
 glTranslatef (0.0f, 0.0f, -10.0f); // На десять единиц в экран
 if (env)               // Включено отображение эффектов
 {
   glEnable(GL_TEXTURE_GEN_S); // Вкл. автогенерация координат текстуры по S (Новое)
   glEnable(GL_TEXTURE_GEN_T); // Вкл. автогенерация координат текстуры по T (Новое)
 }
```

Я добавил следующий код в последнюю минуту. Он вращает сцену по оси x и оси y (основываясь на значении angle) и, наконец, сдвигает сцену на две единицы по оси z. Это переместит все вглубь экрана. Если вы удалите эти три строки кода ниже, объект будет просто крутиться в середине экрана. С этими тремя строками, объекты будут немного двигаться и одновременно вращаться :).

Если вы не понимаете, как делается вращение и передвижение… этот урок для вас слишком сложный :).

```
  glRotatef(angle*2.3f,1.0f,0.0f,0.0f); // Немного вращает объекты по оси x
  glRotatef(angle*1.8f,0.0f,1.0f,0.0f); // Делает то же самое только по оси y
  glTranslatef(0.0f,0.0f,2.0f); // После вращения перемещение
```

Код ниже проверяет, какой из эффектов мы хотим прорисовать. Если значение effect равно 0, мы делаем небольшое вращение и рисуем куб. Поворот вращает куб по x,y и z осям. К настоящему времени вы должны иметь код, чтобы создать куб, родивший в вашей голове :).

```
switch (effect) // Какой эффект?
{
  case 0: // Эффект  0 - Куб
    glRotatef (angle*1.3f, 1.0f, 0.0f, 0.0f); // Вращение по оси x
    glRotatef (angle*1.1f, 0.0f, 1.0f, 0.0f); // Вращение по оси y
    glRotatef (angle*1.2f, 0.0f, 0.0f, 1.0f); // Вращение по оси z
    glBegin(GL_QUADS); // Начало рисования куба
      //Передняя грань
      glNormal3f( 0.0f, 0.0f, 0.5f);
      glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f,  1.0f);
      glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f,  1.0f);
      glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f,  1.0f,  1.0f);
      glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f,  1.0f,  1.0f);
      //Задняя грань
      glNormal3f( 0.0f, 0.0f,-0.5f);
      glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
      glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f,  1.0f, -1.0f);
     glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f,  1.0f, -1.0f);
      glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
      //Верхняя грань
      glNormal3f( 0.0f, 0.5f, 0.0f);
      glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f,  1.0f, -1.0f);
      glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f,  1.0f,  1.0f);
      glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f,  1.0f,  1.0f);
      glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f,  1.0f, -1.0f);
      // Нижняя грань
      glNormal3f( 0.0f,-0.5f, 0.0f);
      glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
      glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
      glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f,  1.0f);
      glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f,  1.0f);
      // Правая грань
      glNormal3f( 0.5f, 0.0f, 0.0f);
      glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
      glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f,  1.0f, -1.0f);
      glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f,  1.0f,  1.0f);
      glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f,  1.0f);
      // Левая грань
      glNormal3f(-0.5f, 0.0f, 0.0f);
      glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
      glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f,  1.0f);
      glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f,  1.0f,  1.0f);
      glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f,  1.0f, -1.0f);
    glEnd();            // Конец рисования нашего куба
    break;              // Конец нулевого эффекта
```

Теперь мы выводим сферу. Мы начинаем с небольшого вращения по всем осям. Далее мы рисуем сферу. Сфера будет иметь радиус 1.3f из 20 ломтиков и 20 срезов. Я выбрал значение двадцать, потому что я не хотел, чтобы сфера была совершенно гладкой. При использовании меньшого количества кусков сфера будет грубой (не гладкой) и будет не совсем очевидно, что сфера вращается, когда сферическое наложение включено. Проиграйтесь с этими значениями. Важно обратить ваше внимание на то, что большая детализация требует больше процессорного времени.

```
case 1: // Эффект 1 - сфера
   glRotatef(angle*1.3f, 1.0f, 0.0f, 0.0f); // Вращение по оси x
   glRotatef(angle*1.1f, 0.0f, 1.0f, 0.0f); // Вращение по оси y
   glRotatef(angle*1.2f, 0.0f, 0.0f, 1.0f); // Вращение по оси z
   gluSphere(quadratic,1.3f,20,20); // Прорисовка сферы
   break; //Конец прорисовки сферы
```

Сейчас мы нарисуем цилиндр. Мы начнём с простого вращения по x, y, z осям. Наш цилиндр будет иметь одинаковый верхний и нижний радиус равный 1 единице. Цилиндр будет иметь в высоту 3 единицы, и состоять из 32 ломтиков и 32 срезов. Если вы уменьшаете число кусков, цилиндр будет составлен из меньшого количества полигонов и будет казаться менее округленным.
Перед тем как рисовать цилиндр мы сдвинемся на –1.5 единиц по оси z. С помощью этого мы заставим наш цилиндр вращаться вокруг центра экрана. Общее правило к центрированию цилиндра: надо разделить на 2 его высоту и сдвинуться на полученный результат в отрицательном направлении по оси z. Если вы понятия не имеете о том, что я говорю, удалите строчку с translatef(…). Цилиндр будет двигаться вокруг своей оси, вместо центральной точки.

```
case 2: // Эффект 2 - цилиндр
   glRotatef (angle*1.3f, 1.0f, 0.0f, 0.0f);    // Вращение по оси x
   glRotatef (angle*1.1f, 0.0f, 1.0f, 0.0f);    // Вращение по оси y
   glRotatef (angle*1.2f, 0.0f, 0.0f, 1.0f);    // Вращение по оси z
   glTranslatef(0.0f,0.0f,-1.5f);               // Центр цилиндра
   gluCylinder(quadratic,1.0f,1.0f,3.0f,32,32); // Прорисовка цилиндра
   break; //Конец прорисовки цилиндра
  }
```

Затем мы проверяем, является ли env TRUE. Если это так, то мы отключаем сферическое наложение. Мы вызываем glFlush() чтобы сбросить конвейер визуализации (чтобы быть уверенными, что прорисовка текущего кадра полностью завершена до начала прорисовки следующего кадра).

```
 if (env) // Включено наложение окружения?
  {
   glDisable(GL_TEXTURE_GEN_S); // Вкл. автогенерация координат текстуры по S (Новое)
   glDisable(GL_TEXTURE_GEN_T); // Вкл. автогенерация координат текстуры по T (Новое)
  }
  glFlush ();              // Визуализация
}
```

Я надеюсь, что Вам понравился этот урок. Сейчас 2 часа ночи… Я работал над этим уроком последние шесть часов. Звучит безумно, но описать вещи так, чтобы это имело смысл, это нелегкая задача. Я прочитал урок три раза, и всё ещё пробую сделать его проще. Верите вы мне или нет, но для меня это очень важно, чтобы вы понимали, как работает код и почему он работает. Именно поэтому я нескончаемо повторяюсь, чрезмерно комментирую и т.д..
В любом случае. Мне бы хотелось услышать комментарии по поводу этого урока. Если вы найдете ошибки или вы хотели бы помочь мне сделать урок лучше, пожалуйста, войдите со мной в контакт, поскольку я сказал, что это моя первая попытка с AVI. Обычно я не пишу урок по теме, которую я только что изучил, но мое волнение извлекло всё самое лучшее из меня, плюс факт, что по этой теме очень мало информации обеспокоил меня. Я надеюсь, что я открою дверь потоку высококачественных демок с проигрыванием AVI и исходного кода. Может случиться… может нет. В любом случае вы можете использовать этот код тогда когда пожелаете нужным.

Огромное спасибо Фредстеру за его AVI файл с изображением лица. Лицо было одним из шести AVI, которые он послал мне для моего урока. Ни один мой вопрос не остался без ответа. Я посылал ему письма, и он выручил меня… Большое спасибо.

Большое спасибо, Джонатану Блоку. Если бы не он этого урока не существовало бы. Он заинтересовал меня AVI форматом высылая кусочки кода из его персонального AVI проигрывателя. Он также ответил мне на все вопросы относительно его кода. Важно то, что я ничего не заимствовал из его кода, его код был использован только для того чтобы понять, как проигрыватель работает. Мой проигрыватель открывает, декодирует и запускает AVI файлы, используя совершенно другой код!

Большое спасибо, каждому посетителю сайта. Без Вас этого сайта не было бы!

Урок 36. Радиальное размытие и текстурный рендеринг

Radial Blur & Rendering To A Texture

Привет! Меня зовут Дарио Корно (Dario Corno), еще меня знают как rio of SpinningKids. Я занимаюсь построением сцен с 1989 года. Прежде всего, хочу объяснить, для чего я решил написать эту небольшую статью. Я бы хотел, чтобы все, пожелавшие скачать демонстрационную программу, смогли разобраться в ней и в эффектах, которые в ней реализованы.

Демонстрационные программы часто показывают, как путем простого, а иногда грубого программирования добиться неплохих художественных возможностей программы. Вы можете увидеть просто убойные эффекты в демонстрационных примерах, существующих на данный момент. Кучу таких программ вы найдете по адресу http://www.pouet.net или http://ftp.scene.org.

Теперь закончим со вступлением и приступим к изучению программы.

Я покажу, как создавать эффект «eye candy» (реализованный в программе), похожий на радиальное размытие. Иногда его относят к объемному освещению, но не верьте этому, это просто имитация радиального размытия! ;D

Радиальное размытие обычно достигалось (когда еще не было аппаратного рендеринга) смазыванием каждого пикселя исходного изображения в направлении от центра размытия.

На современной аппаратуре пока довольно трудно реализовать размытие с помощью использования буфера цвета (по крайней мере, это относится ко всем «gfx»-видеокартам), поэтому для получения подобного эффекта мы пойдем на маленькую хитрость.

И как плюс, при изучении построения эффекта радиального размытия вы научитесь простому способу текстурного рендеринга!

В качестве трехмерного объекта я выбрал пружину, поскольку это нетривиальная фигура, да и обычные кубы мне уже порядком поднадоели :)

Хочу заметить, что эта статья является скорее руководством по созданию эффектов. Я не буду слишком вдаваться в подробности программы. Постарайтесь больше понимать все сердцем :)

Ниже приведены объявления переменных и необходимый заголовочный файл:

```
#include <math.h>  // Нам потребуются некоторые математические операции

float    angle;          // Угол вращения спирали
float    vertexes[3][3]; // Массив трех вершин
float    normal[3];      // Массив нормали
Gluint  BlurTexture;     // Номер текстуры
```

Функция EmptyTexture() создает пустую текстуру и возвращает ее номер. Нам потребуется выделить некоторую свободную память (а точнее 128*128*4 беззнаковых целочисленных).

128*128 – размер текстуры (128 пикселей ширины и столько же высоты), цифра 4 означает, что для каждого пикселя нам нужно 4 байта для хранения компонент RED, GREEN, BLUE и ALPHA.

```
GLuint EmptyTexture()     // Создание пустой текстуры
{
  GLuint txtnumber;      // Идентификатор текстуры
  unsigned int* data;    // Указатель на хранимые данные

  // Выделение памяти под массив пустой текстуры (128x128x4 байт)
  data = (unsigned int*)new GLuint[((128 * 128)* 4 * sizeof(unsigned int))];
```

После выделения памяти нам нужно обнулить ее при помощи функции ZeroMemory, передав ей указатель (data) и размер обнуляемой памяти.

Хочу обратить ваше внимание на то, что характеристики магнификации (увеличения) и минификации (уменьшения) текстуры (это методы, применяемые для изменения размера текстуры до размера объекта, на который эта текстура «натягивается» - прим. перев.) устанавливаются параметром GL_LINEAR (определяющим цвет точки как среднее арифметическое всех элементов текстуры, входящих в отображаемый пиксел - прим. перев.). А параметр GL_NEAREST при растяжении текстуры дал ба нам не очень красивую картинку.

```
 // Очистка памяти массива
  ZeroMemory(data,((128 * 128)* 4 * sizeof(unsigned int)));
  glGenTextures(1, &txtnumber);           // Создать 1 текстуру
  glBindTexture(GL_TEXTURE_2D, txtnumber); // Связать текстуру
  // Построить текстуру по информации в data
  glTexImage2D(GL_TEXTURE_2D, 0, 4, 128, 128, 0,
     GL_RGBA, GL_UNSIGNED_BYTE, data);
  glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
  delete [] data;   // Удалить data
  return txtnumber; // Вернуть идентификатор текстуры
}
```

Следующая функция нормализует длину векторов нормали. Векторы представлены массивом из 3-х элементов типа float, в котором 1-й элемент – это X, второй – Y и третий – Z. Нормализованный вектор (Nv) выражается через Vn = (Vox / |Vo| , Voy / |Vo|, Voz / |Vo|), где Vo – исходный вектор, |Vo| - его длина, и x, y, z – его компоненты. В программе необходимо сделать следующее: вычислить длину исходного вектора: sqrt(x^2 + y^2 + z^2) ,где x,y,z - это 3 компоненты вектора. Затем надо разделить все три компоненты вектора нормали на полученное значение длины.

```
// Преобразовать вектор нормали (3 координаты) в единичный вектор с единичной длиной
void ReduceToUnit(float vector[3]) {
  float length;      // Переменная для длины вектора
  // Вычисление длины вектора
  length = (float)sqrt((vector[0]*vector[0]) + (vector[1]*vector[1])
         + (vector[2]*vector[2]));
  // Предотвращение деления на нуль
  if(length == 0.0f)   length = 1.0f;
  vector[0] /= length;    // Деление всех трех элементов
  vector[1] /= length;    // на длину единичного вектора
  vector[2] /= length;    // нормали
}
```

Следующая функция подсчитывает нормаль, заданную тремя вершинами (хранящимися в трех массивах типа float). У нас есть два параметра: v[3][3] и out[3], первый из них – матрица со значениями типа float, размерностью m=3 и n=3, где каждая строка из трех элементов является вершиной треугольника. out – это переменная, в которую мы поместим результат - вектор нормали.

Немного (простой) математики. Мы собираемся воспользоваться знаменитым векторным произведением, которое определяется как операция между двумя векторами, дающая в результате вектор, перпендикулярный обоим исходным векторам. Нормаль - это вектор, ортогональный к поверхности, но направленный в противоположную сторону (и имеющий нормализованную (единичную) длину). Теперь представьте себе, что два вектора расположены вдоль сторон одного треугольника, ортогональный вектор (полученный в результате умножения этих векторов) этих двух сторон треугольника является нормалью этого треугольника.

Осуществить это легче, чем понять.

Сначала найдем вектор, идущий от вершины 0 к вершине 1, затем вектор, идущий от вершины 1 к вершине 2, что просто осуществляется путем вычитания каждой компоненты (координаты) стартовой вершины из компоненты следующей вершины стороны треугольника (соответствующей искомому вектору) в порядке обхода треугольника. Так мы получим вектора сторон треугольника. А в результате векторного произведения этих двух векторов (V*W) получим вектор нормали треугольника (если быть точнее, - вектор нормали плоскости, образованной сторонами этого треугольника, так как известно, что для образования плоскости достаточно двух несовпадающих векторов – прим.перев.).

Давайте взглянем на программу.

V[0][] – это первая вершина, V[1][] – это вторая вершина, V[2][] – это третья вершина. Каждая вершина состоит из: V[][0] – x-координата вершины, V[][1] – y-координата вершины, V[][2] – z-координата вершины.

Простым вычитанием всех соответствующих координат первой вершины из координат следующей мы получим ВЕКТОР от первой вершины к следующей. v1[0] = v[0][0] - v[1][0] - это выражение подсчитывает X-компоненту ВЕКТОРА, идущего от ВЕРШИНЫ 0 к вершине 1. v1[1] = v[0][1] - v[1][1] - это выражение подсчитывает Y-компоненту, v1[2] = v[0][2] - v[1][2] подсчитывает Z компоненту и так далее.

Так мы получим два вектора, что даст нам возможность вычислить нормаль треугольника.
Вот формула векторного произведения:

```
out[x] = v1[y] * v2[z] - v1[z] * v2[y]
out[y] = v1[z] * v2[x] - v1[x] * v2[z]
out[z] = v1[x] * v2[y] - v1[y] * v2[x]
```

В итоге в массиве out[] у нас будет находиться нормаль треугольника.

```
void calcNormal(float v[3][3], float out[3])
// Вычислить нормаль для четырехугольников используя 3 точки
{
 float v1[3],v2[3];      // Вектор 1 (x,y,z) & Вектор 2 (x,y,z)
 static const int x = 0  // Определение X-координаты
 static const int y = 1  // Определение Y-координаты
 static const int z = 2; // Определение Z-координаты

 // Вычисление вектора между двумя точками вычитанием
 // x,y,z-координат одной точки из координат другой.

 // Подсчет вектора из точки 1 в точку 0
 v1[x] = v[0][x] - v[1][x];  // Vector 1.x=Vertex[0].x-Vertex[1].x
 v1[y] = v[0][y] - v[1][y];  // Vector 1.y=Vertex[0].y-Vertex[1].y
 v1[z] = v[0][z] - v[1][z];  // Vector 1.z=Vertex[0].y-Vertex[1].z
 // Подсчет вектора из точки 2 в точку 1
 v2[x] = v[1][x] - v[2][x];  // Vector 2.x=Vertex[0].x-Vertex[1].x
 v2[y] = v[1][y] - v[2][y];  // Vector 2.y=Vertex[0].y-Vertex[1].y
 v2[z] = v[1][z] - v[2][z];  // Vector 2.z=Vertex[0].z-Vertex[1].z
 // Вычисление векторного произведения
 out[x] = v1[y]*v2[z] - v1[z]*v2[y];  // для Y - Z
 out[y] = v1[z]*v2[x] - v1[x]*v2[z];  // для X - Z
 out[z] = v1[x]*v2[y] - v1[y]*v2[x];  // для X - Y
 ReduceToUnit(out);      // Нормализация векторов
}
```

Следующая функция задает точку наблюдения с использованием функции gluLookAt. Мы разместим эту точку в точке (0,5,50), она будет направлена на точку (0,0,0)(центр сцены), при этом верхний вектор будет задан с направлением вверх (0,1,0)! ;D

```
void ProcessHelix()      // Рисование спирали (или пружины)
{
 GLfloat x;      // x-координата спирали
 GLfloat y;      // y-координата спирали
 GLfloat z;      // z-координата спирали
 GLfloat phi;    // Угол
 GLfloat theta;  // Угол
 GLfloat v,u;    // Углы
 GLfloat r;      // Радиус скручивания
 int twists = 5; // пять витков
 // Задание цвета материала
 GLfloat glfMaterialColor[]={0.4f,0.2f,0.8f,1.0f};
 // Настройка рассеянного освещения
 GLfloat specular[]={1.0f,1.0f,1.0f,1.0f};
 glLoadIdentity();    // Сброс матрицы модели
 // Точка камеры (0,5,50) Центр сцены (0,0,0)
```

```
// Верх но оси Y
gluLookAt(0, 5, 50, 0, 0, 0, 0, 1, 0);
glPushMatrix();      // Сохранение матрицы модели
// Смещение позиции вывода на 50 единиц вглубь экрана
glTranslatef(0,0,-50);
glRotatef(angle/2.0f,1,0,0);  // Поворот на angle/2 относительно X
glRotatef(angle/3.0f,0,1,0);  // Поворот на angle/3 относительно Y
glMaterialfv(GL_FRONT_AND_BACK,
   GL_AMBIENT_AND_DIFFUSE, glfMaterialColor);
glMaterialfv(GL_FRONT_AND_BACK,GL_SPECULAR,specular);
```

Далее займемся подсчетом формулы спирали и визуализацией пружины. Это совсем несложно, и я не хотел бы останавливаться на этих пунктах, так как «соль» данного урока совсем в другом. Участок программы, отвечающий за спираль, заимствован у друзей из «Listen Software» (и несколько оптимизирован). Он написан наиболее простым и не самым быстрым способом. Использование массивов вершин не делает его быстрее.

```
r=1.5f;        // Радиус
glBegin(GL_QUADS);    // Начать рисовать четырехугольник
for(phi=0; phi <= 360; phi+=20.0)  // 360 градусов шагами по 20
{// 360 градусов * количество_витков шагами по 20
 for(theta=0; theta<=360*twists; theta+=20.0)
  {// Подсчет угла первой точки  (0)
  v=(phi/180.0f*3.142f);
  // Подсчет угла первой точки  (0)
  u=(theta/180.0f*3.142f);
  // Подсчет x-позиции (первая точка)
  x=float(cos(u)*(2.0f+cos(v) ))*r;
  // Подсчет y-позиции (первая точка)
  y=float(sin(u)*(2.0f+cos(v) ))*r;
  // Подсчет z-позиции (первая точка)
  z=float((( u-(2.0f*3.142f)) + sin(v) ) * r);
  vertexes[0][0]=x;  // x первой вершины
  vertexes[0][1]=y;  // y первой вершины
  vertexes[0][2]=z;  // z первой вершины
  // Подсчет угла второй точки (0)
  v=(phi/180.0f*3.142f);
  // Подсчет угла второй точки (20)
  u=((theta+20)/180.0f*3.142f);
  // Подсчет x-позиции (вторая точка)
  x=float(cos(u)*(2.0f+cos(v) ))*r;
  // Подсчет y-позиции (вторая точка)
  y=float(sin(u)*(2.0f+cos(v) ))*r;
  // Подсчет z-позиции (вторая точка)
  z=float((( u-(2.0f*3.142f)) + sin(v) ) * r);
  vertexes[1][0]=x;  // x второй вершины
  vertexes[1][1]=y;  // y второй вершины
  vertexes[1][2]=z;  // z второй вершины
  // Подсчет угла третьей точки  ( 20 )
  v=((phi+20)/180.0f*3.142f);
  // Подсчет угла третьей точки  ( 20 )
  u=((theta+20)/180.0f*3.142f);
  // Подсчет x-позиции (третья точка)
  x=float(cos(u)*(2.0f+cos(v) ))*r;
  // Подсчет y-позиции (третья точка)
  y=float(sin(u)*(2.0f+cos(v) ))*r;
  // Подсчет z-позиции (третья точка)
  z=float((( u-(2.0f*3.142f)) + sin(v) ) * r);
  vertexes[2][0]=x;  // x третьей вершины
  vertexes[2][1]=y;  // y третьей вершины
  vertexes[2][2]=z;  // z третьей вершины
  // Подсчет угла четвертой точки (20)
  v=((phi+20)/180.0f*3.142f);
```

```
    // Подсчет угла четвертой точки (0)
    u=((theta)/180.0f*3.142f);
    // Подсчет x-позиции (четвертая точка)
    x=float(cos(u)*(2.0f+cos(v) ))*r;
    // Подсчет y-позиции (четвертая точка)
    y=float(sin(u)*(2.0f+cos(v) ))*r;
    // Подсчет z-позиции (четвертая точка)
    z=float((( u-(2.0f*3.142f)) + sin(v) ) * r);
    vertexes[3][0]=x;  // x четвертой вершины
    vertexes[3][1]=y;  // y четвертой вершины
    vertexes[3][2]=z;  // z четвертой вершины
    // Вычисление нормали четырехугольника
    calcNormal(vertexes,normal);
    // Установка нормали
    glNormal3f(normal[0],normal[1],normal[2]);
    // Визуализация четырехугольника
  glVertex3f(vertexes[0][0],vertexes[0][1],vertexes[0][2]);
  glVertex3f(vertexes[1][0],vertexes[1][1],vertexes[1][2]);
  glVertex3f(vertexes[2][0],vertexes[2][1],vertexes[2][2]);
  glVertex3f(vertexes[3][0],vertexes[3][1],vertexes[3][2]);
  }
  }
  glEnd();       // Конец визуализации четырехугольника
  glPopMatrix(); // Восстанавливаем матрицу
}
```

Две функции (ViewOrtho и ViewPerspective) написаны для упрощения рисования в ортогональной проекции и возврата перспективную.

Функция ViewOrtho выбирает текущую матрицу проекции и сохраняет ее копию в стеке системы OpenGL. Затем в матрицу проекции грузится единичная матрица, и устанавливается ортогональный просмотр при текущем разрешении экрана.

После этого мы получим возможность рисовать в 2D-координатах от 0,0 в верхнем левом углу и 640,480 нижнем правом углу экрана.

И в конце, делается активной матрица модели для визуализации.

Функция ViewPerspective выбирает текущую матрицу проекции и восстанавливает из стека «неортогональную» матрицу проекции, которая была сохранена функцией ViewOrtho. Потом также выбирается с матрица модели, чтобы мы могли заняться визуализацией.

Советую применять эти две процедуры, позволяющие легко переключаться между 2D и 3D-рисованием и не волноваться об искажениях матрицы проекции и матрицы модели.

```
void ViewOrtho()      // Установка ортогонального вида
{
  glMatrixMode(GL_PROJECTION);  // Выбор матрицы проекции
  glPushMatrix();      // Сохранить матрицу
  glLoadIdentity();    // Сбросить матрицу
  glOrtho( 0, 640 , 480 , 0, -1, 1 );  // Ортогональный режим (640x480)
  glMatrixMode(GL_MODELVIEW);  // Выбор матрицы модели
  glPushMatrix();      // Сохранить матрицу
  glLoadIdentity();    // Сбросить матрицу
}

void ViewPerspective()          // Установка вида перспективы
{
  glMatrixMode( GL_PROJECTION ); // Выбор матрицы проекции
  glPopMatrix();                 // Восстановить матрицу
  glMatrixMode( GL_MODELVIEW );  // Выбрать матрицу вида
  glPopMatrix();                 // Восстановить матрицу
}
```

Ну а теперь будем учиться «подделывать» эффект размытия.

Нам требуется нарисовать сцену размытой от центра во всех направлениях. Способ рисования не должен понизить быстродействие. Непосредственно считывать и записывать пиксели мы не можем, и если мы хотим сохранить совместимость с большинством видеокарт, различные специфические возможности видеокарт также неприменимы.

Так неужели все понапрасну …?

Ну нет, решение совсем близко. OpenGL дает нам возможность «размытия» текстур. Конечно, это не реальное размытие, просто в процессе масштабирования текстуры выполняется линейная фильтрация ее изображения, и, если напрячь воображение, результат будет похож на «размытие Гаусса».

Так что же произойдет, если мы поместим много растянутых текстур прямо сверху на 3D-сцену и затем промасштабируем их?

Ответ вы уже знаете …– радиальное размытие!

Здесь у нас три проблемы: как в реальном времени мы будем создавать текстуры и как мы будем точно совмещать 3D-объект и текстуру?

Решение проще, чем вы себе представляете!

Проблема первая: текстурный рендеринг.

Она просто решается форматом пикселов фонового буфера. Визуализация текстуры (да и визуализация вообще) без использования фонового буфера очень неприятна для созерцания!

Визуализация текстуры производится всего одной функцией! Нам надо нарисовать наш объект и затем скопировать результат (ПЕРЕД ТЕМ КАК ПЕРЕКЛЮЧИТЬ РАБОЧИЙ И ФОНОВЫЙ БУФЕР в текстуру с использованием функции glCopytexSubImage.

Проблема вторая: подгонка текстуры точно на передней стороне 3D-объекта.

Нам известно, что если мы поменяем параметры области просмотра без установки правильной перспективы, у нас получится растянутая визуализация наших объектов. Например, если мы установим слишком широкую область просмотра (экран), визуализация будет растянута по вертикали.

Решением этой проблемы будет во-первых то, что надо установить режим вида OpenGL, равной размеру нашей текстуры (128x128). После визуализации нашего объекта в изображение текстуры мы визуализируем эту текстуру в текущем разрешении OpenGL-экрана. Таким образом, OpenGL сначала переносит уменьшенную копию объекта в текстуру, затем растягивает ее на весь экран. Вывод текстуры происходит поверх всего экрана и поверх нашего 3D-объекта в том числе. Надеюсь, я ничего не забыл. Еще небольшое уточнение… Если вы возьмете содержимое экрана размером 640x480 и затем ужмете его до рисунка в 256x256 пикселей, то его также можно будет использовать в качестве текстуры экрана и растянуть на 640x480 пикселей. Качество окажется, скорее всего, не очень, но смотреться будет как исходное 640x480 изображение.

Забавно! Эта функция в самом деле совсем несложная и является одним из моих самых любимых «дизайнерских трюков». Она устанавливает размер области просмотра (или внутреннего окна) OpenGL равным размеру нашей текстуры (128x128), или нашего массива BlurTexture, в котором эта текстура хранится. Затем вызывается функция, рисующая спираль (наш 3D-объект). Спираль будет рисоваться в окне в 128x128 и поэтому будет иметь соответствующие размеры.

После того, как в 128x128-области визуализируется спираль, мы подключаем BlurTexture и копируем буфера цвета из области просмотра при помощи функции glCopyTexImage2D.

Параметры определяются следующим образом:

Слово GL_TEXTURE_2D показывает, что мы используем двумерную текстуру. 0 - уровень мип-мапа, с которым мы совершаем копирование буфера, это значение по умолчанию. Слово GL_LUMINANCE определяет формат копируемых данных. Я использовал GL_LUMINANCE, поскольку с этим результат гораздо красивее, это значение позволяет копировать в текстуру лишь светящуюся часть буфера. Тут же могут быть использованы значения GL_ALPHA, GL_RGB, GL_INTENSITY и так далее.

Следующие два параметра указывают OpenGL на угол экрана, с которого начнется копирование данных (0,0). Далее идут ширина и высота копируемого прямоугольника экрана. И последний параметр необходим, если мы хотим рамку, которая нам не нужна.

Теперь, когда у нас есть копия буфера цвета (с уменьшенной спиралью) в массиве BlurTexture, мы можем очистить буфер и вернуть области просмотра ее естественные размеры (640x480 – во весь экран).

ВАЖНОЕ ЗАМЕЧАНИЕ:

Этот трюк возможен лишь при поддержке двойной буферизации форматом пикселя. Причиной данного условия является то, что все эти подготовительные процедуры скрыты от пользователя, поскольку полностью совершаются в фоновом буфере.

```
void RenderToTexture()    // Визуализация в текстуру
{
  // Изменить область просмотра (в соответствии с размером текстуры)
  glViewport(0,0,128,128);
  ProcessHelix();    // Нарисовать спираль
  glBindTexture(GL_TEXTURE_2D,BlurTexture); // Подключить нашу текстуру
  // Копирование области просмотра в текстуру (от 0,0 до 128,128... без рамки)
  glCopyTexImage2D(GL_TEXTURE_2D, 0, GL_LUMINANCE, 0, 0, 128, 128, 0);
  glClearColor(0.0f, 0.0f, 0.5f, 0.5); // Цвет фона
  // Очистка экрана и фоновом буфера
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glViewport(0 , 0,640 ,480);// Область просмотра = (0,0 - 640x480)
}
```

Функция просто рисует несколько смешенных полигонов-прямоугольников на переднем плане нашей 3D-сцены, с использованием текстуры BlurTexture, которую мы уже подготовили. Путем постепенного изменения прозрачности (альфа) полигона и масштабирования текстуры у нас получится нечто, похожее на радиальное размытие.

В первую очередь я отключаю флаги GEN_S и GEN_T (я фанатею по сферическому наложению, и мои программы обычно их включают :P).

Включаем 2D-текстурирование, отключаем проверку глубины, выбираем соответствующую функцию смешивания, включаем смешивание и подключаем BlurTexture.

Следующее, что мы делаем, это переключаемся на ортогональный вид сцены, что упрощает рисование полигонов, которые точно совпадают с размерами экрана. Это выстроит текстуры поверх 3D-объекта (с постепенным растяжением текстуры до размеров экрана). Таким образом, проблема два решена. Два зайца одной выстрелом! (первый «заяц» - рисование полигона в нужном месте экрана, второй «заяц» - вывод текстуры также в требуемом месте над «размываемой» спиралью, «выстрел» - переход на ортогональный вид, дающий возможность работать с обычными 2D-координатами – прим. перев.).

```
void DrawBlur(int times, float inc)  // вывод размытого изображения
{
  float spost = 0.0f;    // Начальное смещение координат
  float alphainc = 0.9f / times;  // Скорость уменьшения прозрачности
  float alpha = 0.2f;    // Начальное значение прозрачности
  // Отключить автоопределение координат текстуры
  glDisable(GL_TEXTURE_GEN_S);
  glDisable(GL_TEXTURE_GEN_T);
  glEnable(GL_TEXTURE_2D);  // Включить наложение 2D-текстур
  glDisable(GL_DEPTH_TEST); // Отключение проверки глубины
  glBlendFunc(GL_SRC_ALPHA,GL_ONE);// Выбор режима смешивание
  glEnable(GL_BLEND);    // Разрешить смешивание
  glBindTexture(GL_TEXTURE_2D,BlurTexture);// Подключить текстуру размытия
  ViewOrtho();      // переключение на ортогональный вид
  alphainc = alpha / times;  // alphainc=0.2f / число_раз визуализации размытия
```

Мы много раз выводим рисунок текстуры для создания эффекта размытия, масштабируя его изменением координат текстуры, и тем самым увеличивая степень размытия. Всего рисуется 25 прямоугольников с текстурой, растягиваемой на 1.015 за каждый проход цикла.

```
 glBegin(GL_QUADS);    // Рисуем прямоугольники
 for (int num = 0;num < times;num++)// Количество походов = times
 {
  // Установить значение alpha (начальное = 0.2)
  glColor4f(1.0f, 1.0f, 1.0f, alpha);
  glTexCoord2f(0+spost,1-spost);  // Координаты текстуры  (0,1)
  glVertex2f(0,0);    // Первая вершина(0,0)
  glTexCoord2f(0+spost,0+spost);  // Координаты текстуры  (0,0)
  glVertex2f(0,480);  // Вторая вершина(0,480)
  glTexCoord2f(1-spost,0+spost);  // Координаты текстуры  (1,0)
  glVertex2f(640,480);// Третья вершина (640,480)
  glTexCoord2f(1-spost,1-spost);  // Координаты текстуры  (1,1)
  glVertex2f(640,0);  // Четвертая вершина (640,0)
  // Увеличение spost (Приближение к центру текстуры)
  spost += inc;
  // Уменьшение alpha (постепенное затухание рисунка)
  alpha = alpha - alphainc;
 }
 glEnd();        // Конец рисования
 ViewPerspective();        // Назад к перспективному виду
 glEnable(GL_DEPTH_TEST);  // Включить проверку глубины
 glDisable(GL_TEXTURE_2D); // Отключить 2D-текстурирование
 glDisable(GL_BLEND);      // Отключить смешивание
glBindTexture(GL_TEXTURE_2D,0); // Отвязать текстуру
}
```

И вуаля: самая короткая функция Draw, когда-либо виданная, дающая возможность увидеть превосходный визуальный эффект.

Вызывается функция RenderToTexture. Здесь один раз рисуется растянутая пружина маленького размера, благодаря изменению параметров области вывода. Растянутая пружина копируется в нашу текстуру, затем буфер очищается.

Затем мы рисуем «настоящую» пружину (трехмерный объект, который вы видите на экране) при помощи функции ProcessHelix().

Наконец, мы рисуем последовательность «смешено-размазанных» полигонов-прямоугольников спереди пружины. То есть текстурированные прямоугольники будут растянуты и размазаны по изображению реальной 3D-пружины.

```
void Draw (void)      // Визуализация 3D-сцены
{
 glClearColor(0.0f, 0.0f, 0.0f, 0.5);// Очистка черным цветом
 // Очистка экрана и фонового буфера
 glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
 glLoadIdentity();   // Сброс вида
 RenderToTexture();  // Визуализация в текстуру
 ProcessHelix();     // Рисование спирали
 DrawBlur(25,0.02f); // Эффект размытия
 glFlush ();         // «Сброс» конвейера OpenGL
}
```

Надеюсь, данное руководство вам понравилось, хотя оно больше ничему и не учит, кроме визуализации в текстуру, но определенно может добавить красивый эффект в ваши 3D-программы.

Со всеми комментариями и предложениями по улучшению реализации этого эффекта прошу слать письма на rio@spinningkids.org. (а если вы хотите получить исправленную и улучшенную версию программы, пишите на lake@tut.by, будет интересно пообщаться :) - прим. перев.).

Вы вольны использовать этот код бесплатно, тем не менее, если вы собираетесь использовать его в своих разработках, отметьте это и попытайтесь понять, как он работает, только так и никак иначе. Кроме того, если вы будете использовать этот код в своих коммерческих разработках (разработки, за которые вы потребуете плату), пожалуйста, выделите мне некоторый кредит.

И еще я хочу оставить вам всем небольшой список заданий для практики (домашнее задание) :D

1) Переделайте функцию DrawBlur для получения горизонтального размытия, вертикального размытия и каких-нибудь других интересных эффектов (круговое размытие, например).

2) Поменяйте параметры функции DrawBlur (увеличьте или уменьшите), чтобы получился красивый световой эффект под вашу музыку.

3) Поиграйте с функцией DrawBlur, нашей небольшой текстурой и значением GL_LUMINANCE (крутой блеск!).

4) Симитируйте объемные тени при помощи темных текстур взамен светящихся.

Ок. Теперь, должно быть, все.

Посетите мой сайт на http://www.spinningkids.org/rio, где есть много подобных статей, и со временем будет больше.

Урок 37. Мультипликационное закрашивание.

Cel-Shading

Наблюдая, как люди всё ещё пишут мне, спрашивая об исходном коде для статьи, которую я недавно написал на GameDev.net и, понимая, что второй вариант этой статьи (с исходными кодами для каждого API) далёк от того, чтобы быть законченным даже наполовину, я сварганил этот урок для NeHe (который фактически должен бы был лечь в основу статьи), что бы вы все, гуру OpenGL, могли поэкспериментировать с ним. Извините за выбор модели, но недавно я много играл в Quake II…

Примечание: оригинал статьи для этого кода можно найти на:

http://www.gamedev.net/reference/programming/features/celshading

В этом уроке на самом деле нет объяснения теории, только код. Ответ на вопрос, ПОЧЕМУ это работает, можно найти по ссылке, указанной выше. Теперь, чёрт возьми, ПРЕКРАТИТЕ ПИСАТЬ И ПРОСИТЬ ИСХОДНЫЙ КОД!!!!

Наслаждайтесь!

Прежде всего, нам необходимо подключить несколько дополнительных заголовочных файлов. Первый из них (math.h) для использования функции sqrtf (квадратный корень), а второй (stdio.h) для обеспечения доступа к файлам.

```
#include <math.h>      // Заголовочный файл математической библиотеки
#include <stdio.h>     // Заголовочный файл для стандартной библиотеки ввода\вывода
```

Теперь мы собираемся определить несколько структур для хранения данных (для сохранения сотен массивов вещественных чисел). Первая – структура tagMATRIX. Если вы внимательно посмотрите, то обнаружите, что мы сохраняем матрицу как одномерный массив из 16 вещественных чисел, а не двумерный размерностью 4x4. Это соответствует тому, как в OpenGL хранятся матрицы. Если бы мы использовали массив 4x4, то значения располагались бы в неправильном порядке.

```
typedef struct tagMATRIX    // Структура для хранения матрицы в формате OpenGL
{
  float Data[16];           // Мы используем размерность 16 из-за формата матрицы OpenGL
} MATRIX;
```

Вторая структура – векторный класс. Она просто хранит значения для X, Y и Z.

```
typedef struct tagVECTOR    // Структура для хранения вектора
{
  float X, Y, Z;            // Компоненты вектора
} VECTOR;
```

Третья – структура вершины. Для каждой вершины необходима только её нормаль и позиция (без координат текстуры). Они ДОЛЖНЫ храниться только в этом порядке, иначе процесс загрузки данных из файла пройдет не корректно (я затратил много сил, чтобы понять это, возможно, этот опыт научит меня разбивать код на части).

```
typedef struct tagVERTEX    // Структура для хранения одной вершины
{
  VECTOR Nor;       // Нормаль вершины
  VECTOR Pos;       // Позиция вершины
} VERTEX;
```

Наконец, структура многоугольника. Я знаю, что так глупо хранить вершины, но это просто и работает. Вообще, я использовал бы массив вершин, массив многоугольников и помещал бы номера индексов трех вершин в структуру многоугольника, но так проще показать вам, что происходит.

```
typedef struct tagPOLYGON    // Структура для хранения многоугольника
{
  VERTEX Verts[3];          // Массив из 3-х структур VERTEX
} POLYGON;
```

Далее тоже довольно простой материал. Смотрите комментарии, объясняющие назначение каждой переменной.

```
bool   outlineDraw  = true;        // Флаг рисования контура
bool   outlineSmooth = false;      // Флаг сглаживания линий
float  outlineColor[3] = { 0.0f, 0.0f, 0.0f }; // Цвет линий
float  outlineWidth = 3.0f;        // Ширина линий
VECTOR lightAngle;                 // Направление света
bool   lightRotate = false;        // Флаг поворота источника света
float  modelAngle  = 0.0f;         // Угол наклона модели по оси Y
bool   modelRotate = false;        // Флаг поворота модели
POLYGON *polyData  = NULL;         // Информация о многоугольнике
int    polyNum     = 0;            // Количество многоугольников
GLuint shaderTexture[1];           // Хранилище для одной текстуры
```

Далее идет функция чтения модели. Формат хранения модели крайне прост. В первых нескольких байтах хранится число многоугольников в сцене, а остальная часть файла – массив структур tagPOLYGON. Благодаря этому, при считывании данных нет необходимости сортировать их в какой-либо особой последовательности.

```
BOOL ReadMesh ()        // Чтение содержимого файла «model.txt»
{
  FILE *In = fopen ("Data\\model.txt", "rb");  // Открытие файла
  if (!In)
    return FALSE;        // FALSE, если файл не открыт
  fread (&polyNum, sizeof (int), 1, In);  // Считывание заголовка (т.е. кол-во многоугольников)
  polyData = new POLYGON [polyNum];       // Резервирование памяти
  fread (&polyData[0], sizeof(POLYGON)*polyNum, 1, In); // Чтение данных о всех многоугольниках
  fclose (In);             // Закрытие файла
  return TRUE;             // Файл отработан
}
```

Теперь несколько основных математических функций. Функция DotProduct рассчитывает угол между двумя векторами или плоскостями, функция Magnitude рассчитывает длину вектора, а функция Normalize уменьшает вектор до единичной длины.

```
inline float DotProduct (VECTOR &V1, VECTOR &V2)  // Вычисление угла между двумя векторами
{
 return V1.X * V2.X + V1.Y * V2.Y + V1.Z * V2.Z; // Возвращает угол
}

inline float Magnitude (VECTOR &V)      // Вычисление длины вектора
{
 return sqrtf (V.X * V.X + V.Y * V.Y + V.Z * V.Z); // Возвращает длину вектора
}

void Normalize (VECTOR &V)        // Создаёт вектор единичной длины
{
 float M = Magnitude (V);        // Вычисление длины вектора

 if (M != 0.0f)          // Удостовериться, что нет деления на 0
 {
  V.X /= M;             // Нормализация трёх составляющих
  V.Y /= M;
  V.Z /= M;
 }
}
```

Эта функция производит вращение вектора, используя предоставляемую ей матрицу. Пожалуйста, обратите внимание, что она ТОЛЬКО вращает вектор – что не имеет никакого отношения к расположению вектора. Это используется при вращении нормалей, когда необходимо удостовериться, что они указывают в правом направлении при расчете освещения.

```
void RotateVector (MATRIX &M, VECTOR &V, VECTOR &D) // Вращение вектора с использованием матрицы
{
 D.X =(M.Data[0] * V.X) + (M.Data[4] * V.Y) + (M.Data[8]  * V.Z);  // Поворот вокруг оси X
 D.Y =(M.Data[1] * V.X) + (M.Data[5] * V.Y) + (M.Data[9]  * V.Z);  // Поворот вокруг оси Y
 D.Z =(M.Data[2] * V.X) + (M.Data[6] * V.Y) + (M.Data[10] * V.Z);  // Поворот вокруг оси Z
}
```

Первая главная функция движка… Initialise, выполняет то, что означает. Я вырезал несколько строк кода, т.к. объяснять их нет необходимости.

```
//Здесь располагается любой код инициализации графической библиотеки и пользователя
BOOL Initialize (GL_Window* window, Keys* keys)
{
```

Следующие три переменные используются при загрузки файла с данными о закрашивании (shader-файл). Line используется для чтения строки из текстового файла, в то время как shaderData хранит действительные параметры закрашивания. Вам может быть интересно, почему у нас 96 значений вместо 32. Ну, необходимо конвертировать значения шкалы оттенков серого цвета (greyscale) в RGB, чтобы OpenGL мог их использовать. Можно по-прежнему хранить значения как оттенки серого цвета, но дальше мы будем использовать значения для компонентов RGB при загрузке текстуры.

```
 char  Line[255];    // Хранилище для 255 символов
 float shaderData[32][3];  // Хранилище для 96 значений тени
 FILE *In = NULL;    // Указатель на файл
```

При прорисовке линий, мы хотим удостовериться, что они точные и гладкие. Первоначально этот режим отключён, но, нажимая клавишу «2», можно переключаться между режимами вкл\выкл.

```
 glShadeModel (GL_SMOOTH);   // Включение плавного цветового закрашивания
 glDisable (GL_LINE_SMOOTH); // Первоначальное отключение сглаживания линий
 glEnable (GL_CULL_FACE);    // Разрешить удаление задних граней
```

Мы отключаем освещение OpenGL, потому что мы делаем все вычисления освещения самостоятельно.

```
 glDisable (GL_LIGHTING);  // Отключение освещения OpenGL
```

Здесь мы загружаем файл закрашивания. Это просто 32 числа с плавающей запятой, сохранённые в кодировке ASCII (для простой модификации), каждое на отдельной строке.

```
  In = fopen ("Data\\shader.txt", "r");  // Открытие файла закрашивания
  if (In) // Проверка на открытие файла
  {
    for (i = 0; i < 32; i++) // Цикл по 32 значениям шкалы оттенков серого цвета
    {
      if (feof (In))         // Проверка на конец файла
        break;
      fgets (Line, 255, In); // Текущая строка
```

Здесь мы конвертируем значения серого цвета в RGB, как описано выше.

```
// Копирование значений
      shaderData[i][0] = shaderData[i][1] = shaderData[i][2] = atof (Line);
    }
    fclose (In);        // Закрытие файла
  }
  else
    return FALSE;       // Ужасно-ужасно неверно
```

Теперь мы загружаем текстуру. Как ясно указывается, не используйте никакой фильтрации на текстуре, иначе она будет выглядеть, мягко говоря, странно. Используется GL_TEXTURE_1D, потому что это одномерный массив значений.

```
  glGenTextures (1, &shaderTexture[0]); // Получение свободного ID текстуры
  // Связывание с текстурой. В дальнейшем это будет ID текстуры.
  glBindTexture (GL_TEXTURE_1D, shaderTexture[0]);
  // Чёрт возьми, не позволяйте OpenGL использовать двух\трёхлинейную фильтрацию!
  glTexParameteri (GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
  glTexParameteri (GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
  // Подгрузка
  glTexImage1D (GL_TEXTURE_1D, 0, GL_RGB, 32, 0, GL_RGB , GL_FLOAT, shaderData);
```

Теперь установите направление освещения. Я установил его вдоль оси Z, что значит – свет будет падать на лицевую сторону модели.

```
  lightAngle.X = 0.0f;    // Направление вдоль оси X
  lightAngle.Y = 0.0f;    // Направление вдоль оси Y
  lightAngle.Z = 1.0f;    // Направление вдоль оси Z
  Normalize (lightAngle); // Нормализация направления света
```

Загрузите набор граней из файла (описано выше).

```
  return ReadMesh (); // Возвращает значение функции ReadMesh()
}
```

Функция, обратная описанной выше… Deinitialize, удаляет текстуру и данные многоугольника, созданные функциями Initalize и ReadMesh.

```
void Deinitialize (void)  // Здесь может располагаться любой код деинициализации пользователя
{
  glDeleteTextures (1, &shaderTexture[0]); // Удаление текстуры
  delete [] polyData;     // Удаление данных многоугольника
}
```

Главный демонстрационный цикл. Все что он делает – это обрабатывает ввод и модифицирует угол. Средство управления следующие:

<SPACE> = Переключатель вращения
1 = Переключатель прорисовки контура
2 = Переключатель сглаживания контура
<UP> = Увеличение ширины линии
<DOWN> = Уменьшение ширины линии

```
void Update (DWORD milliseconds)     // Здесь производится модернизация движения
{
 if (g_keys->keyDown [' '] == TRUE) // Нажата клавиша <SPACE>?
 {
  modelRotate = !modelRotate;      // Переключение вращения модели вкл\выкл
  g_keys->keyDown [' '] = FALSE;
 }

 if (g_keys->keyDown ['1'] == TRUE) // Нажата клавиша «1»?
 {
  outlineDraw = !outlineDraw;      // Переключение прорисовки контура вкл\выкл
  g_keys->keyDown ['1'] = FALSE;
 }

 if (g_keys->keyDown ['2'] == TRUE) // Нажата клавиша «2»?
 {
  outlineSmooth = !outlineSmooth;  // Переключение сглаживания контура вкл\выкл
  g_keys->keyDown ['2'] = FALSE;
 }

 if (g_keys->keyDown [VK_UP] == TRUE) // Нажата стрелка «Вверх»?
 {
  outlineWidth++;                  // Увеличение ширины линии
  g_keys->keyDown [VK_UP] = FALSE;
 }

 if (g_keys->keyDown [VK_DOWN] == TRUE) // Нажата стрелка «Вниз»?
 {
  outlineWidth--;                  // Уменьшение ширины линии
  g_keys->keyDown [VK_DOWN] = FALSE;
 }

 if (modelRotate)                  // Проверка включен или нет режим вращения
  modelAngle += (float) (milliseconds) / 10.0f; // Модификация угла основанного на таймере
}
```

Функция, которую вы все ждали. Функция Draw делает всё – вычисляет значения закрашивания, отображает набор граней, отображает контуры - и это действительно так.

```
void Draw (void)
{
```

TmpShade используется для хранения значений закрашивания для текущей вершины. Вся информация о вершине рассчитывается в одно и то же время, в том смысле, что мы многократно используем одну и ту же переменную.

Структуры TmpMatrix, TmpVector и TmpNormal так же используются для расчета информации о вершине. Значение TmpMatrix устанавливается однократно при запуске функции и не меняется до следующего вызова функции Draw. С другой стороны, TmpVector и TmpNormal изменяются при расчете каждой последующей вершины.

```
  float TmpShade;    // Временное значение закрашивания
  MATRIX TmpMatrix;  // Временная структура MATRIX
  VECTOR TmpVector, TmpNormal; // Временная структура VECTOR
```

Давайте очистим буферы и матричные данные.

```
  glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Очистка буферов
  glLoadIdentity (); // Перезагрузка матрицы
```

Первая проверка – хотим ли мы иметь гладкий контур. Если так, то включаем сглаживание. Если нет – отключаем. Просто!

```
if (outlineSmooth)          // Проверка – хотим ли гладкий контур
{
  glHint (GL_LINE_SMOOTH_HINT, GL_NICEST); //Использовать качественные вычисления
  glEnable (GL_LINE_SMOOTH); // Включить сглаживание
}
else // Мы не хотим гладкий контур
glDisable (GL_LINE_SMOOTH); // Отключить сглаживание
```

Затем мы устанавливаем область просмотра. Отодвигаем камеру на две единицы назад, а затем поворачиваем модель заданный угол. Обратите внимание: модель будет вращаться на месте, т.к. мы в начале переместили камеру. Если бы мы делали наоборот, модель вращалась бы вокруг камеры.

Затем мы берём вновь созданную матрицу у OpenGL и сохраняем её в TmpMatrix.

```
glTranslatef (0.0f, 0.0f, -2.0f);          // Отодвигаемся на две единицы от экрана
glRotatef (modelAngle, 0.0f, 1.0f, 0.0f); // Вращаем модель вдоль оси Y
glGetFloatv (GL_MODELVIEW_MATRIX, TmpMatrix.Data); // Берём сгенерированную матрицу
```

Начинаются чудеса. Вначале включаем одномерное текстурирование, а затем разрешаем использование текстуры с оттенками серого цвета для закрашивания. OpenGL пользуется ей в качестве таблицей соответствия. Затем мы устанавливаем цвет модели (белый). Я выбрал белый, т.к. он даёт возможность отображать подсветки и затенения намного лучше, чем другие цвета. Предлагаю вам не использовать чёрный…

```
// Код эффекта мультипликационного закрашивания
glEnable (GL_TEXTURE_1D); // Включить одномерное текстурирование
glBindTexture (GL_TEXTURE_1D, shaderTexture[0]); // «Захват» нашей текстуры
glColor3f (1.0f, 1.0f, 1.0f); // Установка цвета модели
```

Теперь начинаем прорисовку треугольников. Мы просматриваем каждый многоугольник в массиве, а затем по очереди все его вершины. На первом шаге информация о нормали копируется во временную структуру. Таким образом, мы можем вращать нормали, но сохранить их первоначальные значения (без снижения точности).

```
glBegin (GL_TRIANGLES); // «Сказать» OpenGL, что мы рисуем треугольники
for (i = 0; i < polyNum; i++)    // Цикл по каждому многоугольнику
{
  for (j = 0; j < 3; j++)        // Цикл по каждой вершине
  {
    TmpNormal.X = polyData[i].Verts[j].Nor.X;  // Заполнение структуры TmpNormal
    TmpNormal.Y = polyData[i].Verts[j].Nor.Y;  // значениями нормали текущей вершины
    TmpNormal.Z = polyData[i].Verts[j].Nor.Z;
```

Во-вторых, мы вращаем нормаль в соответствии с матрицей ранее взятой у OpenGL. Затем мы нормализуем её, что бы она ни вела себя странно.

```
    // Вращение в соответствии с матрицей
    RotateVector (TmpMatrix, TmpNormal, TmpVector);
    Normalize (TmpVector); // Нормализация новой нормали
```

В-третьих, мы получаем скалярное произведение векторов вращаемой нормали и направления света lightAngel. Затем мы урезаем значение до диапазона 0-1 (с диапазона от -1 до +1).

```
    // Вычисление значения закрашивания
    TmpShade = DotProduct (TmpVector, lightAngle);
    if (TmpShade < 0.0f)
      TmpShade = 0.0f;  // Установка значения в 0, если TmpShade отрицательно
```

В-четвёртых, мы передаём это значение OpenGL как координаты текстуры. Текстура закрашивания ведёт себя как таблица соответствия (значение закрашивания – это индекс), которая (как я думаю) и является главной причиной того, почему была инвертирована одномерная текстура. Затем мы передаём позицию вершины OpenGL, и повторяем цикл. Повторяем. Повторяем. Думаю, вы ухватили идею.

```
        glTexCoord1f (TmpShade);    // Установка координат текстуры значением закрашивания
       // Отправка вершин
       glVertex3fv (&polyData[i].Verts[j].Pos.X);
       }
    }
  glEnd (); // Завершение рисования
  glDisable (GL_TEXTURE_1D); // Отключение одномерных текстур
```

Теперь переходим к контурам. Контур может быть определён как «край соприкосновения двух многоугольников, один из которых обращён к наблюдателю, а другой наоборот». В OpenGL это соответствует тому, когда тест глубины установлен в значение меньшее или равное текущему значению (GL_LEQUAL). Мы также смешиваем линии для лучшего отображения.

Теперь разрешаем смешивание и устанавливаем режим. Говорим OpenGL, отобразить задние многоугольники как линии, и устанавливаем ширину этих линий. Запрещаем прорисовку передних многоугольников и устанавливаем тест глубины меньшим или равным текущему значению Z. После того, как установлен цвет линий, перебираем в цикле все многоугольники, отображая их вершины. Необходимо передавать только позицию вершины, а не нормаль или значение закрашивания, т.к. всё, что нам нужно - это контур.

```
  // Код прорисовки контура
  if (outlineDraw)        // Проверка на необходимость прорисовки контура
  {
    glEnable (GL_BLEND);  // Разрешить смешивание
    // Установить режим смешивания
    glBlendFunc (GL_SRC_ALPHA ,GL_ONE_MINUS_SRC_ALPHA);
    glPolygonMode (GL_BACK, GL_LINE); // Прорисовка задних многоугольников в виде линий
    glLineWidth (outlineWidth);       // Установка ширины линии
    glCullFace (GL_FRONT);            // Запрет на прорисовку видимых многоугольников
    glDepthFunc (GL_LEQUAL);          // Изменение режима глубины
    glColor3fv (&outlineColor[0]);    // Установка цвета контура
    glBegin (GL_TRIANGLES);           // Сообщение того, что хотим нарисовать
      for (i = 0; i < polyNum; i++)   // Цикл по каждому многоугольнику
      {
        for (j = 0; j < 3; j++)       // Цикл по каждой вершине
        {
          // Передача положения вершины
          glVertex3fv (&polyData[i].Verts[j].Pos.X);
        }
      }
    glEnd (); // Сообщение о завершении прорисовки
```

После этого, возвращаем все назад, как было до этого, и выходим.

```
    glDepthFunc (GL_LESS);      // Возврат в исходное положение режима теста глубины
    glCullFace (GL_BACK);       // Возвращение режима прорисовка граней
    glPolygonMode (GL_BACK, GL_FILL); // Возвращение режима прорисовки задних граней
    glDisable (GL_BLEND); // Отключение смешивания
  }
}
```

Теперь вы видите, что мультипликационное закрашивание - это не так трудно, как кажется. Безусловно, методику можно сильно улучшить. Хороший пример – игра XIII (http://www.nvidia.com/object/game_xiii.html), которая заставляет вас думать, что вы в мультяшном мире. Если вы хотите глубже понять технику отображения мультипликационных объектов, то можете посмотреть главу "Non-Photorealistic Rendering" в книге «Real-time Rendering» (Muller, Hains). Если вам нравится читать статьи в сети, обширный список ссылок можно найти здесь: http://www.red3d.com/cwr/npr/ .

Урок 39. Введение в физический симулятор

Introduction to Physical Simulations

Если вы знакомы с физикой и хотите написать физический симулятор – этот урок поможет вам. Для того чтобы достигнуть успеха вам понадобятся знания векторных операций в 3D и базовых физических понятий, таких как сила и скорость.

В этом уроке вы найдете код очень простого движка физики. (Скорее просто пример, для движка все слишком упрощено – прим. переводчика).

Содержание (в порядке следования):

Дизайн:
 * class Vector3D ---> Объект представляющий 3D - вектор или точку в пространстве

Сила и движение:
 * class Mass ---> Объект представляющий массу.

Как работает симуляция:
 * class Simulation ---> Контейнер для симуляции масс.

Управление симуляцией из приложения:
 * class ConstantVelocity :
  public Simulation ---> Объект создающий массу с постоянной скоростью.

Применение силы:
 * class MotionUnderGravitation :
  public Simulation ---> Объект создающий массу, движущуюся под воздействием гравитации.
 * class MassConnectedWithSpring :
  public Simulation ---> Объект создающий массу, соединенную с пружиной в точке.

Дизайн:
Создание физического движка задача не всегда простая. Но есть несколько очевидных зависимостей: приложение зависит от движка, а движок в свою очередь зависит от библиотек математики. Наша задача получить контейнер для симуляции движения масс. Наш движок будет включать в себя класс «Mass» и класс «Simulation» - наш контейнер. После того как эти классы будут готовы, мы сможем писать приложения. Но, прежде всего нам понадобится математическая библиотека. Эта библиотека содержит только один класс «Vector3D», который используется для представления точек, векторов, положений, скорости и силы в 3D.

 * class Vector3D ---> Объект представляющий вектор или точку в пространстве

Vector3D – единственный класс нашей математической библиотеки. Он хранит значения x, y и z координат и реализует простейшие векторные операции: сложение, вычитание, умножение, деление. Поскольку этот урок, прежде всего о физике, я не буду вдаваться в детали, описывая этот класс. Если вы посмотрите Lesson39.h вам сразу станет ясно насколько он прост.

Сила и движение:

Для работы с физикой нам необходимо понимать, что же такое масса. Масса имеет положение и скорость. Масса имеет вес на Земле, Луне, Марсе или в любом другом месте, где существует гравитация. Вес различается в зависимости от силы гравитации. (Точнее вес это сила, с которой тело (под воздействием гравитации) действует на горизонтальную опору или подвес – прим. переводчика). Масса тела остается постоянной при любых условиях.

После того как мы поняли, что такое масса, давайте, разберемся с силой и движением. Масса с не нулевой скоростью в пространстве движется в направлении вектора скорости. Поэтому вектор скорости – одна из причин изменения положения массы в пространстве. Вторая причина – прошествие времени. Т.е. изменение положения зависит от того, насколько быстро движется масса и сколько времени прошло. Если к этому месту что-то остается не ясным, обдумайте отношения между положением, скоростью и временем. (А так же сдуйте пыль с учебника физики 6-го класса, глава начала механики – прим. переводчика).

Скорость массы изменяется, если существует сила, воздействующая на массу. Вектор скорости стремится к направлению силы. Эта тенденция пропорциональна силе и обратно пропорциональна массе. Изменение скорости за единицу времени называется ускорением. Чем больше сила, воздействующая на массу, тем больше ускорение. Чем больше масса, тем меньше ускорение.

Отсюда:
ускорение = сила / масса

Из этого, знаменитая формула:
сила = масса * ускорение

(мы будем часто использовать формулу ускорения)

До подготовки «физического носителя» для симуляции, вы должны знать, в каком окружении мы будем работать. В этом уроке наше окружение – пустое пространство. ;)
Во первых, мы должны определить в каких единицах мы будем выражать время и массу. Я решил в качестве единиц времени использовать секунды и для расстояния (положения) - метры. Отсюда единицы скорости – метры в секунду (м/с), ускорения (м/с/с или м/с2). Единицы массы – килограммы (кг).

* class Mass ---> Объект представляющий массу.

Теперь начнем применять теорию! Мы должны написать класс представляющий массу. Он должен хранить значение массы, положение, скорость, и воздействующую силу.

```
class Mass
{   public:
        float     m;        // Значение массы.
        Vector3D  pos;      // Положение в пространстве.
        Vector3D  vel;      // Скорость.
        Vector3D  force;    // Воздействующая сила.

       Mass(float m)       // Конструктор
       {
         this->m = m;
       }
       ...
```

Мы хотим воздействовать на массу силой. В единицу времени на массу может воздействовать несколько сил. Векторная сумма этих сил дает нам общее значение вектора силы в конкретную единицу времени. До того как мы начнем применять силы к массе нам необходимо обнулить силу в классе (force). После этого нам остается только добавлять силы.

```
        (class Mass продолжение)

        void applyForce(Vector3D force)
        {
           this->force += force;    // Внешнюю силу прибавляем к «нашей».
        }
        void init()          // Обнуляем «нашу» силу
        {
          force.x = 0;
          force.y = 0;
          force.z = 0;
        }

    ...
```

Итак, чтобы воздействовать силами на массу нам нужно:

1. Сбросить силу (метод init())
2. Добавить все воздействующие силы (applyForce(Vector3D))
3. Итерационно изменить время

Здесь, изменение времени организовано по методу Эйлера (Euler). Метод Эйлера очень простой. Существуют и более продвинутые методы, но для большинства приложений этого метода достаточно. Большая часть игрушек использует именно его. Этот метод вычисляет скорость и положение массы, в следующий момент времени исходя из примененных сил и прошедшего времени. Итерации организованы в методе void simulate(float dt):

```
(class Mass продолжение)

void simulate(float dt)
{
 vel += (force / m) * dt;  // Изменение в скорости добавляем к
                           // текущей скорости. Изменение
                           // пропорционально ускорению
                           // (сила/масса) и изменению времени
 pos += vel * dt;          // Изменение в положении добавляем к
                           // текущему положению. Изменение в
                           // положении Скорость*время
}
```

Как должна происходить симуляция:
В каждой итерации происходит один и тот же процесс. Силы обнуляются, добавляются все воздействующие силы, рассчитывается новая скорость и положение. Этот процесс продолжается до тех пор, пока изменяется время. Он организован в классе Simulation.

* class Simulation ---> Контейнер для симуляции масс.

(Автор выбрал не слишком удачное определение – этот класс просто организует массив и к контейнерам не имеет никакого отношения – прим. переводчика).
Класс Simulation хранит массы как свои переменные. Задача этого класса создавать/удалять массы и проводить симуляцию.

```
class Simulation
{   public:
      int   numOfMasses;   // Количество масс в контейнере.
      Mass** masses;       // Массы хранятся в 1d массиве
                           // указателей
       // Конструктор создает numOfMasses масс с массой m.
       Simulation(int numOfMasses, float m)
      {
        this->numOfMasses = numOfMasses;
        masses = new Mass*[numOfMasses]; // Создаем массив указателей.

        // Создаем Mass и заносим его в массив
        for (int a = 0; a < numOfMasses; ++a)
            masses[a] = new Mass(m);
      }
      virtual void release()     // Чистим массив масс
      {
       for (int a = 0; a < numOfMasses; ++a)
       {
         delete(masses[a]);
         masses[a] = NULL;
       }
       delete(masses);
       masses = NULL;
      }
      Mass* getMass(int index)
      {
       // Если индекс выходит за рамки массива возвращаем null
       if (index < 0 || index >= numOfMasses) return NULL;
       // Возвращаем массу по индексу
       return masses[index];
      }
      ...
```

Процедура симуляции имеет три шага:

1. init() – устанавливаем силу (Mass->force) в 0
2. solve() – применяем силы
3. simulate(float dt) – конец итерации

```
  (class Simulation продолжение)

  virtual void init()      // вызываем init() для каждой массы
  {
    for (int a = 0; a < numOfMasses; ++a)
        masses[a]->init();
  }
  virtual void solve()
  {
   // Нет кода т.к. в базовом классе у нас нет сил
   // В других контейнерах мы переопределим этот метод
  }
  virtual void simulate(float dt)  // Итерация для каждой массы
  {
   for (int a = 0; a < numOfMasses; ++a)
       masses[a]->simulate(dt);
  }
  ...
```

Объединим процедуру симуляции в один метод:

```
  (class Simulation продолжение)

  virtual void operate(float dt) // Полная процедура симуляции.
  {
    init();           // 1. Силу в 0
    solve();          // 2. Применяем силы
    simulate(dt);     // 3. Итерация
  }
};
```

Теперь у нас есть простейший движок физики. Он базируется на библиотеке математики и состоит из классов Mass и Simulation. Теперь мы готовы приступить к разработке приложений (реальных применений). Приложения, которые мы обсудим:

1. Масса с постоянной скоростью
2. Масса в условиях гравитации
3. Масса, соединенная с точкой пружиной

Управление симуляцией из приложения:

До того как мы преступим к написанию специфических вариантов симуляции, нам необходимо узнать, как же мы будем управлять «процессом». В этом примере движок и приложение управляющее им, находятся в разных файлах. В файле приложения находится функция:

```
void Update (DWORD milliseconds)    // обновление движения.
```

Эта функция вызывается при обновлении каждого кадра. "DWORD milliseconds" время прошедшее с момента предыдущего обновления кадра. Здесь нужно сказать, что приращение времени мы будем получать в миллисекундах, в этом случае процесс симуляции будет идти параллельно с реальным временем. Чтобы перейти к следующему шагу симуляции, мы вызываем метод void operate(float dt), для этого нам необходимо знать значение dt. Поскольку этот метод получает время в секундах, мы должны сделать соответствующие поправки (см. код ниже). Затем мы используем slowMotionRatio – эта переменная определяет насколько медленно идет время по отношению к реальному. Мы делим dt на это значение и получаем новое dt. Теперь мы можем прибавить dt к timeElapsed. "timeElapsed" – время нашей симуляции.

```
void Update (DWORD milliseconds)
{
  ...
  ...
  ...
   float dt = milliseconds / 1000.0f; // Преобразуем миллисекунды в секунды
  dt /= slowMotionRatio;             // Делим на slowMotionRatio
  timeElapsed += dt;                 // Изменяем кол-во прошедшего времени
  ...
```

Теперь dt практически готово, но есть еще один важный момент – dt влияет на точность симуляции. Если dt не достаточно мало, симуляция будет не стабильной, и движение будет рассчитываться не точно. Для того чтобы выяснить максимальное допустимое значение dt используется анализ стабильности. В этом уроке мы не будем останавливаться на таких подробностях – для серьезных научных расчетов это необходимо, но для большинства игр достаточно подобрать значение «на глаз».

Например, в авто-симуляторе оправдано использовать dt от 2 до 5 миллисекунд для обычной машины и от 1 до 3 для гоночной. Для аркадных симуляторов значение dt может колебаться 10 до 200 мс. Чем меньше значение dt, тем больше процессорного времени потребуется для обсчетов. Именно поэтому физические движки так редко использовались в старых играх. (На самом деле все притянуто за уши, реальные причины отсутствия нормальной физики в старых играх несколько глубже – прим. переводчика).

В следующем коде мы определяем максимальное возможное значение dt 0.1с (100мс). С помощью этого значения мы получим количество итераций к моменту текущего обновления. Запишем формулу:

```
int numOfIterations = (int)(dt / maxPossible_dt) + 1;
```

numOfIterations это число итераций, которые необходимо выполнить. Скажем прога работает со скоростью 20fps, которые дают dt=0.05 с. Количество итераций (numOfIterations) будет 1. (т.е 1 итерация в 0.05с.) Если бы dt было 0.12 с. - numOfIterations было бы 2. Сразу после "int numOfIterations = (int)(dt / maxPossible_dt) + 1;", dt рассчитывается еще раз, делим его на numOfIterations и получаем dt = 0.12 / 2 = 0.06. dt было больше максимально допустимого значения (0.1с). Теперь мы имеем dt = 0.06, но поскольку итерации 2 в результате получим 0.12 с. Изучите следующий кусок кода и удостоверьтесь, что все поняли.

```
  ...
  float maxPossible_dt = 0.1f; // максимально возможное dt = 0.1 с
                               // Необходимо чтобы мы не «вылетели» за
                               // пределы точности.

  // Рассчитываем количество итераций, которые необходимо провести в ходе текущего
  // обновления(зависит от maxPossible_dt и dt).

  int numOfIterations = (int)(dt / maxPossible_dt) + 1;
  if (numOfIterations != 0)      // Проверяем деление на 0
     dt = dt / numOfIterations; // dt нужно обновить, опираясь на
                                // numOfIterations.
  // мы должны повторить симуляцию "numOfIterations" раз.
  for (int a = 0; a < numOfIterations; ++a)
  {
    constantVelocity->operate(dt);
    motionUnderGravitation->operate(dt);
    massConnectedWithSpring->operate(dt);
  }
}
```

Начнем писать приложения:
1. Масса с постоянной скоростью

* class ConstantVelocity :
 public Simulation ---> Объект создающий массу с постоянной скоростью.

Масса с постоянной скоростью не нуждается, в каких либо внешних силах. Нам всего лишь нужно создать 1 массу и установить ее скорость в (1.0f, 0.0f, 0.0f), т.е. она будет двигаться вдоль оси x со скоростью 1 м/с. Мы напишем наследника от класса Simulation – класс ConstantVelocity:

```
class ConstantVelocity : public Simulation
{
public:
        // Конструктор сначала создает предка с 1й массой в 1 кг.
        ConstantVelocity() : Simulation(1, 1.0f)
        {
          // Масса создалась, и мы устанавливаем ее координаты и скорость
          masses[0]->pos = Vector3D(0.0f, 0.0f, 0.0f);
          masses[0]->vel = Vector3D(1.0f, 0.0f, 0.0f);
        }
};
```

Метод operate(float dt) класса ConstantVelocity рассчитывает следующее положение массы. Он вызывается основным приложением до перерисовки окна. Скажем, приложение выполняется с 10 fps. Соответственно dt при вызове operate(float dt) будет 0.1 с. Когда произойдет вызов simulate(float dt) массы, ее новое положение будет увеличено на скорость (velocity) * dt, т.е.

```
Vector3D(1.0f, 0.0f, 0.0f) * 0.1 = Vector3D(0.1f, 0.0f, 0.0f)
```

При каждой итерации масса двигается на 0.1 метра вправо. После 10 кадров она переместится вправо на 1 метр. Скорость была 1.0 м/с и масса перемещается на 1.0 м в течение 1 с. Это совпадение или логический результат? Если этот вопрос вызывает у вас затруднение, обдумайте рассуждения изложенные выше.
Когда вы запускаете приложение, вы видите массу, с постоянной скоростью перемещающуюся в направлении x. Приложение имеет два режима движения. При нажатии F3 время будет замедленно в 10 раз (по отношению к реальному). При нажатии F3 время будет идти параллельно реальному. На экране вы увидите линии представляющие координатную плоскость. Расстояние между этими линиями 1 метр. Используя эти линии не трудно заметить, что в режиме реального времени масса перемещается на 1 метр в секунду (соответственно, при замедленном времени 1 метр в 10 секунд). Техника, описанная выше обычна для симуляции в реальном времени. Для того чтобы ее применять, вам необходимо четко оговорить единицы.

Применение силы:

При симуляции массы движущейся с постоянной скоростью мы не применяем сил к массе, т.к. мы знаем, что в случае применения сил тело ускоряется. Когда мы хотим получить ускоренное движение, мы применяем силу. При симуляции мы применяем силы в методе "solve". Когда операции доходят до фазы "simulate" мы имеем результирующий вектор силы (полученный суммированием всех векторов). Он определяет движение.

Скажем, мы хотим применить силу (=1) к массе в направлении x. Тогда мы пишем в методе solve:

```
mass->applyForce(Vector3D(1.0f, 0.0f, 0.0f));
```

Если мы хотим применить другую силу (=2) в направлении y мы добавим:

```
mass->applyForce(Vector3D(0.0f, 2.0f, 0.0f));
```

Таким образом, вы имеете возможность применять любые силы.

2. Масса в условиях гравитации

* class MotionUnderGravitation :
  public Simulation ---> Объект создающий массу, движущуюся под воздействием гравитации.

Класс MotionUnderGravitation создает массу и применяет силу (гравитации) к ней. Сила гравитации равна массе умноженной на ускорение свободного падения (g):

F = m * g

Ускорение свободного падения это ускорение свободного тела. На земле, когда вы бросаете предмет он наращивает скорость на 9.81 м/с в секунду (пока на него действует только сила гравитации). Ускорение свободного падения – константа для всех масс и равна 9.81 м/с/с. (Это не зависит от массы – все массы падают с одинаковым ускорением).

Класс MotionUnderGravitation имеет такой конструктор:

```
class MotionUnderGravitation : public Simulation
{
 Vector3D gravitation;    // ускорение свободного падения

 // Конструктор сначала создает предка с 1й массой в 1 кг.
 // Vector3D Gravitation - ускорение свободного падения

 MotionUnderGravitation(Vector3D gravitation):Simulation(1, 1.0f)
 {
  this->gravitation = gravitation;
  masses[0]->pos = Vector3D(-10.0f, 0.0f, 0.0f);
  masses[0]->vel = Vector3D(10.0f, 15.0f, 0.0f);
 }
 ...
```

Конструктор получает Vector3D gravitation, который является ускорением свободного падения, затем приложение использует его в расчетах.

```
 // Поскольку мы применяем силу, нам понадобится метод "Solve".
 virtual void solve()
 {            // Применяем силу ко всем массам
  for (int a = 0; a < numOfMasses; ++a)
    // Сила гравитации это F = m * g.
     masses[a]->applyForce(gravitation * masses[a]->m);
 }
```

Вы, наверное, заметили в коде формулы силы F=m*g. Приложение создает MotionUnderGravitation с параметром Vector3D(0.0f, -9.81f, 0.0f). -9.81 означает ускорение в направлении – y, таким образом, мы получаем «падающую» массу. Запустите приложение, и понаблюдайте что происходит.

3. Масса, соединенная с точкой пружиной

* class MassConnectedWithSpring :
  public Simulation ---> Объект создающий массу соединенную с точкой пружиной.

В этом примере мы хотим присоединить массу к фиксированной точке пружиной. Пружина должна тянуть массу в сторону точки, к которой она присоединена. В конструкторе класс MassConnectedWithSpring устанавливает точку присоединения и положение массы.

```
class MassConnectedWithSpring : public Simulation
{
 public:
  float springConstant;    // больше springConstant, сильнее сила
                           // притяжения.
  Vector3D connectionPos;  // Точка
  // Конструктор сначала создает предка с 1й массой в 1 кг.
  MassConnectedWithSpring(float springConstant) : Simulation(1, 1.0f)
  {
   // установили springConstant.
   this->springConstant = springConstant;
   // и connectionPos.
   connectionPos = Vector3D(0.0f, -5.0f, 0.0f);
   // положение массы на 10 метров правее connectionPos.
   masses[0]->pos = connectionPos + Vector3D(10.0f, 0.0f, 0.0f);
   // Скорость 0
   masses[0]->vel = Vector3D(0.0f, 0.0f, 0.0f);
 }
 ...
```

Скорость массы 0 и положение на 10 метров правее точки присоединения connectionPos соответственно в начале массу должно тянуть влево. Сила пружины определяется так

F = -k * x

Значение k определяет насколько жесткой должна быть пружина, а x – расстояние от массы до точки присоединения. Отрицательный знак в формуле означает, что это сила притяжения. Если бы знак был положительным пружина толкала бы массу, что, в общем-то не удовлетворяет нашим требованиям ;).

```
virtual void solve() // Будет применяться сила пружины
{
  for (int a = 0; a < numOfMasses; ++a)
  {
   // Находим вектор от массы до точки притяжения
   Vector3D springVector = masses[a]->pos - connectionPos;
   // Применяем силу опираясь на формулу
   masses[a]->applyForce(-springVector * springConstant);
  }
}
```

Сила притяжения в коде выше такая же, как и в формуле F=-k*x. Здесь вместо x мы использовали Vector3D поскольку мы хотим работать в 3D. «springVector» определяет расстояние между положением массы и connectionPos, а springConstant заменяет k. Чем больше значение springConstant, тем больше сила, и тем быстрее движется масса.

В этом уроке я попытался показать ключевую концепцию физической симуляции. Если вы интересовались физикой, у вас не займет много времени создать свой, новый движок.

Jeff Molofee (NeHe)

Урок 40. Моделирование движений веревки

Rope Physics

Эмуляция Веревки

В этом уроке мы с Вами рассмотрим, как эмулировать гибкие движения веревки. Это моделирование основано на простой библиотеке физического моделирования, которая рассмотрена в уроке 39. Это урок будет Вам полезен в том случае, если Вы знаете, как моделировать воздействие силы на массы, как позиция и скорость массы вычисляются с помощью итераций во время моделирования, и как трехмерные векторные операции используются в физике. Если Вы сомневаетесь в том, что хорошо знаете эти темы, то читайте про них в уроке 39 или другую подходящую документацию, и разработайте несколько прикладных программ.

Цель физического моделирования состоит в том, чтобы сформировать данные о воздействиях на массы, основанные на физики, которые будут действовать так же как в естественной среде. Движение, полученное при моделировании, не может быть точно таким же, как в природе. Модель, которая описывает движение, должна формировать необходимые физические данные. Модель, которую мы создаем должна быть точно сформулирована, мы должны задать, как точно и детально мы описываем движение, которое мы стремимся смоделировать, чтобы формировать необходимые физические данные. Мы хотим описать движение атомов, электронов или фотонов, или мы стремимся аппроксимировать движение кластера частиц? Каков масштаб того, что мы хотим видеть? Какова арена действия пространства и времени?

Тот масштаб пространства и времени, которые мы хотим наблюдать, связан с:

1. Математикой движения
2. Производительностью компьютера, который мы используем для моделирования

1. Математика Движения:

Здесь, математика движения называется "классической механикой", в которой массы рассматриваются как частицы и ускорение им придается силами, действующими во времени. В масштабе, который мы можем наблюдать невооруженным глазом, классическая механика, допустима для использования. Поэтому, мы можем использовать классическую механику для моделирования объектов и механизмов из нашей обычной жизни. В уроке 39, сила тяготения и сила упругости пружины применялась к массам в 1 кг при помощи классической механики. В этом уроке, мы будем использовать классическую механику для моделирования гибких движений веревки.

2. Производительность компьютера, который мы используем для моделирования:

Производительность компьютера для моделирования, определяет, степенью детализации, которую мы можем наблюдать. Например, при моделировании пешехода на медленном компьютере, мы бы подумали бы об удалении моделирования движений пальцев. Пальцы ступни, конечно, играют важную роль. Хотя и без моделирования ступней мы могли бы получить идущего человека. Возможно, качество движения было бы низким, но затраты на вычисления будут ниже. В случае пешехода, производительность компьютера вынуждает нас выбирать, что в первую очередь моделировать ступни или ноги, или пальцы на руке или на ноге.

Разработка физических данных для веревки:

Имея в наличии классическую механику (как математику движения) и компьютер с процессором не менее 500 МГц, мы будет проектировать физические данные для моделирования веревки. Во-первых, мы должны определить, как детально мы хотим наблюдать процесс движения веревки. При кодировании, мы будем использовать Physics1.h из урока 39. В Physics1.h, мы имеем класс массы (class Mass), который представляет массу как точечную частицу. Мы можем использовать этот класс массы. Если мы связываем массы, подобные точкам, друг с другом как при моделировании пружины, мы сможем сформировать физическую модель, чтобы эмулировать веревку. Исходя из этой модели, которую мы рассматриваем, мы определяем насколько точной будет эмуляция движений веревки. Мы сможем понять, что мы готовы воспроизвести покачивание или помахивание веревкой, но мы не сможем воссоздать скручивание веревки. (Чтобы понять, что такое скручивание веревки, возьмите веревку в руки и зажмите в ладони, и потирайте ладони друг об друга, тогда веревка будет скручиваться.) Мы не можем наблюдать скручивание, потому что мы используем в данной модели точки. Точки не могут вращаться вокруг оси, чтобы эмулировать скручивание веревки (примечание переводчика: но если вы будет каждый фрагмент веревки представлять не линией, а цилиндром, то вы вполне сможет эмулировать и скручивание веревки). Давайте будем использовать эту модель и примем что, движение веревки ограничено покачиванием и помахиванием. Давайте, также определим, что мы хотим наблюдать движение помахивания веревкой, с максимальной детализацией 10 см. Это означает, что у веревки будут нарушения непрерывности до 10 см. Я выбрал эти ограничения, потому что я хочу использовать в веревке приблизительно 50 или 100 частиц (из-за производительности), и я хочу, чтобы эта веревка была приблизительно от 3 до 4 метров длиной. Что означает, что есть приблизительно от 3 до 8 см между частицами веревки, что не превышает уровень нарушения непрерывности, который мы выбрали (10 см).

Определение уравнения движения:

Уравнение движения с точки зрения математики представляет из себя дифференциальное уравнение второго порядка и концептуально означает силы, действующие в физическом окружении. Давайте использовать концептуальное значение, потому что это звучит лучше. Определение уравнения движения означает определение сил. В модели веревки, силы будут действовать на частицы, которые составляют веревку. Первая сила будет упругое растяжение/натяжение пружины между этими частицами. Ниже, каждая частица помечена как "o", и пружина показана как "----":

O----O----O----O

1 2 3 4

Частица 1 связана с 2, 2 с 3, и 3 с 4. Мы имеем 4 частицы в этой веревке и 3 пружины. Пружина - источник силы между двумя частицами. Помните, что сила упругости сформулирована так:

Сила = -k * x
k: константа жесткости пружины
x: расстояние массы от точки до места присоединения

Формула для вычисления упругости пружины, которую мы будем использовать, будет похожа на эту, но немного другая. Если бы мы использовали указанную формулу, то при этом веревка бы сжималась! Поскольку, если x не ноль (x - расстояние между двумя связанными массами в нашей модели веревки), возникает сила. Поэтому все частицы

веревки были бы стянуты друг другу, пока x не стало бы равным нолю. Это не то, что мы хотим. Вообразите, что веревка лежит на столе. Мы хотим, чтобы наша веревка осталась неизменной подобно веревке на столе. Так или иначе, мы должны сохранить в этом случае неизменной длину веревки. Чтобы сделать это, сила между двумя любыми частицами должна быть ноль, тогда x имеет положительное значение. Давайте перепишем формулу так:

Сила =-k * (x - d)
k: константа жесткости пружины
x: расстояние массы от точки до места присоединения
d: положительная константа, задающее расстояние, при котором пружина будет устойчивой

С этой формулой ясно, что, если расстояние между двумя массами равняется d, никакая сила не будет действовать. Пусть мы имеем 100 частиц. Если мы выбираем d как 5 см (0.05 метра), мы имеем устойчивую веревку в 5 метров, когда она лежит на столе. Когда x - больше чем d, пружина вытягивается, а когда меньше сжимается.

Теперь, формула производит нужное движение, но требуется больше. Требуется некоторое трение. Если нет трения, физическая система сохраняет энергию. Если мы не используем коэффициент трения, веревка никогда не кончит качаться. Но вначале давайте взглянем на код.

Класс пружины

Класс пружины связывает две массы и прикладывает силу к каждой из этих масс.

```
class Spring // Объект для представления пружины с внутренней силой трения двух связанных масс
        // Пружина нормальной длины (длина, при которой пружина не приводится
        // в действие любыми силами)
{
public:
  Mass* mass1;              // Первая масса – один конец пружины
  Mass* mass2;              // Вторая масса – другой конец пружины

  float springConstant;      // Константа жесткости пружины
  float springLength;        // Длина, при которой пружина не приводится в действие любыми силами
  float frictionConstant;    // Константа трения пружины

  Spring(Mass* mass1, Mass* mass2,
   // Конструктор
    float springConstant, float springLength, float frictionConstant)
  {
    this->springConstant = springConstant;
    this->springLength = springLength;
    this->frictionConstant = frictionConstant;
    this->mass1 = mass1;
    this->mass2 = mass2;
  }

  void solve()              // Метод решение: действие сил
  {
    Vector3D springVector = mass1->pos - mass2->pos; // Вектор между 2 массами
     float r = springVector.length(); // Расстояние между 2 массами
     Vector3D force;                  // Сила инициализируется нулевым значением

    if (r != 0) // Чтобы не было деления на ноль
      // Силы пружины добавляются в силу
      force += -(springVector / r) * (r - springLength) * springConstant;
    ...
```

В конструкторе задаются переменные mass1, mass2, и константы. Наиболее интересен метод solve(). В этом методе происходит применение сил. Чтобы применить силу, мы должны написать формулу пружины, которую мы получили:

force = -k * (x - d)

Трехмерный вектор, задающий расстояние между массами:

Vector3D springVector = mass1->pos - mass2->pos; (Вектор между 2 массами)

найден. Затем нулевая сила создана:

Vector3D force;

Затем сила пружины добавлена к ней:

force += (springVector / r) * (r - springLength) * (-springConstant);

Чтобы реализовать формулу выше, мы, во-первых, получаем вектор единичной длины между массами:

(springVector / r)

Затем с используем этот вектор совместно с (x - d) частью формулы:

(springVector / r) * (r - springLength)

Далее мы умножаем вектор на:

(-springConstant)

Который соответствует -k в первоначальной формуле (минус означает натяжение, а не отталкивание). Мы завершили вычисление части силы связанной с натяжением. Давайте, вычислим трение. Это трение в пружине. Пружина имеет тенденцию терять энергию при действии сил действующих в ней. Если Вы применяете силу к массе в направлении противоположном тому, куда движутся массы, то вы заставите массы двигаться медленнее (т.е. как будто будет действовать сила трения). Поэтому, можно вычислить силу трения из скорости движения массы:

Сила трения = -k * скорость
k: константа, задающая величину трения
Скорость: скорость массы, на которую действует сила трения

Формула трения могла бы быть написана и по-другому, но и эта формула будет прекрасно работать в нашей модели веревки. В этой формуле рассматривается только одна масса. В пружине мы имеем две массы. Мы можем вычислить разницу между скоростями двух масс и получить относительную скорость. Это и будет внутреннее трение в пружине.

```
   (void solve() continued)

   force += -(mass1->vel - mass2->vel) * frictionConstant; // Сила трения
   mass1->applyForce(force);  // Добавим силу к mass1
   mass2->applyForce(-force); // Добавим силу к mass2
 }                 // Конец метода solve
```

force += -(mass1->vel - mass2->vel) * frictionConstant;

Выше, сила трения, полученная из разницы скоростей масс, добавлена к силе пружины. Сила применяется к mass1:

mass1->applyForce(force);

И противоположная сила применяется к mass2:

mass2->applyForce(-force);

В физике, все взаимодействия происходят между двумя частицами. Сила всегда действует на две массы в противоположных направлениях. В моделировании, если одна масса незначительна по сравнению с другой силой, действующей на большую массу можно пренебречь, поскольку ускорение большой массы будет маленькое. Например, сила гравитации Земли притягивает маленькую массу к себе, но и эта масса тянет Землю к себе, но мы пренебрегаем этой силой, так как она ничтожно мала (примечание переводчика: даже, если сила не мала, но действует постоянно и быстро не меняется, то ее просто учесть как константу).

Мы написали уравнение движения, которое описывает, как действует сила упругости в веревке. Чтобы завершить наше моделирование, мы должны воспроизвести среду, в которой находится веревка, и рассмотреть внешние, по отношению к ней, силы, действующие на нее. Давайте вначале введем гравитацию в эту искусственную среду. Когда есть тяготение, массы испытывают силу гравитации. Во-вторых, я также хотел бы иметь и трение о воздух, которое можно просто определить как:

Сила трения = -k * скорость
k: константа, задающая величину трения
Скорость: скорость массы, на которую действует сила трения

В-третьих, пусть у нас будет некая плоская поверхность (что-то типа стола), по которой мы сможем таскать конец подвешенной веревки. Поэтому, наше уравнение движения надо расширить. Надо добавить тяготение, трение о воздух и силы, действующие со стороны полоской поверхности. Гравитационную силу добавить просто:

Сила = (ускорение гравитации) * масса

Тяготение и трение о воздух будут действовать на каждую частицу на веревке. Но, что можно сказать относительно сил, действующих со стороны поверхности? Сила от поверхности будет также действовать на каждую массу. Мы должны сформулировать модель такого взаимодействия. Моя модель довольно проста: поверхность выталкивает массу наверх и проявляет силу трения. Сила должна действовать на массу только тогда, когда эта масса касается поверхности.

Задание начальных значений для моделирования

Наша среда готова к моделированию. Единицами измерения расстояния будут метры, секунды (для времени), и кг (для веса).

Для того чтобы задать начальные значения, мы должны определить ориентацию веревки перед началом моделирования и определить некоторые константы. Пусть гравитация действует в отрицательном направлении у с 9.81 м/с/с. Пусть один конец веревки подвешен в 4 метрах над поверхностью. Пусть веревка лежит горизонтально до запуска симуляции. Чтобы это выполнить, мы должны разнести частицы на 5 см друг от друга (4 метра / 80 =0.05 метров = 5 см). Также зададим, что это нормальная длина пружины (длина при которой сила упругости равна нулю) - 5 см так, чтобы эта веревка была без напряжения в начале моделирования. Определим общую массу веревки равную 4 кг (тяжелая веревка, цепь). Тогда каждая масса будет весить 0.05 кг (50 граммов). До того как двигаться дальше, посмотрим, что у нас есть:

1. Гравитационное ускорение: 9.81 м/с/с в отрицательном направлении у
2. Число масс: 80
3. Нормальное расстояние между двумя соседними массами: 5 см (0.05 метров)
4. Вес массы: 50 граммов (0.05 кг)
5. Ориентация веревки: горизонтальная, без напряжения

Затем, мы можем найти константу пружины. Когда мы вешаем веревку за верхний конец, она конечно вытянется. Пружина наверху веревки больше всего вытянется. Я не хочу, чтобы пружина вытянулась больше, чем на 1 см (0.01 м). Вес, который эта пружина несет – это почти вся веревка (частица на верхнем конце исключительна). Сила равна:

f = (масса веревки) * (гравитационное ускорение) = (4 кг) * (9.81) ~ = 40 Н

Сила упругости пружины должна сбалансировать 40 N:

сила пружины = -k * x = -k * 0.01 м

Сумма этих сил должна быть равной нулю:

40 Н + (-k * 0.01 м) = 0

Отсюда мы вычисляем k:

k = 4000 Н / м

Для простоты примем, что k равно 10000 Н/м, что дает более жесткую веревку, которая приблизительно вытянется на 4 мм.

Чтобы найти константу трения в пружине, мы должны проделать более сложные вычисления. Поэтому, будем использовать значение, которое я подобрал экспериментальным путем:

springFrictionConstant = 0.2 Н/(м/с)

Константа трения пружины равна 0.2 Н/(м/с) и прекрасно подходит для нашей веревки, чтобы она выглядела реалистично (это мое мнение после того, как я провел моделирование).

Прежде, чем перейти к трению о воздух и силам, воздействующим от поверхности, давайте взглянем на класс RopeSimulation. Этот производный класс класса Simulation из Physics1.h, который мы рассмотрели в уроке 39. Класс Simulation имеет четыре метода для выполнения моделирования. Вот они:

1. virtual void init() ---> Сброс сил.
2. virtual void solve() ---> Намеченные силы применяются.
3. virtual void simulate(float dt) ---> Позиция и скорость итерационно меняются.
4. virtual void operate(float dt) ---> Методы 1., 2., и 3. вместе вызываются.

В классе RopeSimulation, мы переопределим метод solve() и simulate(float dt), поскольку мы имеем специальную реализацию этих методов для веревки. Мы применим метод solve(), и закрепим верхний конец веревки в методе simulate(float dt).

Класс RopeSimulation производный от класса Simulation (из Physics1.h). В нем моделируется веревка с точечными частицами, связанными пружинами. Пружина имеет внутреннее трение и нормальную длину. Один конец веревки привязан к точке пространства названном "Vector3D ropeConnectionPos". Эта точку можно сдвинуть методом "void setRopeConnectionVel (Vector3D ropeConnectionVel)". RopeSimulation создает трение о воздух и плоскую поверхность (или землю) с нормалью направленной в положительном направлении y. RopeSimulation вычисляет силу, приложенную к этой поверхности. В коде, поверхность – называется "земля".

Класс RopeSimulation начинается так:

```
class RopeSimulation : public Simulation // Объект моделирования веревка,
                                         // который взаимодействует с плоской поверхностью
                                         // и воздухом
{
public:
  Spring** springs; // Пружины связывают numOfMasses масс
  Vector3D gravitation; // Ускорение гравитации (Гравитация будет применена ко всем массам)
  Vector3D ropeConnectionPos; // Точка в пространстве, которая используется для задания позиции
                              // первой массы в пространстве (с индексом 0)
  Vector3D ropeConnectionVel; // Скорость перемещения ropeConnectionPos
                              // с помощью нее мы можем раскачивать веревку
  float groundRepulsionConstant; // Константа для представления того,
                                 // насколько сильно земля будет отталкивать массы
  float groundFrictionConstant; // Константа трения, которое возникает от земли, используется
                                // для скольжения веревки по земле
  float groundAbsorptionConstant; // Константа поглощения трения об землю, используется
                                  // для вертикальных столкновений веревки с землей
  float groundHeight; // Y координата земли (земля это плоская поверхность расположенная
                      // лицевой гранью с положительном направлении оси Y
  float airFrictionConstant; // Константа трения о воздух
```

Класс имеет конструктор с 11 параметрами:

```
  RopeSimulation(                 // Длинный предлинный конструктор
    int numOfMasses,              // 1. Число масс
    float m,                      // 2. Вес каждой массы
    float springConstant,         // 3. Насколько пружина тугая
    float springLength,           // 4. Длина спокойной пружины
    float springFrictionConstant, // 5. Трение изнутри
    Vector3D gravitation,         // 6. Гравитация
    float airFrictionConstant,    // 7. Трение воздуха
    float groundRepulsionConstant, // 8. Отталкивание от земли
    float groundFrictionConstant, // 9. Трение о землю
```

```
  float groundAbsorptionConstant,// 10. Поглощение земли
  float groundHeight             // 11. Высота земли(Y позиция)
  ) : Simulation(numOfMasses, m) // Суперкласс создает массы с весом m каждая
{
  this->gravitation = gravitation;
  this->airFrictionConstant = airFrictionConstant;
  this->groundFrictionConstant = groundFrictionConstant;
  this->groundRepulsionConstant = groundRepulsionConstant;
  this->groundAbsorptionConstant = groundAbsorptionConstant;
  this->groundHeight = groundHeight;

  for (int a = 0; a < numOfMasses; ++a) // Начальные позиции масс
  {
    masses[a]->pos.x = a * springLength; // X-позиция masses[a] с расстоянием
                                         // springLength от его соседа
    masses[a]->pos.y = 0; // Y-позиция равна 0
    masses[a]->pos.z = 0; // Z-позиция равна 0
  }

  springs = new Spring*[numOfMasses - 1]; // Создание [numOfMasses - 1] точек для пружины
                // ([numOfMasses - 1] пружин необходимы для numOfMasses)

  for (a = 0; a < numOfMasses - 1; ++a) // Создание пружин
  {
    // Пружина между массой "a" и массой "a + 1".
    springs[a] = new Spring(masses[a], masses[a + 1],
                            springConstant, springLength, springFrictionConstant);
  }
}
```

Всего создается [numOfMasses - 1] пружин (вспомните рисунок: O----O----O----O). Массы первоначально горизонтальном положении. Затем применение сил реализовано в методе solve, уравнения движения будут решены в процессе моделирования. Метод solve выгляди так:

```
  void solve() // solve() переопределен, поскольку мы применяем силы
  {
    for (int a = 0; a < numOfMasses - 1; ++a) // Применение сил для всех пружин
    {
      springs[a]->solve();
    }

    for (a = 0; a < numOfMasses; ++a) // Цикл применения сил ко всем массам
    {
      masses[a]->applyForce(gravitation * masses[a]->m); // Сила гравитации
      // Трение о воздух
      masses[a]->applyForce(-masses[a]->vel * airFrictionConstant);

      if (masses[a]->pos.y < groundHeight) // Силы от земли, если массы коснулись земли
      {
        Vector3D v; // Временный вектор
        v = masses[a]->vel; // Взять скорость
        v.y = 0;            // Пренебречь компонент скорости в Y-направлении

        // Скорость в Y-направлении пренебрежем, поскольку мы будем применять силу трения
        // для создания эффекта скольжения. Скольжение параллельно земле. Скорость в Y-направлении
        // будет использоваться для эффекта поглощения
        // Сила трения о землю
        masses[a]->applyForce(-v * groundFrictionConstant);
        v = masses[a]->vel; // Взять скорость
        v.x = 0;            // Пренебречь x и z компонентами скорости
        v.z = 0;            // Мы будем использовать v в эффекте поглощения
```

```
      // Выше, мы получили скорость, которая вектор которой направлен вертикально
      // земле и это будет использовано в эффекте поглощения

      if (v.y < 0) // Пусть энергия поглощения только, затем масса сталкивается с землей
      {
        // Эффект поглощения
        masses[a]->applyForce(-v * groundAbsorptionConstant);
      // Земля будет отталкивать массы подобно пружине.
      // "Vector3D(0, groundRepulsionConstant, 0)" – создаем вектор в направлении
      // плоскости нормали с модулем groundRepulsionConstant.
      // (groundHeight - masses[a]->pos.y) – мы отталкиваем массу,
      // как только она сталкивается с землей
      Vector3D force = Vector3D(0, groundRepulsionConstant, 0) *
        (groundHeight - masses[a]->pos.y);
      masses[a]->applyForce(force); // Сила отталкивания земли
    }
  }
}
```

Вначале, в коде выше, вычисляются все силы упругости пружин (порядок не имеет значения). Затем вычисляются силы общие для всех масс в цикле for(;;). Это силы гравитации, трения о воздух и силы от земли. Вычисление сил от земли выглядит немного запутанным, но это фактически столь же просто, как и вычисление других сил. Эффект скольжения веревки по земле обеспечивается силой трения, при вычислении которой пренебрегли скоростью в y направлении. Направление y – это направление вверх от лицевой стороны земли. Эффект скольжения не должен быть в направлении на лицевую сторону земли. Именно поэтому y опущено. Иначе дело обстоит с эффектом поглощения. Сила поглощения применяется только в направлении лицевой стороны земли. Исключение для эффекта поглощения состоит в том, что не надо применять силы, когда масса движется вдоль земли. Иначе веревка бы прилипала к земле, в то время как мы тянем ее вверх. Мы реализуем этот исключительный случай если v.y < 0. Наконец есть сила отталкивания от земли. Земля отталкивает массы точно так же как пружина, выталкивая массу наверх.

В классе RopeSimulation моделирование начинается с первой частицы веревки. Цель состоит в том, чтобы создать способ раскачивания веревки с верхнего конца. Для моделирования используются значения ropeConnectionVel и ropeConnectionPos.

```
void simulate(float dt) // переопределено поскольку мы хотим моделировать веревку
{
  Simulation::simulate(dt); // Моделирование масс
  ropeConnectionPos += ropeConnectionVel * dt; // Итерация изменения позиции
  if (ropeConnectionPos.y < groundHeight) // Веревка не будет двигаться под землей
  {
    ropeConnectionPos.y = groundHeight;
    ropeConnectionVel.y = 0;
  }

  masses[0]->pos = ropeConnectionPos; // Сдвиг верхней массы
  masses[0]->vel = ropeConnectionVel; // Изменение скорости верхней массы
}
```

С помощью этого метода устанавливается ropeConnectionVel:

```
void setRopeConnectionVel(Vector3D ropeConnectionVel)
{
  this->ropeConnectionVel = ropeConnectionVel;
}
```

Эта функция используется при моделировании. Используя клавиши мы задаем ropeConnectionVel, и мы можем перемещать веревку так, как если бы мы держали ее за один конец.

Есть некоторые константы, значение которых очень трудно вычислить прежде, чем мы запустим моделирование. Вот подходящие значения этих константы (взято из Physics2Application.cpp):

```
RopeSimulation* ropeSimulation =
  new RopeSimulation(
    80,                 // 80 частиц (масс)
    0.05f,              // Каждая частица имеет вес в 50 грамм
    10000.0f,           // springConstant в веревке
    0.05f,              // Нормальная длина пружины в веревке
    0.2f,               // Константа внутреннего трения пружины
    Vector3D(0, -9.81f, 0), // Ускорение гравитации
    0.02f,              // Константа трения о воздух
    100.0f,             // Константа отталкивания земли
    0.2f,               // Константа трения скольжения о землю
    2.0f,               // Константа поглощения земли
    -1.5f);             // Высота земли
```

Изменяя эти значения, Вы сможете попробовать различные варианты движения веревки. Отметьте, что "высота земли" равна -1.5 метром. Веревка вначале находится в y = 0. При этом мы видим веревку, которая раскачивается над землей, а затем сталкивается с ней. Вспомните, что в уроке 39 задавалось максимальное значение dt. В этом случае я нашел, что этот максимум dt должен быть равен 0.002 секунды. Если Ваши изменения в параметрах уменьшат максимум dt, то моделирование может быть неустойчивым, и моделирование не будет работать. В этом случае Вам надо найти новый максимум dt. Увеличение силы и/или уменьшение масс вызывают также появление неустойчивости, потому что ускорение при этом возрастает (вспомните "ускорение = сила/масса").

Так же как и в уроке 39, моделирование используется в файле приложения (Physics2Application.cpp):

```
float dt = milliseconds / 1000.0f; // Конвертирование миллисекунд в секунды
float maxPossible_dt = 0.002f; // Максимум dt, сек
                       // Это необходимо, чтобы не было потери точности dt
// Вычислим число итераций для обновления
int numOfIterations = (int)(dt / maxPossible_dt) + 1;
if (numOfIterations != 0) // Чтобы не было деления на ноль
  dt = dt / numOfIterations; // dt обновлено в зависимости от numOfIterations
for (int a = 0; a < numOfIterations; ++a) // "numOfIterations" итераций моделирования
  ropeSimulation->operate(dt);
```

После запуска приложения, используете клавиши курсора, и клавиши HOME и END, чтобы сдвинуть веревку. Попробуйте с ней поиграть. Наблюдайте помахивание и раскачивание.

Процедура моделирования загружается процессор. Поэтому, рекомендуется оптимизировать ваш транслятор. При компиляции в Visual C++, когда выбран режим окончательной сборки (Release), моделирование веревки выполняет в 10 раз быстрее, чем при компиляции в отладочном режиме (Debug). В отладочном режиме нужен минимум 500 МГц процессор. В режиме окончательной сборки требование гораздо меньше.

В этом уроке представлено полное моделирование. Есть и настройки, и теория, и проект, и реализация. Более продвинутое моделирование основывается на рассмотренном материале. Наиболее часто используются концепции, которые рассмотрены в этом примере с веревкой. Это также подходит и для физического моделирования в программировании игр. Пробуйте использовать физику в ваших программах и создать ваши собственные демонстрации и игры.

Комментарии или вопросы, пожалуйста, посылайте по адресу:

Jeff Molofee (NeHe)

Урок 41. Объемный туман и загрузка изображений через интерфейс IPicture

Volumetric Fog & IPicture Image Loading

Вашему вниманию предлагается еще один занимательный урок. На этот раз я буду пытаться объяснить, что такое объемный туман, используя расширение glFogCoordf. Чтобы запустить эту демонстрационную версию, ваша видео плата должна поддержать расширение "GL_EXT_FOG_COORD". Если Вы не уверены, поддерживает ли ваша плата это расширение, у вас есть два пути: Первое скачать исходники на VC++, и посмотреть, выполняется ли он. Второе скачать урок 24, и просмотреть список расширений, поддерживаемых вашей видео платой.

В этом уроке я познакомлю вас с кодом NeHe IPicture, который способен загружать BMP, EMF, GIF, ICO, JPG и WMF файлы с вашего компьютера или web-странички. Вы также узнаете, как использовать расширение "GL_EXT_FOG_COORD", чтобы создавать круто смотрящийся объемный туман (туман, который может перемещаться в замкнутом пространстве, не изменяя остальную часть сцены).

Если этот урок не работает на вашей машине, первом делом, вы должны проверить, что у вас стоит самый свежий видеодрайвер. Если же у вас драйвер самый новый, а демка по-прежнему не работает, то … вы может быть захотите купить новую видео плату. Начиная с GeForce 2 все будет прекрасно работать, и стоит не дорого. Если ваша плата не поддерживает расширение для вывода объемного тумана, кто говорит, что она не поддерживает и другие расширения?

Те из вас, кто не смог запустить демку, и чувствует себя обделенным … помните следующее: каждый день я получаю по крайней мере 1 письмо, в котором от меня требуют написать новый урок. Многие из уроков, которые Вы просили уже в сети! Люди не читают все, что уже есть в сети, а останавливаются на той теме, которой они больше всего заинтересовались. Некоторые уроки слишком сложные, и потребовали бы от меня несколько недель программирования. Наконец, есть уроки, которые я смог бы написать, но обычно избегаю этого, потому что я знаю, что они не будут выполняться на всех видюхах. Теперь, когда платы типа GeForce достаточно дешевы, чтобы любой со скидкой может позволить себе одну из них, я больше не могу не писать такие уроки. Честно говоря, если ваша видео плата поддерживает только базовые расширения, вы многое теряете! И если я буду продолжать пропускать такие темы как расширения, то уроки будут запаздывать!

Сказано – сделано! Давайте проштудируем кое-какой код!!!

Начало кода очень похоже на основу старого, и почти идентично новому коду NeHeGL. Единственное отличие - дополнительная строка кода, подключающая библиотеку OLECTL. Этот заголовочный файл должен быть включен, если Вы хотите, чтобы код IPicture функционировал. Если Вы не напишете эту строчку, то у вас возникнут ошибки при попытке использовать IPicture, OleLoadPicturePath и IID_IPicture.

Точно так же как и в базовом коде NeHeGL, мы используем комментарий *pragma (lib...) чтобы автоматически включить требуемые библиотечные файлы! Заметьте, нам не надо больше включать glaux библиотеку (я – уверен, многие из вас сейчас улыбаются).

Следующие три строчки кода проверяют, определено ли значение CDS_FULLSCREEN. Если нет (что верно для большинства трансляторов), мы задаем ему значение 4. Я знаю, что многие из вас писали мне об этом по электронной почте, чтобы узнать о причине ошибок при компилировании кода, используя CDS_FULLSCREEN в DEV C ++. Добавьте эти три строки, и ошибок у вас не будет!

```
#include <windows.h>        // Заголовочный файл Windows
#include <gl\gl.h>          // Заголовочный файл библиотеки OpenGL32
#include <gl\glu.h>         // Заголовочный файл библиотеки Glu32
#include <vfw.h>            // Заголовочный файл для «Видео для Windows»
#include "NeHeGL.h"         // Заголовочный файл NeHeGL.h
#include <olectl.h>         // Заголовочный файл для библиотеки
                    // управляющих элементов OLE (используется в BuildTexture)
#include <math.h>           // Заголовочный файл для математической библиотеки
                    // (используется в BuildTexture)

#pragma comment( lib, "opengl32.lib" ) // Искать OpenGL32.lib при линковке
#pragma comment( lib, "glu32.lib" )    // Искать GLu32.lib при линковке

#ifndef CDS_FULLSCREEN        // CDS_FULLSCREEN не определяется некоторыми
#define CDS_FULLSCREEN 4      // компиляторами. Определяем эту константу
#endif                        // Таким образом мы можем избежать ошибок
```

```
GL_Window*  g_window;                // структура окна
Keys*    g_keys;                     // клава
```

В следующей части кода, мы устанавливаем цвет нашего тумана. В данном случае мы хотим, чтобы он был темно-оранжевого цвета. Небольшое количество красного (0.6f), смешанного с еще меньшим количеством зеленого (0.3f) даст нам цвет, который мы хотим.

Переменная camz типа Glfloat будет использована позже, чтобы определять позицию нашей камеры внутри длинного и темного коридора! Мы будем перемешаться вперед и назад вдоль коридора, выполняя преобразования на оси Z, перед тем как выполнить рисование.

```
// Наши переменныes
GLfloat  fogColor[4] = {0.6f, 0.3f, 0.0f, 1.0f}; // цвет тумана
GLfloat camz;                                    // «глубина» камеры по Z
```

Точно так же как CDS_FULLSCREEN имеет предопределенное значение 4, переменные GL_FOG_COORDINATE_SOURCE_EXT и GL_FOG_COORDINATE_EXT также имеют предопределенные значения. Как упомянуто в комментариях, значения были взяты из заголовочного файла GLEXT. Его можно свободно скачать из сети. Огромное спасибо Льву Повалаеву за создания такого ценного заголовочного файла! Эти значения должны быть установлены, если вы хотите, чтобы код компилировался! Результатом является то, что, мы имеем два новых списка перечислений (GL_FOG_COORDINATE_SOURCE_EXT и GL_FOG_COORDINATE_EXT).

Чтобы использовать функцию glFogCoordfExt, мы должны объявить прототип функции через typedef, чтобы наша функция соответствовала точке входа расширения. А если по-русски, то нам надо сообщить нашей программе число параметров и тип каждого параметра, которые подаются на вход функции glFogCoordfExt. В данном случае, мы передаем один параметр для этой функции, и это - переменная с плавающей точкой.

Затем мы должны объявить глобальную переменную нашего типа (PFNGLFOGCOORDFEXTPROC). Это - первый шаг создания нашей новой функции (glFogCoordfEXT). Эта переменная объявлена глобальной, для того чтобы мы смогли использовать команду где - угодно в нашем коде. Название, которое мы используем, должно точно соответствовать фактическому имени расширения. Фактическое имя расширения - glFogCoordfEXT и название, которое мы используем - также glFogCoordfEXT.

Однажды использовав wglGetProcAddress, чтобы присвоить переменной определенного выше типа адрес функции расширения драйвера OpenGL, мы можем вызывать glFogCoordfExt, как если бы это была обычная функция. Подробнее об этом после!

Последняя строка подготавливает все необходимое для нашей единственной текстуры.

Так, что пока мы имеем вот что ...

Мы знаем, что PFNGLFOGCOORDFEXTPROC принимает одно значение с плавающей точкой (GLfloat coord). Поскольку glFogCoordfExt – имеет тип PFNGLFOGCOORDFEXTPROC, то вызов этой функции будет выглядеть вот так: glFogCoordfEXT (GLfloat coord). Наша функция определена, но не будет ничего делать, потому что glFogCoordfExt – NULL (мы по-прежнему должны прикрутить glFogCoordfExt к адресу функции расширения OpenGL драйвера).

Надеюсь, что это понятно… на самом деле, все очень просто, когда вы уже знаете, как это работает... но описывать это чрезвычайно трудно (по крайней мере, для меня, это так). Если кто-либо может переписать этот раздел урока, используя простую и не сложную формулировку, пришлите ее мне! Единственный способ, с помощью которого я смог объяснить это лучше использовать иллюстрации, но сейчас я спешу выложить этот урок в сеть!

```
// переменные необходимые для FogCoordfEXT
#define GL_FOG_COORDINATE_SOURCE_EXT  0x8450          // значение из GLEXT.H
#define GL_FOG_COORDINATE_EXT    0x8451               // значение из GLEXT.H

typedef void (APIENTRY * PFNGLFOGCOORDFEXTPROC) (GLfloat coord); // прототип функции
PFNGLFOGCOORDFEXTPROC glFogCoordfEXT = NULL; // наша функция glFogCoordfEXT

GLuint  texture[1]; // единственная текстура для стен
```

Ну, а теперь, чтобы было веселее, фактический код, который превращает изображение в текстуру, использует волшебство IPicture :).

Эта функция требует путь (путь к фактическому изображению, которое мы хотим загрузить, или имя файла, или URL в сети) и идентификатор текстуры (например... texture [0]).

Мы должны создать контекст устройства для нашего временного растра. Нам также необходимо иметь указатель на место в памяти для хранения растра (hbmpTemp), в указетель на интерфейс IPicture, а также строковую переменную для хранения пути (до файла или URL). Еще нам понадобится по 2 переменных для хранения ширины и высоты изображения. Переменные lwidth и lheight хранят фактическую ширину и высоту изображения. Переменные lwidthpixels и lheightpixels хранят ширину и высоту в пикселях, откорректированных, чтобы исправить максимальный размер текстуры, который может отобразить видеокарта. Максимальный размер текстуры будет храниться в glMaxTexDim.

```
int BuildTexture(char *szPathName, GLuint &texid)
  // читаем изображение и преобразуем его в текстуру
{
 HDC    hdcTemp;                    // DC для растра
 HBITMAP    hbmpTemp;           // иногда храним в ней растр
 IPicture  *pPicture;           // интерфейс IPicture
 OLECHAR    wszPath[MAX_PATH+1]; // полный путь до картинки (WCHAR)

 char    szPath[MAX_PATH+1];     // полный путь до картинки
long    lWidth;                 // ширина в логических единицах
 long    lHeight;               // высота в логических единицах
 long    lWidthPixels;          // ширина в пикселях
 long    lHeightPixels;         // высота в пикселях
 GLint    glMaxTexDim ;         // максимальный размер текстуры
```

Следующий раздел кода берет имя файла и проверяет его, является ли он URL'ом или путем до файла. Мы делаем это, путем проверки, содержит ли имя файла http://. Если имя файла - URL, мы копируем имя в szPath.

Если имя файла не содержит URL, мы получаем текущий каталог. Если вы сохранили демонстрационную версию в C:\wow\lesson41, и пробовали загружать data\wall.bmp, программе нужно знать полный путь до wall.bmp файла, а не только то, что bmp файл сохранен в папке 'data'. GetCurrentDirectory найдет текущий путь, т.е. путь до папки, где лежит .EXE файл и папка 'data'.

Если .exe был сохранен в "c:\wow\lesson41", то текущая директория была бы "c:\wow\lesson41". Мы должны добавить "\\" к концу пути до текущего каталога, а затем "data\wall.bmp". "\\" - это просто "\". Если мы все это соединим "c:\wow\lesson41" + "\" + "data\wall.bmp", то у нас получится "c:\wow\lesson41\data\wall.bmp". Теперь понятнее?

```
 if (strstr(szPathName, "http://")) // Если путь содержит http:// то...
 {
  strcpy(szPath, szPathName); // прикрутить PathName к szPath
 }
 else // иначе загрузить из файла
 {
  GetCurrentDirectory(MAX_PATH, szPath); // дайте каталог, где мы сидим
  strcat(szPath, "\\");                 // добавим к концу пути '\'
  strcat(szPath, szPathName);           // приклеим PathName
 }
```

Так что у нас есть сейчас полный путь в переменной szPath. Теперь нам надо его перекинуть из ASCII в Unicode так, чтобы OleLoadPicturePath воспринял его. Первая строчка кода делает это за нас. Результат сохраняется в wszPath.

CP_ACP означает кодовую страницу ANSI. Второй параметр определяет обработку не отображаемых символов (далее по коду мы его игнорируем). Символы в строке szPath будут конвертированы в расширенные символы. 4-ый параметр – длина строки с расширенными символами. Если это значение установлено в -1, то считается, что строка заканчивается нулевым символом (NULL). wszPath – то место, куда будет сохранена сконвертированная строка, и MAX_PATH - максимальный размер пути до файла (256 символов).

После преобразования пути в Unicode, мы попытаемся загрузить изображение, используя OleLoadPicturePath. Если все хорошо, то pPicture будет указывать на данные изображения, и код результата будет сохранен в hr.

Если загрузка не пройдет, то программа прекратит свою работу.

```
 MultiByteToWideChar(CP_ACP, 0, szPath, -1, wszPath, MAX_PATH);
 // преобразуем к юникоду
 HRESULT hr = OleLoadPicturePath(wszPath, 0, 0, 0, IID_IPicture, (void**)&pPicture);

 if(FAILED(hr))  // Если загрузка не удачна
   return FALSE; // вернем False
```

Теперь мы должны создать временный контекст устройства. Если все пройдет нормально, hdcTemp будет содержать совместимый контекст устройства. Если программа не может получить совместимый контекст устройства, pPicture освобождается и программа вылетает (завершается).

```
 hdcTemp = CreateCompatibleDC(GetDC(0)); // создать совместимый с устройством Windows контекст
 if(!hdcTemp)            // ну что, создали?
 {                       // не-а… :(
   pPicture->Release();  // уменьшение счетчика ссылок на IPicture
   return FALSE;         // солгать
 }
```

Теперь пришло время допросить видеокарту, чтобы выяснить какой максимальный размер текстуры она поддерживает. Этот код важен, потому что это он будет стараться сделать изображение хорошего качества на всех видюхах. Он не только изменит размер текстуры, кратный степени 2. Он сделает так, чтобы ваш рисунок хорошо вписался в память видео платы. Это позволит вам загружать изображения любой ширины или высоты. Единственный недостаток состоит в том, что пользователи с плохими видео платами утратят много деталей при попытке просмотра изображения с высоким разрешением.

Что касается кода, мы используем glGetIntegerv (...), чтобы получить максимальное разрешение текстуры (256, 512, 1024, и т.д.) поддерживаемое конкретной видеокартой. Затем мы проверяем фактический размер изображения с помощью: pPicture-> get_width (&lwidth).

Мы используем причудливые вычисления, чтобы преобразовать ширину изображения в пиксели. Результат хранится в lwidthpixels. Тоже самое делаем с высотой. Мы получаем высоту изображения от pPicture и сохраняем значения пикселя в lheightpixels.

```
 // получить максимально возможное разрешение изображения
 glGetIntegerv(GL_MAX_TEXTURE_SIZE, &glMaxTexDim);

 pPicture->get_Width(&lWidth); // получить ширину изображения
 lWidthPixels  = MulDiv(lWidth, GetDeviceCaps(hdcTemp, LOGPIXELSX), 2540);
 pPicture->get_Height(&lHeight); // получить высоту изображения
 lHeightPixels  = MulDiv(lHeight, GetDeviceCaps(hdcTemp, LOGPIXELSY), 2540);
```

Затем мы проверяем, что ширина изображения в пикселях меньше чем максимальная ширина, поддерживаемая видео платой.

Если ширина изображения в пикселях - меньше чем максимальная поддерживаемая ширина, мы делаем размер изображения кратной степени двойки, исходя из текущей ширины изображения в пикселях. Мы добавляем 0.5f, чтобы изображение всегда было больше, если его размер близок к следующему размеру. Например, если наша ширина изображения была 400, и видео плата поддержала максимальную ширину 512, то было бы лучше сделать ширину 512. Если бы мы сделали ширину 256, то изображение бы многое утратило, но подробнее об этом позже.

Если размер изображения больше максимального, поддерживаемого нашей видео платой, то мы делаем размер изображения максимально возможным.

То же самое повторяем и для высоты. В итоге ширина и высота изображения будут сохранены в lwidthpixels и lheightpixels.

```
 // преобразовать изображение к ближайшей степени двойки
 if (lWidthPixels <= glMaxTexDim)
 // если ширина изображения меньше либо равна максимально-допустимому пределу карточки
   lWidthPixels = 1 << (int)floor((log((double)lWidthPixels)/log(2.0f)) + 0.5f);
 else
```

```
  // иначе установить размер равный максимальной степени двойки,
  // которую поддерживает карточка
  lWidthPixels = glMaxTexDim;
 // то же самое повторяется для высоты
 if (lHeightPixels <= glMaxTexDim)
  lHeightPixels = 1 << (int)floor((log((double)lHeightPixels)/log(2.0f)) + 0.5f);
 else
  lHeightPixels = glMaxTexDim;
```

Теперь, когда мы загрузили данные изображения, и знаем высоту и ширину, которая нам нужна, мы должны создать временный растр. Переменная bi будет содержать заголовочную информацию растра, а pBits будет хранить фактические данные изображения. Мы хотим создать 32-х битный растр с шириной lwidthpixels и высотой lheightpixels., а также, чтобы изображения было в формате RGB, и чтобы изображение имело одну битовую плоскость.

```
 // создать временный растр
 BITMAPINFO  bi = {0}; // нужный нам тип растра
 DWORD    *pBits = 0;  // указатель на биты растра

 bi.bmiHeader.biSize     = sizeof(BITMAPINFOHEADER); // размер структуры
 bi.bmiHeader.biBitCount = 32; // 32 бита
 bi.bmiHeader.biWidth    = lWidthPixels;  // ширина кратная степени двойки
 // Сделаем изображение расположенным вверх (положительное направление оси Y)
 bi.bmiHeader.biHeight   = lHeightPixels;
 bi.bmiHeader.biCompression  = BI_RGB;    // RGB формат
 bi.bmiHeader.biPlanes    = 1;           // 1 битовая плоскость
```

MSDN сообщает: функция CreateDIBSection создает DIB (независимый от устройства растр), куда программы могут писать напрямую. Функция возвращает вам указатель на память, где хранятся битовые значения растра. Вы можете позволить системе выделить память для растра.

hdcTemp - наш временный контекст устройства. bi - наша инфа о растре (информация заголовка). DIB_RGB_COLORS говорит нашей программе: “Мы хотим хранить RGB данные, а не индексы в логической палитре!” (каждый пиксель будут иметь красное, зеленое и синее значения).

pBits - то, куда данные изображения будут сохранены (указывает на данные изображения). Последние два параметра можно опустить.

Если программа не смогла создать временный растр, мы “заметаем следы” (чистим память) и возвращаем ЛОЖЬ, чтобы завершилась программа.

Если все тип-топ, то мы сводим счеты с временным растром. Мы используем SelectObject, чтобы прикрутить растр к временному контексту устройства.

```
 // создавая растр, таким образом, мы можем установить глубину цвета,
 // а также получить прямой доступ к битам.
 hbmpTemp = CreateDIBSection (hdcTemp, &bi, DIB_RGB_COLORS, (void**)&pBits, 0, 0);

 if(!hbmpTemp)        // создали?
 {                    // сам вижу что нет
  DeleteDC(hdcTemp);  // убить контекст устройства
 pPicture->Release(); // уменьшить счетчик количества интерфейсов IPicture
   return FALSE;      // вернуть ЛОЖЬ
 }
 // есть растр!
 SelectObject(hdcTemp, hbmpTemp); // загрузить описатель временного растра
                                  // в описатель временного контекста устройства
```

Теперь нам надо заполнить наш растр данными нашего изображения. pPicture->Render сделает это для нас. При этом также изменятся размеры изображения к любому размеру, который мы пожелаем (в этом случае lwidthpixels на lheightpixels).

hdcTemp - наш временный контекст устройства. Первые два параметра после hdcTemp - горизонтальное и вертикальное смещение (число пустых пикселей слева и сверху). Мы хотим, чтобы изображение заполнило растр полностью, так что мы выбираем 0 для горизонтального смещения и 0 для вертикального смещения.

Четвертый параметр - горизонтальное разрешение растра, а пятый параметр - вертикальное разрешение. Эти параметры контролируют, насколько изображение растянуто или сжато, чтобы соответствовать разрешению, которое мы хотим.

Следующий параметр (равный нулю) - горизонтальное смещение, от которого мы хотим читать исходные данные. Мы рисуем слева направо, так что смещение равно нулю. Это станет понятным после того, как вы увидите, что мы делаем с вертикальным смещением (будем надеяться, по крайней мере).

Параметр lHeight - вертикальное смещение. Мы будем читать данные снизу исходного изображения до верха, используя смещение lHeight, мы начинам с самого нижнего края исходного изображения.

lWidth – количество бит, которые нужно скопировать из исходного изображения. Мы хотим копировать все данные в горизонтальном направлении из исходного изображения. lWidth охватывает все данные слева направо.

Предпоследний параметр немного отличается от других. У него отрицательное значение. (lHeight, чтобы быть точным). Это значит, что мы хотим копировать все данные вертикально снизу до верху. Таким образом, изображение будет перевернуто и скопировано в результирующий растр.

Последний параметр не используется.

```
// отрисовка IPicture в растр
pPicture->Render(hdcTemp, 0, 0, lWidthPixels, lHeightPixels, 0, lHeight, lWidth, -lHeight, 0);
```

Теперь у нас есть растр с шириной lWidthpixels и высотой lHeightpixels. Новый растр был зеркально перевернут.

К сожалению, данные сохранены в формате BGR. Так что мы должны поменять красные и синие пиксели, чтобы создать растр и изображение RGB. В то же самое время, мы устанавливаем значение альфа-канала в 255. Вы можете изменить это значение на что угодно.

```
// преобразовать из BGR в RGB формат и устанавливаем значение
// Alpha = 255
for(long i = 0; i < lWidthPixels * lHeightPixels; i++) // Цикл по всем пикселям
{
  BYTE* pPixel  = (BYTE*)(&pBits[i]); // берем текущий пиксель
  BYTE  temp  = pPixel[0];            // сохраняем первый цвет в  переменной
                                      // Temp (Синий)
  pPixel[0]  = pPixel[2];             // ставим  Красный на место (в первую позицию)
  pPixel[2]  = temp;                  // ставим значение Temp в третий параметр (3rd)
  pPixel[3]  = 255;                   // установить значение alpha =255
}
```

Наконец, после всей этой работы, у нас есть растр, который может использоваться как текстура. Мы связываем его с texid, и генерируем текстуру. Мы хотим использовать линейную фильтрацию для min и mag фильтров (классно смотрится).

Мы получаем данные изображения от pBits. При создании текстуры, мы используем lwidthpixels и lheightpixels в последний раз, чтобы установить ширину и высоту текстуры.

После того, как 2D текстура была сгенерирована, мы можем освободить память. Нам больше не нужен ни временный растр, ни временный контекст устройства. Удаляем их. А еще мы можем освободить pPicture! Eyyy! :)))

```
glGenTextures(1, &texid); //  создаем текстуру
// типичная генерация текстуры, используя данные из растра
glBindTexture(GL_TEXTURE_2D, texid); // делаем привязку к texid
// (измените для нужного вам типа фильтрации)
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
// (измените, если хотите использовать мипмап-фильтрацию)
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
```

```
  // (мипмап - множественное отображение (последовательность текстур одного
  //  и того же изображения с уменьшающимся разрешением по мере удаления отображаемого
  // объекта от наблюдателя))

  glTexImage2D(GL_TEXTURE_2D, 0, 3, lWidthPixels, lHeightPixels, 0, GL_RGBA,
            GL_UNSIGNED_BYTE, pBits);

  DeleteObject(hbmpTemp); // удаляем объект
  DeleteDC(hdcTemp);      // удаляем контекст устройства
  pPicture->Release();    //уменьшает счетчик IPicture
  return TRUE;            // вернуть ПРАВДУ (все ОК)
}
```

Следующий код проверяет, поддерживает ли пользовательская видео плата расширение EXT_FOG_coord. Этот код может вызываться ТОЛЬКО после того, как ваша OpenGL программа имеет контекст для визуализации. Если Вы попробуете вызвать это раньше, чем вы создадите окно, то у вас будут ошибки.

Первое, что мы делаем - создаем строку с именем нашего расширения.

Затем мы выделяем достаточное количество памяти, чтобы хранить список OpenGL расширений, поддерживаемых пользовательской видеоплатой. Список поддерживаемых расширений - извлекается командой glGetString (GL_EXTENSIONS). Возвращенная информация копируется в glextstring.

Как только мы получаем список поддерживаемых расширений, мы используем strstr, чтобы определить, находится ли наше расширение (Extension_Name) в списке поддерживаемых расширений (glextstring).

Если расширение не поддерживается, возвращаем FALSE, и программа завершается. Если все идет хорошо, мы освобождаем glextstring (мы больше не нуждаемся в списке поддерживаемых расширений).

```
int Extension_Init()
{
  char Extension_Name[] = "EXT_fog_coord";
  // выделяем память для строки расширения
  char*glextstring=(char*)malloc(strlen((char*)glGetString(GL_EXTENSIONS))+1);
  // копируем список расширений в glextstring
  strcpy (glextstring,(char *)glGetString(GL_EXTENSIONS));
  if (!strstr(glextstring,Extension_Name)) // поддерживается ли расширение??
    return FALSE; // если нет – то вернуть FALSE
  free(glextstring); // иначе освобождаем память
```

В самом начале этой программы мы определили glFogCoordfExt. Однако команда не будет работать, пока мы не присоединим функцию к фактическому OpenGL расширению. Мы делаем это, передавая glFogCoordfExt адрес расширения тумана OpenGL. Когда мы вызываем glFogCoordfExt, фактический код расширения выполнится, и мы получим параметр, переданный glFogCoordfExt.

Мне приходиться с сожалением признать, что это та часть кода, которую сложно описать в простых терминах (по крайней мере, для меня).

```
  // устанавливаем и активируем glFogCoordEXT
  glFogCoordfEXT = (PFNGLFOGCOORDFEXTPROC) wglGetProcAddress("glFogCoordfEXT");
  return TRUE;
}
```

В коде функции инициализации мы, прежде всего, проверяем, поддерживается ли расширение, загружаем нашу текстуру, и настраиваем OpenGL.

До этого наша программа должна иметь RC (контекст рендеринга). Это важно, потому что вы должны иметь контекст рендеринга прежде, чем вы можете проверить, поддерживаться ли расширение конкретной видеоплатой.

Поэтому мы вызываем Extension_Init () чтобы определить, поддерживает ли плата расширение. Если расширение не поддерживается, Extension_Init () возвращает ложь и это приводит к завершению программы. Если вам нужно отобразить какое-либо сообщение, вы можете это сделать. Сейчас программа просто не запуститься.

Если расширение поддерживается, мы попытаемся загрузить нашу текстуру wall.bmp. Идентификатором для этой текстуры будет texture[0]. Если по каким-либо причинам текстура не загружается, программа завершится.

Инициализация проста. Мы активизируем двухмерное наложение текстуры и задаем черный цвет очистки экрана. Очищаем буфер глубин 1.0f. Мы задаем тест глубины типа «меньше или равно» и разрешаем тест глубины. Модель закрашивания устанавливаем равным плавному закрашиванию, и мы выбираем наилучшую коррекцию перспективы.

```
BOOL Initialize (GL_Window* window, Keys* keys)
    // код инициализации
{
 g_window  = window; // параметры окна
 g_keys    = keys;   // ключевые значения

  // начало пользовательской инициализации
 // проверяем и активизируем расширение тумана, если оно доступно
 if (!Extension_Init())
   return FALSE; // вернуть False если расширение не поддерживается

 if (!BuildTexture("data/wall.bmp", texture[0])) // загрузить текстуру стены
   return FALSE; // вернуть False если загрузка не прошла

 glEnable(GL_TEXTURE_2D);   // вкл отображение текстуры
 glClearColor (0.0f, 0.0f, 0.0f, 0.5f); // черный фон
 glClearDepth (1.0f);       // установить глубину буфера
 glDepthFunc (GL_LEQUAL);   // тип теста глубины
 glEnable (GL_DEPTH_TEST);  // вкл тест глубины
 glShadeModel (GL_SMOOTH);  // выбрать плавное закрашивание
 // установить вычисления перспективы к максимально точными
 glHint (GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
```

Ну, а, теперь можно по-прикалываться. Мы должны задать туман. Начинаем с активизации тумана. Режим визуализации, который мы используем, линейный. Цвет тумана установлен с помощью fogColor (оранжевый).

Затем мы должны установить стартовую позицию тумана. Это - наименее плотная часть тумана. Для простоты, мы будем использовать 1.0f как наименьшее значение плотности (FOG_START). Мы будем использовать 0.0f как наибольшее значение плотности тумана (FOG_END).

Согласно документации, которую я прочел, настраивая туман, используя GL_NICEST, он будет отображаться по-пиксельно. При использовании GL_FASTEST туман будет выводиться по вершинам. Лично я не вижу никакой разницы.

Последняя команда glFogi (...) сообщает OpenGL, что мы хотим установить наш туман, используя координаты вершин. Это позволяет устанавливать туман в любом месте нашей сцены ни влияя на всю сцену (улет!).

Мы устанавливаем начальное значение camz в -19.0f. Коридор на самом деле имеет длину 30 единиц. Поэтому -19.0f перемещает нас, почти в начало коридора (коридор визуализируется от -15.0f до +15.0f по оси Z).

```
 // устанавливаем туман
 glEnable(GL_FOG);               // активизируем туман
 glFogi(GL_FOG_MODE, GL_LINEAR);  // затенение линейно
 glFogfv(GL_FOG_COLOR, fogColor); // устанавливаем цвет тумана
 glFogf(GL_FOG_START,  0.0f);     // устанавливаем начальную плотность тумана
 glFogf(GL_FOG_END,    1.0f);     // устанавливаем конечную плотность тумана
 glHint(GL_FOG_HINT, GL_NICEST);  // вычисляем туман по пикселям

 // делаем туман на координатах вершин
 glFogi(GL_FOG_COORDINATE_SOURCE_EXT, GL_FOG_COORDINATE_EXT);

 camz =  -19.0f; // устанавливаем значение камеры по оси Z = -19.0f
 return TRUE;    // возвращаем TRUE (инициализация прошла успешно)
}
```

Эта часть кода вызывается всякий раз, когда пользователь выходит из программы. Мусора у нас нет, поэтому и кода нет :-).

```
void Deinitialize (void)              // код выхода
{
}
```

Тут мы обрабатываем клавиатуру. Подобно всем предыдущим урокам, мы проверяем, нажата ли кнопка ESC. Если да, то приложение закрывается. Если нажата клавиша F1, мы переключаемся из полноэкранного режима в оконный режим или из оконного в полноэкранный.

Другие две кнопки, которые мы проверяем – это стрелки «вверх» и «вниз». Если кнопка «вверх» нажата, и значение camz - меньше чем 14.0f, мы увеличиваем camz. При этом наблюдатель будет перемещаться вперед по коридору. Если бы значение было больше 14.0f, мы бы прошли через заднюю стену. Но мы не хотим, чтобы это произошло :).

Если нажата кнопка «вниз», и значение camz большее чем -19.0f, мы уменьшаем camz. При этом наблюдатель будет перемещаться назад по коридору. Если бы значение было меньше -19.0f, коридор был бы слишком далек от экрана, и виден был бы только вход в коридор. Опять же... Это было бы не хорошо!

Значение camz увеличивается и уменьшается на число прошедших миллисекунд, деленных на 100.0f. Это должно заставить программу выполняться с одинаковой скоростью на всех типах процессоров.

```
void Update (DWORD milliseconds)     // выполняем обновление движения
{
  if (g_keys->keyDown [VK_ESCAPE])   // ESC нажата?
    TerminateApplication (g_window); // прекращаем работу

  if (g_keys->keyDown [VK_F1])   //  F1 нажата?
    ToggleFullscreen (g_window); // переход в полноэкранный режим или в оконный

  if (g_keys->keyDown [VK_UP] && camz<14.0f) // стрелка вверх нажата?
    camz+=(float)(milliseconds)/100.0f;     // приближаем объект

  if (g_keys->keyDown [VK_DOWN] && camz>-19.0f) // стрелка вниз нажата?
    camz-=(float)(milliseconds)/100.0f;         // отдаляем объект
}
```

Я уверен, что вы готовы умереть, чтобы получить визуализацию, но нам надо кое-что еще сделать, чтобы отобразить коридор. Сначала мы должны очистить экран и буфер глубины. Мы сбрасываем матрицу вида модели и сдвигаем камеру на значение camz.

Увеличивая или уменьшая значение camz, коридор станет ближе или дальше от наблюдателя. Это создаст впечатление, что наблюдатель двигается вперед или назад через коридор. Все гениальное просто!

```
void Draw (void)
{
  glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // очищаем экран и буфер глубины
  glLoadIdentity (); // сбрасываем матрицу вида модели
  glTranslatef(0.0f, 0.0f, camz); // устанавливаем Z координату камеры
```

Камера установлена, теперь пришло время визуализировать первый квадрат. Это будет задняя стенка (стенка в конце коридора).

Мы хотим, чтобы эта стенка была в самом плотном слое тумана. Если Вы взглянете на код инициализации, вы увидите, что GL_FOG_END - наиболее плотная часть тумана... и значение его 1.0f.

Туман устанавливается тем же самым способом, каким вы управляете координатами текстуры. Значение GL_FOG_END является самым плотным туманом и имеет значение 1.0f. Поэтому для вершин нашего первого полигона мы передаем в glFogCoordfExt значение 1.0f. Этот полигон задает самую дальнюю стену (стену, которую вы будете наблюдать в конце тоннеля) координаты по осям X и Y равны -2.5f. Там будет наиболее плотный туман.

```
glBegin(GL_QUADS); // задняя стена
  glFogCoordfEXT(1.0f); glTexCoord2f(0.0f, 0.0f); glVertex3f(-2.5f,-2.5f,-15.0f);
  glFogCoordfEXT(1.0f); glTexCoord2f(1.0f, 0.0f); glVertex3f( 2.5f,-2.5f,-15.0f);
  glFogCoordfEXT(1.0f); glTexCoord2f(1.0f, 1.0f); glVertex3f( 2.5f, 2.5f,-15.0f);
  glFogCoordfEXT(1.0f); glTexCoord2f(0.0f, 1.0f); glVertex3f(-2.5f, 2.5f,-15.0f);
glEnd();
```

Теперь у нас имеется задняя стена с наложенной на нее текстурой в плотном тумане. Нарисуем пол. Тут есть небольшое отличие, но как только вы поймете принцип, вам все станет ясно.

Подобно всем квадратам, пол имеет 4 точки. Значение Y всегда равно -2.5f. Значение по оси X левой вершины равно -2.5f, а правой вершины -2.5f, и пол рисуется по оси Z от -15.0f до +15.0f.

Нам нужно, чтобы дальняя часть пола находилась в тумане. Снова присвоим вершинам glFogCoordfExt значение 1.0f. Заметьте, что любая вершина, нарисованная со значением по оси Z равным -15.0f имеет значение glFogCoordfExt равным 1.0f...?!

Часть пола, ближайшая к наблюдателю (+15.0f) будет иметь наименьшую плотность тумана. GL_START_FOG - наименее плотный туман и имеет значение 0.0f. Так что, для этих точек присвоим переменой glFogCoordfExt значение 0.0f.

То, что вы должны видеть, когда запустите программу, это на самом деле плотный туман на полу возле задней части и светлый туман спереди. Туман недостаточно густой, чтобы заполнить весь коридор. В действительности, он рассеивается где-то в середине коридора, даже если GL_START_FOG = 0.0f.

```
glBegin(GL_QUADS); // пол
  glFogCoordfEXT(1.0f); glTexCoord2f(0.0f, 0.0f); glVertex3f(-2.5f,-2.5f,-15.0f);
  glFogCoordfEXT(1.0f); glTexCoord2f(1.0f, 0.0f); glVertex3f( 2.5f,-2.5f,-15.0f);
  glFogCoordfEXT(0.0f); glTexCoord2f(1.0f, 1.0f); glVertex3f( 2.5f,-2.5f, 15.0f);
  glFogCoordfEXT(0.0f); glTexCoord2f(0.0f, 1.0f); glVertex3f(-2.5f,-2.5f, 15.0f);
glEnd();
```

Потолок выводится так же, как и пол, с единственным отличием в том, что потолок рисуется по оси Y со значением координаты равным 2.5f.

```
glBegin(GL_QUADS); // потолок
  glFogCoordfEXT(1.0f); glTexCoord2f(0.0f, 0.0f); glVertex3f(-2.5f, 2.5f,-15.0f);
  glFogCoordfEXT(1.0f); glTexCoord2f(1.0f, 0.0f); glVertex3f( 2.5f, 2.5f,-15.0f);
  glFogCoordfEXT(0.0f); glTexCoord2f(1.0f, 1.0f); glVertex3f( 2.5f, 2.5f, 15.0f);
  glFogCoordfEXT(0.0f); glTexCoord2f(0.0f, 1.0f); glVertex3f(-2.5f, 2.5f, 15.0f);
glEnd();
```

Правая стенка рисуется точно также. Только по оси X значение всегда 2.5f. Самые дальние точки на оси Z по-прежнему устанавливаются равными glFogCoordfExt (1.0f), а самые близкие по оси Z равны glFogCoordfExt (0.0f).

```
glBegin(GL_QUADS); // правая стена
  glFogCoordfEXT(0.0f); glTexCoord2f(0.0f, 0.0f); glVertex3f( 2.5f,-2.5f, 15.0f);
  glFogCoordfEXT(0.0f); glTexCoord2f(0.0f, 1.0f); glVertex3f( 2.5f, 2.5f, 15.0f);
  glFogCoordfEXT(1.0f); glTexCoord2f(1.0f, 1.0f); glVertex3f( 2.5f, 2.5f,-15.0f);
  glFogCoordfEXT(1.0f); glTexCoord2f(1.0f, 0.0f); glVertex3f( 2.5f,-2.5f,-15.0f);
glEnd();
```

Будем надеяться, теперь вы понимаете, как все это работает. Все что находится на расстоянии должно иметь значение 1.0f. Все что ближе - должно быть равно 0.0f.
В любом случае, вы можете поэкспериментировать со значениями GL_FOG_START и GL_FOG_END, чтобы увидеть, как они влияют на сцену.
Эффект не выглядит убедительным, если Вы поменяете начальные и конечные значения. Иллюзия создается задней стеной, которая является полностью оранжевой! Эффект смотрится лучше всего в тупиках или узких углах, где игрок не может отвернуться от тумана!
Этот тип тумана смотрится лучше всего, когда игрок может видеть комнату с туманом, но фактически не может войти в нее. Хорошим примером может быть глубокая яма, закрытая чем-то наподобие решетки. Игрок может смотреть вниз, но не может попасть туда.

```
  glBegin(GL_QUADS); // левая стена
    glFogCoordfEXT(0.0f); glTexCoord2f(0.0f, 0.0f); glVertex3f(-2.5f,-2.5f, 15.0f);
    glFogCoordfEXT(0.0f); glTexCoord2f(0.0f, 1.0f); glVertex3f(-2.5f, 2.5f, 15.0f);
    glFogCoordfEXT(1.0f); glTexCoord2f(1.0f, 1.0f); glVertex3f(-2.5f, 2.5f,-15.0f);
    glFogCoordfEXT(1.0f); glTexCoord2f(1.0f, 0.0f); glVertex3f(-2.5f,-2.5f,-15.0f);
  glEnd();
  glFlush ();
}
```

Я надеюсь, вам понравился этот урок. Он был создан в течение 3 дней... по 4 часа в день. Большинство времени было потрачено на написание текста, который Вы в данный момент читаете.

Я хотел сделать трехмерную комнату с туманом в одном углу. К сожалению, у меня, было, очень мало времени, чтобы поработать с кодом.

Даже при том, что коридор в этом уроке очень прост, фактический эффект тумана довольно крут! Чтобы использовать данный эффект в ваших собственных проектах потребуется совсем немного изменений.

Эти урок показывает Вам, как использовать glFogCoordfExt. Это быстро, круто и легко! Стоит обратить внимание, что это - только ОДИН из многих различных способов создать объемный туман. Тот же самый эффект может быть создан, используя смешивание, частицы, маски, и т.д.

Как всегда... Если вы нашли ошибки в этом уроке, сообщите мне. Если Вы считаете, что Вы можете описать часть кода лучше (моя формулировка не всегда понятна), намыльте мне :) !

Много текста было написано поздно ночью, и хотя это не оправдание, но мой текст становится от этого немного хуже, поскольку я сонный в это время. Пожалуйста, присылайте мне по электронной почте письма, если вы нашли в тексте повторяющиеся слова, ошибки в правописании и т.п.

Первоначальная идея для этого урока была послана мне довольно давно. С тех пор я потерял это письмо. Тому, кто послал мне это письмо хочу сказать … спасибо!

Jeff Molofee (NeHe)

Урок 43. FreeType шрифты в OpenGL.

FreeType Fonts in OpenGL

Итак, этот небольшой урок покажет вам, как использовать с OpenGL библиотеку шрифтов FreeType. Используя библиотеку FreeType мы можем создавать текст со сглаживанием краев, который выглядит намного лучше, чем текст сделанный с использованием растровых шрифтов - также нам будет легче поворачивать его и работать с функциями выбора объектов.

Мотивация

Сегодня мы будем печатать текст, используя одновременно и растровые шрифты WGL и шрифты сделанные с помощью FreeType (оба Arial Black Italic).

Основная проблема состоит в том, что используемые растровые шрифты в OpenGL по определению бинарные изображения. Смысл этого в том, что растровые изображения в OpenGL имеют только один бит на один пиксель. Если вы будете увеличивать текст, используя WGL, результат будет похож на этот:

Поскольку растры бинарные, в них нет оттенков серого цвета, они показывают только текст.

К счастью очень легко сделать прилично смотрящиеся шрифты, используя GNU FreeType library. Кстати FreeType использовали в Blizzard для вывода шрифтов в их играх, так что вы знаете всю прелесть этой библиотеки.

Вот текст, который я создал при помощи FreeType Library.

Вы можете видеть, что здесь есть оттенки серого цвета на углах текста; это типичный признак шрифта со сглаживанием, оттенки серого цвета делают текст гладким в независимости от расстояния.

Создание программы.

Первый шаг, который мы должны сделать, это получить копию GNUFreeType library. Идите по адресу http://gnuwin32.sourceforge.net/packages/freetype.htm и загрузите бинарные и файлы для разработчиков. Когда вы установите ее, прочитайте лицензионное соглашение, где написано, что если вы используете FreeType в вашей программе, вы должны отметить разработчиков, где-нибудь в вашей документации.
Сейчас нам нужен MSVC, чтобы использовать FreeType. Итак, создайте новый проект как показано в первом уроке, но когда нажмете Project->Setting->Link будьте, уверены, что вы добавили libfreetype.lib в Object Modules /libraries вместе с opengl32.lib, glu32.lib и glaux.lib (если нужно).
Следующее что нам нужно это добавить директорию FreeType library в Tools->Options->Directories. Под "Show Directories For" выберите "Include Files", затем два раза кликните на пустой линии вверху листа каталогов, после вашего щелчка по кнопке "..." появится окно, в котором вы можете выбрать директорию. В этом случае добавьте C:\PROGRAM FILES\GNUWIN32\INCLUDE\FREETYPE2 и C:\PROGRAM FILES\GNUWIN32\INCLUDE. В список используемых директорий. Теперь под "Show Directories For" выберите "Library Files", и добавьте C:\PROGRAM FILES\GNUWIN32\LIB.
В этой точке мы должны быть готовы к компиляции программ используя FreeType, но они не захотят запускать пока не получат доступа к freetype-6.dll. Эта библиотека лежим в каталоге GNUWIN32\BIN, и если вы запихнете ее, куда-нибудь где все ваши программы смогут видеть её (Program Files\Microsoft Visual Studio\VC98\Bin хороший вариант), вы сможете запускать программы использующие FreeType. Но помните то, что если вы будете распространять программу, которая использует FreeType, вы должны будете также распространять копии этой DLL с ней.
Наконец то, теперь мы можем начать писать код. Я решил работать с уроком 13, поэтому скачайте пример к этому уроку, если у вас нет его. Скопируйте lesson13.cpp в директорию вашего проекта и добавьте файл в проект.
Сейчас добавьте и создайте два новых файла:"freetype.cpp" и "freetype.h". Мы поместим все наши функции, которые специфичны для FreeType в эти файлы, и тогда немного изменим lesson13.cpp , чтобы показать написанные функции. Когда мы закончим, у нас будет созданное очень простое приложение OpenGL FreeType library, которое теоретически может быть использовано в любом OpenGL проекте.

Начнём с freetype.h.

Обычно сначала мы подключаем заголовочные файлы FreeType и OpenGL. Также мы подключим некоторые части Standart Template Library, включая STLовские классы обработчики прерываний, которые сделают отладку приложения проще.

```
#ifndef FREE_NEHE_H
#define FREE_NEHE_H

// FreeType заголовочные файлы
#include <ft2build.h>
#include <freetype/freetype.h>
#include <freetype/ftglyph.h>
#include <freetype/ftoutln.h>
#include <freetype/fttrigon.h>

// OpenGL заголовочные файлы
#include <windows.h>                // (GL'у это нужно)
#include <GL/gl.h>
#include <GL/glu.h>

// Некоторые заголовки STL
#include <vector>
#include <string>

// Использование STL библиотеки исключений увеличивает шансы
// на то, что другой человек будет корректно отлавливать посылаемые нами исключения.
#include <stdexcept>

// MSVC будет выплевывать все сорта бесполезных предупреждений, если
// вы создаете векторы строк, эта pragma отключает их
#pragma warning(disable: 4786)
```

Мы поместим информацию нужную каждому шрифту в структуру (это облегчит управление несколькими шрифтами). Как мы видели в уроке 13, при создании шрифта с WGL генерировался набор последовательных списков отображения. Это изящно, потому что мы можем вызывать glCallLists чтобы напечать текст, при помощи лишь одной команды. Когда мы создаем шрифт, мы делаем аналогично, что обозначает то, что list_base поле будет хранить первые 128 списков отображения. Поскольку нам надо использовать текстуры для прорисовки текста, нам также нужно хранилище для 128 связанных текстур. Последний кусок информации это высота в пикселях шрифта, который будет создан (это сделает возможным обработку символов перевода строк в нашей функции печати).

```
//Помещая все в пространство имен, мы можем, не беспокоится о конфликте с чужим
// кодом такой распространенной функцией как print
namespace freetype {

// В этом пространстве, даем себе возможность написать только "vector" вместо "std::vector"
using std::vector;

// тоже самое для строки.
using std::string;

// Здесь мы храним всю информацию о FreeType шрифте, который мы хотим создать
struct font_data {
  float h;                  // Высота
  GLuint * textures;          // Идентификатор
  GLuint list_base;           // Содержит указатель на список отображения

  // Функция инициализации создаст шрифт с
  // высотой h из файла fname
  void init(const char * fname, unsigned int h);

  // Освобождаем ресурсы связанные со шрифтом
  void clean();
};
```

Последнее что нам нужно это прототип функции печати

```
// Главная функция библиотеки -  она будет печатать
// текст в окне по координатам X,Y используя Font ft_font.
// Текущая матрица вида модели также будет применена к тексту

void print(const font_data &ft_font, float x, float y, const char *fmt, ...);

}                       // Закрываем пространство имен
#endif
```

А вот и конец заголовочного файла! Время редактировать freetype.cpp.

```
// Включаем заголовочные файлы
#include "freetype.h"

namespace freetype {
```

Мы используем текстуры, чтобы отобразить каждый символ в нашем шрифте. OpenGL текстуры должны иметь размеры, которые являются степенью двойки, так что мы должны преобразовать наши растры шрифтов сделанные FreeType в размер такого типа. Для этого нам нужна следующая функция:

```
// Эта функция возвращает число в степени два, большее, чем число a
inline int next_p2 (int a )
{
  int rval=1;
  // rval<<=1 это лучше чем rval*=2;
  while(rval<a) rval<<=1;
  return rval;
}
```

Следующее что нам нужно сделать это функцию make_dlist, это действительно самая главная функция. Она использует FT_Face, который является объектом, используемым FreeType для сохранения информации о шрифте, и создания списка отображения, который отвечает за каждый символ.

```
// Создает список отображения на базе данного символа
void make_dlist ( FT_Face face, char ch, GLuint list_base, GLuint * tex_base ) {

  // Первая вещь, которую нам надо сделать, это вывести наш символ
  // в растр. Это делается набором команд FreeType
  // Загрузить глифы для каждого символа.

  if(FT_Load_Glyph( face, FT_Get_Char_Index( face, ch ), FT_LOAD_DEFAULT ))
    throw std::runtime_error("FT_Load_Glyph failed");

  // Поместить глиф в объект.
  FT_Glyph glyph;
  if(FT_Get_Glyph( face->glyph, &glyph ))
    throw std::runtime_error("FT_Get_Glyph failed");

  // Конвертировать глиф в растр.
  FT_Glyph_To_Bitmap( &glyph, ft_render_mode_normal, 0, 1 );
  FT_BitmapGlyph bitmap_glyph = (FT_BitmapGlyph)glyph;

  // С помощью этой ссылки, получаем легкий доступ до растра.
  FT_Bitmap& bitmap=bitmap_glyph->bitmap;
```

Примечание:

"Символы - объекты, назначенные по стандарту Unicode, которые представляют самые маленькие семантические модули языка. Глифы - определенные формы, которыми символы могут быть представлены. Один символ может прописываться как несколько глифов: нижний регистр "a", капительная буква "a" и специальный символ "a" - это три отдельных глифа. Один глиф может также представлять многие символы, как в случае "ffi" связи, который соответствует последовательности трех символов: f, f и i. Для любого символа имеется заданный по умолчанию глиф и позиционированные данные."

Теперь, когда у нас есть растр созданный с помощью FreeType, нам нужно его наложить на текстуру OpenGL. Важно помнить, что пока OpenGL использует термин "растр" это подразумевает двоичные рисунки, в FreeType растры сохраняют 8 битов информации на пиксель, так что FreeType'ские растры смогу сохранять оттенки серого, которые нам нужны для сглаживания.

```
  // Используем нашу вспомогательную функцию для вычисления ширины и высоты
  // текстуры для нашего растра.
  int width = next_p2( bitmap.width );
  int height = next_p2( bitmap.rows );

  // Выделим память для данных текстуры.
  GLubyte* expanded_data = new GLubyte[ 2 * width * height];

  // Поместим данные в расширенный растр.
  // Отмечу, что использован двухканальный растр (Один для
  // канала яркости и один для альфа), но мы будем назначать
  // обоим каналам одно и тоже значение, которое мы
  // получим из растра FreeType.
  // Мы используем оператор ?: для того чтобы поместить 0 в зону вне растра FreeType.
  for(int j=0; j <height;j++) {
    for(int i=0; i < width; i++){
      expanded_data[2*(i+j*width)]= expanded_data[2*(i+j*width)+1] =
        (i>=bitmap.width || j>=bitmap.rows) ?
        0 : bitmap.buffer[i + bitmap.width*j];
    }
  }
```

Когда заполнение закончится, мы сможем создать текстуру OpenGL. Мы включим альфа канал, таким образом, черные куски растра будут прозрачными, а остальные слегка прозрачные (что должно сделать вывод шрифта правильным на любом фоне).

```
  // Теперь мы только устанавливаем параметры
  glBindTexture( GL_TEXTURE_2D, tex_base[ch]);
  glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);

  // Здесь мы создаем текстуру
  // Помните, что используем GL_LUMINANCE_ALPHA, чтобы было два альфа канала данных

  glTexImage2D( GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0,
    GL_LUMINANCE_ALPHA, GL_UNSIGNED_BYTE, expanded_data );

  // После создания текстуры, мы больше не нуждаемся в промежуточных данных.
  delete [] expanded_data;
```

Мы используем наложение текстуры на четырехугольники для прорисовки текста. Это обозначает то, что будет легко поворачивать и увеличивать/приближать текст, и мы также создадим шрифты с текущим цветом OpenGL (ни один из которых не будет использоваться в растрах).

```
  // Создать список отображения
  glNewList(list_base+ch,GL_COMPILE);
  glBindTexture(GL_TEXTURE_2D,tex_base[ch]);

  // Вначале мы сдвинем символ вправо на расстояние между ним и символам до него.
  glTranslatef(bitmap_glyph->left,0,0);

  // Сдвинем вниз в том случае, если растр уходит вниз строки.
  // Это истинно только для символов, таких как 'g' или 'y'.
  glPushMatrix();
  glTranslatef(0,bitmap_glyph->top-bitmap.rows,0);

  // Вычислим какая часть нашей текстуры будет заполнена пустым пространством.
  // Мы рисуем только ту часть текстуры, в которой находится символ, и сохраняем
  // информацию в переменных x и y, затем, когда мы рисуем четырехугольник,
  // мы будем только ссылаться на ту часть текстуры, в которой непосредственно
  // содержится символ.
  float   x=(float)bitmap.width / (float)width,
  y=(float)bitmap.rows / (float)height;

  // Рисуем текстурированный четырехугольник.
  glBegin(GL_QUADS);
  glTexCoord2d(0,0); glVertex2f(0,bitmap.rows);
  glTexCoord2d(0,y); glVertex2f(0,0);
  glTexCoord2d(x,y); glVertex2f(bitmap.width,0);
  glTexCoord2d(x,0); glVertex2f(bitmap.width,bitmap.rows);
  glEnd();
  glPopMatrix();
  glTranslatef(face->glyph->advance.x >> 6 ,0,0);

  // Увеличиваем позицию растра, как если бы это был растровый шрифт.
  // (Необходимо только, если вы хотите вычислить длину текста)
  // glBitmap(0,0,0,0,face->glyph->advance.x >> 6,0,NULL);

  // Завершим создание списка отображения
  glEndList();
}
```

Следующая функция, которую мы создадим, будет использовать make_dlist, для создания наборов списков отображения, соответствующих данному файлу шрифта и высоте пикселя.

FreeType использует шрифты TrueType, так что неплохо бы найти какой файл шрифта TrueType. Шрифты TrueType очень распространены, так что вы можете найти множество сайтов, где вы можете загрузить себе разные шрифты TrueType. Windows 98 используется такой тип для почти всех шрифтов, так что если вы сможете найти старый компьютер, использующий эту ОС, вы можете получить все стандартные шрифты формата TrueType в каталоге windows/fonts.

```
void font_data::init(const char * fname, unsigned int h) {
  // Выделим память для идентификаторов текстуры.
  textures = new GLuint[128];

  this->h=h;

  // Инициализация библиотеки FreeType.
  FT_Library library;
  if (FT_Init_FreeType( &library ))
    throw std::runtime_error("FT_Init_FreeType failed");

  // Объект для хранения шрифта.
  FT_Face face;

  // Загрузим шрифт из файла. Если файла шрифта не существует или шрифт битый,
  // то программа может умереть.
  if (FT_New_Face( library, fname, 0, &face ))
    throw std::runtime_error("FT_New_Face failed (there is probably a problem
                             with your font file)");

  // По некоторым причинам FreeType измеряет размер шрифта в терминах 1/64 пикселя.
  // Таким образом, для того чтобы сделать шрифт выстой h пикселей, мы запрашиваем размер h*64.
  // (h << 6 тоже самое что и h*64)
  FT_Set_Char_Size( face, h << 6, h << 6, 96, 96);

  // Здесь попросим OpenGL, чтобы он выделил память для
  // всех текстур и списков отображения, которые нам нужны.
  list_base=glGenLists(128);
  glGenTextures( 128, textures );

  // Создаем списки отображения шрифтов.
  for(unsigned char i=0;i<128;i++)
    make_dlist(face,i,list_base,textures);
  // Уничтожим шрифт.
  FT_Done_Face(face);

  // Не нужна и библиотека.
  FT_Done_FreeType(library);
}
```

Теперь нам нужна функция для удаления текстур и списков отображения связанных со шрифтом.

```
void font_data::clean() {
  glDeleteLists(list_base,128);
  glDeleteTextures(128,textures);
  delete [] textures;
}
```

Ещё у нас осталось две функции, которые нам надо сделать, перед тем как мы сделаем функцию отображения текста. В OpenGL есть две очень удобных функции, glGet возвращает размеры окна, и glPush/PopAttrib используется для сохранения состояния режимов OpenGL. Если вы незнакомы с этими функциями, вам лучше посмотреть их описание.

```
// Простая функция, в которой сохраняется матрица проекции,
// затем делаем мировые координаты идентичными с координатами окна.
inline void pushScreenCoordinateMatrix() {
  glPushAttrib(GL_TRANSFORM_BIT);
  GLint   viewport[4];
  glGetIntegerv(GL_VIEWPORT, viewport);
  glMatrixMode(GL_PROJECTION);
  glPushMatrix();
  glLoadIdentity();
  gluOrtho2D(viewport[0],viewport[2],viewport[1],viewport[3]);
  glPopAttrib();
}
// Восстановить координаты матрицы проекции.
inline void pop_projection_matrix() {
  glPushAttrib(GL_TRANSFORM_BIT);
  glMatrixMode(GL_PROJECTION);
  glPopMatrix();
  glPopAttrib();
}
```

Наша функция печати очень похожа на такую же из Урока 13, но есть несколько важных различий. Мы разрешаем флаги, которые отражают тот факт, что, мы используем двухканальные текстуры, а не растр. Нам также нужно выполнить немного дополнительной работы для обработки символов перевода на новую строку. Поскольку мы такие добрые самаритяне, мы также заботимся о сохранении и восстановлении внутренних стеков OpenGL.

```
            // Модифицируем функцию glPrint.
void print(const font_data &ft_font, float x, float y, const char *fmt, ...)  {
             // Мы хотим систему координат, в которой расстояние измеряется в пикселях.
  pushScreenCoordinateMatrix();
  GLuint font=ft_font.list_base;
             // Сделаем высоту немного больше, что бы оставить место между линиями.
  float h=ft_font.h/.63f;
    char  text[256];            // Сохраним нашу строку
    va_list  ap;                // Указатель на лист аргументов
 if (fmt == NULL)               // Если это не текст
    *text=0;                    // Тогда ничего не делать
  else {
    va_start(ap, fmt);          // Разбор строки на переменные
    vsprintf(text, fmt, ap);  // И конвертировать символы в числа
    va_end(ap);                 // Результат сохранить в текст
  }
    // Разделим текст на строки.
  const char *start_line=text;
  vector<string> lines;
  for(const char *c=text;*c;c++) {
    if(*c=='\n') {
      string line;
      for(const char *n=start_line;n<c;n++) line.append(1,*n);
      lines.push_back(line);
      start_line=c+1;
    }
  }
  if(start_line) {
    string line;
    for(const char *n=start_line;n<c;n++) line.append(1,*n);
   lines.push_back(line);
  }
  glPushAttrib(GL_LIST_BIT | GL_CURRENT_BIT  | GL_ENABLE_BIT | GL_TRANSFORM_BIT);
  glMatrixMode(GL_MODELVIEW);
  glDisable(GL_LIGHTING);               glEnable(GL_TEXTURE_2D);
  glDisable(GL_DEPTH_TEST);           glEnable(GL_BLEND);
  glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
 glListBase(font);
```

Поскольку мы используем текстурированные четырехугольники, любые преобразования, которые мы применяем к матрице вида модели до того как вызвать glCallLists будут влиять и на текст. Это значит, что есть возможность крутить или масштабировать наш текст (другое преимущество использования WGL растров). Самый хороший способ использовать это преимущество было бы оставить текущую матрицу вида в покое, что допускает любые преобразования, перед тем как функция print выведет текст. Но поскольку мы используем матрицу вида модели, чтобы установить позицию текста, это не сработает. Лучшим для нас было бы сохранить копию матрицу, и применить ее между glTranslate и glCallLists. Это просто сделать, но поскольку мы рисуем текст используя специальную матрицу проекции, то эффект от воздействия матрицы вида модели будет отличаться от того, что мы ожидаем увидеть в масштабе пикселей. Мы бы могли обойти эту проблему, не сбрасывая матрицу проекции в функции печати. Это вероятно хорошая идея в некоторых случаях, но если вы попробуете это, то удостоверьтесь, что вы масштабируете шрифты правильным размером (например, размер 32x32, соответствует размеру 0.01x0.01).

```
  float modelview_matrix[16];
  glGetFloatv(GL_MODELVIEW_MATRIX, modelview_matrix);

  // На каждой строчке мы сбрасываем матрицу вида модели
  // Поэтому строки будут начинаться с правильной позиции.
  // Отмечу, что сброс надо делать до сдвига вниз на h, поскольку затем каждый
  // символ рисуется и это модифицирует текущую матрицу, поэтому следующий
  // символ будет нарисован прямо после него.
  for(int i=0;i<lines.size();i++) {
    glPushMatrix();
    glLoadIdentity();
    glTranslatef(x,y-h*i,0);
    glMultMatrixf(modelview_matrix);

    // Уберите комментарии у следующего оператора и трех строк после вызова glCallLists,
    // если хотите знать длину строки, но не забудьте убрать комментарий
    // у glBitmap в функции make_dist.
    // glRasterPos2f(0,0);
    glCallLists(lines[i].length(), GL_UNSIGNED_BYTE, lines[i].c_str());
    // float rpos[4];
    // glGetFloatv(GL_CURRENT_RASTER_POSITION ,rpos);
    // float len=x-rpos[0]; (Надеюсь, что нет вращения)

    glPopMatrix();
  }
  glPopAttrib();
  pop_projection_matrix();
}
}                       // Закроем пространство имени
```

Библиотека закончена. Откройте lesson13.cpp и мы сделаем некоторые незначительные изменения, чтобы показать все функции, которые мы только что написали.

Добавьте FreeType.h за другими заголовочными файлами:

```
#include "freetype.h" // Заголовочный файл нашей маленькой библиотеки.
```

И пока мы здесь, давайте создадим глобальную переменную font_data.

```
// Сохраним всю информацию о нашем шрифте.
freetype::font_data our_font;
```

Сейчас нам нужно заняться созданием и удалением нашего шрифта. Так что добавьте следующее в конец InitGL.

```
our_font.init("Test.ttf", 16); // Создать шрифт FreeType
```

Добавьте это в начало KillGL Window, чтобы уничтожить шрифт, когда мы закончим.

```
our_font.clean();
```

Теперь мы должны изменить нашу функцию DrawGLScene так чтобы она вызывала функцию печати. По идее это также просто, как и добавить простую строку “hello world” в конце функции, но я хочу сделать чуточку побольше, поскольку я бы хотел показать масштабирование и вращение.

```
int DrawGLScene(GLvoid)                 // Здесь мы рисуем
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Очистка экрана и буфера глубины
  glLoadIdentity();                     // Сброс матрицы вида модели
  glTranslatef(0.0f,0.0f,-1.0f);      // Сдвиг на одну единицу в экран

  // Синий текст
  glColor3ub(0,0,0xff);

  // Позиция текста WGL на экране
  glRasterPos2f(-0.40f, 0.35f);
  glPrint("Active WGL Bitmap Text With NeHe - %7.2f", cnt1);  // Вывод текста

  // Выводим тот же текст, но с вращением и масштабированием.
  // Красный текст
  glColor3ub(0xff,0,0);
  glPushMatrix();
  glLoadIdentity();
  glRotatef(cnt1,0,0,1);
  glScalef(1,.8+.3*cos(cnt1/5),1);
  glTranslatef(-180,0,0);
  freetype::print(our_font, 320, 200, "Active FreeType Text - %7.2f", cnt1);
  glPopMatrix();

  // Уберите комментарий для вывода текста в несколько строк.
  // freetype::print(our_font, 320, 200, "Here\nthere\nbe\n\nnewlines\n.", cnt1);

  cnt1+=0.051f;                 // Увеличим первый счетчик
  cnt2+=0.005f;                 // Увеличим второй счетчик
  return TRUE;                  // Все OK
}
```

Последняя вещь, которую осталось сделать это добавить обработку исключений для надежности. Идите в WinMain и добавьте в начале ее try{..}.

```
  MSG  msg;                 // Структура сообщения
  BOOL  done=FALSE;           // Переменная цикла

  try {                     // Использовать обработку сообщений
```

Затем измените конец функции, чтобы иметь catch{}.

```
  // Завершение
  KillGLWindow();                   // убить окно

  // Захват исключений
  } catch (std::exception &e) {
    MessageBox(NULL,e.what(),"CAUGHT AN EXCEPTION",MB_OK | MB_ICONINFORMATION);
  }

  return (msg.wParam);                  // Выход из программы
}
```

Теперь, если когда-нибудь, что-то случается, появляется исключение, и мы получим небольшое сообщение гласящее, что собственно случилось. Заметьте, что обработка исключений может замедлить код, так что когда вы будете выпускать окончательную версию, желательно было бы выключить exception handling в Project->Settings->C/C++, в категории “C++ Language”.

Вот и все! Скомпилируйте программу и вы должны видеть красивый FreeType сгенерированный текст, движущийся вокруг оригинального растрового текста из Урока 13.

Общие замечания

Есть ряд усовершенствований, который вы можете добавить в библиотеку. С одной стороны непосредственное использование данных шрифта выглядит неуклюже, так что вы можете захотеть создать стандартный кэш шрифтов, чтобы скрыть управление шрифтовыми ресурсами от пользователя. Вы также можете наподобие OpenGL и создать стек шрифтов, который позволил вам ссылаться на шрифт при вызове функции печати. (Эти вещи я всегда делаю в своем коде, но не привожу этого в уроке для простоты). Вы также можете захотеть сделать версию функции print, которая выравнивает текст по центру, тогда вам, вероятно, нужно использовать нижеследующие методы.

Сейчас у меня есть текст, вращающийся вокруг центра. Однако, чтобы получить такой эффект для любого текста, вам нужно точно знать длину текста – это довольно сложно. Один способ получения длины текста – это засунуть команды glBitmap в список отображения в порядке изменения растровой позиции как в матрице вида модели (я оставил нужную линию в коде, но она закомментирована). Тогда мы должны установить позицию x,y перед использованием glCallLists, и использовать glGet чтобы найти её после отрисовки текста – разница даст вам длину текста в пикселях.

Вы должны знать, что FreeType шрифты используют гораздо больше памяти, чем WGL's растровые шрифты (это одно из преимуществ бинарных изображений, они используют мало памяти). Если по каким-нибудь причинам вам нужно минимизировать использование памяти, наверное лучше придерживаться кода из урока 13.

Другое интересное преимущество использования текстурированных четырехугольников то, что четырехугольники, по сравнению с растрами, работают лучше с функциями выбора OpenGL (см Урок 32). Это делает жизнь гораздо проще, если вы хотите создать текст, который отвечает за наведение мыши или кликанье. (Использование WGL шрифтов здесь возможно, но опять же есть хитрый момент в использовании растровых координат, чтобы вычислить длину текста в пикселях).

И в завершении, я даю вам некоторые ссылки на библиотеки шрифтов к OpenGL. В зависимости от ваших целей и компилятора вы можете захотеть использовать одну из них вместо этой (на самом деле их очень много, я только включил то, с чем я немного поработал).

GLTT - эта старая библиотека которая вроде бы уже заброшена, но она получила позитивные отзывы. Основана на FreeType1. Я думаю, что вам надо найти копию старых исходников FreeType1 чтобы скомпилировать в MSVC6. Загрузка доступна отсюда http://gltt.sourceforge.net/index.html.

OGLFT неплохая библиотека шрифтов, основанная на FreeType, потребуется немного усилий, чтобы скомпилировать ее под MSVC. Основная платформа там Linux... http://oglft.sourceforge.net.

FTGL ещё третья библиотека, основанная на FreeType, она была разработана под OS X. http://homepages.paradise.net.nz/henryj/code/#FTGL.

FNT библиотека, основанная не на FreeType, являющаяся частью PLIB. Имеет неплохой интерфейс, использует собственный формат, отлично компилируется под MSVC6. http://plib.sourceforge.net/fnt.

Урок 46. Полноэкранное сглаживание.

Fullscreen AntiAliasing

Привет всем, дружелюбный соседский таракан (the Friendly Neighborhood Roach) здесь с интересным примером, который поможет вашим приложениям достигнуть небывалых вершин. Мы все сталкиваемся с большой проблемой, когда хотим чтобы наши программы на OpenGL лучше выглядели. Этот эффект называется «пикселизация» (или зубчатость или ступенчатость - aliasing), и представляет из себя квадратные зубцы на диагональных гранях относительно квадратных пикселей на вашем экране. Решение проблемы – Anti-Aliasing (Сглаживание граней) используется для размытия таких зубцов, что позволяет создавать более гладкие грани объектов. Один из способов, позволяющих получить сглаживание, называется “Multisampling” (мультисэмплинг, или мультиопрос, или мультивыборка, или далее переводится как метод множественной выборки). Идея состоит в том, чтобы для каждого пикселя выбрать окружающие его пиксели (или субпиксели) с целью определить, нуждается ли данная грань в сглаживании (если пиксель не принадлежит грани, то сглаживать его не надо), но обычно мы избавляемся от ступенчатости при помощи "размазывания" самого пикселя.
Fullscreen AntiAliasing (полноэкранное сглаживание) – это то, в чем программы визуализации, не в масштабах реального времени, всегда имели преимущество. Однако с сегодняшним аппаратным обеспечением мы способны добиться схожего эффекта в реальном времени. Расширение ARB_MULTISAMPLE позволяет нам это сделать. По существу, каждый пиксель представлен своими соседями для определения оптимального сглаживания. Как бы то ни было этот процесс дорого обходится и может замедлить производительность. Например, требуется больше видеопамяти:

```
Vid_mem = sizeof(Front_buffer) +
        sizeof(Back_buffer) +
        num_samples * (sizeof(Front_buffer) +sizeof(Z_buffer))
```

Более подробную информацию относительно сглаживания, также как о том, что я собираюсь рассказать, можно найти по этим адресам:

GDC2002 -- OpenGL Multisample (http://developer.nvidia.com/attach/3464)
OpenGL Pixel Formats and Multisample Antialiasing (http://developer.nvidia.com/attach/2064)
Антиалиасинг сегодня (http://www.nvworld.ru/docs/fsaa2.html)

Вот краткий обзор того, как наш метод будет работать, с учетом вышесказанного. В отличие от других расширений, касающихся визуализации OpenGL, расширение ARB_MULTISAMPLE включается в работу при создании окна визуализации.
Наш процесс выглядит следующим образом:

- Создается обычное окно
- Собираем возможные значения форматов пикселей для последующего сглаживания (InitMultisample)
- Если сглаживание возможно, то уничтожаем окно и создаем его заново, с новым форматом пикселя.
- Для частей, которые мы хотим сгладить, просто вызываем функцию glEnable(GL_ARB_MULTISAMPLE).

Начнем с начала и поговорим о нашем исходном файле – arbMultiSample.cpp. Начинаем со стандартного включения заголовочных файлов gl.h и glu.h, а также windows.h. О arb_multisample.h мы поговорим позже.

```
#include <windows.h>
#include <gl/gl.h>
#include <gl/glu.h>
#include "arb_multisample.h"
```

Две строчки ниже определяют точки входа в список строк WGL. Мы будем их использовать при доступе к атрибутам формата пикселя для определения формата нашего типа. Две другие переменные нужны для доступа к нашим данным.

```
// Объявления, которые мы будем использовать
#define WGL_SAMPLE_BUFFERS_ARB  0x2041
#define WGL_SAMPLES_ARB         0x2042
bool arbMultisampleSupported = false;
int  arbMultisampleFormat = 0;
```

Следующей функцией, о которой мы поговорим, является WGLisExtensionSupported, которая будет использована для сбора информации о WGL расширениях для определения поддерживаемого формата на нашей системе. Мы будем давать описания кода по мере продвижения по нему, так как это легче чем прыгать по странице туда сюда.

Примечание: Код внизу написан Генри Гоффином. Его изменения внесли: Улучшенный разбор расширений GL и решили проблему с выпаданием кода, если первая проверка была не успешной.

```
bool WGLisExtensionSupported(const char *extension)
{
 const size_t extlen = strlen(extension);
 const char *supported = NULL;
        // попытка использовать wglGetExtensionStringARB на текущем контексте, если возможно
 PROC wglGetExtString = wglGetProcAddress("wglGetExtensionsStringARB");
    if (wglGetExtString)
  supported = ((char*(__stdcall*)(HDC))wglGetExtString)(wglGetCurrentDC());
        // Если  проверка не пройдена, то попытаемся использовать стандартное расширение OpenGL
 if (supported == NULL)
  supported = (char*)glGetString(GL_EXTENSIONS);
        // Если и это не поддерживается, тогда работаем без расширений
 if (supported == NULL)
  return false;
        // Начинаем проверку с начала строки, увеличиваем на 1, при false совпадение
 for (const char* p = supported; ; p++)
 {
  // Продвигаем p до следующего возможного совпадения
  p = strstr(p, extension);
    if (p == NULL)
    return false; // Совпадения нет
          //Убедимся, что есть совпадение в начале строки,
          //или первый символ – пробел, или может быть случайное
          //совпадение "wglFunkywglExtension" с "wglExtension"
          // Также убедимся, что текущий символ пустой или пробел
         // или еще "wglExtensionTwo" может совпасть с "wglExtension"
  if ((p==supported || p[-1]==' ') && (p[extlen]=='\0' || p[extlen]==' '))
    return true; // Совпадение
 }  }
```

Примечание переводчика:

Работа с wglGetProcAddress описана в уроке 22. Функция const char *wglGetExtensionsStringARB(HDC hdc) возвращает строку с перечнем расширений, hdc – контекст устройства.
Следующая функция – собственно суть всей программы. В сущности, мы собираемся выяснить поддерживается ли наше arb расширение на нашей системе. По сему мы будем запрашивать контекст устройства с целью выяснить наличие метода множественной выборки. Опять… давайте просто перейдем к коду.

```
bool InitMultisample(HINSTANCE hInstance,HWND hWnd,PIXELFORMATDESCRIPTOR pfd)
{
 // посмотрим, есть ли строка в WGL!
 if (!WGLisExtensionSupported("WGL_ARB_multisample "))
 {
  arbMultisampleSupported=false;
  return false;
 }
 // Возьмем наш формат пикселя
 PFNWGLCHOOSEPIXELFORMATARBPROC wglChoosePixelFormatARB =
  (PFNWGLCHOOSEPIXELFORMATARBPROC)wglGetProcAddress("wglChoosePixelFormatARB");
 if (!wglChoosePixelFormatARB)
 {
  // Мы не нашли поддержки для метода множественной выборки, выставим наш флаг и выйдем.
  arbMultisampleSupported=false;
  return false;
 }

 // Получаем контекст нашего устройства. Нам это необходимо для того, что
 // спросить у OpenGL окна, какие атрибуты у нас есть
 HDC hDC = GetDC(hWnd);
```

```
 int pixelFormat;
 bool valid;
 UINT numFormats;
 float fAttributes[] = {0,0};

 // Эти атрибуты – биты, которые мы хотим протестировать в нашем типе
 // Все довольно стандартно, только одно на чем мы хотим
 // действительно сфокусироваться - это SAMPLE BUFFERS ARB и WGL SAMPLES
 // Они выполнят главную проверку на предмет: есть или нет
 // у нас поддержка множественной выборки
 int iAttributes[] = {
  WGL_DRAW_TO_WINDOW_ARB,GL_TRUE, // Истинна, если формат пикселя может быть использован в окне
  WGL_SUPPORT_OPENGL_ARB,GL_TRUE, // Истинна, если поддерживается OpenGL
  WGL_ACCELERATION_ARB,WGL_FULL_ACCELERATION_ARB, // Полная аппаратная поддержка
  WGL_COLOR_BITS_ARB,24,          // Цветность
  WGL_ALPHA_BITS_ARB,8,           // Размерность альфа-канала
  WGL_DEPTH_BITS_ARB,16,          // Глубина буфера глубины
  WGL_STENCIL_BITS_ARB,0,         // Глубина буфера шаблона
  WGL_DOUBLE_BUFFER_ARB,GL_TRUE,  // Истина, если используется двойная буферизация
  WGL_SAMPLE_BUFFERS_ARB,GL_TRUE, // Что мы и хотим
  WGL_SAMPLES_ARB, 4 ,            // проверка на 4х тип
  0,0};

          // Сначала посмотрим, сможем ли мы получить формат пикселя для 4х типа
 valid = wglChoosePixelFormatARB(hDC,iAttributes,fAttributes,1,&pixelFormat,&numFormats);
          // Если вернулось True, и наш счетчик форматов больше 1
 if (valid && numFormats >= 1)
 {
  arbMultisampleSupported  = true;
  arbMultisampleFormat  = pixelFormat;
  return arbMultisampleSupported;
 }
          // Формат пикселя с 4х выборкой отсутствует, проверяем на 2х тип
 iAttributes[19] = 2;
 valid = wglChoosePixelFormatARB(hDC,iAttributes,fAttributes,1,&pixelFormat,&numFormats);
 if (valid && numFormats >= 1)
 {
  arbMultisampleSupported  = true;
  arbMultisampleFormat  = pixelFormat;
  return arbMultisampleSupported;
 }
          // возвращаем годный формат
 return  arbMultisampleSupported;
}
```

Примечание переводчика:
Функция:

```
BOOL wglChoosePixelFormatARB(HDC hdc,
                             const GLint *piAttribIList,
                             const GLfloat *pfAttribFList,
                             GLuint nMaxFormats,
                             GLint *piFormats,
                             GLuint *nNumFormats);
```

Выбирает форматы пикселя согласно запрашиваемым атрибутам. hdc – контекст устройства, piAttribIList или pfAttribFList – список желаемых атрибутов (пары {тип, значение} формате целого числа или в формате с плавающей запятой, в конце списка {0,0}, значения типов атрибутов задаются объявлениями define выше или взяты из wglext.h, значение зависит от типа). nMaxFormats – максимальное число форматов, которое будет возвращено. piFormats – массив индексов форматов пикселов, которые совпадают с запрашиваемым. Наилучший формат будет первым. nNumFormats – сколько форматов найдено при запросе.

Теперь, когда у нас есть готовый код запроса, мы должны изменить процесс создания окна. Сперва, мы должны включить наш заголовочный файл arb_multisample.h и поместить прототипы DestroyWindow и CreateWindow в начало файла.

```
#include <windows.h>      // Заголовочный файл библиотеки Windows
#include <gl/gl.h>        // Заголовочный файл  библиотеки OpenGL32
#include <gl/glu.h>       // Заголовочный файл библиотеки GLu32
#include "NeHeGL.h"       // Заголовочный файл NeHeGL основного кода

// ЗДЕСЬ ПРОБЕЖАЛ ТАРАКАН
#include "ARB_MULTISAMPLE.h"

BOOL DestroyWindowGL (GL_Window* window);
BOOL CreateWindowGL (GL_Window* window);
// ENDТАРАКАН
```

Следующий кусок кода должен быть добавлен в функцию CreateWindowGL, код функции был оставлен без изменений, дабы вы могли вносить свои модификации. В общем, мы делаем часть «уничтожения» до конца работы. Мы не можем запросить формат пикселя, для определения типа множественной выборки, пока мы не создадим окно. Но мы не можем создать окно, пока не знаем формат пикселя, который оно поддерживает. Сродни вечному вопросу о курице и яйце. Итак, все, что я сделал – это маленькая двухпроходная система; мы создаем окно, определяем формат пикселя, затем уничтожаем (пересоздаем) окно, если метод множественной выборки поддерживается. Немного круто…

```
  window->hDC = GetDC (window->hWnd); // Забираем контекст данного окна
  if (window->hDC == 0)               // Мы получили контекст устройства?
  {
    // Нет
    DestroyWindow (window->hWnd);     // Уничтожаем окно
    window->hWnd = 0;                 // Обнуляем указатель
    return FALSE;                     // возвращаем False
  }

// ЗДЕСЬ ПРОБЕЖАЛ ТАРАКАН
  // Наш первый проход, множественная выборка пока не подключена, так что мы создаем обычное окно
  // Если поддержка есть, тогда мы идем на второй проход
  // это значит, что мы хотим использовать наш формат пикселя для выборки
  // и так, установим PixelFormat в arbMultiSampleformat вместо текущего
  if(!arbMultisampleSupported)
  {
    PixelFormat = ChoosePixelFormat (window->hDC, &pfd); // найдем совместимый формат пикселя
    if (PixelFormat == 0)             // мы нашли его?
    {
      // Нет
      ReleaseDC (window->hWnd, window->hDC);  // Освобождаем контекст устройства
      window->hDC = 0;                // Обнуляем контекст
      DestroyWindow (window->hWnd);   // Уничтожаем окно
      window->hWnd = 0;               // Обнуляем указатель окна
      return FALSE;                   // возвращаем False
    }
  }
  else
  {
    PixelFormat = arbMultisampleFormat;
  }
// ENDТАРАКАН

  // пытаемся установить формат пикселя
  if (SetPixelFormat (window->hDC, PixelFormat, &pfd) == FALSE)
  {
    // Неудача
    ReleaseDC (window->hWnd, window->hDC); // Освобождаем контекст устройства
    window->hDC = 0;                  // Обнуляем контекст
```

```
    DestroyWindow (window->hWnd);     // Уничтожаем окно
    window->hWnd = 0;                 // Обнуляем указатель окна
    return FALSE;                     // возвращаем False
  }
```

Теперь окно создано и у нас есть правильный указатель, чтобы запросить поддержку множественной выборки. Если поддержка есть, то мы уничтожаем окно и опять вызываем CreateWindowGL с новым форматом пикселя.

```
  // Сделаем контекст визуализации нашим текущим контекстом
  if (wglMakeCurrent (window->hDC, window->hRC) == FALSE)
  {
    // Не удалось
    wglDeleteContext (window->hRC);   // Уничтожаем контекст визуализации
    window->hRC = 0;                  // Обнуляем контекст визуализации
    ReleaseDC (window->hWnd, window->hDC); // Освобождаем контекст устройства
    window->hDC = 0;                  // Обнуляем его
    DestroyWindow (window->hWnd);     // Уничтожаем окно
    window->hWnd = 0;                 // Обнуляем указатель окна
    return FALSE;                     // возвращаем False
  }


// ЗДЕСЬ ПРОБЕЖАЛ ТАРАКАН
  // Теперь, когда наше окно создано, мы хотим узнать какие типы доступны.
  // Мы вызываем нашу функцию InitMultiSample для создания окна
  // если вернулся правильный контекст, то мы уничтожаем текущее окно
  // и создаем новой, используя интерфейс множественной выборки.
  if(!arbMultisampleSupported && CHECK_FOR_MULTISAMPLE)
  {
    if(InitMultisample(window->init.application->hInstance,window->hWnd,pfd))
    {
      DestroyWindowGL (window);
      return CreateWindowGL(window);
    }
  }
// ENDТАРАКАН

  ShowWindow (window->hWnd, SW_NORMAL);  // Сделаем окно видимым
  window->isVisible = TRUE;
```

Ок, и так настройка теперь завершена! Мы настроили наше окно для работы с методом множественной выборки. Теперь повеселимся, собственно делая это! К счастью, ARB решила сделать поддержку метода множественной выборки динамической, это позволяет нам включать и выключать ее вызовами glEnable / glDisable.

```
glEnable(GL_MULTISAMPLE_ARB);
// Визуализируем сцену
glDisable(GL_MULTISAMPLE_ARB);
```

Вот собственно и все! Далее идет код примера, в котором простые квадраты вращаются, чтобы вы могли увидеть, как хорошо метод множественной выборки работает. НАСЛАЖДАЙТЕСЬ!

Урок 48. Вращение объектов с помощью класса ArcBall

ArcBall Rotation

Как Вы думаете, здорово было бы вращать объект, пользуясь только мышью? Используя функциональность класса ArcBall можно довольно просто добиться этого. Здесь я расскажу Вам о моей реализации этого класса, и как его добавить в Ваши проекты.

Моя реализация класса ArcBall базируется на коде Бреттона Вада, который позаимствовал код Кена Шоемака в одной из книг "Графические драгоценности" (Graphic Gems). Однако я исправил ошибки в коде, и оптимизировал его.

Класс ArcBall предназначен для того, чтобы преобразовать координаты курсора мыши в сферические координаты ArcBall, так как, если бы это было непосредственно перед Вами.

Чтобы выполнить это, вначале масштабируем, координаты мыши из диапазона [0…ширина], [0...высота] в диапазон [-1...1], [1...-1] (запомните, что мы меняем знак координаты Y, чтобы получить корректный результат в OpenGL). Фактически, это делается так:

```
MousePt.X  =  ((MousePt.X / ((Width  – 1) / 2)) – 1);
MousePt.Y  = -((MousePt.Y / ((Height – 1) / 2)) – 1);
```

Масштабирование мы делаем для простоты дальнейших математических операций, еще один плюс в том, что компилятор может при этом сделать самостоятельно небольшую оптимизацию кода.

Затем мы определяем длину вектора, и определяем внутри он или снаружи сферы. Если вектор в переделах сферы, то мы возвращаем вектор в сфере, иначе мы нормализуем точку и возвращаем наиболее близкую точку на внешней границе сферы.

Как только мы имеем начальный и конечный вектора, мы можем вычислить вектор перпендикулярный им и углом между ними, который задает вращение. А это вращение можно описать кватернионом. После этого у нас достаточно информации, чтобы вычислить матрицу вращения.

Конструктор класса ArcBall:

```
ArcBall_t::ArcBall_t(GLfloat NewWidth, GLfloat NewHeight)
```

Где NewWidth и NewHeight – ширина и высота окна.

Когда пользователь щелкает мышью, начальный вектор рассчитывается исходя из точки, где он произошел:

```
void    ArcBall_t::click(const Point2fT* NewPt)
```

Когда пользователь двигает мышью с нажатой клавишей, то конечный вектор модифицируется с помощью метода:

```
void    ArcBall_t::drag(const Point2fT* NewPt, Quat4fT* NewRot)
```

И если кватернион задан, то он обновляется.

Если меняется размер окна, мы просто передаём классу ArcBall следующую информацию:

```
void    ArcBall_t::setBounds(GLfloat NewWidth, GLfloat NewHeight)
```

При использовании этого класса в нашем проекте, нам будет необходимо изменить некоторые переменные:

```
// Завершающая трансформация
Matrix4fT  Transform = {  1.0f,  0.0f,  0.0f,  0.0f,
          0.0f,  1.0f,  0.0f,  0.0f,
          0.0f,  0.0f,  1.0f,  0.0f,
          0.0f,  0.0f,  0.0f,  1.0f };

Matrix3fT  LastRot   = {  1.0f,  0.0f,  0.0f,          // Последнее вращение
                    0.0f,  1.0f,  0.0f,
                    0.0f,  0.0f,  1.0f };
```

```
Matrix3fT  ThisRot   = {  1.0f,  0.0f,  0.0f,          // Это вращение
                          0.0f,  1.0f,  0.0f,
                          0.0f,  0.0f,  1.0f };

ArcBallT  ArcBall(640.0f, 480.0f); // экземпляр класса ArcBall
Point2fT  MousePt;                 // Текущие координаты мыши
bool    isClicked  = false;        // Кликнули по мыши?
bool    isRClicked = false;        // Кликнули по правой клавиши мыши?
bool    isDragging = false;        // Потащили мышь?
```

Переменная Transform - это матрица с помощью, которой вращаются все объекты. Переменная LastRot - матрица, в которой сохраняется вращение объекта, когда пользователь отпускает кнопку мышки. Переменная ThisRot - матрица, в которой накапливается вращение объекта во время перетаскивания мышью. Эти переменные вначале инициализируются единичной матрицей.

Когда мы кликаем, мы начинаем крутить объекты с начальной позиции. Когда мы перетаскиваем, мы вычисляем угол поворота от начальной точки до точки, в которой находится указатель. Необходимо заметить, что класс ArcBall не вращается. Чтобы получить одинаковое значение в этих переменных (кумулятивное вращение) мы должны сами об этом позаботиться.

Для этого и нужны LastRot и ThisRot. LastRot можно представить как переменную, показывающую то, что происходило до этого момента. Тогда ThisRot - переменная показывающая то, что происходит с объектами сейчас. Каждый раз, когда мы начинаем перетаскивать мышь, ThisRot изменяется вместе с координатами курсора. Одновременно с этим изменяется Transform. Как только перетаскивание заканчивается, значение ThisRot передаётся в переменную LastRot.

Если мы не будем накапливать значения сами, то модель будет возвращаться к положению до перетаскивания. Например: если мы вращаем модель по оси X на 90 градусов, а затем на 45, мы должны будем увидеть поворот на 135 град, вместо последних 45.

Остальным переменным (кроме isDragged) необходимо вовремя передавать правильные значения. Классу ArcBall необходимо сбрасывать значения каждый раз, когда меняется размер окна. MousePt обновляется тогда, когда мышь движется, или когда нажата кнопка. Переменные isClicked/isRClicked отвечают за нажатие левой/правой кнопки мыши, соответственно. Переменная isClicked используется для распознавания нажатия/перетаскивания. Переменную isRClicked мы будем использовать для возврата в начальное положение всех объектов.

Всё выше описанное выглядит примерно так :

```
void ReshapeGL (int width, int height)
{
  . . .
  ArcBall.setBounds((GLfloat)width, (GLfloat)height); // Обновить границы для ArcBall
}

// Обслуживание сообщений
LRESULT CALLBACK WindowProc (HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
  . . .
  // Сообщения от мыши для ArcBall
  case WM_MOUSEMOVE:
    MousePt.s.X = (GLfloat)LOWORD(lParam);
    MousePt.s.Y = (GLfloat)HIWORD(lParam);
    isClicked   = (LOWORD(wParam) & MK_LBUTTON) ? true : false;
    isRClicked  = (LOWORD(wParam) & MK_RBUTTON) ? true : false;
    break;

  case WM_LBUTTONUP:   isClicked  = false; break;
  case WM_RBUTTONUP:   isRClicked = false; break;
  case WM_LBUTTONDOWN: isClicked  = true;  break;
  case WM_RBUTTONDOWN: isRClicked = true;  break;
  . . .
}
```

Теперь самое время для того, чтобы вставить код клика. Всё это очевидно, если понять написанное выше.

```
if (isRClicked) // Если правая клавиша мыши нажата, сбросить все вращения
{
  // Сброс вращения
  Matrix3fSetIdentity(&LastRot);

  // Сброс вращения
  Matrix3fSetIdentity(&ThisRot);

  // Сброс вращения
  Matrix4fSetRotationFromMatrix3f(&Transform, &ThisRot);
}

if (!isDragging)              // Нет перетаскивания
{
  if (isClicked)              // Первый клик
  {
    isDragging = true;        // Начать перетаскивание
    LastRot = ThisRot;        // Присвоить последнему статическому вращению
                              // значение последнего динамического вращения
    ArcBall.click(&MousePt); // Обновить начальный вектор и выполнить перетаскивание
  }
}
else
{
  if (isClicked)  // Все еще клик, по этому и перетаскивание
  {
    Quat4fT     ThisQuat;

    ArcBall.drag(&MousePt, &ThisQuat); // Обновить окончательный вектор
                                       // и получить вращение как кватернион
    Matrix3fSetRotationFromQuat4f(&ThisRot, &ThisQuat); // Конвертировать кватернион в Matrix3fT
    Matrix3fMulMatrix3f(&ThisRot, &LastRot); // Добавить текущее вращение в последнее вращение
    Matrix4fSetRotationFromMatrix3f(&Transform, &ThisRot); // Вычислить нашу финальную трансформацию
  }
  else // Больше нет перетаскивания
    isDragging = false;
}
```

Это всё что нам необходимо. Осталось только сделать так, чтобы положение моделей постоянно обновлялось. Это достаточно просто:

```
  glPushMatrix();             // Выполнить динамическую трансформацию
  glMultMatrixf(Transform.M); // Применить динамическую трансформацию
  glBegin(GL_TRIANGLES);      // Начать рисовать модели
  . . .
  glEnd();                    // Завершить рисование моделей
  glPopMatrix();              // Сбросить динамическую трансформацию
```

Я сделал пример, демонстрирующий всё вышеописанное. Вы можете использовать не только мой код, но и встраивать туда какие-то свои конструкции. Программы созданные с помощью этой библиотеки могут работать сами по себе и не требуют ничего дополнительного.

Теперь, когда вы видите насколько это всё просто, вы можете спокойно добавлять класс ArcBall в свои приложения!

Урок X1. Улучшенная обработка ввода с использованием DirectInput и Windows

Вы должны использовать самые современные технологии, чтобы конкурировать с такими играми как Quake и Unreal. В этом уроке я научу вас, как подключить и использовать DirectInput и как использовать мышь в OpenGL под Windows. Код этого урока базируется на коде урока 10. Начнем.

Мышь

Первое, что нам понадобиться, это переменная для хранения X и Y позиции мыши.

```
typedef struct tagSECTOR
{
  int numtriangles;
  TRIANGLE* triangle;
} SECTOR;

SECTOR sector1;            // Наша модель
POINT mpos;                // Позиция мыши (Новое)

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);  // Объявление WndProc
```

Отлично, как вы видите, мы добавили новую переменную mpos. Структура POINT состоит из двух переменных – x и y, мы будем использовать их для того, чтобы вычислить вращение сцены. Далее мы изменим, часть функции CreateGLWindow() так, как показано ниже.

```
ShowCursor(FALSE);             // Убрать указатель мыши (Изменено)
if (fullscreen)                // Если полноэкранный режим?
{
  dwExStyle=WS_EX_APPWINDOW;
  dwStyle=WS_POPUP;
}
```

Выше мы переместили вызов ShowCursor(FALSE) так, чтобы курсора мыши не было видно не только в полноэкранном режиме, но и в оконном тоже. Теперь нам нужно получить и установить координаты мыши каждый кадр, поэтому измените функцию WinMain() так как показано ниже:

```
SwapBuffers(hDC);              // Смена буферов (двойная буферизация)
GetCursorPos(&mpos);           // Получить текущую позицию мыши (Новое)
SetCursorPos(320,240);         // Установить мышь в центр окна (Новое)
heading += (float)(320 - mpos.x)/100 * 5;//Обновить направление движения (Новое)
yrot = heading;                // Обновить вращение вокруг оси Y (Новое)
lookupdown -= (float)(240 - mpos.y)/100 * 5;//Обновить вращение вокруг X (Новое)
```

Сначала мы получили позицию мыши при помощи функции GetCursorPos(POINT p). Смещение от центра окна даст нам информацию о том, куда и насколько нужно вращать камеру. Затем мы устанавливаем мышь в центр окна для следующего прохода используя SetCursorPos(int X, int Y).

Замечание: Не устанавливайте позицию мыши в 0,0! Если вы сделаете это, то не сможете обработать перемещение мыши вверх и влево, потому что 0,0 это левый верхний угол окна. 320, 240 – это центр окна для режима 640x480.

После того как мы позаботились о мыши, нужно изменить часть кода для выполнения перемещения.

```
float = (P - CX) / U * S;
```

P – точка, в которую мы устанавливаем мышь каждый кадр
CX – текущая позиция мыши
U – единицы
S – скорость мыши (будучи истинным квакером я люблю в этом месте значение 12).

По этой формуле вычисляются значения переменных heading и lookupdown.

С мышью вроде как разобрались. Идем дальше.

Клавиатура (DirectX 7)

Теперь мы можем при помощи мыши вращать камеру в нашем мире. Следующий шаг использовать клавиатуру для перемещения вперед, атаки и приседания. Довольно болтовни, начнем кодировать.

Сначала я расскажу, как использовать DirectX7. Первый шаг – настройка компилятора. Я покажу, как сделать это на примере Visual C++, настройка других компиляторов может отличаться от предложенного способа.

Если у вас еще нет DirectX SDK, то вам придется его заиметь, например, скачать с сайта MicroSoft, и проинсталлировать его.

После этого, в VisualStudio зайдите в меню Project->Settings. Выберите закладку Link и в строке Object/libraty modules в начало строки добавьте dinput.lib dxguid.lib winmm.lib. Библиотеки DirectInput, DirectX GUID и Windows Multimedia соответственно, последняя необходима для таймера. Возможно, вам также понадобиться войти в меню Tools->Options и на закладке Directories добавить пути (Include files и Library files) к DirectX SDK и переместить их наверх списка.

Теперь DirectInput готов к использованию, можно начинать программировать!

Мы нуждаемся в подключении заголовочных файлов DirectInput, для того чтобы мы могли использовать некоторые его функции. Также мы нуждаемся в создании и добавлении новых переменных для DirectInput и для устройства DirectInput клавиатуры. Сделаем мы это следующим образом:

```
#include <stdio.h>            // Заголовочный файл стандартного ввода/вывода
#include <gl\gl.h>            // Заголовочный файл библиотеки OpenGL32
#include <gl\glu.h>           // Заголовочный файл библиотеки GLu32
#include <gl\glaux.h>         // Заголовочный файл библиотеки Glaux
#include <dinput.h>           // DirectInput функции  (Новое)

LPDIRECTINPUT7       g_DI;     // DirectInput (Новое)
LPDIRECTINPUTDEVICE7 g_KDIDev; // Устройство клавиатуры (Новое)
```

В последних две строчках объявляются переменные для DirectInput (g_DI) и для устройства клавиатуры (g_KDIDev), последнее будет получать данные и обрабатывать их. Константы DirectInput не сильно отличаются от стандартных констант Windows.

```
Windows       DirectInput
VK_LEFT       DIK_LEFT
VK_RIGHT      DIK_RIGHT
... и так далее
```

Основное отличие в замене VK на DIK. Хотя некоторые названия изменили существенно. Все DIK константы объявлены в файле dinput.h.

Теперь нужно написать функцию инициализации DirectInput'а и устройства клавиатуры. Под CreateGLWindow() добавьте следующее:

```
// Инициализация DirectInput (Новое)
int DI_Init()
{
 // Создание DirectInput
 if ( DirectInputCreateEx( hInstance,    // Instance окна
      DIRECTINPUT_VERSION,             // Версия DirectInput
      IID_IDirectInput7,
      (void**)&g_DI,                   // DirectInput
      NULL ) )                         // NULL параметр
 {
  return(false);                       // Не создался DirectInput
 }

 // Создание устройства клавиатуры
 if ( g_DI->CreateDeviceEx(  GUID_SysKeyboard,
           // Какое устройство создается (клавиатура, мышь или джойстик)
      IID_IDirectInputDevice7,
```

```
      (void**)&g_KDIDev,      // Устройство клавиатуры
      NULL ) )                // NULL параметр
  {
   return(false);             // Не создалось устройство клавиатуры
  }

  // Установка формата данных для клавиатуры
  if ( g_KDIDev->SetDataFormat(&c_dfDIKeyboard) )
  {
   return(false);           // Не удалось установить формат данных
   // здесь не хватает функций уничтожения устройства клавиатуры и DirectInput
  }

  // Установка уровня кооперации
  if ( g_KDIDev->SetCooperativeLevel(hWnd, DISCL_FOREGROUND | DISCL_EXCLUSIVE) )
  {
   return(false);           // Не удалось установить режим
   // здесь не хватает функций уничтожения устройства клавиатуры и DirectInput
  }

  if (g_KDIDev)             // Создано устройство клавиатуры? (лишняя проверка)
    g_KDIDev->Acquire();    // Взять его под контроль
  else                      // если нет
   return(false);           // возвращаем false
  return(true);             // все отлично
}

// Уничтожение DirectInput
void DX_End()
{
  if (g_DI)
  {
    if (g_KDIDev)
    {
      g_KDIDev->Unacquire();
      g_KDIDev->Release();
      g_KDIDev = NULL;
    }

    g_DI->Release();
    g_DI = NULL;
  }
}
```

Этот код в достаточной мере снабжен комментариями и должен быть понятен. Первое – мы инициализируем DirectInput и при помощи него создаем устройство клавиатуры, которое затем берем под контроль. Можно также использовать DirectInput для мыши, но на данном этапе средств Windows вполне достаточно.
Теперь нужно заменить старый код обработки ввода с клавиатуры на новый, использующий DirectInput. Изменить предстоит много, приступим.

Удалите следующий код из WndProc():

```
  case WM_KEYDOWN:          // Клавиша была нажата?
  {
    keys[wParam] = TRUE;    // Пометить ее как нажатую
    return 0;               // Вернуться
  }

  case WM_KEYUP:            // Клавиши была отпущена?
  {
    keys[wParam] = FALSE;   // Отменить пометку
    return 0;               // Вернуться
  }
```

В начале программы необходимо сделать следующие изменения:

```
BYTE  buffer[256];       // Новый буфер вместо Keys[] (Изменено)
bool  active=TRUE;       // Флаг активности окна
bool  fullscreen=TRUE;   // Флаг полноэкранного режима
bool  blend;             // Смешивание ON/OFF
bool  bp;                // Состояние кнопки смешивания
bool  fp;                // Состояние F1 (Изменено)
...

GLfloat  lookupdown = 0.0f;
GLfloat  z=0.0f;         // Глубина в экран
GLuint  filter;          // Фильтр (Удалено)

GLuint texture[5];       // Для текстур (Изменено)
```

В функции WinMain()

```
 // Создание окна OpenGL
 if (!CreateGLWindow("Justin Eslinger's & NeHe's Advanced DirectInput Tutorial",640,480,16,fullscreen))  // (Изменено)
 {
  return 0;           // Выйти, если не удалось создать окно
 }

 if (!DI_Init())        // Инициализация DirectInput (Новое)
 {
  return 0;
 }
 ...

 // Отрисовка сцены, пока окно активно и не была нажата клавиша Esc
 if ((active && !DrawGLScene())) // (Изменено)
    ...
 // Обновить состояние клавиатуры (Новое)
 HRESULT hr = g_KDIDev->GetDeviceState(sizeof(buffer), &buffer);
 if ( buffer[DIK_ESCAPE] & 0x80 )  // Тест клавиши Escape (Изменено)
 {
  done=TRUE;
 }
 if ( buffer[DIK_B] & 0x80)    // Нажата клавиша B? (Изменено)
 {
  if (!bp)
  {
   bp = true;               // Нажата клавиша смешения (Новое)
   blend=!blend;
   if (!blend)
   {
    glDisable(GL_BLEND);
    glEnable(GL_DEPTH_TEST);
   }
   else
   {
    glEnable(GL_BLEND);
    glDisable(GL_DEPTH_TEST);
   }
  }
 }
 else
 {
  bp = false;
 }
```

```
if ( buffer[DIK_PRIOR] & 0x80 )   // Page Up? (Изменено)
{
  z-=0.02f;
}

if ( buffer[DIK_NEXT] & 0x80 )    // Page Down? (Изменено)
{
  z+=0.02f;
}

if ( buffer[DIK_UP] & 0x80 )      // Вверх? (Изменено)
{
  xpos -= (float)sin(heading*piover180) * 0.05f;
  zpos -= (float)cos(heading*piover180) * 0.05f;
  if (walkbiasangle >= 359.0f)
  {
    walkbiasangle = 0.0f;
  }
  else
  {
    walkbiasangle+= 10;
  }

  walkbias = (float)sin(walkbiasangle * piover180)/20.0f;
}

if ( buffer[DIK_DOWN] & 0x80 )    // Вниз? (Изменено)
{
  xpos += (float)sin(heading*piover180) * 0.05f;
  zpos += (float)cos(heading*piover180) * 0.05f;
  if (walkbiasangle <= 1.0f)
  {
    walkbiasangle = 359.0f;
  }
  else
  {
    walkbiasangle-= 10;
  }
  walkbias = (float)sin(walkbiasangle * piover180)/20.0f;
}

if ( buffer[DIK_LEFT] & 0x80 )    // Влево? (Изменено)
{
  xpos += (float)sin((heading - 90)*piover180) * 0.05f;  ( Modified )
  zpos += (float)cos((heading - 90)*piover180) * 0.05f;  ( Modified )
  if (walkbiasangle <= 1.0f)
  {
    walkbiasangle = 359.0f;
  }
  else
  {
    walkbiasangle-= 10;
  }
  walkbias = (float)sin(walkbiasangle * piover180)/20.0f;
}
  if ( buffer[DIK_RIGHT] & 0x80 )    // Вправо? (Изменено)
{
  xpos += (float)sin((heading + 90)*piover180) * 0.05f;  ( Modified )
  zpos += (float)cos((heading + 90)*piover180) * 0.05f;  ( Modified )
  if (walkbiasangle <= 1.0f)
  {
    walkbiasangle = 359.0f;
  }
```

```
    else
    {
      walkbiasangle-= 10;
    }
    walkbias = (float)sin(walkbiasangle * piover180)/20.0f;
  }

  if ( buffer[DIK_F1] & 0x80)    // F1 нажато? (Изменено)
  {
    if (!fp)                    // Если не была нажата (Новое)
    {
      fp = true;                // F1 нажата (Новое)
      KillGLWindow();           // Уничтожить текущее окно (Изменено)
      fullscreen=!fullscreen;   // Переключить режим (Изменено)

      // Пересоздать окно
      if (!CreateGLWindow("Justin Eslinger's & NeHe's Advanced Direct Input Tutorial",640,480,16,fullscreen))  (Изменено)
      {
        return 0;       // Выйти, если не удалось создать окно (Изменено)
      }

      if (!DI_Init())    // Переинициализировать DirectInput (Новое)
      {
        return 0;       // Выйти, если не удалось
      }
    }
  }
  else
  {
    fp = false;         // F1 отпущена
  }

  // Shutdown
  // Выход
  DX_End();             // Уничтожить DirectInput (Новое)
  KillGLWindow();       // Уничтожить окно
  return (msg.wParam);   // Выйти из программы
}
```

Функция DrawGLScene() изменилась следующим образом.

```
  glTranslatef(xtrans, ytrans, ztrans);
  numtriangles = sector1.numtriangles;

  // Для каждого треугольника
  for (int loop_m = 0; loop_m < numtriangles; loop_m++)
  {
    glBindTexture(GL_TEXTURE_2D, texture[sector1.triangle[loop_m].texture]);  // (Изменено)
    glBegin(GL_TRIANGLES);
```

Отлично, теперь обсудим все это немного. Мы заменили весь старый код, использующий ввод с клавиатуры средствами Windows на код использующий средства DirectInput. Так же были изменены и сами клавиши управления для удобства использования.

Теперь в нашей игре есть поддержка мыши и клавиатуры через DirectInput, что дальше? Нам нужен таймер, для того чтобы регулировать скорость. Без таймера мы обрабатываем клавиатуру каждый кадр, из-за этого скорость перемещения и вращения будет различной на различных компьютерах.

Добавим переменную для корректировки и структуру для работы с таймером.

```
POINT  mpos;
int  adjust = 5;           // Корректировка скорости (Новое)

// информация для таймера (Новое)
struct
{
  __int64    frequency;            // Частота
  float    resolution;             // Точность
  unsigned long mm_timer_start;      // Стартовое значение
  unsigned long mm_timer_elapsed;    // Прошедшее время
  bool performance_timer;            // Использовать эффективный таймер
  __int64 performance_timer_start;   // Стартовое значение эффективного таймера
  __int64 performance_timer_elapsed; // Прошедшее время по эффективному таймеру
} timer;

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
```

Этот код обсуждался в уроке 21. Рекомендую просмотреть его.

```
// Инициализация таймера (Новое)
void TimerInit(void)
{
  memset(&timer, 0, sizeof(timer));  // Очистить структуру
  // Проверить есть ли возможность использования эффективного таймера
  if (!QueryPerformanceFrequency((LARGE_INTEGER *) &timer.frequency))
  {
    // нет эффективного таймера
    timer.performance_timer = FALSE;
    timer.mm_timer_start = timeGetTime(); // Использовать timeGetTime()
    timer.resolution = 1.0f/1000.0f;      // Установить точность .001f
    timer.frequency = 1000;               // Установить частоту 1000
    // Установить прошедшее время равным стартовому
    timer.mm_timer_elapsed = timer.mm_timer_start;
  }
  else
  {
   // доступен эффективный таймер
    QueryPerformanceCounter((LARGE_INTEGER *) &timer.performance_timer_start);
    timer.performance_timer = TRUE;
                // Вычисление точности и частоты
    timer.resolution = (float) (((double)1.0f)/((double)timer.frequency));
    // Установить прошедшее время равным стартовому
    timer.performance_timer_elapsed = timer.performance_timer_start;
  }
}
// Получить время в миллисекундах (Новое)
float TimerGetTime()
{
  __int64 time;                // Время храниться в 64-битном целом
  if (timer.performance_timer) // Если используется эффективный таймер
  {
    // Получить текущее время по эффективному таймеру
    QueryPerformanceCounter((LARGE_INTEGER *) &time);
    // вернуть текущее время мину стартовое с данной точностью и в миллисекундах
    return ( (float) ( time - timer.performance_timer_start) * timer.resolution)*1000.0f;
  }
  else
  {
    // вернуть текущее время минус стартовое с данной точностью и в миллисекундах
   return( (float) ( timeGetTime() - timer.mm_timer_start) * timer.resolution)*1000.0f;
  }
}
```

Напомню, что пояснения кода таймера есть в 21-ом уроке. Убедитесь, что к проекту добавлена библиотека winmm.lib.

Теперь мы добавим кое-что в функцию WinMain().

```
 if (!DI_Init())          // Инициализация DirectInput  (Новое)
 {
  return 0;
 }
 TimerInit();             // Инициализация таймера (Новое)
...

    float start=TimerGetTime();
    // Получить время перед отрисовкой (Новое)
    // Отрисовка сцены. Esc – выход.
    // Если окно активно и был выход (Изменено)
    if ((active && !DrawGLScene()))
    {
     done=TRUE;       // ESC DrawGLScene сигнализирует о выходе
    }
    else          // обновить сцену
    {
     // Цикл ожидания для быстрых систем (Новое)
     while(TimerGetTime()<start+float(adjust*2.0f)) {}
```

Теперь программа должна выполняться с корректной скоростью. Следующая часть кода предназначена для вывода уровня, как в уроке 10.

Если вы уже скачали код этого урока, то вы уже заметили, что я добавил несколько текстур для сцены. Новые текстуры находятся в папке Data.

Изменения структуры tagTriangle.

```
typedef struct tagTRIANGLE
{
 int  texture; // (Новое)
 VERTEX  vertex[3];
} TRIANGLE;
```

Изменения кода SetupWorld

```
for (int loop = 0; loop < numtriangles; loop++)
{
 readstr(filein,oneline); // (Новое)
 sscanf(oneline, "%i\n", &sector1.triangle[loop].texture);  // (Новое)
 for (int vert = 0; vert < 3; vert++)
 {
```

Изменения кода DrawGLScene

```
 // Для каждого треугольника
 for (int loop_m = 0; loop_m < numtriangles; loop_m++)
 {
  // (Модифицировано)
  glBindTexture(GL_TEXTURE_2D, texture[sector1.triangle[loop_m].texture]);
  glBegin(GL_TRIANGLES);
```

В функцию LoadGLTextures добавлена загрузка дополнительных текстур

```
int LoadGLTextures()          // Загрузка картинок и конвертирование их в текстуры
{
 int Status=FALSE;           // Статус
 AUX_RGBImageRec *TextureImage[5];         // Массив текстур
 memset(TextureImage,0,sizeof(void *)*2);  // Инициализация указателей NULL
```

```
 if( (TextureImage[0]=LoadBMP("Data/floor1.bmp")) &&  // Загрузка текстуры пола
  (TextureImage[1]=LoadBMP("Data/light1.bmp"))&&     // Загрузка текстуры освещения
  (TextureImage[2]=LoadBMP("Data/rustyblue.bmp"))&&  // Стены
  (TextureImage[3]=LoadBMP("Data/crate.bmp")) &&     // решетки
  (TextureImage[4]=LoadBMP("Data/weirdbrick.bmp")))  // потолок
 {
  Status=TRUE;
  glGenTextures(5, &texture[0]);        // Создание текстур
  for (int loop1=0; loop1<5; loop1++)
  {
   glBindTexture(GL_TEXTURE_2D, texture[loop1]);
   glTexImage2D(GL_TEXTURE_2D, 0, 3, TextureImage[loop1]->sizeX,
   TextureImage[loop1]->sizeY, 0,
     GL_RGB, GL_UNSIGNED_BYTE, TextureImage[loop1]->data);
   glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
   glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
  }
  for (loop1=0; loop1<5; loop1++)
  {
   if (TextureImage[loop1]->data)
   {
     free(TextureImage[loop1]->data);
   }
   free(TextureImage[loop1]);
  }
 }
 return Status; // Вернуть статус
}
```

Теперь вы можете использовать возможности DirectInput. Я потратил много времени на написание этого урока и надеюсь, что он вам пригодиться. Спасибо, что потратили свое время на чтение этого урока!

© Justin Eslinger (BlackScar)
blackscar@ticz.com
http://members.xoom.com/Blackscar/

Урок X2. Отсечение по пирамиде видимости в OpenGL.

Введение:

Представьте, что вы пишете программу, в которой можно бродить по виртуальному миру, который битком набит деревьями, зданиями, машинами, людьми и т.д. Интересно, как бы выглядела ваша функция рисования этого мира?

Проще всего – это в цикле прорисовывать каждый из этих объектов. В конце концов, OpenGL отсекает все сам, поэтому вам не придется заботиться об объектах за кадром. Когда все будет завершено, выглядеть это будет просто замечательно.

Конечно, если у вас немного объектов, этот метод годится. Но как только ваш мир немного разрастется, вы заметите, что скорость прорисовки будет стремительно падать.

Обычно в каждый момент только малая доля объектов в вашем мире видна. Если бы мы могли определить, виден тот или иной объект до его прорисовки, мы бы могли его просто пропустить и не посылать лишних данных процессору.

Один из способов это сделать называется отсечением по пирамиде видимости (frustum culling).

Определения:

Прежде, чем пойдем дальше, давайте ответим на несколько простых вопросов:

Что такое view frustum?

View frustum (пирамида видимости) – это часть пространства, в которой находятся все объекты, видимые из данной точки в данный момент. Она определяется шестью гранями усеченной пирамиды (т.е. пирамиды со срезанной вершиной). Если какая-то точка находится внутри пирамиды видимости, ее видно. Если вне пирамиды, значит, эту точку не видно.

[Обратите внимание – я говорю, что точка видима, хотя это не совсем так. Ее может закрывать другой объект, но она по крайней мере в поле зрения.]

Что такое - плоскость?

(прим. перев. – интересно было бы посмотреть на программиста, который не знает, что такое плоскость J)

Плоскость можно представить как бесконечно широкий и длинный лист бумаги. Любая точка пространства либо принадлежит плоскости, либо «спереди» от плоскости, либо «за» плоскостью.

Плоскость определяется четырьмя числами: A,B,C и D, где {A,B,C} – вектор нормали к этой плоскости, а D – расстояние до начала координат. Если вы вообще ничего из этого не понимаете, не падайте духом – на самом деле чтоб это использовать, понимать этого не надо.

Профминимум

Итак, прежде всего, нам надо знать: какие числа определяют текущую пирамиду видимости.

Вычислять это вручную может быть достаточно сложно. К счастью, приложив минимум усилий, мы можем заставить OpenGL сделать это за нас. Все, что нам надо - это извлечь текущие матрицы PROJECTION и MODELVIEW, скомбинировать их и узнать нужные значения.

Прежде всего, договоримся, что мы будем хранить значения задающее усеченную пирамиду видимости в глобальной переменной:

float frustum[6][4];

Это - двумерный массив 6*4 (шесть плоскостей, для каждой четыре числа: A, B, C, и D).

Теперь функция, которая извлекает числа и заполняет массив. Вам придется вызывать эту функцию для каждого кадра, строго перед рисованием.

```
ExtractFrustum()
{
  float   proj[16];
  float   modl[16];
  float   clip[16];
  float   t;

  /* Узнаем текущую матрицу PROJECTION */
  glGetFloatv( GL_PROJECTION_MATRIX, proj );

  /* Узнаем текущую матрицу MODELVIEW */
  glGetFloatv( GL_MODELVIEW_MATRIX, modl );

  /* Комбинируем матрицы(перемножаем) */
  clip[ 0] = modl[ 0] * proj[ 0] + modl[ 1] * proj[ 4] + modl[ 2] * proj[ 8] + modl[ 3] * proj[12];
  clip[ 1] = modl[ 0] * proj[ 1] + modl[ 1] * proj[ 5] + modl[ 2] * proj[ 9] + modl[ 3] * proj[13];
  clip[ 2] = modl[ 0] * proj[ 2] + modl[ 1] * proj[ 6] + modl[ 2] * proj[10] + modl[ 3] * proj[14];
  clip[ 3] = modl[ 0] * proj[ 3] + modl[ 1] * proj[ 7] + modl[ 2] * proj[11] + modl[ 3] * proj[15];
    clip[ 4] = modl[ 4] * proj[ 0] + modl[ 5] * proj[ 4] + modl[ 6] * proj[ 8] + modl[ 7] * proj[12];
  clip[ 5] = modl[ 4] * proj[ 1] + modl[ 5] * proj[ 5] + modl[ 6] * proj[ 9] + modl[ 7] * proj[13];
```

```
clip[ 6] = modl[ 4] * proj[ 2] + modl[ 5] * proj[ 6] + modl[ 6] * proj[10] + modl[ 7] * proj[14];
clip[ 7] = modl[ 4] * proj[ 3] + modl[ 5] * proj[ 7] + modl[ 6] * proj[11] + modl[ 7] * proj[15];
    clip[ 8] = modl[ 8] * proj[ 0] + modl[ 9] * proj[ 4] + modl[10] * proj[ 8] + modl[11] * proj[12];
clip[ 9] = modl[ 8] * proj[ 1] + modl[ 9] * proj[ 5] + modl[10] * proj[ 9] + modl[11] * proj[13];
clip[10] = modl[ 8] * proj[ 2] + modl[ 9] * proj[ 6] + modl[10] * proj[10] + modl[11] * proj[14];
clip[11] = modl[ 8] * proj[ 3] + modl[ 9] * proj[ 7] + modl[10] * proj[11] + modl[11] * proj[15];
    clip[12] = modl[12] * proj[ 0] + modl[13] * proj[ 4] + modl[14] * proj[ 8] + modl[15] * proj[12];
clip[13] = modl[12] * proj[ 1] + modl[13] * proj[ 5] + modl[14] * proj[ 9] + modl[15] * proj[13];
clip[14] = modl[12] * proj[ 2] + modl[13] * proj[ 6] + modl[14] * proj[10] + modl[15] * proj[14];
clip[15] = modl[12] * proj[ 3] + modl[13] * proj[ 7] + modl[14] * proj[11] + modl[15] * proj[15];

/* Находим A, B, C, D для ПРАВОЙ плоскости */
frustum[0][0] = clip[ 3] - clip[ 0];
frustum[0][1] = clip[ 7] - clip[ 4];
frustum[0][2] = clip[11] - clip[ 8];
frustum[0][3] = clip[15] - clip[12];

/* Приводим уравнение плоскости к нормальному виду */
t = sqrt( frustum[0][0] * frustum[0][0] + frustum[0][1] * frustum[0][1] + frustum[0][2] * frustum[0][2] );
frustum[0][0] /= t;
frustum[0][1] /= t;
frustum[0][2] /= t;
frustum[0][3] /= t;

/* Находим A, B, C, D для ЛЕВОЙ плоскости */
frustum[1][0] = clip[ 3] + clip[ 0];
frustum[1][1] = clip[ 7] + clip[ 4];
frustum[1][2] = clip[11] + clip[ 8];
frustum[1][3] = clip[15] + clip[12];

/* Приводим уравнение плоскости к нормальному виду */
t = sqrt( frustum[1][0] * frustum[1][0] + frustum[1][1] * frustum[1][1] + frustum[1][2] * frustum[1][2] );
frustum[1][0] /= t;
frustum[1][1] /= t;
frustum[1][2] /= t;
frustum[1][3] /= t;

/* Находим A, B, C, D для НИЖНЕЙ плоскости */
frustum[2][0] = clip[ 3] + clip[ 1];
frustum[2][1] = clip[ 7] + clip[ 5];
frustum[2][2] = clip[11] + clip[ 9];
frustum[2][3] = clip[15] + clip[13];

/* Приводим уравнение плоскости к нормальному */
t = sqrt( frustum[2][0] * frustum[2][0] + frustum[2][1] * frustum[2][1] + frustum[2][2] * frustum[2][2] );
frustum[2][0] /= t;
frustum[2][1] /= t;
frustum[2][2] /= t;
frustum[2][3] /= t;

/* ВЕРХНЯЯ плоскость */
frustum[3][0] = clip[ 3] - clip[ 1];
frustum[3][1] = clip[ 7] - clip[ 5];
frustum[3][2] = clip[11] - clip[ 9];
frustum[3][3] = clip[15] - clip[13];

/* Нормальный вид */
t = sqrt( frustum[3][0] * frustum[3][0] + frustum[3][1] * frustum[3][1] + frustum[3][2] * frustum[3][2] );
frustum[3][0] /= t;
frustum[3][1] /= t;
frustum[3][2] /= t;
frustum[3][3] /= t;
```

```
  /* ЗАДНЯЯ плоскость */
  frustum[4][0] = clip[ 3] - clip[ 2];
  frustum[4][1] = clip[ 7] - clip[ 6];
  frustum[4][2] = clip[11] - clip[10];
  frustum[4][3] = clip[15] - clip[14];

  /* Нормальный вид */
  t = sqrt( frustum[4][0] * frustum[4][0] + frustum[4][1] * frustum[4][1] + frustum[4][2] * frustum[4][2] );
  frustum[4][0] /= t;
  frustum[4][1] /= t;
  frustum[4][2] /= t;
  frustum[4][3] /= t;

  /* ПЕРЕДНЯЯ плоскость */
  frustum[5][0] = clip[ 3] + clip[ 2];
  frustum[5][1] = clip[ 7] + clip[ 6];
  frustum[5][2] = clip[11] + clip[10];
  frustum[5][3] = clip[15] + clip[14];

  /* Нормальный вид */
  t = sqrt( frustum[5][0] * frustum[5][0] + frustum[5][1] * frustum[5][1] + frustum[5][2] * frustum[5][2] );
  frustum[5][0] /= t;
  frustum[5][1] /= t;
  frustum[5][2] /= t;
  frustum[5][3] /= t;
}
```

Уууф. Многовато. Как-то не очень похоже, что из-за этого программа будет работать быстрее, да? Будет, поверьте. Нам придется вызывать эту функцию только один раз для кадра, так что не волнуйтесь из-за этих громоздких функций типа sqrt() и вообще математики.

Точка лежит в пирамиде?

Хорошо, мы наконец-то получили уравнения плоскостей, как же мы будем проверять, видим объект или нет? Давайте начнем с одной точки.

Учитывая то, что мы теперь знаем о плоскостях и точках, мы можем делать следующий вывод:

Точка в пирамиде, если она находится перед всеми плоскостями одновременно.

[Это так, потому что наши 6 векторов нормалей вида {A,B,C} лежат в пирамиде (т.е. все плоскости как бы смотрят внутрь пирамиды). Если бы было наоборот, то точка бы лежала ЗА всеми плоскостями.]

Уже неплохо! Это и есть основа всех последующих вычислений. Следующий шаг – понять, находится ли точка перед плоскостью или нет. Для этого нам надо посчитать расстояние от точки плоскости. Если расстояние положительно, значит, точка лежит перед плоскостью, отрицательна – значит за плоскостью.

Вот - формула для вычисления расстояния точки до плоскости:

distance = A * X + B * Y + C * Z + D

Где A, B, C, и D - четыре числа, которые определяют плоскость и X, Y, и Z - координаты точки.

Теперь мы можем написать функцию для проверки - видима точка или нет.

```
bool PointInFrustum( float x, float y, float z )
{
  int p;
  for( p = 0; p < 6; p++ )
    if( frustum[p][0] * x + frustum[p][1] * y + frustum[p][2] * z + frustum[p][3] <= 0 )
      return false;
  return true;
}
```

Функция просто пробегает в цикле по всем плоскостям, каждый раз считая расстояние от точки до каждой из плоскостей. Если точка не удовлетворяет условию хоть для одной плоскости, мы можем смело выходить и возвращать FALSE, так что в среднем тело цикла будет выполняться три раза.

Просто, не так ли?

Кстати, эта функция выкидывает все точки, находящиеся НА плоскостях – мне так больше нравится. Если вы не хотите их выкидывать, измените оператор "<=" на "<".

Ограничивающие тела

Перед тем, как мы перейдем к более сложным проверкам, давайте поговорим об ограничивающих телах.

Например, пусть у вас есть объект, состоящий из множества полигонов (т.е. многоугольников). В принципе, мы можем вызывать функцию PointInFrustum() для каждой вершины в модели, но сама эта проверка будет занимать больше времени, чем простая прорисовка всех объектов.

[Вообще-то, этот метод и не всегда бы правильно работал – если бы объект включает себя целиком пирамиду, то все точки объекта были бы вне пирамиды, но объект пришлось бы рисовать все равно.]

Что же мы делаем? Представьте сферу, которая включает в себя целиком объект. Теперь, вместо того, чтобы проверять на видимость сам объект, давайте лучше проверим эту сферу. Если хоть какая-то часть сферы видна, мы рисуем весь объект (саму сферу мы, конечно, не рисуем).

Поэтому мы можем решать: проверять каждую вершину из объекта или проверять одну, описанную вокруг объекта. Конечно, сфера может находиться в пирамиде, когда сам объект невидим, но пусть уж с этим OpenGL разбирается.

Такая сфера называется ограничивающей сферой. Это один из примеров ограничивающих тел. Другое распространенное ограничивающее тел – это параллелепипед. Вы можете использовать тела любых других форм, но сферы и параллелепипеды обычно самые удобные.

Идея состоит в том, чтобы вычислить ограничивающую сферу для каждого объекта, как только мы его загружаем в память. Чтобы хранить в памяти сферу, нужно хранить точку и ее радиус, всего четыре числа. Для куба – то же самое, для параллелепипеда нужно 8 точек, но это все равно проще, чем проверять сложный объект.

Так что же лучше, куб или сфера? Когда как. Мы вернемся к этому вопросу позже.

Эта сфера в пирамиде?

Теперь мы знаем – надо проверять сферу, но как? Да почти так же, как точку:

Here you go:

```
bool SphereInFrustum( float x, float y, float z, float radius )
{
  int p;

  for( p = 0; p < 6; p++ )
     if( frustum[p][0] * x + frustum[p][1] * y + frustum[p][2] * z + frustum[p][3] <= -radius )
        return false;
  return true;
}
```

В качестве параметров мы передаем X, Y и Z центра сферы и ее радиус. Единственное отличие в самой проверке от проверки точки – то, что мы сравниваем расстояние с радиусом, а не с нулем.

Забавный вариант

Посмотрите на этот вариант функции SphereInFrustum():

```
float SphereInFrustum( float x, float y, float z, float radius )
{
  int p;
  float d;

  for( p = 0; p < 6; p++ )
  {
    d = frustum[p][0] * x + frustum[p][1] * y + frustum[p][2] * z + frustum[p][3];
    if( d <= -radius )
      return 0;
  }
  return d + radius;
}
```

Это почти то же самое, только мы возвращаем 0, если сфера вне пирамиды, иначе мы возвращаем радиус плюс расстояние до последней проверенной плоскости.
Зачем нам это нужно? Последняя плоскость пирамиды – это ближняя грань пирамиды, поэтому мы сразу получаем расстояние от камеры до объекта.
Это здорово, потому что мы можем использовать это, чтобы изменять уровень детализации. Если объект очень близко, нам нужно будет рисовать очень много полигонов, а если он далеко, то бы обойдемся и менее детальным изображением. И это все не будет тратить дополнительного времени – все равно мы это вычисляем!
[Заметьте, что просто использовать расстояние до объекта, чтобы регулировать детализацию - это не всегда правильно. Может, вам захочется учитывать и радиус сферы тоже. Например, очень большой объект, даже если он находится далеко, придется рисовать тщательно.]

Этот параллелепипед в пирамиде?

Точка не может превосходить по размеру пирамиду, да и тест со сферой работает правильно, даже если эта сфера включает в себя целиком пирамиду. Но с параллелепипедом будет немного сложнее.
Мы, конечно, можем проверить все 8 вершин параллелепипеда, но если он включает в себя целиком пирамиду, то все точки будут вне пирамиды, но объект будет виден.

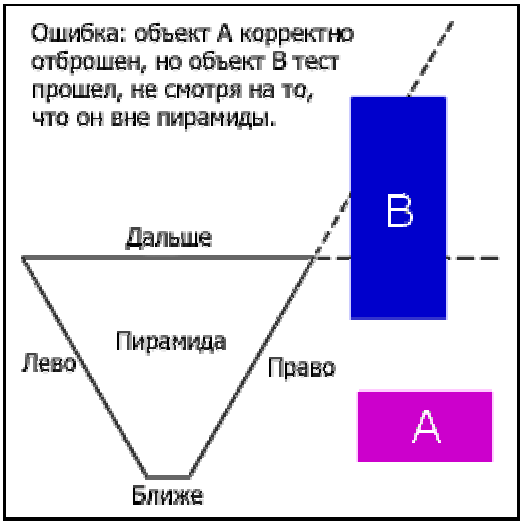
В данном примере мы возьмем частный случай параллелепипеда – куб.

Вот это - (и это работает, даже если пирамида целиком внутри куба):

```
bool CubeInFrustum( float x, float y, float z, float size )
{  int p;

  for( p = 0; p < 6; p++ )
  {
    if( frustum[p][0] * (x - size) + frustum[p][1] * (y - size) + frustum[p][2] * (z - size) + frustum[p][3] > 0 )
      continue;
    if( frustum[p][0] * (x + size) + frustum[p][1] * (y - size) + frustum[p][2] * (z - size) + frustum[p][3] > 0 )
      continue;
    if( frustum[p][0] * (x - size) + frustum[p][1] * (y + size) + frustum[p][2] * (z - size) + frustum[p][3] > 0 )
      continue;
    if( frustum[p][0] * (x + size) + frustum[p][1] * (y + size) + frustum[p][2] * (z - size) + frustum[p][3] > 0 )
      continue;
    if( frustum[p][0] * (x - size) + frustum[p][1] * (y - size) + frustum[p][2] * (z + size) + frustum[p][3] > 0 )
      continue;
    if( frustum[p][0] * (x + size) + frustum[p][1] * (y - size) + frustum[p][2] * (z + size) + frustum[p][3] > 0 )
      continue;
    if( frustum[p][0] * (x - size) + frustum[p][1] * (y + size) + frustum[p][2] * (z + size) + frustum[p][3] > 0 )
      continue;
    if( frustum[p][0] * (x + size) + frustum[p][1] * (y + size) + frustum[p][2] * (z + size) + frustum[p][3] > 0 )
      continue;
    return false;
  }
  return true;
}
```

В качестве аргументов мы передаем функции координаты центра куба и расстояние, которое равно половине ребра куба (по аналогии со сферой). Затем мы проверяем каждую вершину куба с каждой плоскостью пирамиды. Как только мы находим вершину, находящуюся спереди какой-то из плоскостей, мы сразу переходим к следующей плоскости, экономя время. Если мы проверили, все восемь вершин и оказалось, что они находятся за плоскостью, мы сразу можем выходить, потому что куб явно невидим. Если оказалось, что перед каждой плоскостью пирамиды находится хотя бы по одной вершине куба, значит, он видим.
ПРИМЕЧАНИЕ: Функция, приведенная выше, будет иногда выдавать ошибочные результаты: TRUE вместо FALSE, т.е. объект будет виден, хотя он вне пирамиды. Это происходит, когда все вершины куба находятся не за плоскостями пирамиды, но они все равно вне пирамиды. Поэтому иногда вам придется прорисовывать ненужные объекты. Но обычно такое расположение встречается достаточно редко, и, соответственно, не сильно сказывается на общей скорости.
Чтобы проверка была абсолютно правильной, нужно также проверять расположение 8 вершин пирамиды относительно шести граней ограничивающего параллелепипеда. Если плоскости параллелепипеда располагаются параллельно осям, то можно обойтись простыми проверками больше-меньше вместо проверок плоскостей параллелепипеда. В любом случае, это будет упражнением для менее ленивых, чем я ;-)



Смешанные тесты

Функции, приведенные выше, проверяли, находится ли хоть какая-нибудь часть ограничивающего тела в пирамиде или нет. Эти функции подходят для большинства случаев, но иногда нужно знать: находится ли ограничивающее тело целиком в пирамиде или только частично.
Зачем это нужно? Например, пусть у вас есть ограничивающее тело, включающее в себя много других ограничивающих тел, они в свою очередь тоже включают себя ограничивающие тела, и так далее. Если вы проверите какое-то большое ограничивающее тело, и оно окажется вне пирамиды, значит, все внутренние тела тоже не видны, значит, их проверять не надо. Если тело полностью внутри пирамиды, то и все внутренние тела тоже внутри, и опять их проверять не требуется. То есть, мы будем рекурсивно проверять внутренние объекты только в случае, если тело находится частично в пирамиде.
Вот версия функции SphereInFrusum(), которая возвращает 0, если сфера полностью вне пирамиды, 1 если частично и 2 или полностью внутри.

```
int SphereInFrustum( float x, float y, float z, float radius )
{   int p;     int c = 0;     float d;

  for( p = 0; p < 6; p++ )
  {
    d = frustum[p][0] * x + frustum[p][1] * y + frustum[p][2] * z + frustum[p][3];
    if( d <= -radius )
      return 0;
    if( d > radius )
      c++;
  }
  return (c == 6) ? 2 : 1;
}
```

Как и в предыдущих функциях, мы выходим, если сфера находится полностью за какой-либо из плоскостей, но теперь мы еще проверяем, пересекает ли она эту плоскость. Если нет, то мы просто прибавляем единицу к счетчику. Тогда в конце мы можем просто проверить – если сфера не пересекает ни одну из плоскостей, т.е. счетчик равен шести, то сфера полностью внутри пирамиды.

Вот версия CubeInFrustum(), делающая то же самое.

```
int CubeInFrustum( float x, float y, float z, float size )
{
  int p;
  int c;
  int c2 = 0;

  for( p = 0; p < 6; p++ )
  {
    c = 0;
    if( frustum[p][0] * (x - size) + frustum[p][1] * (y - size) + frustum[p][2] * (z - size) + frustum[p][3] > 0 )
      c++;
    if( frustum[p][0] * (x + size) + frustum[p][1] * (y - size) + frustum[p][2] * (z - size) + frustum[p][3] > 0 )
      c++;
    if( frustum[p][0] * (x - size) + frustum[p][1] * (y + size) + frustum[p][2] * (z - size) + frustum[p][3] > 0 )
      c++;
    if( frustum[p][0] * (x + size) + frustum[p][1] * (y + size) + frustum[p][2] * (z - size) + frustum[p][3] > 0 )
      c++;
    if( frustum[p][0] * (x - size) + frustum[p][1] * (y - size) + frustum[p][2] * (z + size) + frustum[p][3] > 0 )
      c++;
    if( frustum[p][0] * (x + size) + frustum[p][1] * (y - size) + frustum[p][2] * (z + size) + frustum[p][3] > 0 )
      c++;
    if( frustum[p][0] * (x - size) + frustum[p][1] * (y + size) + frustum[p][2] * (z + size) + frustum[p][3] > 0 )
      c++;
    if( frustum[p][0] * (x + size) + frustum[p][1] * (y + size) + frustum[p][2] * (z + size) + frustum[p][3] > 0 )
      c++;
    if( c == 0 )
      return 0;
    if( c == 8 )
      c2++;
  }
  return (c2 == 6) ? 2 : 1;
}
```

Эта функция проверяет восемь вершин относительно каждой из граней, считая, сколько из вершин находятся перед гранью. Если нет таких вершин, то мы можем сразу выходить, т.к. куб находится полностью вне пирамиды. Если все вершины находятся перед плоскостью, мы увеличиваем счетчик и в конце его сравниваем с 6.

Можно также возвращать байт – каждый бит отвечает за какую-то плоскость пирамиды. Устанавливаем бит в единицу, если тело полностью находится перед соответствующей плоскостью, и оставить бит равным нулю в противном случае. Если в конце этот байт равен нулю, значит, тело вне пирамиды. Если он равен 255, значит, , считая, сколько из вершин находятся перед гранью. ьюроверяем,тело полностью внутри пирамиды. Иначе – только частично внутри.

Сферы ПРОТИВ параллелепипедов

Голосов за использование ограничивающих сфер примерно столько же, сколько за параллелепипеды. Но мне кажется, что лучше использовать сферы, когда это возможно. Вот почему:

- Проверка сферы на видимость более правильная, чем у параллелепипеда – они никогда не возвращает неправильного значения.
- Проверка сферы проходит быстрее всего: примерно 3 операции сравнения, тогда как в случае с кубом это как минимум 6 сравнений, а как максимум – все 48!
- Если объект вращается вокруг центра ограничивающей сферы, сферу пересчитывать не надо. А вот если мы ограничиваем объект кубом, то некоторые части объекта могут «вылезти» за пределы куба.
- Сфера занимает меньше памяти. Конечно, если вы используете куб, то вы тоже можете обойтись четырьмя числами, но вот в случае с параллелепипедом вам нужно будет уже 8 чисел.

Правда, у сфер тоже есть один недостаток. Представьте, что объект длинный и тонкий, как кусок провода. Ограничивающую сферу придется строить очень большую, будет много пустого места в сфере. Если у вас будет много таких объектов, то вам придется прорисовывать много ненужного.

Обойти это можно несколькими способами. Один из них – это использовать длинный тонкий ограничивающий параллелепипед вместо сферы. Пустого места в ограничивающем теле будет мало, правда, и проверка займет больше времени.

Другой способ – это разбить объект на множество маленьких кусочков, и для каждого из них создать собственную ограничивающую сферу. И вообще, вы можете применять этот способ, если у вас есть просто большой или сложный объект.

Есть еще один способ – не разбивать объект на части, а определить несколько ограничивающих сфер для него. Если хоть одна будет видна, рисуем весь объект. Так как сферы в этом случае могут перекрываться, ускорит эта проверка весь процесс или нет, зависит от объекта.

В общем, решение за вами. Может, вам захочется использовать некую комбинацию методов.

Испытание Треугольников и Других Многоугольников

Меня часто спрашивают, как проверить принадлежность многоугольника пирамиде. Действительно, это то же самое, что и CubeInFrustum () , но для произвольного числа точек. Действуем точно так же: если все точки находятся за какой-нибудь плоскостью пирамиды, значит, многоугольник невидим (конечно, мы предполагаем, что многоугольники выпуклые).

[Примечание: Иногда такая функция возвращает TRUE вместо FALSE – я объяснял это в CubeInFrustum()]

Вот функция:

```
bool PolygonInFrustum( int numpoints, POINT* pointlist )
{
  int f, p;

  for( f = 0; f < 6; f++ )
  {
    for( p = 0; p < numpoints; p++ )
    {
      if( frustum[f][0] * pointlist[p].x + frustum[f][1] * pointlist[p].y + frustum[f][2] * pointlist[p].z + frustum[f][3] > 0 )
        break;
    }
    if( p == numpoints )
      return false;
  }
  return true;
}
```

В качестве аргумента мы передаем функции количество проверяемых точек и указатель на сам массив точек (структура POINT в данном случае должна содержать координаты X, Y и Z, тогда как существующая структура содержит только две координаты). Мы проверяем расположение точек относительно каждой из граней пирамиды и обрываем проверку плоскости, если какая-то из точек находится перед текущей плоскостью. Если все точки находятся за какой-либо из граней, мы сразу же возвращаем FALSE. Если в конце оказывается, что перед каждой из плоскостей находится хотя бы по точке, то многоугольник потенциально видим.

[Конечно, у этой функции есть свои применения, но в играх глупо проверять каждый многоугольник на видимость, легче предоставить это OpenGL. Лучше браковать целые группы многоугольников одним тестом.]

Оптимизация

Вы можете проделать следующие оптимизации, правда, неясно, будет ли заметна разница:

- Вы можете убрать приведение уравнений плоскостей к нормальному виду в ExtractFrustum(). Тогда полученные расстояния от точек до плоскостей будут отличаться от настоящих ровно в несколько раз. Тесты точки и параллелепипеда будут работать правильно, а тест сферы – нет. Если вы не используете ограничивающие сферы, вы можете таким образом избежать выполнения лишних команд, правда, это будет в большинстве случаев незаметно.

- На самом деле функцию ExtractFrustum() необходимо вызывать, только если положение камеры изменилось. Если от кадра к кадру ваши матрицы PROJECTION и MODELVIEW не меняются, лучше не пересчитывать уравнения плоскостей.
- Попробуйте развернуть циклы в функциях проверки.
Пример

nehex2.zip (48 Kb)
http://pmg.org.ru/nehe/nehex2.zip

Клавиши управления в примере:

Esc - Выход
Up – Переместить камеру вперед
Стрелка вниз – Переместить камеру назад
Стрелка влево - Повернуть камеру налево
Стрелка вправо - Повернуть камеру направо
U - Наклонить камеру вверх
D - Наклонить камеру вниз
Keypad + - Добавить объект (максимум 1000)
Keypad - - Убрать объект (минимум один)
W - Увеличить угол просмотра (FOV)
T – Уменьшить угол просмотра (FOV)
G - Включить/выключить сетку
M - Изменить режим (объекты могут быть точками, сферами и кубами)
C - Включить/выключить отсечение по пирамиде видимости

Примечания к примеру:

- Этот исходный текст основан на NeHe Production’s OpenGL tutorial (nehe.gamedev.net), и я старался сделать свой исходный код похожим на код NeHe.
- Вы, вероятно, заметили, что в режиме кубов программа работает быстрее. Это не из-за проверки пирамиды, это потому, что кубы легче рисовать.
- Включите отсечение и попробуйте разные значения FOV (т.е. угла обзора), и чем больше ваша пирамида, тем больше объектов надо прорисовывать.
- Главный цикл просто прокручивает все объекты. Было бы лучше хранить объекты в octree (восьмеричное дерево) или использовать какой-нибудь другой метод организации памяти, но все это вы сделаете сами.