Об авторе.
Феликс Кергер является студентом компьютерных технологий в Техническом университете Дармштадта, и занимается разработкой трехмерных приложений более 5 лет, использующих Ogre3D.
Он вел несколько переговоров по разработке 3D приложений в различных конференциях, и работает уже более 3-х лет в качестве помощника научного руководителя Института Фраунгоферова по изучению компьютерной графики. Он также работает в качестве внештатного журналиста, и предоставляет отчеты Game Developer конференции Европы.

Предисловие.
Создание трехмерных сцен является интересным и в тоже время сложным занятием, но процесс является чрезвычайно полезным и увлекательным.
Эта книга расскажет Вам, как вы можете сами создавать трехмерные приложения с помощью Ogre3D. Этот движок является одним из крупнейших в мире среди 3D движков, и позволяет своим пользователям создавать и легко взаимодействовать со сценой.
Эта книга не сможет показать все мельчайшие детали, она скорее обеспечит прочное введение, с которым Вы, как читатель, сможете начать использование Ogre 3D самостоятельно. После окончания прочтения книги, Вы сможете использовать документацию и вики, чтобы найти необходимую информацию и использовать более сложные технологии, которые не рассматриваются в этой книге.

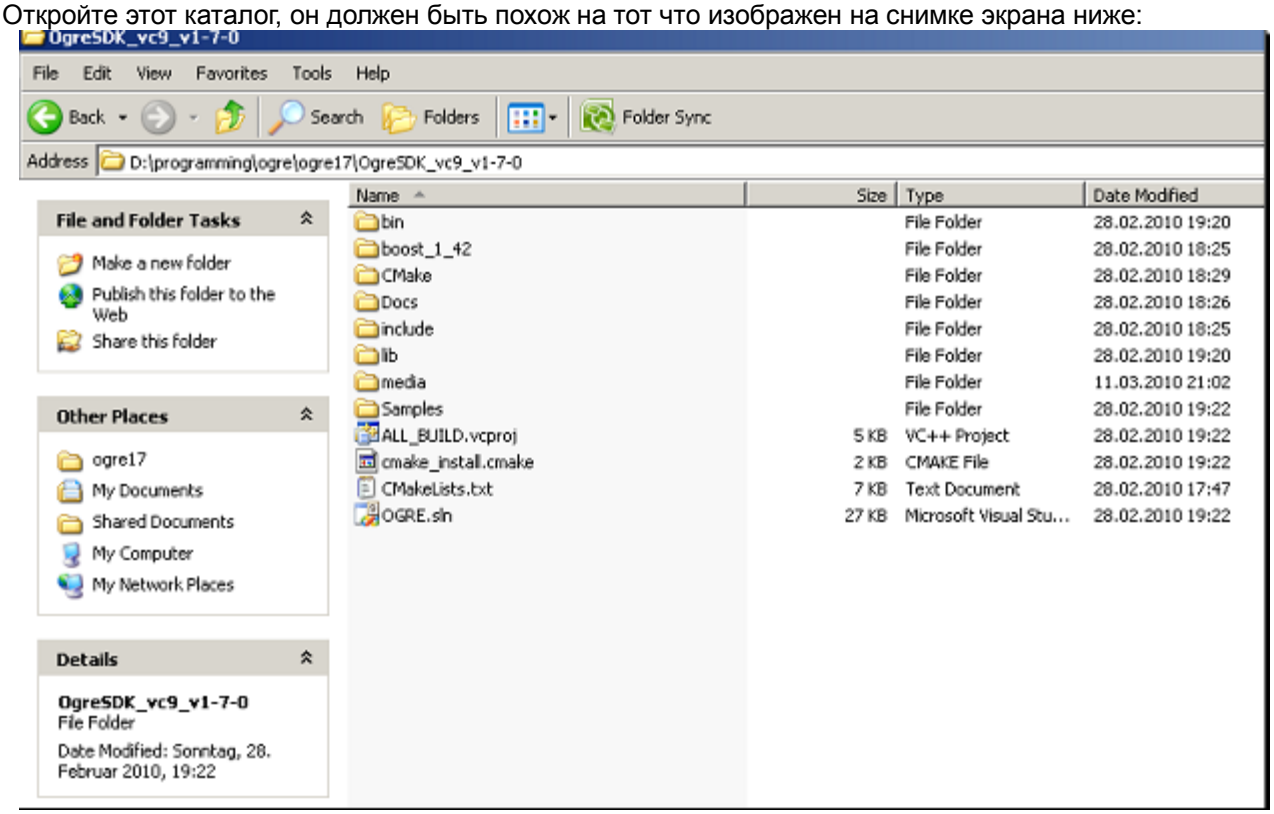
Что нужно иметь для прочтения и понимания книги.
Нужно четкое понимание C++ и как создавать приложения на основе этого языка программирования. Конечно нужен компилятор или среда разработки, чтобы компилировать примеры приложений. В этой книге будет использоваться Microsoft Visual Studio, но вы можете использовать любой другой редактор. Ваш компьютер должен иметь видео карту с 3D ускорением. Лучше всего если графическая карта поддерживает Direct X 9.0, потому что Ogre 3D мы будем его использовать.

ГЛАВА 1
Установка Ogre3D.

Загрузка и установка новой библиотеки является первым шагом обучения и его использования.
В этой главе мы сделаем следующее:

- Скачаем и установим Ogre 3D
- Настроим среду разработки
- Создадим нашу первую сцену

Итак, давайте начнем.
Загрузка и установка Ogre 3D
Перейдите на сайт http://www.ogre3d.org/download/sdk
Загрузите соответствующий пакет.
Скопируйте скачанный пакет туда где хотите расположить Ogre 3D
Дважды щелкните на пакет для запуска установщика.
Теперь у вас появится новый каталог с именем похожим на OgreSDK_vc9_v1-7-1.
Откройте этот каталог, он должен быть похож на тот что изображен на снимке экрана ниже:



Что мы сделали, Мы только что загрузили соответствующий пакет Ogre 3D SDK для нашей системы. Ogre 3D является кросс-платформенной системой рендеринга, поэтому есть много различных пакетов для разных платформ. После загрузки мы извлекли Ogre 3D.

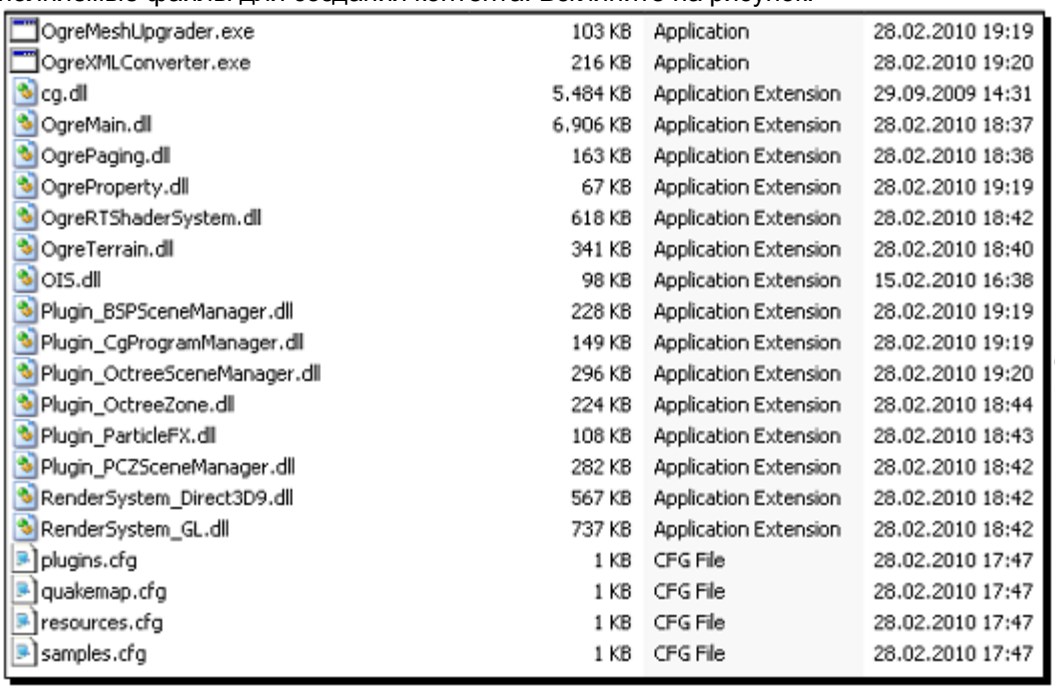
Различные версии Ogre 3D SDK.

Ogre поддерживает множество различных платформ, из-за этого существует множество различных пакетов для загрузки. Ogre 3D имеет несколько вариантов загрузки для Windows, один для MacOSX, и один пакет для Ubuntu. Существует также пакет для MinGW и для iPhone. Если Вы хотите, Вы можете загрузить исходный код и построить Ogre 3D самостоятельно. В этой главе основное внимание будет уделяться Windows. Если Вы хотите использовать другую операционную систему, Вы можете найти руководство на страницах вики http://www.ogre3d.org/wiki. Вики содержит подробные руководства по настройке среды для различных платформ. Остальная часть книги полностью зависит от платформы. Если вы хотите использовать другую систему разработки, не стесняйтесь, делайте это, на содержание книги это не повлияет, только что касается настройки компилятора.

Изучение SDK

Прежде чем приступить к сборке примеров, которые включены в SDK, давайте взглянем на него. Мы будем рассматривать SDK на платформе Windows. На Linux или MacOSX структура может выглядеть иначе. Сначала откроем каталог bin. Мы увидим два каталога, а именно debug и release. Тоже самое и в каталоге lib. Причина в том, что Ogre 3D поставляется нам в виде динамически подключаемой библиотеки. Это позволяет использовать отладочную сборку во время разработки. Когда мы закончим проект, построим его в режиме release, для того что бы получить полную производительность от Ogre 3D.

Когда мы откроем один из каталогов debug или release, мы увидим множество файлов dll, немного cfg, и exe. Исполняемые файлы для создания контента. Взгляните на рисунок:



OgreMeshUpgrader.exe	103 KB	Application	28.02.2010 19:19
OgreXMLConverter.exe	216 KB	Application	28.02.2010 19:20
cg.dll	5,484 KB	Application Extension	29.09.2009 14:31
OgreMain.dll	6,906 KB	Application Extension	28.02.2010 18:37
OgrePaging.dll	163 KB	Application Extension	28.02.2010 18:38
OgreProperty.dll	67 KB	Application Extension	28.02.2010 19:19
OgreRTShaderSystem.dll	618 KB	Application Extension	28.02.2010 18:42
OgreTerrain.dll	341 KB	Application Extension	28.02.2010 18:40
OIS.dll	98 KB	Application Extension	15.02.2010 16:38
Plugin_BSPSceneManager.dll	228 KB	Application Extension	28.02.2010 19:19
Plugin_CgProgramManager.dll	149 KB	Application Extension	28.02.2010 19:19
Plugin_OctreeSceneManager.dll	296 KB	Application Extension	28.02.2010 19:20
Plugin_OctreeZone.dll	224 KB	Application Extension	28.02.2010 18:44
Plugin_ParticleFX.dll	108 KB	Application Extension	28.02.2010 18:43
Plugin_PCZSceneManager.dll	282 KB	Application Extension	28.02.2010 18:42
RenderSystem_Direct3D9.dll	567 KB	Application Extension	28.02.2010 18:42
RenderSystem_GL.dll	737 KB	Application Extension	28.02.2010 18:42
plugins.cfg	1 KB	CFG File	28.02.2010 17:47
quakemap.cfg	1 KB	CFG File	28.02.2010 17:47
resources.cfg	1 KB	CFG File	28.02.2010 17:47
samples.cfg	1 KB	CFG File	28.02.2010 17:47

OgreMain.dll является самой главной библиотекой. Она составлена из исходного кода Ogre 3D и она будет загружаться с нашим приложением. Все библиотеки начинающиеся с префикса plugin_ являются плагинами для Ogre 3D, которые добавляют новые возможности в движок с помощью интерфейсов Ogre 3D. Это может быть практически что угодно, но обычно это используется для добавления новых функций, таких как улучшенная система частиц и менеджеры сцен. Это будет обсуждаться позже. Сообщество Ogre 3D создало большое количество плагинов, которые можно найти на вики. SDK включает в себя лишь наиболее используемые плагины. Далее в этой книге, мы узнаем, как использовать некоторые из них. Библиотеки в которых имя начинается с префикса RenderSystem_ обозначают системы визуализации для различных систем. В нашем случае это Direct3D9 и OpenGL. В дополнение к этим двум есть еще Direct3D10, Direct3D11, OpenGL ES (для встраиваемых систем).

Кроме исполняемых файлов и файлов библиотек, есть еще файлы cfg. CFG файлы это файлы конфигурации, Ogre 3D может загружать их при запуске. В Plugins.cfg просто перечислены все плагины, которые Ogre должен запустить при запуске. Это как правило Direct3D и OpenGL, и некоторые дополнительные менеджеры сцен, quakemap.cfg это конфигурационный файл необходимый для загрузки уровня из Quake3. Нам не нужен этот формат, но сделать это возможно. Resources.cfg содержит список всех ресурсов, таких как сетки, текстуры или анимация, которые Ogre 3D следует загрузить при запуске. Ogre 3D может загрузить ресурсы из каталога или из zip архива. Если мы посмотрим содержимое этого файла мы увидим чтото похожее на

Zip=../../media/packs/SdkTrays.zip
FileSystem=../../media/thumbnails

Zip – означает, что этот ресурс находится в zip архиве, а FileSystem означает, что ресурс находится в каталоге. Resources.cfg позволяет легко загружать новые ресурсы. Последний файл samples.cfg, мы не будем его использовать. Это все простой список необходимый для загрузки примеров для SampleBrowser. Но у нас еще нет примеров, давайте построим один из них.

Примеры Ogre 3D

Ogre 3D поставляется с большим количеством примеров, которые показывают различные эффекты и что можно сделать на основе этого движка. Прежде чем мы начнем работу, мы посмотрим на примеры, что бы иметь представление о том, что умеет Ogre 3D.

Пришло время построить примеры Ogre 3D.

Что бы получить первое впечатление о движке, мы построим примеры и посмотрим на них.

1. Перейдем в каталог Ogre 3D
2. Откроем файл Ogre3D.sln
3. Нажмем правой кнопкой на решении вызовем пункт построить
4. Visual Studio должен приступить к сборке примеров. Это может занять несколько минут, вы успеете налить себе чашечку чая.
5. Если все прошло нормально, перейдите в каталог Ogre3D/bin
6. Выполните SampleBrowser.exe
7. Вы должны увидеть на экране следующее:

Первое приложение с использованием Ogre 3D. В этой части мы создадим простое 3D приложение которое будет показывать нам одну модель.

Пришло время настроить проект и среду разработки.

Как и в любой другой библиотеке нам нужно настроить нашу среду разработки прежде чем мы приступим к написанию программы.

1. Создайте новый пустой проект
2. Создайте файл исходного кода и назовите его main.cpp
3. Добавьте функцию main

```
int main(void)
{
 return 0;
}
```

4. Подключите *#include "Ogre\ExampleApplication.h"*
5. Добавьте пути Путь_к_OgreSDK\include\; Путь_к_OgreSDK\boost_1_42; Путь_к_OgreSDK\boost_1_42\lib

6. Добавьте новый класс в main.cpp

```
class Example1 : public ExampleApplication
{
```

```
public:
void createScene()
{
}
};
```

7. Добавьте следующий код в верхней части функции main:
   ```
   Example1 app;
   app.go();
   ```
8. Добавьте пути для библиотек Ogre 3D Путь_к_OgreSDK\Lib\Debug
9. Свяжите с библиотеками OgreMain_d.lib и OIS_d.lib
10. Скомпилируйте проект
11. Установите приложению рабочий каталог Путь_к_OgreSDK\bin\debug
12. Запустите приложение. Вы должны увидить диалоговое окно настроек Ogre 3D

13. Нажмите OK и запустите приложение. Вы увидите черное окно. Нажмите ESCAPE для выхода из приложения.

Что мы сейчас сделали.

Мы создали наше первое Ogre 3D приложение. Для компиляции нам необходимо было установить различные включения и пути к библиотекам, чтобы компилятор смог найти Ogre 3D. Ogre 3D включает каталог со всеми заголовочными файлами и каталогом OIS. OIS расшифровывается как объектно-ориентированная система ввода/вывода, и ExampleApplication ее использует. OIS не является частью Ogre 3D, это самостоятельные проект, у него другие разработчики, она просто поставляется вместе с Ogre 3D.

Ogre 3D предоставляет поддержку потоков, поэтому мы включили boost. В противном случае мы не построим ни одно приложение на Ogre 3D.

Потом мы добавили путь к библиотекам. В данном случае мы используем библиотеки отладки, что бы ничего не случилось если что-то пойдет не так. Если мы захотим использовать release мы должны связать приложение с библиотеками OgreMain.lib и OIS.lib. В зависимости от режима debug или release, Ogre 3D использует различные файлы библиотек dll. Для того что бы наше приложение могло найти все библиотеки, мы в IDE задаем ей рабочий каталог с этими библиотеками.

ExampleApplication

Мы создали новый класс, Example1, который наследуется от ExampleApplication. ExampleApplication это класс, который предоставляется Ogre 3D SDK, для того чтобы ваше обучение было легче, предлагая более высокий уровень абстракции. ExampleApplication за нас устанавливает камеру, и загружает модели, мы можем перемещаться по нашей сцене. Для этого мы всеголишь унаследовали класс и переопределили функцию createScene(). Мы будем использовать этот класс для того чтобы

спасти нас от лишней работы, пока мы не очень хорошо знаем Ogre 3D. Позже мы разработаем свой каркас приложения.
В функции main мы создали новый экземпляр нашего приложения и приказали запуститься go(). При загрузке Ogre 3D загружает файлы конфигурации Ogre.cfg, resources.cfg и plugins.cfg. Если мы используем debug версии, каждый файл должен быть с префиксом _d на конце. Это полезно, потому что настройки могут отличаться в различных версиях debug и release. Ogre3D.cfg содержит настройки, которые мы выбираем в диалоговом окне, для того чтобы каждый раз не выбирать настройки, когда мы запускаем приложение. Ogre 3D SDK поставляется с большим количеством моделей и текстур, мы их будем использовать в этой книге. Resources.cfg указывает на их местоположение, обратите внимание , что пути относительные, вот почему нам нужно установить рабочий каталог.

Загрузка первой модели
У нас есть базовое приложение, но оно еще ничем не заполнено, это довольно скучно. Теперь мы загрузим объект для получения более интересной сцены.

Пришло время загрузить модель
Загружается модель очень просто. Нам просто нужно добавить две строчки кода.
Добавьте следующие две строчки кода в функцию createScene():
Ogre::Entity ent = mSceneMgr->createEntity("MyEntity","Sinbad.mesh");*
mSceneMgr->getRootSceneNode()->attachObject(ent);
Скомпилируйте приложение еще раз.
После запуска приложения вы увидите маленькую зелененькую фигурку.
Подойдите поближе с помощью мыши и клавиш WASD.
Закройте приложение.

Что мы сделали.
С помощью *mSceneMgr->createEntity("MyEntity", "Sinbad.mesh");* мы сказали движку, что хотим загрузить новый экземпляр модели sinbad.mesh. MsceneMgr является указателем на менеджер сцены Ogre 3D, созданный для нас в ExampleApplication. Для создания нового объекта, Ogre необходимо знать какой файл модели использовать, и мы можем задать новое имя ему при создании. Важно, чтобы имя было уникально, и не может использоваться дважды, иначе произойдет исключение. Если мы не указываем имя, Ogre это сделает за нас автоматически. Мы рассмотрим это позднее. Теперь у нас есть экземпляр модели и что бы отобразить ее, нам нужно прикрепить ее к сцене. Прикрепление объекта (entity) к сцене очень просто: *mSceneMgr-> getRootSceneNode () -> attachObject (ent);*
Эта строка передает наш объект сцене, что бы мы могли увидеть его. И что мы видим, Синбад, талисман, модель. Мы еще не раз увидим эту модель в нашей книге.

Подведем итоги

Мы узнали как работает Ogre 3D, какие библиотеки нам нужно связать и какие каталоги необходимо подключить. Кроме того мы научились использовать класс ExampleApplication, мы загрузили модель и отобразили ее. После этого введения мы узнаем больше о том как Ogre 3D организует сцены и как мы можем управлять сценой, все это в следующей главе.

ГЛАВА 2
РАБОТА СО СЦЕНОЙ В OGRE 3D

Эта глава познакомит нас с работой со сценой, и как мы можем использовать Ogre для создания сложных сцен.
В этой главе мы:

- узнаем три основные операции в 3D пространстве
- как организована сцена
- сможем работать в различных 3D пространствах

Итак начнем.

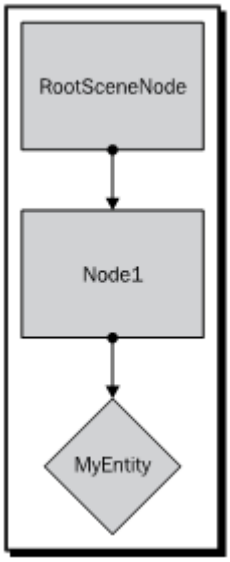
Создание узла сцены (Scene Node)
В предыдущей главе мы установили Ogre 3D, загрузили модель и передали ее нашей сцене. Теперь мы узнаем как создать новый узел (Node) сцены.

Пришло время создать узел сцены
Мы будем использовать код из главы 1, модифицировать его для создания новых узлов сцены. В старой версии нашего кода у нас было две строки
Ogre::Entity ent = mSceneMgr->createEntity("MyEntity","Sinbad.mesh");*
mSceneMgr->getRootSceneNode()->attachObject(ent);
Заменим ее следующей:
Ogre::SceneNode node = mSceneMgr->createSceneNode("Node1");*
Затем добавьте следующие две строки. Их порядок не повлияет на результат сцены:
mSceneMgr->getRootSceneNode()->addChild(node);
node->attachObject(ent);
Скомпилируйте и запустите приложение. Вы увидите тоже самое, что сделали в первой главе.

Что мы сделали.
Мы создали новый узел сцены с именем Node1. Затем мы разместили этот узел в корне сцены. Мосле этого мы прикрепили ранее созданную модель, чтобы ее было видно на вновь созданном узле сцены.

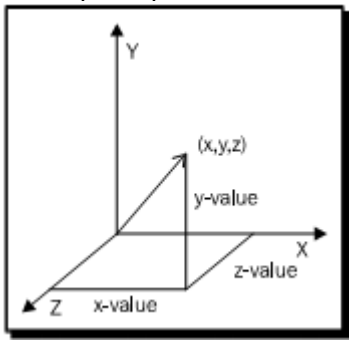
Как работает RootSceneNode
Вызов mSceneMgr->getRootSceneNode() возвращает корневой узел сцены. Этот узел является переменной сцены. Что бы чтото увидеть, нам нужно прикрепить это к корневому узлу сцены, который будет являться дочерним по отношению к корневому узлу сцены. Короче говоря, долна выстраиваться цепочка в виде родитель — потомок, начинающихся с корневого узла сцены. Такую организацию сцены использует Ogre 3D, ка бы дерево узлов, которое имеет один корень, и каждый узел может иметь узлы-потомки. Мы уже использовали эту особенность, когда вызывали mSceneMgr-> getRootSceneNode()->AddChild (node); Там мы добавили узел в качестве дочернего к корню. Непосредственно мы добавили еще один вид потомка node->attachObject(ent); У нас есть два различных вида объектов, мы можем добавить к сцене узел. Первый тип может быть добавлен в качестве потомка самих потомков. Второй тип, не является потомком и не может иметь потомков. Рассматривается как листья дерева. Есть множество вещей которые мы можем добавить к сцене, освещение, системы частиц, и т. д.

Позже мы узнаем как это использовать. Сейчас нам нужны только модели (entity), на текущей сцене это выглядит следующим образом:

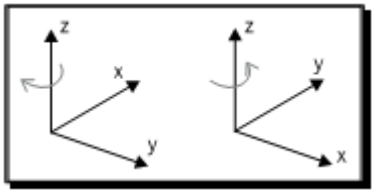


Первое что мы должны понимать, это что такое сцены и для чего они нужны. Сцена используется для представления различных частей, связанных друг с другом в 3D пространстве.

3D пространство
Ogre 3D является трехмерным графическим движком, поэтому мы должны знать базовые вещи. Основная составляющая - это вектора, которые представляются в виде x, y, z.



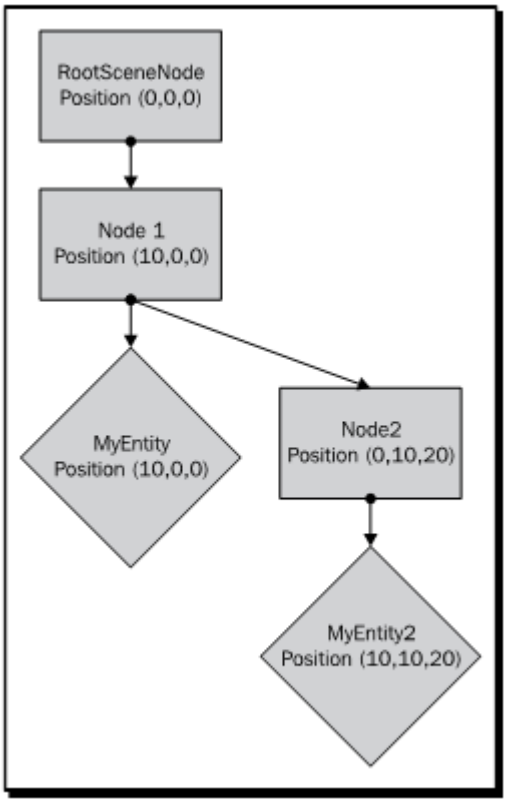
Каждая позиция в 3d пространстве, может быть представлена в виде такой тройки значений — вектором. Важно знать что существуют различные виды системы координат в трехмерном пространстве. Единственное различие в них это направление осей, и направление вращения. Широко используются две системы, а именно левая и правая. На следующем рисунке мы видим левую и правую систему координат:



Название левая или правая основано на том, что мы можем изобразить ориентацию осей с помощью левой или правой руки. Большой палец это ось абсцисс, указательный это ось ординат, *а средний ось. (дословно с англ.).* Мы должны держать нашу руку так, что бы большой и указательный пальцы находились под углом 90 градусов, а также средний по отношению к указательному тоже на 90 градусов. При использовании левой руки мы получаем левую систему координат, при использовании правой руки получаем правую систему координат.
Ogre использует правую систему координат.

Сцена.
Сцена является одним из наиболее часто используемых понятий в трехмерной графике. Мы уже обсуждали, что сцена имеет древовидную структуру. Но мы не оговорили самую важную функцию сцены. Это преобразования в трехмерном пространстве. Преобразование состоит из трех вещей.

Позиция это тройка значений X, Y и Z – которые описывают расположение объекта на сцене. Вращение задается при помощи кватернионов, это математическое понятие присущее вращению. Масштабирование производится очень легко, вектором X, Y, Z. Главное, что следут помнить — преобразование задается относительно родительского узла. Если мы изменим ориентацию родителей, то эти изменения затронут и потомков. Если мы передвинем родителя вдоль оси X, то потомки тоже передвинуться. Вам станет понятнее если вы взгляните на изображение:



Расположение MyEntity в этой сцене будет (10, 0, 0) и MyEntity2 будет на (10, 10, 20). Давайте попробуем это в Ogre 3D.

Настроим расположение узла сцены.
Добавьте следующую строку после создания сцены из предыдущего урока:
node->setPosition(10,0,0);
Создайте вторую модель, добавив следующую строку:
Ogre::Entity ent2 = mSceneMgr->createEntity("MyEntity2","Sinbad.Mesh");*
Создадим второй узел:
Ogre::SceneNode node2 = mSceneMgr->createSceneNode("Node2");*
Добавим второй узел к первому:
node->addChild(node2);
Настроим позицию второго узла
node2->setPosition(0,10,20);
Присоединим созданную модель ко второму узлу:
node2->attachObject(ent2);

Скомпилируйте программу и вы увидите 2 экземпляра Синбада:

Что мы сделали

Создали сцену. Первая новая функция setPosition, как не сложно догадаться, перемещает объект x, y, z. Имейте ввиду, что позиция задается относительно родительского узла. Мы добавили второй узел node2, к которому прикреплен MyEntity2. Мы хотели чтобы наш node2 был расположен в точке 10,10,20 так как node первый был в позиции 10,0,0 мы переместим node2 в точку 0,10,20. Когда обе позиции сложатся получится что node2 расположен в позиции 10,10,20.

Поворот узла сцены.

Мы уже умеем устанавливать расположение узла сцены. Теперь мы научимся вращать, и другому способу изменить положение узла сцены.

Пришло время научиться поворотам

Удалите весь код из createScene().

Сначала, создайте экземпляр Sinbad.mesh, затем создайте новый узел сцены (10,10,10). Потом прикрепите модель к узлу, а также прикрепите узел к корневому узлу сцены.

```
Ogre::Entity* ent = mSceneMgr->createEntity("MyEntity","Sinbad.mesh");
Ogre::SceneNode* node = mSceneMgr->createSceneNode("Node1");
node->setPosition(10,10,0);
mSceneMgr->getRootSceneNode()->addChild(node);
node->attachObject(ent);
```

Опять же создадим новый экземпляр модели, а также новый узел сцены и расположим в точке (10,0,0).

```
Ogre::Entity* ent2 = mSceneMgr->createEntity("MyEntity2","Sinbad.mesh");
Ogre::SceneNode* node2 = mSceneMgr->createSceneNode("Node2");
node->addChild(node2);
node2->setPosition(10,0,0);
```

Теперь добавьте следующие две строки, что бы повернуть модель, и присоединить объект к узлу сцены:

```
node2->pitch(Ogre::Radian(Ogre::Math::HALF_PI));
node2->attachObject(ent2);
```

Сделайте тоже самое, но на этот раз используйте функцию yaw вместо pitch, и используем функцию translate, вместо setPosition.

```
Ogre::Entity* ent3 = mSceneMgr->createEntity("MyEntity3","Sinbad.mesh");
Ogre::SceneNode* node3 = mSceneMgr->createSceneNode("Node3",);
node->addChild(node3);
node3->translate(20,0,0);
node3->yaw(Ogre::Degree(90.0f));
node3->attachObject(ent3);
```

Сделайте тоже самое, но с функцией roll:
Ogre::Entity ent4 = mSceneMgr->createEntity("MyEntity4","Sinbad.mesh");*
Ogre::SceneNode node4 = mSceneMgr->createSceneNode("Node4");*
node->addChild(node4);
node4->setPosition(30,0,0);
node4->roll(Ogre::Radian(Ogre::Math::HALF_PI));
node4->attachObject(ent4);
Скомпилируйте, у вас должно получиться следующее:

Что мы сделали.
Мы повторили код, который писали ранее, 4 раза и всегда изменяли мелкие детали. Первый просто повторили — ничего особенного, это всего лишь код что был раньше, это будет наша эталонная модель, что бы понять, что происходит с другими.
Во втором случае мы добавили следующие строки:
node2->pitch(Ogre::Radian(Ogre::Math::HALF_PI));
Функция pitch, вращает узел сцены вокруг оси X. Эта функция ожидает радиан для вращения, и мы использовали Ogre::Radian(Ogre::Math::HALF_PI), для поворота на 90 градусов. Далее мы заменили SetRotation на translate. SetRotation задает положение узла на сцене, а translate, задает новое положение, относительно нынешнего положения узла сцены. Если узел сцены имеет расположение 10,20,30 и мы вызываем SetPosition(30,20,10) то узел будет расположен в позиции 30,20,10, а если мы вызовем translate(30,20,10), узел станет располагаться в позиции 40,40,40. Это небольшая но важная разница, обе функции могут быть полезны при разных обстоятельствах. Например когда мы хотим позиционировать узел на сцене мы используем SetPosition(x,y,z). Однако если мы хотим переместить узел уже находившийся на сцене мы можем использовать translate(x,y,z).
Кроме того потом мы заменили pitch(Ogre::Radian(Ogre::Math::HALF_PI)) на yaw(Ogre::Degree(90.0f)). Функция yaw() вращает узел сцены вокруг оси Y. Затем вместо Radian мы использовали Degree. Естественно, pitch и yaw принимают параметры в виде радиана, тем не менее Ogre предоставляет функцию преобразования привычного вида угла к радиану.
Следующий шаг, вращает узел сцены по оси Z, roll(). Опять же мы могли использовать roll(Ogre::Degree(90.0f)), вместо roll(Ogre::Radian(Ogre::Math::HALF_PI)).
Программа при запуске показывает не повернутую модель и все три возможных варианта вращений. Левая модель не поворачивается, модель справа от левой вращается вокруг оси X, следующая вращается вокруг оси Y, и последняя вокруг оси Z. В каждом из этих случаев показано влияние различных функций. Одним словом pitch, yaw, roll вращает узел вокруг осей x, y, z. Мы можем использовать как Radian так и Degree.

Масштабирование узла сцены.
Мы уже рассмотрели две из трех основных операций, которые мы используем в нашей сцене. Пришло время рассмотреть масштабирование.

Пришло время масштабировать узел сцены.
Опять же используем код который писали ранее.
Удалите весь код из createScene() и вставьте следующий блок:
Ogre::Entity ent = mSceneMgr->createEntity("MyEntity","Sinbad.mesh");*
Ogre::SceneNode node = mSceneMgr->createSceneNode("Node1");*
node->setPosition(10,10,0);
mSceneMgr->getRootSceneNode()->addChild(node);
node->attachObject(ent);
Повторим создание модели:
Ogre::Entity ent2 = mSceneMgr->createEntity("MyEntity2","Sinbad.mesh");*
Теперь мы используем функцию, которая создаст узел сцены, и добавит его автоматически как потомка, далее сделаем тоже самое что и ранее:
Ogre::SceneNode node2 = node->createChildSceneNode("node2");*

node2->setPosition(10,0,0);
node2->attachObject(ent2);
Теперь вставьте после setPosition функцию для масштабирования:
node2->scale(2.0f,2.0f,2.0f);
Создайте новую модель:
Ogre::Entity ent3 = mSceneMgr->createEntity("MyEntity3","Sinbad.mesh");*
Теперь вызовем функцию создания потомка, только теперь с параметром:
Ogre::SceneNode node3 = node->createChildSceneNode("node3",Ogre::Vector3(20,0,0));*
После этого добавим масштабирование:
node3->scale(0.2f,0.2f,0.2f);
Скомпилируйте приложение, у вас должно получиться следующее:

Что мы сделали.
Мы создали сцену с масштабируемым узлом. Мы использовали новые функции для добавления потомка узлу: node->createChildSceneNode("node2"). Эта функция, является функцией-членом узла сцены и создает новый узел с заданным именем, и делает его потомком. Потом мы вызвали функцию scale(), которая принимает параметры x,y,z, в которой указывается на сколько узел должен быть масштабирован. Затем мы снова использовали функцию createChildSceneNode()б но теперь уже с параметром, который принимает x,y,z. Ogre 3D имеет класс для сохранения тройки x,y,z называется Ogre::Vector3. Кроме сохранения координат этот класс предлагает функции основных математических операций. Например следующие строки кода:
Ogre::SceneNode node2 = mSceneMgr->createSceneNode("Node2");*
node->addChild(node2);
node2->setPosition(20,0,0);
могут быть заменены на:
Ogre::SceneNode node2 = node->createChildSceneNode("Node2",Ogre::*
Vector3(20,0,0));
Очень удобно.
Кроме scale(), есть функция setScale(). Разница между этими функциями схожа с разницой функций setPosition() и translate().

Правильный подход для работы со сценой.
В этом разделе Вы узнаете, как можно реализовать некоторые задачи проще и понятней. Эта так же расширит ваши понятия о сценах.

Пришло время построить дерево узлов сцены.
На этот раз используем модель ниндзи.
1. Удалите весь код из createScene().

2. Создадим Синбада:
Ogre::Entity ent = mSceneMgr->createEntity("MyEntity","Sinbad.mesh");*
Ogre::SceneNode node = mSceneMgr->createSceneNode("Node1");*
node->setPosition(10,10,0);
mSceneMgr->getRootSceneNode()->addChild(node);
node->attachObject(ent)
3. Теперь создадим ниндзю который будет смотреть за Синбадом.
Ogre::Entity ent2 = mSceneMgr->createEntity("MyEntitysNinja","ninja.mesh");*
Ogre::SceneNode node2 = node->createChildSceneNode("node2");*
node2->setPosition(10,0,0);
node2->setScale(0.02f,0.02f,0.02f);
node2->attachObject(ent2);
4. Скомпилируйте и запустите приложение, и вы увидите ниндзю расположенного возле левой руки Синбада.

5. Изменим позицию на (40,10,0)
node->setPosition(40,10,0);
6. И повернем на 180 градусов
node->yaw(Ogre::Degree(180.0f));
7. Скомпилируем и запустим

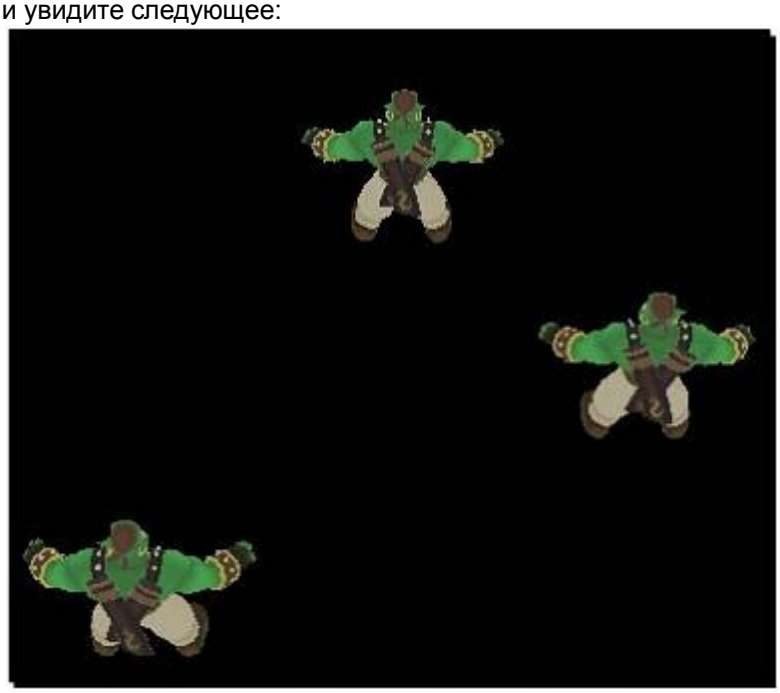
Что мы сделали
Мы сделали ниндзю, который следит за каждым шагом Синбада. Это получилось благодаря тому, что мы сделали ниндзю, потомком Синбада. Когда мы хотели передвинуть Синбада мы использовали его узел сцены. Эта же трансформация повлияла и на ниндзю, потомучто изменения родителя влияют на потомка.

Различные пространства сцены.
В этой части мы узнаем о том что есть различные пространства в кадре и как их можно использовать.

Пришло время перемещаться в мировом пространстве.
Мы собираемся переместить объект, но по другому, не так как делали ранее.
1. Удаляем весь код из createScene()
2. Создаем объект
```
Ogre::Entity* ent = mSceneMgr->createEntity("MyEntity","Sinbad.mesh");
Ogre::SceneNode* node = mSceneMgr->createSceneNode("Node1");
node->setPosition(0,0,400);
node->yaw(Ogre::Degree(180.0f));
mSceneMgr->getRootSceneNode()->addChild(node);
node->attachObject(ent);
```
3. Создадим два новых экземпляра и переместим на 0,0,10
```
Ogre::Entity* ent2 = mSceneMgr->createEntity("MyEntity2","Sinbad.mesh");
Ogre::SceneNode* node2 = node->createChildSceneNode("node2");
node2->setPosition(10,0,0);
node2->translate(0,0,10);
node2->attachObject(ent2);
Ogre::Entity* ent3 = mSceneMgr->createEntity("MyEntity3","Sinbad.mesh");
Ogre::SceneNode* node3 = node->createChildSceneNode("node3");
node3->setPosition(20,0,0);
node3->translate(0,0,10);
node3->attachObject(ent3);
```
4. Запустите приложение и получите следующее:

5. Замените строку node3->translate(0,0,10);
на
```
node3->translate(0,0,10,Ogre::Node::TS_WORLD);
```

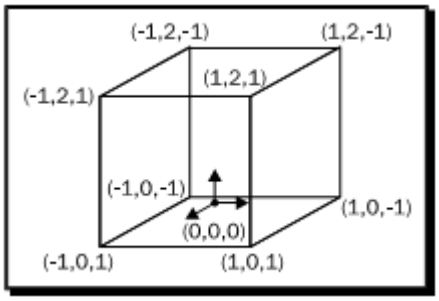
6. Снова запустите, и увидите следующее:



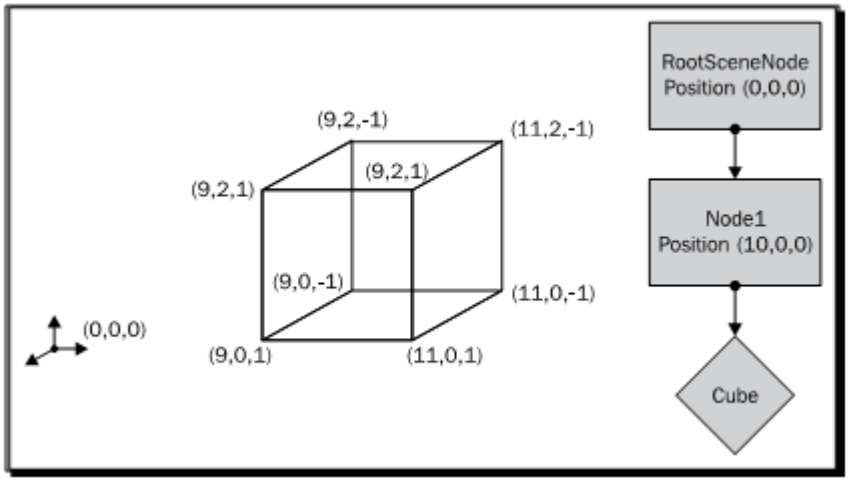
Что мы сделали.

Мы использовали новый параметр в функции translate.

Причина того что модель перемещается по разному, в том что мы указали в функции translate, перемещаться относительно мирового пространства, а не локального (относительно родителя). Всего есть 3 вида пространств. Мировое пространство, пространство вышестоящего и локальное. Взгляните на диаграмму:

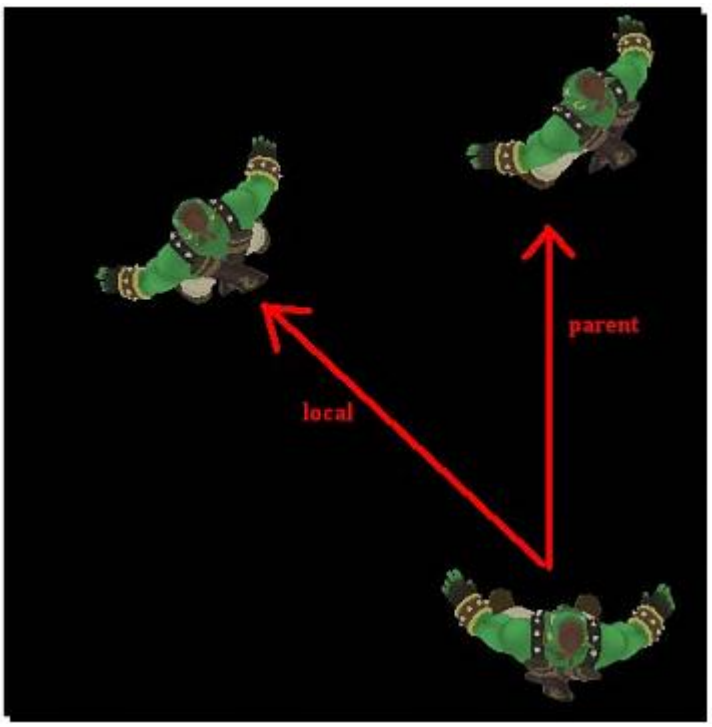


Черная точка в нулевой позиции — это локальная система. Каждая точка куба описывается как перемещенная относительно нулевой точки. Когда сцена показывается, куб должен быть в мировом пространстве, все преобразования сцены применяются для этих точек. Допустим куб прикрепляется к узлу сцены, который будет добавлен в корневой узел со сдвигом (10,0,0). Значит мировое пространство куба будет выглядеть следующим образом:



Разница между двумя кубами в том, что нулевая точка изменила свою позицию, а точнее куб переместился.
Когда мы вызываем translate(), куб перемещается в пространстве, если родитель не определен. При отсутствии родителей куб перемещается так же, как если бы он перемещался в мировом пространстве. Короче говоря, изменения видны на лицо если объект имеет не нулевое вращение. Перемещаясь в локальном пространстве например по оси z, объект будет двигаться вперед относительно своего угла поворота, в пространстве родителя будет двигаться, относительно угла поворота родителя, в мировой системе, будет двигаться просто по оси z.

Перемещение в локальном пространстве.
Мы уже видели, что происходит если производить перемещения в мировом пространстве. Теперь мы переведем пространство в родительское, что бы увидеть разницу.
1. Снова почистим createScene()
2. поставим эталонную модель, поближе к нашей камере.
Ogre::Entity ent = mSceneMgr->createEntity("MyEntity","Sinbad.mesh");*
Ogre::SceneNode node = mSceneMgr->createSceneNode("Node1");*
node->setPosition(0,0,400);
node->yaw(Ogre::Degree(180.0f));
mSceneMgr->getRootSceneNode()->addChild(node);
node->attachObject(ent);
3. Добавим новую модель, повернем на 45 градусов, и переместим на (0,0,20) единиц в родительском пространстве.
Ogre::Entity ent2 = mSceneMgr->createEntity("MyEntity2","Sinbad.mesh");*
Ogre::SceneNode node2 = node->createChildSceneNode("node2");*
node2->yaw(Ogre::Degree(45));
node2->translate(0,0,20);
node2->attachObject(ent2);
4. Добавим еще одну модель, повернем на 45 градусов и переместим на (0,0,20) единиц, только уже в локальном пространстве
Ogre::Entity ent3 = mSceneMgr->createEntity("MyEntity3","Sinbad.mesh");*
Ogre::SceneNode node3 = node->createChildSceneNode("node3");*
node3->yaw(Ogre::Degree(45));
node3->translate(0,0,20,Ogre::Node::TS_LOCAL);
node3->attachObject(ent3);



5. Скомпилируем и запустим:

Что мы сделали

Мы создали эталонную модель, а затем еще две модели, которые были повернуты на 45 градусов вокруг вертикальной оси. Потом мы переместили на (0,0,20) единиц, одну модель в пространстве родителя, другую в локальном пространстве. Модель которая перемещалась в пространстве родителя, и переместился по прямой линии по оси Z. Модель которая перемещалась в локальном пространстве, переместился относительно своего вращения.

Вращения в различных пространствах.

Мы уже научились перемещать объекты в различных пространствах, теперь научимся делать тоже самое только для вращения.

1. Почистим функцию createScene()
2. Добавим код:

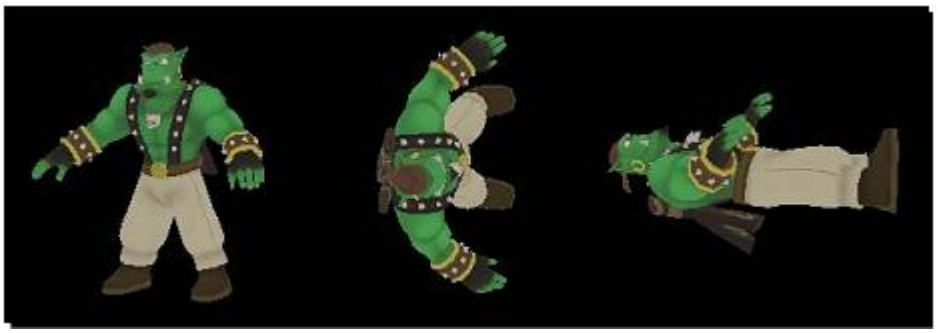
```
Ogre::Entity* ent = mSceneMgr->createEntity("MyEntity","sinbad.mesh");
Ogre::SceneNode* node = mSceneMgr->createSceneNode("Node1");
mSceneMgr->getRootSceneNode()->addChild(node);
node->attachObject(ent);
```

3. Добавим вторую модель и повернем ее как обычно:

```
Ogre::Entity* ent2 = mSceneMgr->createEntity("MyEntity2","sinbad.mesh");
Ogre::SceneNode* node2 = mSceneMgr->getRootSceneNode()->createChildSceneNode("Node2");
node2->setPosition(10,0,0);
node2->yaw(Ogre::Degree(90));
node2->roll(Ogre::Degree(90));
node2->attachObject(ent2);
```

4. Добавим третью модель и используем мировое пространство:

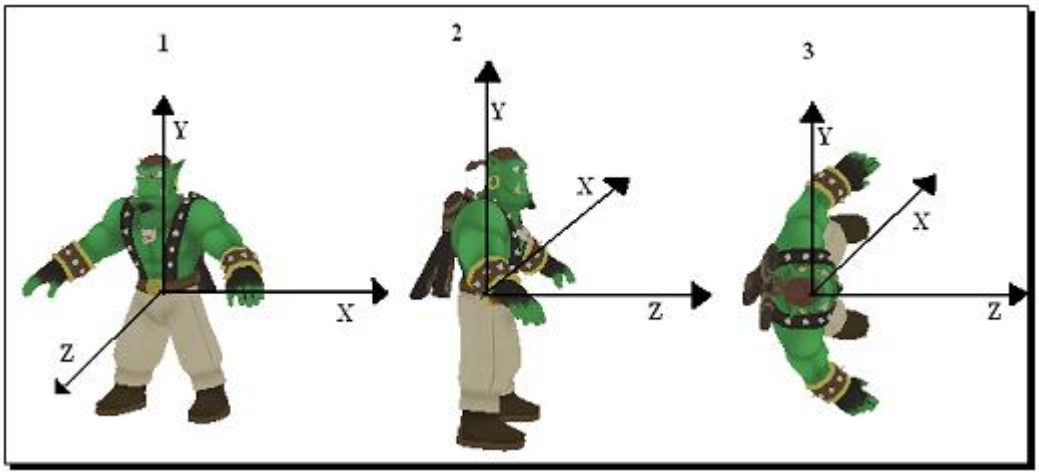
```
Ogre::Entity* ent3 = mSceneMgr->createEntity("MyEntity3","Sinbad.mesh");
Ogre::SceneNode* node3 = node->createChildSceneNode("node3");
node3->setPosition(20,0,0);
node3->yaw(Ogre::Degree(90),Ogre::Node::TS_WORLD);
node3->roll(Ogre::Degree(90),Ogre::Node::TS_WORLD);
node3->attachObject(ent3);
```



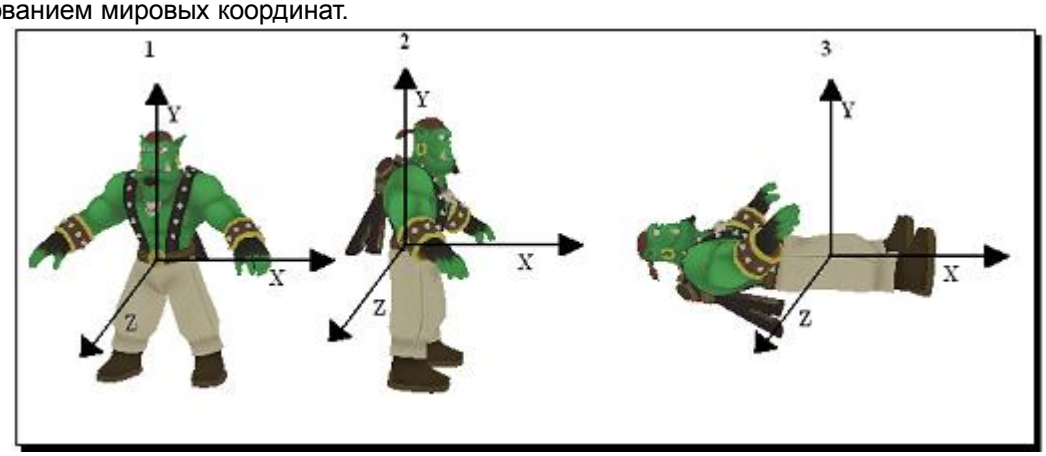
5. Запусим и увидим следующее:

Что мы сделали.

Как всегда мы создали эталонную модель, которая слева на картинке. Повернули вторую модель вокруг оси Y, а затем вокруг оси Z. При вращении по умолчанию используется локальное пространство. Это означает, что после того как мы повернули первую модель на 90 градусов, вокруг оси Y, ориентация оси меняется. Вторая модель использует мировое пространство, и ориентация осей остается неизменной даже если мы повернули узел сцены.



Модель 1 находится в оригинальной системе координат. Модель 2 повернулась по оси Y на 90 градусов, затем повернулась на 90 градусов вокруг оси Z. Теперь посмотрим на те же повороты но с использованием мировых координат.



Здесь мы делаем теже повороты, но изза того что система координат не меняла свою ориентацию, мы получили другой результат.

Масштабирование в различных пространствах.

Масштабирование в различных пространствах не имеет смысла, поэтому это не присутствует в движке.

Подведем итоги.

Мы рассмотрели следующее:

- *Что такое сцена и как с ней работать*
- Различные способы вращения, изменения положения и масштабирование узлов сцены
- Различные виды пространства
- *Различные подходы к созданию сцен*

В следующей главе мы научимся работать со светом, тенями и камерой.

ГЛАВА 3
СВЕТ, КАМЕРА, ТЕНИ

Мы уже научились создавать сложные сцены, но без света и теней.
В этой главе мы узнаем о:

- различных типах источников света и как они работают
- добавим тени к сцене с различными техниками
- узнаем о камерах, и почему они должны быть

Создание плоскости

Прежде чем мы создадим свет и тени, мы должны добавить на нашу сцену плоскость, что бы все это увидеть. В обычном приложении нам не нужна будет плоскость, потому что у нас будет местность, или пол, свет и тени и так будут видны. Свет может рассчитываться и без плоскости, но мы хотим увидеть различные эффекты от света в нашей демонстрационной программе.

Пришло время создать плоскость.

До этого момента мы создавали модели из файлов, на этот раз мы создадим сами из кода.
1. Очистите весь код из createScene()
2. Добавьте следующую строку:

```
Ogre::Plane plane(Vector3::UNIT_Y, -10);
```

3. Теперь создадим плоскость в памяти

```
Ogre::MeshManager::getSingleton().createPlane("plane",
    ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME, plane,
        1500,1500,20,20,true,1,5,5,Vector3::UNIT_Z);
```

4. Получим экземпляр плоскости:

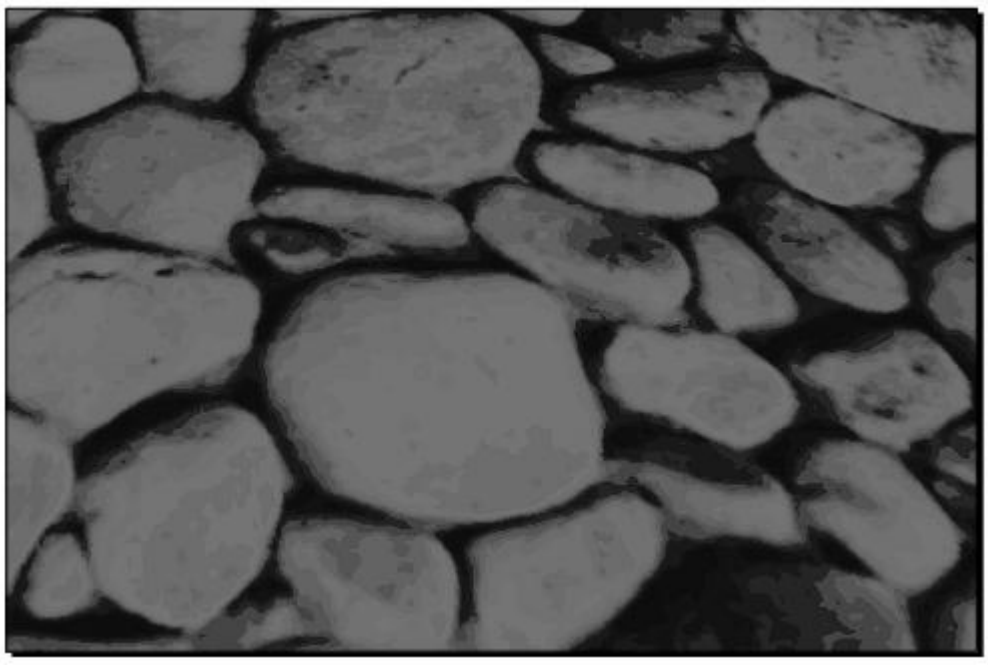
```
Ogre::Entity* ent = mSceneMgr->createEntity("LightPlaneEntity", "plane");
```

5. Присоединим плоскость к нашей сцене:

```
mSceneMgr->getRootSceneNode()->createChildSceneNode()->attachObject(ent);
```

6. Чтобы получить что нибудь кроме белой плоскости, нам нужно установить материал.

```
ent->setMaterialName("Examples/BeachStones");
```

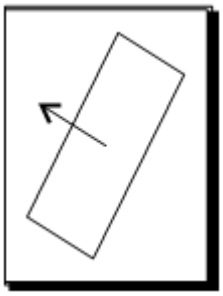


7. Скомпилируем и запустим

Что мы сделали.

Мы создали плоскость и добавили ее к сцене. На втором шаге был создан экземпляр Ogre::Plane. Этот класс описывает плоскости, используя вектор нормали и смещение.
Вектор нормали или проще говоря нормаль, является часто используемой частью трехмерной графики. Вектор нормали строится перпендикулярно поверхности, длина нормали в пределах 0 — 1,

используется для простого расчета света.



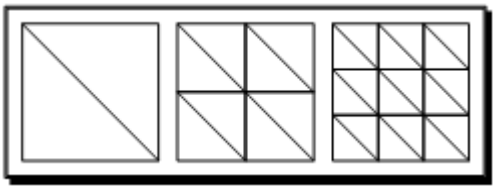
На третьем шаге, мы использовали определение плоскости для создания сетки из нее. Для этого мы использовали MeshManager. Этот менеджер управляет сетками. Помимо управления сетками он может создавать для нас плоскость и многое другое.

```
Ogre::MeshManager::getSingleton().createPlane("plane",
ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME, plane,
        1500,1500,20,20,true,1,5,5,Vector3::UNIT_Z);
```

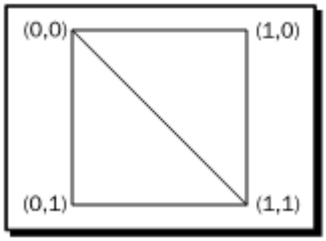
Кроме того мы должны дать плоскости имя. При загрузке сетки из файла имя присваивается как имя ресурса. Кроме того ресурс должен принадлежать группе ресурсов, это похоже на пространство имен в c++, третий параметр это определение плоскости, четвертый и пятый это размеры, шестой и седьмой, это сколько сегментов должна иметь плоскость. Что бы понять что такое сегмент, сделаем небольшое отступление и узнаем о том как модели представлены в 3D пространстве.

Представление 3D модели.

Для представление модели в трехмерном пространстве, она должна быть описана таким образом, что бы компьютер мог понять это наиболее эффективно. Наиболее эффективное представление, это представление в виде треугольников. Наша плоскость может быть представлена в виде двух треугольников, поставленных в форме четырехугольника. Сегмент имеет значения x и y, и мы можем контролировать сколько треугольников создать на плоскости. На следующем рисунке мы видим как плоскость составляется с помощью треугольников с одним, двумя или тремя сегментами на каждую ось. Что бы увидеть это запустите приложение и нажмите клавишу R. Это переключит режим визуализации в каркасный режим, где мы увидим треугольники. Повторное нажатие установит визуализацию в нормальный режим.



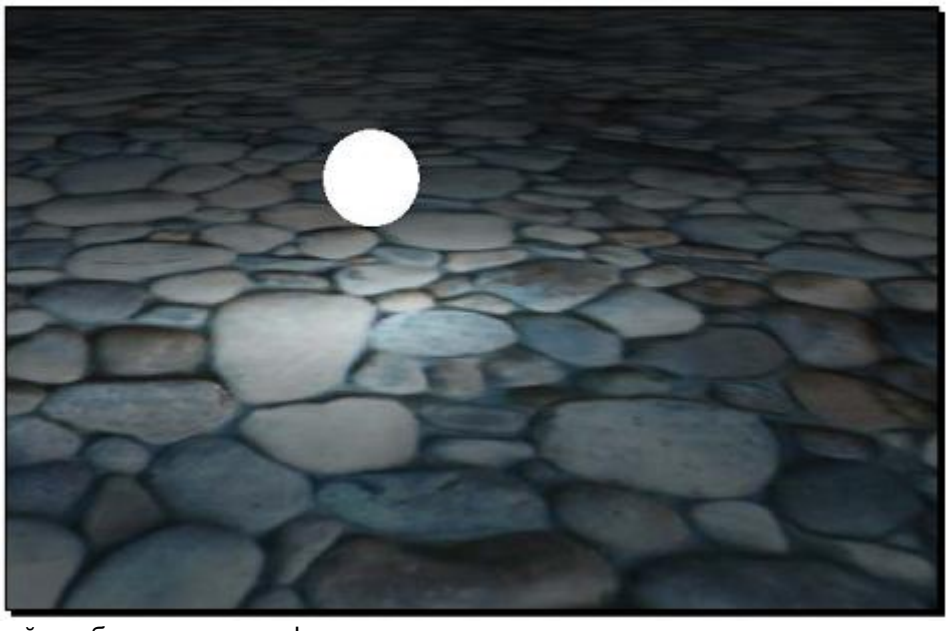
После того как мы определили сколько мы хотим иметь сегментов, мы передаем логический параметр false или true, который означает хотим ли мы посчитать нормали. Как было сказано ранее вектор нормали располагается перпендикулярно поверхности плоскости. Следующие три параметра используются для текстурирования. Координаты текстурирования говорят движку как отобразить текстуру на треугольниках. Так как картинка текстуры является двумерной, координаты текстуры имеют значения x и y. Диапазон значений может принимать от нуля до единицы. (0,0) означает левый верхний угол текстуры, а (1,1) правый нижний. Иногда значения могут быть больше едины, это означает что текстура будет повторяться в зависимости от заданного режима. (2,2) может повторить текстуру 2 раза по обеим осям. Десятый и одиннадцатый параметр, говорит Ogre 3D, как часто мы хотим накладывать текстуру на другую плоскость. Девятый параметр означает, какие координаты текстур мы хотим использовать. Это может быть полезно при наложении более одной текстуры на поверхность. Последний параметр определяет направление «вверх» текстуры. Это также влияет на создание координат текстуры.



В пункте 4 мы создали экземпляр плоскости, которую мы только что создали с помощью MeshManager. Для этого мы использовали имя данное при создании плоскости. На пятом этапе, мы передаем нашу плоскость сцене. На шестом этапе мы установили материал для нашей плоскости. Каждый объект имеет материал наложенный на него. Материал описывает используемые текстуры, как взаимодействует с освещением и многое другое. Мы узнаем обо всем этом в главе о материалах. Если мы не укажем материал, наша плоскость будет просто белой. Так как мы хотим увидеть эффект от освещения, которое мы создадим позже, белый это не лучшее решение для этого. Мы используем материал, которые находятся в каталоге SDK. Этот материал просто добавляет текстуру камня на нашу плоскость.

Добавление точечного источника света.
Теперь когда у нас есть плоскость, мы добавим освещение.

Пришло время создать свет.
Мы создадим точечный источник освещения, и увидим эффект от него:
1. Добавьте следующий код после установки материала для плоскости.
Ogre::SceneNode node = mSceneMgr->createSceneNode("Node1");*
mSceneMgr->getRootSceneNode()->addChild(node);
2. Добавим свет с именем Light1 и скажем Ogre о том что он точечный:
Ogre::Light light1 = mSceneMgr->createLight("Light1");*
light1->setType(Ogre::Light::LT_POINT);
3. Зададим цвет и расположение:
light1->setPosition(0,20,0);
light1->setDiffuseColour(1.0f,1.0f,1.0f);
4. Создадим сферу и поставим ее в позицию где расположен источник света
Ogre::Entity LightEnt = mSceneMgr->createEntity("MyEntity","sphere.mesh");*
Ogre::SceneNode node3 = node->createChildSceneNode("node3");*
node3->setScale(0.1f,0.1f,0.1f);
node3->setPosition(0,20,0);
node3->attachObject(LightEnt);
5. Скомпилируем и запустим приложение. Вы должны увидеть текстуру камня освещенную белым



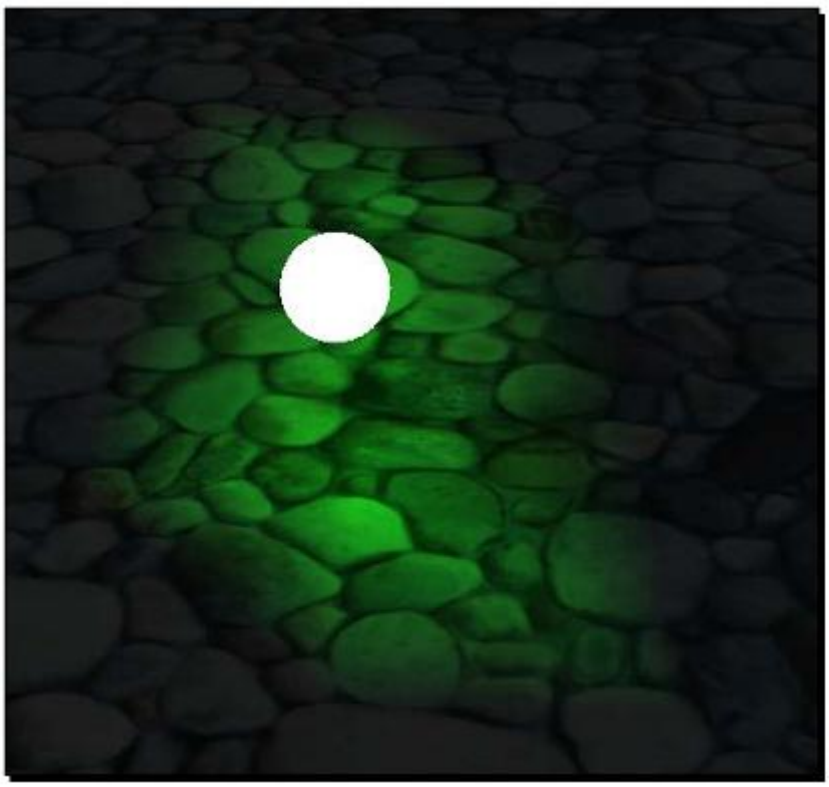
светом, который как бы исходит из сферы.

Что мы сделали.
Мы создали точечный источник света и использовали белую сферу для того что бы отметить где расположен свет.
На первом этапе мы создали узел сцены и добавили его к корневому узлу сцены. Мы создали этот узел для того что бы потом приложить к нему модель сферы. Первая интересная вещь произошла на втором этапе, когда мы создали свет используя менеджер сцены. Каждый объект имеет уникальное

имя и мы дали ему Light1. Если мы не хотим использовать имя, то Ogre3D создаст его за нас. После создания мы сказали Ogre 3D, что хотим что бы наш свет был точечным. Есть три вида источников света, а именно точечный, направленный и прожекторный. Затем мы разместили свет в определенной точке. Позже мы будем создавать другие типы источников света. Точечный источник можно рассматривать как лампочку. Это точка в пространстве которая освещает все вокруг. Потом мы установили источнику света цвет. Каждый цвет описывается как RGB (красный, зеленый, синий). Все параметры имеют диапазон значений от 0 до 1. На 4 этапе мы создали белую сферу в положении света, что бы увидеть где позиционируется свет на сцене.

Добавление прожекторного источника освещения.
Мы создали точечный источник освещения, теперь создадим прожекторный источник, который мы будем использовать.

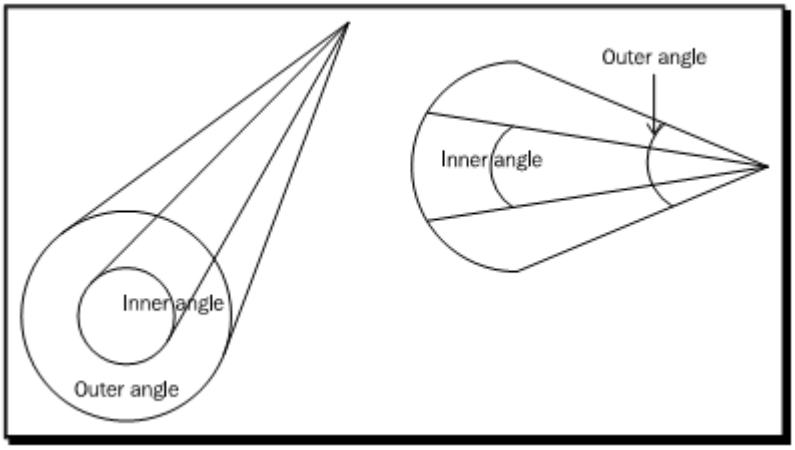
Время создать прожекторный источник освещения.
Мы будем использовать предыдущий код, но немного его изменим, что бы увидеть как работает прожекторный источник освещения.
1. Удалите часть кода, в котором мы создали свет и вставьте в него следующий код для создания нового узла сцены. Будьте внимательны, не удалите код где мы создавали LightEnt.
Ogre::SceneNode node2 = node->createChildSceneNode("node2");*
node2->setPosition(0,100,0);
2. Снова создадим источник освещения, но только теперь с типом LT_SPOTLIGHT:
Ogre::Light light = mSceneMgr->createLight("Light1");*
light->setType(Ogre::Light::LT_SPOTLIGHT);
3. Установим некоторые параметры, мы обсудим их позже:
light->setDirection(Ogre::Vector3(1,-1,0));
light->setSpotlightInnerAngle(Ogre::Degree(5.0f));
light->setSpotlightOuterAngle(Ogre::Degree(45.0f));
light->setSpotlightFalloff(0.0f);
4. Установим цвет и присоединим источник к ранее созданному узлу сцены:
light->setDiffuseColour(Ogre::ColourValue(0.0f,1.0f,0.0f));
node2->attachObject(light);



5. Скомпилируем проект и вы увидите следующее:

Что мы сделали.
Мы создали прожекторный источник освещения и использовали некоторые параметры для света. На первом этапе мы создали новый узел сцены, который использовался позже. На втором этапе мы создали свет, так же как делали ранее, только на этот раз выбрали тип Ogre::LT_SPOTLIGHT, что бы получить прожекторное освещение. На третьем этапе мы задавали различные параметры нашему источнику света.

Теория — прожекторные источники освещения.
Прожекторное освещение имеет эффект фонарика. У него есть позиция, где он расположен и направление в котором он освещает сцену. Направление было первое что мы указали после создания источника освещения. Следующие два параметра это внешние и внутренние углы прожекторного источника освещения. Внутренняя часть прожектора освещает район в полную силу, внешняя часть источника освещения имеет меньшую силу, получается эффект настоящего фонарика. После установки углов, мы установили параметр спада света. Этот параметр описывает на сколько уменьшается интенсивность света при освещении внешней части светового конуса.



В теории мы должны видеть идеальный круг света на плоскости, но мы видим его размыто и деформировано. Причиной этого является то, что освещение ложиться на треугольники, и рассчитывается свет по треугольникам плоскости. Когда мы создавали плоскость, мы указали что хотим иметь 20x20 сегментов. Это довольно низкое разрешение для такой большой плоскости и означает что свет не может быть нормально вычислен. Таким образом, что бы получить более точную картину изменим количество сегментов с 20 до 200.
Ogre::MeshManager::getSingleton().createPlane("plane",ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME, plane,
1500,1500,200,200,true,1,5,5,Vector3::UNIT_Z);

Скомпилируем и запустим:

Теперь гораздо лучше.
Но круг все еще далек от совершенства. Есть несколько техник которые дают гораздо лучшие результаты. Но они достаточно сложны. Но даже не смотря на сложные методы освещения, основы те же, и мы можем изменить программу используя наши источники освещения.
На 4 этапе мы использовали новое описание цвета. Здесь мы использовали для задания цвета класс Ogre::ColourValue(r,g,b)

Направленные источники освещения.
Мы научились создавать прожекторные и точечные источники освещения. У нас остался не рассмотренным еще один тип освещения это направленный свет. Направленный свет — это такой свет который расположен где-то далеко и имеет только цвет и направление, никакого конуса или радиуса как в остальных. Он может рассматриваться как солнечный свет, который идет в одном направлении.

Пришло время создать направленный источник света.
1. Удалите предыдущий код после создания плоскости.
2. Создайте источник освещения, только теперь уже с типом LT_DIRECTIONAL
Ogre::Light light = mSceneMgr->createLight("Light1");*
light->setType(Ogre::Light::LT_DIRECTIONAL);
3. Установите белый цвет и направление света вниз и право:
light->setDiffuseColour(Ogre::ColourValue(1.0f,1.0f,1.0f));
light->setDirection(Ogre::Vector3(1,-1,0));

4. Скомпилируйте и запустите:

Что мы сделали.
Мы создали направленный источник освещения и поставили его светить вправо-вниз с помощью функции setDirection(1,-1,0). В предыдущих примерах у нас получалась полностью не освещенная плоскость, и лишь малая часть была освещена. Здесь мы использовали направленный свет и следовательно вся плоскость освещена. Как было сказано ранее, направленный свет можно использовать как солнце, солнце не имеет радиус, или что нибудь еще, поэтому когда оно светит — оно освещает все.

Отсутствующая вещь.
Мы уже добавили свет, но не кажется ли Вам, что чего-то не хватает.

Пришло время найти, то чего нам не хватает.
Мы используем предыдущий код, что бы узнать чего нам не хватает.
1. После создания освещения, добавьте код для создания Синбада, sinbad.mesh.
Ogre::Entity Sinbad = mSceneMgr->createEntity("Sinbad", "Sinbad.mesh");*
Ogre::SceneNode SinbadNode = node->createChildSceneNode("SinbadNode");*
2. Затем изменим размер модели в три раза и переместим его немного вверх, иначе он будет застрявшим в плоскости.
SinbadNode->setScale(3.0f,3.0f,3.0f);
SinbadNode->setPosition(Ogre::Vector3(0.0f,4.0f,0.0f));
SinbadNode->attachObject(Sinbad);
3. Скомпилируйте и запустите:

Что мы сделали.
Мы добавили экземпляр Синбада на нашу сцену. Наша сцена освещена, но мы не видим, что он не отбрасывает тень, что довольно не реалистично. Следующим шагом будет добавление тени на нашу сцену.

Добавление теней.
Без теней наша сцена выглядит не реалистично.

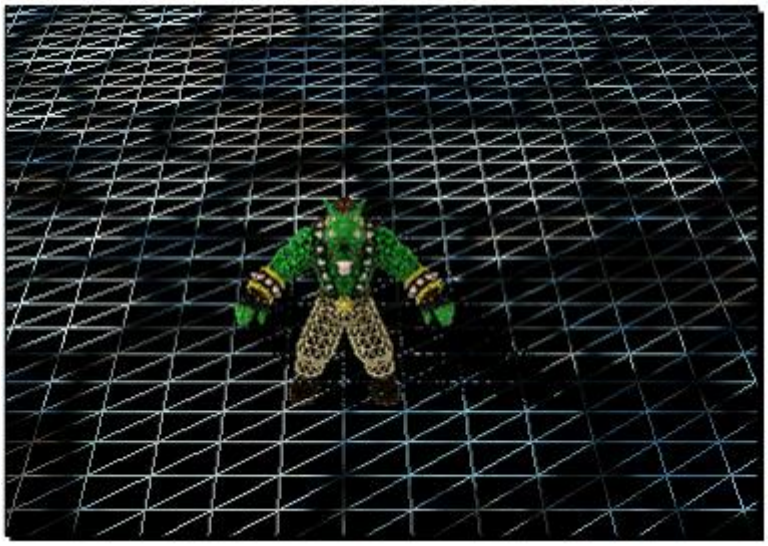
Приступим к добавлению тени.
1. Используем предыдущий код и добавим следующую строку:
mSceneMgr->setShadowTechnique(Ogre:: SHADOWTYPE_STENCIL_ADDITIVE);
2. Скомпилируем и запустим:

Что мы сделали.
С помощью всего лишь одной строки, мы добавили тень в нашу сцену. Ogre 3D делает остальную работу за нас. Ogre 3D поддерживает различные техники теней. Мы использовали трафаретный метод (STENCIL). Трафарет это специальный буфер при отрисовке сцены. Аддитивный (ADDITIVE) подразумевает, что сцена отображается один раз с точки зрения камеры и все источники света складываются при отображении тени. Этот метод достаточно хорош, но более медленный.

Создание камеры.
До сих пор мы использовали камеру, которая была создана за нас в ExampleApplication. Теперь давайте создадим камеру сами. Может быть активная только одна камера, потому что монитор у нас только один. Но использовать в сцене можно несколько камер.

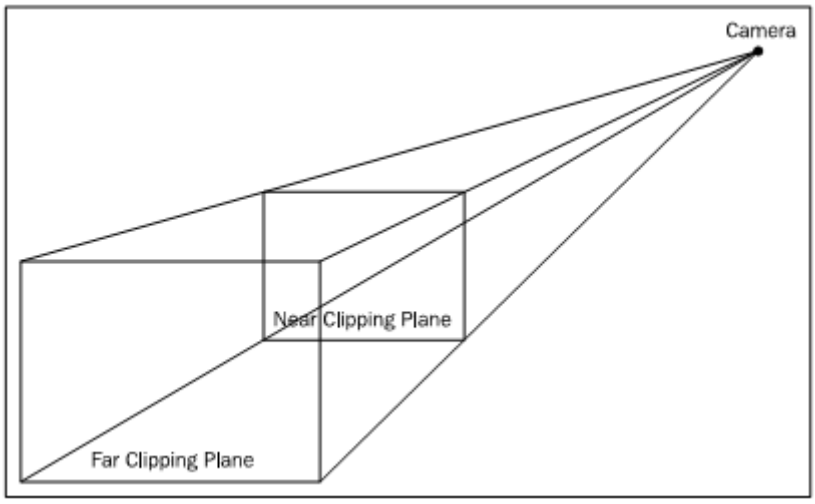
Пришло время создать камеру.
На этот раз мы не будем изменять createScene, оставим все как есть.
1. Создайте новую пустую функцию с именем createCamera в классе ExampleApplication
```
void createCamera() {
}
```
2. Создадим камеру с именем MyCamera1 и присвоем переменной mCamera
mCamera = mSceneMgr->createCamera("MyCamera1");
3. Установим ее в определенной позиции и заставим смотреть внулевую точку:
```
mCamera->setPosition(0,100,200);
mCamera->lookAt(0,0,0);
mCamera->setNearClipDistance(5);
```
4. Изменим режим отображения и установим каркасный:
mCamera->setPolygonMode(Ogre::PM_WIREFRAME);

5. Скомпилируем и запустим:



Что мы сделали.

Мы перегрузили функцию createCamera() которая изначально создавала камеру за нас. После создания мы установили позицию и точку куда она должна смотреть. Далее мы установили дистанцию отсечения. Камера видит только определенную часть сцены, иначе это было бы пустой тратой процессорного времени. При отображении остальные объекты отрезаются из сцены. Этот шаг называется отбор.



Затем мы изменили режим визуализации в каркасный. Этот эффекты мы получали когда нажимали клавишу R. При запуске приложения мы сразу видим разницу, теперь камера находится выше Синбада и смотрит на него.

Одну интересную вещь мы можем наблюдать, что даже после того как мы создали наш собственный экземпляр камеры, мы все еще можем перемещаться по сцене, как раньше. Это потому что мы использовали переменную mCamera из класса ExampleApplication. Особенностью камеры является то, что она также может быть присоединена к узлу сцены.

Создание видового экрана.

Создание экрана связано с концепцией камеры. Таким образом мы сами можем создать видовой экран. Двумерная поверхность экрана это как лист бумаги на который рисуется содержимое сцены, и имеет цвет фона.

Время посмотреть это в действии.

Мы будем использовать код, который использовали ранее, и вновь создадим пустую функцию.

1. Удалим setShadowTechnique() и функция createCamera()
2. Создадим пустую функцию createViewports().

void createViewports() {
}
3. Создадим видовой экран:
Ogre::Viewport vp = mWindow->addViewport(mCamera);*
4. Установим цвет фона и пропорции:
vp->setBackgroundColour(ColourValue(0.0f,0.0f,1.0f));
mCamera->setAspectRatio(Real(vp->getActualWidth()) / Real(vp->getActualHeight()));
5. Скомпилируем и запустим:

Что мы сделали.
Мы создали окно просмотра. Для этого нам нужно было затронуть камеру. Каждое окно может отображать только одну камеру. Конечно параметры камеры могут быть изменены позже с помощью соответствующих функций. Наиболее заметным изменением является изменение цвета фона с черного на синий. Также на 3 этапе мы установили соотношение сторон между шириной и высотой.
Попробуйте поиграться с различными соотношениями сторон и как это влияет на отображение, например вот соотношение сторон где ширина делится на 5.

Подведем итоги.
В этой главе мы научились использовать освещение тени и камеры, а также с плоскостями отсечения. Мы рассмотрели:

- Как освещение может изменить внешний вид нашей сцены.
- Добавили тени к нашей сцене
- Создавали нашу собственную камеру

В следующей главе мы узнаем как пользоваться вводом с клавиатуры и мыши. Мы узнаем что такое FrameListener и как его использовать.

ГЛАВА 4
Использование устройств ввода через FrameListener

До сих пор мы создавали сцены, в которых ничего не происходит, ничего не движется. В этой главе мы исправим это.
- Мы узнаем что такое FrameListener
- Мы узнаем как происходит ввод данных пользователем
- Создадим свой процесс камеры

И так начнем…

Подготовка сцены.

Прежде чем добавлять движущиеся предметы, нам нужно создать сцену, давайте сделаем это.

Пришло время создать сцену.

Мы будем использовать несколько иную версию сцены из предыдущей главы:
1. Удалите код из createCamera() и createScene()
2. Удалите функцию createViewports()
3. Добавьте новый член класса. Это будет указатель на узел сцены:

```
private:
Ogre::SceneNode* _SinbadNode;
```

4. Добавьте плоскость в функции createScene():

```
Ogre::Plane plane(Vector3::UNIT_Y, -10);
Ogre::MeshManager::getSingleton().createPlane("plane",
  ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME, plane,
  1500,1500,200,200,true,1,5,5,Vector3::UNIT_Z);
Ogre::Entity* ent = mSceneMgr->createEntity("LightPlaneEntity",
"plane");
mSceneMgr->getRootSceneNode()->createChildSceneNode()->attachObject(ent);
ent->setMaterialName("Examples/BeachStones");
```

5. Теперь добавим на сцену свет:

```
Ogre::Light* light = mSceneMgr->createLight("Light1");
light->setType(Ogre::Light::LT_DIRECTIONAL);
light->setDirection(Ogre::Vector3(1,-1,0));
```

6. Также мы должны создать экземпляр Синбада и прикрепить его к сцене.

```
Ogre::SceneNode* node = mSceneMgr->createSceneNode("Node1");
mSceneMgr->getRootSceneNode()->addChild(node);
Ogre::Entity* Sinbad = mSceneMgr->createEntity("Sinbad", "Sinbad.
mesh");
_SinbadNode = node->createChildSceneNode("SinbadNode");
_SinbadNode->setScale(3.0f,3.0f,3.0f);
_SinbadNode->setPosition(Ogre::Vector3(0.0f,4.0f,0.0f));
_SinbadNode->attachObject(Sinbad);
```

7. Также активируем тени:

```
mSceneMgr->setShadowTechnique(SHADOWTYPE_STENCIL_ADDITIVE);
```

8. Создадим камиру в положении (0,100,200) и установим взгляд на точку (0,0,0). Добавьте этот код в createCamera();

```
mCamera = mSceneMgr->createCamera("MyCamera1");
mCamera->setPosition(0,100,200);
mCamera->lookAt(0,0,0);
mCamera->setNearClipDistance(5);
```

9. Скомпилируйте и запустите – должен получиться следующий результат:

Что мы сделали.
Мы использовали знания предыдущих глав, что бы создать сцену. Мы уже должны понимать, что мы делали. Если вам все еще не понятно вернитесь к предыдущим главам и попытайтесь понять.

Добавление движений на сцену.
Мы создали сцену, теперь давайте добавим движение.

Пришло время добавить на сцену движение.
До сих пор у нас был только один класс, а именно, ExampleApplication. На этот раз нам нужен другой.
1. Создайте новый класс Example25FrameListener унаследованный от Ogre::FrameListener
```
class Example25FrameListener : public Ogre::FrameListener
{
};
```
2. Создайте член Ogre::SceneNode и назовите его _node.
```
private:
  Ogre::SceneNode* _node;
```
3. Создайте конструктор:
```
public:
  Example25FrameListener(Ogre::SceneNode* node)
  {
    _node = node;
  }
```
4. Добавьте функцию frameStarted(FrameEvent& evt), в которой мы будем двигать узел сцены.
```
bool frameStarted(const Ogre::FrameEvent  &evt)
{
  _node->translate(Ogre::Vector3(0.1,0,0));
  return true;
}
```
5. Добавьте переменную FrameListener, которая создастся потом
```
Ogre::FrameListener* FrameListener;
```

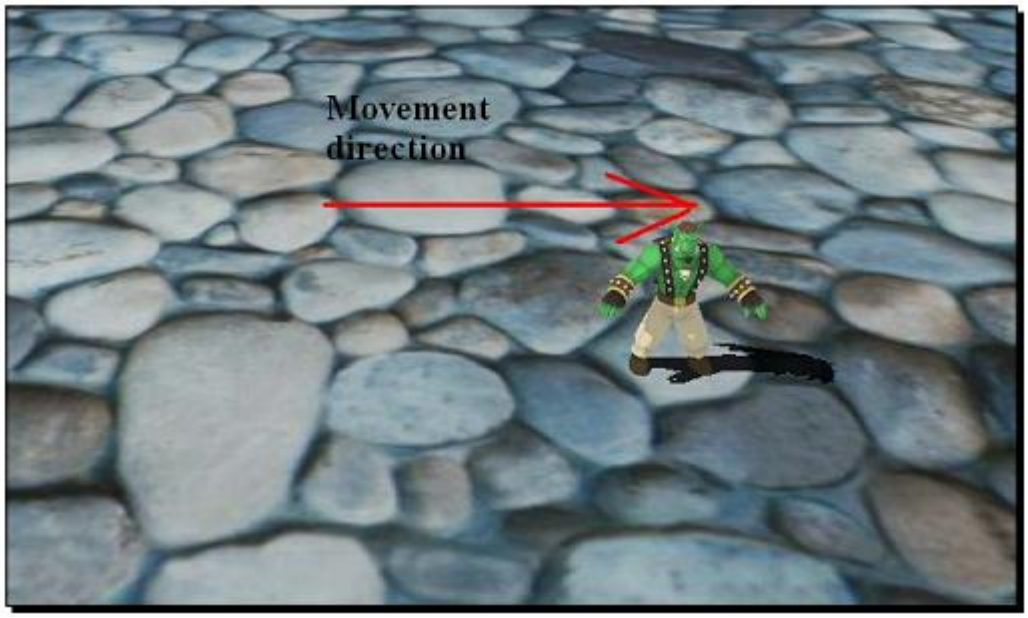
6. В конструкторе ей присвойте NULL

```
Example25()
{
  FrameListener = NULL;
}
~Example25()
{
  if(FrameListener)
  {
  delete FrameListener;
  }
}
```

7. Теперь в функции createFrameListener класса ExampleApplication, создайте FrameListener используя mRoot.

```
void createFrameListener()
{
  FrameListener = new Example25FrameListener(_SinbadNode);
  mRoot->addFrameListener(FrameListener);
}
```

8. Скомпилируйте и запустите приложение. Вы увидите ту же сцену, что и ранее, но на этот раз экземпляр Синбада перемещается вправо, и вы не можете перемещать камеру и закрыть приложение клавишей выхода. Закройте крестиком.



Что мы сделали.

Мы добавили новый класс, который двигает узел сцены.

FrameListener

Мы столкнулись с новым понятием FrameListener. Мы добавляем класс наследуемый от Ogre::FrameListener, и добавляем его с использованием addFrameListener() класса Ogre::Root. Класс получает уведомление когда происходит какое-то событие. В этом примере мы перегрузили frameStarted(). Ogre::Root перебирает все добавленные FrameListener и вызывает у них frameStarted(). В нашем примере мы передвинули узел на 0.1 единицу по оси X. Этот узел был передан FrameListener в конструкторе, поэтому каждый раз когда сцена рисуется, узел немного передвигается, как следствие модель движется.

Как мы заметили во время работы приложения, мы не можем двигать камеру, и закрыть приложение клавишей выхода. Это потому что, передвижение камеры было сделано за нас в классе ExampleApplication. Теперь когда мы заменили на собственную реализацию, мы не можем использовать раннюю из ExampleApplication. Мы будем повторно реализовывать большинство из них в этой главе, так что не стоит беспокоиться. Если необходимо мы все еще можем вызвать функцию базового класса. На шаге 4 наша функция возвращает true. Если вернуть false то приложение будет закрыто.

Изменим код так, что бы время расчитывалось не на основе скорости прорисовки кадров.
В зависимости от компьютера, модели в сцене могут двигаться очень быстро или наоборот очень медленно. Причина в том что за одну единицу времени, разными компютерами может показываться разное количество кадров. Так как у нас за 1 кадр объект передвигается на 0,1 единицу несложно посчитать следующее. Слабый компьютер, который отображает 30 кадров в секунду, за секунду передвинет нашу модель на 0,1 x 30 = 3 единицы, а сильный компьютер, который рисует 200 кадров в секунду, передвинет модель на 0,1 x 200 = 20 единиц. Мы хотим что бы модель двигалась с одинаковой скоростью в секунду на разных машинах. В Ogre это легко реализуемо.

Мы будем использовать предыдущий код и изменим только одну строку кода:
1. Изменение строки, где мы передвигаем модель:
*_node->translate(Ogre::Vector3(10,0,0) * evt.timeSinceLastFrame);*
2. Скомпилируйте и запустите. Вы увидите туже сцену, только теперь модель двигается со скоростью одинаковой на всех компьютерах.

Что мы сделали.
Мы изменили движение модели на основе времени кадра. Мы добились этого путем простого умножения. На шаге 1, мы использовали тот факт, что Ogre 3D при вызове frameStart() содержит время, которое прошло относительно предыдущего кадра:
*Ogre::Vector3(10,0,0) * evt.timeSinceLastFrame);*
Эта строка использует эти данные для расчета того, на сколько мы хотим передвинуть нашу модель. Мы используем вектор и умножаем его на время, прошедшее с последнего кадра в сек. В этом случае мы будем использовать вектор (10,0,0). Это означает, что мы хотим, что бы наша модель перемещалась на 10 единиц за секунду.

Добавление ввода.
Теперь у нас есть движущаяся сцена, но мы хотели бы иметь возможность выхода из приложения, как раньше. Значит нам нужно добавить поддержку ввода, и когда мы нажмем Escape мы должны выйти из приложения. До сих пор мы использовали только Ogre3D, но теперь мы будем также использовать и OIS (объектно-ориентированная система ввода), которая поставляется вместе с Ogre3D SDK. Она используется совместно с FrameListener, но в остальном полностью независима.

Пришло время добавить поддержку ввода.
Опять же мы используем код из предыдущего приложения, и сделаем некоторые дополнения, что бы получить поддержку ввода.
1. Нам нужно добавить новый параметр в конструктор нашего слушателя (FrameListener). Нам нужен указатель на окно визуализации.
Example27FrameListener(Ogre::SceneNode node,RenderWindow* win)*
2. После изменения конструктора, изменим строку инициализации
Ogre::FrameListener FrameListener = new Example27FrameListener(_SinbadNode,mWindow);*
3. После этого нам нужно добавить код в конструктор слушателя, для начала нужны две вспомогательные переменные.
size_t windowHnd = 0;
std::stringstream windowHndStr;
4. Теперь попросим у Ogre3D дескриптор окна
win->getCustomAttribute("WINDOW", &windowHnd);
5. Переведем дескриптор в строку.
windowHndStr << windowHnd;
6. Создадим список параметров для OIS, и добавим к нему дескриптор окна.
OIS::ParamList pl;
pl.insert(std::make_pair(std::string("WINDOW"), windowHndStr.str()));
7. Создадим ввод, используя список параметров
_man = OIS::InputManager::createInputSystem(pl);
8. Затем создадим клавиатуру
_key = static_cast<OIS::Keyboard>(_man->createInputObject(OIS::OISKeyboard, false));*

9. То что мы делали, нужно корректно уничтожать. Добавьте деструктор для FrameListener, который будет уничтожать объекты OIS.
```
~Example27FrameListener()
{
  _man->destroyInputObject(_key);
  OIS::InputManager::destroyInputSystem(_man);
}
```
10. После того как мы закончили инициализацию, добавьте следующий код в frameStarted(), до return
```
_key->capture();
if(_key->isKeyDown(OIS::KC_ESCAPE))
{
  return false;
}
```
11. Добавьте новые переменные в FrameListener.
```
OIS::InputManager* _man;
OIS::Keyboard* _key;
```
12. Скомпилируйте и запустите приложение. По кнопке Escape должен происходить выход из приложения.

Что мы сделали.
Мы создали экземпляр системы ввода и использовали ее, что бы перехватить нажатия клавиш от пользователя. Из-за того что нам понадобился дескриптор окна для системы ввода, мы изменили конструктор FrameListener, где передаем окно визуализации. Затем мы преобразовали дескриптор из числа в строку и добавили эту строку в список параметров OIS. С помощью этого списка, мы создали наш экземпляр системы ввода.

Дескриптор окна.
Дескриптор окна это просто число, которое используется в качестве идентификатора определенного окна. Число создается операционной системой и имеет уникальное значение для каждого окна. Системе ввода необходим этот дескриптор, потому что без него она не сможет получать события ввода. Ogre3D создает окно за нас. Таким образом, для того чтобы получить дескриптор окна мы должны выполнить функцию:
```
win->getCustomAttribute("WINDOW", &windowHnd);
```
Есть несколько атрибутов, которые определяют окно, Ogre3D реализует общие функции чтения. Это также является независимым от платформы. WINDOW это ключевое слово дескриптора окна. Мы должны передать функции указатель куда будет записано значение. После получения дескриптора мы превращаем его в строку используя stringstream, потому что OIS ожидает строку. В процессе создания, мы даем OIS список пар параметров, состоящих из строки идентификатора и значения строки. На 6 этапе, мы создали этот список параметров и добавили строку с дескриптором окна. На 7 шаге мы использовали этот список параметров для создания системы ввода. На 8 шаге мы создаем интерфейс клавиатуры. Этот интерфейс будет использоваться для получения нажатых клавиш. Каждый раз прежде чем сделать кадр мы фиксируем новое состояние клавиатуры с помощью функции capture(). Если мы не будем вызывать эту функцию, мы не получим новое состояние клавиатуры, и поэтому мы не сможем получать события клавиатуры. Потом мы запрашиваем у клавиатуры, нажали ли мы клавишу выхода или нет. Если мы хотим выйти из приложения, нам нужно вернуть в функции отрисовки кадра false, чтобы Ogre3D знал, что мы хотим закрыть приложение. Иначе если вернуть true, приложение продолжит свою работу.

Добавление движения модели.
Теперь у нас есть возможность получить пользовательский ввод с клавиатуры. Давайте используем его для движения Синбада на плоскости.

Мы используем предыдущий код и немного его модифицируем. Мы создадим WASD управление для Синбада.
1. Замените строку, в которой мы перемещали Синбада в FrameListener:
```
Ogre::Vector3 translate(0,0,0);
```
2. Добавьте следующий запрос, после запроса выхода.
```
if(_key->isKeyDown(OIS::KC_W))
{
  translate += Ogre::Vector3(0,0,-10);
}
```

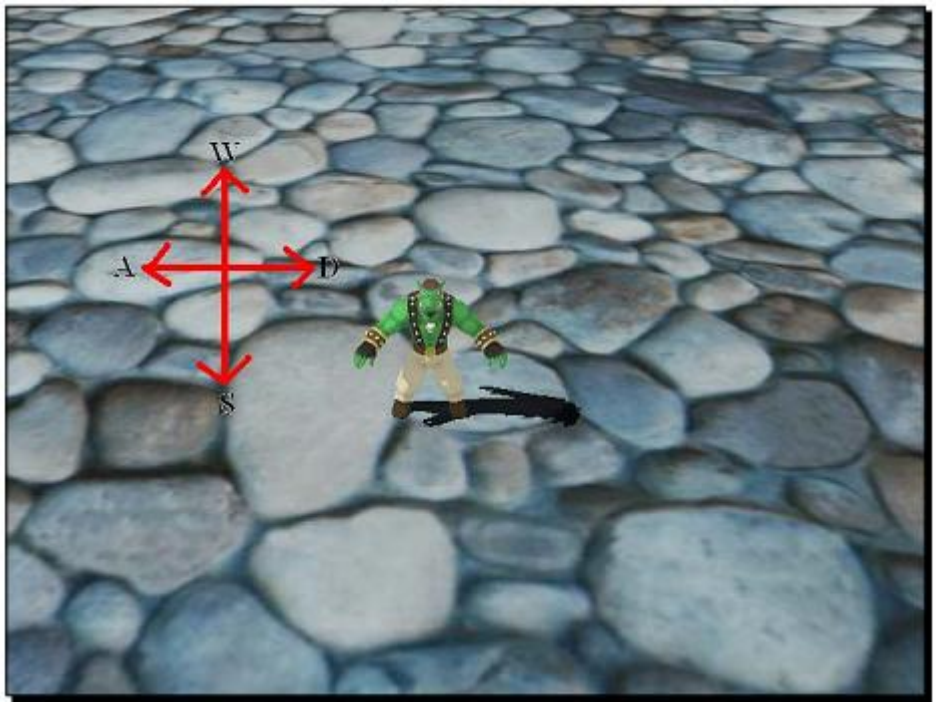
3. Теперь добавьте код для трех других клавиш. Это в основном тоже самое, только код клавиши и направление будут изменяться.

```
if(_key->isKeyDown(OIS::KC_S))
{
  translate += Ogre::Vector3(0,0,10);
}
if(_key->isKeyDown(OIS::KC_A))
{
  translate += Ogre::Vector3(-10,0,0);
}
if(_key->isKeyDown(OIS::KC_D))
{
  translate += Ogre::Vector3(10,0,0);
}
```

4. Теперь с помощью вектора перемещения (translate), передвинем модель, имейте ввиду, что нам нужно это сделать на основе времени предыдущего кадра

```
_node->translate(translate*evt.timeSinceLastFrame);
```

5. Скомпилируйте и запустите приложение, вы сможете управлять Синбадом клавишами WASD.



Что мы сделали.

Мы добавили перемещение с помощью кнопок WASD. Мы запросили состояние всех четырех клавиш и изменяли вектор translate. Затем мы применили этот вектор для модели с использованием времени.

Добавление камеры.

У нас есть выход и мы можем двигать Синбада. Пришло время заставить нашу камеру снова работать.

Мы уже создали нашу камеру, а теперь мы собираемся использовать ее в сочитании с пользовательским вводом.

1. Расширьте конструктор FrameListener таким образом, что бы он принимал указатель на камеру

```
Example30FrameListener(Ogre::SceneNode* node,RenderWindow* win,Ogre::Camera* cam)
```

2. Теперь добавим переменную для хранения указателя камеры

```
Ogre::Camera* _Cam;
```

3. В конструкторе назначим указатель

```
_Cam = cam;
```

4. Изменим инициализацию FrameListener и передадим указатель на камеру

```
Ogre::FrameListener* FrameListener = new Example30FrameListener(_SinbadNode,mWindow,mCamera);
```

5. Для перемещения камеры мы должны получить мышь. И так создайте новую переменную для хранения мыши.
OIS::Mouse _mouse;*
6. В конструкторе инициализируем мышь после клавиатуры
_mouse = static_cast<OIS::Mouse>(_man->createInputObject(OIS::OISMouse, false));*
7. Теперь мы должны фиксировать состояние мыши, как мы делали для клавиатуры.
_mouse->capture();
8. Замените строку перемещения модели на следующую
*_node->translate(translate*evt.timeSinceLastFrame * _movementspeed);*
9. После обработки состояния клавиатуры в frameStarted(), добавьте следующий код обработки состояния мыши
*float rotX = _mouse->getMouseState().X.rel * evt.timeSinceLastFrame* -1;*
*float rotY = _mouse->getMouseState().Y.rel * evt.timeSinceLastFrame * -1;*
10. Применим перемещение и поворот для камеры
_Cam->yaw(Ogre::Radian(rotX));
_Cam->pitch(Ogre::Radian(rotY));
*_Cam->moveRelative(translate*evt.timeSinceLastFrame * _movementspeed);*
11. При уничтожение FrameListener нужно уничтожать и мышь
_man->destroyInputObject(_mouse);
12. Скомпилируйте и запустите приложение. Вы сможете перемещаться по сцене как и раньше.

Что мы сделали.
Мы использовали созданную нами камеру в сочетании с пользовательским вводом. Для того чтобы манипулировать камерой мы должны были передать ее нашему FrameListener. Это было сделано на шагах 1 и 2 с использованием конструктора. Для управления нашей камерой мы хотели использовать мышь. Итак, сначала мы создали интерфейс мыши, это было на шаге 6. На 7 шаге мы обновили состояние мыши capture().

Состояние мыши.
Запрос состояния клавиатуры был сделан с помощью функции isKeyDown(). Что бы получить состояние мыши мы использовали getMouseState(). Эта функция возвращает структуру с состоянием мыши, как экземпляр класса MouseState, который содержит в себе информацию о состоянии кнопок нажаты или нет, и перемещение мыши, относительно последнего захвата состояния. Нам нужна эта информация, для того что бы посчитать, на сколько нам нужно повернуть камеру. Мышь движеться по двум осям – это X и Y. Обе оси сохраняются в отдельные переменные в состоянии мыши. Тогда мы имеем возможность получить относительные или абсолютные значения положения мыши. Так как мы заинтересованы только в движении мыши, а не в позиции, мы используем относительные значения. Абсолютные значения содержат положение мыши на экране. Это необходимо для того что бы определить например нажата ли кнопка в определенной зоне на экране. Для вращения камеры нужно только движение мыши, поэтому мы используем относительные значения. Относительное значение указывает только направление и скорость мыши, но не количество пикселей.

Потом эти значения умножаются на время прошедшее с предыдущего кадра и на -1. -1 мы используем для получения противоположного направления. После вычисления повороты мы применяем эти значения к камере с помощью функций yaw() и pitch(). После этого мы применяем камере вектор перемещения, который получился на основе ввода с клавиатуры. Для этого мы использовали функцию moveRelative() у камеры. Эта функция переводит камеру в локальном пространстве без учета вращения камеры. Это полезно, потому что мы знаем, что в локальном пространстве (0,0,-1) переместит камеру вперед. Если к камере применены повороты, то это будет не верно. Обратитесь к главе о различных пространствах в 3D пространстве для более подробного объяснения.

Добавление каркасного и точечного отображения.
В предыдущей главе мы пользовались клавишей R, который изменял режим отображения в каркасный или точечный режим. Мы хотим добавить эту возможность для нашего собственного FrameListener.

Мы используем код, который только что создали, и как всегда просто добавим в него новые возможности.
1. Нам нужна вспомогательная переменная для хранения текущего режима отображения.
Ogre::PolygonMode _PolyMode;
2. Инициализируем ее как PM_SOLID
_PolyMode = Ogre::PolygonMode::PM_SOLID;
3. Затем мы добавим новое условие в frameStarted(), которая проверяет на нажатие клавиши R. Если нажата, то мы изменяем режим визуализации, если текущий режим обычный, то мы изменим на каркасный.

```
if(_key->isKeyDown(OIS::KC_R))
{
  if(_PolyMode == PM_SOLID)
  {
    _PolyMode = Ogre::PolygonMode::PM_WIREFRAME;
  }
```

4. Если режим каркасный, то изменим его на точечный

```
else if(_PolyMode == PM_WIREFRAME)
  {
    _PolyMode = Ogre::PolygonMode::PM_POINTS;
  }
```

5. Если точечный, то обратно в обычный режим

```
else if(_PolyMode == PM_POINTS)
  {
    _PolyMode = Ogre::PolygonMode::PM_SOLID;
  }
```

6. Теперь применим новые параметры

```
_Cam->setPolygonMode(_PolyMode);
}
```

7. Скомпилируйте и запустите приложение, у вас должна появиться возможность переключения режима визуализации.

Что мы сделали.
Мы использовали функцию setPolygonMode() для изменения режима визуализации из твердого состояния в каркасное и в точечное. Мы всегда сохраняем последнее состояние, что бы знать на какое состояние переключить в следующий раз. Мы заметили одну вещь, что режим изменяется довольно быстро, пока нажата клавиша R. Это потому что мы проверяем клавишу R в каждом кадре. И у нас получается нажимать клавишу дольше чем за 1 кадр. В результате приложение думает, что мы нажали клавишу R несколько раз. Это не является хорошим способом.

Добавление таймера.
Решение проблемы это использование таймера. Каждый раз когда вы нажмете клавишу R, будет запускаться таймер, и только после того как пройдет достаточно времени, мы будем снова обрабатывать клавишу R.

1. Добавьте переменную таймера в FrameListener.
Ogre::Timer _timer;
2. Сбросим его в конструкторе
_timer.reset();
3. Теперь добавим проверку, если таймер работает более чем 0,25 секунд с момента последнего нажатия R, то мы будем обрабатывать R.
if(_key->isKeyDown(OIS::KC_R) && _timer.getMilliseconds() > 250)
{
4. Если прошло достаточно времени, мы должны сбросить таймер.
_timer.reset();
5. Скомпилируйте и запустите приложение. Теперь R работает как надо.

Что мы сделали.
Мы использовали еще один новый класс Ogre3D, а именно Ogre::Timer. Этот класс предоставляет нам функциональность таймера. Мы сбрасываем таймер в конструкторе нашего FrameListener и каждый раз, когда пользователь нажмет R, мы проверяем, прошло ли 0,25 секунд, с момента последнего сброса. Если это так, то мы сбрасываем таймер и изменяем режим визуализации. Это гарантирует, что режим отображения будет меняться раз в 0,25 секунд.

Подведем итоги.
В этой главе мы узнали что такое FrameListener и как его использовать. Мы так же рассмотрели основы OIS и как работать с клавиатурой и мышью.
В частности мы рассмотрели:

- Как получить уведомление, когда рисуется новый кадр
- Важные различия между обычным движением и движением на основе времени
- Как реализовать наши собственные движения камеры
- Как изменить режим визуализации

Теперь когда мы научились основным функциям нашего FrameListener, мы собираемся анимировать модель в следующей главе.

Глава 5
Анимация моделей в Ogre3D

В этой главе внимание будет уделено тому как работает анимация в целом и в Ogre3D. Без анимации трехмерная сцена выглядит безжизненно. Это одна из наиболее важных вещей для того, что бы сделать сцену реалистичной и интересной.

В этой главе мы будем:
- воспроизводить анимацию
- комбинировать две анимации
- присоединять к анимации сущность (entity)

Итак давайте начнем.

Добавление анимации.

В предыдущей главе мы добавили интерактивность с помощью пользовательского ввода. Теперь мы добавим другую форму интерактивности к нашей сцене с помощью анимации. Анимация является очень значимой вещью в трехмерных сценах. Без них сцены выглядят безжизненными, но с анимацией сцена оживает. Поэтому давайте добавим анимацию.

Как всегда мы будем использовать код из предыдущей главы, и на этот раз ничего удалять из него не нужно.

1. Для начала добавим новые переменные-члены в FrameListener. Добавим указатель на сущность которую хотим оживить и указатель на используемое состояние анимации.

Ogre::Entity _ent;*
Ogre::AnimationState _aniState;*

2. Затем изменим конструктор FrameListener, что бы получить объект как указатель.

Example34FrameListener(Ogre::SceneNode node,Ogre::Entity* ent,RenderWindow* win,Ogre::Camera* cam)*

3. В теле конструктора назначим указатель на сущность

_ent = ent;

4. После этого, получим состояние анимации, называемое Dance (танец), из объекта и сохраним его в переменную-член, мы создали ее именно для этой цели. Наконец включим анимацию и зациклим ее.

_aniState = _ent->getAnimationState("Dance");
_aniState->setEnabled(true);
_aniState->setLoop(true);

5. Далее мы должны сказать анимации, сколько времени прошло с последнего обновления. Мы будем делать это в frameStarted(), там мы будем узнавать сколько времени прошло.

_aniState->addTime(evt.timeSinceLastFrame);

6. Последнее что нам нужно сделать, это скорректировать ExampleApplication, для работы с новым FrameListener. Добавить новую переменную.

Ogre::Entity _SinbadEnt;*

7. Вместо создания сущности для локального указателя, сохраним его с помощью переменной-члена заменим

Ogre::Entity Sinbad = mSceneMgr->createEntity("Sinbad", "Sinbad.mesh");*

на

_SinbadEnt = mSceneMgr->createEntity("Sinbad", "Sinbad.mesh");

8. Тоже самое необходимо сделать при установки модели узлу.

_SinbadNode->attachObject(_SinbadEnt);

9. И, конечно, при создании FrameListener, добавьте новый параметр.

Ogre::FrameListener FrameListener = new*
Example34FrameListener(_SinbadNode,_SinbadEnt,mWindow,mCamera);

10. Скомпилируйте и запустите приложение. Вы увидите как Синбад танцует.

Что мы сделали.
С помощью нескольких строк кода, мы сделали так, что бы Синбад танцевал. На шаге 1, мы добавили две новые переменные, которые понадобятся нам дальше для анимации модели. Первая переменная, просто указатель на объект, который мы хотим анимировать. Второй указатель Ogre::AnimationState, который используется в Ogre3D для представления связанной с анимацией информации. Шаги 2 и 3 являются простыми, мы изменили конструктор для размещения нового указателя и в шаге 3 мы сохранили указатель в переменную, которую создали на шаге 1. На 4 шаге мы попросили модель вернуть нам анимацию танца. Каждый объект хранит в себе все имеющиеся у него анимации и мы можем запросить их с помощью функции getAnimatiobState(). Эта функция возвращает указатель вида AnimationState или если анимации не существует, возвращает пустой, указатель. После того как мы получили анимацию модели, мы ее включили. Затем мы сказали, что хотим, что бы анимация играла снова и снова, пока мы ее не остановим. Каждый раз когда кадр рисуется нужно передавать время для анимации, чтобы анимация могла обновляться. Мы можем например анимировать модель в замедленном режиме, передавав время как деленное на два.

Проигрывание двух анимаций одновременно.
Мы будем использовать тот же код, который мы использовали для создания первого примера.
1. Поменяем анимацию танца на бег (Runbase)
_aniState = _ent->getAnimationState("RunBase");

2. Скомпилируем и запустим. Вы увидите, что анимация работает, но только для нижней половины модели.

3. Для нашей второй анимации, нам нужен новый указатель для анимации состояния.
Ogre::AnimationState _aniStateTop;*
4. Потом конечно нам нужно получить анимацию для этого состояния, включить и зациклить ее. Анимацию мы будем получать под названием RunTop.
_aniStateTop = _ent->getAnimationState("RunTop");
_aniStateTop->setEnabled(true);
_aniStateTop->setLoop(true);
5. Последнее, что нужно сделать, это добавить к анимации время, как это мы делали для первой анимации
_aniStateTop->addTime(evt.timeSinceLastFrame);
6. Затем снова запустим приложение. Теперь вы увидите как Синбад бежит всем телом.

Что мы сделали.
Мы проиграли две анимации одновременно. С помощью функции playAnimation(AnimationName) невозможно проигрывание двух анимации.
Играть анимаций мы можем столько, сколько захотим. Мы можем даже играть различные анимации с разной скоростью.

Давайте немного пройдемся.
Теперь у нас есть анимация ходьбы, но наша модель не меняет позиции. Теперь мы добавим основные элементы управления движения нашей модели и смешаем с анимацией, используем все что мы знаем.
Как всегда используем предыдущий код.
1. Нам нужны две новые переменные для управления скоростью движения и сохранение вращения.
```
float _WalkingSpeed;
float _rotation;
```
2. В конструкторе мы зададим скорость в 50 единиц в секунду и не иметь вращения по умолчанию.
```
_WalkingSpeed = 50.0f;
_rotation = 0.0f;
```
3. Получим анимации бега и отключим циклическое воспроизведение.
```
_aniState = _ent->getAnimationState("RunBase");
_aniState->setLoop(false);
_aniStateTop = _ent->getAnimationState("RunTop");
_aniStateTop->setLoop(false);
```
4. В frameStarted() нам потребуются две новые локальные переменные, что бы указать, что модель движется или нет и направление движения.
```
bool walked = false;
Ogre::Vector3 SinbadTranslate(0,0,0);
```
5. Кроме того, а frameStarted() мы добавим код для управления движением модели. Мы будем использовать клавиши со стрелками для передвижения. Когда кнопка нажата мы должны указать что наша модель движется в этом кадре, и в каком направлении, и нужно установить вращение таким образом, что бы модель двигалась в направлении куда смотрит.
```
if(_key->isKeyDown(OIS::KC_UP))
{
  SinbadTranslate += Ogre::Vector3(0,0,-1);
  _rotation = 3.14f;
  walked = true;
}
```
6. Также для остальных клавиш
```
if(_key->isKeyDown(OIS::KC_DOWN))
{
  SinbadTranslate += Ogre::Vector3(0,0,1);
  _rotation = 0.0f;
  walked = true;
}
if(_key->isKeyDown(OIS::KC_LEFT))
{
  SinbadTranslate += Ogre::Vector3(-1,0,0);
  _rotation = -1.57f;
  walked = true;
}
if(_key->isKeyDown(OIS::KC_RIGHT))
{
  SinbadTranslate += Ogre::Vector3(1,0,0);
  _rotation = 1.57f;
  walked = true;
}
```

7. Затем после обработки кнопок, нам нужно проверить, нужно ходить нам в этом кадре или нет. Если мы ходим, то нужно проверить кончилась ли анимация. Если да, то мы перезапускаем анимацию.

```
if(walked)
{
  _aniState->setEnabled(true);
  _aniStateTop->setEnabled(true);
  if(_aniState->hasEnded())
  {
    _aniState->setTimePosition(0.0f);
  }
  if(_aniStateTop->hasEnded())
  {
    _aniStateTop->setTimePosition(0.0f);
  }
}
```

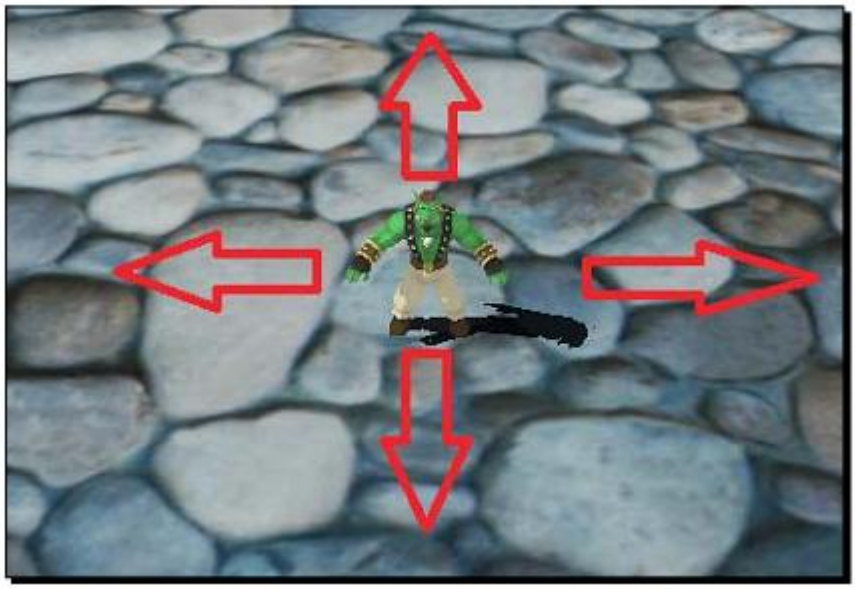
8. Если мы не будем ходить в этом кадре, то нужно установить анимацию равной нулю. Иначе наша модель будет заморожена на пол пути, если мы отпустим кнопку движения. Так что ели мы не ходим мы устанавливаем наши обе анимации в исходное состояние. Кроме того мы отключаем систему анимации, потому что в этом кадре мы в ней не нуждаемся.

```
else
{
  _aniState->setTimePosition(0.0f);
  _aniState->setEnabled(false);
  _aniStateTop->setTimePosition(0.0f);
  _aniStateTop->setEnabled(false);
}
```

9. Последнее что нам нужно сделать это применить перемещение и поворот к нашей модели.

```
_node->translate(SinbadTranslate * evt.timeSinceLastFrame * _WalkingSpeed);
_node->resetOrientation();
_node->yaw(Ogre::Radian(_rotation));
```

10. Скомпилируем и запустим приложение.



Что мы сделали.

Мы создали первое приложение, в котором мы анимируем движение персонажа вместе с его перемещением. Можно сказать, что мы создали первое интерактивное приложение в реальном времени. В пунктих 1 и 2 были созданы и проинициализированы некоторые переменные, которые понадобятся в дальнейшем. На шаге 3 мы изменили алгоритм анимации персонажа, ранее у нас анимация воспроизводилась постоянно, теперь мы не хотим, что бы она играла постоянно, а только когда персонаж идет. Большинство изменений производится в методе frameStarted(). В пункте 4 мы создали некоторые переменные которые позже нам понадобятся, а именно, логическое значение, определяющее движется ли модель в этом кадре, и вектор передвижения модели. На 5 и 6 шагах мы захватываем состояние клавиатуры и в соответствии с направлением движения мы устанавливаем

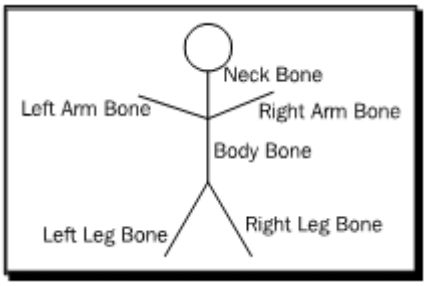
вектор перемещения и устанавливаем логическую переменную, которая отвечает за передвижение. Мы использовали эту переменную на 7 шаге, если она true, то наша модель перемещается, включается анимация и проверяется на окончание анимации. Если анимация кончается, то начать ее заново, для того чтобы она могла играть снова и снова. Так как мы не хотим играть анимацию если персонаж не движется, то мы устанавливаем анимацию в исходное положение и на 8 шаге отключаем. На 9 шаге мы перемещаем модель. Затем мы сбрасываем ориентацию и применяем новый поворот. Сброс нужен для того, чтобы повернуть объект относительно нулевого поворота, т.к. yaw задает не абсолютный поворот, а только лишь добавляет новый градус к предыдущему.

Добавим мечи.
Теперь у нас есть ходьба и анимированая модель, которая может контролироваться пользователем. Теперь мы научимся добавлять объекты к нашей анимированой модели.
Как всегда мы использовали код из предыдущего приложения.
1. В конце функции createScene() создайте 2 экземпляра меча и назовите их Sword1 и Sword2.
Ogre::Entity sword1 = mSceneMgr->createEntity("Sword1", "Sword.mesh");*
Ogre::Entity sword2 = mSceneMgr->createEntity("Sword2", "Sword.mesh");*
2. Теперь добавим мечи к нашей модели используя имена костей.
_SinbadEnt->attachObjectToBone("Handle.L", sword1);
_SinbadEnt->attachObjectToBone("Handle.R", sword2);
3. Скомпилируйте и запустите приложение и вы увидите, что Синбад держит 2 меча.

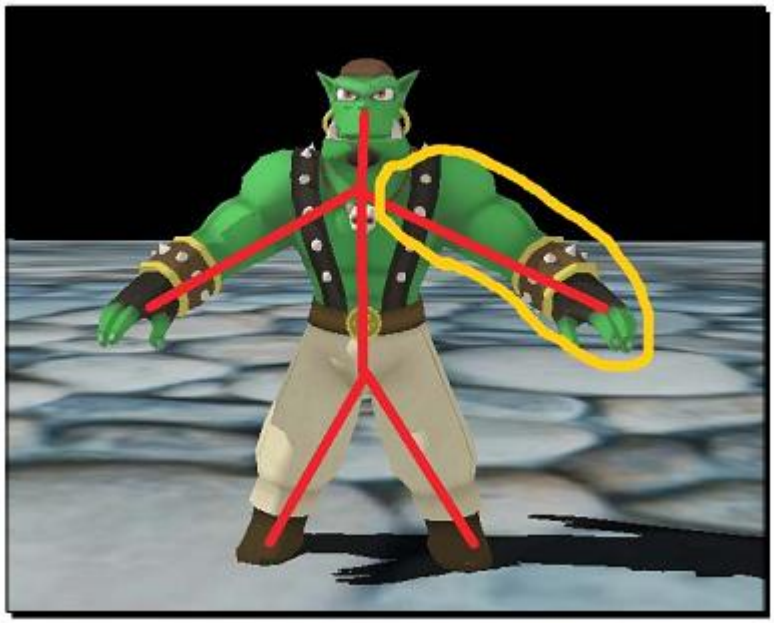
Что мы сделали.
Мы создали два экземпляра мечей и прикрепили их к костям модели. Самая интересная часть это функция под названием attachObjectToBone(). Что бы понять, что эта функция делает, нужно понять как вообще сохраняется и работает анимация.

Анимация.
Для анимации мы используем так называемые скелеты и кости. Эта системы была позаимствована от природы, в природе почти везде используется скелет. С помощью этого скелета, животные и люди могут двигать части тела в определенных направлениях. Например мы можем сделать в кулаке пальцы. Художник создает модель по скелету, что бы потом анимировать. Скелет состоит из костей и суставов.

Вот упрощенный рисунок:



Каждая кость влияет на изменение только одной части модели, например кость на левой руке изменяет левую руку на модели, все остальные детали не будут изменятся.



Сустав, кости могут иметь эффект радиуса, художник создал сложные анимации для Синбада, которые мы используем. Кости имеют имена, так же как и анимации. Ogre 3D позволяет использовать эти кости, что бы приложить к ним другие объекты. Это большое приемущество, так как прикрепленная модель начинает двигаться вместе с костью. Поэтому при анимации модели, мечи всегда будут находится в руках Синбада. Если бы этой функции небыло, то было бы невозможно анимировать мечи вместе с Синбадом.

Как узнать все названия анимаций модели.

Мы уже знаем, что названия анимациям задает художник, сейчас мы узнаем как узнать названия всех анимаций модели в консоле.

Мы используем код из предыдущего приложения.

1. В конце функции createScene() получим все анимации модели

Ogre::AnimationStateSet set = _SinbadEnt->getAllAnimationStates();*

2. Определим итератор

Ogre::AnimationStateIterator iter = set->getAnimationStateIterator();

3. Выведем на консоль все имена анимаций

```
while(iter.hasMoreElements())
{
  std::cout << iter.getNext()->getAnimationName() <<
std::endl;
}
```

4. Скомпилируем, запустим и увидим как в окне консоли напечатались все названия анимаций.

Что мы сделали.
Мы попросили модель дать нам список, содержащий все анимации. Затем мы в цикле распечатали в консоль все названия. Мы видим, что есть много анимаций, которых мы еще не использовали, но есть и которые уже использовали.

Подведем итоги.
Мы многому научились в этой главе, а именно как использовать анимации, что бы сделать сцену более интересной.
В частности мы рассмотрели следующее:

- Как получить анимации модели и проиграть их
- Как можно включить / выключить цикл анимации, и для чего нам передавать функции время последнего кадра
- Как можно проиграть две анимации одновременно
- Как к костям можно прикреплять разные объекты
- Как получить все анимации модели

В следующей главе вы увидите еще один аспект Ogre 3D, использование нескольких менеджеров сцен.

ГЛАВА 6
МЕНЕДЖЕРЫ СЦЕНЫ

Ogre 3D предлагает множество функций. В этой главе мы коснемся некоторых методов, которые мы не использовали раньше, только тех которые являются чрезвычайно важными для создания сложных 3D сцен.

В этой главе вы узнаете:

- Как изменить текущий менеджер сцены
- Что такое Octree
- Как создать собственный entity в коде
- Как ускорить наше приложение с помощью статической геометрии

Итак, давайте начнем…

Начнем с чистого листа.

На этот раз мы будем использовать новое пустое приложение.

1. Для начала подключим ExampleApplication

```
#include "Ogre\ExampleApplication.h"
```

2. Создадим новый класс приложения унаследованный от ExampleApplication и создадим новый метод createScene()

```
class Example41 : public ExampleApplication
{
public:
  void createScene()
  {

  }

};
```

3.Последнее что мы сделаем, создадим функцию main и запустим в ней наше приложение.

```
int main (void)
{
  Example41 app;
  app.go();
  return 0;
}
```

4. Как мы делали ранее, нужно описать все каталоги и библиотеки в настройках проекта. После запуска приложения вы должны получить черное окно и возможность выхода по ESCAPE.

Что мы сделали.

Мы создали новое приложение, которое наследуется от ExampleApplication и ничего не делает. Приложение имеет пустую функцию createScene(), потому что это чисто виртуальная функция из базового класса и если мы не заменим ее мы не сможем создать экземпляр нашего класса.

Получение типа менеджера сцены.

В этом разделе мы напечатаем имя и тип менеджера сцены, который используем.

1. Мы используем предыдущий код и модифицируем createScene()

```
std::cout << mSceneMgr->getTypeName() << "::" << mSceneMgr-
>getName() << std::endl;
```

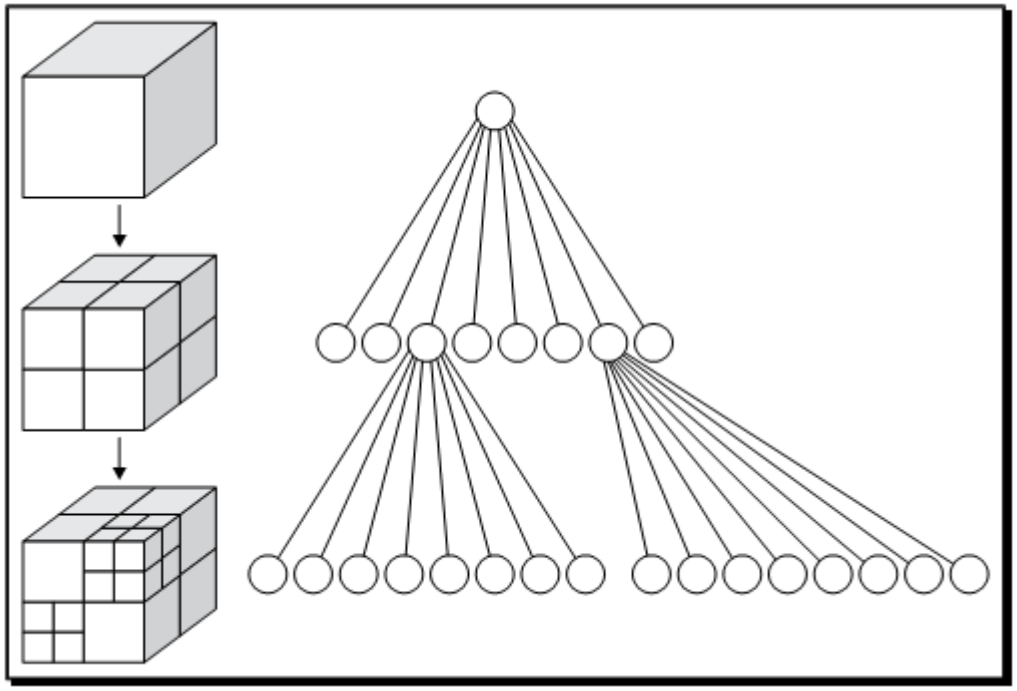
2. Скомпилируем и запустим. В консоль выведет сообщение OctreeSceneManager :: ExampleSMInstance.

Что мы сделали.

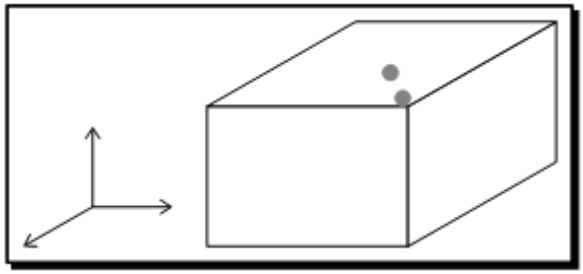
Мы добавили строку которая выводит тип и название менеджера сцены. В этом случае менеджер сцены называется ExampleSMInstance, который берется из ExampleApplication. SM означает SceneManager, если вы еще не догадались. Более интересная часть это тип, который в нашем случае OctreeSceneManager. Прежде чем вдаваться в подробности о том, что такое OctreeSceneManager, давайте обсуди, что такое менеджер сцены в целом.

Что такое менеджер сцены.
Менеджер сцены делает много вещей, которые будут очевидны, когда взглянем на документацию. Есть много функций, которые начинаются на create, delete, get, set и has. Мы уже использовали некоторые из этих функций, такие как createEntity(), createLight() или getRootSceneNode(). Одна из важнейших задач менеджера сцены – это управление объектами. Менеджер сцены работает как фабрика для объектов, узлов, моделей, света а также многих других типов объектов он может создать, или уничтожить. Каждый раз, когда мы хотим создать или уничтожить узел сцены мы используем менеджер сцены, в противном случае Ogre 3D может попытаться освободить ту же память позже, что может привести к необратимым последствиям.
Кроме управления объектами, он занимается оптимизацией сцены и расчетом позиции каждого объекта сцены для отображения. При отрисовке, пересчитываются положения только измененных узлов сцены, остальное остается от последнего кадра. Это экономит много времени вычислений и является важной задачей для менеджера сцены. Так же менеджер не обрабатывает для отображения невидимые в кадре объекты, что тоже экономит ресурсы. Существуют различные алгоритмы отборки невидимых объектов, это реализуется в различных менеджерах сцены. Все это время мы использовали OctreeSceneManager. Этот менеджер использует для хранения Octree сцены, отсюда и это название. Так что же такое Octree?

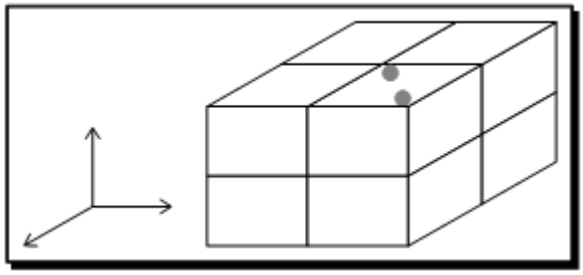
Octree.
Из названия ясно, что Octree является своего рода дерево. Как и всякое дерево, у него есть корень, и каждый узел имеет родителя. Что делает его особенным, это то, что каждый узел имеет не более восьми детей, отсюда и название Octree. Ниже приводится диаграмма, показывающая Octree:



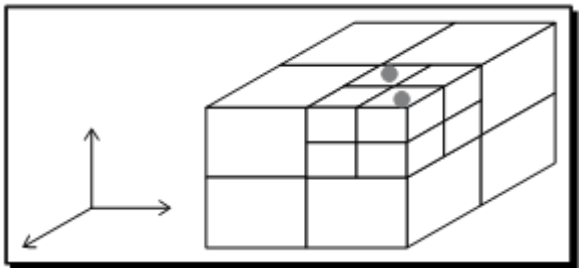
Так зачем же Ogre 3D использует Octree для хранения 3D сцены? Octree обладает некоторыми свойствами, которые чрезвычайно полезны для хранения 3D сцен. Одним и них является то, что он может иметь до восьми детей. Например, если у нас есть 3D сцена с двумя объектами в ней, мы можем заключить эту сцену в куб.



Если мы разделим этот куб на половину его ширины, высоты и глубины, мы получим восемь новых кубиков, эти восемь кубиков можно рассматривать как восемь детей исходного куба.



Теперь два объекта в правом верхнем кубе. Семь других пусты. Мы делим куб содержащий эти два объекта.



Теперь каждый куб имеет по одному объекту. Это свойство Octree делает отбор легким и быстрым. При отображении сцены, у нас есть камера с плоскостью отсечения внутри сцены. Что бы определить какие объекты должны быть нарисованы, мы начинаем с корня Octree. Если плоскость отсечения пересекает куб, мы продолжаем. Это должно быть всегда так, потому что Octree на уровне 0 охватывает всю сцену с камеры. Затем мы продолжаем и переключаемся на детей корневого узла. Мы проверяем на отсечение каждого ребенка, и если оно пересекается, то мы продолжаем с детьми уже этого куба.Кроме того, если куб полностью усеченный, мы не должны идти углубляться, потому что мы знаем, что дети находятся в кубе. Мы делаем это пока не перейдем к пустоте, затем переходим к следующему ребенку. Красота этого алгоритма в том, что можно отказаться от рисования ненужных объектов всего за несколько шагов.
Этот подход похож на бинарное дерево поиска. Разница лишь в том, что он использует восемь детей вместо двух.

Прочие типы менеджеров сцены.
Мы видели менеджер одной из сцен, а теперь давайте посмотрим на другой.
1. Используем код из предыдущего примера, удалим весь код из createScene()
2. Добавим новую функцию под названием chooseSceneManager()
```
virtual void chooseSceneManager(void)
{
}
```
3. Теперь добавим код в новую функцию, что бы определить нужные ресурсы.
```
ResourceGroupManager::getSingleton().addResourceLocation("../../media/packs/chiropteraDM.pk3", "Zip",
        ResourceGroupManager::getSingleton().getWorldResourceGroupName(), true);
```
4. После определения добавим код для загрузки ресурсов
```
ResourceGroupManager::getSingleton().initialiseResourceGroup(
        ResourceGroupManager::getSingleton().getWorldResourceGroupName());
```
5. Теперь нам нужно вызвать createSceneManager()
```
mSceneMgr = mRoot->createSceneManager("BspSceneManager");
```
6. Теперь скажем менеджеру сцены, что мы хотим отобразить ранее загруженные ресурсы
```
mSceneMgr->setWorldGeometry("maps/chiropteradm.bsp");
```
7. Скомпилируйте и запустите приложение. Вы увидите уровень из известной игры.

Что мы сделали.
Мы использовали chooseSceneManager(), что бы создать сцену, менеджер которой отличается от установленного по умолчанию. В этом случае мы использовали BspSceneManager. BSP это двоичный формат карты, в которой используются техники хранения информации из старых шутеров. BSP разбивает уровень на выпуклые части и сохраняет их в виде дерева. Это делает отображение более быстрым на старых графических картах. В настоящее время это не всегда так, и поэтому BSP больше не используется.

Менеджер ресурсов.
В первой строке мы использовали новый для нас менеджер ResourceGroupManager. Этот менеджер несет ответственность за все файлы загружаемые в течение жизни нашего приложения. Во время запуска, менеджер получает список каталогов и zip архивов, из которых мы хотим загружать файлы. Этот список можно читать из файла resources.cfg, или можно все каталоги прописать в коде приложения. После этого мы можем создать объект используя только его имя файла. Нам не нужен полный путь к файлу, так как менеджер уже проиндексировал его, на самом деле индексный файл загружается только в момент создания объекта. Индексация помогает нам не загружать в память одну и ту же модель несколько раз. Когда нам нужно несколько экземпляров одного и того же объекта, менеджер копирует ранее загруженную модель, и не происходит обращения к диску.
AddResourceLocation() принимает путь к папке или ZIP архиву, второй параметр определяет, какой это тип, как правило это может быть либо zip архив, либо папка. Мы можем, если это необходимо, добавить свои собственные типы ресурсов, и загрузчик к ним. Это полезно если мы хотим использовать собственные типы пакетов.
Третий параметр это имя группы ресурсов. Группы ресурсов это как пространства имен в C++, поскольку мы загружаем карту, которая является частью игрового мира, мы будем использовать предопределенное имя группы ресурсов WorldResourceGroup. Последний параметр указывает Ogre 3D использовать рекурсию при загрузке или нет. Если нет, то ресурсы будут загружены только из каталога, если с рекурсией, то ресурсы будут загружены из подкаталогов. По умолчанию рекурсия отключена.
При вызове функции initialiseResourceGroup(), мы говорим Ogre 3D проиндексировать все файлы ResourceGroup, которые мы еще не индексировали. Конечно мы должны указать какую группу ресурсов индексировать. После вызова этой функции, мы можем использовать все файлы из этой группы ресурсов.

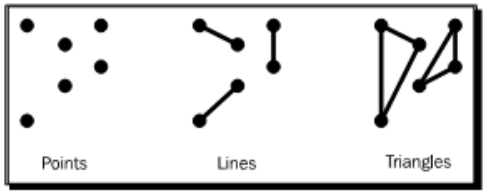
setWorldGeometry
setWorldGeometry() представляет собой специальный вызов функции, который говорит BspSceneManager-у какую карту уровня загрузить.

Создание нашей собственной модели.
Мы увидели как использовать различные менеджеры сцены и, как загружать BSP уровни. Сейчас мы увидим как создать сетку в коде. Мы собираемся сделать плоскость с травой.
1. Нам нужна простая программа:
```
class Example43 : public ExampleApplication
{
private:

public:
  void createScene()
  {

  }
};
```
2. Первое, что нужно сделать в createScene() является определение плоскости. Мы будем использовать плоскость в кажестве основы для нашей травы:
```
Ogre::Plane plane(Vector3::UNIT_Y, -10);
Ogre::MeshManager::getSingleton().createPlane("plane",ResourceGroupManager::DEFAULT_RESOURCE_
GROUP_NAME, plane,1500,1500,200,200,true,1,5,5,Vector3::UNIT_Z);
```
3. Затем установим материал для нашей плоскости. Мы будем использовать материал GrassFloor из примеров:
```
Ogre::Entity* ent = mSceneMgr->createEntity("GrassPlane","plane");
mSceneMgr->getRootSceneNode()->createChildSceneNode()->attachObject(ent);
ent->setMaterialName("Examples/GrassFloor");
```
4. Затем к сцене добавим направленный свет, иначе будет слишком темно, что бы что либо увидеть:
```
Ogre::Light* light = mSceneMgr->createLight("Light1");
light->setType(Ogre::Light::LT_DIRECTIONAL);
light->setDirection(Ogre::Vector3(1,-1,0));
```

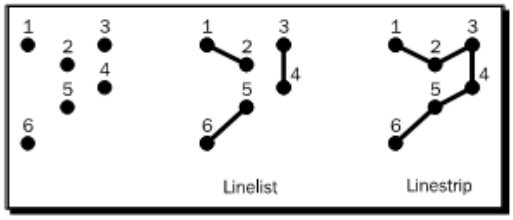
5. Создадим новый ManualObject и вызовем метод begin():
Ogre::ManualObject manual = mSceneMgr->createManualObject("grass");*
manual->begin("Examples/GrassBlades", RenderOperation::OT_TRIANGLE_LIST);
6. Добавим первый полигон с позицией текстурных координат для каждой вершины:
manual->position(5.0, 0.0, 0.0);
manual->textureCoord(1,1);
manual->position(-5.0, 10.0, 0.0);
manual->textureCoord(0,0);
manual->position(-5.0, 0.0, 0.0);
manual->textureCoord(0,1);
7. Нужно добавить еще один треугольник, что бы завершить квадрат:
manual->position(5.0, 0.0, 0.0);
manual->textureCoord(1,1);
manual->position(5.0, 10.0, 0.0);
manual->textureCoord(1,0);
manual->position(-5.0, 10.0, 0.0);
manual->textureCoord(0,0);
8. Мы закончили создание объекта, скажем об этом движку:
manual->end();
9. Наконец, создадим новый узел и приложим к нему свой объект:
Ogre::SceneNode grassNode =*
mSceneMgr->getRootSceneNode()->reateChildSceneNode("GrassNode2");
grassNode->attachObject(manual);
10. Скомпилируйте и запустите, вы должны увидеть плоскость с текстурой травы, и траву, которая находится выше плоскости.

Что мы сделали.
Мы нарисовали плоскость с травой и мы создали четырехугольник и наложили на него текстуру травы. Шаги с 1 по 4, должны быть легкими, так как это мы уже рассматривали. Разница лишь в тм, что мы использовали различные материалы. Материалы более подробно мы рассмотрим в следующей главе. В 5 пункте произошло кое-что новое. Мы создали ManualObject.

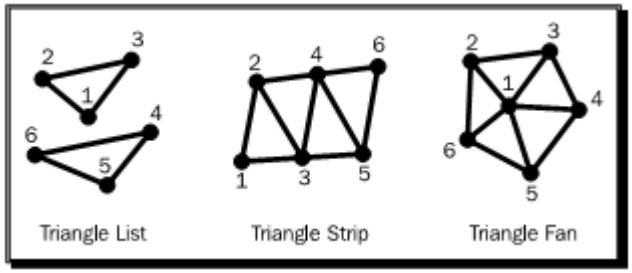
ManualObject
С помощью ManualObject можно создать много различных объектов. Что бы создавать модели мы должны задавать координаты точек, которые образуют треугольники. Мы уже говорили, что все объекты состоят из треугольников. Для четырехугольника нам требуется 2 треугольника. На 5 шаге мы создали новый свой объект, и назвали его просто как трава. Метод begin() нуждается в указании материала, как мы собираемся обрабатывать информацию по вершинам. Есть 6 различных способов как мы можем составлять точки и 3 вещи, которые можем создавать с помощью своего объекта, а именно точки, линии и треугольники.



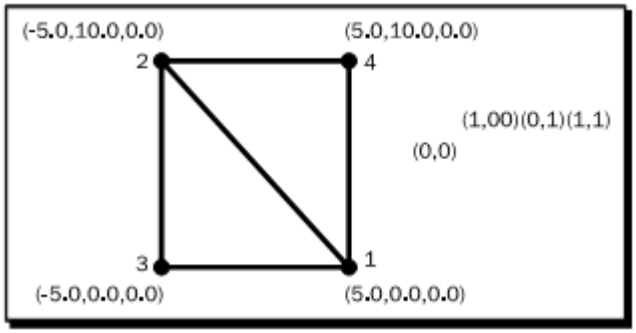
Точки просто хранятся в виде списка. Каждый раз, когда мы добавляем новые позиции, мы создаем новую точку. Этот режим называется OT_POINT_LIST. Для линий, существует 2 различных способа создания. Простой способ заключается в использовании первой и второй позиции в качестве первой линии, третий и четвертый на вторую линию, и так далее. Это называется OT_LINE_LIST. Другой способ заключается в использовании первые две точки в качестве первой линии, затем каждая новая точка определяет конечную точку нового списка и использует последнюю точку как начальную для следующей линии, это OT_LINE_STRIP.



Треугольники могут быть определены тремя способами. Первый и самый простой способ это список треугольников. Первые три точки образуют треугольник, вторые три точки – второй треугольник и так далее. Это называется OT_TRIANGLE_LIST. Второй способ это когда первые три точки определяют треугольник, а каждая следующая точка образует новый треугольник беря свое начало от двух последних точек. Это называется OT_TRIANGLE_STRIP. Последний вариант заключается в использовании первых трех точек для предыдущего треугольника, затем каждый новый треугольник берет первую точку списка и последнюю созданную это называется OT_TRIANGLE_FAN.

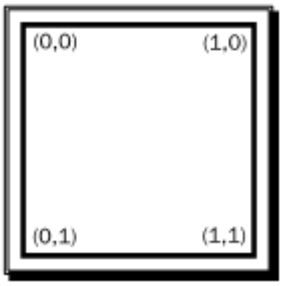


Мы видим, что в зависимости от режима ввода, получается разное количество треугольников для описания 3D фигур. Для списка треугольников нам нужны три точки на каждый треугольник. С STRIP или FAN нам нужны 3 точки для первого треугольника, затем по одной на каждый новый.
Во время начала определения точек мы говорим что хотим использовать список треугольников, что бы описать наш квадрат. Мы хотим, что бы наш четырехугольник был на 10 единиц в длину и 10 единиц в высоту.



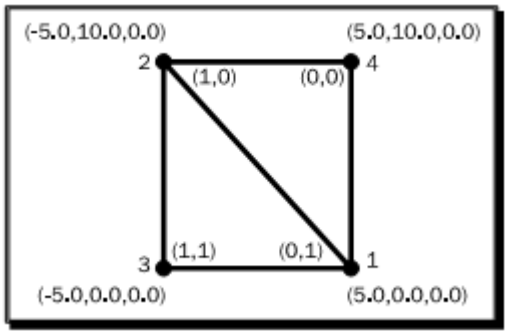
Первому треугольнику нужны точки 1, 2 и 3. Второй треугольник нуждается в точках 1, 2 и 4.
На 6 шаге определены первые точки треугольника, на шаге 7 вторые. Вы наверное заметили, что после каждой новой точки вызывается фунция textureCoord().

Текстуры.
Для Ogre 3D, что бы иметь возможность наложить образ травинки на наш квадрат, каждая вершина нуждается в объявлении текстурных координат, кроме своей позиции. Они состоят из двух значений (U, V). (U,V) описывает место на изображении, где 0,0 означает верхний левый угол изображения, а 1,1 означает правый нижний.



Если мы будем использовать значение больше 1, то текстура начнет отображать различные эффекты в зависимости от настроек материала. Если мы будем использовать режим WRAP, то текстура будет повторяться. В режиме CLAMP все значения больше 1 будут равняться 1. В режиме MIRROR единица превращается в 0, а 0 в единицу. Это является зеркальным отображением текстуры. Последний режим использует рамку, все что вне диапазона [0,1], будет рисоваться в цвет границы.

Информация о текстурных координатах в нашем квадрате выглядит следующим образом:

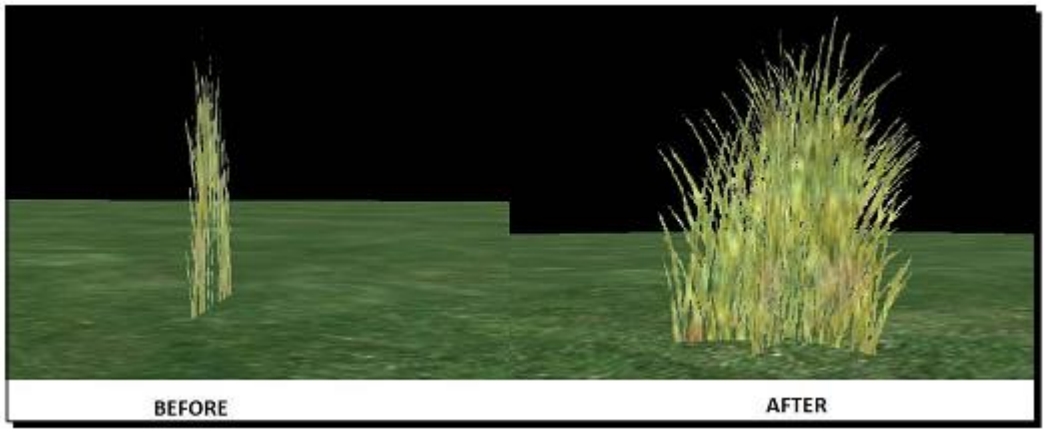


Обратите внимание на пункты 6 и 7. Сравните строки кода с предыдущей картинкой, положение и координаты текстур должны совпадать. На 8 шаге мы закончили создавать объект, а на 9 шаге мы прикрепили его к узлу сцены.

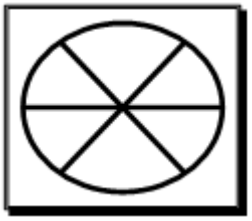
Добавление объема траве.
Мы сумели сделать травинку, но при перемещении камеры становится сразу ясно, что это всеголишь 2D картинка, и не имеет никакого объема. Эта проблема не может быть легко решена, если только не делать каждую травитку трехмерной моделью, но если делать большие луга таким методом, то это быстро понизит быстродействие. Есть несколько методов, что бы избежать эту проблему. Сейчас мы увидим один из них.

1. Мы будем использовать предыдущий код, но добавим еще два квадрата в наш объект:
//third triangle
manual->position(2.5, 0.0, 4.3);
manual->textureCoord(1,1);
manual->position(-2.5, 10.0, -4.3);
manual->textureCoord(0,0);
manual->position(-2.0, 0.0, -4.3);
manual->textureCoord(0,1);
//fourth triangle
manual->position(2.5, 0.0, 4.3);
manual->textureCoord(1,1);
manual->position(2.5, 10.0, 4.3);
manual->textureCoord(1,0);
manual->position(-2.5, 10.0, -4.3);
manual->textureCoord(0,0);
2. Еще один квадрат:
//fifth triangle
manual->position(2.5, 0.0, -4.3);
manual->textureCoord(1,1);
manual->position(-2.5, 10.0, 4.3);
manual->textureCoord(0,0);
manual->position(-2.0, 0.0, 4.3);
manual->textureCoord(0,1);
//sixth triangle
manual->position(2.5, 0.0, -4.3);
manual->textureCoord(1,1);
manual->position(2.5, 10.0, -4.3);
manual->textureCoord(1,0);
manual->position(-2.5, 10.0, 4.3);
manual->textureCoord(0,0)

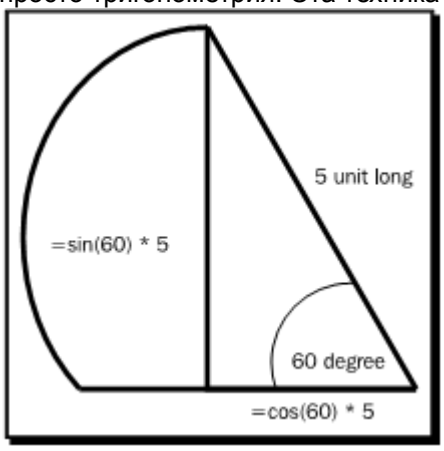
3. Скомпилируйте и запустите приложение, а затем переместитесь по траве.



Что мы сделали.
Мы исправили проблему, когда увидели, что трава выглядит как плоское изображение. Что бы решить эту проблему, мы просто создали еще 2 новых квадратика и повернули их, а затем засунули их друг в друга. Пример на диаграмме:



Каждый четырехугольник имеет туже длину, так как мы можем представить картину ввиде круга разделенного на 6 частей. 3 квадрата пересекающихся в центре означают, что у нас есть 6 углов в 60 градусов, что делает всего 360 градусов. Как можно вычислить точки для других двух квадратов? Это просто тригонометрия. Эта техника в компьтерных играх называется billboarding.

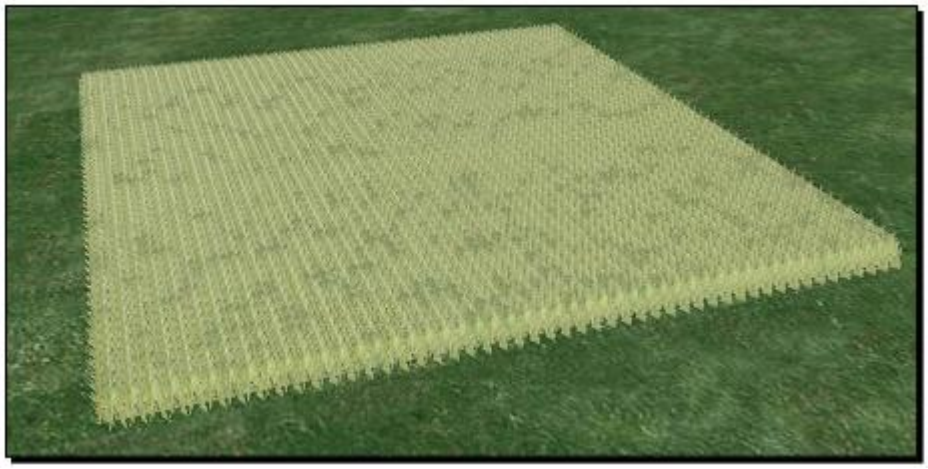


Создание поля с травой.
Теперь когда у нас есть модель одной травинки, давайте создадим целое поле с травой.
1. Нужно несколько экземпляров нашей травинки, что бы их сделать нужно преобразовать в сетку:
manual->convertToMesh("BladesOfGrass");
2. Мы хотим создать поле с травой 50x50. Так что нам нужно 2 цикла:
```
for(int i=0;i<50;i++)
{
  for(int j=0;j<50;j++)
  {
```
3. Внутри цикла создадим безымянный узел сцены и безымянную сущность entity:
```
Ogre::Entity * ent = mSceneMgr->createEntity("BladesOfGrass");
Ogre::SceneNode* node =
mSceneMgr->getRootSceneNode()->createChildSceneNode(Ogre::Vector3(i*3,-10,j*3));
node->attachObject(ent);
```

4. Закроем циклы:

```
  }
}
```

5. Скомпилируйте и запустите, вы увидите поле травы. В зависимости от вашего компьютера, это приложение может быть довольно медленным.



Что мы сделали.

На шаге 1, мы использовали новую функцию из класса своего объекта, которая преобразует объект в сетку, и мы можем создавать экземпляры используя функция createEntity() менеджера сцены. Для того, что бы испоьлзовать новый объект, нужно имя, которое будет использоваться в дальнейшем в качестве параметра для функции createEntity(). Мы использовали BladesOfGrass как описательное имя. Так как мы хотим несколько экземпляров нашей травы мы создали 2 цикла в пункте 2, каждый работает 50 итераций. На 3 шаге мы добавили в цикл тело. Мы создавали новый объект с использованием имени нашей сетки с травой. Внимательный читатель может заметить, что мы не использовали createEntity с двумя параметрами, именем объекта и именем сетки. Мы не всегда должны задавать имя создаваемым сущностям, так как имена должны быть уникальными, и в цикле это было бы неудобно.

Посмотрим какие имена за нас генерирует Ogre.

1. Мы должны добавить одну новую строку, что бы получить имена. Добавьте в конец цикла следующее:

```
std::cout << node->getName() << "::" << ent->getName() << std::endl;
```

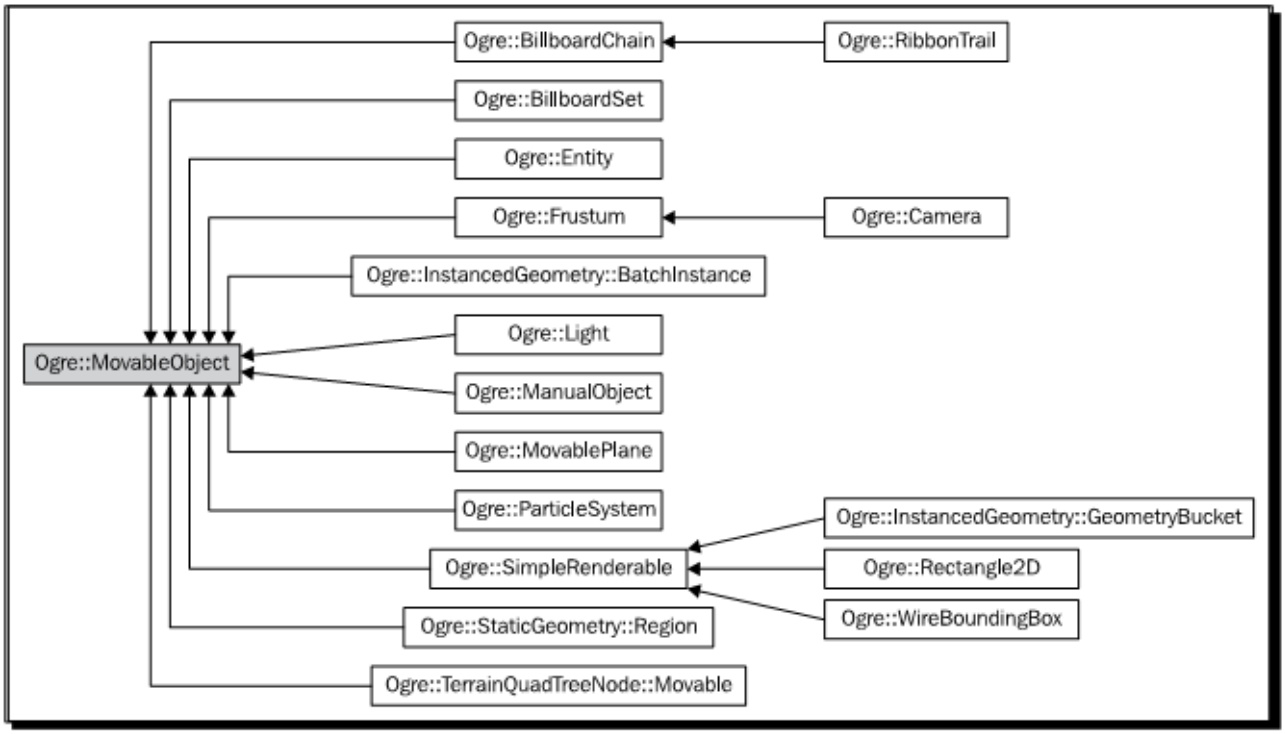
2. После компиляции и запуска, в окне консоли вы увидите примерно следующее:

Unnamed_2495::Ogre/MO2494
Unnamed_2496::Ogre/MO2495
Unnamed_2497::Ogre/MO2496
Unnamed_2498::Ogre/MO2497
Unnamed_2499::Ogre/MO2498
Unnamed_2500::Ogre/MO2499

Будут описаны имена для всех 2500 созданных объектов.

Что мы сделали.

Мы просто напечатали имена узлов сцены, что бы понять как именуются объекты автоматическим способом. Мы видим, что имена узлов сцены используют следующую схему: Unnamed_Nr, где Nr является счетчиком, возрастающий каждый раз, когда мы создаем новый безымянный узел сцены. Сущности используют похожую схему, но они используют MO_ вместо Unnamed. MO является короткой формой для MOvableObject. MovableObject является базовым классом для всех подвижных объектов. Все что может быть перемещено в сцене, наследуется от MovableObject. Вот пример из документации:



Source: `http://www.ogre3d.org/docs/api/html/classOgre_1_1MovableObject.html`

Мы видим, что даже камера является подвижным объектом, это нужно для того что бы мы могли прикрепить ее к узлу сцены. Только производные класса MovableObject могут прикрепляться к узлам сцены.

Статическая геометрия.

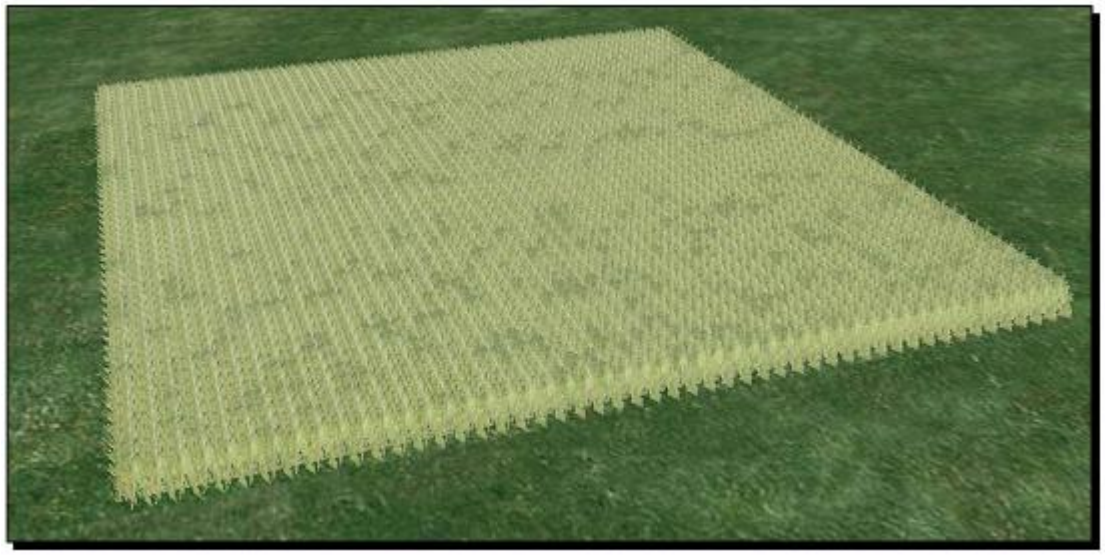
Мы создали поле травы, но приложение может быть довольно медленным в зависимости от вашего компьютера. Сделаем приложение быстрее, используя класс Ogre 3D StaticGeometry.

Мы намерены изменить код предыдущего примера, что бы сделать его быстрее:

1. Удалите оператор печати в консоль, нам это больше не нужно.
2. Теперь вернемся к ManualObject и удалим все повторяющиеся pozition() функции:

```
manual->position(5.0, 0.0, 0.0);
manual->textureCoord(1,1);
manual->position(-5.0, 10.0, 0.0);
manual->textureCoord(0,0);
manual->position(-5.0, 0.0, 0.0);
manual->textureCoord(0,1);
manual->position(5.0, 10.0, 0.0);
manual->textureCoord(1,0);
manual->position(2.5, 0.0, 4.3);
manual->textureCoord(1,1);
manual->position(-2.5, 10.0, -4.3);
manual->textureCoord(0,0);
manual->position(-2.0, 0.0, -4.3);
manual->textureCoord(0,1);
manual->position(2.5, 10.0, 4.3);
manual->textureCoord(1,0);
manual->position(2.5, 0.0, -4.3);
manual->textureCoord(1,1);
manual->position(-2.5, 10.0, 4.3);
manual->textureCoord(0,0);
manual->position(-2.0, 0.0, 4.3);
manual->textureCoord(0,1);
manual->position(2.5, 10.0, -4.3);
```

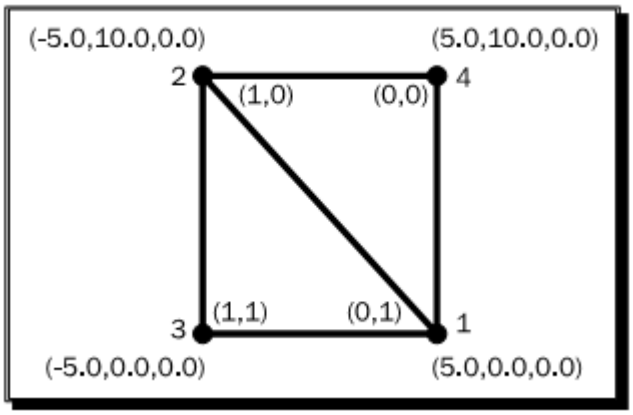
```
manual->textureCoord(1,0);
```
3. Теперь мы опишем треугольники которые мы хотим создать с помощью так называемых индексов. Первый четырехугольник состоит из двух треугольников, первый использует первые три точки, а второй использует первую, вторую и четвертую. Имейте ввиду, точки считаются начиная с нуля.
```
manual->index(0);
manual->index(1);
manual->index(2);
manual->index(0);
manual->index(3);
manual->index(1);
```
4. Опишем два остальных квадрата:
```
manual->index(4);
manual->index(5);
manual->index(6);
manual->index(4);
manual->index(7);
manual->index(5);
manual->index(8);
manual->index(9);
manual->index(10);
manual->index(8);
manual->index(11);
manual->index(9);
```
5. Создадим экземпляр класса StaticGeometry:
```
Ogre::StaticGeometry* field = mSceneMgr->createStaticGeometry("FieldOfGrass");
```
6. Теперь в цикле добавим создаваемые сущности:
```
for(int i=0;i<50;i++)
{
  for(int j=0;j<50;j++)
  {
    Ogre::Entity * ent = mSceneMgr->createEntity("BladesOfGrass");
    field->addEntity(ent,Ogre::Vector3(i*3,-10,j*3));

  }
}
```
7. Закончим создание статической геометрии:
```
field->build();
```
8. Скомпилируйте и запустите приложение, и вы увидите то же самое поле траы, но приложение теперь выполняется намного быстрей.



Что мы сделали.
Мы создали туже сцену, чть и раньше, но на этот раз она работает быстрее. Почему? Единственная причина, почему работает быстрее, это использование статической геометрии. Так как же статическая геометрия отличается от ранее описаного метода.

Рендеринг.
Каждый раз, когда мы отрисосвываем сцену, Ogre 3D и графическая карта должны выполнить некоторые действия. Мы уже рассмотрели некоторые из них, но не все. Теперь давайте обсудем некоторые действия, которые мы не встречали до сих пор. Мы уже знаем, что существуют различные пространства объекта, например локальное или мировое.
Мы также знаем, для того что бы показать объект на сцене, нужно превратить локальное пространство в мировое. Переход от локального пространства в мировое, представляет собой сочетание простых математических операций, но они занимают некоторое время вычислений. При отображении наших 2500 объектов травы, Ogre 3D начинает считать все позиции для каждой травинки. Это очень много операций в кадре, но еще хуже то, что каждая травинка отправляется отдельно на GPU для рендеринга. Это занимает много времени и поэтому приложение работает медленно.
Мы можем решить эту проблему с помощью статической геометрии. Мы создали экземпляр класса статической геометрии, используя менеджер сцены на 5 шаге. Тем не менее, внутри цикла, мы добавили создание сущностей не в узел сцены, а непосредственно в статическую геометрию, и в качестве второго параметра дали позиции.
После завершения добавления статической геометрии мы должны были вызвать build(). Эта функция вычисляет мировые координаты для всех добавленных объектов. К тому же эта функция объединяет все объекты в одно целое для оптимизации. Цена которую мы за это платим – это невозможность двигать объекты, добавленные в статическую геометрию. Как привило статическая геометрия используется для всего, что не движется на сцене, потому что это дает огромное ускорение практически без недостатков. Одним из недостатков является то, что когда у нас есть большая часть сцены в статической геометрии, отбор менее эффективен, потому что, когда одна часть статической геометрии внитри усеченного зрения, то все это должно быть вынесено.

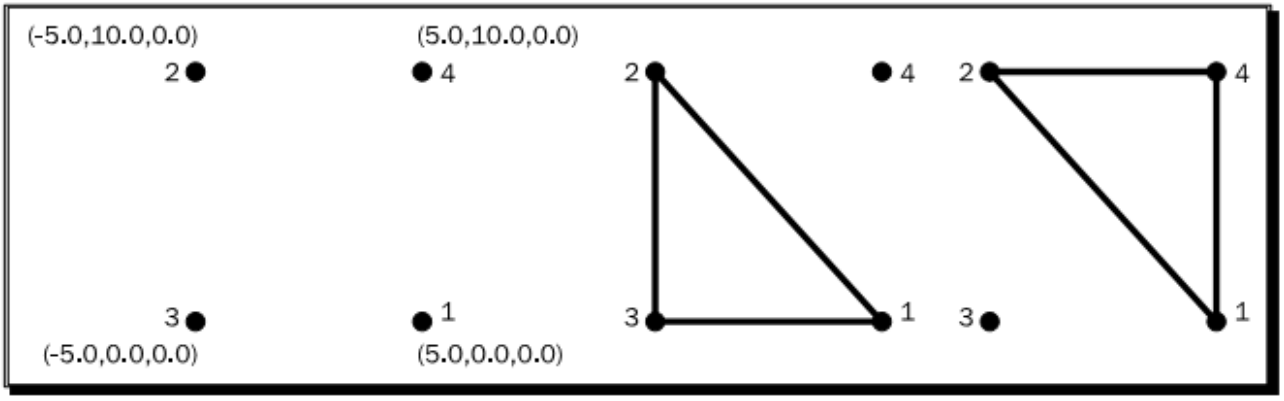
Индексы.
Мы обнаружили, что можем добавлять только сущности, которые используются в статической геометрии. Но мы не обсуждали индексы. Вернемся к нашему квадрату:



Этот квадрат имеет 4 точки, которые определяют два треугольника, которые образуют квадрат. Когда мы смотрим код для создания этого четырехугольника, мы замечаем, что мы добавили шесть точек вместо четырех, а 2 точки дублируются.

```
//First triangle
manual->position(5.0, 0.0, 0.0);
manual->textureCoord(1,1);
manual->position(-5.0, 10.0, 0.0);
manual->textureCoord(0,0);
manual->position(-5.0, 0.0, 0.0);
manual->textureCoord(0,1);
//Second triangle
manual->position(5.0, 0.0, 0.0);
manual->textureCoord(1,1);
manual->position(5.0, 10.0, 0.0);
```

manual->textureCoord(1,0);
manual->position(-5.0, 10.0, 0.0);
manual->textureCoord(0,0);
Точки 1 и 2 на рисунке добавлены 2 раза, потому что они используются в обоих треугольниках. Одним из способов предотвращения дублирования этой информации является использование двухступенчатой системы для описания треугольников. Первое что мы делаем это создаем список точек, которые будут использоваться, а второе это список который определяет порядок создания треугольников.



Здесь мы определили 4 точки, а затем сказали, что один треугольник использует точки 1, 2 и 3, а второй использует точки 1, 2 и 4. Это избавляет нас от добавления дублированной информации. Здесь может показаться, что это небольшая разница, но когда у нас в модели несколько тысяч треугольников, это может изменить ситуацию. Статическая геометрия требует, что бы мы использовали сущности только с индексами, таким образом класс статической геометрии может создать один большой список всех точек, и список всех индексов. Для больших уровней это является большой экономией памяти.

Подведем итоги.
Мы изменили текущий менеджер сцены, создали поле с травой, и ускорили приложение с помощью статической геометрии.
В частности мы рассмотрели:

- Что такое ManualObject
- Зачем использовать индексы в нашей 3D модели
- Как и когда использовать статическую геометрию.

Мы уже испоьлзовали материалы в этой главе. В следующей главе мы будем создавать собственные материалы.

ГЛАВА 7
МАТЕРИАЛЫ

Без материалов не получится показать нашу сцену с мельчайшими деталями. Эта глава является введением в обширную область применения материалов. Материалы – действительно важная тема, и необходимо понимать их, для произведения красивых сцен.

В этой главе вы узнаете:

- Как создать свой собственный материал
- Применить текстуру четырехугольнику
- Лучше поймете, как работает конвейер рендеринга
- Как использовать шейдеры для создания эффектов, которые без них невозможны

Итак, приступим…

Создание белого квадрата.

В предыдущей главе мы создали свою собственную модель в коде. Теперь мы будем ее использовать для экспериментов.

Мы начнем с пустого приложения и вставим код создания нашего четырехугольника

1. Начнем с создания ManualObject:

```
Ogre::ManualObject*  manual = mSceneMgr->createManualObject("Quad");
manual->begin("BaseWhiteNoLighting", RenderOperation::OT_TRIANGLE_LIST);
```

2. Создадим четыре точки нашего квадрата:

```
manual->position(5.0, 0.0, 0.0);
manual->textureCoord(0,1);
manual->position(-5.0, 10.0, 0.0);
manual->textureCoord(1,0);
manual->position(-5.0, 0.0, 0.0);
manual->textureCoord(1,1);
manual->position(5.0, 10.0, 0.0);
manual->textureCoord(0,0);
```

3. Используем индексы для описания квадрата:

```
manual->index(0);
manual->index(1);
manual->index(2);
manual->index(0);
manual->index(3);
manual->index(1);
```

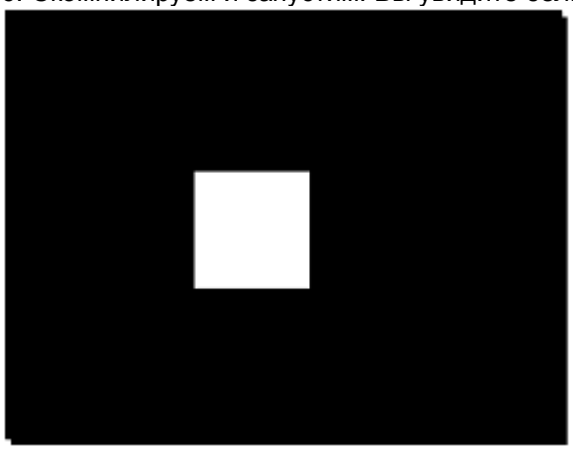
4. Закончим создание квадрата и конвертируем в сетку:

```
manual->end();
manual->convertToMesh("Quad");
```

5. Создадим экземпляр квадрата и прикрепим его к узлу сцены:

```
Ogre::Entity * ent = mSceneMgr->createEntity("Quad");
Ogre::SceneNode* node = mSceneMgr->getRootSceneNode()->createChildSceneNode("Node1");
node->attachObject(ent);
```

6. Скомпилируем и запустим. Вы увидите белый квадрат:



Что мы сделали.
Мы использовали знания из предыдущей главы для создания четырехугольника. Следующим шагом является создание нашего собственного материала.

Показывать белый квадрат – это неинтересно, так что давайте создадим наш первый материал.
1. Измените название материала при создании квадрата на MyMaterial:
manual->begin("MyMaterial1", RenderOperation::OT_TRIANGLE_LIST);
2. Создайте в каталоге с Ogre 3D media\materials\scripts новый файл под названием Ogre3DBeginnersGuide.material.
3. Напишите в него следующий код:

```
material MyMaterial1
{
  technique
  {
     Pass
    {
        texture_unit
        {
           texture leaf.png
        }
    }
  }
}
```

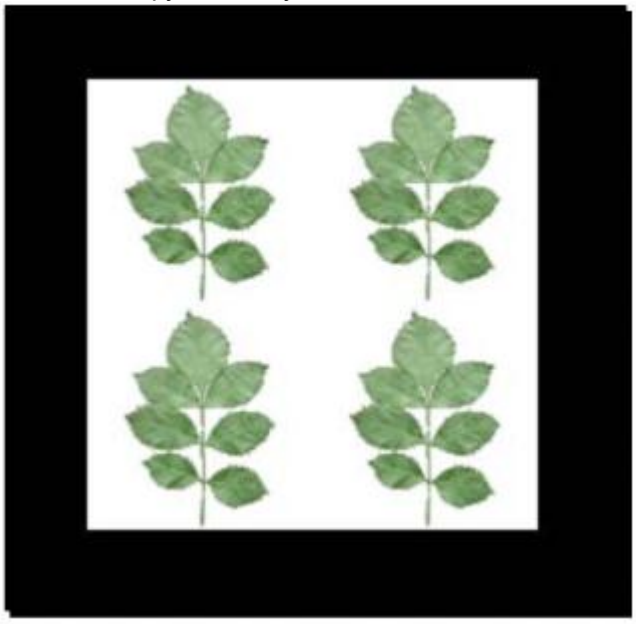
4. Скомпилируйте приложение и запустите. Вы увидите как на белом квадрате рисуется растение:

Что мы сделали.
Мы создали наш первый файл материала. В Ogre 3D материалы могут быть определены в специальных файлах материалов. Что бы движок смог найти материал, он должен быть помещен в каталог указанный в ресурсах (resources.cfg). Мы также могли дать путь к файлу непосредственно из кода программы, используя ResourceManager, как мы это делали в предыдущей главе с картой BSP. Для использования нашего материала, определенного в файле, мы обязаны были использовать имя материала при создании ManualObject.

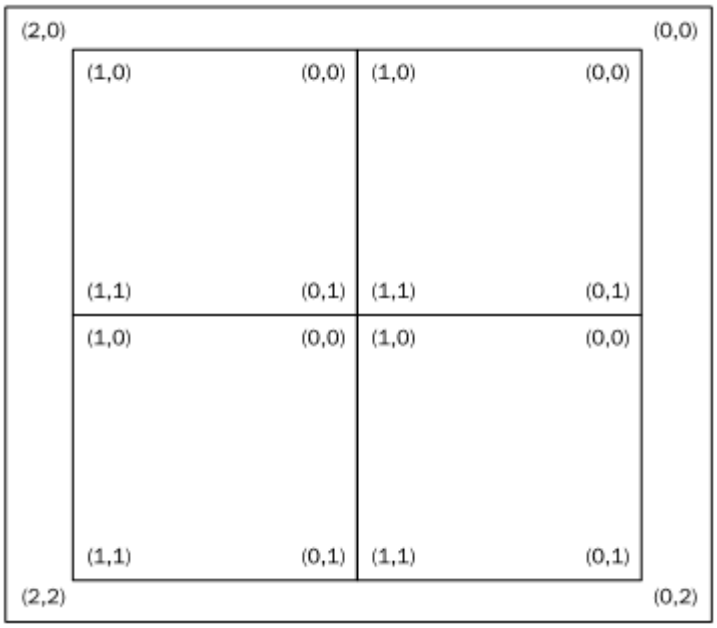
Материалы.
Каждый материал начинается с ключевого слова material, название материала, а затем открыть фигурный скобки. Что бы завершить материал, нужно закрыть фигурную скобку. Эта технология должна быть вам знакома. Каждый материал состоит из одного или нескольких методов; technique описывает способ достижения желаемого эффекта. Так как есть много разных графических карт с разными возможностями, мы можем определить несколько методов в файле материала, и Ogre 3D выбирает первый подходящий метод. Внутри техники может быть несколько проходов (pass). В основном мы сейчас будем создавать материалы с одним проходом. Тем не менее для некоторых материалов требуется два или даже три прохода. Далее в проходе мы определяем текстурные блоки. Текстура определеяет один текстурный блок и его свойства. На этот раз мы использовали одно свойство – это файл изображения. Мы использовали leaf.jpg, эта текстура поставляется вместе с Ogre

3D SDK, и индексируется в resources.cfg, поэтому мы можем использовать ее без кокой либо работы с нашей стороны.

Создадим другой материал.
Создадим новый материал и назовем его MyMaterial2, в котором используем изображение Water02.jpg.
В предыдущей главе мы обсуждали различные алгоритмы используемые при наличии текстурных координат вне диапазона 1,0. Теперь давайте создадим несколько материалов, что бы увидеть их в действии.
1. Используем квадрат из предыдущего примера. Изменим текстурные координаты от 0 до 2:
manual->position(5.0, 0.0, 0.0);
manual->textureCoord(0,2);
manual->position(-5.0, 10.0, 0.0);
manual->textureCoord(2,0);
manual->position(-5.0, 0.0, 0.0);
manual->textureCoord(2,2);
manual->position(5.0, 10.0, 0.0);
manual->textureCoord(0,0);
2. Скомпилируем и запустим:



Что мы сделали.
Мы просто изменили координаты текстур с нуля до двух, у нашего квадрата. Это означает, что Ogre необходимо использовать один из своих алгоритмов обрабатывания текстурных координат, который больше 1. По умолчанию используется режим WRAP. Это означает, что каждое значение больше 1, будет преобразовываться в диапазон от 0 до 1. Ниже приводится диаграмма показывающая этот эффект.



Мы видим как получается отображение текстуры в режиме WRAP. Наша текстура не может показать полезность этого метода. Что бы увидеть преимущество, мы изменим программу.

1. Создадим новый материал, похожий на предыдущий, только изменим используемую текстуру на terr_rock6.jpg:

```
material MyMaterial3
{
   technique
   {
      pass
      {
         texture_unit
         {
            texture terr_rock6.jpg
         }
      }
   }
}
```

2. Изменим название материала при создании квадрата на MyMaterial3:
manual->begin("MyMaterial3", RenderOperation::OT_TRIANGLE_LIST)
3. Скомпилируем и запустим. И увидим как на квадрат легла текстура гор:

Что мы сделали.
На этот раз, кажется, что квадрат покрыт одной текстурой. Мы не видим очевидных повторений, как было с текстурой растения. Причиной этого является то, что текстура была сделана таким образом, чтобы повторяться без швов. Такая текстура называется бесшовная. У таких текстур все стороны идеально подходят друг у другу. Если бы это было не так, мы бы увидили, как они повторяются.

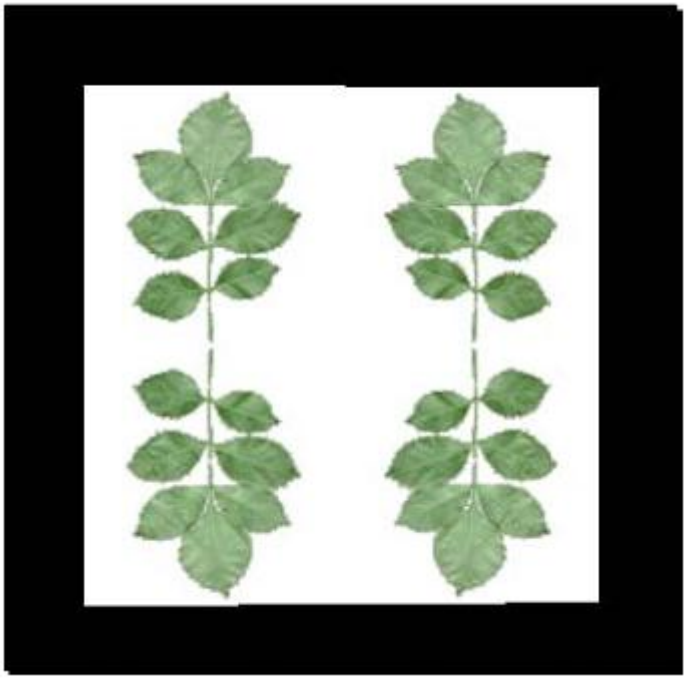
Используем другие алгоритмы текстурирования.
Мы уже увидели как работает режим WRAP, а теперь посмотрим на режим CLAMP.
1. Мы будем использовать тот же проект, только создадим новый материал. Создадим материал с названием MyMaterial4:

```
material MyMaterial4
{
   technique
   {
      pass
      {
         texture_unit
         {
            texture terr_rock6.jpg
         }
      }
   }
}
```

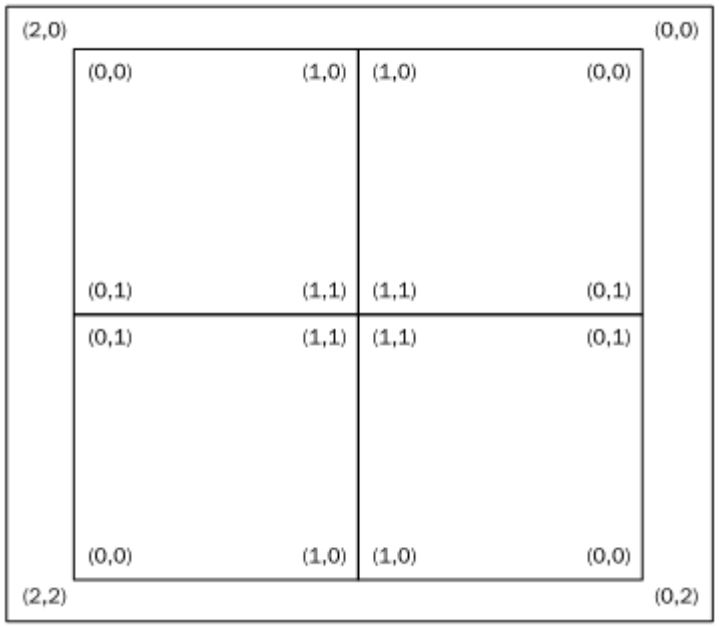
2. Внутри блока текстур, добавьте строку, которая говорит, что нужно использовать CLAMP:
tex_address_mode clamp
3. Изменим создание квадрата:
manual->begin("MyMaterial4", RenderOperation::OT_TRIANGLE_LIST);
4. Скомпилируйте и запустите приложение. Вы должны увидеть текстуру камня в правом верхнем углу экрана. Три другие части будут отображаться различными зветами.

Что мы сделали.
Мы изменили алгоритм отображения текстуры в режим CLAMP. Этот режим использует границы пикселов текстуры для заполнения, всех координат текстур, которые больше 1.

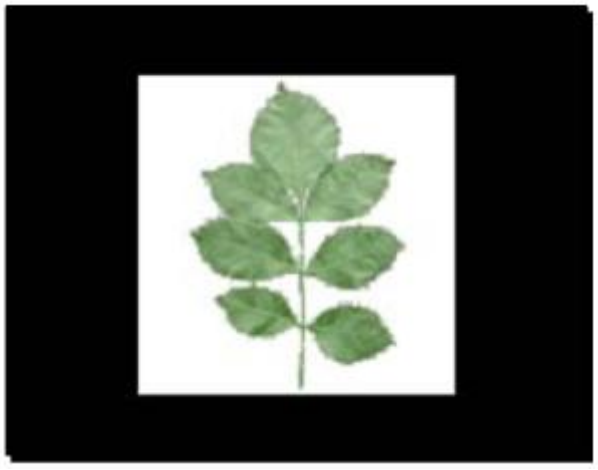
Использование зеркального режима.
1. Создадим новый материал MyMaterial5, используя предыдущий в качестве шаблона.
2. Изменим режим отображеня на mirror:
tex_address_mode mirror
3. Изменим изображение текстуры на листья:
texture leaf.png
4. Скомпилируем и запустим:



Что мы сделали.
Мы снова изменили алгоритм отображения текстуры, на этот раз в зеркальный режим. Mirror представляет собой простой, но эффективный режим. Используется для текстурирования больших площадей, как каменная стена. Каждый раз, когда текстурная координата больше 1, она переворачивается, а затем используется как режим WRAP. Взгляните на диаграмму:



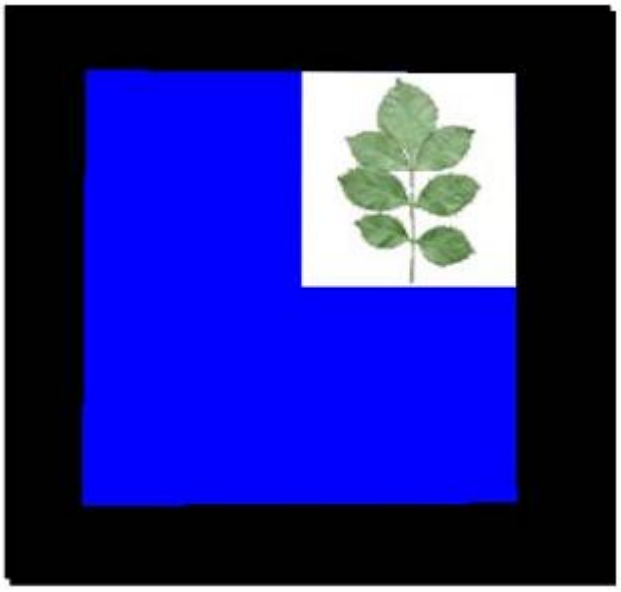
Режим Border.
1. Создадим новый материал MyMaterial6, используя предыдущий в качестве шаблона.
2. Изменим алгоритм отображения на boder:
tex_address_mode border
3. Изменим у куба используемый материал:
manual->begin("MyMaterial6", RenderOperation::OT_TRIANGLE_LIST);
4. Скомпилируем и запустим, увидим одно растение:



Что мы сделали.
Куда же делись другие листья? Режим границы (Border) не создает копий нашего изображения. Когда координаты текстуры больше, чем 1, то эти координаты окраживаются в цвет границы по умолчанию, очевидно, что цветом по умолчанию является черный, а черный рассматривается как нулевое значение цвета.

Изменение цвета границы.
Если бы мы могли использовать только черный цвет границы, то эта функция былабы бесполезной. Давайте посморим как изменить цвет границы.

1. Скопируйте предыдущий материал и назовите его MyMaterial7.
2. После настройки режима текстурирования добавьте следующую строку, что бы установить цвет границы в синий:
tex_border_color 0.0 0.0 1.0
3. Скомпилируйте и запустите приложение. На этот раз мы также видим только один экземпляр текстуры, но остальные части квадрата закрашены в синий цвет:



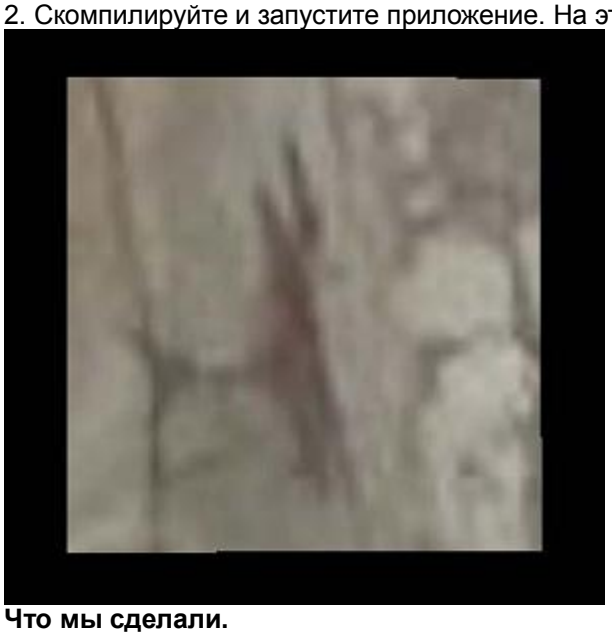
Что мы сделали.
Мы изменили цвет границы с черного на синий. Кроме того мы можем использовать любой цвет, который может быть описан значением RGB. Эта текстура может быть использована при выводе логотипов на таких объектах как гоночная машина. Нам нужно только установить цвет границы в цвет автомобиля, а затем добавить текстуру.

Прокрутка текстуры.
Мы видели несколько режимов текстурирования, но это только один атрибут файла материала. Теперь мы испробуем другой атрибут, который может быть весьма полезным.
На этот раз изменим наш квадрат, что бы увидеть эффект нового материала.
1. Изменим используемый материал на MyMaterial8, а также изменим текстурные координаты с 2, до 0.2:
```
manual->begin("MyMaterial8", RenderOperation::OT_TRIANGLE_LIST);
manual->position(5.0, 0.0, 0.0);
manual->textureCoord(0.0,0.2);
manual->position(-5.0, 10.0, 0.0);
manual->textureCoord(0.2,0.0);
manual->position(-5.0, 0.0, 0.0);
manual->textureCoord(0.2,0.2);
manual->position(5.0, 10.0, 0.0);
manual->textureCoord(0.0,0.0);
```
2. Теперь создайте новый материал. На этот раз не будем использовать режимы, и используем изображение terr_rock6.jpg:
```
material MyMaterial8
{
   technique
   {
      pass
      {
         texture_unit
         {
            texture terr_rock6.jpg
         }
      }
   }
```

}
3. Скомпилируйте и запустите приложение. Вы увидети часть каменной текстуры, которую видели ранее:

Что мы сделали.
Мы видим только часть текстуры, потому что наш квадрат использует текстурные координаты только до 0.2, это означает, что четыре пятых текстуры не отображается на нашем квадрате.

Теперь, когда мы подготовили наш квадрат, давайте включим прокрутку текстуры.
1. Добавьте следующую строку в структура текстурного блока:
scroll 0.8 0.8
2. Скомпилируйте и запустите приложение. На этот раз вы увидите другую часть текстуры.



Что мы сделали.
Атрибут scroll меняет координаты текстуры с заданным смещением. На следующей диаграмме показано влияние прокрутки.

Этот атрибут может быть использован для изменения координат текстуры, без изменения UV координаты самой модели.

Анимированая прокрутка текстуры.
То что мы сделали не совсем захватывает дух. Давайте добавим динамичности прокрутке.
1. Измените строку scroll в материале:
scroll_anim 0.01 0.01
2. Скомпилируйте и запустите. Если вы посмотрите внимательно, вы увидите как текстура плавно переходит с левого верхнего в нижний правый угол.

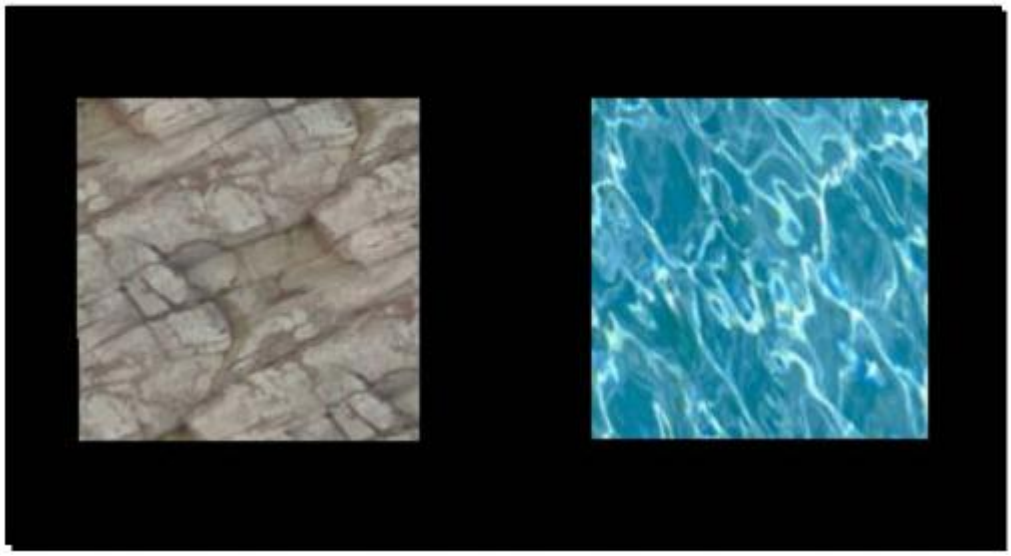
Что мы сделали.
Мы использовали другой атрибут, чтобы сделать текстуру движущейся. Этот атрибут аналогичен атрибуту scroll, только смещение указывается, изменением положения в секунду.
Другие атрибуты можете посмотреть в разделе WIKI на официальном сайте. http://www.ogre3d.org/docs/manual/manual_17.html#SEC9

Наследование материалов.
Прежде чем коснуться более сложной темы, как шейдеры, мы постараемся изучить наследование материалов.
Мы создадим два новых материала и один новый квадрат.
1. Для этого примера нам нужен один квадрат, который просто отображает одну текстуру:
```
manual->begin("MyMaterial11", RenderOperation::OT_TRIANGLE_LIST);
manual->position(5.0, 0.0, 0.0);
manual->textureCoord(0.0,1.0);
manual->position(-5.0, 10.0, 0.0);
manual->textureCoord(1.0,0.0);
manual->position(-5.0, 0.0, 0.0);
manual->textureCoord(1.0,1.0);
manual->position(5.0, 10.0, 0.0);
manual->textureCoord(0.0,0.0);
manual->index(0);
manual->index(1);
manual->index(2);
manual->index(0);
manual->index(3);
manual->index(1);
manual->end();
```
2. Новый материал будет использовать текстуру камня и использовать атрибут rotate_anim, который вращает текстуру с заданной скоростью. Но самое главное, чтобы название текстурного блока было texture1:
```
material MyMaterial11
{
   technique
   {
      pass
      {
         texture_unit texture1
         {
          texture terr_rock6.jpg
          rotate_anim 0.1
         }
      }
   }
}
```
3. Теперь давайте создадим еще один квадрат и переместите его на 15 единиц по оси X, что бы он не пересекался с первым квадратом. Кроме того используйте setMaterialName(), что бы изменить используемый материал на MyMaterial12:
```
ent = mSceneMgr->createEntity("Quad");
ent->setMaterialName("MyMaterial12");
node =  mSceneMgr->getRootSceneNode()-createChildSceneNode("Node2",Ogre::Vector3(15,0,0));
node->attachObject(ent);
```

4. Последнее, что нужно сделать, это создать MyMaterial12. Мы будем наследовать MyMaterial11 и установим другую тестуру, которую хотим использовать.

```
material MyMaterial12 : MyMaterial11
{
  set_texture_alias texture1 Water02.jpg
}
```

5. Скомпилируйте и запустите приложение, и вы должны увидеть два квадрата с вращающимися текстурами, одна камень, другая вода.



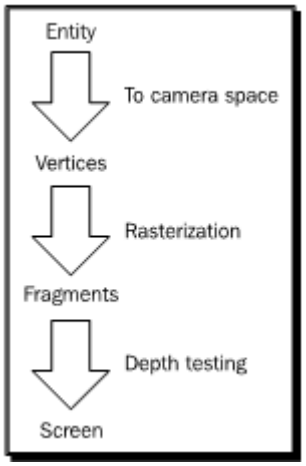
Что мы сделали.

Мы создали два квадрата, каждый со своим собственным материалом. В шаге 2, мы создали наш материал для квадрата и использовали новый атрибут rotate_anim x, который вращает текстуру x оборотов в секунду, ничего сложного. Кроме того, мы дали текстурному блоку имя texture1. Нам понадобилось это имя позже. На 3 шаге, мы создали еще один квадрат и изменили материал командой setMaterialName(). Важной частью был шаг 4, где мы создали новый материал путем наследования. Это понятие должно быть вам знакомо. Синтаксис такой же как и в C++, Новое_имя : Имя_родителя. В нашем случае MyMaterial12 наследуется от MyMaterial11. Затем мы использем атрибут set_texture_alias, который связывает изображение water02.jpg с текстурой texture1. В этом случае мы заменили terr_rock6.jpg на water01.jpg. Поскольку это едиственное изменение, которое мы хотели сделать, можем остановиться на достигнутом.

Использование псевдонимов в текстурах позволяет создавать нам много материалов, которые отличаются только изображением, без необходимоти писать каждый материал от начала и до конца, и мы все знаем, что дублирования следует избегать, если это возможно. Мы рассмотрели много вещей о материалах, но это далеко не все возможности. На официальном WIKI можете ознакомиться более подробно.

Использование шейдеров.
В этой главе мы использовали функции фиксированного конвейера FFP. Фиксированный конвейер не предоставляет больших возможностей разработчику. Мы всего лишь можем изменить некоторые заложеные ранее параметры. Вот где шейдеры могут помочь. Шейдеры это небольшие программы, которые могут быть загружены в видеокарту, а затем функционировать как часть процесса визуализации. Шейдеры можно рассматривать как маленькие программы написанные на C подобном языке с небольшим, но мощным набором функций. С шейдерами мы можем почти полностью контролировать то как наши объекты будут визуализироваться, а так же добавлять различные эффекты, которые не были возможны с помошью FFP.

Конвейер визуализации.
Что бы понять шейдеры, мы сначала должны понять как работает процесс визуализации вцелом. При визуализации, каждая вершина нашей модели переводится с локального пространства в пространство камеры, а каждый треугольник растеризуется. Это означает, что видеокарта вычисляет, как представить изображение модели. Эти части изображения называются фрагментами. Каждый фрагмент затем обрабатывается и управляется. Мы могли бы применить определенную часть текстуры на этот фрагмент в нашей модели, или просто назначить модели один цвет. Ниже приводится упрощенный график этого конвейера:



Почти каждое новое поколение графических карт вводит новые типы шейдеров. Задача вершинных шейдеров заключается в преобразовании вершин в пространство камеры, и при необходимости изменить их, таким образом можно выполнять анимацию на GPU. Пиксельный шейдер получает растровый фрагмент и можно наложить текстуру, или манипулировать различными способами, например для освещения модели с точностью до пикселя. Есть и другие этапы шейдеров, такие как геометрический шейдер, но мы не будем обсуждать их в этой книге, потому что они довольно новые, не очень широко поддерживаются, а также находятся за пределами этой книги.

Давайте напишем наш первый вершинный и фрагментный шейдер.
1. В нашем приложении нужно изменить только используемый материал. Измените его на MyMaterial13. Так же удалите второй четырехугольник:
manual->begin("MyMaterial13", RenderOperation::OT_TRIANGLE_LIST);
2. Теперь нам нужно создить этот файл материала. Во первых мы определим фрагментный шейдер. Ogre 3D нужно пять штук информации о шейдерах:

- Имя шейдера
- На каком языке написано
- В каком файле сохраняется
- Какая основная функция шейдера
- В какой профиль мы хотим компилировать

3. Всю эту информацию опишем в файле материала

```
fragment_program MyFragmentShader1 cg
{
  source Ogre3DBeginnersGuideShaders.cg
  entry_point MyFragmentShader1
  profiles ps_1_1  arbfp1
}
```

4. Вершинные шейдеры требуют тех же параметров, но мы их также должны определить. А так же определим параметр по умолчанию.

```
vertex_program MyVertexShader1 cg
 {
  source Ogre3DBeginnerGuideShaders.cg
  entry_point MyVertexShader1
  profiles vs_1_1 arbvp1
  default_params
  {
   param_named_auto worldViewMatrix worldviewproj_matrix
  }
}
```

5. Сам материал использует только имена шейдеров для ссылки на них.

```
material MyMaterial13
{
   technique
   {
      pass
      {
         vertex_program_ref MyVertexShader1
         {
         }
         fragment_program_ref MyFragmentShader1
         {
         }
      }
   }
}
```

6. Теперь нужно написать сами шейдеры. Создайте файл с именем Ogre3DBeginnersGuideShaders.cg в каталоге materials\programs из каталога Ogre 3D SDK.

7. Каждый шейдер сосотит из функции. Одно из отличий, мы можем использовать ключевые слова параметров как входные значения. Выходные значения используются для рендеринга следующего шага. Параметры вершинных шейдеров обрабатываются, а затем передаются в пиксельный в качестве входных параметров. Выходные параметры пиксельного шейдера используются для создания конечного результата визуализации. Не забывайте использовать правильное имя в качестве точки входа, оначе Ogre 3D ее не найдет. Давайте начнем с фрагментного шейдера, потому что он проще:

```
void MyFragmentShader1(out float4 color: COLOR)
```

8. Фрагментный шейдер будет возвращать синий цвет для каждого пикселя:

```
{
   color = float4(0,0,1,0);
}
```

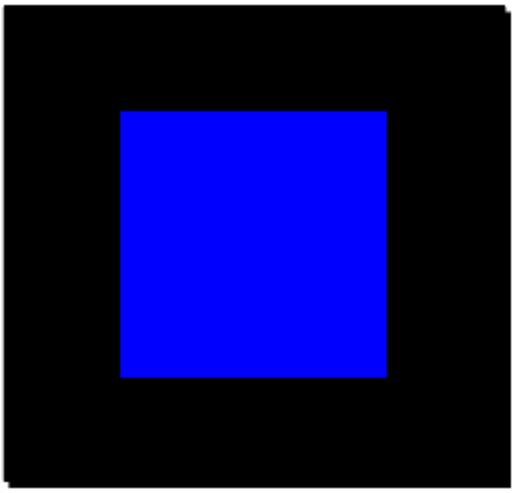
9. Вот и все. Теперь мы подходим к вершинному шейдеру. Вершинный шейдер имеет 3 параметра, положение вершин, переведенное положение вершин, и какую матрицу мы используем для перевода:

```
void MyVertexShader1(
      float4 position       : POSITION,
      out float4 oPosition    : POSITION,
      uniform float4x4 worldViewMatrix)
```

10. Внутри шейдера, мы используем матрицы и входящее положение для расхода исходящей позиции:

```
{
   oPosition = mul(worldViewMatrix, position);
}
```

11. Скомпилируем и запустим. Вы должны увидеть наш квадрат, который на этот раз синий.



Что мы сделали.

Немало вещей здесь произошло, мы начнем с шага 2. Здесь мы определили фрагментный шейдер, который собираемся использовать. Как и говорилось ранее, Ogre3D нужно 5 кусков информации о шейдере. Определяется фрагментный шейдер ключевым словом fragment_program. Изначально шейдеры писались на Assembler, потому что небыло другого языка который можно использовать. Затем вскоре, что бы облегчить старания программистов был разработан язык высокого уровня. Сегодня есть 3 различных языка шейдеров: HLSL, GLSL и CG. HLSL используется в DirectX, а GLSL используется в OpenGL. CG был разработан NVIDIA в сотрудничестве с Microsoft и является языком, который мы собираемся использовать. Этот язык во время запуска компилируется в ассемблер. CG умеет составлять ассемблерный код как для HLSL, так и для GLSL. Поэтому мы и используем его. Остальные три параметра записываются в фигурных скобках. Мы указываем файл шейдера **source**, мы не должны указывать полный путь к файлу шейдера, потому что Ogre 3D сканирует файлы при запуске программы. Еще один параметр это название функции, которая будет являться точкой входа **entry_point**. В коде файла мы создали MyFragmentShader1, именно это имя мы и даем в качестве параметра. Это означает, что каждый раз первой будет вызываться эта функция. Функция имеет только один выходной параметр – цвет float4. Префикс out означает, что это выходной параметр. Float4 это просто тип, который хранит в себе 4 значения с плавающей точкой. Для цветов это RGBA. После этого мы написали COLOR. В CG это называется семантическое описание того, что параметр используется для описания цвета.

Последняя часть информации, которую мы передаем, использует ключевое слово **profiles** со значениями ps_1_1 и arbfp1. Что бы понять это, нан нужно окунуться в историю шейдеров. С каждым новым поколение видеокарт, появлялись новые версии шейдеров. Это начиналось, как простой C подобный язык программирования, при условии что сейчас действительно сложные и мощные системы. Сейчас существует несколько различных версий шейдеров, и каждый с уникальным набором функций. Ogre 3D необходимо знать какие из этих версий мы собираемся использовать. Ps_1_1 означает версию пиксельных шейдеров 1.1 и arbfp1 – средство фрагментной программы версии 1. Нам нужны оба профиля, потому что ps_1_1 это DirectX, а arbfp1 это OpenGL. Хоть мы и говорим о кроссплатформенности, но иногда нам приходится определять значения для обоих платформ. Все множества можно найти по адресу http://www.ogre3d.org/docs/manual/manual_18.html.

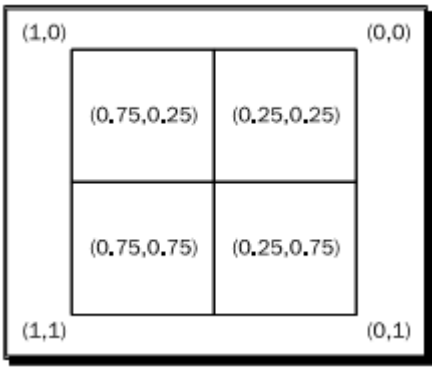
Это все, что необходимо для определения фрагментного шейдера в нашем файле материала. На 3 шаге, мы определили наш вершинный шейдер. Эта часть кода похожа на определение фрагментного шейдера. Главное отличие является в блоке default_params. Этот блок определяет параметры, которые даются во время выполнения шейдеров. param_named_auto определяет параметр, который автоматически передается в шейдер из Ogre 3D. После этого мы должны дать параметру имя и значение которое он должен принять. Мы назвали параметр worldViewMatrix, любое другое имя тоже будет работать. Ценность представляет слово **worldviewproj_matrix.** Это ключевое слово говорит Ogre 3D, что нужно использовать матрицу WorldViewProjection. Эта матрица используется для преобразования локальных координат, в координаты камеры. Список всех значений ключевых слов можно найти на http://www.ogre3d.org/docs/manual/manual_23.html#SEC128. Как использовать эти значения мы рассмотрим в ближайшее время. На шаге 4 мы как всегда мы определили наш материал с одной техникой и одним проходом. Мы не определяем текстурные блоки, но используем ключевое слово **vertex_program_ref**. После этого мы должны написать название вершинной программы, в нашем случае это MyVertexShader1. Если бы мы захотели, мы могли бы положить в фигурные скобки еще несколько параметров, но нам это не нужно по этому мы открываем и сразу закрываем фигурные скобки. Тоже самое и для **fragment_program_ref**.

Написание шейдеров.
Теперь, когда мы определили все необходимые вещи в файле материала, давайте напишем сам код шейдера. На 6 шаге мы определаем главную функцию с параметром, который мы обсуждали ранее. На 7 шаге определяется функция тела, для этого шейдера код очень прост. Мы создали переменную типа float4 для описания цвета (0,0,1,0), синий цвет, и присваиваем цвет к параметру color. Фишка в том, что все, что связано с этим материалом будет окрашено в синий цвет. Давайте рассмотрим вершинный шейдер. На 8 шаге определена функция заголовка. Наш вершинный шейдер имеет три параметра, два из них помечены как POSITION, используя семантики CG, и матрицу как float4x4 worldViewMatrix. Перед определением параметра матрицы есть ключевое слово uniform.
Каждый раз, когда наш шейдер вызывается, ему передается новая вершина в качестве входного параметра POSITION. Вычисляется положение новой вершины, и сохраняется в выходной параметр POSITION. Это не относится к worldViewMatrix. Ключевое слово uniform обозначает, что параметр является постоянным в пределах одного кадра, и не меняется с каждым вызовом, в отличии от POSITION. На 9 шаге описывается тело вершинного шейдера. Нам нужно умножить положение вершины на матрицу, что бы перевести данные в пространство камеры. Далее эти новые значения сохраняются в выходной параметр POSITION.

Текстурирование с шейдерами.
Мы нарисовали четырехугольник синим цветом, но мы хотели бы использовать текстуру.
1. Создадим новый материал с именем MyMaterial14. Кроме этого, нужно создать два новых шейдера с именами MyFragmentShader2 и MyVertexShader2. Добавьте объявление шейдера в файле материала и добавьте текстуру в материал.
```
texture_unit
{
  texture terr_rock6.jpg
}
```
2. Нам нужно добавить два новых параметра для нашего фрагментного шейдера. Первый это текстурные координаты. Здесь мы опять используем семантику CG. Отметим первый параметр как TEXCOORD0. Следующий параметр типа sampler2D, с атрибутом uniform, т.к. текстура не меняется за один кадр, и означает что параметр назначается из вне программы CG.
```
void MyFragmentShader2(float2 uv       : TEXCOORD0,
        out float4 color : COLOR,
        uniform sampler2D texture)
```
3. В фрагментном шейдере заменим создание цвета следубщей строкой:
```
color = tex2D(texture, uv);
```
4. Вершинный шейдер также нуждается в новых параметрах. Один float2 для входящих текстурных координат, и один float2 для исходящих текстурных координат, оба семантикой TEXCOORD0.
```
void MyVertexShader2(
    float4 position       : POSITION,
    out float4 oPosition    : POSITION,
    float2 uv       : TEXCOORD0,
    out float2 oUv       : TEXCOORD0,
    uniform float4x4 worldViewMatrix)
```
5. В теле также высчитывается положение точки на камере:
```
oPosition = mul(worldViewMatrix, position);
```
6. Выходные текстурные координаты такие же как и входные:
```
oUv = uv;
```
7. Не забудьте изменить используемый материал в коде программы, затем скомпилируйте и запустите приложение. Вы увидите текстурированный четырехугольник.

Что мы сделали
На первом шаге мы добавили текстурный блок с текстурой горы. На втором шаге добавили float2 для сохранения текстурных координат, также мы использовали sampler2D. Sampler2D это просто название для функции поиска текстур, и поэтому она не меняется и приходит извне программы CG, мы объявили ее как uniform. На третьем шаге мы использовали функцию tex2D, хоторая принимает в качестве параметров sampler2D и float2, а возвращает цвет float4. Эта функция использует float2 в качестве текстурных координат для получения цвета из sampler2D. На 4 шаге мы добавили две переменные текстурных координат в вершинный шейдер, один как входящий, другой как исходящий. На пятом шаге присвойли входящий параметр в исходящий.

Что происходит при визуализации.
Наш вершинный шейдер получает каждую вершину и преобразует его в пространство камеры. После того как все вершины треугольника прошли этот этап, происходит растеризация. В ходе этого процесса треугольник разбит на фрагменты. Каждый фрагмент является координатой на экране. Если пиксель не перекрывается другим объектом, то пиксель закрасится нужным цветом. После этого процесса, каждый фрагмент имеет свою координату текстуры, и мы используем ее, что бы задать цвет пикселя из текстуры. На следующем рисунке приведен пример четырехугольника, который поделен на 4 фрагмента. Каждый фрагмент имеет свои координаты текстуры. Это также показывает, как мы можем представлять текстурные координаты, связаные с пикселями.



Сочетание цветов и текстурных координат.
Создайте новый вершинный и фрагментный шейдер под названием MyVertexShader3 и MyFragmentShader3 соответственно. Фрагментный шейдер должен сделать все в зеленом тоне, а вершинный должен вычислить позиции вершин в пространстве камеры и просто передать координаты текстуры в фрагментный шейдер.

Интерполяция значений цветов.
Что бы лучше увидеть эффект интерполяции, давайте заменим текстуры цветами.
Изменим немного наш код.
1. Настроим новый материал.
2. Единственное что нам нужно изменить в отличие от предыдущего, это удалим текстуру.
3. В коде изменим текстурные координаты на цвет

```
manual->position(5.0, 0.0, 0.0);
manual->color(0,0,1);
manual->position(-5.0, 10.0, 0.0);
manual->color(0,1,0);
manual->position(-5.0, 0.0, 0.0);
manual->color(0,1,0);
manual->position(5.0, 10.0, 0.0);
manual->color(0,0,1);
```

4. Вершинный шейдер также нуждается в некоторой корректировке.

```
void MyVertexShader4(
      float4 position       : POSITION,
      out float4 oPosition    : POSITION,
   float4 color    :COLOR,
   out float4 ocolor    :COLOR,
      uniform float4x4 worldViewMatrix)
{
   oPosition = mul(worldViewMatrix, position);
  ocolor = color;
}
```

5. Фрагментный шейдер теперь имеет два цветовых параметра, один входящий, другой исходящий.

```
void MyFragmentShader4(  float4 color : COLOR,
         out float4 oColor : COLOR)

{
   oColor = color;
}
```

6. Скомпилируйте и запустите приложение. Вы должны увидеть четырехугольник, с правой стороны голубой, а с левой зеленый, они должны плавно переходить один в другой.

Что же произошло.

На 3 шаге, мы увидели другой manualObject, а именно, добавили цвета к вершине, с помощью трех значений r,g,b. На 4 шаге заменили текстурные координаты параметрами цвета. То же самое относится и к шагу 5. Это не очень сложно и не интересно, но оно показывает как работает интерполяция. Это дает лучшее понимание того как вместе работают вершины и шейдерная программа

Замена четырехугольника моделью.

Стало немного скучно использовать в качестве примера квадратик, так что давайте поэксперементируем с Синбадом.

С помощью следующего кода мы будем использовать Синбада вместо четырехугольника:

1. Удалите весь код связаный с четырехугольником, а создание узла сцены не трогайте.
2. Создайте экземпляр sinbad.mesh и прикрепите его к узлу сцены, а также установите ему материал MyMaterial14:

```
void createScene()
{
  Ogre::SceneNode* node = mSceneMgr->getRootSceneNode()->createChildSceneNode("Node1");
  Ogre::Entity* ent = mSceneMgr->createEntity("Entity1","Sinbad.mesh");
  ent->setMaterialName("MyMaterial14");
  node->attachObject(ent);
}
```

3. Скомпилируйте и запустите приложение и вы увидите, что Синбад покрывается текстурой горной поверхности.

Что мы сделали.
Все, что мы сейчас наделали должно быть вам знакомо. Мы создали экземпляр модели, прикрепили ее к узлу сцены и изменили ее материал на MyMaterial14.

Сделаем эффект пульсирования модели с помощью шейдера.
До этого момента мы работали только с фрагментным шейдером. Теперь попробуем поработать с вершинными шейдерами.
Добавить импульс для нашей модели очень легко, нужно только изменить немного существующий код.
1. На этот раз нам нужно изменить только вершинный шейдер, а фрагментный шейдер будем использовать старый. Создайте новый вершинный шейдер с именем MyVertexShader5 и используйте его в новом файле материала с именем MyMaterial17, а фрагментный шейдер используйте MyFragmentShader2, так как этот шейдер будет просто текстурировать модель.

```
material MyMaterial17
{
  technique
  {
     pass
     {
    vertex_program_ref MyVertexShader5
         {
         }
       fragment_program_ref MyFragmentShader2
         {
         }

    texture_unit
         {
            texture terr_rock6.jpg
         }
     }
  }
}
```

2. Новый вершинный шейдер выглядит также как тот, который был раньше, только в этот мы добавим новый параметр в блок default_params под название pulseTime который будет получать значение time.

```
vertex_program MyVertexShader5 cg
 {
  source Ogre3DBeginnerGuideShaders.cg
  entry_point MyVertexShader5
  profiles vs_1_1 arbvp1
  default_params
  {
    param_named_auto worldViewMatrix worldviewproj_matrix
    param_named_auto pulseTime time
  }
}
```

3. В самом приложении нам менять ничего не нужно. Единственное, что осталось сделать, это создать новый вершинный шейдер. MyVertexShader5 основан на MyVertexShader3. Просто добавьте новую строку, которая будет изменять значение X переменной oPosition:

```
void MyVertexShader5( uniform float pulseTime,
        float4 position        : POSITION,
        out float4 oPosition    : POSITION,
    float2 uv       : TEXCOORD0,
    out float2 oUv       : TEXCOORD0,
        uniform float4x4 worldViewMatrix)
{
    oPosition = mul(worldViewMatrix, position);
  oPosition.x *= (2+sin(pulseTime));
  oUv = uv;
}
```

4. Скомпилируйте и запустите приложение. Вы должны увидеть синбада пульсирующего по оси X.

Что мы сделали.

Мы сделали пульсирующую модель по оси X. Нам нужен второй параметр для вершинного шейдера, который будет содержать время. Мы использовали синус времени к значению которого мы прибавили два, для того чтобы получить значение между 1 и 3, на это значение мы умножали текущее значение X вершины в каждом кадре. Этим методом можно добиться различных эффектов в играх.

Подведем итоги.

Мы многому научились в этой главе, а именно:

- Как создавать новые материалы
- Как накладывать текстуры на объект с использованием материалов
- Как создавать шейдер и ссылаться на него из файла материала
- Как работать с геометрией используя вершинный шейдер

В следующей главе мы будем научимся использовать пост-эффекты для улучшения качества изображения на экране, и научимся создавать совершенно новый визуальный стиль.

ГЛАВА 8
КОМПОЗИТОРЫ ПОСТ-ОБРАБОТКИ

В этой главе мы собираемся добавить композитор для улучшения качества изображения. Эта глава расскажет вам как создавать эффекты и объединять их вместе для создания новых интересных эффектов.
В этой главе мы будем:

- Создавать скрипт композитора и применим к нашей сцене
- Работать с экраном для разбиения на две части (split screen)
- Использовать клавиатуру для регулирования настроек шейдера

Итак, продолжим…

Подготовка сцены
Мы собираемся использовать композитор, но перед этим нам нужно подготовить сцену над которой мы будем эксперементировать.
Мы будем использовать пример из предыдущей главы.
1. Удалите строку, которая изменяет материал модели. Нам нужно использовать исходный материал:
ent->setMaterial(Ogre::MaterialManager::getSingleton().
getByName("MyMaterial18"));
2. Класс приложения должен выглядеть следующим образом:

```
class Example69 : public ExampleApplication
{
private:
public:
  void createScene()
  {
    Ogre::SceneNode* node =
 mSceneMgr->getRootSceneNode()->createChildSceneNode("Node1",Ogre::Vector3(0,0,450));
    Ogre::Entity* ent = mSceneMgr->createEntity("Entity1","Sinbad.mesh");
    node->attachObject(ent);
  }
};
```

3. Скомпилируйте и запустите. Вы увидите синбада с нормальной его текстурой

Что мы сделали.
Мы создали простую сцену на которой мы будем наблюдать эффекты композитора.

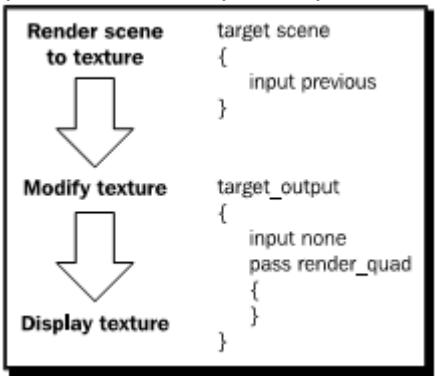
Добавление первого композитора.
Прежде чем объяснять, что такое композитор, давайте используем его, а затем обсудим технические подробности.
1. Нам нужен новый материал, который ничего не делает. Назовем новый материал Ogre3DBeginnersGuide/Comp1

```
material Ogre3DBeginnersGuide/Comp1
{
  technique
   {
      pass
      {
     texture_unit
         {
         }
      }
   }
}
```
2. Далее нужно создать файл для хранения скриптов композитора. В том же каталоге с материалом создайте файл Ogre3DBeginnersGuide.compositor
3. Этот файл определяет наш композитор по той же схеме как мы делали для материалов.
```
compositor Compositor1
{
   technique
   {
```
4. Далее нужно определить цель, нашу сцену, прежде, чем мы сможем вносить изменения:
```
texture scene target_width target_height PF_R8G8B8
```
5. Далее нужно определить содержание цели, в нашем случае это ссцена, которая только что отрисовалась:
```
target scene
      {
         input previous
      }
```
6. Последним шагом в композиторе, является описание сценария выходных данных:
```
target_output
      {
```
7. Композитор выводит результат на четырехугольник, который покрывает весь экран. Этот четырехугольник должен использовать материал Ogre3DBeginnersGuide/Comp1 и вводная текстура имеет изображение нашей сцены.
```
input none
pass render_quad
{
material Ogre3DBeginnersGuide/Comp1
input 0 scene
}
```
8. Закончили описание композитора. Закроем все фигурные скобки
```
      }
   }
}
```
9. Теперь у нас есть готовый сценарий композитора, давайте применим его в нашей сцене. Для этого мы используем compositorManager и окно нашей камеры. Добавим некоторый код в createScene():
```
Ogre::CompositorManager::getSingleton().addCompositor(mCamera->getViewport(), "Compositor1");
Ogre::CompositorManager::getSingleton().setCompositorEnabled(mCamera->getViewport(),  "Compositor1", true);
```
10. Скомпилируйте и запустите приложение. Вы увидите туже сцену, что и раньше.

Что мы сделали.

Мы добавили наш первый композитор используя файл композитора, содержащий сценарий. На 1 шаге мы создали пустой материал, который показывает все без добавления эффекта. На 2 шаге мы создали файл, который хранит в себе скрипт композитора. 3 шаг, нам довольно знаком, мы назвали наш композитор Compositior1 и определили technique, которые могут использоваться для различных графических карт так же как и в скриптах материалов. Самое интересное начинается с 4 шага, здесь мы создали новую текстуру с именем scene, которая имеет размеры экрана и 8 бит на пиксел.

Как работает композитор.
Зачем нам создавать новую текстуру? Чтобы понять это, нам нужно сперва понять как работает композитор. Композитор изменяет внешний вид сцены после того как она была нарисована. Это похоже на пост-обработку в кино, где они добавляют все компьютерные эффекты после съемок фильма. Для того, что бы это сделать, нужно отобразить сцену в качестве текстуры, которая в дальнейшем может быть изменена. Мы создали эту текстуру на 4 шаге, а на 5 шаге мы сказали Ogre3D чем мы ее будем заполнять, а именно сценой, которая нарисовалась. Теперь у нас есть текстура с изображением нашей сцены. Следующий шаг что нужно сделать – это создать новое изображение, которое будет отображаться на композиторе. Это использовано в шаге 6 *target_output*. На 7 шаге определается вывод. Что бы сделать изменение текстуры сцены, нам нужен четырехугольник, который покрывает весь экран. Это достигается путем использования идентификатора render_quad. Есть несколько других идентификаторов, их можно найти на сайте http://www.ogre3d.org/docs/manual/manual_32.html#SEC163. В нутри блока прохода, определяется несколько атрибутов, которые нам нужно использовать. Первое это материал который мы хотим испрользовать. Материал мы определили заранее, он ничего не делает, поэтому сцена показывается без изменений. Следующий атрибут указывает какую текстуру использовать в качестве данных для экрана. Мы не увидим изменений, пока мы не добавим изменяющий код в материал. На 9 шаге мы добавляем композитор в коде приложения, нужно указать какая камера используется и какое окно.
Ниже приводится диаграмма, которая показывает процесс работы композитора:



Изменение текстуры.
Мы отобразили нашу сцену в текстуру и показали ее без изменений, это бессмысленно, нужно добавить спецэффекты.
Для изменения текстуры, мы будем использовать фрагментный шейдер. Нужно изменить материал для использования фрагментного шейдера. Скопируйте материал и файл композитора и задайте имена Compositior2 и Ogre3DBeginnersGuide/Comp2.
1. *fragment_program_ref MyFragmentShader5*
{
}
2. Не забудьте определить программу перед материалом.
```
fragment_program MyFragmentShader5 cg
{
  source Ogre3DBeginnersGuideShaders.cg
  entry_point MyFragmentShader5
  profiles ps_1_1  arbfp1
}
```
3. Помимо этого создадим фрагментный шейдер. Этот шейдер принимает в качестве входных данных, координаты текстуры и текстуру.
```
void MyFragmentShader5(float2 uv : TEXCOORD0,
        out float4 color : COLOR,
        uniform sampler2D texture)
{
```
4. Получим цвет цвет из тексутры в положении текстурных координат
```
float4 temp_color = tex2D(texture, uv);
```
5. Сконвертируем цвет в чернобелый
```
float greyvalue = temp_color.r * 0.3 + temp_color.g * 0.59 + temp_
color.b * 0.11;
```

6. Используем этот цвет в выходном значении:
color = float4(greyvalue,greyvalue,greyvalue,0);
7. Скомпилируем и запустим приложение, вы должны увидеть чернобелое изображение

Что мы сделали.
Мы добавили фрагментный шейдер для нашего композитора. После отображения сцены в текстуру, она была передана в шейдер. Фрагментный шейдер запрашивает цвет текстуры исходя из текстурных координат, затем он преобразует цвета в черно-белый цвет, просто умножая цвета на коэффициенты. Для красного цвета 0.3, для зеленого 0.59, и для синего 0,11. После этого эффекта, попробуем создать эффект инвертирования цветов.

Инвертирование изображения.
С помощью другого композитора создадим эффект инвертирования цветов.
Используя код черно-белого композитора, создадим другой, который инвертирует цвета.
1. Скопируйте шейдер, материал и композитор, потому что позже мы их будем комбинировать. Задайте им новые имена MyFragmentShader6, Ogre3DBeginnersGuide/Comp3 и композитор Compositor3.
2. На этот раз получим цвета и вычтем каждый компонент цвета из единицы, тем самым получив перевернутое значение.
color = float4(1.0 - temp_color.r,1.0 - temp_color.g, 1.0 - temp_color.b,0);
3. Скомпилируйте и запустите приложение. Вы увидите белый фон и Синбада, отображаемого странными цветами.

Что мы сделали.
Мы изменили фрагментный шейдер так, что бы он инвертировал цвета, а не преобразовывал их в черно-белый. Больше ничего не поменялось. Преобразование RGB цвета очень легко, нужно просто вычесть каждый компонент цвета из единица.

Объединение композиторов

Для объединения двух композиторов, мы должны создать новый.

1. Для нового композитора нам нужны две текстуры, одна для хранения изображения сцены, вторая для хранения промежуточного результата

```
compositor Compositor4
{
  technique
  {
    texture scene target_width target_height PF_R8G8B8
    texture temp target_width target_height PF_R8G8B8
```

2. Заполните текстуру сццены так как это делалось ранее, а потом заполните промежуточную текстуру чернобелым материалом

```
target scene
    {
      input previous
    }
    target temp
  {
      pass render_quad
      {
        material Ogre3DBeginnersGuide/Comp2
        input 0 scene
      }
    }
```

3. Затем спомощью временной текстуры обратим ее цвета и используем для вывода

```
target_output
      {
        input none
        pass render_quad
        {
          material Ogre3DBeginnersGuide/Comp3
          input 0 temp
        }
      }
 }
}
```

4. Скомпилируйте и запустите. Вы увидите инвертированное чернобелое изображение.

Что мы сделали.

Мы создали вторую вспомогательную текстуру, которая была использована целью рисования черно-белого материала, а затем эта текстура с черно-белым изображением была использована для вывода инвертированого изображения на экран.

Уменьшение количества текстур.

В предыдущем примере мы использовали две текстуру, одну для оригинальной сцены, другую для хранения промежуточного результата. Давайте попробуем использовать только одну текстуру.

С помощью следующего кода мы собираемся уменьшить количество используемых текстур.
1. Нам нужен новый композитор, на этот раз только с одной текстурой
```
compositor Compositor5
{
  technique
  {
    texture scene target_width target_height PF_R8G8B8
```
2. Заполним текстуру сценой
```
target scene
{
  input previous
}
```
3. Используем эту же текстуру и для входных и для выходных данных
```
target scene
{
 pass render_quad
 {
  material Ogre3DBeginnersGuide/Comp2
  input 0 scene
 }
}
```
4. Снова используем эту текстуру для конечного результата
```
target_output
{
 input none
 pass render_quad
 {
  material Ogre3DBeginnersGuide/Comp3
  input 0 scene
 }
}
```
5. Фигурные закрытые скобки
```
}
}
```
6. Скомпилируйте и запустите, вы увидите тот же результат, но теперь используется только одна текстура.

Что мы сделали.
Мы изменили наш композитор для использования только одной текстуры и обнаружили, что мы можем использовать одну текстуру, как для входных, так и для выходных данных.

Объединение композиторов в коде.
В последних двух примерах мы видели, как мы можем создать более сложные композиторы, их объединение и работа с различными материалами. Для того что бы объединить композиторы нам пришлось писать целый сценарий. Было бы неплохо объединять композиторы в коде, без необходимости писать сценарий. В следующем примере используем последовательность композиторов.
На этот раз нам не нужен новый композитор. Мы только немного изменим приложение.
1. Сначала добавьте и включите композитор инвертирования цветов
```
Ogre::CompositorManager::getSingleton().addCompositor(mCamera->getViewport(), "Compositor3");
Ogre::CompositorManager::getSingleton().setCompositorEnabled(mCamera->getViewport(), "Compositor3", true);
```
2. Затем добавьте черно-белый композитор
```
Ogre::CompositorManager::getSingleton().addCompositor(mCamera->getViewport(), "Compositor2");
Ogre::CompositorManager::getSingleton().
setCompositorEnabled(mCamera->getViewport(), "Compositor2", true);
```
3. Снова скомпилируйте и запустите приложение. Результат должен быть тот же, только на этот раз мы объединили композиторы в коде, а не в скрипте.

Что мы сделали.
Мы объединили два композитора используя функцию addCompositor(). Можно добавить любое число композиторов и объединить их вместе, каждый композитор будет принимать вывод другого. Если нам нужно добавить композитор в определенное место, то в функцию addCompositior() мы должны добавить третий параметр.

Кое что посложнее.
До этого момента мы использовали очень легкие композиторы. Давайте сделаем сложный.
Нам нужен новый композитор, материал и фрагментный шейдер.
1. В скрипте композитора ничего особенного. Нам нужна одна текстура для сцены, ее мы используем для вывода

```
compositor Compositor7
{
   technique
   {
      texture scene target_width target_height PF_R8G8B8
      target scene
      {
          input previous
      }

      target_output
      {
         input none
         pass render_quad
         {
            material Ogre3DBeginnersGuide/Comp5
            input 0 scene
         }
      }
   }
}
```

2. В самом материале тоже ничего нового, просто как обычно определите шейдер.

```
material Ogre3DBeginnersGuide/Comp5
{
  technique
   {
      pass
      {
     fragment_program_ref MyFragmentShader8
         {
         }

     texture_unit
         {
         }
      }
   }
}
```

3. Не забывайте определять шейдерную программу перед определение материала

```
fragment_program MyFragmentShader8 cg
{
  source Ogre3DBeginnersGuideShaders.cg
  entry_point MyFragmentShader8
  profiles ps_1_1  arbfp1
}
```

4. Теперь самое интересное. Заголовок шейдера не изменяется с предыдущего урока, изменяется только тело функции. Нам нужно две переменных, num и stepsize. Stepsize это единица деленная на num

float num= 50;

float stepsize = 1.0/ num;

5. Затем нужно использовать эти переменные и координаты текстуры для того чтобы вычислить новые координаты текстуры

*float2 fragment = float2(stepsize * floor(uv.x * num),stepsize * floor(uv.y * num));*

6. Используем новые координаты для получения цвета

color = tex2D(texture, fragment);

7. Измените программу так, что бы использовать только новый композитор. Затем скомпилируйте и запустите. Вы должны увидеть пикселизацию.

Что мы сделали.

Мы создали другой композитор, который сильно изменил нашу сцену. Сделал ее почти неузнаваемой. Шаг 1 и 2 вам знаком и не должно возникнуть вопросов. На 3 шаге мы устанавливаем параметры, которые позже нам понадобятся. Num это количество пикселей, который мы хотим использовать, мы установили в 50. Stepsize – нам нужна для вычисления текстурных координат. На 4 шаге мы вычислили наши новые текстурные координаты, используя старые и переменные.

Изменим число пикселей.

У нас есть композитор, который может уменьшить число пикселей в кадре, но у нас число пикселей жестко прописанов коде программы. Для использования других значений пикселей нам придется писать новый композитор. Это не эффективно. Первое что мы сделаем – это определим число пикселей не в шейдере а в материале.

Теперь мы собираемся управлять количеством пикселей из файла материала, а не из шейдера.

1. Создайте новый фрагментный шейдер, который имеет все старые параметры и один новый uniform float numpixels

```
void MyFragmentShader9(float2 uv : TEXCOORD0,
        out float4 color : COLOR,
        uniform sampler2D texture,
        uniform float numpixels)
{
```

2. Затем используйте новый параметр в коде шейдера

float num = numpixels;

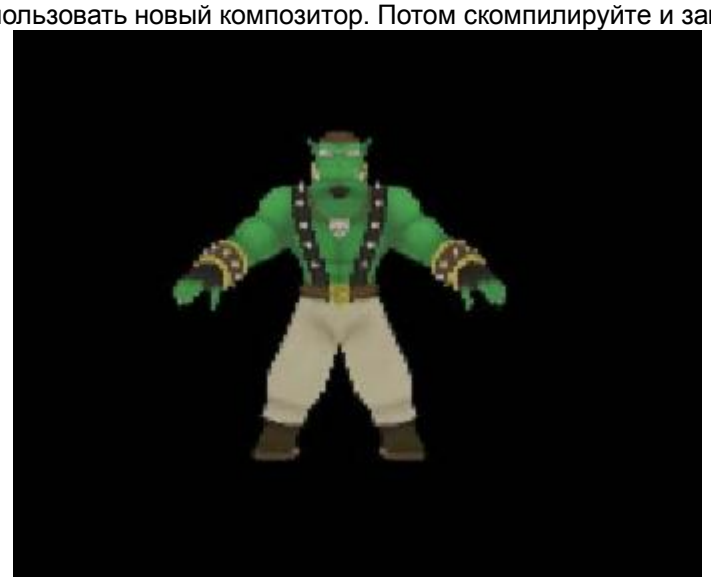
3. Остальну чвасть программы шейдера мы не трогаем.

```
float stepsize = 1.0/num;

float2 fragment = float2(stepsize * floor(uv.x * num),stepsize *
floor(uv.y * num));
  color = tex2D(texture, fragment);
}
```

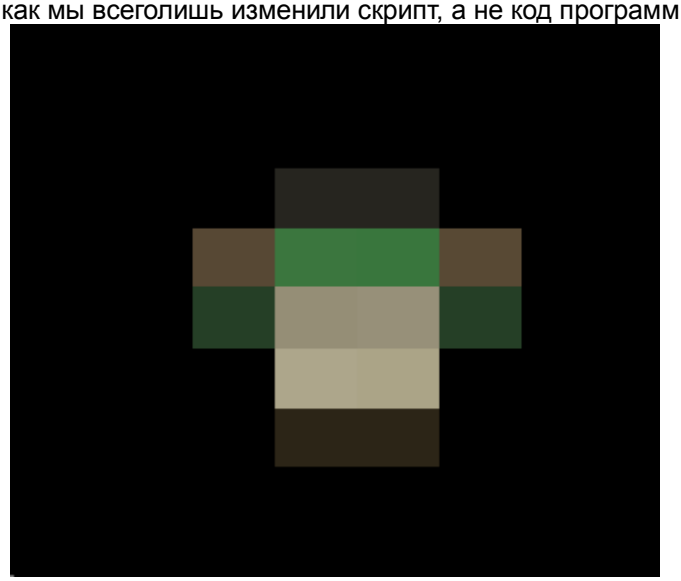
4. Скопируем пердыдуший материал, который будет являться почти копией предыдущего. Только немного изменим определений фрагментного шейдера, мы добавим новое значение по умолчанию в блок param. В этом блоке определите новый параметр numpixels float значение которого будет 500.

```
fragment_program MyFragmentShader9 cg
{
  source Ogre3DBeginnerGuideShaders.cg
  entry_point MyFragmentShader9
  profiles ps_1_1  arbfp1

  default_params
  {
    param_named numpixels float 500
  }
}
```

5. Теперь создайте новый композитор, который использует новый материал, и измените код программы, что бы использовать новый композитор. Потом скомпилируйте и запустите приложение.



6. Теперь измените значение numpixels в 25 и запустите приложение снова. Не нужно ничего перекомпилировать, так как мы всеголишь изменили скрипт, а не код программы.



Что мы сделали.
Мы сделали так, что бы можно было изменить количество пикселей без необходимости перекомпилирования приложения. Нам просто пришлось добавить универсальную переменную и добавить ее определение в блок default_params используя ключевое слово param_named. Имя и тип должно совпадать с именем переменной в фрагментном шейдере.

Установка переменной шейдера из кода программы.
Мы сделали определение переменной numpixels из файла материала, а теперь попробуем непосредственно из кода.

Мы можем использовать предыдущие материалы и композитор, только изменим код программы.
1. Мы не можем непосредственно работать с плоскостью экрана, так как наше приложение о нем ничего не знает. Единственный способ взаимодействия через слушателей (Listener). Ogre 3D обеспечил интерфейс для слушателей композитора. Му можем его использовать для создания нового
1. *class CompositorListener1 : public Ogre::CompositorInstance::Liste*
ner
{
public:
2. Переопределим метод установки материала
void notifyMaterialSetup (uint32 pass_id, MaterialPtr &mat)
{
3. Используем указатель на материал для изменения параметра numpixels на 125
mat->getBestTechnique()->getPass(pass_id)->getFragmentProgramParameters()-
>setNamedConstant("numpixels",125.0f);
}
4. Добавьте следующий код в createScene(), что бы получить экземпляр наборщика
Ogre::CompositorInstance comp =*
Ogre::CompositorManager::getSingleton().getCompositorChain(mCamera->getViewport())-
>getCompositor("Compositor8");
5. Нам нужно создать переменную слушателя
private:
CompositorListener1 compListener;*
6. Затем при создании приложения установите слушателю значение NULL
Example78()
{
compListener = NULL;
}
7. То что мы создаем, мы должны и удалить. Добавьте к приложению деструктор, который уничтожает экземпляр слушателя
~Example78()
{
delete compListener;
}
8. Теперь создайте слушателя и добавьте его в композитор
compListener = new CompositorListener1();
comp->addListener(compListener);
9. Скомпилируйте и запустите приложение, вы увидите следующую картину:

Что мы сделали.
Мы изменили код нашей программы таким образом, что бы можно было определать переменную шейдера из кода программы, вместо изменения в материале. Мы не имеем прямого доступа к плоскости экрана, но Ogre 3D нам предоставляет для этого интерфейс слушателя, от которого мы наследовали свой класс. Функция получает id прохода и указатель на материал. С помощью указателя на материал мы смогли определить параметр фрагментной программы. Это довольно длинный вызов, так как параметр находится в глубине иерархии классов. При использовании обычных объектов, мы можем добраться до параметров без использования слушателей.

Изменение количества пикселей при работающем приложении.
Мы уже научились изменять число пикселей из кода программы. Давайте попробуем сделать изменение числа пикселей взависимости от пользовательского ввода.
Мы собираемся использовать знания которые получили в трейтьей главе, связанной с пользовательским вводом.
1. Наше приложение нуждается в FrameListener. Добавьте новую переменную для сохранения указателя на приложение
Ogre::FrameListener FrameListener;*
2. Следующий шаг обнуление указателей
```
Example79()
{
  FrameListener = NULL;
  compListener = NULL;
}
```
3. А также и удаление
```
~Example79()
{
  if(compListener)
  {
    delete compListener;
  }
  if(FrameListener)
  {
    delete FrameListener;
  }
}
```
4. Добавьте создание FrameListener, который мы определим позже. Остальная часть приложения не изменится
```
void createFrameListener()
{
  FrameListener = new Example79FrameListener(mWindow,compListener);
  mRoot->addFrameListener(FrameListener);
}
```

5. Прежде, чем создать FrameListener, нужно изменить слушателя композитора. Нужна переменная для хранения числа пикселей

```
class CompositorListener1 : public Ogre::CompositorInstance::Liste
ner
{
private:
  float number;
```

6. Инициализируйте в приложении 125 пикселей

```
public:
  CompositorListener1()
  {
    number = 125.0f;
  }
```

7. Измените название функции переопределения материала с notifyMaterialSetup на notifyMaterialRender, и задайте вместо фиксированного значения пикселей, переменную

```
void notifyMaterialRender(uint32 pass_id, MaterialPtr &mat)
{
  mat->getBestTechnique()->getPass(pass_id)->getFragmentProgramParameters()-
>setNamedConstant("numpixels",number);
}
```

8. Определите задание и получение количества пикселей

```
void setNumber(float num)
{
  number = num;
}
float getNumber()
{
  return number;
}
```

9. Теперь добавьте класс FrameListener, у которогу будут три переменные, менеджер по вводу, класс клавиатуры и указатель на наборщик.

```
class Example79FrameListener : public Ogre::FrameListener
{
private:
  OIS::InputManager* _man;
  OIS::Keyboard* _key;
  CompositorListener1* _listener;
```

10. В конструкторе нам нужно создать пользовательский ввод и сохранить указатель

```
Example79FrameListener(RenderWindow* win,CompositorListener1*
listener)
{
  _listener = listener;
  size_t windowHnd = 0;
  std::stringstream windowHndStr;
  win->getCustomAttribute("WINDOW", &windowHnd);
  windowHndStr << windowHnd;
  OIS::ParamList pl;
  pl.insert(std::make_pair(std::string("WINDOW"),
windowHndStr.str()));
  _man = OIS::InputManager::createInputSystem( pl );
  _key = static_cast<OIS::Keyboard*>(_man->createInputObject(
OIS::OISKeyboard, false ));

}
```

11. А также корректное удаление всего что мы наделали

```
~Example79FrameListener()
{
  _man->destroyInputObject(_key);
  OIS::InputManager::destroyInputSystem(_man);
}
```

12. переопределите frameStarted, и напишите код, который по ESCAPE будет закрывать приложение

```
bool frameStarted(const Ogre::FrameEvent  &evt)
{
  _key->capture();

  if(_key->isKeyDown(OIS::KC_ESCAPE))
  {
    return false;
  }
```

13. При нажатии клавиши вверх нужно получить число пикселей и увеличить на 1

```
if(_key->isKeyDown(OIS::KC_UP))
{
  float num = _listener->getNumber();
  num++;
  _listener->setNumber(num);
  std::cout << num << std::endl;
}
```

14. Аналогично для уменьшения

```
if(_key->isKeyDown(OIS::KC_DOWN))
{
  float num = _listener->getNumber();
  num--;
  _listener->setNumber(num);
  std::cout << num << std::endl;
}
```

15. Завершим функцию frameStarted

```
  return true;
}
```

16. Скомпилируйте и запустите, я приведу пример трех различных уровней пикселизации

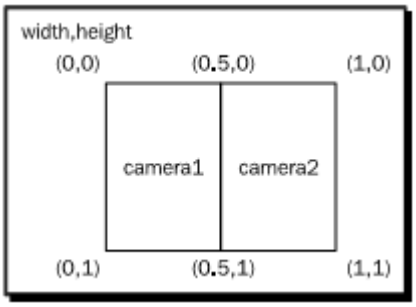
Что мы сделали.
Мы изменили наше приложение так, что мы сами можем управлять уровнем пикселизации с помощью стрелок на клавиатуре. Шаг 1 и шаг 4 добавили и создали FrameListener. Шаг 2 инициализировал FrameListener и CompositorListener переменной NULL. На 3 шаге описали удаление переменных. На 5 и 6 шаге мы вставили новую переменную для хранение числа пикселей в композиторе. На 4 шаге мы изменили метод, который мы переопределали, чтобы установить параметр в шейдер. Это необходимо так как notifyMaterialSetup вызывается только один раз при создании композитора, а notifyMaterialRender вызывается каждый кадр. Так как нам нужно менять число пикселей во время выполнения программы, лучшее решение это менять параметр. На 8 шаге мы релизовали методы get и set для управления числом пикселей, а 9 шаг начал реализовывать FrameListener. Нам нужен указатель на слушатель композитора, для того что бы мы могли менять значение пикселей, поэтому мы добавили указатель на слушатель композитора. На 10 шаге мы получили указатель на слушатель композитора и сохранили его в переменной. На 11 шаге ничего нового. На 13 и 14 шаге мы использовали методы set и get, что бы упаравлять числом пикселей в композиторе. 15 шаг закончил FrameListener.

Добавление разделения экрана
До этого момента мы видели, как добавлять композитор к области просмотра, но есть некоторые другие интересные вещи, которые можно сделать с областью просмотра, например разделение экрана.
1. После того как мы поиграпись с пикселями, добавим разделение экрана. Нам не нужен весь код предыдущего приложения, так что удалите FrameListener и CompositorListener.
2. Нам нужна еще одна камера, так что приготовим указатель
```
private:
  Ogre::Camera* mCamera2;
```
3. В createScene() нужно только создать экземпляр sinbad.mesh И присоединить его к узлу сцены.
```
void createScene()
{
  Ogre::SceneNode* node = mSceneMgr->getRootSceneNode()->createChildSceneNode();
  Ogre::Entity* ent = mSceneMgr->createEntity("Sinbad.mesh");
  node->attachObject(ent);
}
```
4. Теперь нам нужна функция createCamera(), в которой мы создадим камеру, которая будет смотреть на модель
```
void createCamera()
{
  mCamera = mSceneMgr->createCamera("MyCamera1");
  mCamera->setPosition(0,10,20);
  mCamera->lookAt(0,0,0);
  mCamera->setNearClipDistance(5);
```
5. Теперь вторую камеру в другой позиции
```
  mCamera2 = mSceneMgr->createCamera("MyCamera2");
  mCamera2->setPosition(20,10,0);
  mCamera2->lookAt(0,0,0);
  mCamera2->setNearClipDistance(5);
}
```
6. У нас есть камеры, и теперь нам нужно переопределить область просмотра, метод createViewports()
```
void createViewports()
{
```
7. Создадим область просмотра, которая перекрывает левую половину экрана, используя первую камеру.
```
Ogre::Viewport* vp = mWindow->addViewport(mCamera,0,0.0,0.0,0.5,1.0);
vp->setBackgroundColour(ColourValue(0.0f,0.0f,0.0f));
```
7. Затем создайте вторую область просмотра, которая будет показывать на правой стороне экрана, используя вторую камеру
```
Ogre::Viewport* vp2 = mWindow->addViewport(mCamera2,1,0.5,0.0,0.5,1.0);
vp2->setBackgroundColour(ColourValue(0.0f,0.0f,0.0f));
```
8. Обе камеры нуждаются в корректировке формата изображения, иначе изображение выгладит странно
```
mCamera->setAspectRatio(Real(vp->getActualWidth()) / Real(vp->getActualHeight()));
mCamera2->setAspectRatio(Real(vp2->getActualWidth()) / Real(vp2->getActualHeight()))
```

9. Скомпилируйте и запустите приложение. Вы должны увидеть одно и тоже с разных ракурсов

Что мы сделали.

Мы создали приложение с двумя областями просмотра, у каждой есть своя камера, которая смотрит на объект с разных точек. Так как мы хотим смотреть на модель с разных точек, каждая область просмотра должна иметь свою камеру. Поэтому на 2 шаге мы сделали новый указатель на камеру. На 3 шаге мы сделали простую сцену, которая содержит одну модель. На 4 шаге мы переопределили createCamera() и создали две камеры. Одна смотрит вдоль оси Z, другая вдоль оси X. На 6 шаге переопределили createViewports(). На 7 шаге создали первую область просмотра и добавили первую камеру к окну просмотра. Это было сделано используя функцию addViewport(). Первый параметр принимает камеру, второй определяет у какой области просмотра более высокий приоритет, 3,4,5,6 определяют положение, ширину и высоту области просмотра. Каждое значение находится в диапазоне 0 – 1.



На 9 шаге мы установили форматы изображения для каждой камеры. Кроме того если мы будем управлять мышью, мы увидим, что изменения происходят только в левой части экрана. Это потому что по умолчанию управляется камера указатель на которую называется mCamera.

Объединение всего этого

Мы научились создавать композиторы и разделять область просмотра. Теперь мы объединим все вместе, используем разные композиторы для каждой области просмотра.

Мы собираемся использовать предыдущий код, но нам нужно много чего добавить и удалить.

1. Начнем с фрагментного шейдера. Помимо обычных параметров, добавьте uniform параметр float4, в нем будет храниться фактор цветового канала

```
void MyFragmentShader10(float2 uv       : TEXCOORD0,
          out float4 color : COLOR,
          uniform sampler2D texture,
          uniform float4 factors
         )
{
  color = tex2D(texture, uv);
  color *= factors;
}
```

2. Создайте новый файл материала, который использует этот шейдер, и добавьте параметр со значением по умолчанию (1,1,1,0). Это означает неизмененный цвет

```
fragment_program MyFragmentShader10 cg
{
  source Ogre3DBeginnerGuideShaders.cg
  entry_point MyFragmentShader10
  profiles ps_1_1  arbfp1

  default_params
  {
    param_named factors float4 1 1 1 0
  }


}
material Ogre3DBeginnersGuide/Comp7
{
  technique
    {
       pass
       {
     fragment_program_ref MyFragmentShader10
          {
          }

     texture_unit
          {
          }
       }
    }
}
```

3. Добавьте композитор использующий этот материал

```
compositor Compositor9
{
   technique
    {
       texture scene target_width target_height PF_R8G8B8
       target scene
       {
          input previous
       }
       target_output
       {
          input none
          pass render_quad
          {
             material Ogre3DBeginnersGuide/Comp7
             input 0 scene
          }
       }
    }
}
```

4. У нас есть 3 цветных канала, мы хотим изменить цвет для четырех областей просмотра, значит нам нужно 3 композиторного слушателя, в которых будут изменяться параметры материала. Во первых установим красный канал. Параметр будем менять при создании композиторного материала, так как нам не нужно каждый кадр задавать параметр

```
class CompositorListener2 : public Ogre::CompositorInstance::Listener
{
public:
  void notifyMaterialSetup (uint32 pass_id, MaterialPtr &mat)
```

```
{
    mat->getBestTechnique()->getPass(pass_id)->getFragmentProgramParameters()-
>setNamedConstant("factors",Ogre::
Vector3(1,0,0));
  }
}
```
5. Также для синего и зеленого
```
class CompositorListener3 : public Ogre::CompositorInstance::Liste
ner
{
public:
  void notifyMaterialSetup (uint32 pass_id, MaterialPtr &mat)
  {
    mat->getBestTechnique()->getPass(pass_id)->getFragmentProgramParameters()-
>setNamedConstant("factors",Ogre::
Vector3(0,1,0));
  }
};
class CompositorListener4 : public Ogre::CompositorInstance::Liste
ner
{
public:
  void notifyMaterialSetup (uint32 pass_id, MaterialPtr &mat)
  {
    mat->getBestTechnique()->getPass(pass_id)->getFragmentProgramParameters()-
>setNamedConstant("factors",Ogre::
Vector3(0,0,1));
  }
};
```
6. Определим указатели на 4 области просмотра
```
class Example83 : public ExampleApplication
{
private:
  Ogre::Viewport* vp;
  Ogre::Viewport* vp2;
  Ogre::Viewport* vp3;
  Ogre::Viewport* vp4;
```
7. Создадим камеру, которая будет смотреть на Синбада
```
void createCamera()
{
  mCamera = mSceneMgr->createCamera("MyCamera1");
  mCamera->setPosition(0,10,20);
  mCamera->lookAt(0,0,0);
  mCamera->setNearClipDistance(5);
}
```
8. Настроим области просмотра на одну камеру
```
void createViewports()
{
  vp = mWindow->addViewport(mCamera,0,0.0,0.0,0.5,0.5);
  vp->setBackgroundColour(ColourValue(0.0f,0.0f,0.0f));
  vp2 = mWindow->addViewport(mCamera,1,0.5,0.0,0.5,0.5);
  vp2->setBackgroundColour(ColourValue(0.0f,0.0f,0.0f));
  vp3 = mWindow->addViewport(mCamera,2,0.0,0.5,0.5,0.5);
  vp3->setBackgroundColour(ColourValue(0.0f,0.0f,0.0f));
  vp4 = mWindow->addViewport(mCamera,3,0.5,0.5,0.5,0.5);
  vp4->setBackgroundColour(ColourValue(0.0f,0.0f,0.0f));
  mCamera->setAspectRatio(Real(vp->getActualWidth()) /
Real(vp->getActualHeight()));
}
```

9. Добавим в приложение указатели на композиторные слушатели
```
CompositorListener2* compListener;
CompositorListener3* compListener2;
CompositorListener4* compListener3;
```
10. Обнулим каждый
```
Example83()
{
  compListener = NULL;
  compListener2 = NULL;
  compListener3 = NULL;
}
```
11. И добавим для уничтожения
```
~Example83()
{
    if(compListener)
  {
    delete compListener;
  }
  if(compListener2)
  {
    delete compListener2;
  }
  if(compListener3)
  {
    delete compListener3;
  }
}
```
12. В createScene() после создания модели, добавим композитор к нашей первой области просмотра, и присоединим к нему слушатель, который добавит красный канал
```
Ogre::CompositorManager::getSingleton().addCompositor(vp, "Compositor9");
Ogre::CompositorManager::getSingleton().setCompositorEnabled(vp, "Compositor9", true);
Ogre::CompositorInstance* comp =
 Ogre::CompositorManager::getSingleton().getCompositorChain(vp)->getCompositor("Compositor9");
compListener = new CompositorListener2();
comp->addListener(compListener);
```
13. Тоже самое со 2 и 3 областью просмотра, где будет использоваться зеленый и синий канал
```
Ogre::CompositorManager::getSingleton().addCompositor(vp2, "Compositor9");
Ogre::CompositorManager::getSingleton().setCompositorEnabled(vp2, "Compositor9", true);
Ogre::CompositorInstance* comp2 =
Ogre::CompositorManager::getSingleton().getCompositorChain(vp2)->getCompositor("Compositor9");
compListener2 = new CompositorListener3();
comp2->addListener(compListener2);
Ogre::CompositorManager::getSingleton().addCompositor(vp3, "Compositor9");
Ogre::CompositorManager::getSingleton().setCompositorEnabled(vp3, "Compositor9", true);
Ogre::CompositorInstance* comp3 =
Ogre::CompositorManager::getSingleton().getCompositorChain(vp3)->getCompositor("Compositor9");
compListener3 = new CompositorListener4();
comp3->addListener(compListener3);
```
14. Теперь скомпилируйте приложение. Вы увидите четыре идентичных изображения, только представленных в разных цветовых каналах.

Что мы сделали

Мы мспользовали знания которые получили в этой главе, что бы создать приложение с четырьмя областями просмотра и одним композитором в комбинации с тремя слушателями. Ничего нового мы не делали, если что-то не понятно перечитайте главу внимательно.

Подведем итоги

Мы многое узнали об областях просмотра в этой главе.

Мы научились:

- Создавать скрипты композиторов и добавлять к нашей сцене
- Использовать шейдеры в композиторах
- Изменять параметры шейдеров в композиторах
- Объединять композиторы, что бы не писать один и тот же код
- Создавать композитор в котором можно менять параметры шейдера на лету

Мы очень многое изучили в Ogre 3D. Но до сих пор мы использовали вспомогательный класс ExampleApplication. В следующей главе мы будем писать свой собственный ExapleApplication снуля.