ORXONOX Coding Tutorial mit Ogre

1 Ogre auschecken und kompilieren

Als erstes erstellen wir einen neuen Ordner im Homeverzeichnis, zum Beispiel 'orxonox' und wechseln in ihn:

```
cd
mkdir orxonox
cd orxonox
```

Nun müssten wir eigentlich Ogre kompilieren. Da wir dies aber auf den Tardis-Rechnern zur Zeit nicht selbst tun, sondern eine vorkompilierte Version verwenden (unser Dank gilt dem hilfsbereiten ISG.ee-Team), könnt ihr direkt zum zweiten Kapitel springen. Wer auf einem Notebook oder zu Hause arbeiten will, sollte sich an die folgende Anleitung halten. Für Benutzer mit anderen Betriebssystemen gibt es ausführliche Tutorials auf der Ogre-Wiki unter folgender Adresse:

http://www.ogre3d.org/wiki/index.php/Building_From_Source

Kurze Anleitung zum selbst kompilieren unter Linux

Wir checken nun Ogre über SVN in ein neues Verzeichnis, zum Beispiel 'ogre', aus. Dazu benutzen wir den svn co Befehl. Dies dauert ein Weilchen, insbesondere wenn mehrere User gleichzeitig auf den Server zugreifen.

```
svn co https://svn.orxonox.net/ogre ogre
```

Nun kompilieren wir Ogre:

```
cd ogre
aclocal-1.9.5 (möglicherweise nicht notwendig)
./bootstrap
./configure
make
make install
```

Wenn alles funktioniert hat, sollten sich im Ordner orxonox/ogre/Samples/Common/bin/Release mehrere Binaries befinden. Dabei handelt es sich um die Beispielprogramme, mit denen verschiedene Funktionalitäten von Ogre demonstriert werden.

2 Tutorialcode auschecken und kompilieren

Nun wollen wir das Tutorial auschecken. Damit ihr die nötigen Rechte habt (auf dem trunk habt ihr keine Schreibrechte), befindet sich das Tutorial in einem eigenen Branch. Stellt sicher, dass ihr euch wieder im Ordner 'orxonox' befindet (falls ihr das erste Kapitel übersprungen habt, erstellt den Ordner wie oben beschrieben). Das Auschecken erfolgt mit folgendem Befehl:

```
svn co https://svn.orxonox.net/orxonox/branches/tutorial tutorial
```

Wir kompilieren den Tutorialcode mit einem Shellscript. (Vorsicht: Das Skript wurde für die Bedingungen auf den Tardis-Rechnern erstellt. Wenn ihr Ogre selbst kompiliert habt, ist es am besten, wenn ihr den Ordner 'Tutorial' nach 'orxonox/ogre/Samples' verschiebt und ein Makefile von den anderen Beispielapplikationen übernehmt.)

```
cd tutorial
cd Tutorial
cd build
chmod -v 755 build-script
./build-script
```

Nun sollte im Ordner 'orxonox/tutorial/Tutorial/build' ein Binary namens 'main' erscheinen. Damit alles richtig funktioniert, sollte die Datei nicht direkt ausgeführt werden. Wir verwenden dafür ein eigenes Shellscript:

```
./run-script
```

Wenn man es ausführt, müsste ein schwarzer Bildschirm zu sehen sein, mit ESC lässt sich die Applikation wieder beenden.

3 Ein bisschen Theorie

Auszug aus der Ogre-Wiki (http://www.ogre3d.org/wiki/):

How Ogre Works

A broad topic. We will start with SceneManagers and work our way to Entities and SceneNodes. These three classes are the fundamental building blocks of Ogre applications.

SceneManager Basics

Everything that appears on the screen is managed by the SceneManager (fancy that). When you place

objects in the scene, the SceneManager is the class which keeps track of their locations. When you create Cameras to view the scene the SceneManager keeps track of them. When you create planes, billboards, lights and so on, the SceneManager keeps track of them.

There are multiple types of SceneManagers. There are SceneManagers that render terrain, there is a SceneManager for rendering BSP maps, and so on.

Entity Basics

An Entity is one of the types of object that you can render on a scene. You can think of an Entity as being anything that's represented by a 3D mesh. A robot would be an entity, a fish would be an entity, the terrain your characters walk on would be a very large entity. Things such as Lights, Billboards, Particles, Cameras, etc would not be entities.

One thing to note about Ogre is that it separates renderable objects from their location and orientation. This means that you cannot directly place an Entity in a scene. Instead you must attach the Entity to a SceneNode object, and this SceneNode contains the information about location and orientation.

SceneNode Basics

As already mentioned, SceneNodes keep track of location and orientation for all of the objects attached to it. When you create an Entity, it is not ever rendered on the scene until you attach it to a SceneNode. Similarly, a SceneNode is not an object that is displayed on the screen. Only when you create a SceneNode and attach an Entity (or other object) to it is something actually displayed on the screen.

SceneNodes can have any number of objects attached to them. Let's say you have a character walking around on the screen and you want to have him generate a light around him. The way you do this would be to first create a SceneNode, then create an Entity for the character and attach it to the SceneNode. Then you would create a Light object and attach it to the SceneNode. SceneNodes may also be attached to other SceneNodes which allows you to create entire hierarchies of nodes.

One major concept to note about SceneNodes is that a SceneNode's position is always relative to its parent SceneNode, and each SceneManager contains a root node to which all other SceneNodes are attached.

4 Content hinzufügen

Wir arbeiten in den Kapiteln vier bis acht ausschliesslich mit der Datei `main.cpp`. Sie befindet sich im Ordner `orxonox/tutorial/Tutorial/src`. Öffnet sie und folgt dem nachfolgenden Tutorial, welches teilweise der Ogre-Wiki entnommen wurde.

Your first Ogre application

Find the `TutorialApplication::createScene` member function. We will only be manipulating the contents of this function in this part of the tutorial. The first thing we want to do is set the ambient light for the scene so that we can see what we are doing. We do this by calling the `setAmbientLight` function and specifying what color we want. Note that the `ColourValue` constructor expects values for red, green, and blue in the range between 0 and 1. Add this line to `createScene`:

```
mSceneMgr->setAmbientLight(ColourValue(0.3, 0.3, 0.3));
```

The next thing we need to do is create an Entity. We do this by calling the SceneManager's `createEntity` member function:

```
Entity* head = mSceneMgr->createEntity("head", "ogrehead.mesh");
```

Ok several questions should pop up. First of all, where did `mSceneMgr` come from, and what are the parameters we are calling the function with? The `mSceneMgr` variable contains the current SceneManager object (this is done for us by the ExampleApplication class). The first parameter to `createEntity` is the name of the Entity we are creating. All entities must have a unique name. You will get an error if you try to create two entities with the same name. The '`ogrehead.mesh`' parameter specifies the mesh we want to use for the Entity. Again, the mesh that we are using has been preloaded for us by the ExampleApplication class.

Now that we have created the Entity, we need to create a SceneNode to attach it to. Since every SceneManager has a root SceneNode, we will be creating a child of that node:

```
SceneNode *node = mSceneMgr->getRootSceneNode()
        ->createChildSceneNode("OgreHeadNode", Vector3(0, 0, 0));
```

This long statement first calls the `getRootSceneNode` of the current SceneManager. Then it calls the `createChildSceneNode` method of the root SceneNode. The parameter to `createChildSceneNode` is the name of the SceneNode we are creating. Like the Entity class, no two SceneNodes can have the same name.

Finally, we need to attach the Entity to the SceneNode so that the Ogre-head has a location to be rendered at:

```
node->attachObject(head);
```

And that's it! Compile and run your application. You should see the Ogre-head on the screen.

Damit der Raum nicht völlig leer ist, wollen wir eine Umgebung hinzufügen. Diese in 3D zu modellieren, würde aber den Rahmen dieses Tutorials bei weitem übersteigen, weshalb wir uns darauf beschränken, eine Skybox hinzuzufügen.

```
mSceneMgr->setSkyBox(true, "Examples/SpaceSkyBox");
```

Hintergrundinformation: *Eine Skybox ist ein Würfel mit der texturierten Seite nach innen. Als Texturen werden sechs quadratische Bilder verwendet, welche sich nahtlos zusammenfügen lassen, so dass, aus der Mitte betrachtet, der Eindruck einer dreidimensionalen Umgebung entsteht. Damit man sich trotzdem bewegen kann, ohne diesen Effekt zu zerstören, darf die Darstellung der Skybox nicht von der Position der Kamera abhängen (d.h. man kann sich nicht auf eine Wand der Skybox zubewegen - der Abstand bleibt immer konstant). Diese Technik wird in nahezu jedem 3D-Spiel verwendet.*

5 Licht erzeugen

Auszug aus der Ogre-Wiki:

Types of Lights

There are three types of lighting that Ogre provides.

- Point (`LT_POINT`) - Point light sources emit light from them in every direction.
- Spotlight (`LT_SPOTLIGHT`) - A spotlight works exactly like a flashlight does. You have a position where the light starts, and then light heads out in a direction. You can also tell the light how large of an angle to use for the inner circle of light and the outer circle of light (you know how flashlights are brighter in the center, then lighter after a certain point?).
- Directional (`LT_DIRECTIONAL`) - Directional light simulates far away light that hits everything in the scene from a direction. Lets say you have a night time scene and you want to simulate moonlight. You could do this by setting the ambient light for the scene, but that's not exactly realistic since the moon does not light everything equally (neither does the sun). One way to do this would be to set a directional light and point in the direction the moon would be shining.

Lights have a wide range of properties that describes how the light looks. Two of the most important properties of a light is its diffuse and specular color. Each material script defines how much diffuse and specular lighting the material reflects.

Creating the Lights

To create a Light in Ogre we call SceneManager's `createLight` member function and supply the light's name, very much like how we create an Entity. After we create a Light, we can either set the position of it manually or attach it to a SceneNode for movement. Unlike the Entity object, light only has `setPosition` and `setDirection` (and not the full suite of movement functions like translate, pitch, yaw, roll, etc). So if you need to create a stationary light, you should call the `setPosition` member function. If you need the light to move (such as creating a light that follows a character), then you should attach it to a SceneNode instead.

So, lets start with a basic point Light. The first thing we will do is create the light, set its type, and set its position:

```
Light *light = mSceneMgr->createLight("Light1");
light->setType(Light::LT_POINT);
light->setPosition(Vector3(0, 100, 0 ));
```

Now that we have created the light, we can set its diffuse and specular color. Let's make it red:

```
light->setDiffuseColour(1.0, 0.0, 0.0);
light->setSpecularColour(1.0, 0.0, 0.0);
```

Now compile and run the application. Success! We can now see the Ogre-head with red light on it. Be sure to also look at him from the side, a complete silhouette. One thing to notice is that you do not "see" the light source. You see the light it generates but not the actual light object itself.

Um die Lichtquelle sichtbar zu machen, erstellen wir ein Billboard. Dies geschieht mit folgendem Code:

```
BillboardSet *bbs = mSceneMgr->createBillboardSet("bb", 1);
bbs->createBillboard(Vector3::ZERO, ColourValue(1.0, 0.0, 0.0));
bbs->setMaterialName("Examples/Flare");
```

Hintergrundinformation: *Ein Billboard ist ein 2D-Objekt, welches nur aus einer Textur besteht, welche häufig transparent ist. Das Billboard dreht sich immer zur Kamera, d.h. man kann es nie von der Seite sehen. Dies eignet sich für sogenannte Flares (so etwas wie ein Lichtschein), um Lichtquellen eine Aura und einen gewissen Blendeffekt zu verleihen. In alten Spielen sieht man häufig auch Grasbüschel und ganze Bäume (oder applaudierende Zuschauer in Sportspielen), welche nur aus einer Textur bestehen und sich immer der Kamera zuwenden. In aktuellen Spielen wird dieser Effekt eigentlich nur noch für Partikel und Flares verwendet.*

Um das Billboard und die Lichtquelle zu vereinen, benötigen wir eine zusätzliche SceneNode. Da wir später damit spielen wollen, deklarieren wir sie global:

```
SceneNode *lightNode;
```

Bitte denkt daran: Globale Variabeln sind unschön. Würden wir Orxonox mit globalen Variabeln programmieren, würde das Projekt keine drei Wochen überleben. Die korrekte Vorgehensweise wäre: Einen Licht-Manager implementieren, das Licht erstellen, es dem Manager übergeben und später wieder abrufen. Für dieses Tutorial wäre dies aber übertrieben, deshalb drücken wir ein Auge zu.

In `createScene` fügen wir nun folgenden Code hinzu:

```
lightNode = mSceneMgr->getRootSceneNode()->createChildSceneNode("LightNode", Vector3(0, 100, 0));
```

Nun müssen wir das Billboard und die Lichtquelle an die SceneNode hängen:

```
lightNode->attachObject(bbs);
lightNode->attachObject(light);
```

Aber Vorsicht: Weil wir sowohl `light` als auch `lightNode` an der Position `(0, 100, 0)` erstellt haben und die Position eines Objekts relativ zur SceneNode gerechnet wird, ist unser Licht nun plötzlich an der Position `(0, 200, 0)`. Weil wir fortan nur noch die SceneNode bewegen wollen, soll das Licht die relative Position `(0, 0, 0)` haben, d.h. es soll sich genau auf der SceneNode befinden (was zur absoluten Position `(0, 100, 0)` führt. Alles klar?). Dies erreichen wir mit einem erneuten Aufruf von `setPosition`:

```
light->setPosition(0.0, 0.0, 0.0);
```

Ein kleines Stück Theorie aus der Ogre-Wiki:

Before we go any further, we need to talk about screen coordinates and Ogre Vector objects. Ogre (like many graphics engines) uses the x and z axis as the horizontal plane, and the y axis as your vertical axis. As you are looking at your monitor now, the x axis would run from the left side to the right side of your monitor, with the right side being the positive x direction. The y axis would run from the bottom of your monitor to the top of your monitor, with the top being the positive y direction. The z axis would run into and out of your screen, with out of the screen being the positive z direction.

6 Der Frame Listener

Weil das Ganze in dieser Form etwas langweilig ist, wollen wir die Lichtquelle bewegen. Da wir bisher den ganzen Code in eine Funktion geschrieben haben, welche nur beim Start der Applikation aufgerufen wird, können wir nicht einfach weiterprogrammieren.

Eine naive Idee wäre es, an dieser Stelle eine Schlaufe zu programmieren, in welcher wir die lightNode bewegen. Dies kann aber unmöglich funktionieren, da diese Schleife den gesamten Programmablauf blockieren würde und wir so gar nicht erst ein Bild sehen würden. Wir sind also gezwungen, eine Funktion zu erstellen, welche im laufenden Programm bei jedem Durchlauf (Tick) einmal aufgerufen wird. Dies realisieren wir mit einem Frame Listener.

Auszug aus der Ogre-Wiki:

Introduction

In the previous tutorials we only looked at what we could do when we add code to the `createScene`

method. In Ogre, we can register a class to receive notification before and after a frame is rendered to the screen. This FrameListener interface defines two functions:

```
bool frameStarted(const FrameEvent& evt)
bool frameEnded(const FrameEvent& evt)
```

Ogre's main loop (`Root::startRendering`) looks like this:

1. The Root object calls the `frameStarted` method on all registered FrameListeners.
2. The Root object renders one frame.
3. The Root object calls the `frameEnded` method on all registered FrameListeners.

This loops until any of the FrameListeners return false from `frameStarted` or `frameEnded`. The return values for these functions basically mean "keep rendering". If you return false from either, the program will exit. The FrameEvent object contains two variables, but only the `timeSinceLastFrame` is useful in a FrameListener. This variable keeps track of how long it's been since the `frameStarted` or `frameEnded` last fired. Note that in the `frameStarted` method, `FrameEvent::timeSinceLastFrame` will contain how long it has been since the last `frameStarted` event was last fired (not the last time a `frameEnded` method was fired).

Ein Frame Listener ist im Tutorial Code bereits enthalten. In der Funktion `createFrameListener` der Tutorial Applikation wird der Frame Listener erzeugt.

Damit sich das Licht bewegt, fügen wir folgenden Code in die Funktion `frameStarted` des TutorialFrameListeners ein:

```
lightNode->translate(Vector3(0, -10 * evt.timeSinceLastFrame, 0));
```

Diese Zeile lässt das Licht jede Sekunde 10 Units nach unten wandern (da es an der Position `(0, 100, 0)` erstellt wurde, erreicht es nach 10 Sekunden den Ursprung des Systems).

Zum Nachrechnen: *-10 ist die Geschwindigkeit, evt.timeSinceLastFrame ist die Anzahl Sekunden seit dem letzten Frame. Bei 10 Frames pro Sekunde ist evt.timeSinceLastFrame demnach 0.1, wodurch sich das Licht jedes Frame 1 Unit nach unten (auf der Y-Achse) bewegt, in einer Sekunde also tatsächlich 10 Units.*

7 Eigene Ideen verwirklichen

Ihr habt nun ein bisschen Zeit, um mit dem Code zu spielen. Ändert den Code, der das Licht bewegen lässt, um einen neuen Bewegungsablauf zu erzeugen. Zum Beispiel könnte das Licht an der Position `(0, -100, 0)`

umkehren und sich wieder nach oben bewegen. Ihr könnt auch Winkelfunktionen verwenden, um eine kreisförmige Bewegung zu erzeugen. Selbstverständlich dürft ihr auch die Farbe des Lichts variieren oder weitere Lichter hinzufügen. Wir werden euren Code anschliessend individuell anschauen.

8 Auf SVN commiten

Damit eure Werke nicht verloren gehen, commiten wir alle beendeten Tutorials auf einen Branch des Orxonox Repository. Dazu muss die main.cpp umbenannt werden, weil ihr euch sonst gegenseitig die Files überschreibt. Hängt euer Kürzel an den Dateinamen an, damit wir sie unterscheiden und zuordnen können:

```
mv main.cpp main_euerkuerzel.cpp
```

Damit wir die Datei hochladen können, müssen wir sie zuerst bei SVN adden, denn die umbenannte Datei ist für SVN eine neue Datei (solltet ihr eine Datei direkt auf SVN umbenennen wollen, müsst ihr den '`svn mv dateialt dateineu`' Befehl verwenden). Adden geht mit folgendem Befehl:

```
svn add main_euerkuerzel.cpp
```

Bevor wir die Datei hochladen können, müssen wir zuerst das Repository neu auschecken, um sicherzugehen, dass inzwischen nichts verändert wurde (ausserdem wird es von SVN verlangt, weil wir `main.cpp` umbenannt haben):

```
svn up
```

Nun kann die Datei hochgeladen werden. Damit später auf einen Blick erkannt werden kann, was bei diesem Update verändert wurde, muss ein Kommentar angegeben werden. Wir schreiben: 'Tutorialcode von euerkuerzel'. Folgender Befehl bewirkt ein Hochladen (commiten) mit Kommentar:

```
svn ci -m "tutorialcode von euerkuerzel"
```

Falls dabei eine Fehlermeldung erscheint, hat vermutlich ein anderer User zwischenzeitlich seinen Code commitet. Wiederholt in diesem Fall den `svn up` Befehl.

9 SVN Konflikte lösen

Damit ihr einmal gesehen habt, was bei SVN schief gehen kann, wenn mehrere Personen den selben Teil des selben Files verändern, öffnen wir die Datei `name.txt` und editieren sie. Schreibt dazu euren Namen hinein:

Your Name: Euer Name

Speichert die Datei und commitet sie:

```
svn ci -m "crash!"
```

Der erste User wird Glück haben: Die Datei wird problemlos hochgeladen. Bei allen anderen wird es zu einem Problem kommen: Erst beklagt sich SVN über die veraltete Revision. Wenn dann mit `svn up` geupdatet werden soll, kommt es zum Konflikt, weil mehrere User die selbe Zeile verändert haben.

In diesem Fall ist dies harmlos, da es nur ein Test ist. Sollten aber bei wichtigen Files Konflikte auftreten, müsst ihr euch unbedingt mit der anderen Person absprechen, um fehlerhaftes Mergen zu vermeiden. Im Zweifelsfall könnt ihr euch natürlich auch an uns wenden.

10 SVN Übersicht

Allgemeines

Subversion (SVN) ist ein *Version Control System*, ein Programm zur Versionsverwaltung von Dateien, hauptsächlich Software-Quellcode. Subversion ist eine Weiterentwicklung von CVS (das noch heute viel eingesetzt wird, z.B bei SourceForge), welches seinerseits eine Weiterentwicklung von RCS (Revision Control System) ist. Subversion erlaubt mehreren Entwicklern gemeinsam an einem Projekt zu arbeiten, ohne sich gegenseitig Code zu löschen.

Funktionsweise

Subversion vereinfacht die Verwaltung von Quellcode dadurch, dass es alle Dateien eines Software-Projektes an einer zentralen Stelle, einem so genannten *Repository*, speichert. Dabei können jederzeit einzelne Dateien verändert werden, es bleiben jedoch alle früheren Versionen erhalten, einsehbar und wiederherstellbar. Ausserdem können Unterschiede zwischen bestimmten Versionen verglichen werden.
Ein Entwickler holt sich die Daten vom Server (meist über http/https) und arbeitet zu Hause an einer lokalen Kopie. Sobald er seine Arbeit veröffentlichen will, wird er die Daten (die Änderungen) wieder auf den Server laden (*commit*). Der Server wird dann seine eigene Kopie updaten und schauen, dass alle anderen Entwickler an eine aktuelle Version gelangen können. Probleme können sich ergeben, wenn mehrere Mitarbeiter gleichzeitig an der selben Stelle in einer Datei Änderungen vornehmen. In diesem Fall muss der Entwickler manuell die verschiedenen Änderungen zusammenfügen (*merge*).

Anwendungslexikon

Im Folgenden findet Ihr eine Zusammenfassung der wichtigsten Befehle, welche ihr bei SVN benötigen werdet. Unter vielen Systemen gibt es grafische Oberflächen für SVN, welche die Handhabung wesentlich vereinfachen. Unter Linux können wir RapidSVN[1] empfehlen, auf Windows hat sich TortoiseSVN[2] bewährt.

- Daten herunterladen: Dieser Befehl lädt die neueste Version von `[repository]` an die angegebene `[destination]`.

```
svn checkout [repository] [destination]
svn co [repository] [destination]
```

- Dateien hinzufügen: Mit diesem Befehl können neue Dateien oder Verzeichnisse (`[file]`) zum Projekt hinzugefügt werden.

```
svn add [file]
```

- Dateien löschen: Um Dateien aus dem Repository zu entfernen, muss dieser Befehl verwendet werden.

```
svn rm [file]
```

[1] http://rapidsvn.tigris.org/
[2] http://tortoisesvn.tigris.org/

- Änderungen anzeigen: Damit können lokale Änderungen eingesehen werden.

```
svn diff
```

- Neue Version herunterladen: Synchronisiert das lokale Repository mit dem Server und lädt alle Änderungen herunter.

```
svn up
```

- Daten hochladen: Dieser Befehl lädt die lokalen Änderungen auf den Subversion Server hoch. Zur besseren Verständlichkeit wird ein Kommentar `[message]` hinzugefügt.

```
svn checkin -m "message"
svn ci -m "message"
```

- Weitere Informationen: Falls du mehr über einen gewissen `[command]` erfahren willst[3], bietet sich der help Befehl an.

```
svn help [command]
```

Subversion in Orxonox

Orxonox verwendet eine spezielle Syntax der Log-Nachrichten beim commiten von Änderungen. Diese Notation erlaubt dem Betrachter einer Log-Nachricht, jederzeit die Tätigkeit und den Arbeitsort festzustellen:

```
svn ci -m "orxonox/[folder]: [message]"
```

Wobei `[folder]` den Ort bezeichnet, an dem du gearbeitet hast, also z.B. in *trunk* oder in *branches/student1*. Die Nachricht `[message]` sollte möglichst kurz und einfach beschreiben, was verändert wurde und weswegen (dies ist insbesondere dann wichtig, wenn man Code von anderen Leuten verändert).
Auch wenn es zwischendurch mühsahm erscheinen mag: Es ist sehr wichtig, dass du **immer eine Log-Nachricht spezifizierst** beim commiten, da die anderen Entwickler ansonsten den Inhalt des Commits selber herausfinden müssen.

[3] Alternative Informationsquelle: http://svnbook.red-bean.com/