Master Fachprojekt WS 2009/10

Erstellen einer Echtzeitgrafikanwendung in OGRE 3D

Lukas Kalinowski (937763)
Stefan Gohlke (937313)
Frederic Frieß (947998)

Betreuer: Prof. Dr. Martin Rumpler

Inhaltsverzeichnis

1 Projektplanung

Im Rahmen des Fachprojektes soll auf Basis des bestehenden Spiels „*Flubbers*“ [FLUB] aus der Bachelorarbeit von Lukas Kalinowski mit dem Titel „Spiele in einer 3D-Welt – Von der Idee bis zur Realisierung“ eine kleine Spiel-Anwendung realisiert werden. Dabei liegt der Schwerpunkt auf der optischen Präsentation und der Integration eines Wii-Controllers zur Interaktion. Im Folgenden werden die Projektziele und Ausgangslage erläutert.

1.1 Projektziele

Es soll zunächst für jedes Teammitglied eine geeignete Arbeitsumgebung geschaffen werden. Damit ist insbesondere die nichttriviale Einrichtung von Eclipse für C++ und das Aufsetzen eines Versioning Systems (SVN) als gemeinsame Ressourcenbasis, gemeint. Weiterhin war die Vorstellung aller Teammitglieder einen anspruchsvoll modellierten, texturierten und animierten Charakter als Protagonist zu steuern.
Die Steuerung der Figur soll über die Wii-Remote der Nintendo Wii Spielkonsole erfolgen. Inspiriert von aktuellen Third-Person-Games soll die Anwendung ebenfalls über eine dynamische Kamera verfügen.
Zur ansprechenden Demonstration der Anwendung soll eine Testumgebung erstellt werden, in der jedes Feature der Anwendung zur Geltung kommt.

1.2 Persönliche Ziele

Stefan Gohlke:
Ein Projektziel ist das Erstellen von Animationen mit dem Programm Blender für eine Echtzeitgrafik-Anwendung anzueignen und generell die notwendigen Zusammenhänge und Arbeitsschritte beim Erstellen einer solchen Anwendung mit Ogre3D kennenzulernen.
Die Interaktionsmöglichkeiten mit der Wii-Steuerung ist ebenfalls ein persönliches Lernziel.

Frederic Frieß:
Das wichtigste Ziel beschäftigt sich mit der Modellierung und Texturierung des Hauptcharakters. Er soll durch eine humorvolle Idee, dem Geometrisches Erscheinungsbild und einer harmonischer Farbkomposition visuell ansprechend wirken. Ich habe mir zum Ziel gesetzt, den Entstehungsprozess mit Einsatz der folgenden Verfahren zu realisieren: Poly-Modelling, Skulpting, Texture-Painting, 2D-Textur Bearbeitung und Shader-Programmierung. Hierbei will ich die Tools Blender3D und ZBrush nutzen.Ein weiteres Ziel beschäftigt sich mit der Entwicklung einer dynamischen Thirde-Person-Kamera. Diese soll die einfache und

intuitive Steuerung der Wii-Remote unterstützen. Ebenfalls erhoffe ich mir durch die Integration der Kamera-Steuerung und der teambasierten Einbindung der Wii-Remote den Einstieg und den Umgang mit der Ogre3D-Engine.

Lukas Kalinowski:
Ein Lernziel ist es eine eigne Gameloop zu entwickeln, die unabhängig von der Rechnerarchitektur stabil und flüssig läuft und alle Komponenten verwaltet, sowie die Verwendung von Threads für diese Aufgabe.
Außerdem erhoffe ich mir die Einbindung der Wii-Remote, sowie das Auslesen der Daten um den Charakter steuern zu können, zu erlernen. Weiterhin möchte ich die Anvisierung auf dem Bildschirm über die Infrarotschnittstelle der Wii-Remote integrieren. Das Überblenden von verschiedenen Animationen zur Laufzeit je nach Aktion ist auch noch ein persönliches Lernziel. Zuletzt steht noch das Projektmanagment für mich als Lernziel, dies beinhaltet das Einbinden der Komponenten, sowie die Integration der Arbeiten der Teammitglieder.

1.3 Ausgangslage

Als Grundlage des Projekts dient das Spiel „*Flubbers*“ aus der Bachelorarbeit von Lukas Kalinowski mit dem Titel „Spiele in einer 3D-Welt – Von der Idee bis zur Realisierung“. Weiter ist die WiiUse-Bibliothek [WIIUSE] für die Einbindung des Wii-Controllers vorhanden. Ebenfalls steht der Leveleditor VWE, nach Quelle [VWE], zur Verfügung.

1.4 Anforderungsanalyse

Eine grundsätzliche Anforderung an Interaktive 3D-Computer-Grafik ist ihre Echtzeitfähigkeit, dies setzt eine flüssige (performante) Darstellung der Inhalte und verzögerungsfreie Reaktionen auf Eingaben voraus. Im Allgemeinen kann bei einer Bildwiederholrate von 30 Bildern pro Sekunde von einem flüssigen Ablauf ausgegangen werden.
Die Anforderung an den Charakter ist, dass er auf aktuellen Technologien und Konzepten, wie beispielsweise Shader und Animationsblending entwickelt wird.
Eine weitere Anforderung ist die benutzerfreundliche und intuitive Steuerung der Anwendung mit Hilfe der Wii-Remote.

1.5 Projektstrukturplan und Aktivitätenzeitplan

Der Projektstrukturplan für dieses Projekt befindet sich im Anhang und auf dem Datenträger im Ordner Projektplan.

2 Projektbericht

2.1 Arbeitsverteilung

Um die Arbeitsverteilung innerhalb des Projektes zuteilen zu können werden in der folgenden Tabelle die Namen und das mitwirken im Arbeitspacket durch die Kapitelnummer gekennzeichnet.

Name	**Kapitelnummer**
Frieß	1, 1.1, 1.2, 1.3, 1.4, 1.5, 2.1, 2.2, 2.3, 2.4, 2.4.1, 2.4.2, 2.4.3, 2.4.4, 2.4.5, 2.4.6, 2.4.7, 2.4.8, 2.4.9, 2.6.5, 3, 4, 5, 5.1, 5.2, 5.2, 5.4
Kalinowski	1, 1.1, 1.2, 1.3, 1.4, 1.5, 2.1, 2.2, 2.3, 2.6, 2.6.1, 2.6.2, 2.6.3, 2.6.4, 2.6.5, 2.6.6, 3, 4, 5, 5.1, 5.2, 5.2, 5.4
Gohlke	1, 1.1, 1.2, 1.3, 1.4, 1.5, 2.1, 2.2, 2.3, 2.5, 2.5.1, 2.5.2, 2.5.3, 2.5.4, 2.5.5, 3, 4, 5, 5.1, 5.2, 5.2, 5.4

2.2 Prinzipielle Vorgehensweise

Im weiteren Verlauf wird ein Charakter modelliert, texturiert und animiert und später in geeigneter Form in das Spiel eingebunden. Hierfür wird im Vorfeld auf Shader-Technologien wie Normal-Mapping, Diffuse-Maps, Ambient-Occlusion-Maps und Specular-Maps geachtet.
Zur Animation des Charakters wird ein Skelett erstellt und mit Weight-Painting an das Polygonmodell gebunden. Die Animationen beschränken sich auf grundsätzliche Bewegungen zur Navigation in der Welt.
Der fertige Charakter wird durch Exportereinsatz in die Ogre-Engine eingebunden. Anschließend werden die Wii-Eingabeaktionen auf den Charakter „gemappt“. Das heißt der Charakter wird über die Wii-Controller gesteuert und es besteht die Möglichkeit über die Infrarotkamera auf dem Bildschirm zu zielen und Geschosse abzufeuern.
Ein weiteres Ziel ist eine „Third-Person-Kamera“, die in einer dynamischen Art und Weise den Spieler verfolgt. Abschließend wird eine Umgebung für das Spielszenario mit dem Leveleditor erstellt in dem die entwickelten Komponenten getestet werden können.

2.3 Aufsetzen der Arbeitsumgebung

Zunächst wird die Arbeitsumgebung eingerichtet. Dies beinhaltet das Installieren des Gestaltungswerkzeuges *Blender 2.49* (zuzüglich Exporter *OgreXmlConverter* und *Python 2.6*) und *Gimp 2*. Des Weiteren wird die Grafikengine *Ogre 3D 1.6.2* installiert und der in früheren Projekten entwickelte Leveleditor (*Virtual World Editor*) von Lukas Kalinowski und Daniel Bies wird für die Levelgestaltung eingerichtet. Zur Entwicklung wird *Eclipse mit C++ Plugin* und den benötigten Komponenten eingerichtet. Dabei übernehmen die folgenden Bibliotheken die folgenden Aufgaben: Für die 3D-Visualisierung kommt die *Ogre-Engine* zum Einsatz. Die physikalischen Effekte werden von der *OgreNewt* Bibliothek übernommen. Die Audio-Unterstützung wird mit Hilfe von *OgreAL* realisiert. Zur Verwendung des *WII-Controllers* werden Hardwarekomponenten wie Bluetooth-Dongle, IR-Leiste und der Controller an sich, sowie die Softwarekomponente Bluetooth-Treiber (*Bluesoleil*) und die *Wiiuse-Bibliothek* eingesetzt.

2.4 Charakter Erstellung

Dieses Kapitel erläutert den Erstellungsprozess eines Next-Gen Charakters. Vorweg sollen zunächst stichpunktartig die nötigen Schritte aufgelistet werden. Der Workflow teilt sich in folgende Punkte auf:

- Ideensammlung und Entscheidung
- Modellieren eines Base Mesh durch Box-Modeling
- UV-Unwrapping des Base Mesh
- Virtual Sculpting
- Extrahieren von Oberflächeninformationen über eine Normalmap
- Mahlen einer Farbtextur
- Erzeugen von Schatten, Glanz und Leucht- Texturen für den Shader
- Programmieren des Shader und erstellen des Ogre-Materials

Der notwendigen Schritte und Funktionen dieser 7 Schritte werden nun in den folgenden Abschnitten genauer beleuchtet.

2.4.1 Ideensammlung und Entscheidung

Am Anfang eines Game Charakters steht eine Idee. Wir haben uns maßgeblich an aktuellen Game-Charakteren orientiert und inspirieren lassen. Die Folgende Abbildung sammelt die Inspirationsquellen. Internet Communitys oder Foren wie www.cgsociety.org, www.pixologic.com, *www.dominancewar.com*, oder www.3d.sk eignen sich sehr gut um sich inspirieren zu lassen.

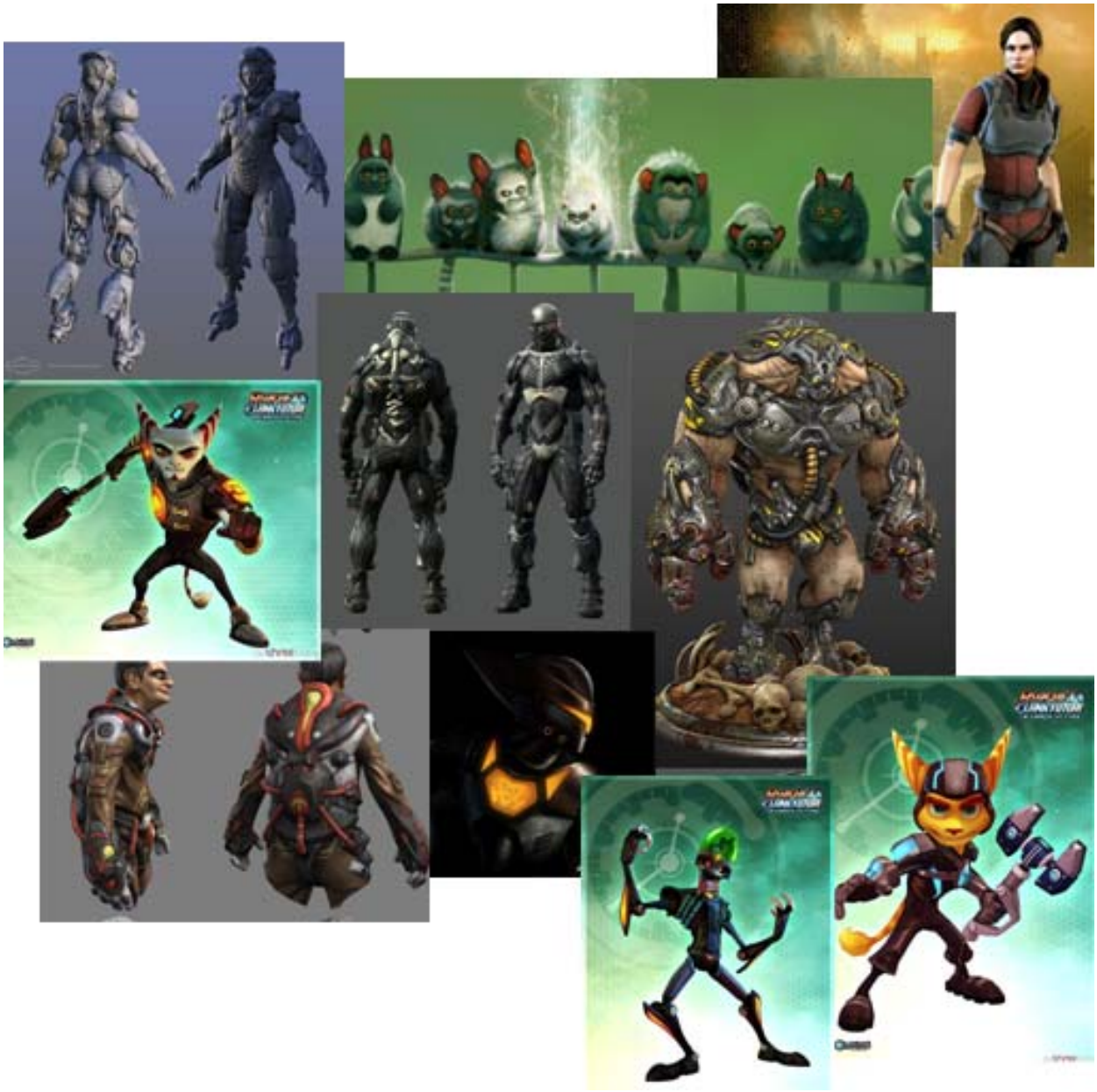

Abbildung 1: Inspirationsquellen

Auf Basis dieser Grundlagen entwickelte sich ein ungefähres Bild des Charakters. Die Idee war ein kleines haariges, nagerartiges Alien, das in einem kraftvoll und stärke ausstrahlenden Roboter-Anzug sitzt und diesen steuert (siehe Abbildung 2 links). Zunächst

wurde diese erste Idee mit Bleistift und Papier festgehalten, wie man in Abbildung 1: Inspirationsquellensehen kann. Die Figur wurde auf den Namen „Eddy“ getauft. Der Anzug in dem der Charakter sitzt nennt sich „Robo-Suit“.

Abbildung 2: Skizze (links); Unterteilung (rechts)

2.4.2 Vorbereitung

Bevor man aufbauend auf dem Konzept mit der Modellierung beginnt, kann es sehr hilfreich sein, sich vorher noch weitere Gedanken über den Aufbau des Charakters zu machen. Auf Basis der Konzeptzeichnung kann beispielsweise der Charakter in kleine Stücke zerteilt werden (siehe Abbildung 2 rechts). Es ist wesentlich einfacher viele kleine Modelle zu modellieren und diese am Ende zu verschmelzen, als von Anfang an ein großes Modell zu bauen. In unserem Fall kann beispielsweise der Roboter-Rumpf, die Arme, die Beine, die Alien-Arme, der Alien-Kopf und das Steuerkreuz separat modelliert werden.

2.4.3 Modeling

Der Modellierungsschritt ist dazu da eine ungefähre Grundstruktur des Charakters zu erzeugen. Beim „Virtual Sculpting“ wird diese dann optimiert und verfeinert. Da das modellierte Mesh eine Basis zum Sculpting darstellt, wird es auch als „Base Mesh“ bezeichnet. Wie oben schon erwähnt, setzt man das Base Mesh aus kleineren Meshs zusammen. Dies erlaubt mehr Freiheit in der Anordnung und der Modellierung der Körperteile. Das Tool unserer Wahl für diese Arbeit ist Blender 3D [BLEND]. Welche Modellierungs-Technik der Modellierer nutzt ist Geschmacksache. In der Regel wird das sogenannte Box-Modelling verwendet. Hier wird mit einer sehr einfachen Geometrie wie

einer Box begonnen. Durch Hinzufügen von Flächen (Extrudieren) und Transformieren von Vertex-Gruppen formt man das Objekt. Ist ein Objekt symmetrisch so kann mit einer Mirror-Funktionalität gearbeitet werden. Diese spiegelt die Veränderungen von einer Seite auf die andere.

Bereits bei diesem sehr groben Schritt muss extrem sauber und genau gearbeitet werden, da in späteren Schritten die Unterteilungsfunktion Subsurface angewandt wird. So muss darauf geachtet werden, dass die Topologie des Meshs möglichst aus quadratischen Polygonen, also Polygonen mit vier Vertices, besteht. Die Anordnung der Quadrate sollte in einem natürlichen Fluss folgen. Beispielsweise versucht man bei menschlichen oder tierischen Objekten die Polygone so anzuordnen, dass sie den Muskelgruppen folgen. Mit Hilfe eines sauberen und guten „Edgeflow“ kann man ein Mesh extrem optimieren und somit mit weniger Polygonen auskommen. Ebenfalls erscheint das Mesh dann weniger kantig, was bei organischen Objekten stören kann. Das Finden und Beibehalten eines gelungenen Edgeflows erfordert viel Übung.

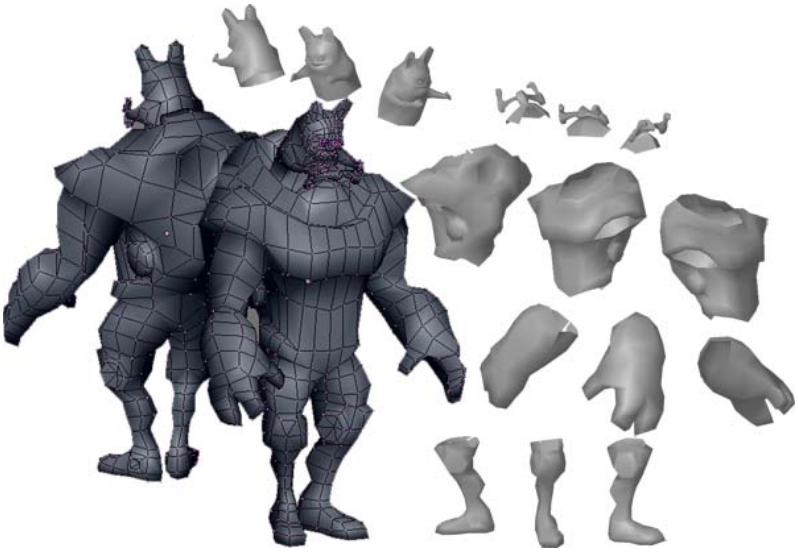

Abbildung 3: Bestandteile des Base Meshs

Im Fall unseres Charakters wurde mit dem Rumpf des Roboters angefangen, dann die Beine und Arme hinzugefügt. Das Alien wurde separat auf die gleiche Weise modelliert. In der Abbildung 3 sieht man die einzelnen Meshs aus verschieden Blickwinkeln, die Teile vor der Verschmelzung und dann das fertige Base-Mesh.

2.4.4 UV-Unwrapping

Bevor man nun anfängt die Geometrie des Charakters weiter zu verfeinern, kommt ein Schritt der nicht undbedingt etwas mit der Modellierung zu tun hat, aber wichtig für die weiteren Schritte ist. Gemeint ist das „UV-Unwrapping“. Unter UV-Unwrapping versteht man,

das Zuordnen der 3D Vertices des Meshs zu einer UV-Koordinate auf einer 2D Fläche, sprich der Textur. Es ist kompliziert ein Bild passend auf einen 3D-Körper zu legen. Ein bekanntes Beispiel ist das Bekleben einer Kugel mit der Weltkarte. 3D-Progamme bieten verschiedene Verfahren an, dieses Problem zu lösen. Das Problem ist, dass diese Zuweisung mit der Komplexität des Meshs schwieriger wird. Die Textur muss mehr gezogen werden um alle Punkte abbilden zu können. Aus diesem Grund kann der Modellierer Schnittkanten angeben. Diese sagen dem Programm, an welchen Stellen es das Mesh zerschneiden darf. Dort löst es die Verbindung und projiziert die Einzelteile auf eine 2D-Fläche.

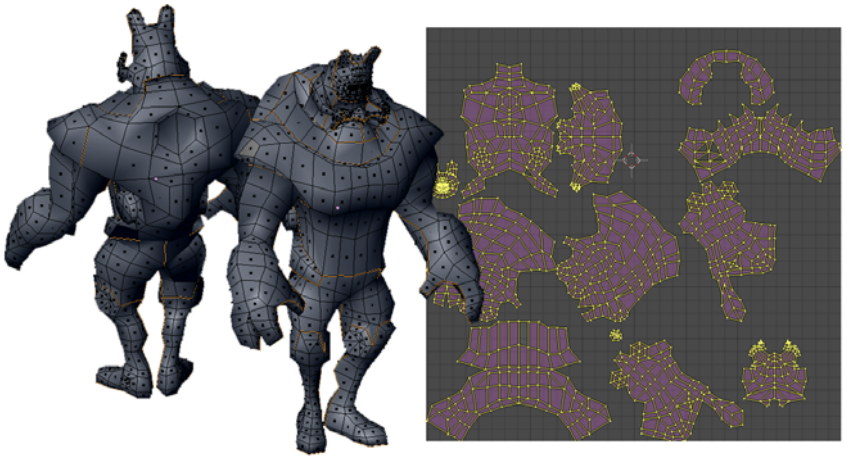

Abbildung 4: Automatisch erzeugtes UV Layout

Nun liegt es am Modellierer, wie die projizierten Schnipsel angeordnet werden sollen. Diese Anordnung nennt man UV-Layout. Jeder einzelne Schnipsel kann nun wieder in der 2D-Ebene verschoben, skaliert oder rotiert werden. Um Texturplatz zu sparen, kann man beispielsweise jeweils Beine und Arme aufeinander legen, da diese in der Regel symmetrisch sind. Im Gesichtsbereich ist eine Symmetrie nicht zu empfehlen, weil diese sehr schnell auffällt und unnatürlich erscheint. Es ist auch üblich, dass für detailreichere Regionen mehr Texturplatz verwendet wird, um eine höhere Auflösung an diesen Stellen zu erreichen. Um die Platzverteilung und Lage der Textur auf dem Modell anzeigen zu lassen, wird in diesem Stadium ein farbiges Schachbrettmuster als Textur genutzt (siehe Abbildung 5).

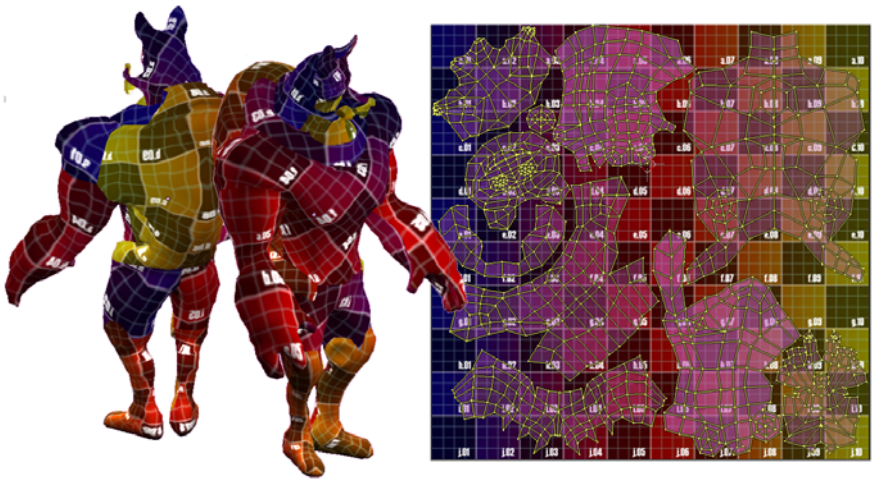

Abbildung 5: Manuell erzeugtes Layout mit Farbiger Test-Textur

2.4.5 Virtual Sculpting

Im Gegensatz zu dem im Abschnitt 2.4.3 erläuterten Box-Modeling, basiert „Virtual Sculpting“ nicht auf direkten Mesh Manipulationen, sondern verformt das Base Mesh auf einem intuitiveren Weg. Das Prinzip von Virtual Sculpting wird bereits exakt durch seinen Namen beschrieben. Man kann sich das Base Mesh als Rohmodell einer Knetfigur vorstellen. Durch mehrfaches unterteilen der Polygone über einen Subdivision-Modifikator wird die Auflösung des Base Meshs erhöht. Der Modifikator zerteilt jedes Polygon in vier kleinere. Durch mehrfache Durchführung dieser Operation erlangt man ein Mesh mit einer sehr hohen Polygonanzahl und somit auch Auflösung. Damit dies auch garantiert ist, muss, wie auch schon angesprochen, bereits beim Modellieren des Base Meshs sauber gearbeitet werden (siehe Abschnitt 2.4.3). Auf diesem hoch aufgelösten Mesh, auch High Poly-Mesh genannt, können nun mit Hilfe verschiedener Werkzeugspitzen Details in das Mesh eingearbeitet werden. Je höher die Auflösung ist, desto feinere Details lassen sich herausarbeiten. Die Werkzeugspitze wird mittels einer Height-Map realisiert. Anhand ihrer Höhen-Informationen und der Bewegung der Maus verformt das Programm dann das darunterliegende Mesh und verändert somit die Geometrie. In Abbildung 6 sieht man einige Beispiele von Height-Maps. Im Grunde bestehen sie nur aus einem Schwarz-Weiß Bild. Die weißen Regionen entsprechen den hohen Regionen.

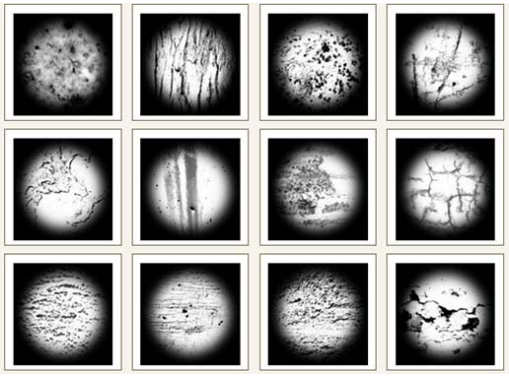

Abbildung 6: Werkzeugspitzen zum Skulpting

Virtual Sculpting wird bereits in vielen gängigen 3D-Paketen angeboten. Wir haben uns allerdings für das Tool „ZBrush“[ZBRUSH] entschieden, da es auf diese Aufgabe spezialisiert ist und als Vorreiter für diese Technologie gilt. Den gesamten Sculpting-Prozess des Charakters zu erläutern würde den Umfang des Berichts überschreiten, aus diesem Grund wird in der Abbildung 7 die Details des fertig „gesculpteten“ Charakters zusammengefasst. Die grundlegenden Funktionsweisen dieses Prozesses sind in [AW08] und [JW08] sehr gut erläutert.

Abbildung 7: Fertig gesculpteter Charakter

Da das Hi-Poly-Mesh auf dem Base Mesh basiert, wird das Base Mesh ebenfalls verformt. Ist der Skulpting-Prozess abgeschlossen und man entfernt wieder die Subdivision - Modifikatoren kann man erkennen, dass sich dieses Mesh von dem Base Mesh unterscheidet. Dieses neue Mesh wird als Low-Poly-Mesh bezeichnet und entspricht dem Hi-Poly-Mesh nur in einer geringeren Auflösung. Dieses Mesh ist die Grundlage für die Texturgenerierung und 3D-Anwendung. In der Regel würde nun in hochqualitativen Produktionen auf Basis der Hi-Poly-Meshs separat ein neues Low-Poly-Mesh modelliert werden, welches sich optimal an die Details anschmiegt und somit eine neue Topologie des Meshs definiert. Auf diese „Retopology“ wurde im Rahmen dieses Fachprojektes allerdings verzichtet, da sie mit extrem viel Zusatzaufwand verbunden wäre.

2.4.6 NormalMap und Ambient Occlusion

Das durch das Virtual Sculpting erzeugte extrem große Mesh kann selbstverständlich nicht in einer Echtzeit-3D-Anwendung verwendet werden, weil die Polygonanzahl das zu viel Rechenleistung in Anspruch nehmen würde. Wie im vorherigen Abschnitt schon erläutert, wird das Low-Poly-Mesh für die Charakter-Darstellung in der Anwendung verwendet. Da es aber nicht so fein aufgelöst ist, um alle Details des Sculpting-Schrittes zu zeigen, wird hier eine Technik namens Normal-Mapping angewandt, mit derer Hilfe die Details des Hi- auf das Low- Poly-Mesh übertragen werden können. In einer Normalmap werden die geometrischen Details in Form von Normalen in eine Textur geschrieben. Diese Normalen werden dann bei der Lichtberechnung verwendet um die Details in der Schattierung des Objektes zu Visualisieren, anstatt in der Geometrie. Die Details der Oberfläche bleiben somit optisch erhalten. Sichtbar ist das niedrige Detail lediglich an der Silhouette, die noch immer der des niedrig aufgelösten Modells entspricht. Somit kann ein großer Detailreichtum in Schattierungen erzielt werden, ohne die Zahl der Polygone zu erhöhen (Siehe Abbildung 8).

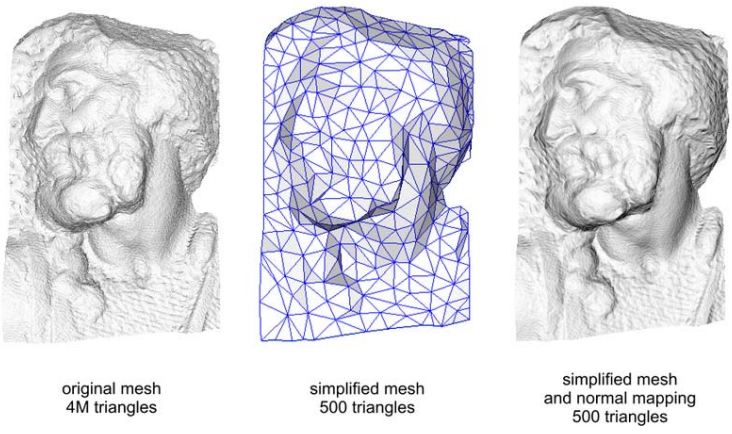

Abbildung 8: Vergleich von Normalmaps

Die Normalmap kann in der Regel nur computergestützt generiert werden. Diese Aufgabe übernimmt das Skulpting-Tool. Es berechnet anhand der Unterschiede zwischen Hi- und Low-Mesh die Normalmap. In Abbildung 9 ist das Mesh mit angewandter Normalmap zu sehen. Dort erscheint der Charakter wesentlich detaillierter als z.B. in Abbildung 4 obwohl beide Objekte auf der gleichen Geometrie basieren.

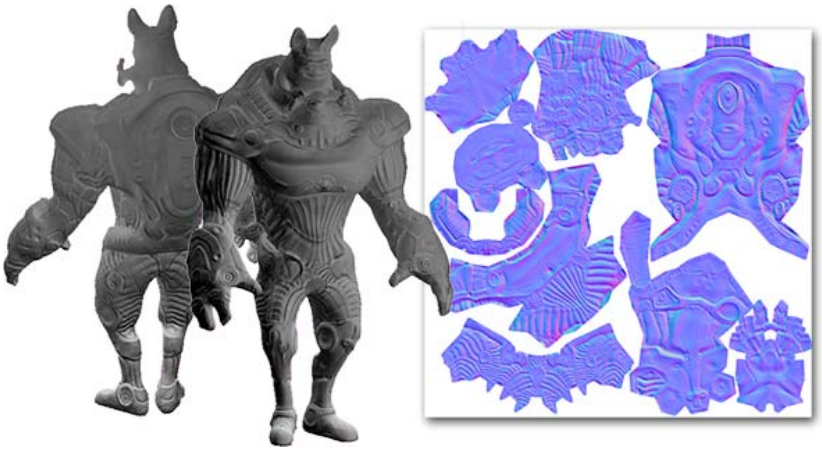

Abbildung 9: Normalmap und dessen Anwendung auf das Objekt

Ein anderes sehr subtiles, den Realismus aber extrem steigerndes Element, ist „Ambiente Occlusuion“ (AO). In unserer alltäglichen Umgebung existiert das Phänomen, dass wenn zwei beliebige Objekte aufeinander liegen ein weicher Schatten an der Auflagekante zwischen den Objekten entsteht. Software-Pakete wie Blender können diese AO simulieren und in eine Textur rechnen. Nutzen wir nun das Hi-Poly-Mesh zur Berechnung der AO und legen die Textur über das Low-Poly-Mesh, so wirkt das Mesh schon viel detailreicher, obwohl die Details überhaupt nicht in der Geometrie existieren. Diese subtilen Schatten sind fest in die Textur gebacken und werden nicht durch die Beleuchtungsrechnung beeinflusst was aber gar nicht oder kaum auffällt.

Abbildung 10: Mesh mit Ambinet Occlusion Textur

2.4.7 Texture Painting

Bis zum jetzigen Zeitpunkt ist der Charakter zwar detailreich aber farblos. Mit Hilfe von Blender ist es möglich ein Texturierungskonzept namens Poly-Paint zu nutzen. Hierfür müssen die UV-Koordinaten des zu texturierenden Objekts entsprechend aufgespannt sein, wie es in Abschnitt UV-Unwrapping erläutert wurde. Poly-Paint bedeutet, dass Farben direkt auf das Mesh, also die Polygone, gemalt werden kann. Diese Eigenschaft ist sehr hilfreich, weil man mit aktiviertem Normalmap und AO die unterschiedlichen Bereiche einfach mit einem Pinsel einfärben kann. Synchron zum Bemalen der Textur sieht man in einem anderen Fenster die entstehende Textur. Darüber hinaus kann auch in diese Textur gezeichnet werden und die Änderungen sind direkt auf dem Mesh sichtbar. Leider ist dieses Werkzeug in Blender noch nicht sehr ausgereift. So hat man noch keine Modifikatoren um spiegelbildlich zu arbeiten, beziehungsweise die Textur in mehreren Ebenen zu organisieren. Diese Funktionen wurden allerdings schon angekündigt und werden in kommenden Blender Versionen implementiert sein. Um diese fehlenden Funktionen ausgleichen zu können, wurde für das Projekt ein Bildbearbeitungsprogramm hinzugezogen. Durch Einsatz dieser Programme ist letztendlich die in Abbildung 11: Mesh mit diffuser Textursichtbare Farbtextur, auch Diffusemap genannt, entstanden.

Abbildung 11: Mesh mit diffuser Textur

2.4.8 FX-Maps

Auf Basis der Diffusemap ist es nun möglich weitere Maps zu erzeugen, die für den Game-Charakter interessant sind. Die erste Effekt-Textur, die verwendet wird, nennt sich Glow- oder Emission-Map. Sie gibt an, in welchen Bereichen der Charakter leuchtet, beziehungsweise den Anschein macht zu leuchten. An dieser Stelle ist nur interessant, wie

die entsprechende Textur für diesen Effekt erzeugt wird. Ziel ist es, die gelben Bereiche, die auch schon in der Diffusemap zu sehen sind zu extrahieren. Das verwendete Bildbearbeitungsprogramm bietet die Möglichkeit ein Color-Pick auszuführen. Dadurch werden alle Bereiche in einem Farbintervall selektiert. Maskiert man nun die nicht selektierten Bereiche hat man das Ziel erreicht.

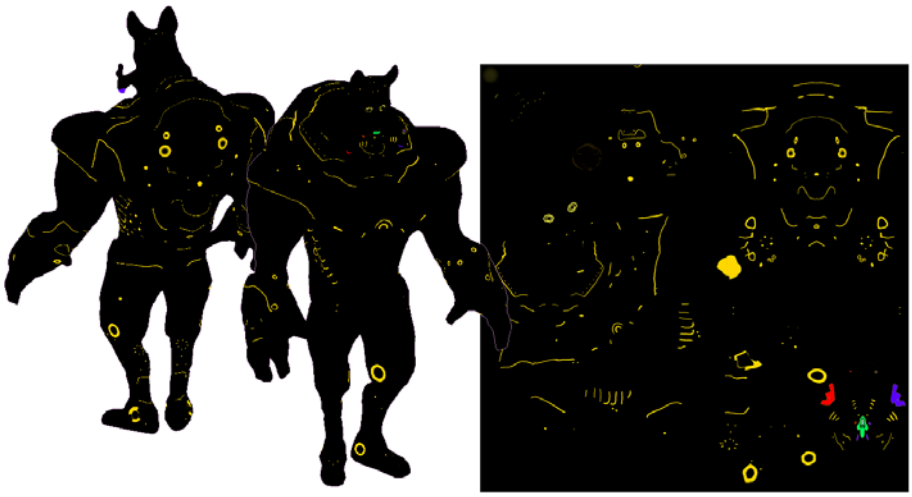

Abbildung 12: Mesh nur mit Glow Textur

Der zweite Effekt, der verwendet wird, basiert auf einer Schwarz-Weiß-Textur. Anhand der Graustufen kann der Grad der Reflektion von Materialien ausgelesen werden. Helle Bereiche reflektieren das ankommende Licht stark, dunkle Bereiche hingegen absorbieren Licht. In Abbildung 13 wurde die Textur auf das Mesh angewandt. Das hellgraue Metall aus der Diffusemap bekommt durch die „Specularmap“ einen sehr hohen Glanz verliehen. Dadurch werden diese Stellen abhängig von der Aug- und Lichtposition heller.

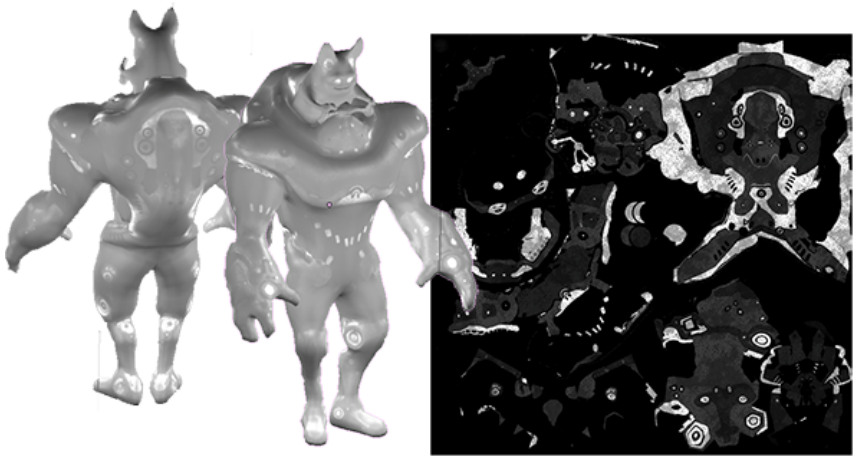

Abbildung 13: Mesh mit Glanztextur

2.4.9 Shader Programmierung

Der Shader, welcher alle oben erstellten und erläuterten Maps kombiniert, ist in der Shader-Sprache HLSL (High Level Shader Language) geschrieben und stammt aus dem Ogre3D Forum [FXM]. Um den Shader in die Ogre3D-Engine integrieren zu können müssen im Material-Script des Objektes die Quellen der Shader-Programme und ihre Parameter definiert werden. Der dazugehörige Quelltext ist kommentiert. Grundsätzlich besteht ein Shader immer aus zwei getrennten Programmen: Dem Vertex- (.vert) und dem Fragment-Shader(.frag). Der Vertex-Shader operiert auf den Vertices des Objektes und kann dort Lichtberechnungen oder Koordinatentransformationen vornehmen. Der Fragmentshader nutzt die Berechnungen des Vertex-Shaders und berechnet die Farbe jedes einzelnen Pixels (Fragment) des Objektes.

Soll nun der Shader von der Anwendung beeinflusst werden, spricht die Anwendung das Material-Script an. Dort können Zugriffsparameter gesetzt werden, welche dann in den Shader gespeist werden. In unserem konkreten Fall sollte ein Pulsieren der Glowmap realisiert werden. Hierzu wurde in dem Material-Script die Variable glowIntensity angelegt, auf die der Fragment-Shader zugreift. Abhängig vom Wert von glowIntensity wird die Glowmap mehr oder weniger intensiv gezeichnet. Von der Anwendung werden dann lediglich die unten aufgeführten Zeilen aufgerufen. Damit verschafft sich die Anwendung Zugriff auf den Parameter glowIntensity im Material-Skript und weist ihm den Intensität-Wert zu, welcher zyklisch über eine Sinus-Funktion im Bereich von 0 – 2 PI variiert.

```
psParams =
     pEddyMaterial->getTechnique(0)->getPass(0)->getFragmentProgramParameters();

psParams->setNamedConstant("emissionIntensity",
                           (Ogre::Math::Sin((Ogre::Radian)rad)));
```

Zuletzt ist zu sagen, dass die Thematik „Shader“ sehr komplex und umfangreich ist und aus diesem Grund an dieser Stelle auf fremden Code zurückgegriffen wurde, welcher für unsere Zwecke verändert wurde. In Abbildung 14 ist der fertige Charakter zu sehen.

Abbildung 14: Geshadeter Charakter

2.5 Charakter Animation mit Blender

Bei der Wahl eines geeigneten Animationswerkzeugs standen *Cinema 4D* und das freie 3D Paket *Blender* zur Wahl. Die Entscheidung fiel zugunsten von *Blender* aus, da sich das Programm am besten in den Produktionsprozess zwischen Modellierung, Texture-Mapping und dem Export zu integrieren schien. Als Modellierungswerkzeug und für das Texture-Mapping war *Blender* bereits eingeplant. Darüber hinaus lagen die meisten Erfahrungen sowohl im Umgang mit dem Werkzeug, als auch mit einem speziellen *Blender-to-OGRE Format-Exporter* vor, so dass es vernünftiger erschien, das Eingliedern eines weiteren Werkzeugs zu vermeiden. Insbesondere der Austausch, von in unterschiedlichen Formaten vorliegenden 3D-Objekten, birgt immer die Gefahr von Fehlern, Inkompatibilitäten und anderen Problemen. Weiter hätte bei einem Einsatz von *Cinema 4D* als einziger stabiler Exporter nur ein proprietäres Exportwerkzeug zur Verfügung gestanden.

Für das Projekt wurde ausschließlich die stabile Version 2.49b eingesetzt und mit dem Plugin *Blender Exporter* versehen. Für das auf Python basierende Plugin war außerdem eine *Python 2.54* Installation Voraussetzung.

2.5.1 Prinzipielle Vorgehensweise beim Animieren

Neben dem Wissen um die programminternen Konzepte des Animationswerkzeugs, sind einige grundsätzliche Vorüberlegungen zum Animieren eines 3D Echtzeit-Charakters nötig.
Das übliche Vorgehen im Animationsprozess gliedert sich in fünf Arbeitsschritte:

- Erstellen eines Skeletts
- Verbinden des Skeletts mit einem Mesh
- Wirkungsbereich und -intensität der Knochen auf das Mesh einstellen
- Animieren des Charakters
- Exportieren in das Format der Zielapplikation

Hinter den ersten drei Arbeitsschritten verbirgt sich ein Prinzip, dass dem schichtweisen Aufbau eines Menschen aus der Realität nachempfunden ist. Es existieren ein (reduziertes) Knochengerüst und eine darüber liegende Hautschicht, die mit dem Gerüst verbunden ist.
Anders als in der Realität verrichten die Knochen bei der 3D Charakter-Animation gleichzeitig die Arbeit der Muskeln. Die Hautschicht (das Mesh) verformt sich abhängig von den Bewegungen der Knochen. Wie stark sich das Mesh verformt hängt von dem Drehpunkt des jeweiligen Knochens und dessen Einflussintensität auf die Meshabschnitte ab.
Beim *Weight-Painting* werden für jeden Knochen der Einflussbereich und die Einflussstärke festgelegt, indem Farbmarkierungen auf das Mesh aufgemalt werden. Die Intensitätsskala in Blender ist in 25 Prozent-Schritte aufgeteilt. Je kleiner der Wert ist, desto wenige stark wird das Mesh von einem ausgewählten Knochen beeinflusst.
Im Anschluss an die Gewichtung können weitere Hilfsmittel in das Modell integriert werden, die z.B. Bewegungsräume einschränken oder ausgehend von einem Knochen die Stellung vorhergehender Knochen berechnen können. Diese „Sonderausstattungen“ werden je nach Vorhaben, Geschmack und Verwendungszweck hinzugefügt. Anschließend ist das Modell fertig „geriggt“ und damit bereit für die Animation.
Bevor mit dem Animieren begonnen wird, werden alle Bewegungen notiert, die für den Charakter erstellt werden müssen. Im Falle von 3D Echtzeit Anwendungen müssen die Animationen in der Regel in zyklischer Form vorliegen, so dass die in der Schleife abgespielten Animationen immer wieder nahtlos von vorne beginnen. Das dies so ist, ist der Abhängigkeit von den Eingaben des Benutzers geschuldet, denn hält der Benutzer eine Taste gedrückt, so muss sich eine Bewegung entsprechend lange wiederholen können. Das Gleiche gilt selbst für Bewegungen, die nicht so sehr eine nahtlose Wiederholung erfordern, wie z.B. ein Schwerthieb. Sollen Animationen „übergeblendet“ werden, also nicht zusammengehörige Bewegungen weich ineinander übergehen, ist darauf zu achten, dass sich diese Bewegungen nicht zu sehr unterscheiden. Ebenfalls wichtig ist es darauf zu

achten, dass kombinierbare Animationen nicht zu komplex ausgearbeitet werden. So lassen sich z.B. Beinbewegungen und Oberkörperbewegungen voneinander entkoppeln.
Ist es für die Anwendung wichtig, dass die Bewegungen schnell wechseln, sollten möglichst kurze Einzelanimationen die Regel sein, um mehr Kontrolle bei den Übergängen zu behalten. Das Normen von zu kombinierenden Animationsclips auf eine einheitliche Länge, kann ebenfalls empfehlenswert sein.
Liegen alle Animation vor, muss das Modell mit Hilfe eines Exporters in das Zielformat überführt werden. Dazu werden spezielle Dateien erzeugt, welche die Geometriedaten, Animationsdaten und Materialdaten enthalten. Diese sind nötig, damit eine Grafikengine das Modell mit seinen Texturen und Animationen darstellen kann.

2.5.2 Aufbau eines Skeletts

Das Entwerfen des Skeletts wird meistens durch das 3D Mesh bereits vorgegeben. Handelt es sich bei dem Modell um einen zweibeinigen Charakter, gibt es im Grunde nur zwei verschiedene Möglichkeiten das Skelett aufzubauen [BLEND2].
Der Hauptunterschied zwischen diesen beiden Methoden besteht dabei in der unterschiedlichen Beinkonstruktion. Diese entscheidet darüber, ob sich der Charakter vogelähnlich oder menschenähnlich fortbewegt. Die vogelähnliche Variante wird häufig bei Fantasy und Science Fiction Figuren eingesetzt.

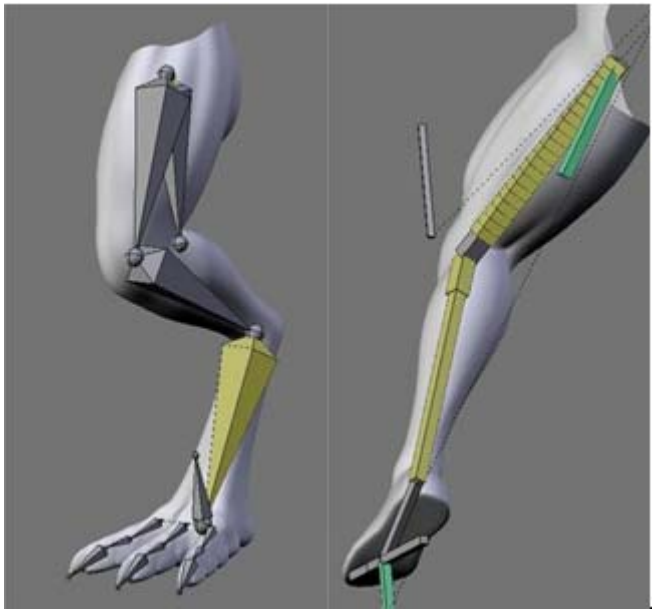

Abbildung 15 Links ist der typische Aufbau eines Vogelbeins zu sehen.
Auf der rechten Seite der typische Aufbau eines menschlichen Beins.

Ist die Entscheidung für einen Skeletttyp getroffen, gilt es die Granularität des Knochengerüsts zu bestimmen. Ziel ist es dabei, möglichst alle gewünschten Bewegungen bei kleinstmöglicher Knochenanzahl darstellen zu können, um den Berechnungsaufwand

gering zu halten. Weiter sollten die Knochen so gesetzt werden, dass unerwünschte oder unnatürliche Bewegungen nach Möglichkeit gar nicht erst ausgeführt werden können. Dies erleichtert das Animieren zu einem späteren Zeitpunkt, da sich unerwünschte und meist nicht so einfach zu lokalisierende Effekte, nicht so leicht einstellen können.

Beim Einsetzen der Knochen sollte beachtet werden, dass eine Knochen-Hierarchie entsteht, bei der untergeordnete Knochen von den darüber liegenden mit bewegt werden. Wird die richtige Reihenfolge bereits beim Aufbau des Knochengerüsts bedacht, entfällt das spätere Umorganisieren der Hierarchie.

Da demnach der erste Knochen gleichzeitig der Wurzelknochen ist, wird üblicherweise mit dem untersten Wirbel begonnen und sich zunächst bis zu den Schultern emporgearbeitet. Arm- und Bein-Bones sollten mit der Spiegelfunktion erzeugt werden, diese spart Zeit und ermöglicht eine gute Gleichmäßigkeit.

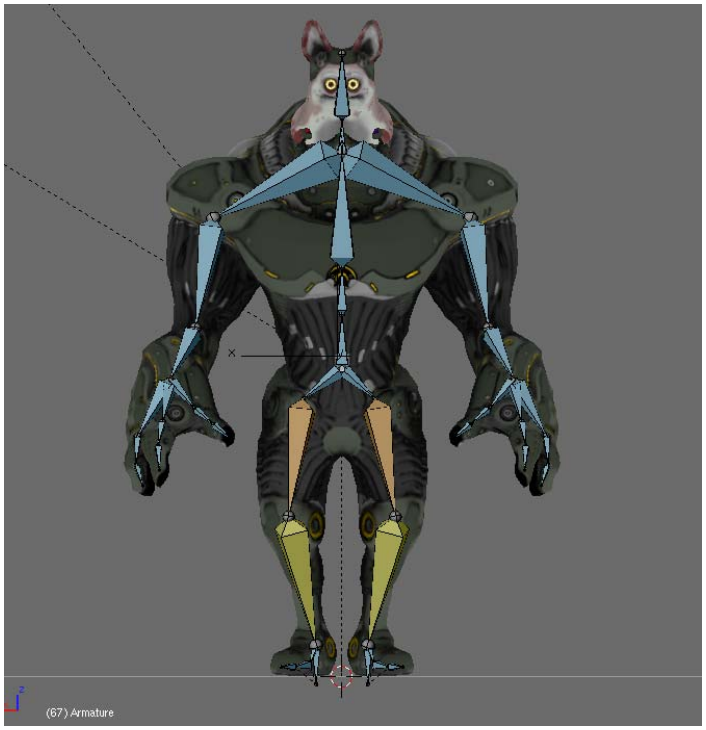

Abbildung 16: Die Schienbeine (gelb) und die Oberschenkel (orange) gehören zu einer IK-Kette.

Das Skelett für das Projektmodell wurde nach menschlichem Vorbild aufgebaut. Die Klauen sind relativ detailliert, um Handbewegungen, wie z.B. ein Faustballen, zu ermöglichen. Der Brustbereich wird nur durch einen Wirbelknochen abgedeckt, um die typische Steifigkeit des Brustkorbs im Vorfeld festzulegen.

Abbildung 17: Ein kleiner Knochen in der Mitte des Steuers wird für Lenkbewegungen des Aliens verwendet.

Ein einzelner kleiner Knochen wird in der Mitte des Steuers platziert, an den später die Arme und der Oberkörper des Aliens und das vollständige Steuer gebunden werden. Dadurch kann eine Bewegung realisiert werden, bei der das Alien nach links oder rechts lenkt und sich dabei mit in die Kurve legt. Das Alien besteht nur aus zwei Knochen, einem für den Körper und einem für den Kopf.

Bei der Animation für eine Echtzeit-Anwendung sollte besonders auf eine konsistente und einem Konventions-Schema entsprechenden Benennung der Knochen Wert gelegt werden. Eingebaute Leerzeichen können beispielsweise Fehler beim Exportieren verursachen. Abgesehen davon, erleichtert es die eigene Arbeit und die der Kollegen. Da es mehrere verschiedene Schemata gibt, an denen eine Orientierung möglich ist, ist es in erster Linie wichtig, ein einziges durchgängig zu verwenden. In diesem Projekt wurde der Name ggf. gefolgt von einer Zahl, z.B. für Fingerglieder, einem Punkt und einem Kleinbuchstaben für die Seite angegeben. Bei Mehrteiligen Bezeichnern wurden z.T. Unterstriche, nie aber Leerzeichen verwendet.

Für die Animation der Beine wurde *Inverse Kinematik* eingesetzt. Dies hat einige Vorteile für den Animator, kann aber gleichzeitig zu Problemen beim Exportieren führen. Ein Besonderer Vorteil beim Animieren ist, dass sich der Bodenkontakt sehr einfach einhalten lässt. Die IK-Kette schränkt die Bewegung bereits so ein, dass sie das Bein immer im Bereich des normal Möglichen befindet. Wird der Bereich verlassen, ist dies am auseinanderdriften der Knochen leicht zu erkennen. Aus diesen Gründen wird *IK* besonders häufig für Beinanimationen eingesetzt. Darüber hinaus ist es einfacher das Bein an einem Punkt in die richtige Position zu ziehen, als diesen Vorgang über einzelne Knochenrotationen durchzuführen. Dieser

Vorteil kann aber auch gleichzeitig ein Nachteil sein, da nicht jede Bewegung mit den Beschränkungen der *IK* durchführbar ist. Der gewichtigste Nachteil kann sich schlimmstenfalls erst beim Export in das Ausgabeformat bemerkbar machen, sollte der Exporter keine *IK* unterstützen. Dann kann es schwierig werden, einen fehlerfreien Export durchzuführen oder sogar dazu führen, dass ein Modell gar nicht erst exportiert werden kann. Da *Forward Kinematik* und *Inverse Kinematik* in den 3D Paketen üblicherweise unterschiedlich realisiert sind, ist es in der Regel nicht mehr möglich eine *Inverse Kinematik Animation* in eine *Forward Kinematik Animation* umzuwandeln. Das Testen des ausgewählten Exporters kann also viel zusätzliche Arbeit ersparen.

Bevor mit dem Gewichten der Bones begonnen wurde, wurden alle Knochen auf ihren „Roll-Wert“ überprüft, also ob sie um ihre eigene Achse rotiert sind. Diese Rotationen sollten grundsätzlich vermieden werden, da es sonst zu Fehldarstellungen kommen kann. Als eine Besonderheit in Blender müssen an dieser Stelle die Transformationsdaten des gesamten Knochengerüsts und auch des Meshs in die Objektdaten geschrieben werden. Danach gilt die aktuelle Position als Null-Position. Hier kann es ebenfalls zu schweren Fehlern kommen, sollte dieser Schritt ausgelassen werden. Da in unserer Projektapplikation der Charakter hauptsächlich von hinten betrachtet wird, wurde der Charakter in Blender um 180 Grad gedreht, bevor die Daten geschrieben wurden. Dies erleichtert es dem Programmierer bei der Einbettung in die OGRE-Engine, ist aber im Grunde eine Frage der Absprache zwischen Animator und Programmierer.

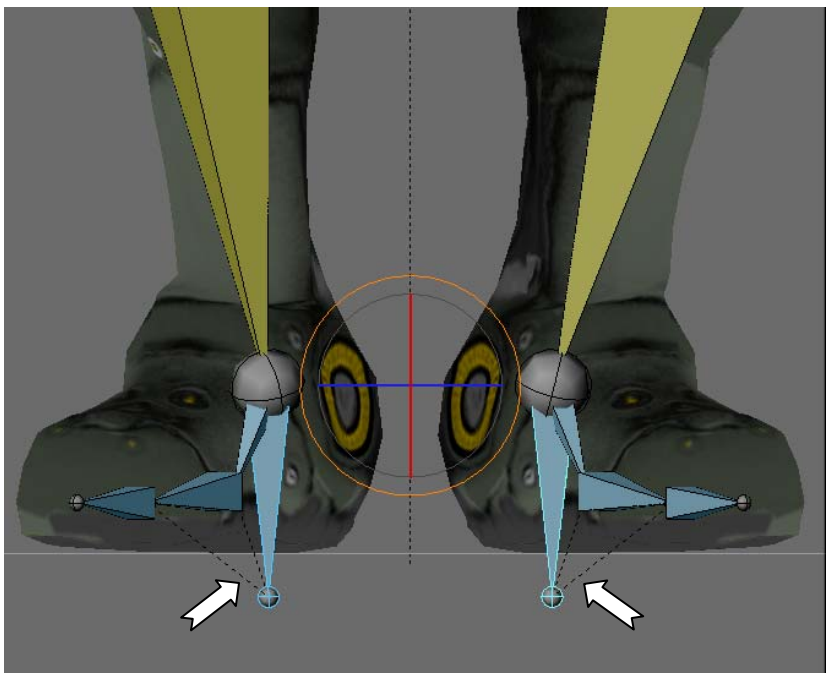

Abbildung 18: Als Anfasspunkt, von dem aus die Knochenstellung berechnet wird, dient ein zusätzlicher Knochen unter der Verse jedes Fußes

2.5.3 Gewichtung der Knochen

Nachdem das Skelett erstellt war, wurden die Knochen im Weight-Paint-Modus gewichtet. Als hilfreich erwies sich dabei, beim Aufmalen der Gewichtungen die Texturdarstellung zu deaktivieren und nur für das Testen auf Verformungen wieder hinzu zu schalten.

Eine gute Wichtung aller Knochen kann zeitintensiv sein, da möglichst viele Bewegungsfälle durchprobiert werden müssen. Umso mehr Bewegungen möglich sind, ohne dass unschöne Verzerrungen oder Überschneidungen auftreten, desto besser ist die Wichtung. Allerdings hängt die erreichbare Qualität in diesem Arbeitsgang auch maßgeblich von der Beschaffenheit des Modells ab, weswegen es wichtig ist, dass der Modellierer um die Vorhaben des Animators weiß. Um Verzerrungen zu reduzieren und ein organischeres Verhalten des Modells zu erzeugen, stehen fünf verschieden starke Gewichte zur Verfügung. Jedes Polygon des Charakters muss dabei ein Gewicht zugewiesen bekommen, damit das Modell korrekt dargestellt wird. Überprüft werden kann dies, wenn das ganze Modell nach dem Weight-Painting am Wurzelknochen bewegt wird. Bewegen sich einige Polygone nicht mit, dann muss noch nachgebessert werden.

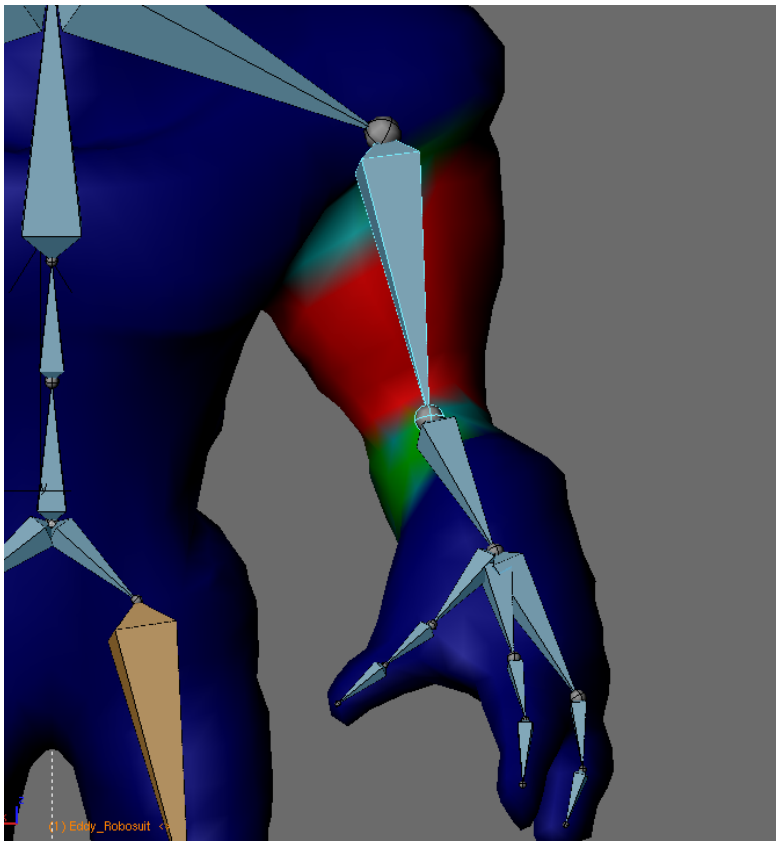

Abbildung 19: Unterschiedliche Weight-Painting Gewichte am Oberarm-Knochen.

2.5.4 Animation des Charakters

Für Eddy Robosuit waren folgende Animation eingeplant worden:

- Gehen
- Laufen
- Springen
- Idle
- Angreifen

Durch Schwierigkeiten, u.a. bei der Integration der Animationen und dem damit verbundenem Zeitverlust, wurden die Animationen für das Springen und das Laufen weggelassen. Aus der Situation heraus kamen dafür zwei neue Animationen hinzu, die das Alien beim Lenken und Lehnen in eine Rechts- und eine Linkskurve zeigen. Diese Animationen konnten leider ebenfalls aus Zeitgründen nicht mehr in die Anwendung integriert werden.

Animation des Walkcycles

Da der Schwierigkeitsgrad des Walkcycles von den erstellten Animationen am Höchsten ist und die meisten Besonderheiten berücksichtigt werden müssen, sei dieser Arbeitsschritt exemplarisch beschrieben.

Bei komplexen Animationen, wie z.B. einem Walkcycle bietet es sich an, entweder vorher einige Zeichnungen von den Bewegungszuständen anzufertigen oder bereits fertige Bewegungsstudien zu verwenden. Ein bekanntes Beispiel ist die Übersicht über die Bewegungen einer zweibeinigen Cartoon Figur von Preston Blair (Abbildung 20).

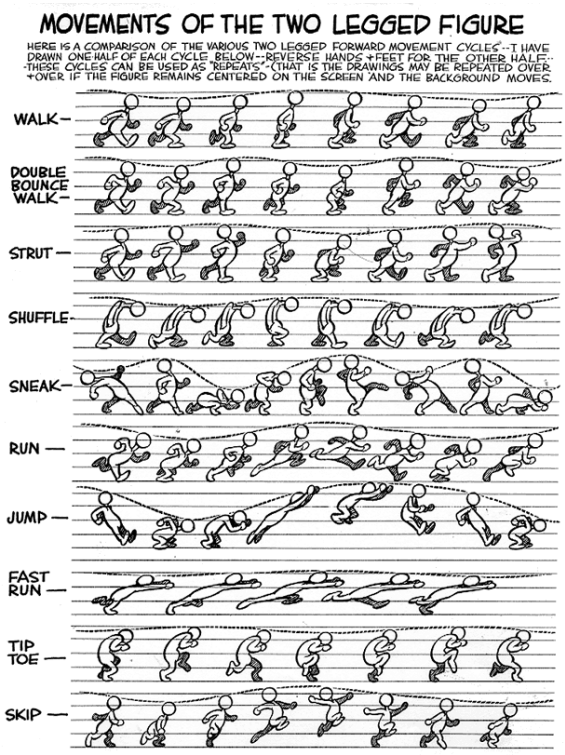

Abbildung 20: Bewegungsstudien eines Cartoon-Zweibeiners von Preston Blair.

Zunächst sollte die Ausgangsposition des Modells gespeichert werden, anschließend kann direkt mit der ersten Bewegungspose begonnen werden. In diesem Projekt wurde mit einer Pose, ähnlich des fünften Bildes der ersten Darstellung in Abbildung 20 begonnen. Der Abstand zwischen zwei Posen betrug dabei konstante drei Bilder, um anfangs eine gewisse Struktur und Übersichtlichkeit zu wahren. Unsymmetrische Verschiebungen sind nicht verboten, sollten aber erst zu einem sehr späten Zeitpunkt vorgenommen werden. Die Bilder zwischen zwei Posen wurden jeweils mit Bezier-Interpolation erzeugt.

Der Animationsprozess sollte in verschiedenen Durchläufen erfolgen, in denen jeder Durchlauf einer bestimmten Körperregion gewidmet wird - sofern dies möglich ist. Optimierungsdurchläufe werden dabei nicht mitgezählt. Der Walkcycle für den Charakter dieses Projekts wurde beispielsweise in fünf Hauptdurchgängen erstellt. Dabei stand die Beinanimation am Anfang. Während dieser Phase wurde eine kleine Ausnahme von der Durchlaufvorgehensweise gemacht: Der Körper verändert die Höhe in den Bewegungsphasen, so dass dieser während des Ausfallschritts etwas tiefer liegt, als in der Schwungphase. Daher wurde der Root-Knochen im gleichen Durchlauf animiert, um ein besseres Gefühl für die Bewegung zu bekommen. Im zweiten Durchlauf kam die Hüftdrehung hinzu, im dritten Durchlauf die Arme, im vierten Durchgang die Schulterdrehung und im fünften Durchlauf schließlich die Bewegungen für das Alien. Die Bewegungen wurden danach mit dem Curve-Editor, einem Funktionsgraph-Editor verfeinert.

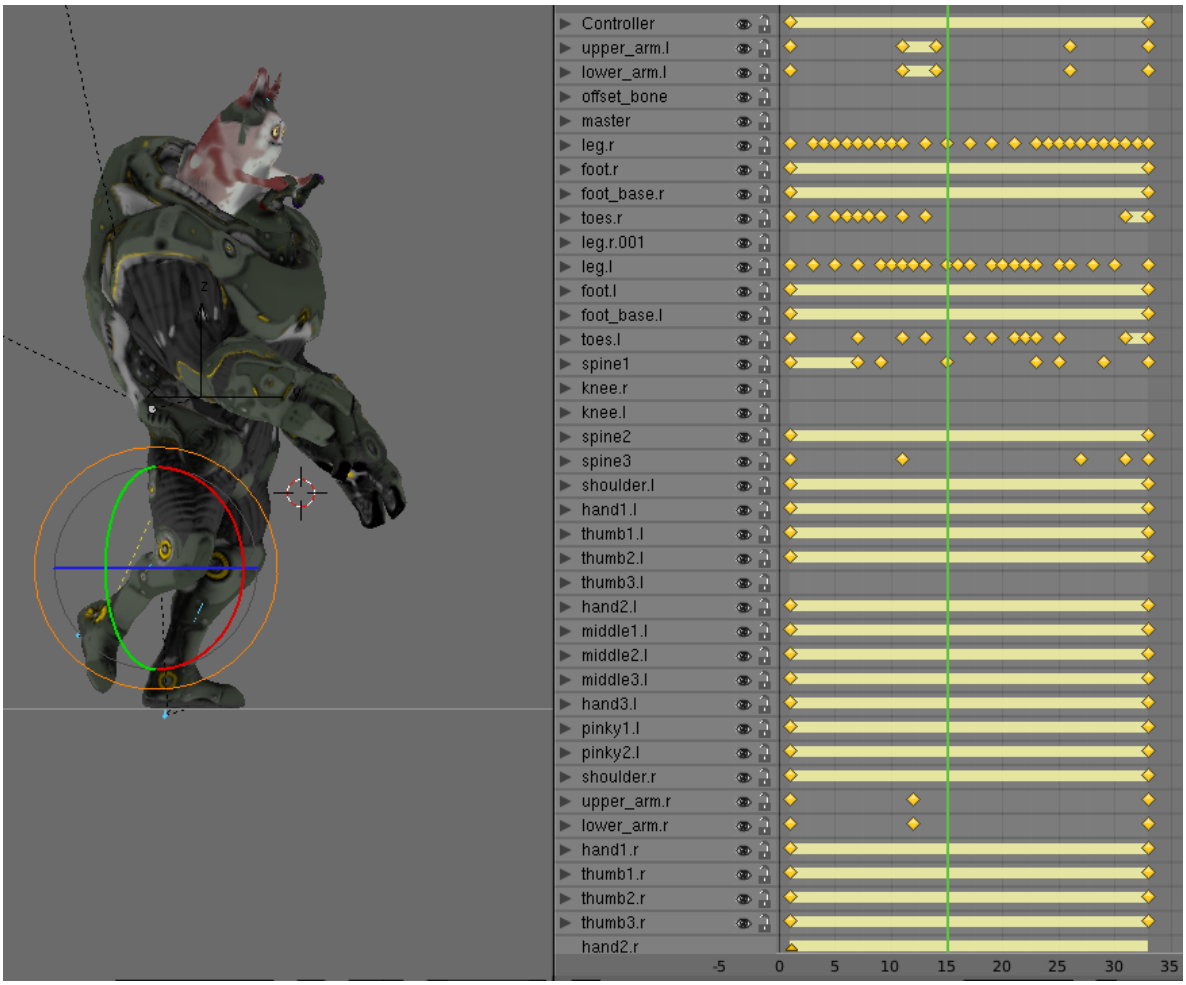

Abbildung 21: Ansicht auf den Action-Editor in dem auf Keyframe-Basis gearbeitet wird.

2.5.5 Export des Charakters in OGRE

Vorweg sei erwähnt, dass es unbedingt empfehlenswert ist, schon zu einem frühen Zeitpunkt des Animierens einige Testexporte durchzuführen. So können Probleme früh aufgedeckt werden, bevor viel Zeit in eine Animation investiert wird.

Um ein Objekt nach OGRE zu exportieren, muss, wie bereits erwähnt, das entsprechende Exporter-Plugin installiert sein. Danach ist es unter File > Export zu finden. Bevor ein Export erfolgen kann muss das Mesh des Charakters im Object-Mode ausgewählt sein. Danach sind die Einstellungen gemäß der Abbildung 21 Vorzunehmen. In der ersten Zeile wird der Name des ausgewählten Meshs angezeigt, direkt darunter können alle Animationen hinzugefügt werden, die exportiert werden sollen. Durch Angabe von Start- und End-Frame ist auch der Export von Teilanimationen möglich. In den Material-Settings ist der Name der .material-Datei anzugeben, in welcher das Shader-Script für das Objekt angelegt wird. Als einzige Option ist „Game Engine Materials“ auszuwählen. Im unteren Bereich muss der Exportpfad angegeben werden und darunter gibt es die Möglichkeit die nach oben zeigende Achse immer als Y-Achse zu bezeichnen. Eine .skeleton-Datei, welche alle Animationen

enthält, kann nach dem Meshnamen benannt und der Endkonverter direkt mit dem Konvertieren beauftragt werden. Um den Ablauf etwas zu verdeutlichen sei erwähnt, dass der Exporter zunächst nur einen Export in ein XML-Zwischenformat vornimmt. Dieses Zwischenformat muss danach mit den „Ogre Command Line Tools“ in das OGRE-Format konvertiert werden. Die eben beschriebene Option sagt aus, dass der Konvertiervorgang, ähnlich einer Stapelverarbeitung, direkt nach dem Export erfolgt. Im Preferences-Dialog des Exporters muss dazu die Position des Tools angegeben werden. Alle weiteren Einstellungen auf dieser Einstellungsseite sollten auf Default belassen werden, da Sie sehr speziell sind.

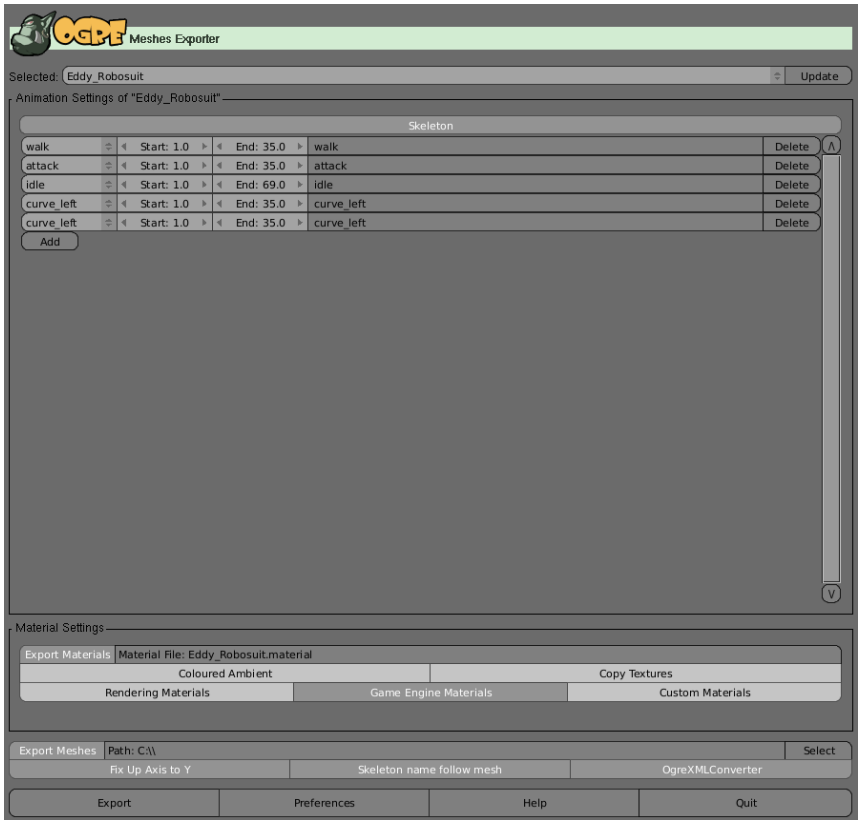

Abbildung 22: OGRE Exporter-Einstellungen

Um zu testen, ob der Export erfolgreich war, können die Dateien entweder in ein Programm eingebunden oder im „OGRE Mesh Viewer“ angezeigt werden. Letztere Variante empfiehlt sich besonders zum unkomplizierten Testen von exportierten Objekten. Der Viewer bietet das Anzeigen von Meshs, Texturen, Materialien und Animationen.

Abschließend sei erwähnt, dass der OGRE-Exporter automatisch Inverse Kinematik-Bewegungen in Bild-für-Bild-Animationen backt, so dass während des Animationsprozesses auf diese Animationstechnik zurückgegriffen werden kann.

2.6 Implementierung

In diesem Kapitel wird die Implementierung des Projektes beschrieben. Zunächst wird die Architektur des Projektes erläutert. Der zweite Abschnitt geht auf die Ansteuerung des Wii-Controllers ein. Abschnitt Charaktersteuerung beschreibt die Funktionsweise von Rotationsanimationen und der Charaktersteuerung mittels des Wii-Controllers. Im darauf folgenden Unterkapitel wird der Zielmodus mit Hilfe des Controllers erklärt und das letzte Unterkapitel konzentriert sich auf Beschreibung der Gameloop.

2.6.1 Systemarchitektur

Dieses Projekt ist die Weiterführung des Projektes „Spiele in einer 3D-Welt - von der Idee bis zur Realisierung“ [FLUB]. Im ersten Schritt erfolgt der Zuschnitt des alten Projektes auf das jetzige. Einige Klassen wurden übernommen, der Inhalt aber nach den neuen Bedürfnissen angepasst. Abbildung 23 gibt einen Überblick über die Klassenhierarchie.

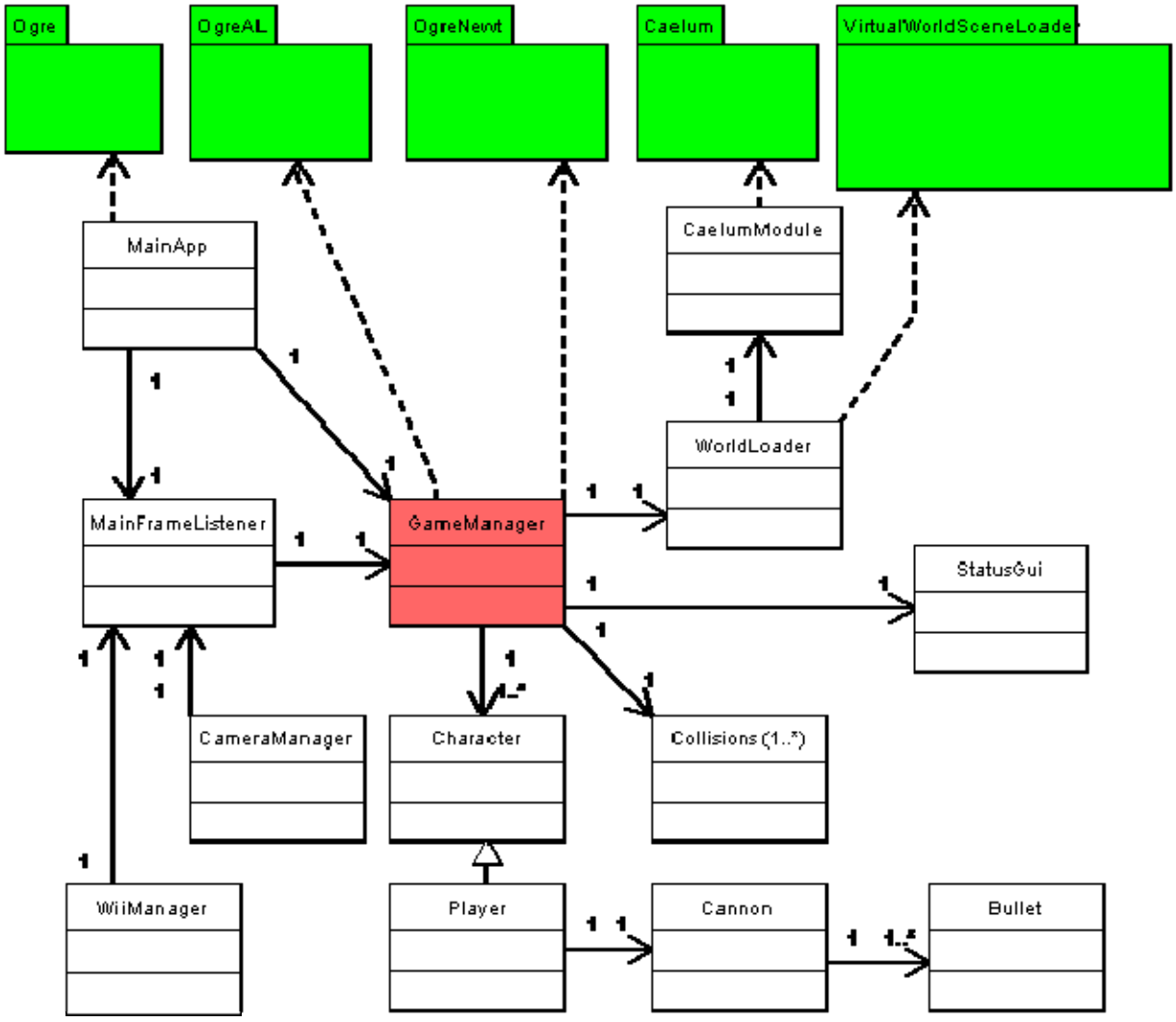

Abbildung 23: Klassenhierarchie

Bei den grün gefärbten Paketen handelt es sich um die Bibliotheken, welche in [FLUB] beschrieben wurden. Diese werden zur Laufzeit eingebunden.

MainApp

In dieser Klasse wird die Ogre-Engine initialisiert. Des Weiteren werden die Klassen GameManager und MainFrameListener mit der MainApp assoziiert.

CaelumModule

Die Bibliothek für Berechnung und Darstellung eines Himmels wird in die Klasse CaelumModule eingebunden. Anschließend werden die wesentlichen Komponenten, wie die zur Darstellung der Sonne, des Mondes, des Sternenhimmels und des Nebels eingestellt, sowie das heutige Datum eingegeben und eine Uhrzeit definiert. Der Tagesverlauf der Simulation orientiert sich an der Uhrzeit.

WorldLoader

Der WorldLoader ist verlinkt mit der Bibliothek VirtualWorldSceneLoader. Diese Bibliothek ist wiederum aus dem früheren Projekt [VWE] entstanden und ist für das Laden aller Objekte, sowie der Landschaft verantwortlich. Der WorldLoader verwaltet alle Objekte und die Landschaft und übermittelt diese dem GameManager zur weiteren Verarbeitung. Die Klasse CaelumModul wird im WorldLoader initialisiert.

StatusGui

StatusGui wird zur Statusanzeige des Spiels benötigt. In ihr wird eine einfache grafische Oberfläche bestehend aus Labels, welche in einem Rahmen angeordnet werden, entwickelt. In den Labels werden notwendige Informationen zum Debuggen, Bilder pro Sekunde, Uhrzeit etc. angezeigt.

Collisions

Diese Bezeichnung ist nur ein Oberbegriff für viele Klassen, welche von einer OgreNewt Basisklasse abgeleitet sind. Über diese Kollisionsklassen werden Kollisionsmomente zweier interagierender Körper abgefangen.

Character

Die Charakterklasse teilte sich im alten Projekt „Flubbers“ in zwei Unterklassen Flubber und Enemy auf. In diesem Projekt ist nur noch die Klasse Player von dieser Klasse abgeleitet, da alle Gegner entfernt wurden. Die Klasse Enemy, wird jedoch nicht gelöscht, da diese eventuell für spätere Projekte wieder zum Einsatz kommen könnte. Bei der Initialisierung des

Charakters wird das 3D-Modell erzeugt sowie der physikalische Körper. **Des Weiteren werden Geräuscheffekte erstellt.**

Player

Die Klasse Player ist von der Charakterklasse geerbt, somit übernimmt diese Klasse alle Standarteinstellungen der Oberklasse, zusätzlich werden aber noch viele weitere Routinen implementiert. Der Player wird über den Benutzer gesteuert. Beim Erzeugen dieser Klasse werden Spielerattribute wie Bewegungsgeschwindigkeit, Sprunghöhe, Animationsarten etc. definiert. Des Weiteren werden Animationsabläufe für die Bewegung und Aktionen des Spielers entwickelt.

Cannon

Die Klasse Player ist mit dieser Klasse assoziiert, der Spieler erhält eine Kanone, welche mehrere Patronen (Bullet Klasse) zum Schießen verwaltet.

Bullet

Eine Kanone besitzt eine Liste von Kugeln, da die Möglichkeit besteht, dass mehrere Kugeln in kurzen Zeitabständen vom Spieler geschossen werden und sich im Umlauf befinden. Eine Kugel wird durch die Klasse Bullet definiert. Sie erhält ein 3D-Kugel- Modell sowie einen geeigneten Kollisionskörper. Wenn der Spieler eine Kugel abfeuert und die Kugel mit der Welt kollidiert wird die Kugel aus der Welt entfernt und wieder in die Liste eingefügt.

GameManager

Die Klasse GameManager ist für die Verwaltung der Simulation verantwortlich. Sämtliche zuvor beschriebenen Klassen werden im GameManager definiert. Sie ist mit den Bibliotheken OgreAL und OgreNewt verlinkt. Im ersten Schritt wird die Physik-Engine initialisiert und **ein SoundManager Objekt von der 3D-Audio Bibliothek OgreAL** erstellt.
Als nächstes wird die Statusanzeige erzeugt, die 3D-Welt eingebunden und der Spieler angelegt. Anschließend werden Materialverknüpfungen für Kollisionen zwischen einigen physikalischen Körpern erstellt.

WiiManager

Der WiiManager dient als Schnittstelle zwischen der Wiiuse Bibliothek [WIIUSE] und dem Projekt. Im WiiManager wird ein Wii-Controller initialisiert. Zudem werden mehrere Funktionen des Joysticks und der Infrarotschnittstelle implementiert.

CameraManager

In der Klasse CameraManager wird die Kamera in die Third-Person-Perspective gesetzt, wenn der Charakter bewegt wird, d. h. die Kamera befindet sich etwas vom Charakter entfernt und folgt ihm. Wenn der Spieler sich im Zielmodus befindet, fährt die Kamera näher an den Spieler heran.

MainFrameListener

Der MainFrameListener ist von einer Klasse zur Verwaltung von Tastatur und Mausnachrichten abgeleitet. In dieser Klasse werden mittels einer eigenen Gameloop die Ogre- und die Physik-Engine aktualisiert. Des Weiteren erfolgt die Steuerung des Spielers über den WiiManager und alternativ mittels Tastatur und Maus.

2.6.2 Wii Ansteuerung

Die Ansteuerung der Wii erfolgt über die Wiiuse Bibliothek. Die Wiiuse Bibliothek ist in der Programmiersprache C geschrieben und Verwaltet die Verknüpfung mehrerer Wii-Controller mit dem PC.[1] Im Folgenden wird erläutert welche Schritte einen Wii-Controller benutzen zu können.

Im ersten Schritt muss dieser mit der Anzahl zu benutzender Controller initialisiert werden. In diesem Fall wurde MAX_WIIMOTES auf 1 gesetzt, da für dieses Projekt nur ein Controller Anwendung fand. Danach wird eine Funktion aufgerufen, welche eine gewisse Zeit nach Controllern sucht. Sobald einer gefunden wurde, erfolgt eine Verbindung mit der Bibliothek. Zuletzt wird noch eine Funktion aufgerufen, welche dem Benutzer durch Vibration des Controllers signalisiert, ob dieser funktionstüchtig ist.

```
wiimotes = wiiuse_init(MAX_WIIMOTES);                //MAX_WIIMOTES = 1
found = wiiuse_find(wiimotes, MAX_WIIMOTES, 5);      //5 Sek. lange suchen
connected = wiiuse_connect(wiimotes, MAX_WIIMOTES);
wiiuse_rumble(wiimotes[0], 1);
```

Damit ist die Initialisierung abgeschlossen. Im nächsten Schritt müssen die Signale des Gerätes abgefragt werden. Zunächst wird die poll() Funktion der Bibliothek aufgerufen, anschließend wird abgefragt ob und welcher Knopf gedrückt wurde.

```
if (IS_PRESSED(wm, WIIMOTE_BUTTON_B))
     buttonB = true;
else
    buttonB = false;
```

[1] Vgl. http://www.wiiuse.net, Internet: 30.01.2010

Wenn der Benutzer den A-Knopf gedrückt hält, wird der Infrarotmodus eingeschaltet und der poll() Funktion die Position der Infrarotkamera abgerufen. Das Zielen auf dem Bildschirm mit dem Wii-Controller ist ein exklusives Feature. Dies wird möglich, da der Controller eine Infrarotkamera besitzt. Zudem wird eine Sensorbar benötigt, welche aus zwei IR-Dioden besteht. Diese kann vom Benutzer auf oder unter dem Bildschirm platziert werden. Die Infrarotkamera nimmt die Lichtpunkte der IR-Dioden wahr und kann durch den Abstand der IR-Dioden zueinander die Position des Controllers in Abhängigkeit zum Bildschirm ermitteln. Die Daten werden daraufhin via Bluetooth an den Rechner gesendet.

```
wiiuse_set_ir(wm, 1);
```

Es stellte sich heraus, dass die Werte hohen Schwankungen unterliegen. D. h. auch wenn der Benutzer den Stick recht ruhig hält treten ständig Wertspitzen auf. Diese Ausreiser müssen herausgefiltert werden, dazu bietet sich nach [GJ09] ein Tiefpassfilter an. In der Tiefpassfilter Funktion werden der aktuelle Wert der Infrarotkamera und der des letzten Frames mit einer Gewichtung kombiniert und hohe Unterschiede geglättet.

Zuletzt werden der Winkel und die Stärke des bewegten Steuerknüppels abgefragt. Beim Testen des Spiels trat zunächst ein seltsames Phänomen auf, der Charakter rutschte durch den Boden, nachdem er einige Zeit bewegt wurde. Nach einigen Untersuchungen stellte sich heraus, das der Wii-Controller zeitweise Werte sendet, welche keiner Zahl entsprechen. Der Rechner fasst diese Werte auf und gibt NAN (Not A Number) an die Applikation weiter. Die Physik-Engine, welche die Bewegung des Charakters beeinflusst, wird dadurch instabil. Um diesem Problem entgegen zu wirken werden die Werte des Steuerknüppels nur an die Anwendung gesendet, wenn diese einer Zahl entsprechen und die Zahl zudem ungleich Null ist.

```
if (! std::isnan(nc->js.ang)  &&   (nc->js.ang   > 0.2f))
     angle = nc->js.ang;
```

2.6.3 Charaktersteuerung

Die Charaktersteuerung erfolgt über den Joystick des Wii-Controllers. Um glatte Rotationen zu gewährleisten wird die Rotationsmethode „Slerp“ verwendet, welche im Unterpunkt „Rotationsanimation“ beschrieben wird. Der Abschnitt „Bewegung des Spielers“ beschäftigt sich darüber hinaus mit der Fortbewegung. Der Abschnitt welche über Quaternions, welche im folgenden Abschnitt zunächst beschrieben wird.

Berechnung der Bewegung im Bezug zur Kamera

Die Metapher bei einer 3rd-Person-Steuerung ist folgende: Der Spieler sitzt in der Kamera und steuert den Charakter von diesem Blickwinkel aus. Drückt der Spieler den Joystick nach vorne, soll sich der Charakter unabhängig von seiner aktuellen Orientierung von der Kamera weg bewegen. Drückt er nach rechts, so soll sich der Charakter unabhängig von seiner aktuellen Orientierung nach rechts bewegen usw. Um dieses Verhalten zu Implementieren sind Koordinatentransformationen notwendig. Die Rohdaten die von dem Controller geliefert werden, sind die Magnitude und ein Winkel. Der Winkel gibt an in welche Richtung der Joystick gedrückt wird und die Magnitude, wie stark er in diese Richtung bewegt wird (Bereich von 0-1). Um die Transformation durchführen zu können muss zunächst über den Winkel ein Richtungsvektor berechnet werden, welcher „stickDirection“ genannt wird. Daraufhin wird ein „forwardVector“ und ein „rightVector“ der Kamera berechnet, welche Auskunft über die Orientierung der Kamera geben. Mit diesen Werten ist es nun möglich die lokale Stick-Orientierung des Joysticks in Abhängigkeit zur Kamera in die Weltkoordinaten umzurechnen. Dieser neue Vektor wird „worldStickDirection“ genannt und seine Orientierung in ein Quaternion gespeichert. Dies wird mittels folgendem Code realisiert [MD01].

```
Ogre::Vector3 worldStickDirection;
worldStickDirection.x = (stickDirection.x * rightVector.x) +
                        (stickDirection.z * forwardVector.x);

worldStickDirection.y = 0;
worldStickDirection.z = (stickDirection.x * rightVector.z) +
                        (stickDirection.z * forwardVector.z);


worldStickDirection.normalise();


Ogre::QuaternionnewOrientation =
       worldStickDirection.getRotationTo(Ogre::Vector3::NEGATIVE_UNIT_Z);
```

Somit wissen wir nun in welche Richtung der Charakter laufen soll, wenn der Joystick bewegt wird. Was noch nicht bekannt ist, ist wie schnell oder langsam er sich in diese Richtung bewegen soll. Dies wird mit Hilfe der Variablen „magnitude“ entschieden. Problem ist, dass kontinuierlich Winkel und Magnitude übermittelt werden, auch wenn der Stick nicht bewegt wird. Der Wert des Winkels ist dann willkürlich und der der Magnitude sehr gering. Das bedeutet selbst wenn der Joystick nicht bewegt wird würde sich der Charakter immer in Richtung des willkürlichen Winkels drehen. Um dies zu verhindern müssen Schwellwerte festgelegt werden. Ab einem Magnitude-Wert von 0.1 kann sicher davon ausgegangen werden das der gelieferte Winkel richtig ist und der Charakter erst ab dann orientiert wird. Überschreitet die Magnitude den Wert 0.5, wird der Charakter zusätzlich in die entsprechende Richtung bewegt. Die Geschwindigkeit der Bewegung kann dann von der Magnitude abhängig gemacht werden.

Rotationsanimation

Um eine interpolierte Rotation eines Körpers zu bewerkstelligen, bietet die Ogre-Engine die Funktion Quaternion::Slerp an. Slerp ist ein mathematisches Konstrukt um lineare interpolierte Rotationsanimationen, auf Basis von Quaternions durchzuführen.[2] Dabei wird die Zeit angegeben, sowie ein Startpunkt und ein Endpunkt. Daraufhin führt die Funktion die Drehung geglättet vom Anfangs- bis zum Endpunkt in der angegebenen Zeit aus.

Ausführung der Bewegung des Spielers

Um die Orientierung des Charakters zu realisieren wird das, nach obiger Beschreibung berechnete Quaternio, der Slerp Funktion übergeben. Somit wird der Charakter flüssig in die neue Richtung gedreht.

```
Ogre::Quaternion delta = Ogre::Quaternion::Slerp(5.0f, getOrientation(),
                                                 newOrientation.Inverse(), true);
pPhysicsBody->setPositionOrientation(getPosition(),  delta);
```

Der Parameter „true“ in der Funktion bedeutet, dass immer in Richtung des kürzesten Pfades gedreht wird. Die Geschwindigkeit der Bewegung wird mit der Magnitude gewichtet. Das bedeutet, bei geringer Bewegung des Sticks, bewegt sich der Charakter auch langsam. Die Ausführung der Bewegung wird nun mit der Physik-Engine gehandhabt. Im ersten Schritt wird die Orientierung (Quaternion) in einen Vektor umgewandelt.

```
Ogre::Vector3 direction = getOrientation() * Ogre::Vector3::NEGATIVE_UNIT_Z;
```

Anschließend wird ein Geschwindigkeitsvektor (velocity) in diese Richtung berechnet. Zusätzlich wird die Y-Komponente des Geschwindigkeitsvektors festgehalten, damit die Schwerkraft darauf einwirken kann. Dies ist notwendig, damit der Spieler immer auf dem Boden bleibt, wenn er beispielsweise einen Berg herunterläuft. Zuletzt wird die Geschwindigkeit auf die berechnete Richtung addiert.

```
pPhysicsBody->setVelocity(pPhysicsBody->getVelocity() * Ogre::Vector3(0, 1, 0) +
direction * magnitude * (speed);
```

Basierend auf der Animationslogik von Flubbers [FLUB] wird eine Geh-Animation ausgeführt.

[2] Vgl. http://en.wikipedia.org/wiki/Slerp Internet: 02.02.2010

2.6.4 Zielmodus

Im Kapitel „Wii Ansteuerung“ wurde erläutert, wie der Benutzer mit dem Wii-Controller zielen kann. Ziel ist es über diese Schnittstelle Geschosse in der 3D-Welt abzufeuern. Nachfolgend wird erläutert, wie dies zu bewerkstelligen ist.

Wenn der Benutzer diesen Zielmodus betreten möchte, wird die A-Taste gedrückt. Daraufhin fährt die Kamera näher zum Charakter, um einen besseren Blickwinkel zur Verfügung zu stellen. Zusätzlich wird die entsprechende Angriffsanimation ausgeführt und im Overlaykoordinatensystem ein Fadenkreuz eingeblendet. Abbildung 24 zeigt einen Screenshot der Welt im Zielmodus.

Abbildung 24: Screenshot vom Zielmodus

Die gefilterten 2D Koordinaten der Infrarotkamera werden ausgelesen. Beim Zielen auf die untere Ecke des Bildschirmrandes waren bei einer Auflösung von 1920:1200 und einem Bildverhältnis von 16:10 die Koordinaten identisch mit der Auflösung. Da Ogre ein eigenes Koordinatensystem besitzt um 2D-Objekte zu verwalten, müssen die Koordinaten des Sensors erst kalibriert werden (Siehe Abbildung 25).

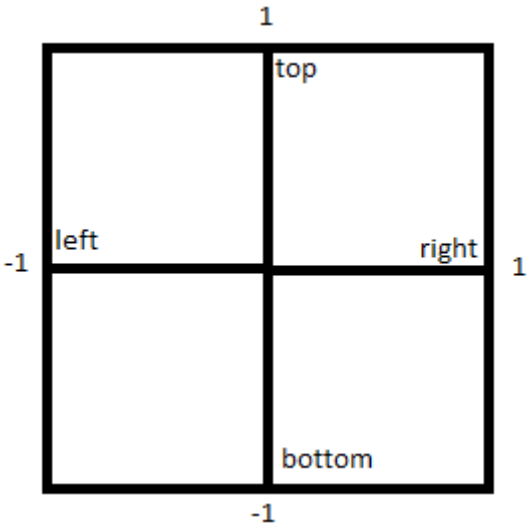

Abbildung 25: Veranschaulichung des 2D-Koordinatensystem in Ogre

Das heißt sie müssen in das Overlaykoordinatensystem von Ogre transformiert werden. Im ersten Schritt wird die Position der Infrarotkamera relativ zur Fenstergröße (wWidth, wHeight) umgerechnet.

```
Ogre::Real left = wiiX / (Ogre::Real) wWidth;
Ogre::Real top = wiiY / (Ogre::Real) wHeight;
Ogre::Real right = (wiiX / (Ogre::Real) wWidth)+(Ogre::Real)(200.0f / wWidth);
Ogre::Real bottom = (wiiY /(Ogre::Real) wHeight)+(Ogre::Real)(200.0f / wHeight);
```

Die Variablen right und bottom sind 200 Pixel groß, da das Bild mit dem Fadenkreuz ebenfalls diese Größe besitzt. Zusätzlich wird spezifiziert, dass wenn der Benutzer an den linken oder den rechten Rand zielt, der Charakter sich beginnt in die jeweilige Richtung zu drehen. Um das Fadenkreuz an der korrekten Position im Bildschirm anzuzeigen, werden im nächsten Schritt die Sensorkoordinaten in Bildkoordinaten umgerechnet.

```
Ogre::Real    transformLeft  =  left * 2.0f - 1.0f;
Ogre::Real    transformRight  =  right * 2.0f - 1.0f;
Ogre::Real    transformTop = 1.0f - top * 2.0f;
Ogre::Real    transformBottom = 1.0f - bottom * 2.0f;
pCrosshairRect->setCorners(transformLeft,  transformTop,  transformRight,
                    transformBottom);
```

Im Folgenden wird erläutert, wie mittels dieser Koordinaten eine Kugel vom Spieler aus in die 3D-Welt abgeschossen werden kann.
Als erstes wird die Startposition der Kugel benötigt. Diese wird dorthin gesetzt, wo sich der rechte ausgestreckte Arm des Charakters befindet. Die Zielposition wird über einen Ray berechnet. Ogre bietet die Funktion „getCameraToViewportRay“ an, um 2D Bildkoordinaten abhängig von der Kamera in 3D Koordinaten umzurechnen.

```
Ogre::Ray aimRay = pCamera->getCameraToViewportRay(
                           left + (Ogre::Real)((200.0f / wWidth) / 2.0f),
                           top + (Ogre::Real)((200.0f / wHeight) / 2.0f));
```

Um die Endposition zu erhalten, wird die Anfangsposition genommen und die Richtung des Rays mit 10000 Metern verlängert. In diese Richtung wird die Kugel mit Hilfe von OgreNewt mit einer Geschwindigkeit von ca. 50 Meter pro Sekunde abgefeuert. Wie das Abfeuern des Geschosses realisiert wurde, kann in Quelle [FLUB] entnommen werden. Es wurde bemerkt, dass es beim Zielen in den Randbereichen zu Abweichungen kommt. Der Grund für diese Abweichungen konnte aus zeitlichen Gründen nicht ermittelt werden.

2.6.5 Das Kamera-System

Dieses Kapitel erläutert die Entwicklung und Implementierung einer Third-Person-Kamera. Schaut man sich moderne Third-Person-Spiele an, so besitzen sie eine sehr intuitive und angenehme Steuerung. Selbst Neueinsteiger beherrschen schnell die Interaktion mit der virtuellen Welt über die Eingabegeräte. Eines der Spiele, welche dieses Kamerasystem populär gemacht haben und als Vorreiter gilt, ist Super Mario 64 [MD01]. Ein Teilziel dieses Fachprojektes war es, eine solches Kamera-System zu entwickeln.

2.6.5.1 Der Aufbau des Systems

Die wichtigste Komponente bei einem Third-Person Kamera-System ist eine frei fliegende Kamera, die dem Charakter ständig folgt. Dabei sollte die Distanz zwischen dem Charakter und der Kamera nicht zu starr wirken, sondern je nach Aktion des Charakters variieren. Bildlich gesprochen kann man sich ein Bungee-Seil zwischen Charakter und Kamera vorstellen. Rennt der Charakter aus dem Stand los, so benötigt die Kamera ein wenig Zeit um nach zu kommen, bleibt er abrupt sehen, federt sie leicht nach. Ein weiteres Element des Systems ist das Bedürfnis der Kamera sich immer hinter den Charakter zu drehen, damit der Spieler die Welt vor dem Charakter sieht. Sowohl die Federkraft als auch das Drehen hinter den Charakter sollte sehr subtil geschehen. Die Algorithmen und die Realisierungen dieser Effekte werden im folgenden Abschnitt „Die Floating Kamera“ näher erläutert.

Die Floating Kamera

In diesem Abschnitt wird die prinzipielle Funktionsweise der Floating-Kamera erläutert. Sie basiert auf Federkräften die mittels „Hooke’s Law“ berechnet werden [KP05]. Die Kamera wird, wie schon erwähnt, in 2 Schritten realisiert. Unterstützend illustriert die Abbildung 26 das Verfahren.

Um die Federfunktionalität zu erreichen wird zunächst ein Vektor zwischen Kamera und Charakter berechnet (lookAtVector). Die Kamera sollte immer einen gewissen Abstand zum Charakter halten. Geht man die Länge dieses Abstandes auf dem lookatVector ab, so findet man einen Punkt und somit auch einen Vektor, welcher targetVector genannt wird. Die Kamera versucht nun diesen Vektor zu erreichen. Der Bewegungs-Vektor der Kamera trägt die Bezeichnung transVelocityVector.
Damit sich die Kamera hinter den Charakter dreht wird ein Vektor entgegen der Charakterorientierung berechnet (backVector). Nun wird zwischen der Kameraposition und diesem „backvector“ der Vektor „rotVelocityVector“ berechnet, der angibt in welche Richtung sich die Kamera zusätzlich drehen muss.

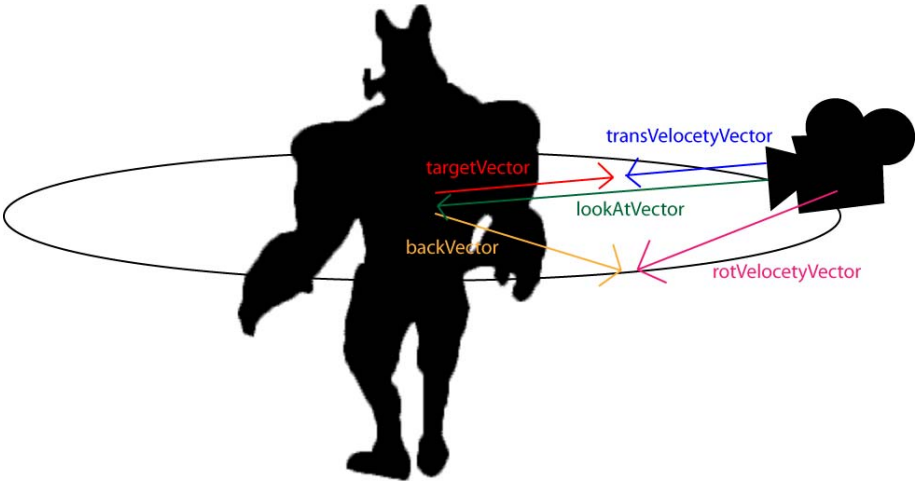

Abbildung 26: Das Kamerasystem

Diese beiden Schritte werden in jedem Frame berechnet und ein gewisser Bruchteil des „transVelocityVector“ und des „rotVelocityVector“ auf die Kameraposition addiert. Somit folgt die Kamera dem Charakter.

2.6.6 Einführung in eine eigene Gameloop

Zunächst wurde im Projekt die Verwaltung von Frames der Ogre-Engine überlassen. Als jedoch der Wii-Controller zum Einsatz kam, von welchem die Signale ebenfalls periodisch abgefragt werden müssen, verschlechterte sich das Laufzeitverhalten. Hinzu kam, dass wenn eine Benutzerinteraktion mit dem Controller statt fand, Verzögerungen auftraten. Um ein konkretes Beispiel zu nennen: Hat der Spieler den Steuerknüppel bewegt, erfolgte die Bewegung des Charakters erst mit einer deutlich sichtbaren Verzögerung. Nach intensiver Recherche in Ogre Foren stellte sich heraus, dass der FrameListener der Ogre-Engine nur entwickelt wurde, um die Aktualisierung der Grafik abzuhandeln. Einige Community Mitglieder gaben den Rat, eine eigne Gameloop zu entwickeln, wenn noch zusätzliche

Komponenten aktualisiert werden müssen. Daher wurde außerplanmäßig eine eigene Gameloop eingeführt. Im Folgenden wird der Ablauf der Gameloop erläutert.

Eine Gameloop ist programmiertechnisch nichts anderes als eine Schleife, in welcher Operationen in einem definierten Intervall periodisch ausgeführt werden, solange die Applikation aktiv ist. In Quelle [DW01] werden einige Varianten einer Gameloop vorgestellt. Die nachfolgende Beschreibung orientiert sich dabei an diesem Artikel.
Die Ogre-Engine besitzt eine Präzisionsuhr, mit dessen Hilfe sich die abgelaufenen Mikrosekunden, seitdem die CPU gestartet oder der Ogre-Timer zurückgesetzt wurde, abfragen lassen. Alle Komponenten werden über diesen Timer aktualisiert. Die Physik-Engine OgreNewt wird mit 60 Fps aktualisiert. Beim Updaten des Spiels würden 25 Fps genügen, jedoch fängt der Charakter bei Bewegungen an zu zucken, weswegen die Echtzeitanwendung vorerst ebenfalls mit 60 Fps aktualisiert wird. Zuletzt müssen die Benutzereingaben über die Maus, Tastatur und den Wii-Controller zyklisch abgefragt werden. Diese wurden zunächst auch mit 60 Fps abgefragt, wodurch wieder Verzögerungen auftraten. Über das Ogre-Forum wurde herausgefunden, dass der Wii-Controller mit 100 Fps aktualisiert werden muss, da ansonsten die Benutzereingaben in eine Queue abgelegt werden. Um diesem Problem entgegenzuwirken, wurden die Benutzereingaben in einen eignen Thread ausgelagert, in welchem diese mit 100 Fps, also alle 10 Millisekunden (statt alle 16,667 Millisekunden bei 60 Fps) abgefragt werden.
Die Quelle [DW01] beschreibt, dass das Rendern der Grafik keiner Begrenzung ausgesetzt sein soll. Es ist jedoch sofort aufgefallen, dass der Bildschirm flimmerte, wenn die Bildrate über 60 Fps stieg. Ogre bietet die Einstellung „VSync“ (Vertical Synchronisation) an, wodurch sich die Spielschleife orientiert an der Hertz-Rate des Bildschirms orientiert. Daher wurde VSync über den Konfigurationsdialog von Ogre eingestellt.
Um Teilkomponenten der Applikation nur zu bestimmten Zeitpunkten zu aktualisieren, wird zuerst die Bildrate berechnet. Bei der Aktualisierungsrate der Physik-Engine entspräche dies in etwa:

```
1000000 / 60 = 16666,667 Mikrosekunden (~0.0167 Sekunden)
```

Im Folgenden wird anhand der Aktualisierung von OgreNewt veranschaulicht, wie die Gameloop funktioniert.

```
ogreTimer.reset();
nextPhysicsUpdate = ogreTimer.getMicrosecondsCPU();
while (gameRunning)
{
     if (ogreTimer.getMicrosecondsCPU() >= nextPhysicsUpdate)
     {
            updateOgreNewt(fps);
            nextPhysicsUpdate += SKIP_TICKS_PHYSICS;
     }
     …
}
```

Nach dem Zurücksetzen des Timers werden die abgelaufenen Mikrosekunden abgefragt. Anschließend wird, solange das Spiel läuft, in der Spielschleife abgefragt, ob genug Zeit verstrichen ist, um die Physik zu aktualisieren. Falls dies der Fall ist, werden in etwa 60 Fps an OgreNewt weitergegeben und der nächste Aktualisierungszeitpunkt um 16666,667 Mikrosekunden erhöht. Bis zum nächsten Update muss diese Zeit wieder verstreichen.

Leider kommt es je nach Rechnerarchitektur ab und an vor, dass die Kamera manchmal zuckt, der Grund für diese Unreinheit konnte bisher nicht herausgefunden werden. Des Weiteren liegt die Fps Zahl beim Starten der Anwendung recht niedrig und steigt erst mit der Zeit. Dabei ist zu beobachten, dass die Bewegung, die Animationen etc. etwas zu schnell ablaufen. Trotz vielen Maßnahmen, zur Anpassung der Geschwindigkeit konnte dieses Problem nicht gelöst werden. In Quelle [DW01] wurde erwähnt, das man einen Interpolationsfaktor benutzen soll, welcher die Zeit zwischen zwei Frames anpasst. Jedoch führte die Einführung dieses Faktors nicht zum Erfolg. Zudem wurde die Zeit zwischen zwei Bildern gemessen und alle zeitlichen Abläufe mit diesem dem ermittelten Wert kombiniert, aber auch diese Maßnahme führte nicht zum Erfolg. Trotz allem ist die Gameloop für dieses Projekt nach eigener Einschätzung angemessen.

3 Fazit

Im Allgemeinen konnten die Zielsetzungen, welche durch den Projektplan vorgegeben waren, bis auf wenige Ausnahmen, eingehalten werden. So konnte das Überblenden von Animation aus zeitlichen Gründen nicht umgesetzt und zwei Animationen nicht mehr erstellt werden.
Blender stellte sich als geeignetes Werkzeug zur Modellierung, Texturierung und Animation eines Charakters für eine Echtzeit-Anwendung heraus. Allerdings gibt es beim Umgang mit dem Programm einige Besonderheiten zu beachten, ohne die es zu Problemen im Arbeitsablauf kommen kann. Beispielsweise muss beim Erstellen eines Skeletts darauf geachtet werden, dass die Transformationsdaten vor dem Animieren bestätigt werden. Darüber hinaus ist das Programm zwar gut dokumentiert, leider aber nicht im Zusammenhang mit dem Exportwerkzeug, so dass sich bei exportspezifischen Problemen, eine Fehlersuche schwierig gestalten kann. Für das Skulpting ist ein externes Programm, wie z.B. ZBrush empfehlenswerter.
Die Entscheidung eine eigene Gameloop zu entwickeln wird Rückblickend als richtig angesehen. Mit dieser wurde die Fps-Rate etwas verbessert und nicht zuletzt die Einbindung des Wii-Controllers ermöglicht. Den Autoren wurde klar, dass es ein komplexes Unterfangen ist, alle Echtzeit-Komponenten miteinander in Einklang zu bringen, was die Voraussetzung für ein stabiles Echtzeitverhalten ist. Die Steuerung über den Wii-Controller und dem dazugehörigen Kamerasystem funktioniert auf dieser Basis zufriedenstellend. Beide Systeme entsprechen unserer Vorstellung von einer intuitiven und angenehmen Steuerung.

Es war sehr interessant die zum Ziel gesetzten Entwicklungsprozesse Schritt für Schritt in der Praxis umzusetzen und auf diese Weise die recht unterschiedliche Handhabung der Tools und Technologien kennen zu lernen. Gerade das Zusammenspiel von gestalterischen und technologischen-Elementen war eine sehr interessante Erfahrung. Es sei angemerkt, dass komplexe Thematiken wie „Shader“ genügend Materie für zukünftige Vertiefungen bieten würden.

4 Quellen

[GL09] Game Loop Article: Internet 17.02.2010
http://dewitters.koonsolo.com/gameloop.html

[GJ09] Gregory, Jason: Pro Game Engine Architecture, 1. Auflage, 888 Worcester Street, Suite 230, Wellesley, MA 02482, A K Peters, Ltd.

[KL09] Kalinowski, Lukas: Spiele in einer 3D-Welt – Von der Idee bis zur Realisierung, Bachelorarbeit 2009, Postfach 1380, 55761 Birkenfeld

[JW08] John Wiley & Sons, Introducing ZBrush, 1. Auflage, 2008, SYBEX, ISBN-10: 0470262796

[KP05] Keith Peters, Foundation ActionScript Animation: Making Things Move! 1.Auflage, 2005, Friends of ed, ISBN-10: 1590595181

[AW08] Antony Ward, Game Character Development, 1. Auflage, 2008, Cengage Learning Services, ISBN-10: 1598634658

[CW08] Carsten Wartmann, Das Blender-Buch, 3.Auflage, 2008, dpunkt.verlag, ISBN-10: 978-3_89864-466-2

[MD01] Mark DeLoura, Game Programming Gems 2, 1.Auflage, 2001, Charles River Media, ISBN-10: 1584500549

[WIIUSE] Wiiuse - The Wiimote C Library: Internet 17.02.2010
http://www.wiiuse.net/

[VWE] Bies Daniel, Kalinowski Lukas: Virtual World Editor, IP-Projektbericht WS 2008/09, Postfach 1380, 55761 Birkenfeld

[FLUB] Kalinowski Lukas: Spiele in einer 3D-Welt – Von der Idee bis zur Realisierung, Bachelorarbeit 2009, Postfach 1380, 55761 Birkenfeld

[BLEND] Blender 2.49b: Internet 17.02.2010
http://www.blender.org/development/release-logs/blender-249/

[BLEND2] Leg Rigs Tutorial: Internet 19.02.2010
http://wiki.blender.org/index.php/Doc:Tutorials/Animation/Armatures/BSoD/Leg Rigs

[ZBRUSH] ZBrush Trial: Internet 17.02.2010
http://www.pixologic.com/zbrush/trial/

[FXM] FX-Map: Internet 17.02.2010
http://www.ogre3d.org/forums/viewtopic.php?t=8914&highlight=shader

5 Anhang

5.1 Verbinden der Wii-Remote mit dem PC

Zum Verbinden des Wii-Controllers mit dem PC wurde die Software
IVT BlueSoleil 2.7.0.13 und ein USB-Bluetoothadapter[3] verwendet.

Verbindungs-Schritte:

1. USB-Bluetoothadapter einstecken.
2. Software installieren
3. Software starten
4. Auf dem Wii-Controller die Buttons mit der Bezeichnung 1 und 2 gemeinsam halten und gehalten lassen (LEDs beginnen zu blinken)
5. Nun in der Software Doppelklick auf den Ball
6. Es erscheint ein Gerät mit der Bezeichnung: Nintendo RVL-CNT-01
7. Rechtsklick auf Gerät und „Dienste aktualisieren“ klicken
8. Jetzt ist wird das Gerät als Human Interface Device erkannt und die Maus-Symbol wird gelb.
9. Erneuter Rechtsklick auf das Gerät und Verbinden→Bluetooth Human Interface Device Service klicken
10. Das Gerät wird grün und eine Verbindung erscheint zum Ball.
11. Gerät ist mit PC gekoppelt und Einsatzbereit ! :-)
12. Um zielen zu können muss noch eine Senserbar auf den Bildschierm plaziert werden[4]

5.2 Installation der Entwicklungsumgebung

Installation der Software:

1. TortoiseSVN-1.6.5.16974-win32-svn-1.6.5.msi installieren
2. OgreSDKSetup1.6.1_CBMingW.exe installieren

2.1 Arbeitplatz →Eigenschaften →Erweitert →Umgebungsvariablen →OGRE_HOME löschen, da es in Eclipse angelegt wird

---

[3] Internet: 19.02.2010 http://www.amazon.de/Trendline24-USB-Bluetoothadapter-2-0-Bluetooth-Stick/dp/B000LLRLGE/ref=sr_1_7?ie=UTF8&s=ce-de&qid=1266589876&sr=8-7

[4] Wireless Remote Sensor-Bar, Internet: http://www.amazon.de/Wireless-Remote-Sensor-Bar-Leiste-f%C3%BCr/dp/B000W4HS3G/ref=sr_1_7?ie=UTF8&s=videogames&qid=1266590914&sr=1-7

3. Newton installieren
4. blender-2.49b-windows.exe installieren
5. python-2.6.3.msi installieren
6. OpenAL11CoreSDK.exe installieren
7. MinGW-5.1.6.exe installieren --> custom auswählen: g++, g77, MinGW Make
8. Eclipse c++ in Programme kopieren

Einrichten von Eclipse:

1. Beim Starten workspace im fachprojekt ordner erstellen
2. File →import →(General aufklappen →Existing Project into Workspace) →auf Next im Wizard drücken →Browse → (Ordner Flubbers auswählen) →Finish drücken
3. In C:\fachprojekt MMI\Flubbers\Debug gehen → copy.bat öffnen → Pfade an System anpassen: Bsp. copy C:\Fachprojekt MMI\Flubbers\Debug\Flubbers.exe C:\Fachprojekt MMI\Flubbers\app\Debug\Flubbers.exe
4. Projekt → Properties öffnen → Eintrag Builders → New_Builder wählen → Edit drücken → Pfad auf lokales copy.bat angeben.
   (Location Bsp. C:\Fachprojekt MMI\Flubbers\Debug\copy.bat Working Directory Bsp. C:\Fachprojekt MMI\Flubbers\Debug)
5. Pfadvariablen in Eclipse anlegen:

5.1 Window --> Preferencies (Nach "String" suchen) --> String Substitution auswählen --> new

5.2 Alle aufgelisteten Variablen anlegen:

Name	**Value**	
OGRE_HOME	C:\Programme\OgreSDK1.6.1	
OGREAL	C:\Fachprojekt_MMI\OgreA	
OPENAL	C:\Programme\OpenAL 1.1 SDK	
OGRE_NEWT	C:\Fachprojekt_MMI\OgreNewt\OgreNewt_Main	NEWTON
	C:\Programme\NewtonSDK\sdk	ETM
	C:\Fachprojekt_MMI\etm-2.3.1	CAELUM
	C:\Fachprojekt_MMI\Caelum-0.4.0\main	VWSL
	C:\Fachprojekt_MMI\VirtutalWorldSceneLoader	TINYXML
	C:\Fachprojekt_MMI\tinyxml_2_5_3\tinyxml	
FLUBBERS_RELOADED	C:\Fachprojekt_MMI\Flubbers Reloaded	
WIIUSE	C:\Fachprojekt_MMI\wiiuse_v0.12_win	

6. H Dateien der Externen Dll's angeben:
   Project --> Properties --> "GCC C++ Compiler" aufklappen --> "Directories"

"${TINYXML}"
"${ETM}/include"
"${VWSL}/include"
"${OGREAL}/include"
"${CAELUM}/include"
"${OPENAL}/include"
"${OGRE_NEWT}/inc"
"${OGRE_HOME}/include"
"${NEWTON}"
"${workspace_loc:/Flubbers Reloaded/include}"

7. Auf Dll's verweisen

   Libraries:

   OIS_d
   libcaelum0.4
   libEtm
   libVirtualWorldSceneLoader
   libOgreAL
   OgreMain_d
   Newton
   OpenAL32
   libOgreNewt
   wiiuse_debug

   Library search path (-L)
   "${FLUBBERS_RELOADED}/app/Debug"

Achtung: Wenn sich im Projektexplorer auch der Ordner app befindet, dieser darf nicht mit kompiliert werden.
→ Rechtsklick auf diesen Ordner → Exclude from Build → Release/Debug anwählen

5.3 Statusberichte

Zeitraum 19.10.09 - 23.10.09

Anlegen der Entwicklungsumgebung:

- Alle benötigten Komponenten installiert
- Ogre mit den nötigen Erweiterungen installiert und angepasst
- Eclipse mit Workspace eingerichtet (Probleme traten auf, hat mehr zeit gekostet als angenommen)
- SVN Repository eingerichtet
- Lowpoly Modell erstellt

Zeitraum 26.10.09 - 30.10.09

Verknüpfung und Kommunikation der WII mit dem PC

- Beschaffung der Hardware
- Installieren der Komponenten (Bluesoleil)
- Test der Verbindung mit wiiusej gui

Zeitraum 02.11.09 - 13.11.09

- Texturieren des Charakters
- Prototypentwicklung der Charaktersteuerung und einer dynamischen Kamera
- Entwicklung einer eigenen Gameloop
- Integration der Wii-Steuerung in die Gameengine
- Rigging des Charakters
- Weight-Painting
- Begin der Charakteranimation

Arbeitspakete wurden keine endgültig abgeschlossen. Bis auf die Charakteranimationen stehen bei den oben genannten
Punkten nur noch Feinheiten aus, so dass wir mit den Abschlüssen in den nächsten Tagen rechnen.

Das Ersetzen der Default-Gameloop durch eine Eigenentwicklung war ursprünglich nicht geplant. Es erwies sich jedoch als notwendig,

da die ursprüngliche Struktur der Gameloop in Verbindung mit der Wii-Steuerung zu erheblichen Synchronitäts-Problemen
zwischen den einzelnen Komponenten geführt hat. Dadurch verringerte sich die Ingame-Framerate auf ein nicht praktikables Niveau.
Sowohl das Ausbalancieren der Gameloop, als auch das Integrieren der Wii-Steuerung stellten sich als schwierige und langwierige Vorgänge heraus.

In den kommenden zwei Wochen werden wir uns um das Verfeinern des Charakters, die Charakter-Animationen und das Integrieren des Meshs und der Animationen in die Testumgebung kümmern.

Zeitraum 16.11.09 - 27.11.09

- Exportieren des Characters als Mesh für die weitere Benutzung für Ogre
- Erstellung von Shaderprogrammen und Materialdateien für:
- DiffuseMap
- NormalMap
- SpecularMap
- GlowMap
- finalles Einbauen der Kamerasteuerung
- Steuernung des Charakteren über die Wii
- Anpassung des Lichts in der Welt (wurde benötigt um die Lichtberechnung des Shaders zu ermöglichen)
- Erstellen einer Geh-Animation

Zeitraum 30.11. - 11.12.09

in den letzten zwei Wochen wurden folgende Arbeiten innerhalb unseres
Fachprojekts vorgenommen:

- Erstellen und Ansteuerung des Zielmodus mit dem Infrarot-Sensor der Wii-Remote
- Einbauen eines Fadenkreuzes
- schießen voll Bällen in Richtung des Fadenkreuzes (nicht trivial!!)
- Physikalische Auswirkung der Bälle in der Umgebung
- Erstellen von Animationen in Blender
- Große Problematiken beim Export Animationen (IK wird in Ogre3D nicht unterstützt)
- Recherche nach Alternativen

In diesen zwei Wochen wurde endgültig das Texturieren-Arbeitspaket beendet.
Leider konnte das Animations-Paket noch nicht fertig gestellt werden, weil es wie oben schon erwähnt massive Problem beim Exportieren der bereizt vorhanden Animationen gab. Somit wird es hier eine Verzögerung geben.
In der kommenden Woche werden wir uns verstärkt um dieses Paket, als auch um die Arbeitsschritte "Animationsblending" und "Steuerung der Animation" kümmern.

Zeitraum 11.12. - 17.12.09
zuletzt wurde an einer Optimierung der Zielfunktion für das Abschießen von Kugeln gearbeitet, da gibt es bislang noch kleinere Ungenauigkeiten.
Als zweites haben wir mit dem Im- und Export der Animationen viel herumprobieren müssen. Dort treten zurzeit immernoch grobe Fehldarstellungen auf.
Während der vorlesungsfreien Zeit werden wir uns darum bemühen eine Lösung für dieses Problem zu finden.
Wie besprochen werden wir darüber hinaus bis Februar keine weiteren (größeren) Arbeiten am Projekt vornehmen.

Im Januar zeigen wir Ihnen gerne den aktuellen Stand des Projekts.

Zeitraum 14.12.2009 – 08.01.2010
in der letzten Dezember Woche, während der Weihnachtspause und der ersten Januar-Woche wurden folgende Arbeiten am Fachprojekt gemacht:

- Angefangen den Quellcode zu Refacturen
- Der Kamerawechsel zwischen First- und Third-Person optimiert
- Die Infrarot-Leiste mit Standfuß und Stromanschluss ausgestattet
- Neue (Größere) Testwelt erstellt
- Problematik des Animationsexports gelöst (leider müssen die Animation komplett nochmal neu Erstellt werden)

Der Abgabetermin ist ja auf den 12.02.10 festgelegt, leider haben wir festgestellt, dass dieser Termin mitten an einem Klausurtag liegt.

Aus zeitlichen Gründen können bis dahin weder die Animationen eingefügt und implementiert werden, noch die Ausarbeitung des Projektberichtes sauber fertiggestellt werden. Des Weiteren haben wir Zeitdruck mit dem AI-Projekt, welches bis Ende Februar abgeschlossen sein muss.
Daher ist unsere Frage ob es möglich ist das Fachprojekt ebenfalls auf Ende Februar zu verschieben.

5.4 Projektstrukturplan und Aktivitätenzeitplan