Umwelt-Campus Birkenfeld, Fachbereich Medieninformatik

Spiele in einer 3D-Welt
- von der Idee bis zur Realisierung

Bachelorarbeit

zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

vorgelegt von

Lukas Kalinowski

Erster betreuender Professor Prof. Dr. Norbert Kuhn

Zweiter betreuender Professor Prof. Dr. Martin Rumpler

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Bachelorarbeit selbstständig und ohne unzulässige fremde Hilfe angefertigt habe.
Die verwendeten Quellen sind vollständig zitiert.
Birkenfeld, den 06. August 2009

(Lukas Kalinowski)

Inhaltsverzeichnis

1 Einleitung 1
1.1 Motivation 1
1.2 Zielsetzung 1
1.3 Aufbau der Arbeit 2

2 Computerspiele 3
2.1 Begriffserklärung 3
2.2 Geschichte und Meilensteine 4
2.3 Genres 8
2.4 Marktanalyse 10

3 Konzept 11
3.1 Idee 11
3.2 Spielbeschreibung 13
3.3 Planung 14

4 Werkzeuge 18
4.1 Grafik-Engine 18
4.2 Physik-Engine 21
4.3 Audio 24
4.4 3D-Grafik-Software 24
4.5 Bildbearbeitungsprogramm 25

5 Einführung in die Werkzeuge 26
5.1 Ogre-Engine 26
5.2 Funktionsweise 26
5.3 Szenen Abfragen 30
5.4 Materialien und Shader 32
5.5 Initialisierung 33
5.6 Objektarten 35
5.6.1 Entities 35
5.6.2 Partikelsysteme 36
5.6.3 Kameras 37
5.7 Landschaft und 3D-Welt 38

5.8 Himmel 39
5.9 Größenordnung 39
5.10 Physik und OgreNewt 40
5.10.1 Physikalische Körper 41
5.10.2 Kollisionsablauf 46
5.10.3 Bewegung und Orientierung 48
5.11 OgreAL 54
5.12 Spielschleife 54
5.13 Performance 56

6 Realisierung 59
6.1 Prinzipielle Vorgehensweise 59
6.2 Modellierung 61
6.3 Material 65
6.4 Beschreibung der Klassen 66
6.5 Editierung der Welt 70
6.6 Welt Laden 71
6.7 Charakter 72
6.7.1 Spieler 76
6.7.2 Gegner 79
6.8 Gegnerverhalten und Künstliche Intelligenz 80
6.9 Kollisionsablauf zwischen Gegner und Spieler 85
6.10 Spiel Speichern 88

7 Fazit 89
7.1 Abschlussbewertung 89
7.2 Arbeitsaufwand 90
7.3 Ausblick 92

Anhang A i

Abkürzungsverzeichnis i

Quellenverzeichnis ii

Abbildungsverzeichnis

Abbildung 2.1: XOX Tic-Tac-Toe-Spiel....4
Abbildung 2.2: Tennis for Two Videospiel....4
Abbildung 2.3: Pong....5
Abbildung 2.4: Pac-Man....6
Abbildung 2.5: Ausschnitt aus Super Mario 64....7
Abbildung 2.6: Ausschnitt aus Gothic 3....7
Abbildung 2.7: Prozentualer Altersanteil der Konsumenten....10
Abbildung 3.1: Skizze Charakter....14
Abbildung 5.1: Struktur des Szenengraphes....27
Abbildung 5.2: Würfel in der Mitte des Weltkoordinatensystems....29
Abbildung 5.3: Position eines Vater- und Kindknotens....30
Abbildung 5.4: Screenshot - Objekt mit einem Quader....31
Abbildung 5.5: Screenshot - Materialdatei....32
Abbildung 5.6: Screenshot - Struktur einer Ressourcen-Datei....34
Abbildung 5.7: Screenshot: 3D-Model in Ogre dargestellt....36
Abbildung 5.8: Screenshot - VWE....39
Abbildung 5.9: Verschiedene sphärische Formen....41
Abbildung 5.10: Verschiedene abgerundete Zylinder....42
Abbildung 5.11: Modell als Mesh und konvexe Hülle....42
Abbildung 5.12: Eulersche Rotationsachsen....48
Abbildung 5.13: Drehungen in verschiedenen Reihenfolgen....49
Abbildung 5.14: Kardanische Aufhängung....50
Abbildung 5.15: Orientierung im Bezug zur negativen Z-Achse....53
Abbildung 5.16: Orientierung im Bezug zur X-Achse....53
Abbildung 5.17: Screenshot - Clip-Culling....57
Abbildung 5.18: Sicht als Kegelstumpf (engl. view frustum)....57
Abbildung 6.1: Screenshot - Charakter Grundkörper....62
Abbildung 6.2: Screenshot - Skelett....62
Abbildung 6.3: Screenshot - Geh Animation....63
Abbildung 6.4: Screenshot - Charakter Textur....64
Abbildung 6.5: Screenshot - Charakter texturiert....64
Abbildung 6.6: Screenshot - Charakter ohne und mit Shader....65
Abbildung 6.7: Screenshot - Materialdatei des Charakterkörpers....66

Abbildung 6.8: Klassenhierarchie des Spiels.....66
Abbildung 6.9: Screenshot - 3D-Welt von Flubbers.....71
Abbildung 6.10: Screenshot - Kollisionshüllen von Bäumen.....72
Abbildung 6.11: Screenshot - Kollisionshülle.....73
Abbildung 6.12: Screenshot - Tret-Animation.....78
Abbildung 6.13: Screenshot - Brauner Gegner.....79
Abbildung 6.14: Zustandsdiagramm Gegnerverhalten.....81
Abbildung 6.15: Screenshot - Tret-Attacke.....84

Tabellenverzeichnis

Tabelle 3.1: Objektarten.....15
Tabelle 4.1: Übersicht von Open Source Grafik-Engines.....20
Tabelle 4.2: Gegenüberstellung verschiedener Engines.....20
Tabelle 4.3: Gegenüberstellung Physik-Engines.....22
Tabelle 4.4: Physik-Engine und Wrapper.....23
Tabelle 4.5: Gegenüberstellung unterstützter Features.....23
Tabelle 4.6: 3D-Grafik-Software Programme mit Konverter.....25
Tabelle 6.1: Animationsarten des Charakters.....63
Tabelle 6.2: Gegnerattribute.....79
Tabelle 7.1: Anzahl verwendeter Ressourcen.....90
Tabelle 7.2: Anzahl an Codezeilen.....91
Tabelle 7.3: Zeitaufwand für das Spiel "Flubbers".....91

1 Einleitung

Unser Alltag wird heutzutage durch elektronische Medien wie insbesondere dem Fernsehen oder Internet geprägt. Kaum jemand kann vor der Ausbreitung der Medien entfliehen. Der rapide Fortschritt der Technik im Computerbereich erlaubt immer realistischere Darstellungen von Computergrafiken um 3D-Objekte auf dem Bildschirm darzustellen, weshalb auch immer mehr Fernsehfilme oder Werbungen mit Grafikanimationen verknüpft werden.

Ein weiterer stetig wachsender Bereich sind Computerspiele, welche in den letzten Jahren immer mehr an Bedeutung gewonnen haben. Sie sind aus unserer modernen Kultur nicht mehr wegzudenken. Die heutige Generation wächst unter dem Einfluss dieser virtuellen Welt auf. Aber auch ältere Menschen werden von Computerspielen angezogen.

1.1 Motivation

Computerspiele dienen vor allem der Unterhaltung und werden von allen Altersschichten genutzt. Da unsere Zivilisation mit den Medien wächst, werden diese in unserem Alltag selbstverständlich. Das rapide Wachstum der Computer-Technologie führt auch dazu, dass Computerspiele immer komplexer und realistischer werden. Viele Spielkonsumenten sind sich oft nicht bewusst, wie eigentlich der Entstehungshintergrund eines Spiels aussieht, beziehungsweise wie aufwendig es ist ein solches zu entwickeln. Heutzutage werden jedoch immer mehr Werkzeuge entwickelt, welche die Berechnung und Darstellung von 3D-Szenen erheblich erleichtern. Der Autor interessiert sich von klein auf für Computerspiele. Durch sein Studium erlangte er wertvolle Erkenntnisse, wie ein Spiel entwickelt wird, deshalb werden in dieser Arbeit notwendige Schritte von der Idee bis zur Realisierung eines Spiels in einer 3D-Welt beschrieben.

1.2 Zielsetzung

Ziel dieser Arbeit ist es, einen Eindruck zu vermitteln, welche Schritte zu vollziehen sind, um Spiele in einer 3D-Welt zu entwickeln.

Der Autor möchte seine im Studium erworbenen Kenntnisse einsetzen, um ein realitätsnahes Spiel in einer 3D-Welt mit physikalischen Abläufen zu entwickeln. Um dies zu ermöglichen, wird eine Einführung in eine Grafik- und Physik-Engine gegeben.

Des Weiteren wird ein Augenmerk auf die künstliche Intelligenz gelegt, mit welcher die Gegner ausgestattet werden.

1.3 Aufbau der Arbeit

Im Kapitel „Computerspiele“ werden einige Begriffe erklärt. Außerdem wird auf die Entstehungsgeschichte, sowie auf eine Marktanalyse eingegangen. Kapitel „Konzept“ gibt Aufschluss um aus einer anfänglichen Idee bis zur Planung eines bestimmten Spiels überzugehen. Damit ein Spiel in einer 3D-Welt entwickelt werden kann, werden in Kapitel „Werkzeuge“ einige Ressourcen vorgestellt, mit deren Hilfe dies ermöglicht wird. Ab Kapitel „Einführung“ erfolgt der praktische Teil dieser Arbeit. In diesem wird eine Einführung in die verwendeten Werkzeuge, sowie in die wesentlichen mathematischen und physikalischen Berechnungen gegeben. Im Kapitel „Realisierung“ werden sämtliche notwendige Schritte erläutert, um das geplante Spiel zu realisieren. Im Letzten Kapitel wird ein Fazit über die Realisierung des Spiels gezogen, sowie ein Ausblick auf eine zukünftige mögliche Weiterführung gegeben. Das komplette Spiel mitsamt des Quellcodes befindet sich auf der beiliegenden CD. Im Hauptordner dieser CD ist eine Anleitung vorhanden, in welcher beschrieben ist, wie das Spiel ausgeführt wird.

2 Computerspiele

In diesem Kapitel wird die Bedeutung von Computerspielen definiert. Zunächst wird erklärt, was ein Computerspiel ausmacht, danach wird auf die Geschichte und Meilensteine eingegangen. Anschließend werden verschiedene Rubriken von Spielen vorgestellt. Zum Abschluss dieses Kapitels werden mittels einer Marktanalyse einige Fakten zum Thema Computerspiele näher beleuchtet.

2.1 Begriffserklärung

Eine allgemeine und genaue Definition, was eigentlich ein Computerspiel ausmacht, kann nur schwer gegeben werden, da ein Jeder das Wort „Computerspiel“ mit Sicherheit anders definieren würde. Eine mögliche Definition des Autors lautet wie folgt:

Als Computerspiel wird ein Medium[1] bezeichnet, in welchem der Spieler einen virtuellen Charakter in einer Welt bewegt und dabei vorgegebene Aufgaben erfüllt um ans Ziel zu gelangen. Er handelt also interaktiv und der Spielablauf ist von seinen Aktionen abhängig. Bei anderen Medien wie zum Beispiel dem Fernsehen, kann der Konsument das Geschehen nicht beeinflussen, er kann es nur passiv erfahren.

Im Allgemeinen werden Computerspiele auch als „Videospiele“, oder im umgangssprachlichen Gebrauch als „Games“ bezeichnet.[2] Wobei der Begriff „Videospiel“ früher eine etwas andere Bedeutung hatte. In den 70er Jahren machte man sich die Erkenntnisse aus der Fernsehtechnologie zunutze und entwickelte Spielautomaten.[3] Ein klare Unterscheidung zwischen Computerspielen und Videospielen existiert heute nicht mehr. Der einzige Unterschied besteht darin, dass Videospiele mittels einer Spielkonsole[4] und Computerspiele auf dem Computer gespielt werden. Technisch gesehen ist eine Xbox 360 oder eine Playstation 3 auch ein Computer, nur das ein Computer nicht nur zum Spielen gedacht ist, sondern auch ein viel breiteres Arbeitsspektrum umfasst. Eine Spielkonsole hingegen dient dem reinen Spielvergnügen. Der Begriff Computerspiel und Videospiel wird daher im Verlauf dieser Arbeit als synonym verwendet.

1 Ein Medium ist ein Kommunikationsmittel, es überträgt Informationen zwischen dem Sender und dem Empfänger.

2 Vgl. http://de.wikipedia.org/wiki/Computerspiel, Internet: 17.06.2009

3 Vgl. http://de.wikipedia.org/wiki/Geschichte_der_Videospiele, Internet: 17.06.2009

4 Daher werden Videospiele auch Konsolenspiele genannt.

2.2 Geschichte und Meilensteine

Im Folgenden wird kurz auf die Geschichte und die Meilensteine der Computerspiele anhand der Quellen [2] und [9, S. 17] eingegangen.

Der Prozess der Spieleentwicklung geht zurück bis in die 50er Jahre. Ein genaues Datum der Entstehung von Videospielen kann jedoch nicht festgelegt werden. Einige Experten sehen in dem Physiker William Higinbotham den Erfinder, andere in der Forschungsgruppe um Alan Kotok und Steve Russell. Die Entstehung muss als ein Werdegang gesehen werden, da die Entwicklung im Zusammenhang mit den technischen Gegebenheiten der jeweiligen Zeit steht. Das erste grafische Computerspiel dessen Name bekannt wurde hieß XOX. Es ist auch unter dem Namen Tic-Tac-Toe geläufig und wurde 1952 von A. Sandy Douglas entwickelt. Abbildung 2.1 zeigt das erste grafische Spiel nach Quelle [3].

Abbildung 2.1: XOX Tic-Tac-Toe-Spiel

William Higinbotham hatte das Ziel, den Fernseher in ein aktives Medium weiter zu entwickeln. 1958 präsentierte er das Videospiel Tennis For Two. Die aus der Quelle [4] zu entnehmende Abbildung 2.2 zeigt das Spiel.

Abbildung 2.2: Tennis for Two Videospiel

Das erste Videospiel welches für Erfolg in den Spielhallen sorgte, war Pong (vgl. die Abbildung 2.3 aus Quelle [5]). Es wurde 1972 von dem Elektrotechniker Nolan Bushnell entwickelt. Er war der erste, der mit seinem Spiel Gewinn machte, was ihm ermöglichte die Firma Atari[5] zu gründen. Bei Pong wird ein Ball durch Abprallen an einer senkrechten Linie, dem sogenannten Schläger, auf dem Bildschirm bewegt. Die Schläger werden von den Spielern gesteuert. Fliegt der Ball an einem Schläger vorbei, so erhält der Gegenspieler einen Punkt.

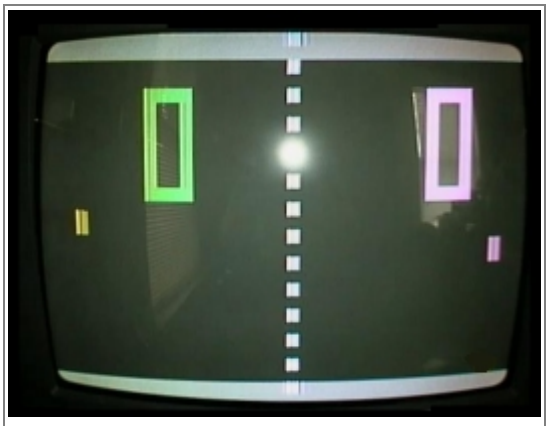

Abbildung 2.3: Pong

Im Jahr 1977 wurde das asiatische Unternehmen Nintendo[6] gegründet. 1980 wurde das Videospiel Donkey Kong produziert. Bei diesem musste der Spieler den Charakter, die spätere Heldenfigur Mario[7], ein Baustellengerüst erklimmen lassen, sämtlichen Gefahren ausweichen und eine Prinzessin aus der Gewalt des bösen Gorillas Donkey Kong[8] zu befreien.

Zur gleichen Zeit erschien das Spiel Pac-Man der Firma Namco[9], welches heute in unzähligen verschiedenen Versionen erhältlich ist. In diesem Spiel muss der Charakter alle im Labyrinth befindlichen Punkte verschlingen, dabei darf er nicht mit den im Labyrinth befindlichen Geistern in Berührung kommen, ansonsten muss er von vorne beginnen. Die Abbildung 2.4 aus Quelle [6] zeigt eine Version des Spiels.

5 http://www.atari.com, Internet: 19.06.2009

6 http://www.nintendo.de, Internet: 19.06.2009

7 http://www.mobygames.com/game/gameboy/super-mario-land/release-info, Internet 19.06.2009

8 http://www.mobygames.com/game/donkey-kong, Internet 19.06.2009

9 http://www.namcogames.com, Internet: 19.06.2009

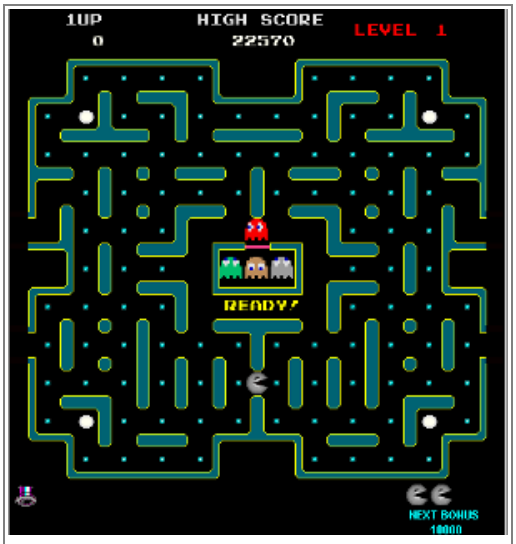

Abbildung 2.4: Pac-Man

Des Weiteren wurden in den 80er Jahren die ersten Heim- und Personal Computer (PC) entwickelt, wobei Heimcomputer zum Spielen und PCs zum Arbeiten vorgesehen waren. Zu diesem Zeitpunkt entstand die Abgrenzung zwischen Computer- und Konsolenspielen. In Amerika wurde die Entwicklung von PCs vorangetrieben, während im Westen begonnen wurde, Heimcomputer wie zum Beispiel Commodores C64[10] und Amiga[11], zu entwickeln. Im Osten spezialisierte man sich auf die TV-gebundenen Spielkonsolen.

Eines der ersten Spiele bei welchen sich der Spieler durch eine dreidimensionale Welt bewegt, war der Ego-Shooter[12] Wolfenstein 3D und ist 1992 erschienen. Kurz darauf wurde das Spiel allerdings wegen Gewaltverherrlichung und nationalsozialistischen Inhalten auf den Index gesetzt.

Das erste 3D Jump 'n' Run Spiel welches nach [45] als Meilenstein gilt, ist Super Mario 64, es wurde 1996 von Nintendo veröffentlicht. Bei diesem Spiel muss die Heldenfigur Mario die entführte Prinzessin befreien, dabei muss er in verschiedene Welten reisen und dort diverse Aufgaben meistern. Das Spiel bietet eine offene dreidimensionale Welt mit einem Schloss als Ausgangspunkt. Zu den einzelnen Orten in der Welt gelangt man indem man verschiedene Portale, die in Form von Bildern an Wänden im Schloss hängen, benutzt. Abbildung 2.5 aus [46] zeigt einen Screenshot des Spiels.

10 http://de.wikipedia.org/wiki/Commodore_64, 19.06.2009

11 http://amiga.com, 19.06.2009

12 Siehe Unterabschnitt 2.3

Abbildung 2.5: Ausschnitt aus Super Mario 64

Im Jahr 2006 kam laut [47] das 3D-Rollenspiel Gothic 3 auf den Markt. In diesem Spiel steuert der Benutzer einen Charakter aus der Third-Person-Perspektive[13] durch eine riesige dreidimensionale Welt. Der Charakter spielt dabei die Hauptfigur in einer umfangreichen Story, hat aber unabhängig von dieser Rahmenhandlung die Möglichkeit die Welt zu erkunden. Er muss gegen verschiedene Gegner wie zum Beispiel Menschen, Orks, Tiere usw. kämpfen. Die aus [48] stammende Abbildung 2.6 zeigt einen Ausschnitt des Spiels.

Abbildung 2.6: Ausschnitt aus Gothic 3

Heutzutage erscheinen in kurzen Abständen verschiedenste Spiele für eine Vielfalt an Konsolen und den PC.

13 Siehe Kapitel 5.6.3

2.3 Genres

Spiele lassen sich gemäß [8, S. 81] je nach der Art der Interaktion und der zu bewältigenden Aufgabe in unterschiedliche Genres einteilen. Anders als bei Filmen oder Büchern, bei welchen der Inhalt maßgeblich für die Einstufung in eine bestimmte Rubrik ist, wird das Genre bei einem Spiel unabhängig von seinem Inhalt bestimmt.
Um ein Beispiel zu nennen, ist ein Actionspiel das im Weltraum oder in der Wüste spielt, immer noch ein Actionspiel. Das Genre eines Spiels ist abhängig vom technischen Standard. Da es nicht möglich ist die Realität nachzubilden, wird bei einem Spiel nur ein bestimmter Bereich simuliert. Bei dem im Unterabschnitt 2.2 erwähnten Spiel Pong besteht die einzige Aufgabe darin, den Ball nicht am Schläger vorbei fliegen zu lassen, es kann somit in das Genre „Geschicklichkeit“ eingestuft werden.
Heute ist eine eindeutige Klassifizierung eines Spiels in ein bestimmtes Genre nur eingeschränkt möglich, da Spiele immer aufwendiger werden. So können zum Beispiel im 3D Action-Shooter Doom 3[14] in einem Spielautomaten verschiedene, tatsächlich existierenden Spielen nachempfundene „Minigames“ gespielt werden.[15] Spiele lassen sich also in mehrere Kategorien einordnen, wobei eine Kategorie aus mehreren Unterkategorien bestehen kann. Das in Kapitel 6 zu realisierende Spiel kann auch keinem bestimmte Genre zugeordnet werden.
Im Folgenden werden einige Hauptkategorien nach [7] und [8] beschrieben:

- **Action**
  In einem Actionspiel muss der Spieler unter Zeitdruck verschiedene Aufgaben durch Geschicklichkeit meistern. In der Regel geht es darum, durch kämpferischen Einsatz Herausforderungen zu bewältigen.

- **Adventures**
  In diesem Genre ist der Spieler in den Ablauf des Spiels involviert. Dabei müssen Aufgaben oft in Form von Rätseln gelöst werden. Durch eine Folge von Aktionen des Spielers, welche im Gegensatz zum oben beschriebenen Genre „Action“ ohne Zeitdruck stattfinden, entsteht eine Geschichte.

14 http://www.idsoftware.com/games/doom/doom3, Internet: 19.06.2009

15 http://www.yiya.de/reviews/d/doom0302.shtml, Internet: 19.09.2009

- **Jump 'n' Run**
  In diesem Genre muss der Spieler durch ein oder mehrere Welten laufen und verschiedene Aufgaben erledigen. Er muss zum Beispiel über Hindernisse springen, versteckte Gegenstände ausfindig machen, über Klippen springen und so weiter.

- **Strategie**
  Strategiespiele erfordern taktisches und zumeist logisches Denken um die gegebenen Anforderungen zu meistern. Bei Warcraft 3[16] zum Beispiel muss der Spieler Festungen bauen, Kriege führen und diese durch taktisches Handeln gewinnen.

- **Rollenspiele**
  In Rollenspielen muss der Spieler verschiedene Aufträge ausführen, er kann also die Geschichte aktiv mitgestalten. Es können meistens mehrere Charaktere gewählt werden, von denen jeder einzigartige Eigenschaften besitzt. Ein Schwertkämpfer ist beispielsweise sehr stark, ein Bogenschütze hingegen kann Gegner durch geschickten Einsatz von Pfeil und Bogen besiegen. Eine besondere Charakteristik bei Rollenspielen ist die Aufstiegsmöglichkeit nachdem zahlreiche Gegner besiegt wurden, was zur Erhöhung der Fähigkeitspunkte eines Charakters führt. Bei diesem Genre geht es auch darum Gegenstände zu sammeln, sich mit diesen auszurüsten, oder sie zu verkaufen.

- **Shooter- / Ego-Shooter**
  Diese Art zeichnet sich dadurch aus, dass der Charakter meist in der „Ich-Perspektive“[17] durch mehrere Welten bewegt und mit Hilfe von leistungsstarken Schusswaffen Gegner bekämpft.

Dies war nur ein grob beschriebener Ausschnitt der bekanntesten Kategorien. Es existieren noch weitaus mehrere Genres, sowie Kombinationen dieser.

16 http://eu.blizzard.com/de/war3, Internet: 19.06.2009

17 Siehe Kapitel 5.6.3

2.4 Marktanalyse

Seit es Computerspiele gibt, erfreuen diese sich wachsender Beliebtheit. Die Organisation ESA[18] veröffentlichte 2008 eine Studie über die Computer- und Videospiel Industrie in den USA. Diese hat nach [24] ergeben, dass in 65% der Haushalte Computer- und Videospiele konsumiert werden. Abbildung 2.7 veranschaulicht die Aufteilung dieser 65% in Altersklassen:

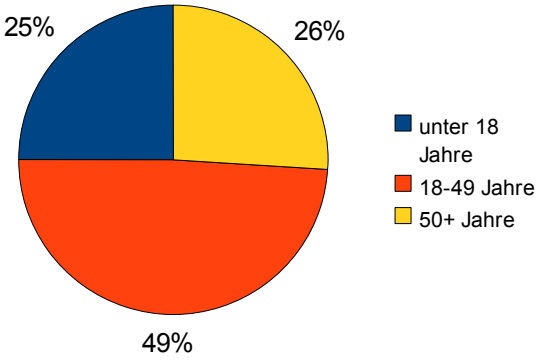

Abbildung 2.7: Prozentualer Altersanteil der Konsumenten

Aus diesen Zahlen wurde das Durchschnittsalter eines Spielkonsumenten auf 35 Jahre ermittelt. Zudem wurde analysiert, dass der Anteil des weiblichen Geschlechts 40% betrug, was angesichts der Tatsache, dass Computerspiele heutzutage immer noch als Männerdomäne angesehen werden, überrascht. Des Weiteren wurde festgestellt, dass das Alter von Spielern, die Computer- oder Videospiele kaufen sich im Schnitt auf 40 Jahre beläuft, woraus hervorgeht, dass Spiele nicht nur von Jugendlichen genutzt werden, sondern von allen Altersschichten. In Deutschland könnte es sich ähnlich verhalten.

Nach Quelle [25], verzeichnete die Spielindustrie 2001 in den USA einen Umsatz von 9,4 Milliarden Dollar. Dieser Umsatz überstieg damals das erste Mal den der Filmindustrie in den USA, welcher sich auf 8,35 Milliarden Dollar belief. Im Januar 2009 wurde nach [26] veröffentlicht, dass die Spielindustrie 2008 einen Gewinn von 11,7 Milliarden Dollar erzielte.

18 The Entertainment Software Association

3 Konzept

Die Realisierung eines Spiels in einer 3D-Welt ist ein langer Prozess mit vielen Hindernissen, die es zu bewältigen gilt. Zuerst muss ein Spielkonzept erstellt werden. In diesem Kapitel werden zunächst allgemeine notwendige Informationen gegeben um von einer anfänglichen Idee in die Spielbeschreibung überzugehen. Anschließend wird ein vom Autor festgelegtes „Muster-Spiel“ geplant, welches in Kapitel 6 realisiert wird. Sämtliche Schritte werden begleitet durch die Quellen [8, S. 74-83, S. 158] und [22] durchgeführt.

3.1 Idee

Am Anfang der Entwicklung eines Spiels in einer 3D-Welt stehen zahlreiche kreative Überlegungen, die sich immer weiter verdichten, bis sich ein konkreteres Konzept herauskristallisiert. „Ideen zu entwickeln hat eine gewisse Ähnlichkeit mit dem Lösen eines Puzzlespiels. Am Anfang hat man zwar ein Ziel, kann aber vor lauter losen Teilen kein Bild sehen. Nur Chaos.“[19] Der Autor hat sich folgende Fragen gestellt um dieses im zuvor genannten Zitat erwähnte „Chaos“ zu ordnen und gibt zugleich mögliche Antworten anhand von Beispielen:

- Worum soll es im Spiel gehen?
  Der Held soll eine entführte Person oder einen gestohlenen wichtigen Gegenstand aus den Fängen des Bösen befreien.
  Das Böse schreitet immer weiter voran und muss bekämpft werden.
  Der Held soll alle Hindernisse überwinden und an einen festgelegten Ort als Ziel gelangen.
  Es soll eine Stadt aufgebaut werden.
  Kriege gegen feindliche Armeen sollen geführt werden.

- Wie sollen der Charakter und die Gegner aussehen?
  Der Charakter und die Gegner sollen der menschlichen Form ähneln.
  Sie sollen in der Gestalt eines Tieres oder skurriler Fantasiewesen auftreten.
  Es sollen Wesen, die mythischen Geschichten entnommen wurden, wie beispielsweise Drachen, Zauberer, griechische Götter etc., sein.

19 http://www.wie-ideen-entstehen.de/intro.htm, Internet: 24.06.2009

Der Charakter ist ein Erdbewohner und die Feinde sind außerirdische Wesen.

- Zu welcher Zeit soll die Geschichte spielen?
  Das Spiel läuft im Mittelalter, Gegenwart oder Zukunft ab.
  Das Spiel ist keiner Zeit zugehörig.

- Welche Gestalt soll die Spielwelt haben?
  Die Welt ist ein fremder Planet.
  Teile der Erde sollen nachempfunden werden.
  Es existieren mehrere Welten.
  Die Spielwelt soll keine Welt im eigentlichen Sinne sein, sondern ein Weltraum.
  Sie soll sich in bestimmten abgegrenzten Schauplätzen, wie zum Beispiel einer Stadt, abspielen.
  Es soll eine Fantasiewelt frei erfunden werden.

- Wie sollen Gegner besiegt werden?
  Gegner sollen durch geschickte Kampfeinsätze eliminiert werden.
  Sie sollen mit Hilfe von Schusswaffen vernichtet werden.
  Der Spieler muss vor den Gegner flüchten und darf nicht gefangen werden.

- In welches Genre lässt sich das Spiel einordnen?
  Im Kapitel 2.3 wurde auf verschiedene Genres eingegangen, mögliche Antworten können dort entnommen werden.

Diese Liste kann beliebig fortgeführt werden.

Außerdem kann sich an anderen Medien für eine Spielidee orientiert werden. Es existieren unzählige Filme und Bücher, die inspirieren können. Des Weiteren könnten sich bereits im Markt verfügbare Spiele für eine Anregung sorgen.
Um eine grobe Vorstellung über das zu entwickelnde Musterspiel zu erhalten, werden im konkreten Fall folgende Antworten gewählt:

- Der Held soll einen gestohlenen wichtigen Gegenstand aus den Fängen des Bösen zu befreien.
- Der Charakter und die Gegner sollen in der Gestalt eines Tieres oder skurriler Fantasiewesen auftreten.
- Das Spiel ist keiner Zeit zugehörig.
- Es soll eine Fantasiewelt frei erfunden werden.
- Gegner sollen durch geschickte Kampfeinsätze eliminiert werden.
- Das Spiel soll dem Genre 3D-Jump 'n' Run zugehörig sein.

3.2 Spielbeschreibung

Dieser Abschnitt gibt näheren Aufschluss über das Prinzip und die wichtigsten Features eines Spiels, er ist wie eine Anleitung, an welcher sich orientiert werden kann.
Im Laufe des vorherigen Abschnitts wurde aus der Idee durch die vom Autor gewählten Antworten ein grober Ablauf des Spiels festgelegt. Daraus entsteht nun die Geschichte des Spiels und die Spielbeschreibung:
Ein Dorf wird von Feinden angegriffen, mit dem Ziel ein goldenes Artefakt zu stehlen. Dieses Artefakt dient dem Schutz vor böser Magie. Der Held hat das Ziel, dieses Artefakt wieder zu erlangen. Er muss sich durch eine große abstrakte Welt kämpfen, in der überall Gegner lauern, welche er durch geschickte „Tret-Attacken“ eliminieren muss. Als Resultat erhält er Erfahrung pro vernichteten Gegner. Hat er genug Gegner zerstört, steigt er um einen Level, dadurch bekommt er mehr Energie und Stärke. Gegner bekommen abhängig von der Stärke des Spielers viel Energie abgezogen und der Spieler abhängig von der Stärke eines Gegners. Des Weiteren sind überall in der Welt Gegenstände versteckt, findet der Spieler eine Hähnchenkeule, erhält er Stärke, entdeckt er einen Energiekasten, wird seine Energie aufgefüllt und bei einem Herz bekommt er ein Leben dazu. Um zum Ort, an welchem das Artefakt bewacht wird zu gelangen, muss der Spieler viele Hindernisse meistern. Er muss mittels eines Trampolins über Klippen springen, sich auf Förderbändern halten, Kisten zurecht schieben um auf höher gelegene Objekte zu gelangen und in Gebäude gehen um Gegenstände aufzusammeln. An bestimmten Stellen in der Welt befinden sich Speicherstellen, erreicht der Spieler eine solche, so wird der aktuelle Spielstand abgespeichert. Wenn er keine Energie mehr hat, wird er an die zuletzt besuchte Speicherstelle gebracht. Im Verlauf der Zeit wechseln sich Tag und Nacht ab und Gegner werden nachts aggressiver.

Nach Vollendung der Spielbeschreibung ist klar was zu tun ist.

3.3 Planung

Nachdem die Beschreibung des Spiels steht, kann noch nicht sofort mit der Realisierung begonnen werden, da zuerst noch einige Schritte geplant werden müssen. Diese stellen eine mögliche Reihenfolge der Vorgehensweise bei der Planung eines Spiels dar. Zudem werden dadurch die Anforderungen an das Spiel ermittelt:

Schritt 1: Recherche nach geeigneten Werkzeugen

Zu Beginn muss festgestellt werden, welche technischen Mittel zur Verfügung stehen, und was sich im Rahmen des Möglichen befindet. In Kapitel 4 werden einige Tools vorgestellt, welche zur Entwicklung eines eindrucksvollen 3D-Spiels beitragen könnten.

Schritt 2: Charaktere

Folglich werden sich Gedanken über das Aussehen, des Spielers und der Gegner gemacht. Die Bewegungen des Spielers und der Gegner soll realistisch wirken, deshalb müssen die Modelle animiert werden. Eine realistische Geh-Animation ist kompliziert und sehr aufwendig zu bewerkstelligen, deswegen werden die Charaktere eine einfache Form haben und nur aus zwei Abschnitten bestehen, dem Ober- und Unterteil. Durch Animation des Unterteils, soll der Charakter hüpfen. Abbildung 3.1 stellt eine vom Autor gefertigte Skizze eines Charakters dar:

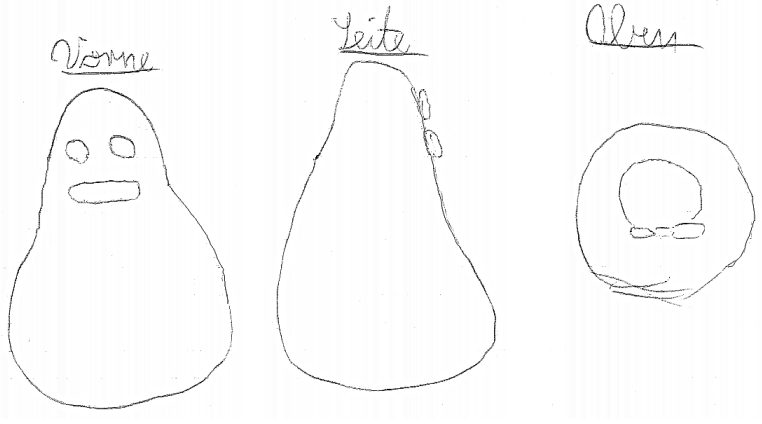

Abbildung 3.1: Skizze Charakter

Mit Hilfe dieser Skizze, kam dem Autor die Idee seinen Charakteren eine gummiartige Konsistenz zu verleihen, die einen an Wackelpudding erinnert. Dies brachte ihn dazu, dem Spiel den Titel „Flubbers“ zu geben.

Schritt 3: Umgebung

Es müssen sich Gedanken über die Umgebung gemacht werden. Eine flache Welt würde dem Spieler kaum einen Anreiz bieten, es muss also ein Werkzeug her, mit welchem sich komplexe Landschaften formen lassen. In Kapitel 5.7 wird ein Tool vorgestellt, welches sich hervorragend für diese Aufgabe eignet. Damit die Welt belebt wird, muss sie mit verschiedenen Objekten ausgestattet werden. Tabelle 3.1 liefert einen Überblick über die in Erwägung zu ziehenden Objekte:

Natur (feste Objekte)	**Feste Objekte**	**Lose Objekte**
Bäume	Bunker	Bänke
Pflanzen	Häuser	Stühle
Büsche	Trampoline	Tische
Steine	Plattformen	Kisten
Felsen	Förderbänder	Fässer
	Mauern	Kugeln
	Hängebrücken	Heilpäckchen
	Laternen	Hähnchenkeulen
	Speicherstellen	Herzen
	Betten	Kanonen

Tabelle 3.1: Objektarten

Aufgrund dieser großen Aufzählung stellt sich die Frage, ob alle Objekte selbst modelliert oder ob teilweise fertige freie Modelle zur Verstärkung im Internet herangezogen werden sollen. Alle Modelle müssen zudem mit Texturen versehen werden. Zu diesem Zweck geeignete Werkzeuge werden in Kapitel 4.4 und 4.5 aufgeführt.

Schritt 4: Audio

Ein Spiel soll nicht geräuschlos vonstatten gehen, daher werden Geräuscheffekte und eine Hintergrundmusik unabdingbar. Geräuscheffekte können mit einem

Mikrofon selbst aufgenommen, oder aus dem Internet genommen werden. Eine Hintergrundmusik selbst zu komponieren hingegen, erfordert Kenntnisse auf diesem Gebiet, weshalb in Betracht gezogen wird, nach geeigneten freien Musikstücken im Internet zu suchen, in Kapitel 4.3 wird näher auf mögliche Ressourcen eingegangen.

Schritt 5: Programmstruktur

Die Struktur des Programms, beispielsweise welche Klassen benötigt werden und wie diese miteinander verknüpft sind, muss geplant werden. Dies wird über ein Klassendiagramm veranschaulicht.

Schritt 6: Intro

Nach Vollendung der Vorarbeit erfolgt nun die Planung des eigentlichen Spiels. Zuerst muss ein Spielintro erstellt werden. Dort erscheint der Spieltitel sowie die Vorgeschichte.

Schritt 7: 3D-Welt

Anschließend muss die 3D-Welt editiert und in das Spiel geladen werden. Dabei kann die Benutzung eines Leveleditors eine große Rolle spielen.

Schritt 8: Spieler

Der Akteur soll über die Tastatur bewegt und über die Maus orientiert werden. Die Kameraperspektive soll sich auf 3rd-Person[20] belaufen.

Schritt 9: Gegner und Verhalten

Gegner sollen eine künstliche Intelligenz[21] besitzen und abhängig von dieser auf den Spieler reagieren. Darüber hinaus wird ein Kampfsystem realisiert, damit Gegner angreifen und vernichtet werden können. Das Verhalten und die Bewegungen der Feinde werden über Aktionen gesteuert, je nach Aktion wird die entsprechende Animation abgespielt. Alle Spieler und Gegneraktionen sollen durch entsprechende Geräuscheffekte begleitet werden. Außerdem bietet sich ein bestimmtes Werkzeug an (beschrieben in Kapitel 5.8), das die Berechnung von Tag, Nacht und der

20 Siehe Kapitel 5.6.3

21 Siehe Kapitel 6.8

Spielzeit ermöglicht. Gegner orientieren sich an der Tageszeit und sollen sich nachts aggressiver verhalten als tagsüber.

Schritt 10: Spielablauf

Der Spielablauf kann aus Unterabschnitt 3.2 entnommen werden.

Schritt 11: Speichern und Laden eines Spielfortschritts

In der Welt sollen sich mehrere Speicherstellen befinden. Beim Betreten einer solchen, soll der Spielstand abgespeichert werden, damit der Akteur für den Fall, dass er „stirbt“ nicht wieder von vorne beginnen muss.

Schritt 12: Ziel

Erreicht der Spieler den Ort an welchem sich das Artefakt befindet und besiegt den Wächter, ist das Spiel erfolgreich beendet.

In Kapitel 2.3 wurde darauf hingewiesen, dass ein Spiel nicht unbedingt zu einer Kategorie gehören muss, sondern mit anderen kombiniert werden kann. Im Verlauf dieses Kapitels stellte sich heraus, dass das geplante Spiel nicht nur in das Genre 3D-Jump 'n' Run passt, sondern auch zum Teil der Kategorie Rollenspiele zugeordnet werden kann, da nur eine große Welt verwendet wird und die zuvor erwähnte Aufstiegsmöglichkeit besteht.

4 Werkzeuge

Ein 3D Spiel ohne jegliche Hilfe von Werkzeugen zu realisieren ist nur mit immensem Wissen, Erfahrung sowie Zeit- und Arbeitsaufwand möglich. Die Computer-Technologie wird immer effektiver und komplizierter und eine Menge Leute betreiben viel Aufwand um Werkzeuge daran anzupassen.
Es existiert eine große Anzahl an Werkzeugen, die zur Entwicklung eines Spiels beitragen könnten, viele davon sind allerdings nicht frei. In diesem Kapitel wird eine Übersicht an notwendigen Werkzeugen gegeben. Zum Schluss jedes Abschnitts wird jeweils eines davon ausgewählt, und kurz erläutert wieso ausgerechnet dieses verwendet wird.

4.1 Grafik-Engine

Als Grafik-Engine wird nach Quelle [16] ein Programm bezeichnet, welches für die Abbildung der Computergrafik verantwortlich ist. Sie wird häufig verwendet um realistische dreidimensionale Szenen zu erzeugen. Eine Grafik-Engine greift auf Programmierschnittstellen (engl. API) von Rendersystemen zu. Sie abstrahiert die Schnittstellen soweit, dass es für den Entwickler transparent ist, wie der Zugriff erfolgt. Dadurch ist es auch ohne Wissen über die Schnittstellenfunktionalität möglich Echtzeitgrafik-Software zu entwickeln. Echtzeitgrafik beziehungsweise Echtzeitrendern bezeichnet die Möglichkeit viele Bilder hintereinander schnell zu berechnen damit, wie bei einem Film ein flüssiger Ablauf entsteht. Der einzige Unterschied besteht darin, dass der Benutzer bei einer Echtzeitgrafik durch Aktionen die Szene verändern kann und die Grafik daher wieder schnell gerendert werden muss, damit das Ergebnis für den Benutzer sofort sichtbar wird.[22] Daher legen Grafik-Engines für Computerspiele die Priorität auf die Rechengeschwindigkeit zulasten der realistischen Darstellung. Im Gegensatz zum Echtzeitrendern benutzt die Filmindustrie das Offline-Rendering Prinzip um Animationsfilme zu produzieren oder auch Spezialeffekte in gewöhnliche Spielfilme einzusetzen. Offline-Rendering bedeutet, dass mit Hilfe einer 3D-Grafik-Software[23] eine Szene aufgebaut wird, wobei vorher festgelegt wurde, was gerendert werden soll. Daraufhin erfolgt die Berechnung eines Bildes, die mehrere Stunden dauern kann. Beim Offline-Rendering wird versucht, die Szene so realistisch wie

22 Vgl. http://de.wikipedia.org/wiki/Bildsynthese, Internet: 29.06.2009

23 Siehe Unterabschnitt 4.4

möglich zu gestalten, daher müssen Szenen von Animationsfilmen zumeist von vielen Rechnern gleichzeitig berechnet werden. An dem Film „Final Fantasy“[24] und dem gleichnamigen Computerspiels lässt sich der Unterschied zwischen den Medien Film und Computerspiel gut verdeutlichen. Eine Sekunde dieses Films, sprich ca. 24 Bilder, nahmen 90 Stunden Rechenzeit in Anspruch. Im Spiel hingegen müssen diese 24 Bilder in einer Sekunde berechnet werden.
Zuvor wurde erwähnt, dass eine Grafik-Engine auf Schnittstellen von Rendersystemen zugreift. Als Rendersysteme kommen folgende in Betracht:

- **DirectX**
  Dies ist nach Quelle [17] eine Windows Technologie, welche aus einer Anzahl an Schnittstellen besteht. Die Schnittstellen regeln die Kommunikation zwischen einer Grafikkarte, Soundkarte und der Software. DirectX wird unter anderem eingesetzt, um Multimediaanwendungen und Computerspiele zu realisieren. Des Weiteren werden sämtliche Eingabegeräte wie Tastatur, Maus und Joystick zur Steuerung einer Anwendung unterstützt. DirectX kann nur im Zusammenhang mit einem Windows Betriebssystem eingesetzt werden.

- **OpenGL**
  OpenGL ist nach [20, S. 5] und [21] wie DirectX eine Programmierschnittstelle zur Implementierung von Computergrafik. Im Gegensatz zu DirectX ist sie jedoch plattformunabhängig, sie kann also auf allen verfügbaren Betriebssystemen verwendet werden. Eine direkte Unterstützung für Eingabegeräte ist nicht vorhanden. Die Steuerung der Grafikhardware erfolgt über Zustandsautomaten. Das was gezeichnet werden soll, wird also als Zustände verwaltet. Solange der Entwickler keine neue Schnittstellenfunktion aufruft bleiben diese Zustände bestehen.

24 http://www.finalfantasy.de/index.php/cms/FFMOVIE/siteid;29/Final_Fantasy_Der_Film_-_Die_M%C3%A4chte_in_dir/index.html, Internet: 29.06.2009

Neben den populären kommerziellen Engines wie zum Beispiel der „Unreal Engine 3“[25] oder der „Crytec Engine“[26] existieren noch eine Reihe an Open-Source Engines. Tabelle 4.1 gibt einen Überblick über einige Grafik-Engines:

Engines		
CrystalSpace 3D Engine[27]	Horde3D[28]	Irrlicht Engine[29]
Lightfeather 3D Engine[30]	Nebula Device[31]	OGRE[32]
Open Scene Graph[33]		

Tabelle 4.1: Übersicht von Open Source Grafik-Engines

Eine längere Recherche nach Features der Engines aus Tabelle 4.1 hat ergeben, dass die verschiedenen Engines sich darin ähnlich sind. Unterschiede sind nicht ohne weiteres erkennbar. Aufgrund dieser Erkenntnis wird in der folgenden Tabelle 4.2 nur darauf eingegangen, welches Betriebssystem und welche 3D API von den einzelnen Engines unterstützt werden.

	Engine						
Unterstützung	Crystal-Space 3D	Horde3D	Irrlicht	Lightfea-ther 3D	Nebula Device	OGRE	Open Scene Graph
3D API:							
DirectX			•		•	•	
OpenGL	•	•	•	•		•	•
Betriebssystem:							
Windows	•	•	•	•	•	•	•
Linux	•		•	•	•	•	•
Mac OSX	•		•	•	•	•	•
Solaris			•				•

Tabelle 4.2: Gegenüberstellung verschiedener Engines

25 http://www.unrealtechnology.com, Internet: 30.06.2009

26 http://www.crytek.com, Internet: 30.06.2009

27 http://www.crystalspace3d.org/main/Main_Page, Internet: 30.06.2009

28 http://www.horde3d.org, Internet: 30.06.2009

29 http://irrlicht.sourceforge.net, Internet: 30.06.2009

30 http://lightfeather.de/news.php, Internet: 30.06.2009

31 http://nebuladevice.cubik.org, Internet: 30.06.2009

32 http://www.ogre3d.org, Internet: 30.06.2009

33 http://www.openscenegraph.org/projects/osg, Internet: 30.06.2009

Der Unternehmer Walt Colins[34] hat sich die Fähigkeiten und Tauglichkeit der „Crystal Space“-Engine und OGRE[35]-Engine angeschaut und in einer Tabelle[36] einen Vergleich nach bestimmten Kriterien gegeben, und jedes untersuchte Kriterium bewertet.
Vor einigen Jahren bekam der Autor durch einen Kommilitonen einen Einblick in die Ogre-Engine. Durch umfangreiche Tutorials und eine sehr hilfsbereite Community erlangte er das nötige Wissen, um anspruchsvolle grafische Applikationen zu entwickeln. Seither befasst er sich mit dieser Engine und konnte viel Erfahrung sammeln, daher kommt Ogre für das zu realisierende 3D Spiel „Flubbers“ zum Einsatz. Die Engine kann durch viele Komponenten, welche teilweise von Community Mitgliedern entwickelt wurden, erweitert werden.
In den nachfolgenden Abschnitten wird nicht mehr auf allgemein im Internet verfügbare Ressourcen eingegangen, sondern auf Tools, welche mit der Ogre-Engine kooperieren können.

4.2 Physik-Engine

Eine Physik-Engine verhält sich nach Quelle [23, S. 3-5] und Erfahrung des Autors wie ein großer Taschenrechner, sie berechnet alle benötigten mathematischen Funktionen um physikalische Abläufe zu simulieren. Das heißt, alle beweglichen Abläufe von Objekten mit Gravitationseinwirkung, Kollisionsabfragen usw., werden über eine Physik-Engine realisiert. Nachfolgend werden Vor- und Nachteile bei der Benutzung einer Physik-Engine aufgezeigt:

Vorteile

- Aufwand: Da keine Engine eigenhändig entwickelt werden muss, lässt sich somit Zeit sparen.
- Verständnis: Eine Physik-Engine übernimmt die komplizierten Berechnungen, so dass vom Entwickler nur noch Basiswissen gefragt ist um die Funktionen verwenden zu können.
- Funktionalität: Eine Sammlung von vielen Funktionen und Effekten für verschiedenste Abläufe wird von einer Physik-Engine bereitgestellt.

34 http://www.waltcollins.com, Internet: 30.06.2009

35 Object-Oriented Graphics Rendering Engine

36 Nähere Informationen: http://www.arcanoria.com/CS-Ogre.php, Internet: 30.06.2009

- Performance: Physik-Engines haben das Ziel realistische Abläufe effizient zu berechnen.

Nachteile

- Performance: Alleine schon die Tatsache, dass überhaupt eine Physik-Engine eingebunden wird, stellt eine Einbuße bezüglich der Performance dar, da die Engine mit Werten „gefüttert“ werden muss und eine gewisse Laufzeit braucht um verschiedenste Abläufe zu berechnen.
- Komplexität: Der Entwickler muss sich zuerst länger mit einer solchen Engine befassen um zu verstehen wie sie funktioniert und wie sie eingesetzt wird. Je nachdem welches Spiel entwickelt werden soll, kann dieses durchaus ohne Physik-Engine auskommen.

Soll es sich jedoch um ein realitätsnahes Spiel handeln, wie dies für das in dieser Arbeit zu realisierende Spiel der Fall ist, ist die Einbindung einer Physik-Engine unumgänglich. Im Internet existieren viele Physik-Engines. Für diese Arbeit erhalten jedoch nur einige nach [27] eine nähere Betrachtung:

Engine	**Lizenz**
Bullet[37]	Open Source
PhysX[38]	Kommerziell
Newton Game Dynamics[39]	Open Source
Open Dynamics Engine[40]	Open Source

Tabelle 4.3: Gegenüberstellung Physik-Engines

Die in Tabelle aufgeführten Kandidaten wurden für die Ogre-Engine portiert. Das heißt, es werden sogenannte „Wrapper“ von Community-Mitgliedern der Ogre3D-Engine entwickelt, welche als Schnittstellen zwischen Ogre und den externen Bibliotheken agieren.[41]

37 http://www.bulletphysics.com/wordpress, Internet: 02.07.2009

38 http://www.nvidia.com/object/physx_new.html, Internet: 02.07.2009

39 http://www.newtondynamics.com, Internet: 02.07.2009

40 http://www.ode.org, Internet: 02.07.2009

41 Vgl. http://de.wikipedia.org/wiki/Wrapper_(Software), Internet: 02.07.2009

Tabelle 4.4 zeigt nach [28] zu den Physik-Engines die dazugehörigen Wrapper:

Engine	**Wrapper**
Bullet	OgreBullet
PhysX	NxOgre
Newton Game Dynamics	OgreNewt
Open Dynamics Engine	OgreOde

Tabelle 4.4: Physik-Engine und Wrapper

Über Quelle [28] gelangt man zu den einzelnen Wrappern. Das Aufgabengebiet einer Physik-Engine erstreckt sich nach [27] auf folgende Arten von Simulationen:

- Berechnung starrer Körper (engl. rigid body mechanics)
- Verformung dynamischer Körper, wie beispielsweise Kleider, Haare, Wasser etc. (engl. soft body dynamics)
- Masse-Feder Modelle wie zum Beispiel das Verhalten von Seilen und Stoff zu Simulieren (engl. spring dynamics)
- Partikelsysteme, um beispielsweise das Verhalten von Flüssigkeiten oder Feuerwerkskörper zu berechnen (engl. fluid dynamics)

Die Tabelle 4.5 veranschaulicht, welche Features von welcher Engine unterstützt werden:

Feature	**Engine**			
	Bullet	PhysX[42]	Newton Game Dynamics	Open Dynamics Engine
Rigid body mechanics	•	•	•	•
Soft body dynamics	•	•	•	
Spring dynamics		•	•	•
Fluid dynamics		•		

Tabelle 4.5: Gegenüberstellung unterstützter Features

42 Der Nachweis über Features dieser Engine konnte nur auf der folgenden Seite gefunden werden: http://de.wikipedia.org/wiki/PhysX, Internet: 02.07.2009

Ein Problem besteht darin, dass eventuell nicht jeder Wrapper alle Features der dazugehörigen bis jetzt umgesetzt hat.
Für die Entwicklung des Spiels wurden alle Wrapper außer NxOgre, da diese sich noch in der Entwicklungsphase befindet und noch instabil ist, getestet. Der Autor kam dabei mit OgreNewt auf Anhieb am besten zurecht, deswegen kommt dieser Wrapper für die physikalischen Berechnungen zum Einsatz.

4.3 Audio

Über Audio wird ein Tonbereich definiert, welchen der Mensch hören kann.[43] In Spielen wird Audio für Hintergrundmusik und Geräuscheffekte eingesetzt, damit diese an Reiz gewinnen. OpenAL ist nach [30] eine freie plattformunabhängige 3D-Audio Bibliothek. Mittels dieser können eine Anzahl an Soundquellen erstellt werden, welche sich irgendwo im dreidimensionalen Raum befinden und von einem Zuhörer abhängig von seiner Position und Orientierung gehört werden können. Für Ogre sind zwei Wrapper verfügbar OgreAL[44] und OgreOggSound[45]. Diese haben OpenAL für Ogre portiert. OgreOggSound scheint derzeit nur über das Ogre Forum erreichbar zu sein und befindet sich noch in der Entwicklung, deswegen kommt für die Geräuscheffekte und Hintergrundmusik des Spiels OgreAL zum Einsatz. Zudem werden mit OgreAL einige Tutorials über die Funktionalität mitgeliefert.

4.4 3D-Grafik-Software

Mit diesen Werkzeugen können einzelne Objekte bis hin zu ganzen 3D-Szenen modelliert werden. Ein Modell, oder eine sogenannte Mesh zeichnet sich durch die Geometrie eines sichtbaren Objektes in der Anwendung aus. Diese setzt sich aus einer Anzahl an Dreiecken zusammen, welche miteinander verbunden sind.[46] Modelle können auch animiert werden, um somit Bewegungen im dreidimensionalen Raum simulieren zu können. Auch bei diesen Werkzeugen wird die Betrachtung nur auf die beschränkt, welche mit der gewählten Grafik-Engine vereinbar sind. Mit einer 3D-Grafik-Software erstellte Modelle müssen nach Quelle [10, S. 47, S. 247] in ein binäres Mesh-Format, welches für schnelles Laden des Inhaltes optimiert wurde, ex-

43 Vgl. de.wikipedia.org/wiki/Audio, Internet: 03.07.2009

44 http://www.ogre3d.org/wiki/index.php/OgreAL, Internet: 03.07.2009

45 http://www.ogre3d.org/addonforums/viewtopic.php?f=19&t=8588, Internet: 03.07.2009

46 Vgl. www.ogre3d.org/wiki/index.php/Mesh, Internet: 03.07.2009

portiert werden. Das selbe gilt für Animationen und Materialien[47]. OGRE-Community Mitglieder streben die Entwicklung von Programmen an, mit Hilfe derer die Exportierung ermöglicht wird. Im Folgenden werden einige 3D-Grafik-Software Programme zu denen nach Quelle [33] auch ein Exporter verfügbar ist in der Tabelle 4.6 gezeigt:

3D-Grafik-Software	
Autodesk Softimage[48]	Autodesk Maya[49]
Blender[50]	WINGS 3D[51]
LIGHTWAVE 3D[52]	CINEMA 4D[53]

Tabelle 4.6: 3D-Grafik-Software Programme mit Konverter

Für die Erstellung von Modellen, Animationen wird die Open Source 3D-Grafik-Software Blender genommen, da der Autor schon in einem längeren Zeitraum damit arbeitet und der Exporter alle notwendigen Funktionen unterstützt um die Modelle in Ogre zu benutzen.

4.5 Bildbearbeitungsprogramm

Es sind zahlreiche Bildbearbeitungsprogramme im Internet erhältlich. Ein solches Werkzeug ist notwendig um Texturen für Modelle und weitere 2D Grafiken aller Art bearbeiten zu können. Ein sehr bekanntes jedoch kommerzielles Programm ist Photoshop[54]. Der Konkurrent von Photoshop ist das freie Programm Gimp[55]. In dieser Arbeit wird Gimp benutzt, da der Umgang mit diesem Programms in der früheren Vorlesung Computervisualistik vertieft wurde.

Die gestellten Anforderungen können mit all den ausgewählten Werkzeugen realisiert werden.

47 Siehe Kapitel 5.5.4

48 http://www.softimage.com/products/xsi, Internet 03.07.2009

49 http://www.autodesk.de/adsk/servlet/index?siteID=403786&id=11473933, Internet: 03.07.2009

50 http://www.blender.org, Internet: 03.07.2009

51 http://www.wings3d.com, Internet: 03.07.2009

52 http://www.newtek.com/lightwave, Internet: 03.07.2009

53 http://www.maxon.net/de/products/cinema-4d.html, Internet: 03.07.2009

54 http://www.adobe.com/de/products/photoshop/photoshop/prophotographer/?sdid=EUGBY, Internet: 07.07.2009

55 http://www.gimp.org, Internet: 03.07.2009

5 Einführung in die Werkzeuge

Die Entwicklung eines eindrucksvollen Spiels in einer 3D-Welt ist sehr aufwendig, daher ist es von Vorteil einige Tools für diese Aufgabe zur Verfügung zu haben.
Dabei soll jedes Tool für ein Teilgebiet zuständig sein. Die Grafik-Engine sorgt für das Aussehen des Spiels und teilweise für den Ablauf. Über die Physik-Engine werden physikalische Abläufe des Spiels berechnet. Mittels einer Audioschnittstelle werden Geräuscheffekte und die Hintergrundmusik in das Spiel eingebunden und ausgeführt. Ein Leveleditor hilft die 3D-Welt zu erstellen. Mit Hilfe einer 3D-Grafik-Software können die 3D-Objekte modelliert und über ein Bildbearbeitungsprogramm die Texturen bemalt werden.
Im Folgenden erfolgt eine Einführung einiger der in Kapitel 4 ausgewählten Werkzeuge. Es wird grundlegendes Wissen über ihre Funktionalität vermittelt, sowie allgemeine wichtige Aspekte erläutert deren Verständnis notwendig ist, um darauf aufbauend in Kapitel 6 ein Spiel zu realisieren. Die Ogre- und die Physik-Engine werden als „Blackbox“ benutzt, das heißt es können nicht alle mathematischen und physikalischen Hintergründe erläutert werden. Es wird nur veranschaulicht, wie Berechnungen mit Hilfe dieser Engines funktionieren.

5.1 Ogre-Engine

Ogre ist nach Quelle [10, S. 1] eine plattformunabhängige Grafik-Engine zur Darstellung komplexer Echtzeitgrafiken durch Schnittstellen zur Grafikkarte wie Direct3D oder OpenGL. Die Grafik-Engine ist auf einem relativ neuen Stand, so dass der Entwickler auf viele Funktionen moderner Grafikhardware zurückgreifen kann. Sie ist umfangreich dokumentiert und das Erlernen wird durch Tutorials und eine große Community erleichtert. Ogre ist unter der LGPL[56] Lizenz verfügbar.

5.2 Funktionsweise

Nachfolgend wird ein Einblick in die Funktionsweise der Ogre-Engine nach [10, S. 77-90] gegeben.
Ogre verwaltet die Organisation aller Objekte, welche möglicherweise auf dem Bildschirm dargestellt werden über einen „SceneManager“ (Szenenmanager), welcher die Implementierung eines Szenengraphes umfasst. Ein Szenengraph ist

56 http://www.gnu.de/documents/lgpl.de, http://www.gnu.org/licenses/lgpl-3.0.html, Internet: 07.01.2009

eine Datenstruktur für die Objektorganisation. Allgemein gehalten enthält ein Szenenmanager alle Informationen um sämtliche Objekte effizient anzuordnen und somit eine 3D-Szene aufzubauen und zu rendern.
Des Weiteren werden über den Szenenmanager sogenannte „SceneNodes“ (Szenenknoten) verwaltet, welche die Struktur des Szenengraphen widerspiegeln. Sie sind hierarchisch in einem Baum angeordnet. Wenn der Szenenmanager erstellt wird, enthält er zumindest den Wurzelknoten. An diesen können ein oder mehrere Szenenknoten angebracht werden. Soweit es sich nicht um den Wurzelknoten handelt, hat jeder Knoten einen Vaterknoten. Abbildung 5.1 aus [10, S. 80] stellt die Baumstruktur dar.

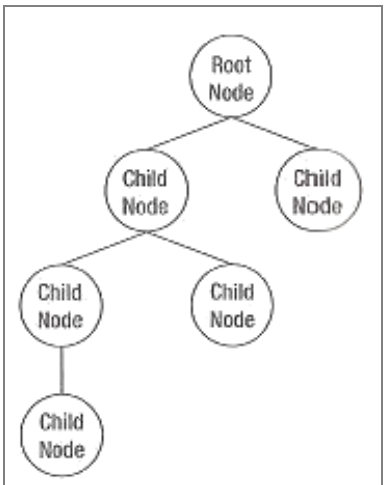

Abbildung 5.1: Struktur des Szenengraphes

Es können nach Bedarf Szenenknoten an andere angefügt und vorhandene entfernt werden. Entfernte Knoten sind noch solange verfügbar bis sie gelöscht werden.
Ein Knoten trägt die folgenden Informationen:

- Die Position, beschrieben durch einen Vektor im dreidimensionalen Raum.
- Den Skalierfaktor, ebenfalls definiert durch einen Vektor.
- Die Orientierung, beschrieben über ein Quaternion[57]

Er repräsentiert also noch nichts Sichtbares. Um jedoch etwas Grafisches auf dem Bildschirm darzustellen, können eine Reihe von Objekten an einen Szenenknoten gehängt werden, einige von ihnen werden im Unterabschnitt 5.6 beschrieben.
Die Ogre-Engine benutzt für alle Berechnungen im dreidimensionale Raum ein

57 Siehe Kapitel 5.10.3

kartesisches[58] Koordinatensystem mit der Y-Achse nach oben gerichtet. Viele 3D-Grafik-Programme besitzen ein Koordinatensystem mit der Z-Achse nach oben. Das heißt Modelle, welche zum Beispiel mit Blender modelliert wurden, müssen um 90° um die X-Achse gedreht werden.
Die positive Z-Achse in Ogre ist auf den Betrachter gerichtet und die positive X-Achse zeigt vom Betrachter aus gesehen nach rechts. Die Erkenntnis, welche Achse in welche Richtung zeigt ist wichtig, da sämtliche Berechnungen davon abhängig sind. Ein Objekt muss sich standardmäßig in irgendeine Richtung orientieren und die Position wird ebenfalls davon abhängig berechnet. Berechnungen wie beispielsweise das Translieren (Verschieben) oder Rotieren von Objekten verlaufen immer in Bezug zu etwas. Dies wird anhand von drei Merkmalen erläutert:

- **„Welt-Raum“** (engl. world space)
  Die Engine besitzt zumindest den Wurzelknoten, dieser repräsentiert den Weltkoordinatenursprung. Ein Objekt besitzt auch einen Koordinatenursprung. In Blender beispielsweise kann der Ursprungspunkt eines 3D-Modells festgelegt werden, standardmäßig befindet sich dieser in der Mitte des Objektes. Wenn dieses an den Wurzelknoten gehängt wird, so befindet es sich genau im Zentrum der Welt. Abbildung 5.2 nach Quelle [10, S. 83] veranschaulicht einen Würfel im Zentrum des Koordinatensystems. In dieser Abbildung ist auch klar ersichtlich, in welche Richtung jede positive Achse jeweils gerichtet ist.

58 http://www.mathepedia.de/Kartesische_Koordinatensysteme.aspx, Internet: 11.07.2009

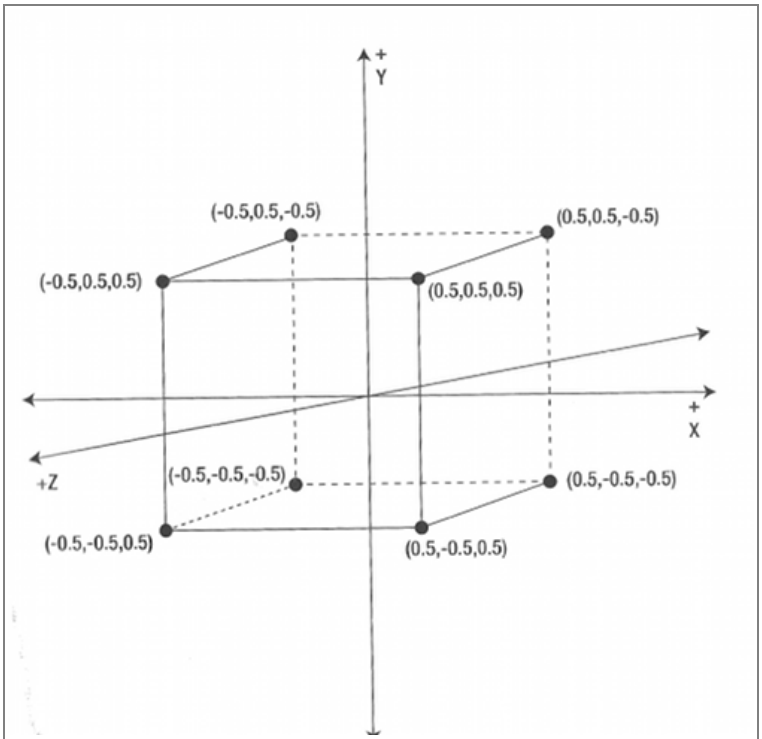

Abbildung 5.2: Würfel in der Mitte des Weltkoordinatensystems

- **„Lokaler Raum“** (engl. local space)
  Der Koordinatenursprung ist wie oben beschrieben der Ursprungspunkt des Objektes selbst. Wenn das Objekt zum Beispiel gedreht[59] wird, so geschieht dies um die eigene Achse. Die Skalierung oder auch Vergrößerung des Objektes verläuft nach den eigenen Werten. Um ein Objekt auf die doppelte Größe zu bringen, wird dies beispielsweise durch den Skaliervektor (2,2,2) erreicht. Die Translation eines Objektes geschieht allerdings im Bezug zum Weltkoordinatensystem, da das Objekt am Wurzelknoten hängt.

- **„Vorgänger-Raum“** (engl. parent space)
  Wenn ein Objekt an einen Szenenknoten angefügt, dieser wiederum an einen Weiteren gehängt wird, so befindet sich der Koordinatenursprung nicht mehr im Zentrum der Welt, sondern dort, wo sich der Vorgängerknoten befindet. Der Vaterknoten markiert also den Ursprungspunkt. Dies ist eine Besonderheit. Wenn ein Objekt an einen Knoten und dieser an den Vorgängerknoten angebracht wird, und nun der Kindknoten um zwei Einheiten entlang der X-Achse platziert wird, entsteht ein Abstand zum Ursprungspunkt des Vorgän-

59 Drehungen von Objekten werden in Kapitel 5.10.3 näher erläutert.

gerknotens. Wird anschließend der Vaterknoten um zwei Einheiten entlang der Y-Achse transliert, so wird das Objekt mit dem zuvor festgelegten Abstand ebenfalls zwei Einheiten nach oben bewegt. Der Vaterknoten besitzt nun die Koordinaten (0, 2, 0) und der Kindknoten (2, 2, 0). Bei einer Drehung verhält es sich ähnlich, statt um die eigene Achse zu drehen wird um die Achse des Vorgängerknotens mit dem zuvor definierten Abstand gedreht. Abbildung 5.3 aus [10, S. 87] zeigt die Koordinaten eines Kindknotens im Bezug zum Vaterknoten:

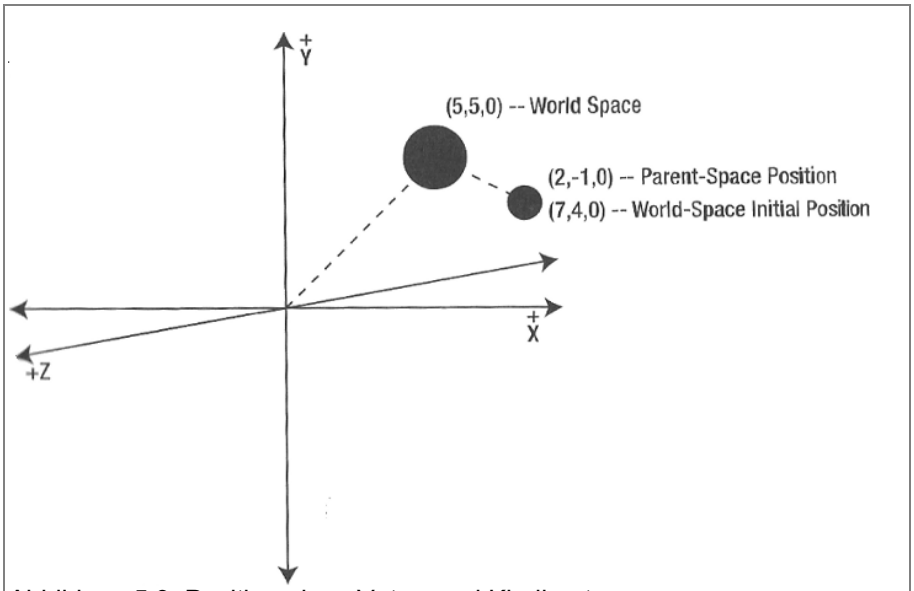

Abbildung 5.3: Position eines Vater- und Kindknotens

5.3 Szenen Abfragen

Im vorherigen Abschnitt wurde auf die Baumstruktur, in welcher die Szenenknoten hierarchisch angeordnet sind, eingegangen. Dieser Baum kann nach bestimmten Kriterien effizient durchsucht werden. Der Szenenmanager kann zum Erstellen von Abfragen bezüglich der Szene genutzt werden, um Informationen über einzelne Knoten und Objekte zu erhalten. Alle Objekte in Ogre besitzen einen „Grenzquader“ (engl. bounding box), der wie das Weltkoordinatensystem orientiert ist und ein Objekt umschließt, so dass es völlig hineinpasst. Abbildung 5.4 zeigt ein Modell mit einer bounding box.

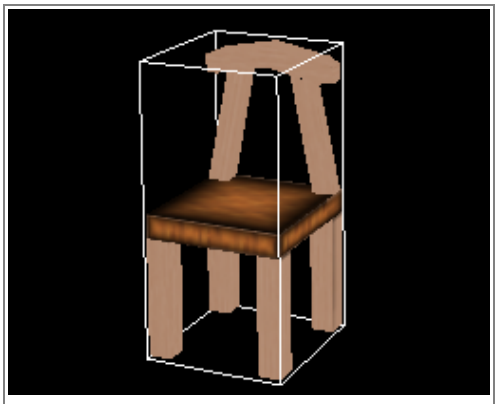

Abbildung 5.4: Screenshot - Objekt mit einem Quader

Im Allgemeinen kann getestet werden, ob etwas mit diesem Quader in Berührung gekommen ist.
Weiterhin werden einige mögliche Arten von Abfragen nach Quelle [10, S. 81, S. 109] vorgestellt und erläutert:

- Abfragen durch einen Strahl (engl. ray queries)
  Als Ray wird eine Linie bezeichnet, die von einem bestimmten Punkt im dreidimensionalen Raum entlang einer definierten Richtung ins Unendliche läuft. Dabei wird getestet ob die Linie mit dem beschriebenen Quader eines Objektes in Berührung kommt. Daraus können zum Beispiel Informationen über den Objekttyp, die Distanz zu einem Objekt etc., erhalten werden.

- Abfragen durch eine Sphäre (engl. sphere queries)
  Eine Sphäre bezeichnet eine dreidimensionale Kugel, gebildet durch einen Vektor, der den Mittelpunkt darstellt und einen Radius. Es besteht die Möglichkeit nachzuprüfen, ob sich ein oder mehrere Objekte innerhalb der Sphäre befinden, aus diesen Ergebnissen können auch Informationen gezogen werden.

- Abfragen durch einen Quader
  Ein Quader entsteht aus zwei entgegengesetzten Vektoren. Es kann herausgefunden werden, ob sich Objekte in diesem Bereich befinden.

Des Weiteren kann eine Abfrage nach benutzerdefinierten Kriterien gefiltert werden, das heißt es können Objekte, die ein bestimmtes vorher festgelegtes Kriterium nicht erfüllen von der Suche ausgeschlossen werden. Wenn beispielsweise in einem Spiel eine Abfrage nur über Gegner stattfinden soll, so erhält zuvor jeder Gegner den entsprechenden Typ (auch Maske genannt) und die Abfrage wird auf diese Maske beschränkt.

5.4 Materialien und Shader

Ogre besitzt nach [10, S. 113-121] eine eigene Material-Deklarationssprache, in welcher beschrieben werden kann, auf welche Art ein Objekt ein Licht reflektiert. Dies erfolgt über Shader (dt. Schattierer). Hierbei handelt es sich um Programme, welche für die Berechnung von komplexen grafischen Lichteffekten eingesetzt werden. Sie werden über eine Shader-Deklarationssprache beschrieben und über den Grafikprozessor (GPU) in einer Grafikpipeline ausgeführt, sodass sie dem zentralen Prozessor (CPU) Rechenarbeit abnehmen. Ogre besitzt verschiedene vorgefertigte Shader, die eingesetzt werden können. Es besteht aber auch die Möglichkeit sie selbst zu implementieren, jedoch ist dies ein riesiges und vor allem komplexes Gebiet, sodass in dieser Arbeit nur vorgefertigte Shader benutzt werden.
Über eine Materialdatei kann also das Aussehen eines Objektes bestimmt werden. Abbildung 5.5 zeigt die Struktur einer Materialdatei anhand eines Beispiels:

```
material SOLID/Glowing/TEX/eyes.png
{
        technique
        {
                pass
                {
                    emissive 1.000000 1.000000 1.000000 1.000000
                    texture_unit
                    {
                         texture eyes.png
                    }
                }
        }
}
```

Abbildung 5.5: Screenshot - Materialdatei

Die Materialdatei besteht aus folgenden Elementen:

- **Material**
  Dies ist eine Identifizierung, dass es sich um eine Materialdatei handelt. Der nachfolgende Bezeichner „SOLID/Glowing/TEX/eyes.png“ ist der Name des Materials, über diesen kann einem Objekt das Material zugeteilt werden.

- **Technique (dt. Technik)**
  Über eine Technik kann festgelegt werden, auf welche Art ein Objekt gerendert wird. Eine Materialdatei kann aus mehreren Techniken bestehen. Im Unterabschnitt 5.13 wird erläutert, wie bestimmt werden kann, welche Technik wann zum Einsatz kommt.

- **Pass**
  In einem Pass kann eine Zeichenoperation beschrieben werden, welche über die GPU ausgeführt wird.

- **Texture Unit** (dt. Textureinheit)
  In einer Textureinheit wird auf eine Textur mittels eines Namens verwiesen.

Wird das Material im zuvor beschriebenen Beispiel bei Abbildung 5.5 einem 3D-Augen-Modell zugewiesen, so erhält das Modell die „eyes.png“-Textur und über den Befehl „emissive 1.000000 1000000 1000000 1000000“ leuchten die Augen weiß.

5.5 Initialisierung

In diesem Abschnitt wird erklärt, welche Schritte unternommen werden müssen um Ogre verwenden zu können.
Zunächst muss das Root-Objekt erzeugt werden, dieses ist für die Verwaltung von Ogre zuständig.

```
Ogre::Root *pRoot = new Ogre::Root();
```

Danach wird über das Root-Objekt der Szenenmanager angelegt:

```
Ogre::SceneManager *pSceneManager =
pRoot->createSceneManager("OctreeSceneManager");
```

Als nächstes müssen die Ressourcen (Materialien, Modelle etc.) geladen werden, dazu wird der Dateiname angegeben „resources.cfg“ und dessen Inhalt in einer Datenstruktur gespeichert. Später muss nur noch der Name der Ressource angegeben werden und es ist für den Entwickler transparent, wo sich diese befindet.

```
Ogre::ConfigFile cf;
cf.load("resources.cfg");
Ogre::ConfigFile::SectionIterator seci = cf.getSectionIterator();
Ogre::String secName, typeName, archName;
while (seci.hasMoreElements())
{
     secName = seci.peekNextKey();
     Ogre::ConfigFile::SettingsMultiMap *settings = seci.getNext();
     Ogre::ConfigFile::SettingsMultiMap::iterator i;
     for (i = settings->begin(); i != settings->end(); ++i)
     {
            typeName = i->first;
            archName = i->second;
            Ogre::ResourceGroupManager::getSingleton()
            .addResourceLocation(archName, typeName, secName);
     }
}
```

Anschließend müssen diese initialisiert werden.

```
Ogre::ResourceGroupManager::getSingleton().initialiseAllResourceGroups();
```

Abbildung 5.5 zeigt die Struktur einer Ressourcen-Datei:

```
[Bootstrap]
Zip=../Media/packs/OgreCore.zip

[General]
FileSystem=../Media
FileSystem=../Media/models
[Flubbers]
FileSystem=../Media/flubbers
[Caleum]
FileSystem=../Media/caelum
[ET]
FileSystem=../Media/ET
[World]
FileSystem=../Media/world1.3
```

Abbildung 5.6: Screenshot - Struktur einer Ressourcen-Datei

Im Folgenden wird ein Renderfenster benötigt um die Szene darstellen zu können. Das Renderfenster ist die Oberfläche, auf welcher die Szene gezeichnet wird.

```
Ogre::RenderWindow *pRenderWindow = pRoot->initialise(true, "Fenstername");
```

Damit die Grafik-Engine Bilder der Szene anzeigen kann, muss eine Kamera[60] erstellt werden:

```
Ogre::Camera *pCamera;
pCamera = pSceneManager->createCamera("KameraName");
pCamera->setPosition(Ogre::Vector3(0, 50, 0));
```

Zusätzlich wird ein Sichtbarkeitsbereich benötigt, welcher den Renderbereich im Renderfenster festlegt:

```
Ogre::Viewport *pViewPort = pRenderWindow->addViewport(pCamera);
```

Danach ist die Einbindung der Grafik-Engine abgeschlossen. Nun können Objekte in die Szene eingefügt werden. Diese werden im nächsten Abschnitt beschrieben.

5.6 Objektarten

In Ogre existieren verschiedene Objektarten, diese sind von der Basisklasse MovableObject abgeleitet. Um einen grafischen Effekt zu erzielen muss ein Objekt an einen Szenenknoten angebracht werden. Das Objekt kann nicht gleichzeitig an mehreren Knoten hängen, aber es ist möglich so viele Objekte wie gewünscht an einen Szenenknoten anzubringen.
Im weiteren Verlauf wird auf die wesentlichen Objektarten eingegangen. In Ogre existiert eine Vielfalt an Objektarten, für diese Arbeit ist jedoch nur ein Teil relevant.

5.6.1 Entities

Über ein Entity wird ein 3D-Modell repräsentiert. Das bedeutet, nachdem ein 3D-Objekt in einer 3D-Grafik-Software modelliert und in das Ogre-konforme Format exportiert wurde, kann dieses über den Mesh-Namen geladen werden. Standardmäßig beinhaltet eine Meshdatei den Namen des zu verwendeten Materials, alternativ kann über einen Befehl eine andere selbst definierte Materialdatei dem Modell zugeordnet werden. Des Weiteren kann ein Mesh aus mehreren Teilmodellen bestehen, welche jeweils auf separate Materialdateien verweisen. Diese Erkenntnis wird sich beim Erstellen der Gegner in Kapitel 6.7.2 zu Nutze gemacht.
Um ein Modell in einer Szene angezeigt zu bekommen muss zunächst ein Entity mit eindeutigem Namen und Dateinamen des Modells über den Szenemanager erzeugt

60 Die Funktionsweise einer Kamera wird im Kapitel 5.6.3 erklärt.

werden und an einen Knoten gehängt werden.

```
Ogre::Entity *pEntity = pSceneManager->createEntity("EntitätName","Case1.mesh");

Ogre::SceneNode *pSceneNode;
pSceneNode =
pSceneManager->getRootSceneNode()->createChildSceneNode("KnotenName");

pSceneNode->setPosition(Ogre::Vector3(0,0,0));
pSceneNode->attachObject(pEntity);
```

In diesem Beispiel wurde eine Kiste erzeugt.

Abbildung 5.7: Screenshot: 3D-Model in Ogre dargestellt

Eine Mesh-Datei kann auch auf eine sogenannte „Skeleton“-Datei verweisen, in dieser sind Animationen des Modells gespeichert. Über den Namen einer Animation kann diese zu einem Entity geladen und mit dem Befehl „setEnabled(true)“ abgespielt werden.

```
Ogre::AnimationState *pAnimState;
pAnimState = pEntity->getAnimationState("Animationsname");
pAnimState->setEnabled(true);
```

Die genaue Funktionsweise wird in Kapitel 6.7.1 anhand der Bewegung des Spielers erläutert.

5.6.2 Partikelsysteme

Partikelsysteme werden eingesetzt um visuelle Effekte wie beispielsweise Rauch, Feuer usw. zu simulieren. In Ogre können Partikelsysteme in einem externen Skript oder im Quellcode beschrieben werden. Es existieren in der Grafik-Engine eine Reihe an Skripts in welchen verschiedene Partikeleffekte implementiert sind und benutzt werden können. Ein Partikeleffekt kann über den Szenenmanager erzeugt werden, dazu muss ein eindeutiger Name und die Bezeichnung des Partikeleffektes aus ei-

nem Skript angegeben werden. Nach Erzeugung wird ein Effekt erst sichtbar, wenn ein Partikelsystem an einen Szenenknoten gehängt wird.

```
Ogre::ParticleSystem *pParticleSystem;
pParticleSystem = pSceneManager->createParticleSystem("Name", "Examples/Smoke");
pSceneNode->attachObject(pParticleSystem);
```

5.6.3 Kameras

Diese Objekte sind notwendig um die Szene aus einem bestimmten Blickwinkel festzuhalten. Das heißt, was der Betrachter von einer 3D-Szene sieht, ist abhängig von der Position und Orientierung der Kamera. Es können mehrere Kameras in Ogre existieren, die Szene eines Renderfensters kann jedoch immer nur von einer aus angezeigt werden. Eine Besonderheit von Kameras ist, dass diese nicht unbedingt an einen Szeneknoten angebracht werden müssen, sie können selbständig platziert und rotiert werden. Wird eine Kamera jedoch an einen Szenenknoten gehängt, so kann sie über diesen rotiert und transliert werden. Wie eine Kamera erstellt wird, kann Unterabschnitt 5.5 entnommen werden. Je nach Spielgenre werden verschiedene Kameraperspektiven unterschieden. Im Folgenden werden nach [8, S. 242] die zwei für Spiele relevanten Arten von Kameraperspektiven kurz erläutert:

- **Ich-Perspektive** (engl. First-Person)
  Bei der Ich-Perspektive[61] ist die Position des Spielers gleichzeitig auch die Kameraposition, das heißt, der Betrachter sieht die 3D-Welt aus den Augen des Spielers.

- **Dritte-Person-Perspektive** (engl. Third-Person)
  Der Betrachter schaut dem Spieler bei dieser Perspektive über die Schultern. Zumeist folgt die Kamera dem Spieler mit einem gewissen Abstand. Es existieren viele Variationen dieser Ansichtsart. Bei einigen Rollenspielen beispielsweise ist die Welt von Oben zu sehen und der Spieler befindet sich im Zentrum des Kamerabereichs.

61 Diese wird auch oft als Ego-Perspektive bezeichnet.

5.7 Landschaft und 3D-Welt

In Ogre existiert ein flacher Boden, welchem eine Textur zugeordnet werden kann. Dieser ist jedoch unzureichend für das zu realisierende Spiel. Eine flache Oberfläche bietet kaum Spielanreiz und würde unrealistisch wirken. Damit eine realitätsnahe Welt erzeugt werden kann, muss eine Landschaft her, mit welcher sich Berge, Täler und Schluchten formen lassen und auf welche mit verschiedenen Texturen gezeichnet werden kann. ETM[62] ist eine solche freie Bibliothek zur Darstellung und Manipulation einer Landschaft, diese wurde von einem Ogre Community-Mitglied entwickelt und kann eingebunden werden. Vom Autor und einem Kommilitonen wurde in früheren Projekten ein Leveleditor entwickelt, in welchen auch ETM integriert wurde. Dieser nennt sich "Virtual World Editor" und ist ein Werkzeug, welches das Erzeugen von 3D-Welten erleichtert. Er besteht aus einem Assistenten zur Festlegung der Umgebung. Aus diesem können die folgenden Komponenten gewählt werden:

- die Art der Oberfläche, ob flach oder zum Formen einer Landschaft
- der Himmel, ob beweglich oder fest
- der Nebel zur Beeinträchtigung der Sichtweite
- und ein Umgebungslicht zur Einstellung der Helligkeit

Nach dem Abschließen des Assistenten wird anhand der Einstellungen die Umgebung in der 3D-Szene generiert. Darauf folgend kann der Benutzer Berge formen und auf der Landschaft mit verschiedenen Texturen malen. Eine weitere wichtige Funktionsart ist das Einfügen von 3D-Objekten in die Welt. Diese können dort auf verschiedenste Weise editiert werden, wie beispielsweise gedreht, skaliert, verschoben etc. Sämtliche Editieraktionen sind durch die Undo-Funktionalität begleitet, sodass jeder Schritt rückgängig gemacht und wiederholt werden kann.
Die Szene kann abgespeichert und zur weiteren Bearbeitung geladen werden.
Der Editor ist durch Plugins[63] erweiterbar, sodass in Zukunft Entwickler eigene Teilwerkzeuge zur Erweiterung der Umgebung erstellen können, wie zum Beispiel die Darstellung von Wasser. Alle Aktionen sind hierarchisch in Werkzeugleisten angeordnet und können über diese bedient werden. Abbildung 5.8 zeigt einen Screenshot des Editors (Quellen: [13], [14], [15]).

62 http://www.ogre3d.org/wiki/index.php/ETM, http://www.oddbeat.de/wiki/etm, Internet: 06.07.2009

63 Ein Plugin ist ein Programm, um welches eine Software zur Laufzeit erweitert werden kann.

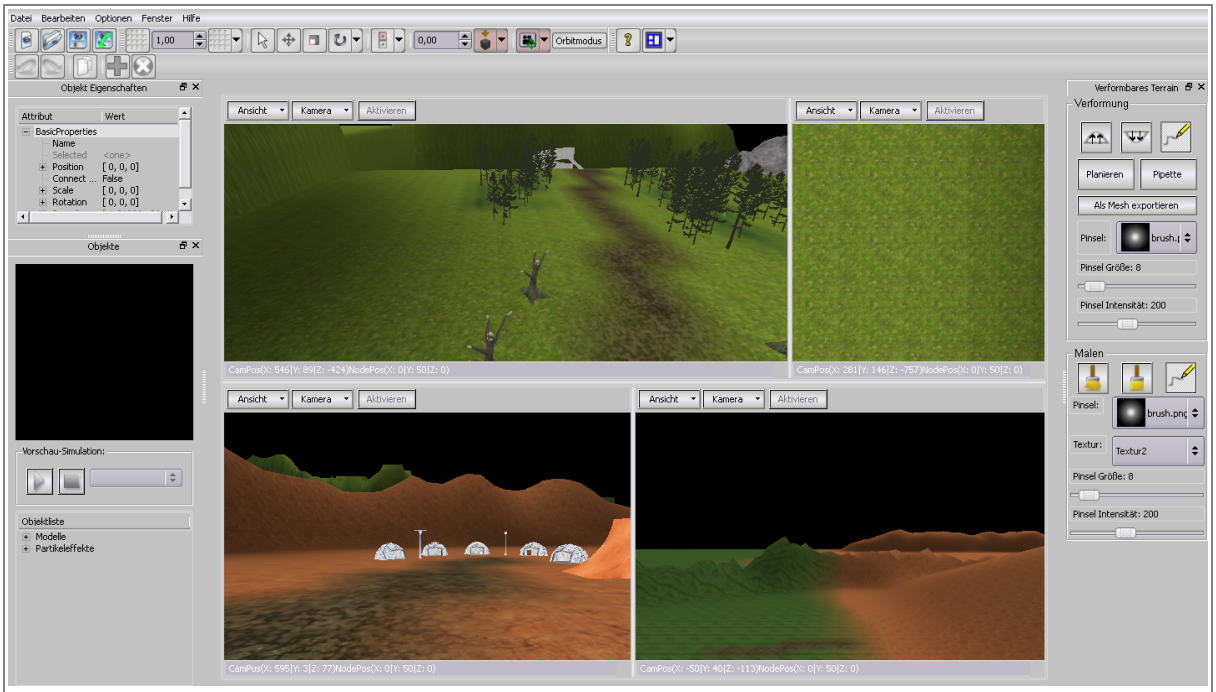

Abbildung 5.8: Screenshot - VWE

5.8 Himmel

Zu einer realistischen Welt gehört auch eine Darstellung eines geeigneten Himmels. In der Grafik-Engine sind verschiedene Himmelsarten verfügbar. Ist jedoch eine realistische Berechnung und Darstellung eines Himmels mit atmosphärischen Effekten gewünscht, so bietet sich die freie Bibliothek Caelum[64] an. Diese wurde von einigen Ogre-Community Mitgliedern entwickelt. Caelum beinhaltet die Berechnung und Darstellung von Sonnenauf- und Untergängen, Mondphasen, einer Sternenkarte und vielem mehr. Welche Schritte unternommen werden müssen, um diese Bibliothek einzubinden kann Quelle [37] entnommen werden.

5.9 Größenordnung

Die Größenordnung ist von enormer Bedeutung. Es ist wichtig, sich vor der Realisierung eines Spiels Gedanken über die Größe einer Welteinheit[65] zu machen und sich für eine Größenordnung zu entscheiden, da alle Objekte welche modelliert werden sich an dieser orientieren müssen. Zudem hängt die Berechnung der Physik in Ogre-

64 www.ogre3d.org/wiki/index.php/Caelum, Internet: 06.07.2009

65 Welteinheit (engl. world unit, Abk. wu)

Newt davon ab. In den Demoprogrammen von Ogre ist ein Meter = 100 wu (1 wu = 1 Zentimeter)[66], OgreNewt behandelt allerdings einen Meter als eine Welteinheit, sowie eine Einheit als einen Kilogramm. Der Entwickler von OgreNewt welcher den Nicknamen „Walaber“ trägt, erwähnt dies im Ogre-Forum.[67] Das heißt alle physikalischen Berechnungen in OgreNewt basieren auf dieser Größenordnung, daher orientiert sich das Spiel auch an der Größenordnung von OgreNewt. Die Welt soll 4 Kilometer² betragen, dies entspricht einer Fläche von 4 Millionen Meter² beziehungsweise Welt Einheiten.

5.10 Physik und OgreNewt

In Kapitel 4.2 wurde der Wrapper OgreNewt ausgewählt, welcher die Portierung der "Newton Game Dynamics" Physik-Engine vollzieht. Im weiteren Verlauf erfolgt eine Einführung in OgreNewt. Außerdem wird ein Verständnis für die wesentlichen mathematischen sowie physikalischen Berechnungen, die für den Ablauf des Spiels notwendig sind, vermittelt.
Zunächst erfolgt die Initialisierung von OgreNewt. Die Grenzen einer Welt, in welcher physikalische Berechnungen stattfinden sollen, werden über zwei Vektoren angegeben.

```
OgreNewt::World* pOgreNewt = new OgreNewt::World();

pOgreNewt->setWorldSize(Ogre::Vector3(-1000,-500,-1000),
                        Ogre::Vector3(1000,500,1000));
```

Der Einfluss der Physik in der virtuellen 3D-Welt wäre in diesem Fall 2000 Welteinheiten beziehungsweise Meter (-1000 bis 1000) entlang der X-Achse, 2000 Meter entlang der Z-Achse und 1000 Meter entlang der Y-Achse vorhanden. Das bedeutet fiktiv, dass die Weltgröße als ein dreidimensionaler großer Quader vorstellbar ist, innerhalb welcher die Physik-Engine arbeitet. Würde ein Objekt außerhalb dieser Grenzen geschleudert, so würde es gestoppt werden, da keine Kraft außerhalb dieses Bereiches herrscht.

66 http://www.ogre3d.org/wiki/index.php/HowTo_(part_II)#How_to_manage_world-units, Internet: 07.07.2009

67 http://www.ogre3d.org/addonforums/viewtopic.php?f=4&t=5742&p=33474&hilit=unit#p33474, Internet: 07.07.2009

5.10.1 Physikalische Körper

Im Folgenden wird anhand der Quellen [38] und [39] erläutert wie die Einbindung von physikalischen starren Körpern (engl. rigid body) für Ogre-Objekte abläuft, welche Kollisionsformen unterstützt werden und wie Kollisionen zwischen zwei Körpern stattfinden.

Die Physik-Engine verwaltet zwei Arten von Körpern, die statischen und dynamischen. Statische oder „feste“ Körper erhalten zum Beginn der Anwendung ihre Position und Orientierung, welche sich nie mehr ändert. Diese Körper erhalten von der Physik-Engine eine unendliche Masse, sodass wenn eine Kollision stattfindet, sie unter keinen Umständen bewegt werden. Damit überhaupt Kollisionen zwischen Objekten stattfinden können müssen diese eine sogenannte „Hülle“ erhalten. OgreNewt beherbergt verschiedene Kollisionsformen. Für statische Objekte wie beispielsweise die Landschaft, bietet sich die Baumkollision (engl. Treecollision) an. Bei dieser Kollisionsart werden alle Polygone eines 3D-Modells durchlaufen und daraus die Hülle geformt. Diese Hülle kann für sehr komplexe Objekte eingesetzt werden, durch Benutzung dieser erhalten Körper automatisch von der Physik-Engine eine unendliche Masse und werden somit statisch. Dynamische oder „bewegliche“ Körper wie beispielsweise eine Kiste hingegen können keine solche komplexe Kollisionsart erhalten, da diese physikalisch simuliert werden müssen. Die Physik-Engine unterstützt viele Kollisionsformen zu diesen Körpern. Nun werden einige kurz erläutert:

- **Sphäre**
  Eine Sphäre wird über einen Vektor angegeben, da sie je nach Angabe zu Ellipsen geformt werden kann.

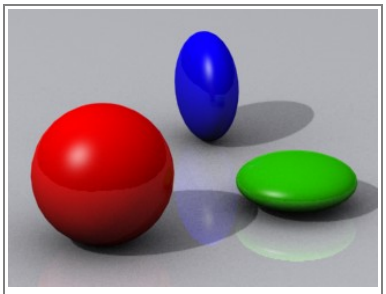

Abbildung 5.9: Verschiedene sphärische Formen

- **Quader** (engl. Box)
  Dies ist die einfachste Kollisionsart, sie wird über einen Vektor beschrieben.
- **Kapsel** (engl. Capsule)
  Eine Kapsel wird über einen Radius und eine Höhe definiert.
- **Zylinder**
  Dieser wird wie bei der Kapsel über einen Radius und eine Höhe beschrieben.
- **Kegel**
  Ein Kegel wird genauso erstellt wie ein Zylinder.
- **Abgerundeter Zylinder** (engl. Chamfer Cylinder)
  Dies ist ein Zylinder, bei welchen die Kanten abgerundet sind. Diese Form kann man sich vorstellen wie einen Donut ohne Loch.

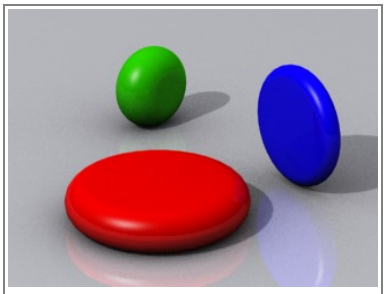

Abbildung 5.10: Verschiedene abgerundete Zylinder

- **Konvexe Hülle** (Convex Hull)
  Eine konvexe Hülle ist eine geometrische Figur, die vorliegt, wenn zu zwei Punkten einer Menge auch deren Verbindungsstrecke zu dieser Menge gehört.[68] Diese Kollisionsform benötigt keine Eingabeparameter.

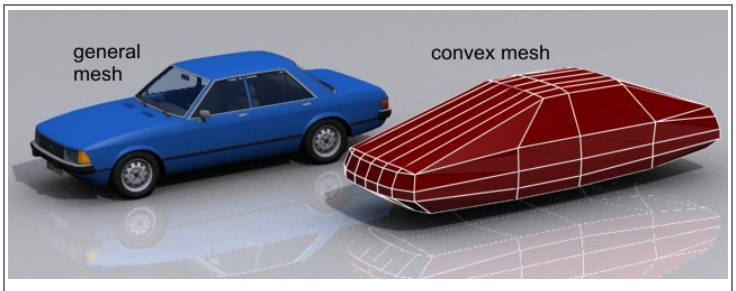

Abbildung 5.11: Modell als Mesh und konvexe Hülle

68 Vgl. de.wikipedia.org/wiki/Konvexe_Menge, Internet: 07.07.2009

In Kapitel 5.6.1 wurde ein Entity bereits erstellt, über welches eine Kiste dargestellt wird. Nun wird anhand dieses Kistenmodells beschrieben, wie sichtbare Ogre-Objekte Kollisionskörper erhalten.
Zunächst wird von diesem der Skaliervektor und die Größe des Grenzquaders[69] benötigt.

```
Ogre::Vector3 scale = pSceneNode->getScale();
Ogre::Vector3 size = pEntity->getBoundingBox().getSize() * scale;
```

Die Größe der Box beträgt einen Meter in jede Richtung. Anhand dieser Werte wird nun eine Box-Kollisionsform erstellt und einem Körper zugeteilt:

```
OgreNewt::Collision* pCol = new OgreNewt::CollisionPrimitives::Box(OgreNewt,
                  size, Ogre::Quaternion::IDENTITY, Ogre::Vector3(0.0,0.5,0.0));
OgreNewt::Body* pBody = new OgreNewt::Body(OgreNewt, pCol);
delete col;
```

In Kapitel 5.2 wurde darauf eingegangen, dass bei einem modellierten Objekt der Ursprungspunkt eventuell nicht im Zentrum liegt, dies ist im Kistenmodell der Fall. Von daher muss die Kollisionshülle 0,5 Meter nach oben verschoben werden. Der Ausdruck: „Ogre::Quaternion::IDENTITY“ bedeutet, dass die Hülle nicht gedreht wird. Danach wird die Masse für das Objekt aus diesen Größen berechnet. Mittels der Masse kalkuliert OgreNewt wie viel Kraft notwendig ist, um dieses zu bewegen.

```
Ogre::Real mass = scale.x * scale.y * scale.z * 20;
```

Die Kiste wurde nicht skaliert, also beträgt die Masse 20 Kilogramm[70]. Im nächsten Schritt wird mittels der errechneten Werte die Trägheit[71] (engl. Inertia) eines Körpers ermittelt. OgreNewt enthält die Berechnungsvorschriften für die Trägheit sämtlicher Kollisionsformen.

```
Ogre::Vector3 inertia = OgreNewt::MomentOfInertia::CalcBoxSolid(mass, size);
pBody->setMassMatrix(mass, inertia);
```

69 Siehe Unterabschnitt 5.3

70 Bei der Erdanziehungskraft von ~ 9,8 Meter pro Sekunde² hätte dieses Objekt eine Masse von 196 Newton.

71 Nähere Informationen: http://www.quantenwelt.de/klassisch/eigenschaften/masse.html, Internet: 07.07.2009

Da das Kistenmodell seinen Ursprungspunkt nicht in der Mitte hat, muss das Zentrum der Masse angepasst werden, ansonsten hätte der Körper ein Ungleichgewicht und würde sich bei einer Bewegung physikalisch unkorrekt verhalten.

```
pBody->setCenterOfMass(Ogre::Vector3(0.0, size.y / 2.0, 0.0));
```

Im nächsten Schritt wird eine vorgefertigte Routine aufgerufen, welche die Gravitationseinwirkung für diesen Körper berechnet. OgreNewt stellt dazu die Funktion setStandardForceCallback() bereit. In dieser Funktion beträgt die Gravitation standardmäßig 9,8 Meter pro Sekunde², dies entspricht in etwa der Erdanziehungskraft. Es stellte jedoch heraus, dass die Anziehung für das Spiel zu schwach ist. Der Spieler, die Gegner und alle Objekte fielen zu langsam zu Boden, nachdem diese in die Luft geschleudert wurden. In Quelle [23, S. 50] wird diese Tatsache bestätigt, dass in Spielen eine Gravitation von 9,8 Meter pro Sekunde² bei der Simulation nicht sehr überzeugend aussieht. Nach mehreren vom Autor durchgeführten Tests erwies sich eine Gravitation von 16,8 Meter pro Sekunde² als geeignet. OgreNewt hat die genannte Schwierigkeit bedacht und bietet die Möglichkeit an, eine eigene Funktion zur Gravitationseinwirkung zu schreiben und diese einzubinden. Dieses Prinzip wird Rückruffunktion[72] genannt. Dabei enthält eine Funktion als Eingabeparameter statt einer Variablen einen Zeiger auf die auszuführende Funktion. Im Folgenden wird eine solche veranschaulicht. Die Klasse, in welcher das Kisten-Modell erstellt wird, erhält die Funktion:

```
void Class::moveCallback(OgreNewt::Body* me)
{
   Ogre::Real mass;
   Ogre::Vector3 inertia;
   me->getMassMatrix(mass, inertia);

   Ogre::Vector3 force(0,-16.8,0);
   force *= mass;
   me->addForce(force);
   me->setOmega(Ogre::Vector3(0.0, angularVelocity, 0.0));
}
```

In dieser wird zunächst die Masse über die Routine getMassMatrix() erhalten, diese beträgt bei der Kiste 20 Kilogramm. Danach wird ein Vektor mit der höheren Gravitationskraft erstellt. Diese Kraft wird jedoch negativ in Richtung der Y-Achse angegeben (-16,8), da das Objekt zum Boden gezogen werden soll. Anschließend wird die

72 http://de.wikipedia.org/wiki/R%C3%BCckruffunktion, Internet: 07.07.2009

Kraft mit der Masse des Körpers multipliziert. Dabei werden alle Komponenten des Kraftvektors mit dieser verlängert:

```
force = (0.0, -16.8, 0.0) * 20.0
force = (0.0, -336.0, 0.0)
```

Das heißt, dass das Kistenmodell mit einer Kraft von 336 Newton nach unten gezogen wird. Zudem wird über setOmega() eine Winkelgeschwindigkeit[73] (engl. angular velocity) um die Y-Achse gesetzt. Dadurch lässt sich der Körper je nach Wert entsprechend schnell drehen. Nun muss die selbst geschriebene Funktion in OgreNewt eingebunden werden.

```
pBody->setCustomForceAndTorqueCallback<Class>(&Class::moveCallback, this);
```

Der Körper wird zum Beginn genauso positioniert und orientiert wie der Szenenknoten.

```
pBody->setStandardForceCallback();
pBody->setPositionOrientation(pSceneNode->getPosition(),
                              pSceneNode->getOrientation());
```

Danach muss der Körper nur noch mit einem Szenenknoten verbunden werden.

```
pBody->attachToNode(pSceneNode);
```

Ab diesem Punkt kann ein Ogre-Objekt nur noch über OgreNewt gesteuert werden.
Jetzt fehlen zum Abschluss noch Informationen zur Identifizierung, um im Fall einer Kollision herauszufinden, welcher Körper mit welchem kollidiert ist.
Dazu wird im ersten Schritt der Typ des Körpers festgelegt.

```
pBody->setType(3);
```

Danach wird ein physikalisches Material für den Körper festgelegt. OgreNewt vergibt pro Materialanfrage eine zufällige Material-ID.

```
OgreNewt::MaterialID *pMaterial = new OgreNewt::MaterialID(OgreNewt);
pBody->setMaterialGroupID(pMaterial);
```

73 http://de.wikipedia.org/wiki/Winkelgeschwindigkeit, Internet: 07.07.2009

Nun erhält der Körper über eine Funktion einen Zeiger auf die Klasse, in welcher er erzeugt wurde.

```
pBody->setUserData(this);
```

Diese Informationen kommen im nächsten Abschnitt zum Einsatz.

5.10.2 Kollisionsablauf

Im weiteren Verlauf wird nach [39] und [40] der Kollisionsablauf zwischen zwei Körpern erläutert.
Kollisionen laufen über OgreNewt automatisch ab. Will man jedoch zum Zeitpunkt einer Kollision zusätzliche Informationen auslesen, sind einige Schritte notwendig.
Für zwei unterschiedliche Körper wird eine Materialverknüpfung erzeugt, welche die Material-ID jedes Körpers erhält.

```
OgreNewt::MaterialPair* pMaterialPair = new OgreNewt::MaterialPair(
    pOgreNewt, pBody->getMaterialGroupID(),pBody2->getMaterialGroupID());
```

Mittels dieser Verknüpfung kann über die Physik-Engine ein Verhalten definiert werden. Es kann eine Reibkraft für die Körper eingestellt werden. Diese Kraft legt nach [23, S. 358-361] fest, wie ein Körper in Kontakt mit einem anderen bewegt wird. Es kann beispielsweise simuliert werden, wie eine Holzkiste auf einer Eisfläche bewegt wird.

```
pMaterialPair->setDefaultFriction(0.01, 0.01);
```

Der erste Parameter beschreibt die statische Reibungskraft, diese Kraft wirkt ein wenn ein Körper auf einem anderen ruhenden Körper nicht mehr bewegt wird. Bei einem hohen Koeffizienten wird das bewegliche Objekt sofort gestoppt und bleibt durch die hohe Reibung liegen. Bei einem niedrigen Koeffizienten rutscht es langsam weiter und wird erst nach einiger Zeit gestoppt, da kaum Reibung vorhanden ist. Der zweite Parameter legt fest, wie ein Objekt, welches sich in Bewegung befindet, die Reibungskraft spürt.
Des Weiteren kann die Elastizität eines Körper eingestellt werden, so dass bei einer Kollision ein Körper vom anderen abgestoßen wird. So kann beispielsweise das Springen auf einem Trampolin simuliert werden.

```
pMaterialPair->setDefaultElasticity(1.4);
```

Beim Kollisionsablauf zweier Körper kann ein Problem entstehen. OgreNewt wird in gewissen Zeitabständen[74] aktualisiert. Wenn ein Körper sich zu schnell auf einen anderen zu bewegt, könnte es passieren, das OgreNewt nicht rechtzeitig aktualisiert wird. Zu diesem Zeitpunkt könnte sich der Körper bereits in einem anderen befinden und würde einfach durch diesen hindurch fliegen. Daher kann nach [42] ein kontinuierlicher Kollisionsmodus eingeschaltet werden.

```
pMaterialPair->setContinuousCollisionMode(1);
```

Dadurch kann die Physik-Engine den Kontakt zweier Körper vorausberechnen.

Bis jetzt können aber nur physikalische Ereignisse von OgreNewt eingebunden werden. Will man jedoch ein eigenes Verhalten definieren, kann eine Klasse von einer OgreNewt Basis-Klasse abgeleitet werden. In dieser können drei Funktionen überschrieben werden.

```
class CustomCallback : public OgreNewt::ContactCallback
{
public:
            CustomCallback();
            ~CustomCallback();
            int userBegin();
            Int userProcess();
            void userEnd();
private:
            Character *pFlubber;
            OgreNewt::Body *pBody;
};
```

Die erste Funktion userBegin() teilt mit, dass sich gerade eine Kollision anbahnt. Des Weiteren kann über den im vorherigen Abschnitt gesetzten Typ des Körpers festgelegt werden, welcher Körper mit welchem kollidiert ist. Im Unterabschnitt 5.10.1 wurde ein Zeiger auf die Klasse über die Funktion setUserData() übergeben. Nun kann über die Funktion getUserData() die Klasse, in welcher der physikalische Körper erzeugt wurde, wieder erhalten werden. Über diese Klasse können benutzerdefinierte Funktionen ausgeführt werden.

74 Siehe Unterabschnitt 5.12

Die Funktion userProcess() wird ausgeführt, wenn zwei Körper tatsächlich kollidiert sind. Es können Informationen über die Kollision gesammelt werden. Hier kann zum Beispiel berechnet werden, dass ein schwächerer Körper, also einer mit weniger Masse vom stärkeren Körper weg gestoßen wird.
In der dritten Funktion userEnd() erhält man die letzte Chance auf eine Kollision zu reagieren. Anhand der gesammelten Daten kann nun zum Beispiel über die benutzerdefinierte Klasse eine Funktion aufgerufen werden um dem Objekt Energie abzuziehen und ein Geräusch abzuspielen.[75]

5.10.3 Bewegung und Orientierung

Objekte können auf verschiedene Weisen bewegt werden. Eine davon ist nach [23, S. 36] das Ändern der Position eines Objektes durch Zugabe von Geschwindigkeit (engl. Velocity) zu einem bestimmten Zeitpunkt. Das heißt, jedes mal wenn die Physik-Engine aktualisiert[76] wird, erhält das Objekt eine festgelegte Geschwindigkeit, beschrieben durch einen Vektor. Darüber hinaus muss bestimmt werden in welche Richtung das Objekt bewegt werden soll. In Ogre und OgreNewt können Objekte auf verschiedene Arten gedreht werden. Im Folgenden werden nach [23, S. 153-157] die Grundlagen zu den in dieser Arbeit relevanten Dreharten erläutert.

Eulersche Drehungen

Objekte können im dreidimensionalem Raum um drei Achsen gedreht werden. Im Bezug zu einem Flugzeug wurden die Begriffe Yaw, Pitch, Roll eingeführt. Yaw beschreibt eine Drehung um die Y-Achse um einen definierten Winkel, Pitch um die X-Achse und Roll um die Z-Achse. Abbildung 5.12 veranschaulicht dies.

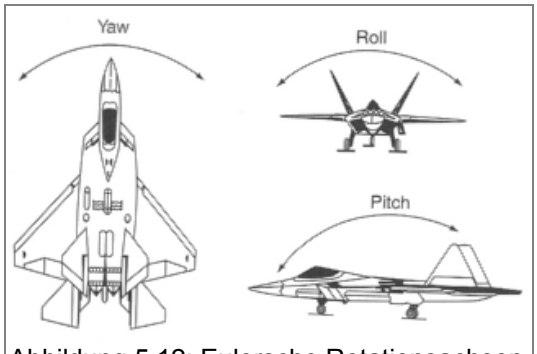

Abbildung 5.12: Eulersche Rotationsachsen

75 Siehe Kapitel 6.9

76 Im Unterabschnitt 5.13 wird näher darauf eingegangen.

Diese Rotationsart bringt jedoch Probleme mit sich. Wenn das Flugzeug beispielsweise zuerst 30° um die X-Achse und anschließend 30° um die Y-Achse gedreht wird, so erhält man ein anderes Ergebnis, als hätte man zuerst um die Y-Achse und danach um die X-Achse gedreht. Abbildung 5.13 stellt dieses Problem dar. Im ersten Teil der Abbildung wird ein Objekt 50° um die X-Achse, 120° um die Y-Achse und 80° um die Z-Achse gedreht. Im zweiten Teil wird in einer anderen Reihenfolge rotiert und das Ergebnis ist ein anderes (Quelle: guerrillacg[77]).

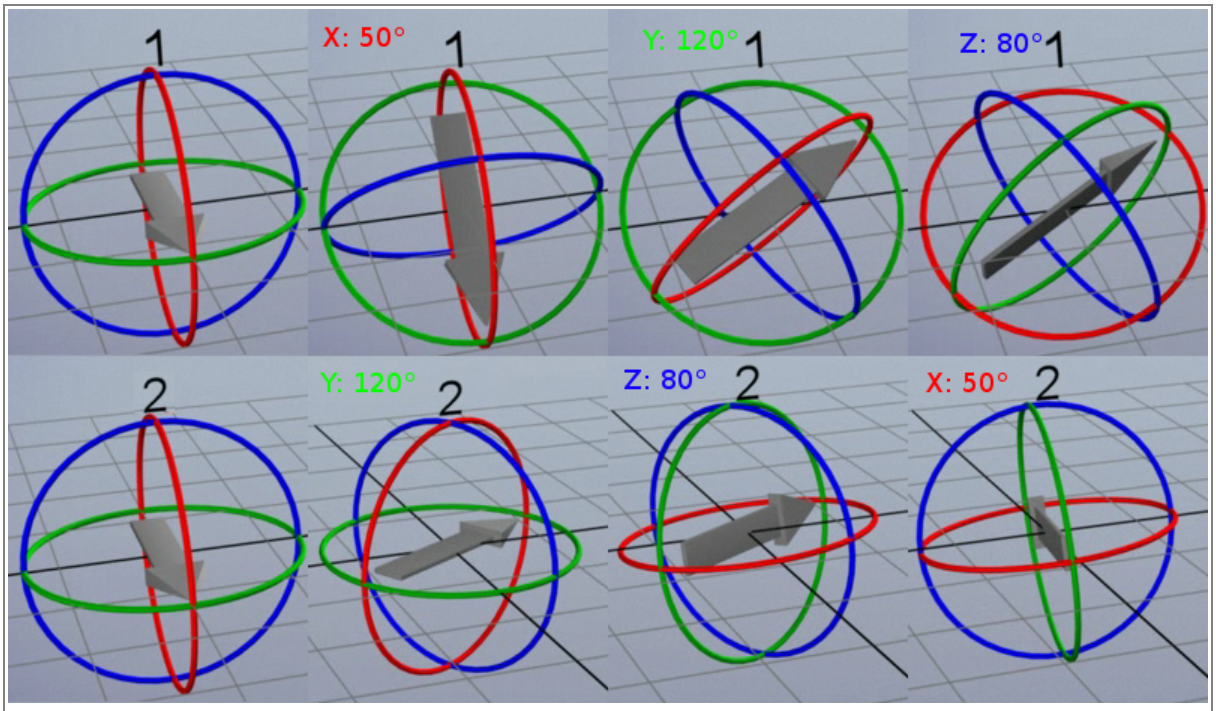

Abbildung 5.13: Drehungen in verschiedenen Reihenfolgen

Ein weiteres Problem ist die sogenannte Blockade der „Kardanischen Aufhängung“[78] (engl. Gimbal Lock). Dieses entsteht wenn ein Objekt um eine Achse so weit gedreht wird, dass es mit einer anderen gleichgerichtet ist. Dadurch geht eine Achse verloren und es ist nicht mehr möglich um diese zu drehen. Die Abbildung 5.14 veranschaulicht dieses Problem (Quelle: guerrillacg[79]).

77 http://www.guerrillacg.org/home/3d-rigging/the-rotation-problem, Internet: 10.07.2009

78 http://de.wikipedia.org/wiki/Gimbal_Lock, Internet: 10.07.2009

79 http://www.guerrillacg.org/home/3d-rigging/euler-rotations-explained, Internet: 11.07.2009

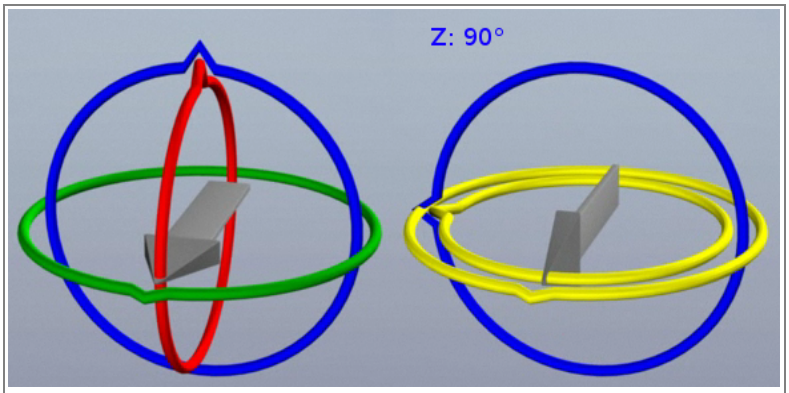

Abbildung 5.14: Kardanische Aufhängung

Nachdem das Objekt 90° um die Z-Achse gedreht wurde, ist diese mit der X-Achse gleichgerichtet. Eine Rotation um die X-Achse ist in diesem Fall ohne Weiteres nicht mehr möglich. Das Hauptproblem ist, dass im Raum, der durch drei Achsen beschrieben ist, eine eulerschen Rotation um diese nacheinander, je nach Reihenfolge der Achsen zu unterschiedlichen Ergebnissen führt, da eine Drehung auf einer vorherigen aufbaut.
Eulersche Drehungen sind in Ogre einfach zu bewerkstelligen und werden oft für Rotationen einer Kamera benutzt, daher auch in dieser Arbeit. Eine Kamera benötigt keine Rotation um die Z-Achse (Roll), deswegen taucht das „Gimbal Lock“ Problem nicht auf.

```
pCamera->pitch(30.0);
pCamera->yaw(30.0);
```

Durch Aufruf der Funktion mit Eingabe des Winkels kann die Kamera um 30° um die Y-Achse und 30° um die X-Achse gedreht werden. Die Reihenfolge spielt dabei keine Rolle, da der Benutzer die Orientierung des Spielers über die Maus vorgibt.

Quaternions

Abhilfe bei den zuvor genannten Problemen mit eulerschen Drehungen schaffen Quaternions. Ein Quaternion besteht aus vier Komponenten es ist also vierdimensional und beschreibt über diese eine Orientierung im dreidimensionalem Raum.

```
Q = (w, x, y, z)
```

Die Komponente „w“ beschreibt einen Winkel und „(x, y, z)“ einen Vektor. Mehrere Rotationen um verschiedene Achsen in unterschiedlicher Reihenfolge führen immer zum gleichen Ergebnis und eine „Kardanische Aufhängung“ kann auch nicht entste-

hen. Der Grund dafür ist, dass ein Quaternion mehr Informationen (die drei Drehachsen und den Winkel) speichert.
Der mathematische Hintergrund[80] von Quaternions ist sehr komplex, aber dennoch nicht Voraussetzung um mit ihnen arbeiten zu können, da Quaternions von Ogre implementiert wurden und OgreNewt diese auch unterstützt. Im Folgenden wird anhand von Beispielen erläutert, wie Drehungen über Quaternions in Ogre zu bewerkstelligen sind.
Zunächst einmal wird eine Operation benötigt, die nach [23, S. 28] in der Geometrie keine Interpretation besitzt. Diese nennt sich das „Komponenten Produkt“. Dabei werden die drei Komponenten eines Vektors miteinander multipliziert. In der Spielprogrammierung ist diese Operation sinnvoll um eine oder mehrere Vektorkomponenten auszuschließen.

```
Programmiertechnisch:
Ogre::Vector3 vec1(1.5, 2.0, 3.0);
Ogre::Vector3 vec2 = vec1 * Ogre::Vector3::NEGATIVE_UNIT_Z;

Mathematisch:
vec2 = (1.5, 2.0, 3.0) * (0.0, 0.0, -1.0)
vec2 = (0.0, 0.0, -3.0)
```

Mit Quaternions sind verschiedene Operationen möglichen, einige werden nun vorgestellt und erläutert.

- **Absolute Rotation**

  Ein Objekt wird 30° gegen den Uhrzeigersinn um die Y-Achse orientiert, egal wie es vorher gedreht war.

  ```
  Ogre::Quaternion q(Ogre::Degree(30), Ogre::Vector3::UNIT_Y);
  pSceneNode->setOrientation(q);
  ```

- **Relative Rotation**

  Die Drehung des Objektes erfolgt abhängig davon in welche Richtung es im Augenblick gerichtet ist.

  ```
  Ogre::Quaternion q(Ogre::Degree(60), Ogre::Vector3::UNIT_X);
  pSceneNode->rotate(q);
  ```

80 Nähere Informationen: http://www.euclideanspace.com/maths/algebra/realNormedAlgebra/quaternions/index.htm, http://de.wikipedia.org/wiki/Quaternion, Internet: 11.07.2009

Es ist nun 30° um die Y-Achse und 60° um die X-Achse orientiert. Nun ist ersichtlich, dass ein Quaternion im Gegensatz zu eulerschen Drehungen mehrere Rotationen auf einmal speichert.

- **Kombinationen von Rotationen**
  Rotationen können auf verschiedene Weisen kombiniert werden. Wenn Quaternions mit anderen ihrer Art um die gleiche Achse multipliziert werden, dann werden die Winkel addiert.

```
Ogre::Quaternion q1(Ogre::Degree(46), Ogre::Vector3::UNIT_Y);
Ogre::Quaternion q2(Ogre::Degree(44), Ogre::Vector3::UNIT_Y);

pSceneNode->setOrientation(q1 * q2);
```

- **Bildung eines Quaternions durch zwei Vektoren**
  Es kann ein Quaternion erzeugt werden, welches die Rotationsdaten von einem Vektor zu einem anderen speichert.

```
Ogre::Vector3 vec1(3.0, 0.0, 10.0);
Ogre::Vector3 vec2(7.0, 0.0, 5.0);

Ogre::Quaternion q = vec1.getRotationTo(vec2);
pSceneNode->setOrientation(q);
```

  In diesem Beispiel wird das Objekt ca. 33,5° um die Y-Achse gedreht.

- **Richtungsvektor aus einem Quaternion erhalten**
  Im vorherigen Beispiel wurde ein Quaternion aus Vektoren gebildet. Nun wird ein normalisierter[81] Richtungsvektor aus einem Quaternion gebildet. Dieses Quaternion muss jedoch mit einer Achse multipliziert werden, damit der Vektor im Bezug zu dieser Achse gerichtet ist.

```
Ogre::Vector3 dir = pSceneNode->getOrientation() *
                                Ogre::Vector3::NEGATIVE_UNIT_Z;
Bei 33.5° beträgt dir = (-0.551867, 0.0, -0.833932)
```

  Abbildung 5.15 zeigt die Orientierung im Bezug zur negativen Z-Achse.

81 Richtungsvektor mit der Länge 1.

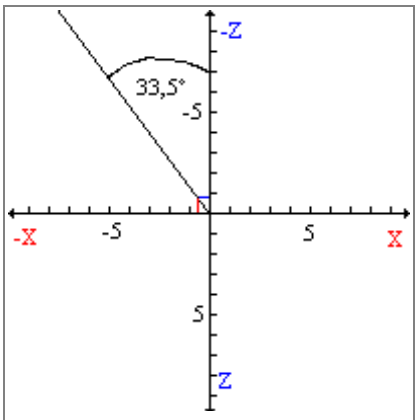

Abbildung 5.15: Orientierung im Bezug zur negativen Z-Achse

Die gleiche Operation mit der X-Achse multipliziert, bewirkt eine Rotation um den gleichen Winkel jedoch in eine andere Richtung.

```
Ogre::Vector3 dir = pSceneNode->getOrientation() * Ogre::Vector3::UNIT_X;
Bei 33.5° beträgt dir = (0.833932, 0.0, -0.551867)
```

Abbildung 5.16 zeigt die Orientierung im Bezug zur X-Achse.

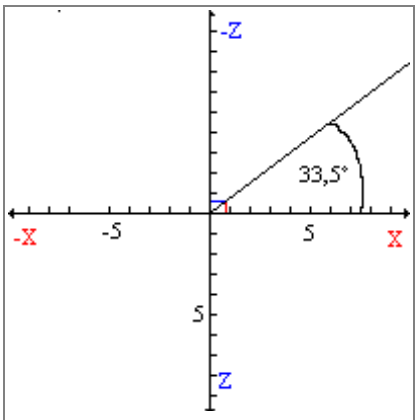

Abbildung 5.16: Orientierung im Bezug zur X-Achse

Damit wären die Grundlagen für Rotationen gegeben. Zum Schluss dieses Abschnitts wird erläutert, wie ein Objekt in eine definierte Richtung bewegt wird. Zunächst wird der Richtungsvektor benötigt.

```
Ogre::Vector3 dir = pSceneNode->getOrientation()* Ogre::Vector3::NEGATIVE_UNIT_Z;
```

Danach kann der physikalische Körper in diese Richtung mit einer festgelegten Geschwindigkeit bewegt werden.

```
pBody->setVelocity(pBody->getVelocity() * Ogre::Vector3::UNIT_Y
                   + dir * speed);
```

Da der Körper der Schwerkraft zu jedem Zeitpunkt ausgesetzt sein soll, wird beim Bewegen des Objektes die aktuelle Höhengeschwindigkeit (pBody->getVelocity() * Ogre::Vector3::UNIT_Y) festgehalten. Dies ist notwendig, damit die Schwerkraft auf die Y-Komponente einwirken kann, während ein Objekt beispielsweise Berg abwärts bewegt wird, sodass das Objekt am Boden bleibt. Hinzu wird der Richtungsvektor addiert und mit dem Faktor „speed“ verlängert.

5.11 OgreAL

OgreAL ist eine Bibliothek, die sich mit der Integration von OpenAL in die Ogre-Engine beschäftigt, so dass Hintergrundmusik und Geräuscheffekte mit Objekten aus der Ogre-Engine verbunden werden können. Der Akteur wird dabei als Zuhörer definiert. Daraus resultiert, dass die Lautstärke aller Geräusche abhängig von der Position ist. Die Orientierung des Akteurs legt fest, aus welchem Lautsprecher ein Geräusch ertönt. Im Folgenden wird gezeigt wie OgreAL initialisiert und eine Audiodatei eingebunden wird.

```
OgreAL::SoundManager *pSoundManager = new OgreAL::SoundManager();
OgreAL::Sound *pBackgroundSound = pSoundManager->createSound("Name","Musikname");
```

Anschließend muss ein Zuhörer definiert werden.

```
pSceneNode->attachObject(pSoundManager->getListener());
```

Zuletzt wird der Hintergrundmusik so eingestellt, dass diese relativ zum Zuhörer abgespielt wird.

```
pBackgroundSound->setRelativeToListener(true);
```

5.12 Spielschleife

In der Spielschleife werden alle Abläufe eines Spiels in ständiger Wiederholung abgearbeitet, bis dieses beendet wird. Die Ogre-Engine stellt dazu eine vordefinierte Spielschleife zur Verfügung. Diese Komponente nennt sich FrameListener.

Der FrameListener berechnet nach [41] durchgelaufene Bilder pro Sekunde[82], den Abstand zwischen zwei Bildern etc. Der FrameListener wird eingesetzt um die Szene zu aktualisieren und darzustellen. Wenn dieser registriert und die Methode Root::startRendering() aufgerufen wird, führt diese die Methode frameStarted jedes registrierten FrameListeners aus. Anschließend wird ein Bild gerendet und die frameEnded Methode jedes registrierten FrameListeners ausgeführt. OgreAL und Caelum sind auch intern als FrameListener implementiert.

In der frameStarted Funktion können alle Vorgänge im Spiel ausgeführt werden. Die Aktualisierung von OgreNewt muss auch in dieser Funktion vorgenommen werden. Diese wird standardmäßig intern mit ungefähr 60 Fps aktualisiert.[83] Es ist Notwendig beide Engines mit 60 Fps synchron zu halten, daher bietet sich die Einstellung VSync (Vertical Synchronisation) im Konfigurationsdialog der Grafik-Engine an. Wenn diese eingeschaltet wird, orientiert sich die Spielschleife an der Hertz-Zahl des Bildschirms.[84] Sie beträgt bei den meisten Bildschirmen 60 Hertz, also werden ca. 60 Bilder pro Sekunde berechnet. Die Methoden frameStarted und frameEnded stellen die Variable timeSinceLastFrame zur Verfügung. Diese hält fest wie viel Zeit zwischen zwei Bildern abgelaufen ist. Bei 60 Bildern pro Sekunde liegt diese bei ungefähr 0,0167 Sekunden[85]. Das heißt es vergehen 16,7 Millisekunden zwischen zwei Bildern. Die timeSinceLastFrame Variable ist von großer Bedeutung, die Geschwindigkeit aller Abläufe wie beispielsweise die Gravitationseinwirkung und die Bewegung und Animation der Charaktere sind von dieser Variable abhängig. Wenn die Bewegungsgeschwindigkeit des Spielers beispielsweise auf 8,4 Meter pro Sekunde gesetzt wird, dann ergibt sich für den Bewegungsfaktor die folgende Formel:

```
Bewegungsfaktor = (Geschwindigkeit * Zeit zwischen zwei Bildern) * 60
```

82 Engl. Frames Per Second (Abk. Fps)

83 www.newtondynamics.com/wiki/index.php5?title=NewtonSetMinimumFrameRate, Internet: 10.07.2009

84 www.uni-protokolle.de/Lexikon/Vsync.html, Internet: 10.07.2009

85 1/60 = ~0,0167 Sekunden

Diese Formel ist notwendig, da die Hertz-Zahl bei einigen Monitoren erhöht werden kann. Der Monitor des Autors kann beispielsweise auf 75 Hertz eingestellt werden. Bei 75 Fps würde die Bewegungsgeschwindigkeit des Spielers öfters aktualisiert werden und dieser sich somit schneller bewegen. Daher muss der Bewegungsfaktor abhängig von der Zahl der Fps angepasst werden.

```
Bei 60 Hz: Bewegungsfaktor = (8,4 * 1 / 60) * 60 = ~8,4 m/s
Bei 75 Hz: Bewegungsfaktor = (8,4 * 1 / 75) * 60 = ~6,72 m/s
```

Durch umfangreiche Tests stellte sich jedoch heraus, dass wenn die Anwendung mit mehr als 60 Fps läuft, der Spielablauf unsanft vonstatten geht, der Spieler und die Gegner bewegten sich nicht gleichmäßig. Daher wird das Spiel für 60 Fps optimiert.

5.13 Performance

Spiele in 3D-Welten werden wie in Kapitel 4.1 erläutert, in Echtzeit gerendert. Von daher spielt die Performance oder auch das Laufzeitverhalten eine große Rolle damit alle Abläufe flüssig vonstatten gehen. Die Performance ist über die Anzahl an durchgelaufener Bilder pro Sekunde feststellbar. Könnten nur wenige Bilder pro Sekunde berechnet werden, würde das Spiel hängen.
In diesem Abschnitt werden einige Maßnahmen erläutert, welche von der Grafik- und Physik-Engine schon getroffen werden und welche selbst getroffen werden müssen.
Eine Grafik-Engine benutzt nach [11, S. 217-218] für die Organisation von darzustellenden Objekten einen Szenengraphen[86]. Da eine Szene aus vielen Objekten besteht, muss das Rendern dieser effizient über den Szenengraph verwaltet werden. Die einfachste Form zur Darstellung ist, alle sich in der Szene befindlichen Objekte zu durchlaufen und diese zu rendern. Dies ist jedoch sehr performancelastig, da obwohl nicht alle Objekte in einer 3D-Welt sichtbar sind, alle Dreiecke[87] aller Objekte berechnet und dargestellt würden. Daher werden Maßnahmen getroffen um Rechenzeit zu sparen. Eine davon ist das „Clip-Culling“. Dabei wird getestet ob sich ein Objekt an oder innerhalb der Kamerasichtgrenze befindet. Dreiecke die sich außerhalb befinden werden abgeschnitten. Abbildung 5.17 veranschaulicht eine Gegenüberstellung eines 3D-Herz Modells im Dreieck-Rendermodus ohne Clip-Culling und mit.

86 Siehe Unterabschnitt 5.2

87 Dreiecke werden im Allgemeinen auch als Polygone bezeichnet.

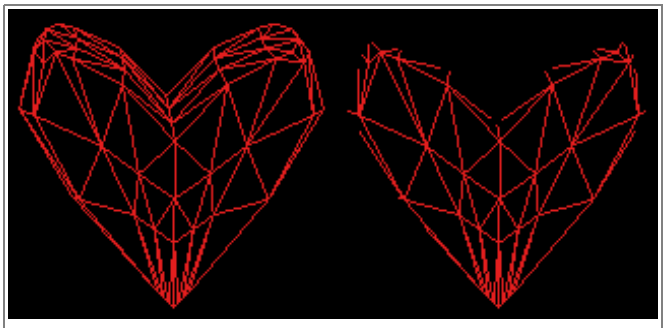

Abbildung 5.17: Screenshot - Clip-Culling

In Ogre wird der Anzeigebereich nach [10, S. 76-77] über eine Kamera definiert. Die aus [10, S. 92] stammende Abbildung stellt zwei Flächen, innerhalb welcher Objekte gerendert werden, dar.

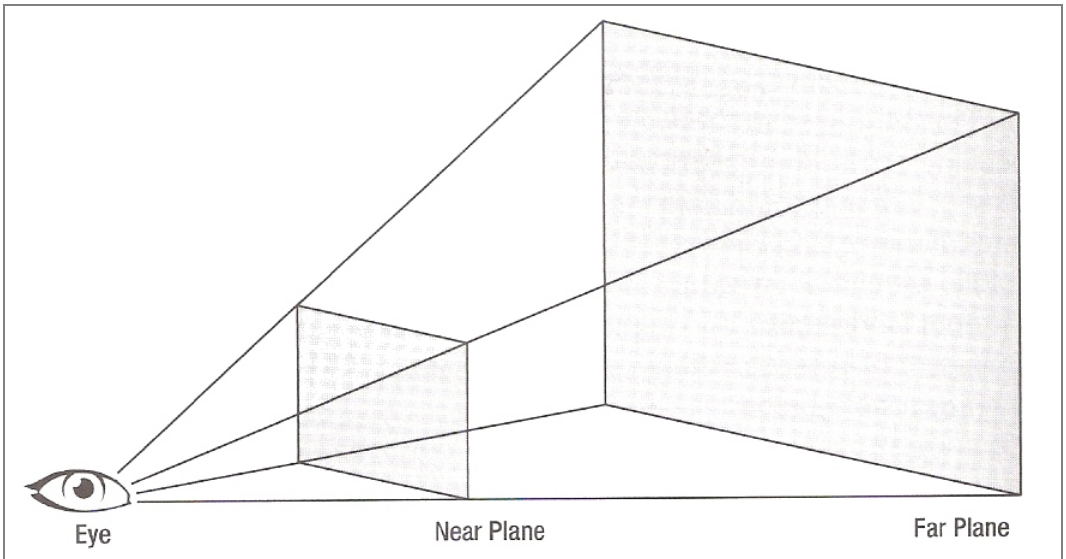

Abbildung 5.18: Sicht als Kegelstumpf (engl. view frustum)

Dieser Bereich kann über die Kamera eingestellt werden.

```
pCamera->setNearClipDistance(0.1);
pCamera->setFarClipDistance(250.0);
```

In diesem Fall wird alles ab einer Distanz von 0,1 bis 250 Welteinheiten dargestellt. Eine weitere Methode nach [20, S. 45] ist das „Backface-Culling“. Vom Betrachter aus sind immer nur die vorderen Dreiecke sichtbar, daher können alle Rückseiten vor dem Rendern entfernt werden, dadurch können im Durchschnitt 50% der Szenenkomplexität gespart werden.

Oft sind weit entfernte Objekte vom Betrachter nur grob und verschwommen zu erkennen, somit müssen diese nicht im Detail gerendert werden. Dies wird mit der Methode „Level Of Detail“[88] ermöglicht. Die Szenenkomplexität kann dadurch mit Zunahme der Distanz reduziert werden. Die Ogre-Engine bietet dieses Verfahren nach [10, S. 71, S. 98, S. 119] für die Kamera, 3D-Modelle und Materialien an. Ein 3D-Modell mit vielen Dreiecken kann mit verschiedenen LOD-Stufen als Mesh exportiert werden. Das bedeutet je nach Distanz wird das gleiche Modell geladen, jedoch mit weniger Polygonen. In Kapitel 6.3 wird die Anwendung von LOD in Materialien beschrieben.
Eine Physik-Engine strebt die Performance bei physikalischen Abläufen ebenfalls an. Ein wichtiges Prinzip nach [23, S. 383] ist, physikalische Objekte in einen „Sleep“ Zustand zu versetzen. Im Allgemeinen werden in einer Schleife alle physikalischen Objekte der Physik-Engine durchlaufen und deren Position, Orientierung etc. aktualisiert. Objekte, welche sich eine gewisse Zeit nicht bewegt haben, werden von der Physik-Engine „schlafen“ gelegt, das heißt sie werden aus dieser Schleife entfernt. OgreNewt macht da keine Ausnahme. Standardmäßig behandelt OgreNewt automatisch wann ein Körper „schlafen“ gelegt wird. Es besteht aber auch die Möglichkeit, die „Sleep“ Behandlung für bestimmte Objekte selbst festzulegen.
Der Entwickler kann also einige Methoden von der Grafik und Physik-Engine benutzen um die Performance anzupassen. Über den FrameListener der Ogre-Engine wird eine wichtige Statistik angeboten. Diese berechnet die Anzahl der Dreiecke in unmittelbarer Umgebung, beziehungsweise bis zur Kamerasichtgrenze. Es können mithilfe dieser Aussagen über die Performance eines Spiels gemacht werden. Bei der Editierung der 3D-Welt, welche in Kapitel 6.5 kurz beschrieben wird, muss zum Beispiel darauf geachtet werden, dass in einem bestimmten Bereich die Anzahl der Dreiecke nicht zu hoch ist, sonst würde das Spiel je nach Leistungsstärke eines Computers zu ruckeln anfangen. Der Entwickler muss also darauf achten, dass die 3D-Welt von der Anzahl und Komplexität der 3D-Modelle ausgewogen editiert wird. Des Weiteren kann er mittels LOD bestimmen, wie die Modelle und Materialien bei einer bestimmten Distanz in Erscheinung treten. Zudem muss darauf geachtet werden, dass nicht zu viele physikalische Effekte in einem Spiel auf einmal stattfinden, dies würde auch einen Einbruch in die Performance bewirken.

88 Abk. LOD

6 Realisierung

Sämtliche in Kapitel 3.3 geplanten Schritte werden im Folgenden in die Tat umgesetzt. Zudem werden die in Kapitel 5 gegebenen Grundlagen angewendet. Es wird ein 3D-Jump'n'-Run Spiel mit Verbindung zu einem Rollenspiel entwickelt, welches den Namen „Flubbers“ trägt. Die Realisierung erfolgt in der Entwicklungsumgebung Eclipse[89] und in der Programmiersprache C++.

6.1 Prinzipielle Vorgehensweise

In diesem Abschnitt wird ein Überblick über die prinzipielle Vorgehensweise bei der Realisierung eines Spiels gegeben.

Ressourcen

Es werden alle in Tabelle 3.1 geplanten Modelle mit einer 3D-Grafik-Software erstellt, oder zum Teil frei verfügbare Modelle aus dem Internet verwendet. Die Texturen für die erstellten Modelle werden mit einem Bildbearbeitungsprogramm angefertigt und mittels der 3D-Grafik-Software in geeigneter Weise auf die Modelle gemappt. Im Anschluss werden die entsprechenden Animationen der Charaktere und der Gegner vollendet. Anschließend werden die kompletten Modelle in ein für eine Grafik-Engine konformes Format exportiert, damit diese dargestellt und die Animationen gegeben falls ausführt werden können. Zudem wird in Materialdateien das Aussehen der Modelle erstellt, sowie die Anpassung der Lichtreflektionen mit Hilfe von Shadern vorgenommen.
Um Partikeleffekte simulieren zu können, werden vorgefertigte angepasst und eingebunden.
Außerdem werden Geräuscheffekte entweder aus dem Internet verwendet oder selbst aufgenommen und in das entsprechende Format exportiert, um diese mit einer 3D-Audio Bibliothek abspielen zu können.

Programmstruktur

Um mit der Programmierung des Spiels beginnen zu können, wird ein Klassendiagramm mit allen benötigten Klassen erstellt. Die Funktionsweise der einzelnen Klas-

89 http://www.eclipse.org, Internet: 12.07.2009

sen wird beschrieben.

Intro

Es wird ein Intro mit dem Titel des Spiels erstellt, welches nach dem Start der Anwendung erscheint. Zudem wird ein Ablauf mit einer Vorgeschichte erstellt, um den Benutzer in das Spiel einzuführen.

3D-Welt

Die komplette Umgebung wird mittels eines Leveleditors erstellt. Zu diesem Zweck kommen die modellierten Objekte zum Einsatz. In diesem Editor wird die Landschaft geformt und das Aussehen bestimmt. Danach werden die Modelle geeignet platziert, sodass eine komplette 3D-Welt entsteht. Diese wird in ein geeignetes Format abgespeichert und in das Spiel geladen. Damit der Spieler und die Gegner sich in dieser Welt bewegen können, erhält die Landschaft mit sämtlichen Objekten geeignete Kollisionshüllen. Dazu kommt eine Physik-Engine zum Einsatz, mittels welcher diese Aufgabe gemeistert werden kann.
Außerdem wird ein Himmel mit Hilfe einer Bibliothek, welche die Berechnung und Darstellung beinhaltet, erstellt.

Spielschleife

Es wird eine Spielschleife erstellt, in welcher das Spiel dargestellt und der Ablauf gesteuert werden.

Charaktere

Das Modell des Spielers und die Modelle der Gegner erhalten geeignete Kollisionshüllen und werden mit der Physik-Engine verbunden, damit alle physikalischen Abläufe mit Hilfe von dieser implementiert werden können. Zudem werden Geräuscheffekte und Partikeleffekte eingebunden. Des Weiteren erhalten die Charaktere verschiedene Attribute, damit diese beim Spielablauf Aktionen ausführen können.

Spielablauf

Je nach Bewegung eines Charakters wird eine entsprechende Animation abgespielt. Der Spieler wird über die Tastatur bewegt und über die Maus orientiert, während die Gegner vom Computer gesteuert. Sie erhalten eine künstliche Intelligenz, welche

über Zustandsautomaten beschrieben wird. Zur Veranschaulichung wird ein Zustandsdiagramm erstellt, in welchem das Gegnerverhalten nachvollzogen werden kann. Zudem wird ein Kampfsystem entwickelt, damit das Spiel an Reiz gewinnt. Der Spieler muss Gegner mit einer Kampfattacke besiegen und die Gegner müssen versuchen, den Spieler zu eliminieren. Außerdem wird der Spieler mit einigen Fähigkeiten ausgestattet, um sich durch die Welt schlagen zu können. Dies wird ermöglicht, indem eigenes Verhalten mittels der Physik-Engine bei einer Kollision zwischen verschiedensten Objekten eingebunden wird. Je nach Objektart wird ein anderes Verhalten bei einer Kollision definiert.
Damit das Spiel jedes Mal fortgeführt werden kann, wird beim Betreten einer Speicherstelle der Spielstand abgespeichert.

Im weiteren Verlauf wird diese Vorgehensweise mit den in Kapitel „Werkzeuge“ ausgewählten Ressourcen beschrieben.

6.2 Modellierung

In diesem Abschnitt wird anhand der Erstellung des „Flubber“-Charakters ein Einblick in die Modellierung, Texturierung und Animation mit Blender gegeben. Viele Informationen und Tutorials über die Modellierung mit Blender können aus Quelle [43] entnommen werden. Auf alle weiteren Modelle aus Tabelle 3.1 kann aus Platzgründen nicht eingegangen werden. Im Folgenden wird zuerst der Grundkörper erzeugt.

Grundkörper

Bei der Erstellung des Grundkörpers wird sich an der Abbildung 3.1 orientiert. Der Körper soll eine runde gummiartige Form besitzen, er wird aus einer Kugel geformt. Danach werden zwei kugelförmige Augen erstellt. Zuletzt wird der Mund modelliert, dieser entsteht durch geeignete Verformungen eines Würfels. Der in Abbildung 6.1 dargestellte Charakter besteht nun aus drei Teilmodellen, welche separat angesprochen werden können.

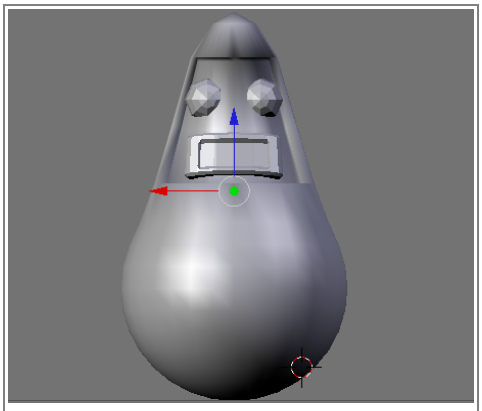

Abbildung 6.1: Screenshot - Charakter Grundkörper

Animation

Im Folgenden soll das Modell durch Animationen zum virtuellen Leben erweckt werden. Eine geeignete und in Ogre unterstützte Animationsart ist nach [43, S. 219] die sogenannte „Skelettanimation“. Wie der Name schon sagt, erhält ein Objekt dabei ein Skelett. Durch Bewegung, Skalierung oder Drehung dieses können umliegende Polygone mit verformt werden. Eine Animation kann nach [43. S. 199] über Schlüsselpositionen (engl. Keyframes) gesteuert werden. Dabei wird eine solche bestimmt und die Zwischenschritte bis zu dieser Schlüsselposition interpoliert Blender. Nachfolgend wird kurz erläutert wie eine Geh-Animation zustande kommt. Ein detailliertes Tutorial für eine Animation findet sich in Quelle [43, S. 199-225].

Um eine Animation starten zu können, wird ein Skelett mit Knochen benötigt. Der Charakter erhält zwei Knochen Ober- und Unterknochen, in Abbildung 6.2 rosa dargestellt.

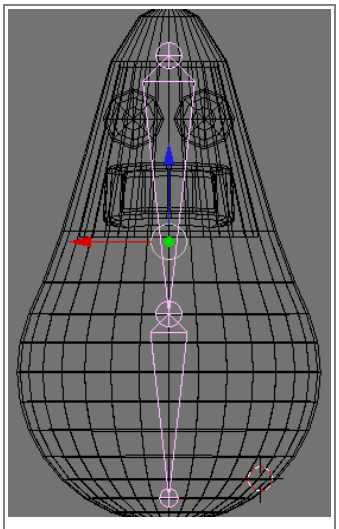

Abbildung 6.2: Screenshot - Skelett

Über einen Verbindungspunkt (Drehgelenk) kann ein Knochen bewegt werden. Zuerst wird der zu bewegende Knochen gewählt, dann der Animationsanfang über den Keyframe festgelegt. Danach kann eine Animationsart (Bewegen, Drehen oder Skalieren) ausgewählt werden. In diesem Fall wird die Animationsart „Drehen“ gewählt, der Unterknochen so um die X-Achse gedreht, dass eine Geh-Animation entsteht und das Animationsende über einen weiteren Keyframe gesetzt. Anschließend wird der Oberknochen gewählt und ein wenig nach oben bewegt, damit das Modell sich später hüpfend fortbewegt.

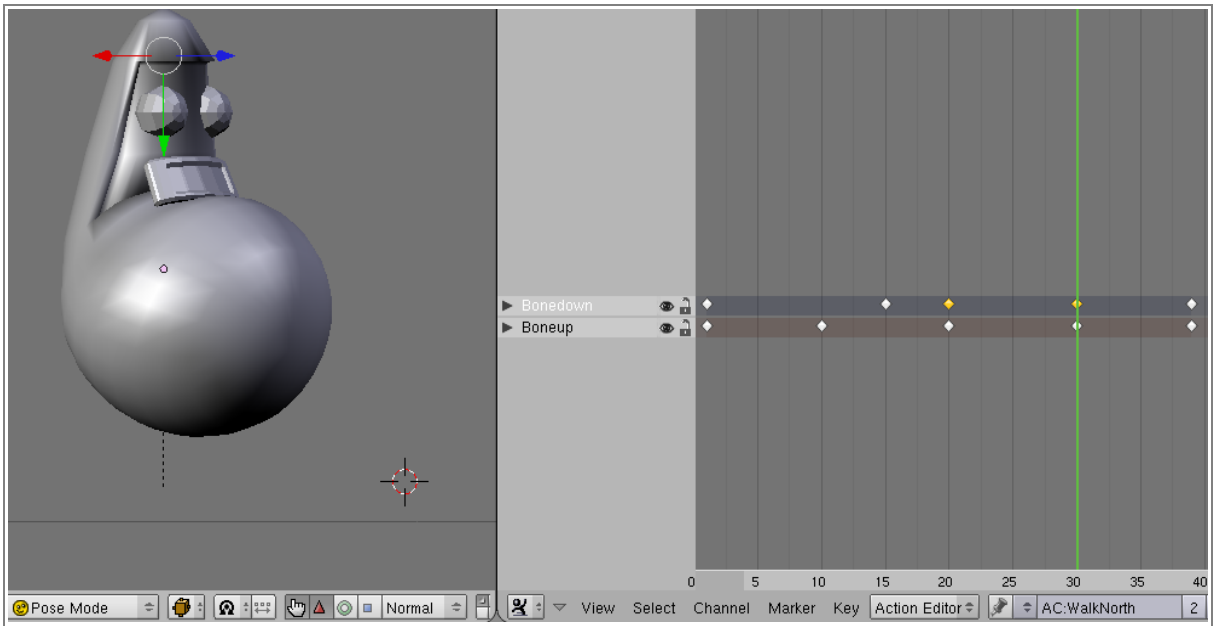

Abbildung 6.3: Screenshot - Geh Animation

Für das Spiel werden 11 Animationen benötigt. Diese werden in Tabelle 6.1 gezeigt.

WalkNorth	WalkSouth	WalkWest	WalkEast	Salto	Kick
Jump	Idle	Hurt1	Hurt2	Dead	

Tabelle 6.1: Animationsarten des Charakters

Texturierung

In diesem Unterabschnitt entstehen im Bildbearbeitungsprogramm Gimp die Texturen und werden auf das zuvor erstellte Modell gemappt.

Im ersten Schritt wird in Gimp ein linearer Farbverlauf von Dunkelgrün nach Hellgrün gezeichnet. Danach wird ein Lichteffekt im Zentrum des Bildes generiert. Durch die-

sen soll der Charakter beim Einsatz eines Shaders[90] ein wenig glänzen. Über einen Lupeneffekt wird das Bild etwas verzerrt. Zum Schluss wird noch der Filter „Nahtlos machen“ eingesetzt. Dieser sorgt dafür, dass das Bild in alle Richtungen wiederholt wird, da das Bild über den Charakter gelegt wird und Anfang und Ende des Bildes in etwa gleich aussehen sollen. In Abbildung 6.4 werden alle Schritte von links nach rechts veranschaulicht.

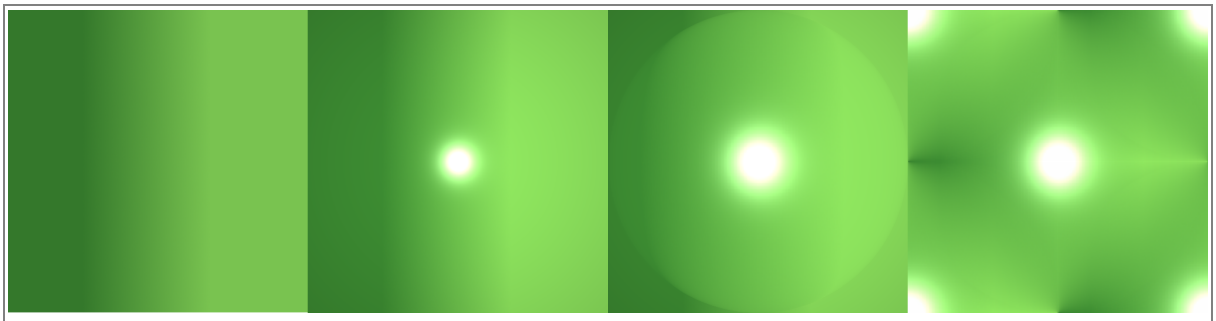

Abbildung 6.4: Screenshot - Charakter Textur

Mit diesem Verfahren können Texturen in verschiedensten Farbverläufen erstellt werden, so dass unterschiedliche Gegner im Spiel entstehen. Nun wird der Körper gewählt und in Blender der Bildeditor-Modus eingeschaltet. Das in Gimp erzeugte Bild wird in den Bildeditor geladen. Dann wird eine spezielle Funktion ausgeführt, welche die Polygone des Charakter-Körpers auf der Textur abbildet. Die Texturierung des Mundes und der Augen geschieht auf ähnliche Weise. Abbildung 6.5 stellt den fertigen Charakter dar.

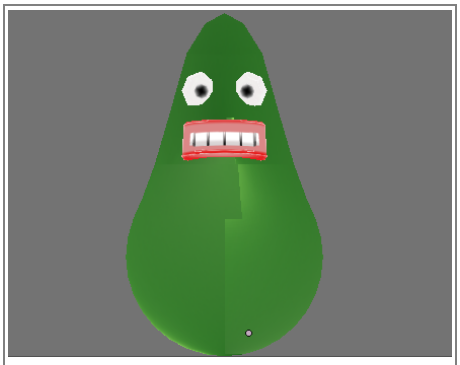

Abbildung 6.5: Screenshot - Charakter texturiert

Die Abbildung der Textur verlief nicht optimal, im obigen Bild ist ein deutlicher Farbunterschied zu sehen. Dieses Problem wird im nächsten Abschnitt beseitigt.

90 Siehe Unterabschnitt 6.3

6.3 Material

Im Kapitel 5.4 wurden Grundlagen für Materialien und Shader erläutert. Diese werden nun angewendet um das Aussehen des Charakters realistischer zu gestalten.
In Ogre existieren bereits eine Menge vorgefertigter Shader. Für den Körper des Charakters wird der Shader „env_map spherical“ benutzt. Environment maps lassen ein Objekt glänzen.[91] Abbildung 6.6 zeigt eine Gegenüberstellung des Charakters ohne und mit dem Shader.

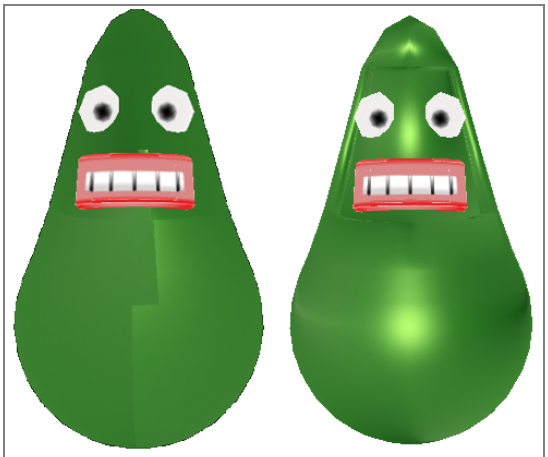

Abbildung 6.6: Screenshot - Charakter ohne und mit Shader

Durch Anwendung des Shaders ist der zuvor beschriebene Farbunterschied verschwunden.
In Kapitel 5.13 wurde die LOD Methode erläutert um Rechenzeit zu sparen. Nun erfolgt am Beispiel des Materials für den Grundkörper des Charakters der Einsatz von LOD. Abbildung 6.7 zeigt eine in zwei Bereiche aufgeteilte Materialdatei.
Der Befehl „lod_distances 50“ im obigen Bild bedeutet, dass von 0 bis 50 Welteinheiten die Technik mit „lod_index 0“ und ab 50 Welteinheiten die Technik „lod_index 1“ ausgeführt wird.[92] Für einen Gegner bedeutet dies, dass wenn dieser mehr als 50 Meter vom Spieler entfernt ist, die Basistextur ohne den Shader geladen wird. Für alle weiteren Modelle, welche aufwendig gestaltete Oberflächen besitzen, kann dies ähnlich eingestellt werden.

91 Vgl. www.ogre3d.org/docs/manual/manual_17.html#SEC93, Internet: 13.07.2009
92 Vgl. http://www.ogre3d.org/docs/manual/manual_15.html#SEC33, Internet: 13.07.2009

```
material SOLID/TEX/bodysurface1.png
{
    lod_distances 50
    technique
    {
        lod_index 0
        pass
        {
            texture_unit
            {
                texture bodysurface1.png
                env_map spherical
            }
        }
    }
```

```
    technique
    {
        lod_index 1
        pass
        {
            texture_unit
            {
                texture bodysurface1.png
            }
        }
    }
}
```

Abbildung 6.7: Screenshot - Materialdatei des Charakterkörpers

6.4 Beschreibung der Klassen

Das zu realisierende Spiel setzt sich aus mehreren Klassen, welche miteinander verknüpft sind, zusammen. Im Folgenden wird die Hierarchie und Funktionsweise der einzelnen Klassen beschrieben. Abbildung 6.8 gibt einen Überblick über die Klassenhierarchie.

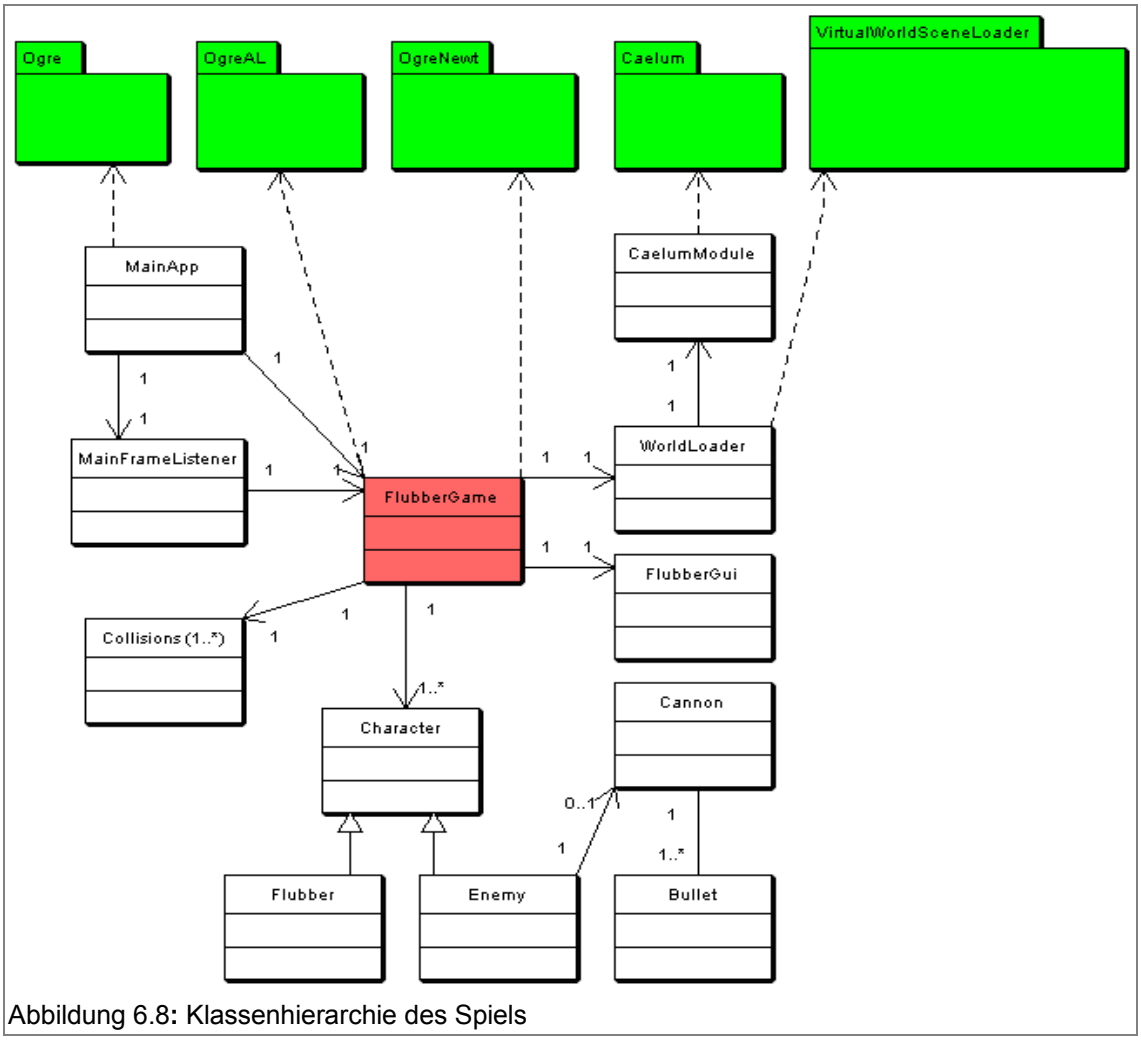

Abbildung 6.8: Klassenhierarchie des Spiels

Bei den grün gefärbten Paketen handelt es sich um die Bibliotheken, welche in Kapitel 5 beschrieben wurden. Diese werden zur Laufzeit eingebunden.

MainApp

In dieser Klasse wird die Ogre-Engine initialisiert. Des Weiteren werden die Klassen FlubberGame 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 eingeben und eine Uhrzeit definiert. Der Tagesverlauf des Spiels orientiert sich an der Uhrzeit.

WorldLoader

Der WorldLoader ist verlinkt mit der Bibliothek VirtualWorldSceneLoader. Diese Bibliothek ist wiederum aus dem früheren Projekt „Virtual World Editor“ entstanden und ist für das Laden aller Objekte, sowie der Landschaft verantwortlich. Der WorldLoader verwaltet alle Objekte und die Landschaft und übermittelt diese an die Klasse FlubberGame zur weiteren Verarbeitung. Die Klasse CaelumModul wird im WorldLoader initialisiert.

FlubberGui

FlubberGui wird zur Statusanzeige des Spiels benötigt. In ihr wird eine einfache grafische Oberfläche bestehend aus Labels, welche in einen Rahmen angeordnet werden, entwickelt. In den Labels werden notwendige Informationen wie zum Beispiel die Energie, 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.[93]

93 Siehe Kapitel 5.10.2 und Unterabschnitt 6.9

Charakter

Die Charakterklasse teilt sich in zwei Unterklassen Flubber und Enemy auf. In der Charakterklasse werden alle gemeinsamen Funktionen der Unterklassen implementiert. Bei der Initialisierung wird das 3D-Modell erzeugt sowie der physikalische Körper. Des Weiteren werden Geräuscheffekte erstellt.[94]

Flubber

Flubber ist von der Charakterklasse geerbt, somit übernimmt diese Klasse alle Standarteinstellungen der Oberklasse, zusätzlich werden aber noch viele weitere Routen implementiert. Der Flubber wird über den Benutzer gesteuert. Beim Erzeugen dieser Klasse werden Spielerattribute wie Energie, Erfahrung, Geschwindigkeit, Animationsarten etc. definiert. Des Weiteren werden Animationsabläufe für die Bewegung und Aktionen des Spielers entwickelt.[95]

Enemy

Diese Klasse ist eine weitere Unterart der Charakterklasse. Die Positionen, Skalierungen und Orientierungen der Gegner werden mithilfe einer Liste, welche über den WorldLoader verfügbar ist, festgelegt. Zudem wird anhand des Skalierfaktors definiert, um welche Gegnerart es sich handelt. Je nach dieser werden andere Attribute und Aktionen entwickelt.[96]

Cannon

Bei der Klasse Cannon handelt es sich um eine bestimmte Kampfart (Attackmode) eines Gegners. Dieser erhält eine Kanone, welche durch die Klasse Cannon beschrieben wird. Die Kanone wird an den Oberknochen des Gegnerskeletts gehängt. Somit wird diese von der Position und Orientierung her mit allen Bewegungsabläufen des Gegners synchronisiert.

Bullet

Eine Kanone besitzt eine Liste von Kugeln, da die Möglichkeit besteht, dass mehrere Kugeln in kurzen Zeitabständen vom Gegner geschossen werden und sich im

94 Siehe Unterabschnitt 6.7

95 Siehe Unterabschnitt 6.7.1

96 Siehe Unterabschnitt 6.7.2 und 6.8

Umlauf befinden. Eine Kugel wird durch die Klasse Bullet definiert. Sie erhält ein 3D-Kugel-Modell sowie einen geeigneten Kollisionskörper. Die Kugel wird in Richtung des Spielers geschossen und von der Liste entfernt. Wenn eine Kollision stattfindet, so wird die Kugel aus der Welt entfernt und wieder in die Liste eingefügt. Bei einer Kollision von Kugel und Spieler wird dem Spieler Energie abgezogen. Damit die Kugel weiß aus welcher Kanone sie geschossen wurde, wird zwischen den beiden Klassen Cannon und Bullet eine bidirektionale Verbindung gewählt.

FlubberGame

Die Klasse FlubberGame ist für die Verwaltung des Spiels verantwortlich. Sämtliche zuvor beschriebenen Klassen werden in FlubberGame definiert. Sie wird mit den Bibliotheken OgreAL und OgreNewt verlinkt. Im ersten Schritt wird die Physik-Engine initialisiert.[97] Danach wird ein SoundManager Objekt von der 3D-Audio Bibliothek OgreAL erstellt.

In dieser Klasse wird auch das Intro erstellt, es erscheint der Titel „Flubbers“. Als nächstes wird die Statusanzeige erzeugt. Danach wird die 3D-Welt eingebunden, der Spieler erzeugt, sowie alle Gegner geladen. Anschließend werden Materialverknüpfungen[98] für Kollisionen zwischen sämtlichen physikalischen Körpern erstellt. Die Simulation der künstlichen Intelligenz von Gegnern wird in dieser Klassen ebenfalls implementiert.

MainFrameListener

Diese Klasse ist von einem FrameListener[99] der Ogre-Engine abgeleitet, sowie von zwei Klassen zur Verwaltung von Tastatur und Mausnachrichten. Im MainFrameListener werden die in FlubberGame definierten Abläufe ausgeführt. In dieser Klasse wird die Ogre- und die Physik-Engine aktualisiert. Des Weiteren werden die in FlubberGame definierten Abläufe ausgeführt. Zudem erfolgt die Steuerung des Spielers mittels Tastatur und Maus.

97 Siehe Kapitel 5.10

98 Siehe Kapitel 5.10.2

99 Siehe Kapitel 5.12

6.5 Editierung der Welt

In diesem Abschnitt wird grob beschrieben wie eine 3D-Welt entsteht. Die Editierung einer 3D-Welt ist ein langer und vor allem kreativer Prozess, daher entsteht sie nach und nach.

Die Welt wird mit Hilfe des Virtual World Editors erzeugt. In den Quellen [13], [14] und [15] wird die Funktionsweise des Editors erläutert.

Zunächst wird der Assistent aufgerufen um die Umgebung festzulegen. In diesem kann im Bereich „Boden“ die „editierbare Oberfläche“ gewählt werden. Dort wird die Größe der Welt und eine Gras-Textur als Standardmaterial angegeben. Diese Bodenart ist Voraussetzung für das Formen und Malen einer Landschaft. Danach wird der Assistent abgeschlossen und es wird zunächst eine flache Ebene generiert.

Danach erfolgt das Formen der Landschaft. Es werden Berge an geeigneten Stellen gezogen, die dem Spieler als Hindernis dienen sollen. Bergspitzen werden mit Eistexturen versehen. Hinzu wird ein Pfad auf die Landschaft gemalt, welcher aus einer Erde-Textur besteht, an diesem Pfad soll sich der Spieler orientieren und ihm bis zum Ziel folgen können.

Als nächstes wird die Natur erzeugt. Zunächst werden mehrere kleinere Anhebungen und Hügel geformt. Dann werden an verschiedenen Stellen Bäume gesetzt, sodass Wälder entstehen. Es werden abwechselnd verschiedene Arten von Bäumen gesetzt und zufällig gedreht um ein wenig Dynamik in die Natur zu bringen. Außerdem werden Gebäude und Mauern platziert. Damit das Spiel nicht zu leicht vom Spieler bewältigt werden kann, werden Hindernisse erstellt, wie beispielsweise Schluchten, über welche der Spieler mittels eines Trampolins springen muss, oder Plattformen platziert, auf welchen er sich halten muss.

Zum Schluss werden überall Gegner an geeigneten Stellen in die Welt platziert. Im Spiel soll es unterschiedliche Gegner geben, daher werden sie geeignet skaliert. Anhand des Skaliervektors wird die Gegnerart bestimmt. Ist ein Gegner zum Beispiel mit dem Vektor (1.1, 1.0, 1.1) skaliert, so handelt es sich um den Gegnertyp zwei, dieser erhält im Spiel später eine braune Textur. Abbildung 6.9 stellt einen Ausschnitt der 3D-Welt für das Spiel dar.

Abbildung 6.9: Screenshot - 3D-Welt von Flubbers

Die erstellte Welt wird zum Schluss abgespeichert, sodass diese im nächsten Unterabschnitt geladen werden kann.

6.6 Welt Laden

Um die editierte Welt in das Spiel laden zu können haben der Autor und ein Kommilitone zuvor die Bibliothek „VirtualWorldSceneLoader“ entwickelt, welche in das Spiel eingebunden werden kann und diese Aufgabe übernimmt. Die Daten, wie beispielsweise Name, Position, Skalierung, Orientierung etc. der Modelle werden aus einer XML-Datei ausgelesen und Entities über die Ogre-Engine erstellt. Anhand des Namens eines Entities wird festgelegt, welchen Kollisionskörper das Modell erhält. In Tabelle 3.1 wurde ein Überblick über verschiedene Gruppen von Objekten gegeben und in Kapitel 5.10.1 wurde auf Kollisionskörper eingegangen. Sämtliche feste Objekte erhalten eine Baumkollision. Eine Ausnahme gilt für Bäume, da die Baumkollision-Funktion[100] auch über alle Äste eines Baumes eine Hülle formen müsste. Daher wird für Bäume eine zylinderförmige Hülle gewählt.

100 Zum Begriff „Baumkollision“ siehe Kapitel 5.10.1

Abbildung 6.10: Screenshot - Kollisionshüllen von Bäumen

Büsche erhalten keine Kollisionskörper, damit der Spieler und die Gegner hindurch laufen können. Bei beweglichen Objekten wird die Hülle je nach Art des Objektes angepasst. Handelt es sich beispielsweise um eine Kugel, so wird eine sphärische Kollisionshülle erstellt.

Je nachdem ob bei einer Objektart später Kollisionen abgefangen werden sollen um eigenes Verhalten einzubinden, werden dementsprechend Material-IDs vergeben und Objekt-Typen gesetzt. Die Charaktere werden in dieser Bibliothek nicht erstellt. Es wird nur eine Liste mit Entity-Informationen an die Hauptanwendung übermittelt, um aus diesen im Unterabschnitt 6.7 die Charakter zu erstellen. Gegner erhalten zudem einen speziellen Abfrage-Typ, um die Szenenabfrage[101] in Ogre auf diese zu beschränken.

In der Klasse WorldLoader wird für die Landschaft über die Baumkollision-Funktion eine Kollisionskarte erstellt. Die Welt besteht aus (257 * 257) Dreieckverbindungen, somit müssen 66049 Dreiecke durchlaufen werden. Daher dauert das Laden der Landschaft ein wenig.

6.7 Charakter

In diesem Abschnitt wird die Charakterklasse genauer erläutert. Bei dieser Klasse handelt es sich um die Basisklasse der Klasse Flubber (Spieler) und Enemy, daher

101 Siehe Kapitel 5.3

gelten alle beschriebenen Aspekte im weiteren Verlauf für den Spieler und die Gegner.

Im vorherigen Abschnitt wurde erwähnt, dass Informationen über einen Charakter in eine Liste angefügt wurden, diese wird nun verwendet um einen Charakter zu erzeugen.

Zunächst wird ein Entity erstellt. Damit der Name eines Charakters eindeutig ist, wird pro Aufruf der Charakterklasse eine statische Variable hochgezählt. Der Name für den zu erstellenden Szenenknoten wird genauso ermittelt. Anschließend wird das Entity an diesen angehängt. Die Informationen über die Position, Skalierung und Orientierung eines Charakters werden der Liste entnommen. Außerdem wird ein Partikelsystem, welches Rauch simulieren soll, erstellt. Dieses Partikelsystem soll später ausgeführt werden, wenn ein Charakter stirbt. In nächsten Schritten wird ein geeigneter Kollisionskörper erstellt. Als Kollisionsform eignet sich die konvexe Hülle[102].

```
OgreNewt::CollisionPrimitives::ConvexHull *pFlubberCol = new
OgreNewt::CollisionPrimitives::ConvexHull(pOgreNewt, pCharNode);

OgreNewt::Body *pPhysicsBody = new OgreNewt::Body(pOgreNewt, pFlubberCol);
delete pFlubberCol;
```

Die Kollisionshülle in Abbildung 6.11 verschwindet im unteren Teil des Körpers, da er sich zum Zeitpunkt der Bildaufnahme, in einer Animation befunden hat und sich der Kollisionskörper der Animation nicht anpasst.

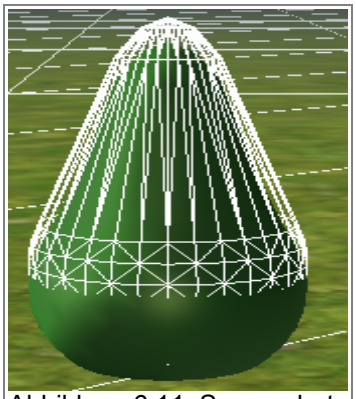

Abbildung 6.11: Screenshot - Kollisionshülle

Das Entity hat einen Größenvektor von ca. (0.75 Meter, 1.3 Meter, 0.75 Meter). Die Vektor-Komponenten werden mit einem Skaliervektor und zusätzlich noch mit dem

102 Siehe Kapitel 5.10.1

Faktor 100 multipliziert.

```
Ogre::Vector3 size = pCharEnt->getBoundingBox().getSize();
Ogre::Real mass = (size.x * scale.x) * (size.y * scale.y) *
                  (size.z * scale.z) * 100.0;
```

Daraus ergibt sich eine Masse von ca. 73.125 Kilogramm ohne Skalierung. Je nach Gegnerart, werden diese jedoch skaliert, dies impliziert für jede Gegnerart jeweils eine andere Masse. Die Form in der obigen Abbildung ähnelt einer Ellipse, daher wird die Trägheit über diese errechnet.

```
Ogre::Vector3 inertia = OgreNewt::MomentOfInertia::CalcEllipsoidSolid(
                                                        mass, size * scale);
pPhysicsBody->setMassMatrix(mass, inertia);
```

Da dieser Körper eine runde Form besitzt, bestünde die Gefahr, dass ein Charakter in der 3D-Welt bei verschiedenen Aktionen, wie zum Beispiel beim Herunterlaufen eines Berges umkippen würde. Um dies zu vermeiden bietet OgreNewt die Methode UpVektor[103] an.

```
OgreNewt::BasicJoints::UpVector *upVector = new
          OgreNewt::BasicJoints::UpVector(pOgreNewt, pPhysicsBody,
          Ogre::Vector3::UNIT_Y);
```

Anschließend werden mehrere Geräuscheffekte erstellt, die später beim „Gehen“, „Treten“ und bei einer Kampfattacke abgespielt werden sollen.
In Kapitel 5.10.1 wurde die Funktion für die Gravitationseinwirkung erläutert. Diese wird für den Charakter erstellt und eingebunden.
Des Weiteren wird eine Funktion erstellt, welche die Steigung des Bodens an der sich ein Charakter befindet, und die Distanz vom Charakter zu einem Objekt errechnet. Dies geschieht ähnlich wie bei einer Szenenabfrage in Ogre mit einem Strahl, der Unterschied dazu besteht jedoch darin, dass der Strahl in Ogre lediglich hinsichtlich eines Kontaktes mit einem Grenzquader (bounding box) eines Objektes testet. Dies ist jedoch in diesem Fall unzureichend, da einzelne Polygone der Landschaft gar nicht getroffen werden könnten. Daher besteht die Notwendigkeit, einen Strahl auf Polygonebene zu testen. OgreNewt bietet diese Möglichkeit an. Im Folgenden wird ein Strahl von der Mitte eines Charakters 500 Meter nach unten abgeschossen, dann wird ermittelt, ob ein Objekt wie beispielsweise eine Kiste oder die Landschaft

103 http://newtondynamics.com/wiki/index.php5?title=NewtonConstraintCreateUpVector, Internet: 13.07.2009

getroffen wurde. Anschließend wird die Distanz zu diesem Objekt ermittelt. Zusätzlich wird, wenn es sich um die Landschaft handelt, über ein Kreuzprodukt die Steigung berechnet.

```
Ogre::Vector3 charPoint = Ogre::Vector3(getPosition().x,
                            getPosition().y + 0.4, getPosition().z);

OgreNewt::BasicRaycast ray(pOgreNewt, charPoint, charPoint +
Ogre::Vector3::NEGATIVE_UNIT_Y * 500);
OgreNewt::BasicRaycast::BasicRaycastInfo contact = ray.getFirstHit();
if (contact.mBody)
{
    height = contact.mDistance * 500;
    if (contact.mBody->getType() == Utilities::TERRAIN)
    {
        Ogre::Vector3 vec = Ogre::Vector3::UNIT_Y;
        Ogre::Vector3 normal = contact.mNormal;
        rise = Ogre::Math::ACos(vec.dotProduct(normal) / (vec.length() *
               normal.length())).valueDegrees();
    }
}
```

Es ist darauf zu achten, dass contact.mDistanz ebenfalls mit 500 Meter verlängert wird, da OgreNewt nur im Bereich [0, 1] rechnet.

Des Weiteren wird eine ähnliche Funktion erstellt, welche einen Strahl eine bestimmte Distanz vom Charakter entfernt abschickt. Dadurch soll ermittelt werden, ob sich vor dem Charakter ein anderes Objekt befindet. Wenn dem so ist, liefert diese Funktion die Kontaktdaten dieses Objektes zurück, nachdem es getroffen wurde. Damit kann beispielsweise überprüft werden, ob sich ein Gegner in der Nähe des Spielers befindet. Diese Funktion benötigt den Richtungsvektor[104] aus der Orientierung des Charakters um einen Strahl in diese Richtung abzuschießen.

```
OgreNewt::BasicRaycast::BasicRaycastInfo Character::getContactAhead(
                                                    Ogre::Real scale)
{
   Ogre::Vector3 direction = getOrientation() * Ogre::Vector3::NEGATIVE_UNIT_Z;

   Ogre::Vector3 charPos = Ogre::Vector3(getPosition().x, getPosition().y + 0.4,
                                         getPosition().z);
   OgreNewt::BasicRaycast ray(pOgreNewt, charPos, charPos +
                                          (direction * scale));
   OgreNewt::BasicRaycast::BasicRaycastInfo contact = ray.getFirstHit();
   return contact;
}
```

104 Siehe Kapitel 5.10.3

6.7.1 Spieler

Der Spieler (Flubber) entsteht aus dem Charakter, zusätzlich besitzt er jedoch noch weitere Eigenschaften.

Im Folgenden wird die Funktionsweise des Spielers erläutert.

Damit der Benutzer die Welt abhängig von der Position und Orientierung des Spielers sehen kann, erhält der Spieler eine Kamera. Das in Kapitel 5.2 beschriebene Merkmal, welches besagt, dass ein Kindknoten relativ zum Vaterknoten bewegt und gedreht wird, wird sich hierbei zunutze gemacht. Die Kamera wird an einen Knoten angebracht, dieser Kameraknoten wird an den Spielerknoten gehängt und anschließend der Abstand gesetzt. Somit kann die Kamera relativ zum Spieler gedreht werden, beziehungsweise um den Spieler herum. Dadurch entsteht die Third-Person Kameraperspektive.[105]

```
pCamNode = pCharNode->createChildSceneNode("CamNode");
pCamNode->attachObject(pCamera);
pCamNode->setPosition(Ogre::Vector3(0.0, 1.5, 1.0));
```

Der Spieler soll über die Tastatur bewegt werden und über die Maus sollen der Spieler und die Kamera gedreht werden. Dazu kann die Bibliothek Object Oriented Input Library (OIS)[106] in Ogre eingebunden werden. Über OIS lassen sich Abfragen der gängigen Eingabegeräte steuern. Im Folgenden wird erklärt, wie die Orientierung mittels der Maus funktioniert.

Zunächst wird der aktuelle Zustand der Maus benötigt.

```
const OIS::MouseState &ms = pMouse->getMouseState();
```

Danach wird die Bewegungsgeschwindigkeit für horizontale und vertikale Bewegungen durch die Maus berechnet.

```
rotateScaleX = -ms.X.rel * rotateSpeedX * evt.timeSinceLastFrame * 60;
rotateScaleY = -ms.Y.rel * rotateSpeedY * evt.timeSinceLastFrame * 60;
```

Die Kameradrehung um die X-Achse wird dahingehend beschränkt, dass nur innerhalb eines gewissen Bereiches gedreht werden kann, ansonsten würde die Szene

105 Siehe 5.6.3

106 http://www.ogre3d.org/wiki/index.php/Using_OIS, Internet: 14.07.2009

nach einer zu weit gehenden Rotation auf dem Kopf stehen. Anschließend erhält die Kamera den Drehfaktor um die eulersche Drehung pitch() ausführen zu können.

```
pFlubberIntern->getCamera()->pitch(rotateScaleY, Ogre::Node::TS_PARENT);
```

Die Angabe Ogre::Node::TS_PARENT bewirkt, dass die Kamera um die X-Achse des Vaterknotens (Flubber) gedreht wird, statt um ihre eigene.
Die Drehung der Kamera um die Y-Achse erfolgt um ihre eigene Achse. Diese wird jedoch über OgreNewt berechnet, da auch der Spieler orientiert werden muss.[107] Dazu wird der Drehfaktor an den Charakter übergeben.

```
pFlubber->setRotationYAxis(rotateScaleX.valueDegrees());
```

Im nächsten Schritt wird die Bewegung des Spielers erläutert.
Je nach Bewegung oder Aktion muss die entsprechende Animation ausgeführt werden. Wenn der Spieler zum Beispiel nach vorne bewegt wird, muss die in Tabelle 6.1 aufgelistete WalkNorth Animation ausgeführt werden. Eine Animation braucht jedoch eine gewisse Zeit um komplett durchlaufen zu werden. Wenn aber der Benutzer die „W“-Taste gedrückt hält, so wird die Funktion um den Spieler zu bewegen in kurzen Zeitabständen ausgeführt, und die Geh-Animation würde immer wieder von vorne starten, ohne jemals beendet zu werden. Daher wird in einer Variable abgespeichert, welche Animation vorher stattfand. Hat sich die Animationsart nicht geändert, so wird die Animation nicht von neuem gestartet, sondern bis zum Ende aktualisiert. Nur wenn sich eine Animation ändert, wird eine neue gestartet.

```
if (pKeyboard->isKeyDown(OIS::KC_W))
{
    pFlubber->oldAnimAction = pFlubber->getAnimAction();
    pFlubber->setAnimAction(Utilities::WALK_NORTH);
    pFlubberIntern->move(Ogre::Vector3::NEGATIVE_UNIT_Z, pFlubber->moveScale);
}
```

Die Bewegung des Spielers vollzieht sich wie in Kapitel 5.10.1 beschrieben, jedoch nur unter der Bedingung, dass die im Abschnitt „Charakter“ berechnete Steigung nicht größer als 40° ist.
Zum Spielgenre Jump 'n' Run gehört auch, den Spieler einen Sprung ausführen zu lassen. Dies birgt jedoch ein Problem. Ein Sprung muss eingeschränkt werden, da-

107 Siehe Kapitel 5.10.1

mit der Spieler nicht zu hoch springt. Nachfolgend wird erklärt wie dies bewerkstelligt wird.

Die Ausführung eines Sprungs funktioniert ähnlich wie die Bewegung des Spielers, nur wird als Richtung die Sprunggeschwindigkeit entlang der Y-Achse angegeben. Damit der Spieler nicht zu hoch springt, wird die ermittelte Höhe aus dem Abschnitt „Charakter“ genommen, diese misst den Abstand zu einem Objekt, wie zum Beispiel des Bodens oder eines anderen Objektes, auf welchem sich der Spieler gerade befindet. Nur Wenn der Wert der Höhe kleiner als 0,8 Meter ist, wird ein Sprung unternommen. Zudem soll ein Sprung auch nur ausgeführt werden, wenn die Steigung der Landschaft nicht mehr als 40° beträgt.

Der Spieler soll zudem gegen Gegenstände treten können. Um dies zu ermöglichen wird mit Hilfe der Funktion getContactAhead() aus dem vorherigen Abschnitt ermittelt, ob sich unmittelbar vor dem Spieler ein loses Objekt befindet. Wenn das Ray den entsprechenden Objekttyp trifft und die „Return“- oder „E“-Taste gedrückt wird, dann wird eine Tret-Animation abgespielt, wenn diese sich fast am Ende befindet, wird eine Funktion durchlaufen, welche das Objekt vom Spieler weg schleudert. Abbildung 6.12 zeigt eine Tret-Animation.

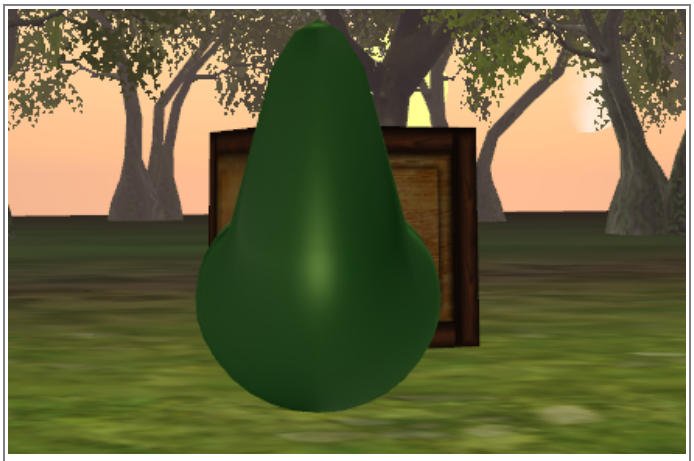

Abbildung 6.12: Screenshot - Tret-Animation

Der Spieler soll über Schluchten springen, fällt er jedoch hinein, käme er wegen der zu hohen Steigung nicht mehr heraus, daher soll er sterben. Dies wird bewerkstelligt, indem dem Spieler Energie abgezogen wird, wenn die Y-Position des Spielers 10 Meter unter Null liegt.

6.7.2 Gegner

Die Gegner stammen auch von der Charakterklasse ab. Diese werden mit Hilfe der Liste mit Entity-Informationen, wie in Kapitel 6.6 erwähnt, erstellt. Damit das Spiel nicht zu eintönig wird, werden verschiedene Gegnerarten erzeugt. Diese unterscheiden sich durch die Skalierung voneinander.

```
if(pNodeFromWorld->getScale() == Ogre::Vector3(1.1,1.0,1.1))
{

   pCharEnt->getSubEntity(2)->setMaterialName("SOLID/TEX/bodysurface2.png");
   pCharEnt->getSubEntity(1)->setMaterialName("SOLID/TEX/eyes2.png");
   speed = 4.0;
   attackMode = 2;
   strength = 4;
   energy = 50;
   experienceFactor = 14;
   opticalRange = 25.0;
}
```

Über die Funktion getSubEntity(1) erhält das Teilmodell, in welchem die Augen modelliert wurden ein bestimmtes Material. Abbildung 6.13 stellt einen braunen Gegner mit rosa gefärbten Augen dar.

Abbildung 6.13: Screenshot - Brauner Gegner

Außerdem werden verschiedene Attribute gesetzt. Tabelle 6.2 beschreibt die Attribute eines Gegners.

Attribut	Beschreibung
speed	Geschwindigkeit: Beschreibt die Bewegungsgeschwindigkeit des Gegners.
attackMode	Attackmodus: Je nach Modus wird eine Attacke anders ausgeführt.
strength	Stärke: Der Spieler bekommt abhängig von der Stärke viel Energie abgezogen.
energy	Energie: Bei einer erfolgreichen Attacke des Spielers bekommt der Gegner Energie abgezogen, wenn dieser keine mehr hat, stirbt er.
experienceFactor	Erfahrungspunkte: Wenn der Spieler den Gegner eliminiert, dann erhält er die Erfahrungspunkte dieses Gegners.
opticalRange	Optische Sichtweite: Über dieses Attribut wird festgelegt, ab welcher Distanz ein Gegner den Spieler sehen kann.

Tabelle 6.2: Gegnerattribute

6.8 Gegnerverhalten und Künstliche Intelligenz

Gegner werden ähnlich bewegt wie der Spieler und können auch über Klippen springen und Objekte treten. Im Gegensatz zum Spieler werden diese jedoch nicht vom Benutzer gesteuert, sondern vom Computer. Sie werden also vom Computer zum Leben erweckt und erhalten eine künstliche Intelligenz. Das heißt, ihr Verhalten wird von der Umwelt und vom Spieler beeinflusst. Nach [44] lässt sich die künstliche Intelligenz in Spielen über endliche Automaten beschreiben. Endliche Automaten sind abstrakte Modelle, welche sowohl bei der Hardware als auch Software Anwendung finden. Diese bestehen aus einer begrenzten Anzahl an Zuständen, welche durchlaufen werden können. Das Verhalten der Gegner in dieser Arbeit lässt sich über Zustände simulieren. Dabei ist das Verhalten abhängig von der Eingabe und dem Zustand, in dem sich der Gegner aufgrund von vorangegangenen Eingaben befindet. Das Ziel eines Gegner besteht darin, den Spieler zu eliminieren. Im Folgenden wird der Lebenslauf eines Gegners zunächst allgemein erläutert.

Gegner die sich zu weit entfernt vom Spieler befinden, sollen nicht dargestellt werden, diese werden „schlafen“ gelegt. Befinden sich Gegner im sichtbaren Bereich des Spielers, dann werden diese „aufgeweckt“. Bis dahin haben die Gegner den Spieler jedoch noch nicht bemerkt, daher sollen diese sich zufällig bewegen. Kommt der Spieler einem Gegner zu nahe, dann müsste dieser den Spieler bemerken. Der Gegner wird dann zum Spieler gedreht, er „reagiert“ quasi auf den Spieler. Damit Gegner erst angreifen, wenn sie den Spieler auch wirklich sehen, bietet sich nach [44] das Environmental Sensing-Prinzip an. Bei diesem erhalten Gegner die Fähigkeit zu „Sehen“, um Objekte, darunter auch den Spieler, wahrnehmen zu können. Das Environmental-Sensing wird in Spielen eingesetzt, damit sich die Gegner dem Spieler gegenüber nicht zu unfair verhalten. Anhand eines Beispiels wird dies verdeutlicht:

Angenommen, der Spieler befindet sich in einem Gebäude und der Gegner außerhalb, zwischen Spieler und Gegner befindet sich also eine Wand. Ohne Environmental Sensing, würde der Gegner den Spieler versuchen anzugreifen, obwohl er ihn gar nicht sehen kann. Daher erhalten Gegner ein Ray in Augenhöhe und erst wenn das Objekt, das sie sehen vom Typ „Spieler“ ist, greifen sie es an.

Spiele können nach [8, S. 499] nicht nur aus einem Zustandsautomaten, sondern aus mehreren bestehen, sie können also eine Hierarchie bilden. Dabei werden Signale von einem Automat in einen anderen gesendet. Dies ist in diesem Spiel auch

der Fall. Wenn der Gegner angreift, wird sein Zustand auf „Aktiv“ gesetzt und ab dann werden über den Angriffsautomaten weitere Zustände abgehandelt, welche den kompletten Angriffsablauf beschreiben. Abbildung 6.14 zeigt ein Zustandsdiagramm beider Automaten.

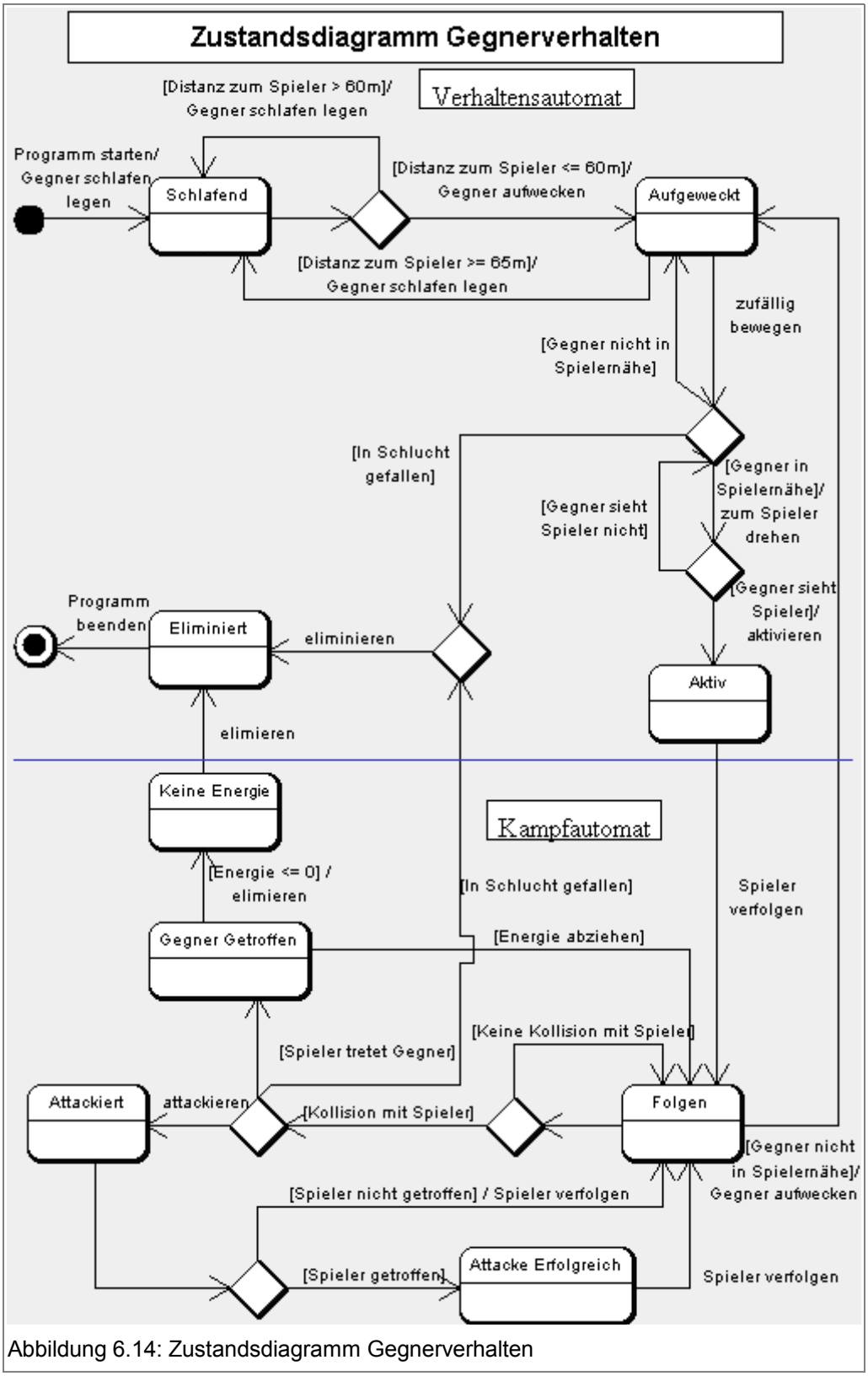

Abbildung 6.14: Zustandsdiagramm Gegnerverhalten

Nachfolgend wird über dieses Zustandsdiagramm erläutert, wie das Verhalten der Gegner realisiert werden kann.
Die Gegner erhalten vier mögliche Verhaltenszustände.

- Schlafend (Sleep)
- Aufgeweckt (Awake)
- Aktiv (Active)
- Eliminiert (Eliminated)

Es kommen zwei Kriterien in Betracht, welche den Übergang eines Zustandes in einen anderen veranlassen. Zum Einen ist die Distanz zum Spieler ein Kriterium. Nur wenn sich Gegner innerhalb einer bestimmten Distanz befinden, kann ein Zustandswechsel erfolgen. Zum Anderen ist die Wahrnehmung, also je nachdem ob der Gegner den Spieler sieht oder nicht, von Bedeutung.
Zu Beginn des Programms werden die geladenen Gegner in den Zustand „Sleep“ versetzt.

```
setCondition(Utilities::SLEEP);
```

Dadurch wird der Körper eingefroren[108] und der Szenenknoten unsichtbar geschaltet. Befindet sich ein Gegner im Umkreis von 60 Metern zum Spieler, wird er aufgeweckt. Im Folgenden wird erläutert wie dies realisiert werden kann.
Der Spieler erhält eine Sphäre[109] mit der Spielerposition als Mittelpunkt und einem Radius von 60 Metern. Innerhalb dieser Sphäre werden ständig Abfragen ausgeführt, wodurch geprüft wird, ob diese ein Objekt vom Typ „Gegner“ liefern. Ist dies der Fall, so wird dieser in eine Hashtabelle[110] zur weiteren Verwaltung des Gegnerverhaltens eingetragen. In die Hashtabelle wird als Schlüssel die Gegner-ID und als Datenobjekt ein Zeiger auf die Charakterklasse eingetragen.
Ein Gegner soll nur eingetragen werden, wenn dieser sich noch nicht in der Hashtabelle befindet, somit muss diese ständig nach der Gegner-ID durchsucht werden. Eine Hashtabelle hat den Vorteil, dass sie bei eindeutigen Schlüsseln eine Suchlaufzeit von O(1) hat, daher kommt eine solche in dieser Arbeit zum Einsatz. Wenn der Gegner eingetragen wird, wechselt er in den Zustand Awake. Dieser bewirkt, dass

108 Siehe Kapitel 5.13

109 Siehe Kapitel 5.3

110 http://de.wikipedia.org/wiki/Hashtabelle, Internet: 15.07.2009

der Körper „aufgetaut“ wird, also physikalische Abläufe stattfinden können und von Ogre sichtbar geschaltet wird.
Ohne die Maßnahme mit der Hashtabelle würde ein Gegner ständig in den Zustand Awake gesetzt werden, somit könnten andere Zustände nie erreicht werden.
Von nun an ist der Gegner von Weitem sichtbar, daher wird dieser zufällig bewegt, solange er sich nicht in der Nähe des Spielers befindet. Eine zufällige Bewegung erfolgt folgendermaßen:
Zunächst erhält der Gegner die Geh-Animation (WalkNorth), danach wird immer wieder die Zeit auf zwei Sekunden gesetzt. Wenn diese abgelaufen ist, wird eine zufällige Winkelgeschwindigkeit ermittelt und die in Kapitel 5.10.1 erwähnte Variable angularVelocity erhält diesen zufälligen Wert.

```
Ogre::Real randomVel = Ogre::Math::RangeRandom(-3.0, 3.0);
angularVelocity = randomVel;
```

Somit ändert der Gegner alle zwei Sekunden geringfügig seine Richtung. Es kann aber ein Problem entstehen, wenn der Gegner mit einem Objekt wie zum Beispiel einem Baum kollidiert, dann kann es passieren, dass er nur noch gegen diesen läuft und nicht ausweichen kann. Um dies zu vermeiden wird mittels der im Kapitel „Charakter“ beschriebenen Hilfsfunktion getContactAhead() getestet, ob sich unmittelbar vor dem Gegner ein Objekt befindet. Ist dies der Fall so wird der Gegner zufällig um 70° oder -70° in eine andere Richtung gedreht. Für die Drehung eignet sich die Winkelgeschwindigkeit allerdings nicht, da ein hoher Wert den Körper nur schneller drehen lasse würde. Daher wird der Gegner mittels eines Quaternions gedreht.

```
Ogre::Real rnd = Ogre::Math::RangeRandom(-3.0, 3.0);
Ogre::Quaternion quat = Ogre::Quaternion::IDENTITY;
if (rnd > 0)
    quat = getOrientation() * Ogre::Quaternion(Ogre::Degree(70),
                                                Ogre::Vector3::UNIT_Y);
else
    quat = getOrientation() * Ogre::Quaternion(Ogre::Degree(-70),
                                                Ogre::Vector3::UNIT_Y);

pPhysicsBody->setPositionOrientation(pCharNode->getPosition(), quat);
```

Kommt der Spieler dem Gegner zu nahe, sodass der Spieler sich in der optischen Sichtweite[111] des Gegners befindet, wird der Gegner zum Spieler gedreht.[112] Nun

111 Siehe Tabelle 6.2

112 Siehe Kapitel 5.10.3

kommt das zuvor beschriebene Environmental-Sensing ins Spiel. Es wird getestet, was der Gegner in seiner optischen Sichtweite erkennt. Wenn der Spieler sich beispielsweise hinter einem Baum versteckt, sieht der Gegner nur den Baum. Kommt der Spieler jedoch vom Baum hervor, sieht der Gegner ihn. Er wechselt in den Zustand Active. Ab diesem Punkt kommt der in Abbildung 6.14 dargestellte „Kampfautomat“ zum Zuge. Dieser enthält die Zustände:

- Folgen (Follow)
- Attackiert (Attack)
- Attacke erfolgreich (Attack_Success)
- Gegner getroffen (Hit)
- Keine Energie (No_Energy)

Zunächst erhält der Gegner den Zustand Follow, die Richtung zum Spieler wird kontinuierlich aktualisiert und der Gegner wird in diese Richtung bewegt, dadurch verfolgt er den Spieler. Sobald dieser mit ihm kollidiert, wechselt der Gegner in den Zustand Attack und beginnt eine Tret-Attacke auszuführen. Trifft der Gegner den Spieler, wechselt der Gegner in den Zustand Attack_Success und dem Spieler wird Energie abgezogen. Abbildung 6.15 zeigt eine Tret-Attacke.

Abbildung 6.15: Screenshot - Tret-Attacke

Wenn der Spieler hingegen eine erfolgreiche Tret-Attacke beim Gegner ausführt, wird der Gegner in den Zustand Hit versetzt und es wird ihm Energie abgezogen. Hat der Gegner keine Energie mehr, so erhält er den Zustand No_Energy und es wird eine Animation ausgeführt, in welcher der Gegner stirbt.

Nun wird wieder in den Verhaltensautomat gewechselt. Der Gegner erhält den Zustand Eliminated und wird aus der Hashtabelle entfernt und der Gegner komplett aus dem Spiel gelöscht. Daraufhin erhält der Spieler die Erfahrungspunkte des Gegners.[113]

Gegner sollen sich nachts aggressiver verhalten. Dies wird bewerkstelligt, indem die Uhrzeit abgefragt wird. Im Zeitraum von 0 bis 4 Uhr werden die in Tabelle 6.2 beschriebenen Gegnerattribute erhöht. Dadurch werden diese stärker. Des Weiteren erhalten sie ein anderes Material für die Augen, damit sie leuchten. In Kapitel 5.4 wurde dies als Beispiel erläutert.

6.9 Kollisionsablauf zwischen Gegner und Spieler

In diesem Abschnitt wird erläutert was genau passiert, wenn der Spieler mit dem Gegner oder der Gegner mit dem Spieler kollidiert.

Im Kapitel 5.10.2 wurde eine allgemeine Einführung in den Kollisionsablauf gegeben und zudem beschrieben, wie bei einem Kontakt zwischen zwei Körpern eigenes Verhalten eingebunden werden kann.

Zunächst wird bei Anbahnung einer Kollision in der Funktion userBegin() ermittelt, wer der Spieler und wer der Gegner ist.

```
int EnemyPlayerCallback::userBegin()
{
   if (m_body0->getType() == Utilities::PLAYER)
   {
       pFlubber = (Character*) m_body0->getUserData();
       pFlubberIntern = dynamic_cast<Flubber *>(pFlubber);
       pEnemy = (Character*) m_body1->getUserData();
       pEnemyIntern = dynamic_cast<Enemy *>(pEnemy);
   }
   else if(m_body1->getType() == Utilities::PLAYER)
   {
      pEnemy = (Character*) m_body0->getUserData();
      pEnemyIntern = dynamic_cast<Enemy *>(pEnemy);
      pFlubber = (Character*) m_body1->getUserData();
      pFlubberIntern = dynamic_cast<Flubber *>(pFlubber);
   }

   pEnemyEnergyLabel->setCaption("Enemy Energy: "
                  + Ogre::StringConverter::toString(pEnemy->getEnergy()));
   return 1;
}
```

Die Variablen m_body0 und m_body1 stammen aus der Basisklasse. OgreNewt hält in diesen fest, welche Körper von der Kollision betroffen sind. Im auf zuvor verwiese-

113 Siehe Kapitel 3.2

nen Kapitel 5.10.1 wurde bei Erzeugung eines Körpers die Funktion setUserData() aufgerufen und ein Zeiger auf die Klasse, in welcher dieser erzeugt wurde, mitgegeben. Wenn der Typ von m_body0 der Spieler ist, kann mit getUserData() in die entsprechende Klasse Charakter oder Spieler gecastet werden. Zudem wird im Label pEnemyEnergyLabel die Energie des Gegners zur Anzeige auf dem Bildschirm festgehalten.
Bei tatsächlichem Kontakt beider Körper führt OgreNewt die Funktion userProcess() aus. In dieser Funktion wird, je nachdem ob Gegner oder Spieler eine erfolgreiche Attacke ausgeführt hat, der jeweils andere weg gestoßen.

```
int EnemyPlayerCallback::userProcess()
{
   Ogre::Vector3 direction = pEnemy->getOrientation() *
                                                Ogre::Vector3::NEGATIVE_UNIT_Z;

   if (pFlubber->isInAttack && (pEnemyIntern->getAction()
            != Utilities::ATTACK_SUCCESS) && (pEnemy->getEnergy() > 0))
   {
       OgreNewt::BasicRaycast::BasicRaycastInfo contact =
                                          pFlubber->getContactAhead(0.4);
       if (contact.mBody)
       {
          if ((contact.mBody->getType() == Utilities::ENEMY))
          {
             pEnemy->getBody()->setVelocity(pEnemy->getBody()->getVelocity() *
             Ogre::Vector3::UNIT_Y + direction * pEnemy->getSpeed() * -2.0);
          }
       }
    }
    if ((pEnemyIntern->getAction() != Utilities::ATTACK)
                                                  && (pEnemy->getEnergy() > 0))
    {
       pEnemyIntern->setAction(Utilities::ATTACK);
       pFlubber->getBody()->setVelocity(pFlubber->getBody()->getVelocity() *
       Ogre::Vector3::UNIT_Y + direction * pEnemy->moveScale);

       if (!pFlubber->isInAttack)
       {
            int diffEnergy = pFlubber->getEnergy() - 1;
            pFlubberIntern->setEnergy(diffEnergy);
       }
    }
    return 1;
}
```

In der Funktion wird im ersten Schritt der Richtungsvektor aus einem Quaternion ermittelt. Danach wird überprüft, ob der Spieler sich in einer Attacke befindet, der Aktionszustand des Gegner nicht auf Attack_Success gesetzt ist und der Gegner noch Energie besitzt. Anschließend wird über die Funktion getContactAhead() ermittelt, ob sich der Gegner unmittelbar vor dem Spieler befindet und die Vorderseite des Spie-

lers in Richtung des Gegners zeigt. Diese Maßnahme ist notwendig, um zu verhindern, dass der Spieler bei einer Kollision mit dem Gegner diesen treffen würde, obwohl er an ihm vorbei getreten hat, beziehungsweise die Tret-Attacke nicht in Richtung des Gegner ausführte.
Anschließend wird der Gegner über die Funktion setVelocity() in die entgegen gesetzte Richtung vom Spieler weg gestoßen.
Wenn der Gegner bei einer Kollision noch nicht den Zustand Attack besitzt, wird dieser nur dann gesetzt, wenn er noch Energie hat. Danach wird der Spieler vom Gegner abgestoßen. Zudem bekommt der Spieler bei bloßer Berührung mit dem Gegner ein wenig Energie abgezogen, wenn der Spieler nicht gerade attackiert.
In der letzten Funktion userEnd() bekommt man eine letzte Chance auf eine Kollision zu reagieren.

```
void EnemyPlayerCallback::userEnd()
{
  if (pFlubber->isInAttack && ((pEnemy->getAction() != Utilities::NO_ENERGY) ||
                      (pEnemyIntern->getAction() != Utilities::ATTACK_SUCCESS)))
  {
    OgreNewt::BasicRaycast::BasicRaycastInfo contact =
                                                pFlubber->getContactAhead(0.4);
    if (contact.mBody)
    {
      if (contact.mBody->getType() == Utilities::ENEMY)
      {
        pEnemyIntern->setAction(Utilities::HIT);
        pFlubberIntern->pCharHit1Sound->play();
        int diffEnergy = pEnemy->getEnergy() - pFlubber->getStrength();
        pEnemy->setEnergy(diffEnergy);
        pEnemyEnergyLabel->setCaption("Enemy Energy: " +
                        Ogre::StringConverter::toString(pEnemy->getEnergy()));
        if (pEnemy->getEnergy() <= 0)
        {
          pFlubberIntern->pCharDeathSound->play();
          pEnemyEnergyLabel->setCaption("");
          pEnemyIntern->setAction(Utilities::NO_ENERGY);
          pFlubberIntern->setExperience(pEnemyIntern->getExperienceFactor());
        }
      }
    }
  }
  if ((pEnemyIntern->getAction() == Utilities::ATTACK_SUCCESS) &&
                                                    (pEnemy->getEnergy() > 0))
  {
     pFlubber->oldAnimAction = pFlubber->getAnimAction();
     pFlubber->setAnimAction(Utilities::HURT2);
     int diffEnergy = pFlubber->getEnergy() - pEnemy->getStrength();
     pFlubberIntern->setEnergy(diffEnergy);
     pFlubberIntern->pCharHit1Sound->play();
  }
}
```

Wenn der Spieler attackiert, der Zustand des Gegners nicht auf No_Energy oder Attack_Success gesetzt ist, geht der Gegner in den Zustand Hit über. Anschließend wird ein entsprechender Geräuscheffekt abgespielt, das Label zur Energieanzeige aktualisiert und dem Gegner soviel Energie abgezogen, wie der Spieler an Stärke besitzt. Hat der Gegner hingegen keine Energie mehr, soll ebenfalls ein entsprechender Geräuscheffekt abgespielt und die Energieanzeige geleert werden. Außerdem erhält der Gegner den Zustand No_Energy und der Spieler erhält die Erfahrungspunkte des Gegners.
In der zweiten Bedingung des Codeausschnittes bekommt der Spieler so viel Energie abgezogen, wie der Gegner an Stärke besitzt.

6.10 Spiel Speichern

Spielstände werden bei Spielen benötigt, die von der Zeit und dem Schwierigkeitsgrad her nicht direkt durchgespielt werden können. Es wäre für den Benutzer frustrierend, zu sterben, nachdem er das Spiel schon weitestgehend bewältigt hat und wieder ganz von vorne beginnen müsste. Deswegen werden Spielstände eingeführt. Betritt der Spieler eine Speicherstelle, so werden in eine Textdatei alle notwendigen Informationen wie zum Beispiel

- Name der Welt
- Energie, Stärke, Erfahrung, Leben des Spielers
- Tag und Uhrzeit
- Position und Orientierung

gespeichert. Des Weiteren wird ein Partikeleffekt eingebunden, welcher bei der ersten Kollision mit dem Boden der Speicherstelle ausgeführt wird.
Wenn der Spieler später stirbt, werden diese Daten geladen und der Spieler kommt an den zuletzt besuchten Speicherstand. Speicherstände werden in der Welt so platziert, dass der Benutzer bis zum Erreichen der nächsten Speicherstelle einige Hürden meistern muss.

7 Fazit

In so einer kurzen Zeit kann kein komplettes Spiel entstehen. Die Beschreibung der Realisierung konnte aus Platzgründen nicht in allen Einzelheiten erfolgen. Alleine schon das Testen des Spiels vollzieht sich über einen längeren Zeitraum. Während in einer Spiele-Firma die einzelnen Aufgaben auf mehrere Personen aufgeteilt werden, nahm der Autor alle Aufgaben selbst wahr.

7.1 Abschlussbewertung

Es konnten alle geplanten Schritte erreicht werden. Die Modellierung, Texturierung und Animation der Charaktere hat zwar viel Zeit in Anspruch genommen, funktioniert aber nun reibungslos. Durch die gesammelte Erfahrung in der Modellierung konnten weitere Modelle erfolgreich erstellt werden.

Das Einbinden und Arbeiten mit der Physik-Engine, konnte nach kurzer Zeit erlernt und erfolgreich für den physikalischen Teil eingesetzt werden.

Beim Laden der Welt mussten Objekte anhand des Mesh-Namens und des Skalierfaktors unterschieden werden, da der Leveleditor keine Benutzerdefinierten Informationen zu einem Objekt bereitstellt.

Die Größe der Welt wurde zu groß gewählt. In dieser Zeitspanne konnte die Welt somit nicht komplett editiert werden. Rückblickend hätte eine Welt von einem Kilometer² ausgereicht. Außerdem ist aufgefallen, das einige Objekte in der Welt ein wenig über dem Boden schweben.

Das Spiel wurde so realisiert, dass sich ein großer Ausschnitt der Welt im Blickfeld des Benutzers befindet, daher müssen viele Objekte gerendert werden. Es existieren Rollenspiele, bei denen man nur von oben auf den Spieler schaut und dadurch nur einen kleinen Abschnitt der Umgebung sieht. Dies ist in diesem Spielgenre nicht der Fall, daher wurden viele Maßnahmen getroffen um Laufzeit zu sparen.

Die Steuerung des Spielers funktioniert weitestgehend problemlos. Bei der Tret-Attacke muss die entsprechende Taste jedoch länger gehalten werden, damit dem Gegner Schaden zugefügt wird. Da ein Schadenserfolg erst fast zum Schluss der Attacke zu verzeichnen ist, wird die Tretattacke ein weiteres Mal ausgeführt, wenn der Spieler nicht die Taste unmittelbar nach einem Schadenserfolg loslässt. Die Bewegung des Spielers innerhalb der Welt konnte erfolgreich bewerkstelligt werden.

Das Verhalten der Gegner zum Spieler läuft mit wenigen Einschränkungen nahtlos ab. Je nach Computer meldet das Spiel im seltenen Fall nach unbestimmter Zeit einen Laufzeitfehler, welcher bisher durch keine Maßnahme ausfindet gemacht werden konnte.
Ein großes aber allgemeines Problem bei der Spieleentwicklung ist, dass Spiele auf unterschiedlichen Rechnern laufen müssen. Die Berechnung der Bilder Pro Sekunde ist abhängig von der Leistungsstärke eines Computers und somit auch der komplette Spielablauf. Glücklicherweise bietet die Ogre-Engine die Einstellung VSync um ein Spiel auf ca. 60 Fps zu begrenzen. Zudem sind die Grafikkarten von Rechner zu Rechner verschieden. Das heißt, die Darstellung von Grafiken ist von der Art der Grafikkarte abhängig.
Das Verhalten und die Aktionen der Gegner wurden so ausbalanciert, dass das Spiel nicht zu leicht zu meistern ist. Dadurch entsteht der Reiz, sich damit länger zu beschäftigen. Eine Einschränkung besteht darin, dass ein Gegner nur einen Sichtpunkt statt eine Sichtregion erhalten hat, da über OgreNewt nur ein Basicraycast bereitgestellt wird. Es wäre zu aufwendig gewesen, einen Sichtbarkeitsbereich zu implementieren. Das ist jedoch nicht weiter tragisch, so sind die Gegner in ihrer Sicht eingeschränkt und der Spieler kann sich besser vor ihnen verstecken. Ein weiteres Problem ist, dass die Gegner den Spieler nicht folgen können, wenn dieser um bestimmte Hindernisse herumläuft. Hierzu wäre ein geeigneter Algorithmus notwendig, damit die Gegner unter fast jeden Umstand den Spieler verfolgen könnten.
Die Berechnung des Tagesablaufs funktioniert reibungslos.
Das Speichern und Laden von Spielständen konnte zufriedenstellend implementiert werden.

7.2 Arbeitsaufwand

Die Entwicklung für das Spiel "Flubbers" begann am 23.03.2009 und endete am 15.06.2009. Im Folgenden wird eine Analyse des Arbeitsaufwands in mehreren Tabellen veranschaulicht. Dazu wird der Zeitaufwand für die einzelnen Aufgaben geschätzt. Außerdem wird der Umfang des Projektes (Anzahl der Ressourcen, Lines of Code, etc.) angegeben. Tabelle 7.1 listet die Anzahl der verwendeten Ressourcen auf:

Anzahl Modelle	Anzahl Partikeleffekte	Anzahl Audioquellen
57	4	14

Tabelle 7.1: Anzahl verwendeter Ressourcen

Von den 57 Modellen wurden 39 eigenhändig erstellt. Die restlichen Modelle, welche frei verfügbar für kommerzielle Zwecke sind, wurden aus [49] und [50] entnommen. Aus den 14 Audioquellen wurden 4 mit dem Mikrofon aufgenommen, die übrigen stammen aus [51] und [52]. Davon sind die Geräuscheffekte frei für kommerzielle Zwecke, während die Musikstücke nur für nicht kommerzielle Zwecke frei verfügbar sind. Tabelle 7.2 stellt die Quellcodezeilen (Lines of Code) dar:

Anzahl Codezeilen	Anzahl Kommentarzeilen	Anzahl Leerzeilen	Insgesamt
5069	1430	988	7487

Tabelle 7.2: Anzahl an Codezeilen

Zuletzt wird der Zeitaufwand für alle Aufgaben geschätzt und in Tabelle 7.3 aufgelistet:

Planung	10 Stunden
Erstellung der Ressourcen	150 Stunden
Erstellung der 3D-Welt	110 Stunden
Implementierung	210 Stunden
Insgesamt	480 Stunden

Tabelle 7.3: Zeitaufwand für das Spiel "Flubbers"

Somit betrug der geschätzte Zeitaufwand für dieses Projekt ca. 480 Stunden.

Im Großen und Ganzen ist der Autor sehr zufrieden mit dieser Arbeit. Es beinhaltete ein großes Risiko, da dem Autor bewusst war, wie schwierig es, ist ein 3D-Spiel mit physikalischen Effekten zu entwickeln. Es konnten wertvolle Erfahrungen gesammelt werden, wie aus einer Idee ein komplettes Spiel in einer 3D-Welt realisiert wird. Die vielen Tutorials auf den jeweiligen Seiten der Grafik-Engine und Physik-Engine haben dem Autor beträchtlich bei der Entwicklung des Spiels geholfen. Des Weiteren

bekam der Autor Einblick in den Ablauf physikalischer Berechnungen. Dem Autor wurde klar, warum die Entwicklungszeit eines Spiels in Firmen oft Jahre beträgt, da er quasi „am eigenen Leibe“ gemerkt hat, dass alleine die Umsetzung von „Flubbers“ sehr aufwendig war und zudem noch viele Ideen aus zeitlichen Gründen nicht umgesetzt werden konnten.

7.3 Ausblick

Die allgemeinen und grafischen Einstellungen des Spiels lassen sich nur sehr umständlich ändern. Hierzu wäre die Entwicklung einer geeigneten grafischen Benutzeroberfläche ein wichtiger Punkt, in welcher der Benutzer alle Einstellungen anhand der Leistung seines Computers individuell anpassen kann. Das entwickelte Spiel bildet durch die physikalischen Abläufe, die Performance und die Künstliche Intelligenz eine Basis für Erweiterungen jeglicher Art. Es kämen noch weitere Modelle in Betracht, mit welchen die Welt bereichert werden könnte. Alle Gegner bestehen aus ein und demselben Körper, deswegen würde es sich anbieten weitere Gegnerarten zu modellieren und zu animieren. Im Spielverlauf kann der Spieler sich schnell verlaufen, daher wäre eine Karte eine schöne Erweiterung, in welcher alle wichtigen Stellen sowie die Spielerposition verzeichnet sind.

Das Spiel ist noch kein vollständiges Rollenspiel. Es fehlen noch Kampfarten, wie zum Beispiel Gegner mit einem Schwert zu schlagen, sowie Aufträge, welche der Spieler erhält und meistern muss. Des Weiteren fehlt die Möglichkeit, Gegenstände aufzuheben, sie ins Inventar des Spielers zu legen, sich mit Ihnen auszurüsten oder sie zu verkaufen.

Das Spiel wurde nur für das Betriebssystem Windows entwickelt, durch die gewählten plattformunabhängigen Werkzeuge könnte es jedoch auch auf andere Betriebssysteme angepasst werden.

Anhang A

Abkürzungsverzeichnis

API:	Application Programming Interface
CPU:	Central Processing Unit
ESA:	The Entertainment Software Association
ETM:	Editable Terrain Manager
FPS:	Frames Per Second
GPU:	Graphics Processing Unit
LOD:	Level Of Detail
OGRE:	Object-Oriented Graphics Rendering Engine
OIS:	Object Oriented Input Library
LGPL:	GNU Lesser General Public License
OpenGL:	Open Graphics Library
VWE:	Virtual World Editor
VSYNC:	Vertical Synchronisation
WU:	World Unit

Quellenverzeichnis

[1] Computerspiel: Internet 17.06.2009
http://de.wikipedia.org/wiki/Computerspiel

[2] Geschichte der Videospiele: Internet 17.06.2009
http://de.wikipedia.org/wiki/Geschichte_der_Videospiele

[3] OXO (Spiel): Internet 17.06.2009
http://de.wikipedia.org/wiki/OXO_(Spiel):

[4] The First Video Game?: Internet 17.06.2009
http://www.bnl.gov/bnlweb/history/higinbotham.asp

[5] PONG-Story – Atari Pong – Home systems: Internet 18.06.2009
http://www.pong-story.com/atpong2.htm

[6] Pac-Man: Internet 19.06.2009
http://pac-man.classicgaming.gamespy.com

[7] Spieletests: Internet 19.06.2009
http://www.spieleratgeber-nrw.de/?siteid=3

[8] Adams Ernest/Rollings Andrew: Fundamentals of game design, Upper Saddle River, 07458 New Jersey, Pearson Educations, Inc 2007

[9] Magdans Frank: Game Generations, Universitätsstr. 55, 35037 Marburg, Schüren Verlag 2008

[10] Junker, Gregory: Pro OGRE 3D Programming, 1. Auflage, 233 Spring Street 6th Floor, New York, NY 10013, Springer-Verlag

[11] Eberly H. David: 3D Game Engine Design, 2. Auflage, 500 Sansome Street, Suite 400, San Francisco, CA 94111

[12] OGRE - Open Source 3D Graphics Engine: Internet 22.06.2009
http://www.ogre3d.org

[13] Bies Daniel, Kalinowski Lukas: Virtual World Editor, Fachprojektbericht Sommersemester 2008, Postfach 1380, 55761 Birkenfeld

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

[15] Bies Daniel, Kalinowski Lukas: Virtual World Editor, Medienpraxis Projektbericht Wintersemester 2008/09, Postfach 1380, 55761 Birkenfeld

[16] Grafik-Engine: Internet 28.06.2009
http://de.wikipedia.org/wiki/Grafik-Engine

[17] DirectX: Internet 28.06.2009
http://de.wikipedia.org/wiki/DirectX

[18] Audacity: Freier Audioeditor und Rekorder: Internet 28.06.2009
http://audacity.sourceforge.net/?lang=de

[19] Bildsynthese: Internet 29.06.2009
http://de.wikipedia.org/wiki/Bildsynthese

[20] Bender Michael, Brill Manfred: Computergrafik, Carl Hanser Verlag München Wien 2006, 2. Auflage

[21] OpenGL Overview: Internet 29.06.2009
http://www.opengl.org/about/overview

[22] Wie Ideen entstehen - Kreativität, Ideenfindung und Innovation: Internet 24.06.2009
http://www.wie-ideen-entstehen.de

[23] Ian Millington: Game physics engine develpment, 500 Sansome Street, Suite 400, San Francisco, CA 94111, Morgan Kaufmann 2007

[24] 2008, SALES, DEMOGRAPHIC AND USAGE DATA: Internet 02.07.2009
http://www.theesa.com/facts/pdfs/ESA_EF_2008.pdf

[25] Spielindustrie: Hollywood am Apparat: Internet 02.07.2009
http://www.spiegel.de/netzwelt/tech/0,1518,208698,00.html

[26] The Entertainment Software Association – Sales & Genre Data: Internet 02.07.2009
http://www.theesa.com/facts/salesandgenre.asp

[27] Physik-Engine: Internet 02.07.2009
http://de.wikipedia.org/wiki/Physik-Engine

[28] Wrappers: Internet 02.07.2009
http://www.ogre3d.org/wiki/index.php/Wrappers

[29] Audio: Internet 03.07.2009
de.wikipedia.org/wiki/Audio

[30] OpenAL: Internet 03.07.2009
http://connect.creativelabs.com/openal/default.aspx

[31] OgreAL: Internet 03.07.2009
http://www.ogre3d.org/wiki/index.php/OgreAL

[32] 3rd person camera system tutorial: Internet 26.06.2009
http://www.ogre3d.org/wiki/index.php/3rd_person_camera_system_tutorial

[33] OGRE Exporters: Internet 03.07.2009
http://www.ogre3d.org/wiki/index.php/OGRE_Exporters

[34] Material: Internet 04.07.2009
http://www.ogre3d.org/wiki/index.php/Material

[35] ETM: Internet 06.07.2009
http://www.ogre3d.org/wiki/index.php/ETM

[36] ETM: Internet 06.07.2009
http://www.oddbeat.de/wiki/etm

[37] Caelum: Internet 06.07.2009
http://www.ogre3d.org/wiki/index.php/Caelum

[38] Collision primitives: Internet 06.07.2009
http://newtondynamics.com/wiki/index.php5?title=Collision_primitives

[39] Newton Game Dynamics: Internet 06.07.2009
http://www.ogre3d.org/wiki/index.php/Newton_Game_Dynamics

[40] Collision detection with Newton: Internet 07.07.2009
http://www.ogre3d.org/wiki/index.php/Collision_detection_with_Newton

[41] Basic Tutorial 4: Internet 10.07.2009
www.ogre3d.org/wiki/index.php/Basic_Tutorial_4

[42] NewtonBodySetContinuousCollisionMode: Internet 12.07.2009
http://www.newtondynamics.com/wiki/index.php5?title=NewtonBodySetContinuousCollisionMode

[43] Carsten Wartmann: Das Blender-Buch, Ringstraße 19, 69115 Heidelberg, 2007 dpunkt.verlag GmbH, 3 Auflage

[44] Building Brains Into Your Games: Internet 14.07.2009
http://www.gamedev.net/reference/articles/article574.asp

[45] Super Mario 64: Internet 04.08.2009
http://de.wikipedia.org/wiki/Super_Mario_64

[46] GamePro: Internet 04.08.2009
http://www.gamepro.de/_misc/galleries/detail.cfm?pk=47509&fk=999691

[47] GOTHIC 3: Internet 04.08.2009
http://www.gothic3.com/

[48] Screenshot aus Gothic 3: Internet 04.08.2009
http://de.gothic3-game.de/components/com_zoom/www/view.php?popup=1&q={obfs:225227208219224263275286227215212265217223203263274275286227215212265219209259224215219214263286227215212265220219208263275286227215212265207219224263276}

[49] Artifex Terra 3D: Internet 05.08.2009
http://www.artifexterra3d.com/forum/ogre-model-mesh-store-dragonsnail/free-model-pack-download/

[50] PagedGeometry Engine: Internet: 05.08.2009
http://www.ogre3d.org/wiki/index.php/PagedGeometry_Engine
[51] Freesound: Internet 05.08.2009
http://www.freesound.org/
[52] The Midi Shrine - Game Music MIDI files: Internet 05.08.2009
http://www.midishrine.com/