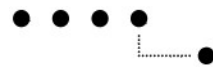


# RENDERING VIRTUELLER WELTEN MIT SHADEAM



## Bachelor-Thesis



**Berner Fachhochschule**  
Technik und Informatik

Betreuer: Urs Künzler  
Experte: Peter Matti  
Autoren: Patrick Joos, Eduardo Hahn Paredes  
Version: 1.0  
Datum: Januar 2013

# Inhaltsverzeichnis

1	Einleitung.....	5
1.1	Zweck dieses Dokuments.....	5
1.2	Aufgabenstellung.....	5
1.3	Projektübersicht.....	5
2	Game-Design.....	7
2.1	Beschreibung.....	7
2.2	Arbeitstitel.....	7
2.3	Genre.....	7
2.4	Geschichte.....	7
2.4.1	Die Vorgeschichte.....	7
2.4.2	Die Haupthandlung.....	8
2.5	Gameplay und Graphische Gestaltung.....	8
2.6	Spielelemente.....	9
2.7	User-Interface (Eingabe).....	9
2.8	Zielpublikum.....	10
2.9	Plattform und Vertriebskanal.....	10
3	Engine-Design.....	11
3.1	Übersicht.....	11
3.2	Module.....	11
3.2.1	Basis- und Grafik-Modul.....	11
3.2.1.1	Scene-Graph.....	13
3.2.2	Input-Modul.....	14
3.2.3	Audio-Modul.....	15
3.2.3.1	Audio aus dem File-System laden.....	15
3.2.3.2	3D-Sound.....	16
3.2.4	Physik-Modul.....	16
3.2.4.1	Optimierungen.....	17
3.2.4.2	Kollisionserkennung.....	18
3.2.5	GUI-Modul.....	20
3.2.6	Logik-Modul.....	20
3.2.6.1	Triggers.....	20
3.2.6.2	Conditions.....	21
3.3	Komponenten.....	21
3.4	Event-System.....	22
3.5	Level-Parsing.....	23
3.5.1	Kommentare.....	23
3.5.2	Abschnitte.....	23
3.5.2.1	Properties.....	24
3.5.2.2	Logic.....	24
3.5.2.3	Free.....	25
3.5.2.4	Objects.....	25
3.6	Plattformunabhängigkeit.....	28
3.6.1	End-User.....	28
3.6.2	Entwickler.....	28
4	Implementation der Shader.....	29
4.1	Shader-Management in OGRE.....	29
4.1.1	Erstellen der Shader.....	29
4.1.2	Parameterübergabe an die Shader.....	29
4.1.2.1	Spezialfall Texturen.....	30
4.1.3	Abstufungen.....	31
4.2	Erde-Shader.....	31
4.2.1	Basic.....	31

4.2.2 Intermediate.....	32
4.2.3 Advanced.....	35
4.3 Feuer-Shader.....	36
4.3.1 Basic.....	36
4.3.2 Intermediate.....	37
4.3.3 Advanced.....	38
4.4 Luft-Shader.....	38
4.4.1 Basic.....	38
4.4.2 Intermediate.....	39
4.4.3 Advanced.....	42
4.5 Wasser-Shader.....	43
4.5.1 Basic.....	43
4.5.2 Intermediate.....	44
4.5.3 Advanced.....	47
4.6 Rotation-Shader.....	47
5 Testing.....	48
5.1 Use-Cases Spiel.....	48
5.2 Shader-Testing.....	48
5.3 Log-Files.....	48
6 Projektmanagement.....	49
6.1 Erfüllung der Anforderungen.....	49
6.2 Zeitplan.....	50
6.3 Zusammenarbeit und Tools.....	50
7 Fazit.....	51
7.1 Ausblick.....	51
7.1.1 Mermelin.....	51
7.1.2 Cotopaxi-Engine.....	52
8 Quellen und Verzeichnisse.....	53
8.1 Literatur.....	53
8.1.1 OpenGL und Shader.....	53
8.1.2 Engine-Design und OGRE .....	54
8.1.3 Mathematik und Game-Design Planung.....	54
8.2 Web-Ressourcen.....	55
8.2.1 OpenGL.....	55
8.3 Abbildungen.....	56
8.4 Tabellen.....	56
8.5 Code-Texte.....	57
9 Anhang.....	58
9.1 Grundlagen.....	58
9.1.1 OpenGL.....	58
9.1.2 Mathematische Begriffe in OpenGL.....	58
9.1.2.1 Koordinatensysteme.....	59
9.1.2.1.1 Object Space und Model Matrix.....	60
9.1.2.1.2 World-Space und View Matrix.....	60
9.1.2.1.3 Camera Space und Projection Matrix.....	60
9.1.2.1.4 Ablauf.....	61
9.1.2.2 Model-View Matrix.....	61
9.1.2.3 Projection-Matrix.....	62
9.1.3 Shader.....	62
9.1.4 Die Rendering-Pipeline.....	63
9.1.5 GLSL Sprachmerkmale.....	64
9.1.5.1 Primitive Datentypen.....	64
9.1.5.2 Arrays.....	64
9.1.5.3 Matrizen.....	65
9.1.5.4 Kennzeichnungen.....	65

9.1.5.5 Operatoren.....	65
9.1.5.6 Integrierte Typen und Funktionen.....	66
9.1.6 GLSL-Shader.....	66
9.1.6.1 Vertex-Shader.....	67
9.1.6.2 Tessellation-Shaders.....	68
9.1.6.2.1 Tessellation Control Shader.....	68
9.1.6.2.2 Primitiven-Generierung.....	68
9.1.6.2.3 Tessellation Evaluation Shader.....	68
9.1.6.3 Geometry-Shader.....	69
9.1.6.4 Adjazenz-Primitiven.....	70
9.1.6.5 Funktionen.....	70
9.1.6.6 Fragment-Shader.....	71
9.1.6.7 Compute-Shader.....	71

# 1 Einleitung

---

## 1.1 Zweck dieses Dokuments

Dieses Dokument ist die Dokumentation der Bachelor-Thesis (Moduls 7321). Es richtet sich an den Experten (Peter Matti), als auch den Betreuer (Urs Künzler) der erwähnten Arbeit und ist dementsprechend verfasst. Das Resultat der Bachelor-Thesis ist ein einfaches 3D-Computer-Spiel bei dem Shader verwendet werden. Der nachfolgende Text soll anhand von ausgewählten Beispielen zeigen, wie das Spiel realisiert wurde. Eine vollumfängliche Code-Dokumentation ist in den Anhängen enthalten.

## 1.2 Aufgabenstellung

Die Anforderungen an die Bachelor-Thesis und somit auch an das Spiel sind im, sich im Anhang befindenden, Pflichtenheft detailliert beschrieben.

## 1.3 Projektübersicht

Als Basis für diese Arbeit gilt das Projekt 2, welches im vorausgegangenen Semester erstellt wurde. Das ursprüngliche Ziel von Projekt 2 war die Evaluation einer Spiele-Engine, welche in der folgenden Thesis verwendet werden sollte. Während dem Projekt 2 wurde aber festgestellt, dass keine der evaluierten Engines die nötigen Anforderungen erfüllte. Somit wurde der Entschluss gefasst eine eigene Engine, basierend auf der Grafik-Engine OGRE, zu erstellen.

Das neue Ziel war dabei einen Prototyp eines bekannten Spiel zu erstellen (Frogger), basierend auf der neu erstellten Engine. Jedoch ist das stellte sich das Entwickeln einer Spiele-Engine als ein grösserer Aufwand heraus, als zunächst angenommen wurde. Das Grundgerüst der Engine wurde schliesslich erfolgreich fertiggestellt.



Abb. 1: Endstand des Projekts 2

Für die Thesis wurde der Entschluss gefasst ein neues Spiel zu beginnen, welches im Rahmen des Machbaren liegen sollte. Als Basis wurde das bekannte Spielzeug Kugel-Labyrinth<sup>1</sup> auserkoren.



*Abb. 2: Kugellabyrinth*

Das Ziel war weiterhin die Arbeit an den Shadern, sowie die Erweiterung der Engine selbst.

---

<sup>1</sup> Die verwendete Abbildung ist dem Waldorfshop entnommen (<http://waldorfshop.eu/spielen-1/spiele/kugellabyrinth-gross.html>)

## 2 Game-Design

---

Es ist wie Alchemie. Man mixt ein altbekanntes Spielzeug mit spannenden Elementen und neuster Grafiktechnologie und erhält eine völlig neue Spielerfahrung. Diese Mixtur ergibt Mermelin – das Kugelabenteuer.

### 2.1 Beschreibung

In Mermelin steuert der Spieler eine Kugel durch ein mittelalterliches Labyrinth. Dabei gilt es Rätsel zu lösen, Fallen zu überwinden und die Kugel sicher auf die nächste Ebene zu befördern. Hilfe erhält der Spieler dabei durch die 4 Elemente der Alchemie, Feuer, Erde, Wasser, Luft.

### 2.2 Arbeitstitel

Mermelin, ein Kunstbegriff aus dem Namen des Zauberers Merlin sowie dem Bern-deutschen Ausdruck für Murmel („Märmeli“).

### 2.3 Genre

Das Spiel ist als Physik-Puzzler ausgelegt, sowie als Geschicklichkeitsspiel. Es basiert dabei auf dem bekannten Kinderspielzeug Kugellabyrinth.

Bekannte Physik-Puzzle Spiele:

- The Ball<sup>2</sup>
- Super Monkey Ball<sup>3</sup>

### 2.4 Geschichte

Die Geschichte des Spiels soll eine möglichst einfache Rahmenhandlung darstellen und den Spieler schnell in die gezeigte Welt eintauchen lassen.

#### 2.4.1 Die Vorgeschichte

Ein schreckliches Unglück ereilt das alte Königreich Bolaria. Ein Grauen aus alter Zeit hat seine Fesseln gesprengt und sucht nun aus der Tiefe seine Verlies nach Macht. Um das Böse zu bekämpfen, orderte König Shperio seinen höchsten Magier in das Verlies zu steigen und das Biest wieder in seinem tiefen Loch zu versiegeln.

Mutig stieg der Magier in das Verlies. Nach langer Zeit spürte man im ganzen Königreich ein Zittern, ein Beben gar. Niemand wusste was geschehen war, doch schien sich das Böse wieder aus dem Königreich zurückzuziehen. Das Volk jubelte, sammelte sich am Eingang des Verlies um den Zauberer gebührend zu empfangen. Niemand sah ihn jedoch je das Verlies verlassen.

Auf einer entfernten Heide hütete ein Junge seine Schafe. Die Schafe verhielten sich komisch, sie frassen nichts und schienen nur das Königsschloss in der Ferne zu beachten. Als das Beben kam, schlief der Junge. Aufgeschreckt stand er auf, blickte nervös um sich. Zunächst schien alles wie gewohnt. Auch die Schafe verhielten sich wieder normal.

Nervös begann der Junge seine Umgebung zu erkunden. Da fand er etwas was ihn zutiefst verwirrte: Eine perfekte Kugel, einer Perle gleich. Doch die Kugel tat etwas, was man normalerweise nicht von einer Kugel erwartet. Sie begann zu sprechen!

„Ich bin der grosse Zauberer des Königs, dreckiger Lämmel! Bring mich zum Schloss, auf der Stelle!“

---

<sup>2</sup> Homepage The Ball: <http://www.theballthegame.com>

<sup>3</sup> Homepage Super Monkey Ball: <http://www.sega.com/games/super-monkey-ball>



## 2.4.2 Die Haupthandlung

Während des Spiels löst der Spieler dann immer wie schwierigere Levels bei denen er immer weitere Elemente zur Verfügung hat. Der Zauberer, der sich als ein freches Frettchen entpuppt gibt dem Jungen meist widerwillig sein Wissen weiter. Am vermeintlichen Ende des Spiels kann der Zauberer aus der Kugel befreit werden. Es stellt sich aber heraus, wie sich durch gewisse Anspielungen während des Spiels schon erahnen lässt, dass der Zauberer böse geworden ist und die Herrschaft über Bolaria beansprucht. Es kommt zum finalen Kampf zwischen dem Jungen und dem Zauberer, wobei der Junge den Zauberer besiegt und zum neuen Hofzauberer wird.

## 2.5 Gameplay und Graphische Gestaltung

Aufgrund des mystischen Hintergrunds des Spiels, sowie dem Labyrinth-Aufbau der Levels bietet sich das Setting eines Dungeon-Crawlers an. Dieses Genre ist heute noch bei Rollenspielen, sowohl digitalen als auch solche, welche noch mit Stift und Papier gespielt werden, sehr beliebt.



Abb. 3: D&D Dungeon Kacheln

Beispiele bekannter Dungeon-Crawler Spiele:

- Dungeons & Dragons (Brett/Pen&Paper Spiel)<sup>4</sup>
- Gauntlet<sup>5</sup>
- Legend of Grimrock<sup>6</sup>

Da sich das Spiel meist in einem Labyrinth oder Dungeon abspielt, ist die graphische Gestaltung schnell ersichtlich. Mauern und Böden werden aus Stein sein, mit Auflockerungen im Level-Design wie Flüssen, Feuer, natürliche Höhlen und ähnlichen Materialien die man in einem Verlies erwartet.

Das Spiel wird zur besseren Übersicht für den Spieler immer aus einer isometrischen Vogelperspektive dargestellt, welche der Spieler frei um die Kugel rotieren kann. Dies hat zur Folge, dass keine Decke erstellt werden muss.

Dabei findet das Spiel nicht immer nur auf einer Ebene statt. Es ist vorgesehen um Levels mit mehreren Ebenen zu erstellen. Dies ermöglicht neue Rätsel. So können Ereignisse auf einer Ebene eine andere beeinflussen (z.B. Die Entstehung eines Wasserfalls).

Die 4 Elemente aus der Lehre der Alchemie haben eine Hauptrolle in diesem Spiel. Um ihre Bedeutung im Spiel zu untermauern, sowie um dem Spieler visuelle Hinweise zu geben, werden die Symbole der

<sup>4</sup> Dungeons&Dragons Online Portal: <http://www.wizards.com/dnd/>; 08.01.2013, 17:30

<sup>5</sup> Wikipedia-Eintrag zu Gauntlet: [http://de.wikipedia.org/wiki/Gauntlet\\_%28Computerspiel%29](http://de.wikipedia.org/wiki/Gauntlet_%28Computerspiel%29); 08.01.2013, 17:30

<sup>6</sup> Homepage zu „Legend of Grimrock“: <http://www.grimrock.net/>; 09.01.2013, 23:45



Altphilosophen verwendet.



Abb. 4: 1. Feuer 2. Erde 3. Wasser 4. Luft

## 2.6 Spielelemente

Für das Spiel sind diverse Hindernisse geplant, welche je nach Element eine andere Funktion haben. Dabei wird durch geschickte Zusammenstellung der Hindernisse ein Level erstellt. Folgend ist eine Liste von möglichen Spielelementen.

- **Element-Wechsler**  
Diese Kacheln, welche eines der oben gezeigten Symbole tragen, ermöglichen es dem Spieler das Element der Kugel zu ändern. Die Kugel muss dabei einfach über die Kachel rollen.
- **Knöpfe**  
Werden durch die Kugel aktiviert und lösen Aktionen innerhalb des Levels aus.
- **Ventilatoren**  
Sind diese aktiv, beeinflussen sie die Bewegung der Kugel. Befindet sich die Kugel im Erde-Modus, so ist die Beeinflussung durch die Ventilatoren kleiner als normal. Im Luft-Modus sind die Ventilatoren gar unüberwindbar. Ventilatoren können von Knöpfen aktiviert und deaktiviert werden.
- **Türen**  
Diese blockieren gewisse Abschnitte eines Levels und können von Knöpfen beeinflusst werden
- **Holzwände**  
Holzwände blockieren den Durchgang der Kugel, können jedoch im Feuer-Modus abgebrannt werden.
- **Flüsse**  
Hinderniss für die Kugel im Feuer-Modus. Sollte die Kugel im Feuer-Modus auf einen Fluss treffen, so erlischt das Feuer und der Spieler muss die Kugel erneut entzünden.

Es sind noch mehr Spielelemente möglich. Dies soll ein Auszug sein und die Möglichkeiten dieser Idee aufzeigen.

## 2.7 User-Interface (Eingabe)

Das primäre Interface für den Benutzer ist die Tastatur, sowie die Maus. Dabei wird das Menü über die Maus gesteuert, während im Spiel die Kugel mittels der Pfeiltasten gesteuert wird. Zusätzlich kann zu Testzwecken die Kamera mittels des Nummernblocks gesteuert werden.

In Zukunft sollen noch mehr Input-Devices ermöglicht werden. Dabei wurden Gamepads, sowie Devices mit Neigungssteuerung anvisiert (Smart-Phones, Wii-Motes, etc.). Dies vor allem, da das Originalspiel auf diesem Prinzip aufgebaut ist (durch kippen der Spieloberfläche wird die Kugel bewegt).

## 2.8 Zielpublikum

Das Spiel ist für alle Altersstufen geeignet. Es soll die wachsende Indie-Community ansprechen. Zusätzlich sollen kurze, aber interessante Levels erstellt werden, wodurch auch Gelegenheitsspieler, vor allem auf ihren Smartphones, Interesse an dem Spiel finden sollen.

## 2.9 Plattform und Vertriebskanal

Das Spiel wird auf PC entwickelt, welches die Primärplattform ist. Die Engine wurde jedoch so entworfen, dass eine Portierung, v.a. auf mobile Geräte ohne grösseren Aufwand möglich ist. Dies wird vor allem dadurch möglich, dass die Engine komplett auf OpenSource Bibliotheken aufgebaut ist. Windows und Linux werden bereits heute gleichermassen unterstützt, Support für Mac ist geplant.

In naher Zukunft soll ein Eintrag auf Greenlight<sup>7</sup> eingereicht werden. Dies soll ein erster Kontakt mit der Community sein sowie helfen Feedback zu sammeln.

Mögliche Vertriebskanäle sind primär Digital, darunter:

- Steam
- Desura
- GOG
- Ubuntu Software-Center

Nach Portierung auf werden auch die mobilen Plattformen werden auch deren Appstores bedient (iPhone App Store, Google Playstore, usw.) .

---

<sup>7</sup> Steam Greenlight: <http://steamcommunity.com/workshop/about/?appid=765&section=faq>; 10:01:2013, 09:30

## 3 Engine-Design

### 3.1 Übersicht

Da für das Spiel keine bestehende Spiel-Engine, wie beispielsweise idTech, Unity oder die Cry-Engine, um drei prominente Beispiele zu nennen verwendet wurde, mussten verschiedene Bibliotheken für die jeweiligen Aufgaben eingebunden und aufeinander abgestimmt werden. Sämtliche Bibliotheken, die für das Projekt verwendet wurden sind OpenSource-Bibliotheken.

Die Architektur der Spiele-Engine, welche nahezu komplett aus Projekt 2 übernommen wurde, ist stark modular aufgebaut. Das Ziel war es, dass jede Programmbibliothek austauschbar sein soll. Dies ist, bis auf die Verwendung von OGRE, welche das Fundament der Engine bildet, auch gelungen.

Aus dem Projekt 2 waren Graphik, Physik und Input bereits vorhanden. Durch den stark modularen Aufbau konnten diese nahezu unverändert übernommen werden. Die fehlenden Module (Audio, Logik, GUI) wurden während der vorliegenden Arbeit ergänzt. Das Logik-Modul nimmt dabei, wie später erklärt, eine Sonderrolle ein.

### 3.2 Module

Beim Modul-artigen Aufbau der Engine entspricht mit Ausnahme des Logik-Moduls jedes Modul einer externen Bibliothek (siehe Abb. 5). Es ist dadurch möglich mit geringem Aufwand einzelne Bibliotheken durch andere zu ersetzen. Dies erhöht in vielerlei Hinsicht die Flexibilität. Eine Ausnahme ist dabei das Grafik-Modul. dort ist die Abhängigkeit, wie schon erwähnt, wesentlich stärker.

Mermelin						Anwendung
Engine						
Input	Audio	Physik	Graphik	Logik	GUI	Modul
OIS	SMFL	Bullet	OGRE		Rocket	Bibliothek
OGRE						
Betriebssystem						

Abb. 5: Architektur der Anwendung

In den folgenden Kapiteln werden die jeweiligen Module kurz beschrieben, sowie die Art und Weise, wie sie mit der Engine kommunizieren.

#### 3.2.1 Basis- und Grafik-Modul

Der Grafik-Teil nimmt in der Engine einen besonderen Platz ein. Er ist verantwortlich für die Darstellung des Programms am Bildschirm und ist bei dieser Art von Spiel somit der wichtigste Teil überhaupt. Die Aufbereitung der 2D- und 3D-Daten und das Abwickeln der oben beschriebenen Rendering-Pipeline geschieht in diesem Modul.

Zudem erben zahlreiche Objekte der Engine direkt aus der dafür verwendeten Bibliothek oder verwenden implizit und explizit Objekte daraus.

Bei der Graphik-Engine handelt sich um OGRE<sup>8</sup> (Open Source 3D Graphics Engine). Wie schon der Name verrät ist es eine Graphik-Engine. OGRE läuft unter der MIT-Lizenz<sup>9</sup> und hat eine sehr aktive Community. Für die Engine wurde der gegenwärtig aktuelle Release, die Version 1.8, eingesetzt. Die Bibliothek bietet viele nützliche Eigenschaften<sup>10</sup>. Die wichtigsten davon sind:

- Unterstützung für sowohl Low-Level- (in Assembler geschriebene), als auch für High-Level-Shaders, wie sie für die vorliegende Arbeit verwendet wurden und zwar nicht nur GLSL, sondern auch Cg und HLSL
- Unterstützung von OpenGL und Direct3D
- Läuft auf Linux, Windows und Mac OSX
- Ist in C++ geschrieben
- Materials (Definitionen können ausserhalb des Codes stattfinden), inkl. LOD (Level of Detail)
- Unterstützung sämtlicher OpenGL FixedFunction Operationen
- Beinhaltet einen hierarchischen Scenegraphen
- Meshes
- Animationen

Einerseits gibt es das Grafik-Modul, welches explizit der Engine die grafische Funktionalität von OGRE zur Verfügung stellt. Dazu gehört alles was effektiv dargestellt werden muss, wie Beleuchtung, Animation, die Shaders, usw. Andererseits gibt es auch das Basis-Modul, von welchem sämtliche Module (inkl. dem Grafik-Modul) erben. Besonders wichtig ist dabei die Klasse `FrameListener`<sup>11</sup> mit den Methoden `frameStarted`, `frameRenderingQueued` und `frameEnded`, welche jeweils einen `FrameEvent` als Argument entgegennehmen und einen `bool` zurückgeben. Ein `FrameEvent` ist dabei nichts anderes als eine Struktur mit den Werten `timeSinceLastFrame` und `timeSinceLastEvent`, welche die verstrichene Zeit in Sekunden zwischen den Frames angibt. Durch das Erben vom `FrameListener` im `BaseModule` können in den Subklassen diese drei Funktionen verwendet und überschrieben werden. Da `FrameListener` kein reines Interface, resp. abstrakte Klasse ist, sondern wie ein Adapter funktioniert, besteht kein Zwang zur Implementierung der erwähnten Methoden. Dies hat den Vorteil, dass in den Unterklassen nur diejenigen Funktionen überschrieben werden, welche effektiv benötigt werden. Der darin enthaltene Code wird bei jedem einzelnen Bild ausgeführt. Es werden mit der Engine eine Bildrate von 60 FPS (Frames-per-second) angestrebt, wobei die Bildrate stark Hardware-abhängig ist. Damit die Performance nicht darunter leidet, dürfen in diesen Teilen des Programms nur Vorgänge stattfinden, welche einer derart häufigen Aktualisierung bedingen.

Gibt eine der drei beschriebenen Methoden `false` zurück wird das ganze Programm beendet. Dies wird entsprechend verwendet um das Spiel sauber herunterzufahren und alle Ressourcen wieder freizugeben.

GLSL-Shader werden von OGRE, wie erwähnt unterstützt, jedoch nicht alle in gleicher Weise. Die beiden Tessellation-Shader, sowie der Compute-Shader werden darin noch nicht unterstützt. Auch mit dem Geometry-Shader gibt es noch Probleme.

OGRE verwendet, wie andere Graphik-Engines auch, das folgende Koordinatensystem<sup>12</sup>:

---

8 Offizielle Homepage: <http://www.ogre3d.org>

9 Weitere Informationen unter <http://opensource.org/licenses/mit-license.php>

10 Eine vollständige Aufzählung der Features von OGRE findet sich unter <http://www.ogre3d.org/about/features>

11 Doxygen API zum OGRE `FrameListener`: [http://www.ogre3d.org/docs/api/html/classOgre\\_1\\_1FrameListener.html](http://www.ogre3d.org/docs/api/html/classOgre_1_1FrameListener.html)

12 Quelle: <http://www.ogre3d.org/wiki/Basic+Tutorial+1>

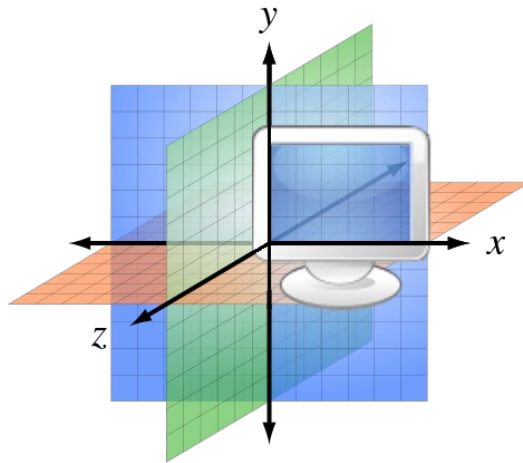


Abb. 6: Das OGRE-Koordinatensystem

### 3.2.1.1 Scene-Graph

OGRE basiert auf dem Konzept des Scene-Graphen. Ein Scene-Graph ist eine Datenstruktur, welche die logische und räumliche Organisation einer grafischen Szene beschreibt. Dabei werden alle Objekte mindestens an eine Scene-Node<sup>13</sup>, der sogenannten Rootscenenode, angehängt.

Jede Scene-Node speichert dabei nur seine relative Position zu ihrer darüberliegenden SceneNode, die Parent-Scene-Node. Dies hat den Vorteil, das sobald eine Scene-Node transformiert wird, ihre anhängenden Scene-Nodes (Child-SceneNodes), direkt ihre neue Position und Ausrichtung übernehmen.

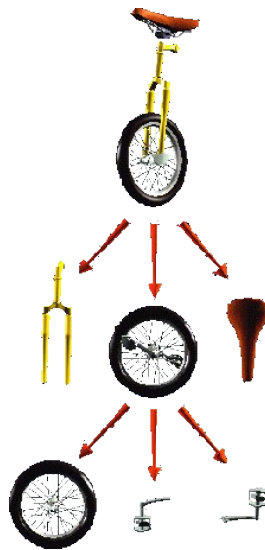


Abb. 7: SceneGraph Beispiel

Als Beispiel dazu sei die oben gesehene Grafik zu betrachten. Bewegt sich das Einrad vorwärts, so wird nur die oberste Scene-Node verschoben. Die darunterliegenden werden automatisch mitgezogen, da sie ja nur relative Positionen zu ihrem Parent gespeichert haben.

Rotiert nun aber das Rad, sind nur dessen Children betroffenen und erhalten eine neue Orientierung. Der Sattel und die Gabel werden von dieser Rotation nicht beeinflusst, da sie keine Children des Rades sind.

Des weiteren ist das Ziel des Scene-Graphen das Rendering zu beschleunigen, da nur Objekte in einem sichtbaren Ast von der Rendering-Pipeline bearbeitet werden müssen.

<sup>13</sup> OGRE SceneNode: [http://www.ogre3d.org/docs/api/html/classOgre\\_1\\_1SceneNode.html](http://www.ogre3d.org/docs/api/html/classOgre_1_1SceneNode.html)

### 3.2.2 Input-Modul

Dieses Modul ist zuständig für die Umsetzung der Benutzereingabe. Es handelt sich dabei im jetzigen Stand der Entwicklung ausschliesslich um Maus- und Tastatur-Eingaben. Das Modul ist jedoch in der Lage beliebige Arten von Inputs entgegen zu nehmen und zu verarbeiten. Zusätzlich kommen also nebst der Maus und dem Joystick eines PCs zum Beispiel auch die Wiimote oder das Gyroskop eines Smartphones in Frage.

Das Modul wird derzeit mit OIS<sup>14</sup> (Object Oriented Input System) realisiert.

Es erbt zusätzlich zum BaseModule von KeyListener und MouseListener aus OIS, sowie den WindowEventListener von OGRE.

Die Eingaben werden auf zwei unterschiedliche Arten verarbeitet: Gepuffert in der überschriebenen `frameRenderingQueued` Funktion und ungepuffert in den Funktionen `keyPressed` und `keyReleased` welche jeweils einen `KeyEvent` entgegennehmen. Der Unterschied lässt sich leicht erklären. Im ersten Fall wird bei jedem Aufruf von `frameRenderingQueued` geprüft, ob eine Bestimmte Taste gedrückt wurde oder nicht.

```
...
bool InputModule::frameRenderingQueued(const Ogre::FrameEvent &evt)
{
    if (key->isKeyDown(OIS::KC_Q) ||
        (key->isKeyDown(OIS::KC_LMENU) && key->isKeyDown(OIS::KC_F4))) {
        return false;
    } else {
        return true;
    }
}
...
```

*Text 1: InputModule: Beenden des Spiels*

Es wird aktiv geprüft, ob eine bestimmte Taste gedrückt ist oder nicht. Nur auf diese Weise ist es möglich Tastenkombinationen, wie Alt+F4 zu realisieren. Es ist aber möglich, dass Tastatur-Eingaben verloren gehen, weil nur die Tasten registriert werden, welche beim Aufruf von `frameRenderingQueued` gedrückt sind. Will man mit einem kontinuierlichen Tastendruck arbeiten gibt es keine andere Möglichkeit. Gerade bei der Steuerung der Kugel im Spiel will der Spieler für die Bewegung in eine bestimmte Richtung die Tastatureingabe nicht wiederholen müssen.

Es gibt jedoch andere Situationen, bei denen es gewünscht ist, dass das einmalige Drücken einer Taste garantiert registriert wird. Dafür sind dann die gepufferten Methoden `keyPressed` und `keyReleased` zuständig. Die Funktionen werden nur dann aufgerufen, wenn ein Tastendruck erfolgt. Beim erstmaligen Druck einer Taste wird `keyPressed` und danach beim Lösen der Taste wird `keyReleased` aufgerufen.

Das Input-Modul hängt kommuniziert auch mit dem GUI-Modul, wenn das Menu sichtbar ist.

---

<sup>14</sup> OIS bei Source-Forge: <http://sourceforge.net/projects/wgois/>

```

...
bool InputModule::keyPressed(const KeyEvent& arg)
{
    switch (arg.key) {
    ...
        case OIS::KC_F3:
        {
            if (ENGINE->getState() != Engine::DEBUGING) {
                ENGINE->setState(Engine::DEBUGING);
            } else {
                ENGINE->setState(Engine::RUNNING);
            }
        }
    ...
    }
    ...
}

```

*Text 2: InputModule: Umschalten zwischen Running- und Debug-Modus*

Der gezeigte Code versetzt beim Druck der F3-Taste die Engine in den Debug-Modus, wenn sie sich im Running-Modus, befindet und verlässt den Debug-Modus in den Running-Modus im umgekehrten Fall.

### 3.2.3 Audio-Modul

Wie der Name es schon verrät existiert auch ein Audio-Modul welches dazu verwendet werden kann Töne oder Musik abzuspielen. Da OGRE eine reine Rendering-Engine ist, musste für den Sound eine andere Lösung gefunden werden. SFML<sup>15</sup> (Simple and Fast Multimedia Library) wurde dabei als ideale Lösung befunden. Als File-Format wird OGG Vorbis<sup>16</sup> verwendet, welches ein freies Format ist.

#### 3.2.3.1 Audio aus dem File-System laden

Die grösste Schwierigkeit ist das Laden von Sound-Files. OGRE bietet von sich aus ein erweiterbares Ressourcen-System, welches für Audio-Dateien ausgebaut werden musste. Dazu wurde eine Audio-Klasse erstellt, welche innerhalb der Engine als Träger für ein Sound-File dient. Die Audio-Datei wird dabei vom File-System ausgelesen und direkt als Stream von SFML ausgelesen.

```

void Audio::loadImpl()
{
    Ogre::DataStreamPtr stream =
        Ogre::ResourceManager::getSingleton().
            openResource( mName, mGroup);

    std::vector<char> bytes;

    // settin output values
    bytes.resize(stream->size());

    // getting the binary file contents
    stream->read(&bytes[0], bytes.size());

    sf::SoundBuffer* buffer = new sf::SoundBuffer();
    buffer->LoadFromMemory(&bytes[0], bytes.size());

    sound = new sf::Sound(*buffer);
}

```

*Text 3: Audio: Laden einer Audio-Datei vom File-System*

<sup>15</sup> SMFL Homepage: <http://www.sfml-dev.org/>

<sup>16</sup> OGG Vorbis: <http://de.wikipedia.org/wiki/Vorbis>; 10.01.2013, 21:00



Mittels `mGroup` kann OGRE ermitteln, in welchem Pfad sich eine beliebige Datei mit dem Namen `mName` befindet. Durch das Erstellen eines `DataStreamPtr` wird die Datei in den Arbeitsspeicher geladen. SFML bietet darauf die Möglichkeit direkt aus dem Arbeitsspeicher eine Datei auszulesen und in ein abspielbares Format zu konvertieren. Dazu wird ein `SoundBuffer`<sup>17</sup> erstellt, welcher den Stream ausliest.

Ein Pointer auf den eben erstellten Sound wird in der Audio-Klasse gespeichert und kann mittels der Methode `getAudio` von anderen Klassen (Entitäten, Komponenten) benutzt werden.

### 3.2.3.2 3D-Sound

In einem 3D Spiel wird selbstverständlich erwartet, dass der Sound sich dynamisch der Position des Spielers oder der Kamera anpasst. SFML bietet diese Möglichkeit auf eine angenehm simple Weise.

Um einen 3D Soundeffekt zu erhalten muss erstmals beachtet werden, dass nur Audio-Datei mit einer Mono-Tonspur in einer 3D-Umgebung platziert werden können. Stereo- oder Mehrspurige Audio-Dateien werden von SFML nicht für 3D akzeptiert.

Jeder Sound muss danach mittels eines 3D Vektors platziert werden. Dies wird automatisch von der Audio-Komponente übernommen, welche die Position der Entität auf alle relevanten Audio-Klassen überträgt. Somit ist die Platzierung der Soundeffekte gegeben.

Ohne dass nun aber die Position des Hörers verändert wird, ist der 3D-Sound Effekt nicht hörbar. Deshalb wird im Audiomodul in jeder Update-Schleife die Position des Hörers an die Kamera angepasst. Dies geschieht über ein globales Objekt, welches von SFML bereitgestellt wird, dem `Listener`<sup>18</sup>.

```
bool AudioManager::frameRenderingQueued(const Ogre::FrameEvent &evt)
{
    Ogre::Vector3 position =
        ENGINE->getCamera()->getNode()->_getDerivedPosition();

    sf::Listener::SetPosition(position.x, position.y, position.z);
    return true;
}
```

*Text 4: AudioManager: Aktualisieren des globalen Listeners*

Es muss dabei darauf geachtet werden, dass hier die Position der Kamera im World-Space benötigt wird.

### 3.2.4 Physik-Modul

Physik ist, neben der Graphik, der zweite grosse Teil welcher nötig ist um das Spiel zu realisieren. Die Kugel muss sich physikalisch korrekt verhalten, damit der Spieler die Bewegungen akkurat schätzen und steuern kann.

Zu diesem Zweck wird Bullet<sup>19</sup> verwendet, eine Open-Source Physik-Engine. Bullet wird sowohl in der Games- als auch in der Filmindustrie erfolgreich eingesetzt. Ausserdem gibt es gute Wrapper, welche OGRE und Bullet miteinander verbinden, was die Integration erheblich vereinfacht.

Bullet bietet eine akkurate Abbildung der physikalischen Gesetze sowie eine gute Kollisionserkennung. Dies ist besonders wichtig für Objekte welche auf Kontakt miteinander reagieren sollen (Knöpfe, Türen, Element-Wechsler, usw..)

Die Physik muss regelmässig aktualisiert werden. Dies geschieht in der Engine einmal pro Frame. Dies liefert in der Regel genügend genaue Ergebnisse.

<sup>17</sup> `sf::SoundBuffer`: [http://www.sfm-dev.org/documentation/1.3/classsf\\_1\\_1SoundBuffer.php](http://www.sfm-dev.org/documentation/1.3/classsf_1_1SoundBuffer.php)

<sup>18</sup> `sf::Listener`: [http://www.sfm-dev.org/documentation/2.0/classsf\\_1\\_1Listener.php](http://www.sfm-dev.org/documentation/2.0/classsf_1_1Listener.php)

<sup>19</sup> Bullet Website: <http://bulletphysics.org/wordpress/>

Bullet baut auf dem Prinzip von Welten auf. Damit ist gemeint, dass alle Objekte, welche Teil der physikalischen Welt sind, sich in einem gemeinsamen Raum befinden. In diesem interagieren sie miteinander, gemäss den physikalischen Gesetzen (Newton'sche Gesetze). Dieser Welt wird nun einmal pro Frame befohlen ein Update zu machen. Dabei finden verschiedene Prozesse statt:

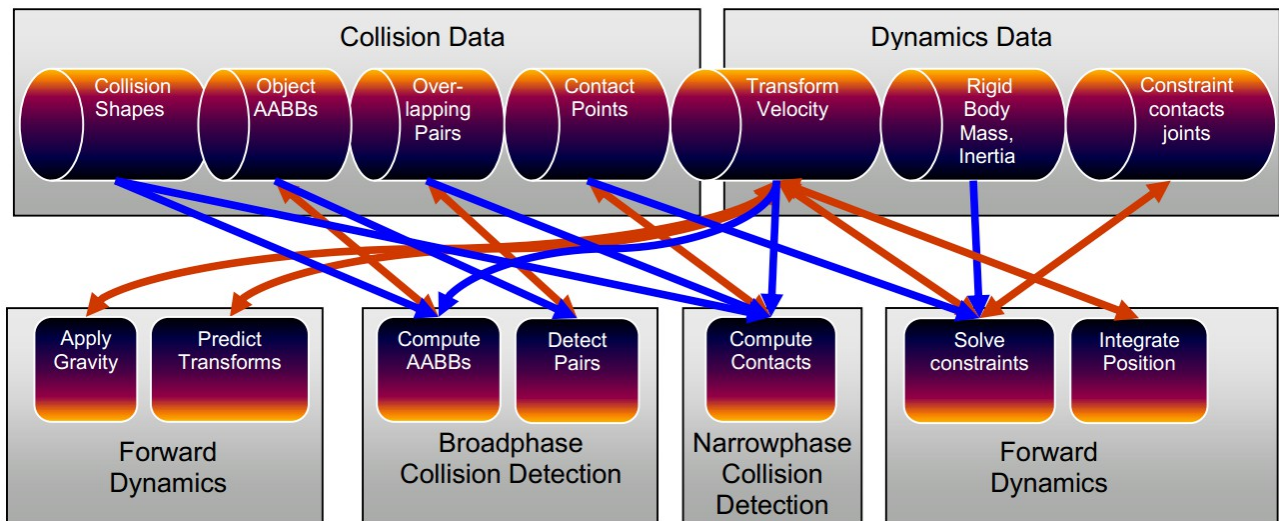


Abb. 8: Graphische Darstellung der Physikpipeline

### 3.2.4.1 Optimierungen

Die Physik ist traditionsgemäss eine sehr zeitintensive Anwendung. Mittels Profiling wurde festgestellt, dass ca. 22% der gesamten Arbeit der Engine im Updaten der Physik verrichtet wird. Daher wurden für diesen Teil der Engine einige Optimierungen vorgenommen.

Normalerweise wird jedes Modul einmal pro Frame aufgerufen. Beim Physik-Modul wurde jedoch entschieden, dass dieses mehrmals während eines Frames aktiviert wird. OGRE erlaubt es an 3 Stellen während des Renderingprozesses einzugreifen, jeweils am Start (`frameStarted`), bevor die GPU aktiv wird (`frameRenderingQueued`), sowie am Ende eines Frames (`frameEnded`).

Zu Beginn des Frames werden Kollisionen abgefragt und behandelt. Dies wird im nächsten Abschnitt detaillierter beschrieben.

Während die GPU am Arbeiten ist, kann dazwischen die CPU die aufwendigen physikalischen Berechnungen bearbeiten. Dies geschieht in der Methode `frameRenderingQueued`.

```
...
bool PhysicsModule::frameRenderingQueued(const Ogre::FrameEvent& evt)
{
    dynamics->stepSimulation(evt.timeSinceLastFrame, 10);
    return true;
}
...
```

Text 5: *PhysicsModule*: Aktualisieren der physikalischen Welt

Die Methode ist nicht sehr umfangreich. Es wird lediglich ein Simulationsschritt ausgeführt. Jedoch ist es genau diese Methode, welche den Grossteil der oben beschriebenen 22% ausmacht.

### 3.2.4.2 Kollisionserkennung

Damit Entities miteinander interagieren können, müssen sie wissen ob sie mit einem interaktiven Objekt kollidiert sind. Dazu wird die Kollisionserkennung von Bullet verwendet.

Es gibt zwei Möglichkeiten um Kollisionen zu behandeln. Die erste Methode ist hier dargestellt:

```
...
bool PhysicsModule::frameStarted(const Ogre::FrameEvent& evt)
{
    int numManifolds = dynamics->getDispatcher()->getNumManifolds();
    for(int i = 0; i < numManifolds; i++){
        btPersistentManifold* contactManifold =
            dynamics->getDispatcher()->getManifoldByIndexInternal(i);
        btCollisionObject* obA =
            static_cast<btCollisionObject*>(contactManifold->getBody0());
        btCollisionObject* obB =
            static_cast<btCollisionObject*>(contactManifold->getBody1());
        PhysicsComponent* phyA =
            static_cast<PhysicsComponent*>(obA->getUserPointer());
        PhysicsComponent* phyB =
            static_cast<PhysicsComponent*>(obB->getUserPointer());
        phyA->handleContact(phyB);
        phyB->handleContact(phyA);
    }
    dynamics->clearForces();
    return true;
}
...
```

Text 6: PhysicsModul: Abarbeitung der ContactManifolds

Nach jeder Aktualisierung der Physik werden Kollisions-Manifolds<sup>20</sup> erstellt. Diese enthalten alle benötigten Informationen über eine Kollision zwischen zwei Objekten.

Diese Liste wird abgearbeitet. Bei jedem Manifold wird die dazugehörige Physik-Komponente ausgelesen und von der Kollision benachrichtigt.

Die zweite Möglichkeit sind Ghost-Objekte<sup>21</sup>. Ein Ghost-Objekt hält eine Liste aller Objekte, welche mit ihm überlappen. Dadurch können Objekte wie Knöpfe oder Druckplatten einfacher erstellt werden.

Dazu wird eine weitere Klasse benötigt: GhostComponent.

Das Physikmodul ruft bei jeder Schleife jede registrierte GhostComponent auf, welche wiederum seine Parent-Entität von Kollisionen benachrichtigt, falls solche vorhanden sind.

```
bool PhysicsModule::frameStarted(const Ogre::FrameEvent&evt)
{
    for (std::map<std::string, GhostComponent*>::iterator it = ghosts.begin();
        it != ghosts.end(); it++) {
        it->second->checkOverlappingObjects();
    }
    dynamics->clearForces();
    return true;
}
```

Text 7: PhysicsModule: Abarbeitung der Ghost-Objekts

<sup>20</sup> btPersistentManifold: <http://bulletphysics.com/Bullet/BulletFull/classbtPersistentManifold.html>

<sup>21</sup> btGhostObject: <http://bulletphysics.com/Bullet/BulletFull/classbtGhostObject.html>

Dies gibt einen enormen Performance-Vorteil, da so nicht jede Entität bei jeder Schleife aufgerufen werden muss, sondern nur solche, welche auch effektiv an einer Kollision teilnehmen.

In der Methode `checkOverlappingObjects` wird nun überprüft, ob eine Entität mit dem aktuellen `GhostObject` kollidiert. Falls ja, werden entsprechende Events ausgelöst:

- `COLLISION_ENTER`  
Ein Objekt kollidiert mit dem aktuellen `GhostObject`, welches in der letzten Updateschleife noch nicht kollidierte
- `COLLISION`  
Ein Objekt kollidiert mit dem aktuellen `GhostObject`, welches bereits in der letzten Updateschleife mit dem aktuellen `GhostObject` kollidierte.
- `COLLISION_EXIT`  
Ein Objekt, welches in der letzten Updateschleife noch mit dem aktuellen `GhostObject` kollidierte, kollidiert nun nicht mehr.

Die Methode `checkOverlappingObjects` ist umfangreich, weshalb hier mit Pseudo-Code der Ablauf kurz erläutert werden soll

```
for(each colliding entity)
{
    Did it collide previously?
    Yes:
        Send COLLISION event
    No:
        Send COLLISION_ENTER event
        Register in previously collided list
}

for(each entity in previously collided list)
{
    Is entity still colliding
    Yes:
        Do nothing
    No:
        Send COLLISION_EXIT event
        Delete from previously collided list
}
```

*Text 8: GhostComponent: Pseudo-Code zur Kollisionsbearbeitung*

### 3.2.5 GUI-Modul

Damit der Spieler mit dem Spiel interagieren kann wird folgendes benötigt:

- Eine Möglichkeit den Input des Spielers zu lesen
- Relevante Daten des Spiels dem Spieler zu präsentieren



Abb. 9: GUI von Mermelin

Der erste Punkt wird dabei vom bereits beschriebenen InputModul erfüllt, der zweite wird nun mit dem GUIModul erreicht.

Das GUIModul baut auf libRocket<sup>22</sup> auf. LibRocket ist ein C++ Benutzerinterface, welches auf HTML und CSS Standards basiert. Dabei benutzt es die Rendering-Möglichkeiten von OGRE um die benötigten Overlays zu rendern.

### 3.2.6 Logik-Modul

Damit im Spiel Türen geöffnet werden können oder Ventilatoren betätigt werden können müssen bestimmte Bedingungen definiert werden und die Möglichkeit geschaffen werden, die Erfüllung derselben zu überprüfen. Das Logik-Modul erfüllt diesen Zweck. Im Gegensatz zu den anderen Modulen ist hierbei keine externe Bibliothek involviert, sondern es wurde von Grund auf selbst entwickelt. Dies, da keine bestehende Lösung als genügend empfunden wurde.

Das Composite Design-Pattern kommt hier zum Einsatz. Das Modul verwaltet zwei Arten von Logik-Komponenten: Die Klassen Trigger (Auslöser) und Condition (Bedingung), welche beide von der abstrakten Klasse LogicComponent abgeleitet sind. Sie verfügen beide über eine check Funktion, welche einen bool zurückgibt und als pure virtual in LogicComponent deklariert ist.

#### 3.2.6.1 Triggers

Auslöser sind in Mermelin die Knöpfe, welche über eine Instanz von LogicComponent verfügen. Einem Trigger können beliebig viele Targets (Ziele) via addTarget angefügt werden. Ein Target ist dabei nichts anderes als eine Condition. Jeder Trigger verfügt über einen Namen, welchen ihn identifiziert. Dieser Name dient der getTrigger-Methode in LogicModule den entsprechenden Trigger in der map, in der sie intern abgespeichert werden wiederzufinden.

---

<sup>22</sup> Homepage der Rocket-Bibliothek: <http://librocket.com/>

Findet eine Kollision der Kugel mit einem Knopf statt wird die `receiveEvent` Methode von Trigger aufgerufen, welche die Funktion `fire` aufruft. Diese benachrichtigt dann alle registrierten Targets mit einem Event des Typs `TRIGGER` und setzt die interne Variable `fired` auf `true`. Die `check`-Funktion liefert dann für diese Trigger-Instanz `true` zurück.

### 3.2.6.2 Conditions

Bedingungen sind Türen und Ventilatoren, welche gleich wie auch die Auslöser über eine Instanz von `LogicComponent` verfügen. Conditions werden rekursiv angewandt. Eine Condition kann aus mehreren Subconditions bestehen. Dabei verfügt jede Condition über einen Operator (`AND`, `OR`) und über eine Reihe von Triggers und Subconditions welche damit verbunden und geprüft werden.

Erhält die Condition einen `TRIGGER`-Event ruft sie die `check`-Funktion auf. Diese ruft dann wiederum für alle Triggers und Subcondition die `check`-Funktion auf.

## 3.3 Komponenten

Game-Engines sind von Natur aus sehr dynamisch und können deshalb durch einen zu monolithischen Aufbau (Objekte nur mittels Vererbung erstellen) sehr schnell unübersichtlich werden. Deshalb wird oftmals ein „Component-Based Entity System“ aufgebaut.

Jede Komponente erbt dabei von einer gemeinsamen Basis-Komponente. Dies ermöglicht es in einer Entität eine simple Liste zu halten, in welcher Zeiger auf alle Komponenten der Entität gespeichert sind.

Da jedes Modul, und dessen entsprechende Komponenten, eine andere Aufgabe und somit auch anders aktualisiert werden müssen macht es Sinn, das nicht die Entität, sondern das Modul selbst die Hoheit über die Komponente behält. Somit ist Erstellung und Verwaltung der Komponenten Aufgabe der Module. Eine Entität kann nur bei einem Modul eine Komponente verlangen, sie aber nicht selbst erzeugen.

Die Sequenz, um eine Komponente zu erstellen sieht demnach wie folgt aus:

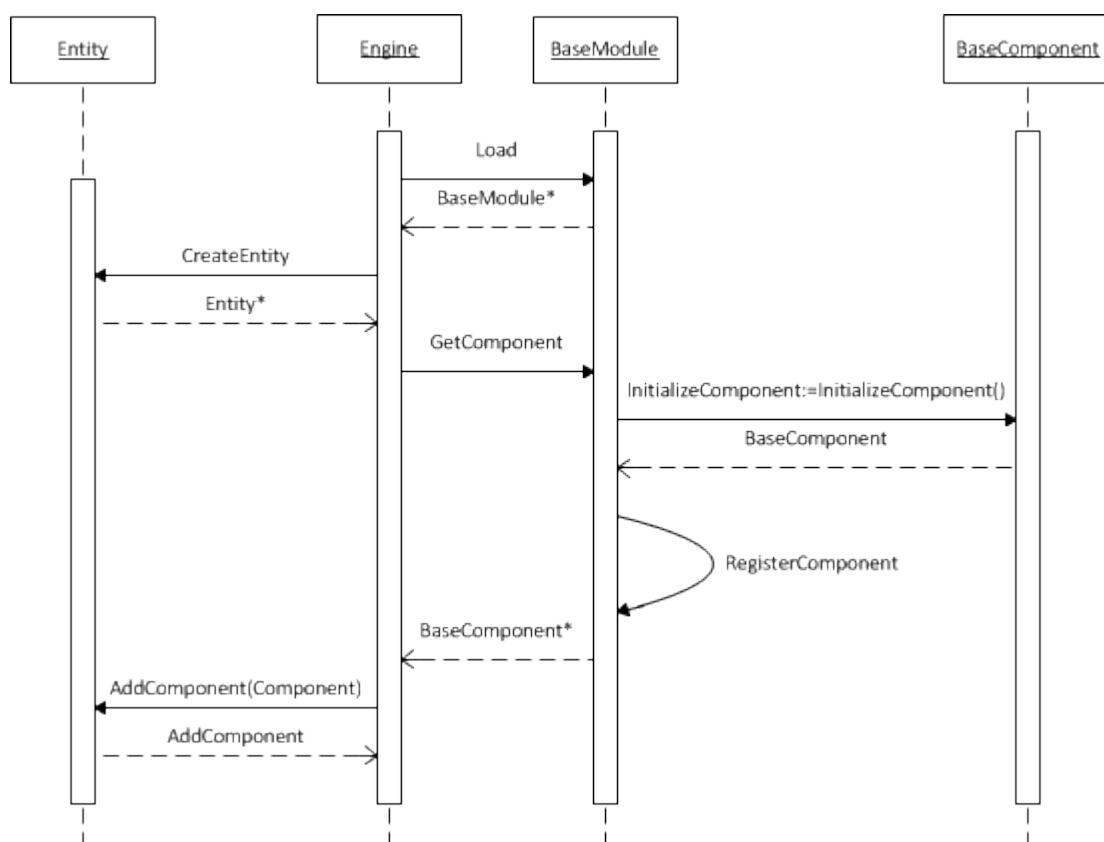


Abb. 10: Erstellung der Komponenten

Ähnlich sieht nun der Prozess zur Entfernung einer Komponente aus:

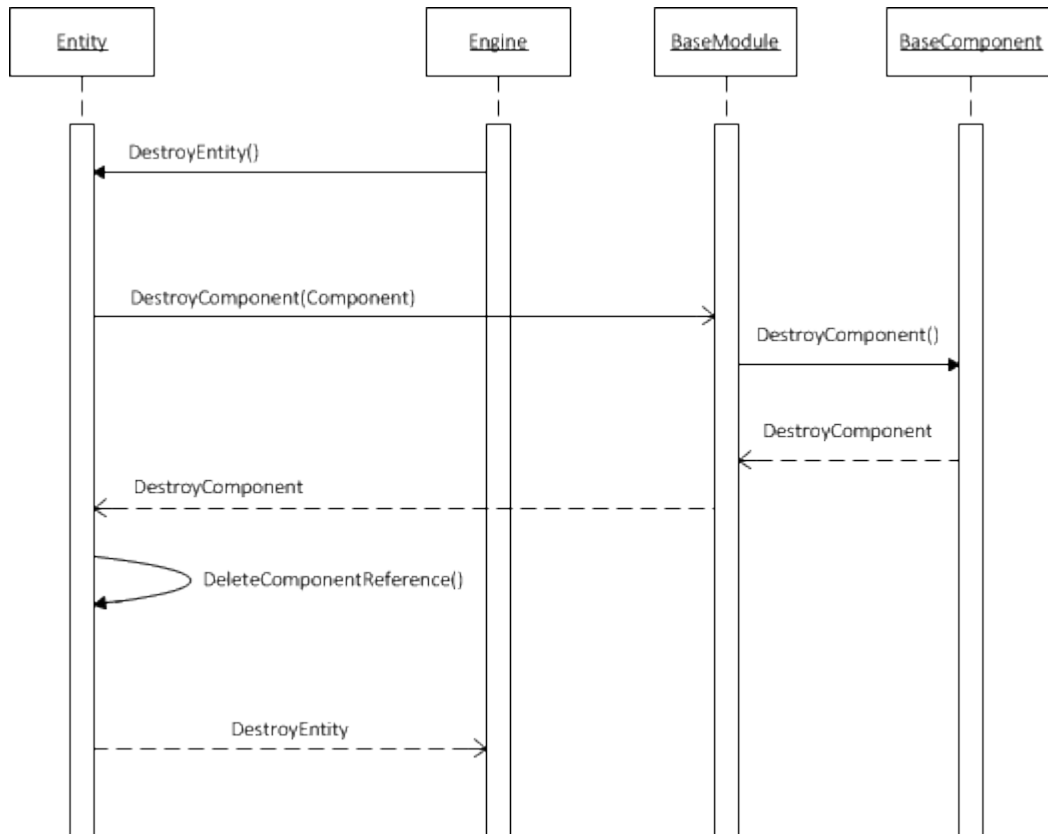


Abb. 11: Entfernung einer Komponente

Wie ersichtlich ist die Entität somit nur eine Containerklasse, was bedeutet, dass sie nur als gemeinsamer Zugriffspunkt für eine Anzahl Komponenten dient. Es tritt somit die berechtigte Frage auf, ob die Entität überhaupt benötigt wird. Die simple Antwort ist nein. Die Engine könnte genauso gut ohne eine Entity-Klasse erstellt werden.

Jedoch bietet die Entity-Klasse ebenfalls einige Vorteile. So können Klassen erstellt werden (welche von der Entity erben), welche direkt selbst auswählen, welche Komponenten sie benötigen. Auch vermag die Entity die Kommunikation zwischen den einzelnen Komponenten zu erleichtern, wie im nächsten Kapitel erläutert wird.

### 3.4 Event-System

Das komponentenorientierte System hat den Nachteil, dass generische Klassen benutzt werden, welche auf irgend eine Art untereinander kommunizieren müssen. Da eine Game-Engine ausserdem eine stark interaktive Anwendung ist, wird eine einfache Kommunikation zwischen allen Objekten innerhalb der Engine benötigt.

Für die CotopaxiEngine wurde deshalb ein sehr flexibles Event-System aufgebaut, welches sowohl die direkte Kommunikation zwischen einzelnen Komponente erlaubt, als auch Broadcasts der Engine bis hinunter zu den Komponente. Kurz gesagt, jedes Objekt innerhalb der Engine hat die Möglichkeit mit einem beliebig anderen zu kommunizieren.

Basis für das Event-System ist dabei die Event-Klasse. Sie enthält einen Event-Typen (momentan gegeben durch ein Enum), sowie einen Pointer auf die sendende Entity/Komponente. Dies ermöglicht es dem Empfänger festzustellen von wo der Event gesendet wurde, falls dies benötigt wird.



Obwohl diese im Programmcode nicht unterschieden werden, kann man 3 Arten Events identifizieren:

**1. Lokal**

Hierbei handelt es sich um Events, die keinen Einfluss auf externe Elemente haben. Sie betreffen nur den internen Status der Entity.

Ein Beispiel wäre der Wechsel der Animation eines Charackters, z.B. wenn der Spieler anfängt zu rennen, sendet eine Input Component einen Event an die Animation Component um ihr den Statuswechsel mitzuteilen.

**2. Global**

Es gibt Fälle, wo ein Event einer Component mehrere Entities betrifft, z.B. wenn ein Spieler eine Tür öffnet (betroffen sind die Tür-Entity, sowie die Spieler-Entity).

Dabei sendet die Input-Component des Spielers einen globalen Event, welcher von der Engine an die betroffenen Entities delegiert wird.

**3. Inter-Modular**

Kommunikation wird teilweise auch nur zwischen Komponenten innerhalb eines Modules benötigt. Dies ist bei Modulen der Fall, welche ein grösseres Zusammenspiel zwischen ihren Komponenten benötigen.

Als Beispiel soll hier das Physik-Modul angeführt werden. Die Kollisionserkennung läuft zwischen Rigidbodies ab, welche in den Physik-Komponenten referenziert sind. Kollidieren nun zwei Komponenten, so tauschen diese direkt ihre Informationen miteinander aus, anstatt den Umweg über die Entity zu nehmen. Wie die Komponenten schlussendlich diese Informationen verarbeiten bleibt ihnen selbst überlassen.

## 3.5 Level-Parsing

Um die Szene für das Spiel aufzubauen müssen die benötigten Informationen aus der Level-Datei gelesen werden. Um den Aufbau möglichst einfach zu halten wurde nicht auf ein existierendes Format, wie XML gesetzt, sondern ein eigenes Format definiert, welches genau den Anforderungen von Einfachheit, Geschwindigkeit und Übersichtlichkeit gerecht wird. Als Basis dienen dabei simple Textdokumente (.txt).

Das Spiel ist Kachel-artig aufgebaut. Eine Kachel entspricht standartmässig einer OGRE-Einheit. Eine Skalierung ist über die Konstante SCALE der Engine möglich.

Das ein Level auf die Weise aufgebaut werden kann, wie nachfolgend noch genauer geschildert wird, setzt voraus, dass die darin verwendeten Objekte normiert sind. Bei der Modellierung muss also genau darauf geachtet werden, dass es später nicht zu Überlappungen oder Spalten kommt.

### 3.5.1 Kommentare

Eine mit # beginnende Zeile in der Text-Datei wird als Kommentar-Zeile interpretiert und ignoriert.

### 3.5.2 Abschnitte

Die Klasse Level ist für das Einlesen der Level-Datei verantwortlich. Die Datei wird Zeilen-weise durch die readFile Funktion eingelesen. Das Format sieht vier verschiedene Abschnitte vor. Ein Abschnitt wird durch den jeweiligen Namen, gefolgt von einem Doppelpunkt, gekennzeichnet. Zum Beispiel Properties:. Die von der readFile angerufene Funktion readLine versetzt die nach dem Zustands-Entwurfsmuster aufgebaute Klasse in einen dafür vorgesehenen Zustand. Abhängig vom Zustand wird dann für jede Zeile eine andere Funktion aufgerufen.

### 3.5.2.1 Properties

Befindet sich der Parser (die Klasse `Level`) im `PROPERTIES` Zustand, so wird die `readPropertiesLine` Methode ausgeführt. In diesem Abschnitt sind die folgenden drei Deklarationen möglich:

- `name = <Level-Name>`                      *Bsp.: `name = ExampleLevel`*
- `start = (x, y, z)`                            *Bsp.: `start = (4, 3, 4)`*
- `camera = (x, y, z)`                         *Bsp.: `camera = (0, 5, -5)`*

Dem Level kann mit `name` eine Bezeichnung angegeben werden, welche als String abgespeichert wird. Mit `start` wird die Anfangsposition der Kugel angegeben und `camera` bestimmt schliesslich den Abstand (Offset) von Kamera und Kugel. Die Angabe ist sowohl mit Ganzzahlen, als auch mit Gleitkommazahlen möglich.

### 3.5.2.2 Logic

Im `LOGIC` Zustand wird die `readLogicLine` Methode ausgeführt. Dabei sind die folgenden drei Deklarationen möglich:

- `button = <Button-Name>, (x, y, z)`
- `door = <Door-Name>, <Ausrichtung>, <Ausdruck>, (x, y, z)`
- `rotor = <Rotor-Name>, <Button-Name>, (x, y, z)`

Diese Liste wird mit der Zeit noch erweitert, sobald neue Spielelemente hinzugefügt werden.

Ein `button` ist ein Knopf, welcher einen Namen und eine Position besitzt. Mit `door` wird eine Türe dargestellt, was einer `Door` Entität entspricht, welche über eine Logik-Komponente (`LogicComponent`) verfügt. Es kann entweder in Richtung der X- oder Z-Achse platziert werden, je nach dem wie die umgebenden Wände es erfordern. Dies geschieht mit den Schlüsselwörtern `horizontal` und `vertical`, wenn das Tor in der Mitte des Feldes erstellt werden soll und mit `top`, `left`, `right` und `bottom`, wenn es sich an einem Rand befinden soll. Danach folgt ein logischer Ausdruck, welcher bestimmt welche Bedingung erfüllt werden muss, damit sich das Tor öffnet. Standardmässig sind alle Türen geschlossen. Dieser Ausdruck besteht durch „:“ getrennte, sowie mit den logischen Operatoren `AND` und `OR` verbundene Namen der betreffenden Knöpfe. Schliesslich erfolgt, wie bei den anderen Entitäten noch die Positionsangabe. Der Rotor funktioniert ähnlich. Er wird allerdings nur von einem einzelnen Knopf aktiviert.

Es folgt zur Verdeutlichung ein kleines Beispiel:

```
...
Logic:
button = b1,(12, 0, 14)
button = b2,(16, 0, 10)
button = b3,(4, 1, 3)
door   = d1, horizontal, b1:AND:b2:OR:b3, (2, 0, 2)
door   = d2, horizontal, b3, (6, 0, 10)
door   = d3, vertical, b3, (4, 0, 13)
rotor  = r1, b1, (6, 0, 3)
...
```

*Text 9: Level-Datei: Beispiel eines Logik-Abschnitts*

Man beachte, dass die Knöpfe zwingend vor ihrer Verwendung deklariert werden müssen.

Bei Logik-Objekten macht es keinen Sinn sie mit nicht-ganzzahligen `float`-Werten zu positionieren, da sie im Raster des Spiels nur korrekt positioniert werden können, wenn sie an einer dafür vorgesehenen Stelle auf dem positioniert werden. Dazu mehr beim `Objects`-Abschnitt.

### 3.5.2.3 Free

Im FREE State können beliebige Meshes an einer beliebigen Stelle positioniert werden. Die Syntax einer solchen Zeile ist die folgende:

<Mesh-Name> = <Beliebiger-Name>, <Masse>, (6, 0, 3)

Zum Beispiel: Cube = cube1, 0.0, (13, 0.5, 4)

Dies bedingt, dass die entsprechende Datei (hier Cube.mesh) existiert. An erster Stelle muss dem Objekt ein Name vergeben werden, danach folgt eine Angabe über die Masse, welche auch entscheidet, ob es sich um ein kinematisches, oder ein dynamisches Objekt in der Physik-Welt handelt. Alle Objekte mit der Masse 0.0 werden als kinematische Objekte behandelt. Schliesslich darf auch hier die Anfangsposition nicht fehlen.

### 3.5.2.4 Objects

Der Aufbau des gekachelten Spielraums geschieht im OBJECTS Zustand. Das Spielfeld besteht aus mehreren Stockwerken. Jedes Stockwerk besteht aus mehreren Schichten, da pro Schicht jeweils nur ein Objekt an einer Stelle (x, z) erstellt werden kann. Es benötigt mehrere Schichten um beispielsweise Mauern, Böden und weitere Objekte darzustellen. Es kann beliebig viele Schichten pro Stockwerk geben.

Jedes Zeichen auf einer ObjectLine-Zeile repräsentiert ein Objekt, welches im 3D-Raum dargestellt werden soll. Die Funktion readObjectLine arbeitet Zeichen für Zeichen ab und erstellt mit Hilfe der Methode readObject das passende Objekt an der zugehörigen Stelle. Diese wird über einen durch drei Variablen erzeugten Vektor bestimmen. Diese Variablen sind objectCounter, floorCounter, lineCounter. Der objectCounter entspricht der Breite, der lineCounter der Länge und der floorCounter der Höhe, resp. dem Stockwerk. Um dies effizient zu implementieren werden zwei Flags verwendet:

- @ zeigt an, dass eine neue Schicht für das selbe Stockwerk erstellt werden soll. Die Variable lineCounter wird auf 0 gesetzt.
- % zeigt an, dass das von nun an ein neues, darüberliegendes Stockwerk erstellt werden soll. Die Variable lineCounter wird auf 0 gesetzt und die Variable floorCounter wird inkrementiert.

Die nachfolgende Tabelle zeigt, auf welches Zeichen für welches Objekt eingesetzt wird





































Zeichen	Mesh / Entität	Objekt	Zeichen	Mesh / Entität	Objekt
f	Floor.mesh		M	WoodWall	
+	Wall_Cross.mesh		N	WoodWall	
*	Hole.mesh		o	WoodWall	
-	Wall_Line.mesh		O	WoodWall	
	Wall_Line.mesh		m	WoodWall	
{	Wall_Curve.mesh		n	WoodWall	
}	Wall_Curve.mesh		Q	Wall_Line.mesh	
C	Wall_Curve.mesh		V	Wall_Line.mesh	
D	Wall_Curve.mesh		B	Wall_Line.mesh	
a	Wall_Corner.mesh		R	Wall_Line.mesh	
[	Wall_Corner.mesh		\$	Wall_Border_T.mesh	
]	Wall_Corner.mesh		<	Wall_Border_T.mesh	
d	Wall_Corner.mesh		!	Wall_Border_T.mesh	
q	Wall_Border_Corner.mesh		>	Wall_Border_T.mesh	
v	Wall_Border_Corner.mesh		F	Typechanger (Fire)	
b	Wall_Border_Corner.mesh		E	Typechanger (Earth)	
r	Wall_Border_Corner.mesh		A	Typechanger (Air)	
G	Goal		W	Typechanger (Water)	

Table 1: Entity-Tabelle

Für die meisten Objekte kann eine gemeinsame .mesh-Datei verwendet werden, welche dann vom Parser beim Einlesen entsprechend rotiert werden. Man beachte, dass auch für die im Logic-Abschnitt definierten Entitäten .mesh-Dateien vorhanden sind, welche aber von den entsprechenden Klassen geladen werden und somit nicht in Tabelle 1 aufgeführt sind.

Alle Objekte wurden mit der Modellierungs-Software Blender 2.63<sup>23</sup> erstellt und mit dem Blender2Ogre Exporter in das OGRE-Mesh-Format gebracht.

<sup>23</sup> Blender-Homepage: <http://www.blender.org/>

Da es sich bei den Level-Dateien nicht um ein binäres Format handelt, können Änderungen leicht vorgenommen werden.

```
...
Objects:
# floor 1
fffffffffffffffffffff
fffffffFffffffffffEfff
fffffffffffffffffffff
fffffff0fffffffffffff
fffffffffffffffffffff
fffffffffffffffffffff
fffffffffffffffffffff
fffffff*fffffffffffff
ffffff*ffffffffffffFfff
fffffffffffffffffffff
fsffffffffffffWfffff
fffEfffffffff0fffff
fffffffffffffffffffff
@
qQQQ$QQQ$QQQ$QQQv
R000|000|00000000V
>-0-+-0-+-M-----<
R0000000|00000000V
R0|00000|00000000V
R0C-}000|00000000V
R000|000|00000000V
R000|000N00000000V
R000|000|00000000V
R0000000|00000000V
R000|000|00000000V
rBBB!BBB!BBBBBBBb
# floor 2
%
0
0
000fffff
000f0f0f
000fffff
@
0
0
000qQQQv
000R00*V
000rBBBb
...
```

Text 10: Level-Datei: Beispiel eines Object-Abschnitts

Der Wert 0 dient dabei als Platzhalter. Es wird kein Objekt erstellt, sondern bloss die `objectCounter` Variable erhöht. Das Linefeed an jedem Zeilen-Ende sorgt dann automatisch für die Inkrementierung von `lineCounter`.

## 3.6 Plattformunabhängigkeit

Eine wichtige Vorgabe für dieses Projekt ist die Plattformunabhängigkeit der Engine. Dies, da sie simultan auf Linux und Windows entwickelt wurde. Somit wurde bereits bei der Architektur der Engine grossen Wert auf Plattformagnostik gelegt, sowohl im Code, als auch bei den verwendeten Libraries.

### 3.6.1 End-User

Da nur OpenSource-Libraries verwendet wurden, ist die Portierung und Kompilierung dieser jeweils meist ohne Probleme möglich, da bereits Anleitungen existieren.

Ein grösseres Problem ist das Management von Assets, wie Texturen, Modellen, Sounds, Shadern, usw. Dies wurde mittels des vorhandenen Ressourcen-Systems von OGRE gelöst, wobei es für einige spezielle Assets erweitert werden musste. Beispiele dazu sind das Laden von Audio-Dateien oder dem Laden eines Levels.

Durch diese Architektur wird eine Portierung auf weitere Plattformen, wie Smartphones oder Konsolen, erleichtert.

Window-Installer und Linux-Distributions-Pakete (wie .rpm oder .deb) ermöglichen die einfache Installation für den End-User. Ein solcher Installer für Windows wurde erstellt.

### 3.6.2 Entwickler

Damit auch auf verschiedenen Betriebssystemen entwickelt werden kann wurde mit Premake4 gearbeitet. Premake<sup>24</sup> ist ähnlich wie Cmake ein Tool zur automatisierten Erstellung von Makefiles und IDE-Projekten. Es arbeitet mit einem Lua-Script<sup>25</sup>.

```
...
    if os.is("linux") then
        configuration { "linux" }
        pchheader "headers/stdafx.h"
        pchsource "src/stdafx.cpp"
        libdirs {
            "/usr/lib/OGRE/", "/usr/local/lib",
            "/usr/lib/x86_64-linux-gnu/", "/usr/lib/"
        }
        includedirs {
            "/usr/include/OGRE/", "/usr/include/ois/",
            "/usr/local/include/", "/usr/local/include/bullet/",
            "/usr/local/include/bullet/ConvexDecomposition",
        }
        links {
            "OgreMain", "OgreTerrain", "OIS", "BulletDynamics",
            "BulletCollision", "LinearMath", "sfml-system",
            "sfml-audio", "pthread", "RocketCore", "RocketControls"
        }
        buildoptions { "-std=c++0x" }
    end
...
```

Text 11: Auszug aus premake4.lua

Es können unter vielem anderem Include-Pfade und Linker-Parameter angegeben werden. Auch Optionen für den Compiler können definiert werden.

<sup>24</sup> Premake: <http://industriousone.com/premake>

<sup>25</sup> Lua ist eine Programmier-Sprache (<http://www.lua.org>)

## 4 Implementation der Shader

Eine kurze Einführung zum Thema Shaders und OpenGL findet sich im Anhang dieses Dokuments (siehe Kapitel 9.1 Grundlagen)

### 4.1 Shader-Management in OGRE

OGRE bietet ein flexibles System an mit welchem Shader zur Laufzeit geladen und kompiliert werden können. Zusätzlich wurde in der Engine eine Shader-Klasse erstellt, welche diese Funktionen abstrahiert und die Möglichkeit bietet Parameter für den jeweiligen Shader anzugeben.

#### 4.1.1 Erstellen der Shader

Damit OGRE die Shader benutzen kann, müssen Materials<sup>26</sup> erstellt werden. Über diese ist es möglich beliebige Parameter zu setzen. Der erstellte Pass<sup>27</sup> wird später benutzt um den Shader zu laden.

```
material = MaterialManager::getSingletonPtr()->create(mName,
ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME);

pass = material->getTechnique(0)->getPass(0);
```

*Text 12: Erstellung eines Materials*

Der Source-Code des Shaders muss dabei in ein HighLevelGpuProgram<sup>28</sup> geladen werden. Dabei müssen der Filename, die Shadersprache sowie der Shadertyp (Vertex-, Geometry-, Fragmentsshader) angegeben werden, damit das Programm die Shader richtig kompilieren kann.

```
HighLevelGpuProgramPtr vertex =
    HighLevelGpuProgramManager::getSingletonPtr()->createProgram(
        shaderName,
        ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME,
        "glsl",
        GpuProgramType::GPT_VERTEX_PROGRAM);

vertex->setSourceFile(vertexString);
pass->setVertexProgram(vertex->getName());
```

*Text 13: Laden des Shader-Code*

Mittels der letzten Code-Zeile im oben gezeigten Programmcode wird nun dem Pass der soeben geladene Shader übergeben. Der Shader wird aber erst kompiliert, sobald er vom Programm verwendet wird, d.h. wenn der Shader effektiv gerendert werden soll.

#### 4.1.2 Parameterübergabe an die Shader

Um korrekt funktionieren zu können benötigen die meisten Shader irgend eine Art Input. In gewissen Fällen reichen bereits die Daten, welche der Shader von der OpenGL Rendering-Pipeline erhält. Jedoch ist das bei komplexeren Shadern bei weitem nicht ausreichend.

<sup>26</sup> Ogre::Material: [http://www.ogre3d.org/docs/api/html/classOgre\\_1\\_1Material.html](http://www.ogre3d.org/docs/api/html/classOgre_1_1Material.html)

<sup>27</sup> Ogre::Pass: [http://www.ogre3d.org/docs/api/html/classOgre\\_1\\_1Pass.html](http://www.ogre3d.org/docs/api/html/classOgre_1_1Pass.html)

<sup>28</sup> Ogre::HighLevelGpuProgram: [http://www.ogre3d.org/docs/api/html/classOgre\\_1\\_1HighLevelGpuProgram.html](http://www.ogre3d.org/docs/api/html/classOgre_1_1HighLevelGpuProgram.html)



Somit muss ein System gegeben sein, welches die manuelle Eingabe von Inputs für Shader erlaubt. Solch ein System ist in OGRE vorhanden. OGRE bietet von sich aus einige Parameter bereits an, welche oft verwendet werden. Solche werden als AutoConstants<sup>29</sup> bezeichnet. Über ein Enum sind diese definiert und werden entsprechend geladen.

```
this->getVertex()->setNamedAutoConstant("light",
    Ogre::GpuProgramParameters::ACT_LIGHT_POSITION,0);

this->getVertex()->setNamedAutoConstant("camera",
    Ogre::GpuProgramParameters::ACT_CAMERA_POSITION);
```

Text 14: Setzen von AutoConstants

Die zweite Möglichkeit um Shader-Parameter zu setzen sind einfache Constants. Diese werden vom Programm/Programmierer verwaltet und können zur Laufzeit jederzeit verändert werden.

```
this->getFragment()->setNamedConstantFromTime("time",Ogre::Real(1));

this->getFragment()->setNamedConstant("startColor",Ogre::ColourValue(1,1,0,1));
this->getFragment()->setNamedConstant("endColor",Ogre::ColourValue(1,0,0,1));
```

Text 15: Setzen von Constants

Dabei muss der Name des Parameters mit dem Parameternamen im Shader übereinstimmen. Als Wert können alle GLSL-Typen verwendet werden, welche ein Equivalent in OGRE besitzen. Dies trifft auf alle Typen zu.

Ein Spezialfall im oben gezeigten Beispiel ist `setNamedConstantFromTime`. Dabei handelt es sich um eine Funktion, welche dem Shader den momentanen Programmzeitpunkt übergibt und diesen immer aktuell hält.

#### 4.1.2.1 Spezialfall Texturen

Der Fragment-Shader arbeitet oftmals mit Texturen, welche ihm natürlich bekannt sein müssen. Über das Setzen von Parametern kann man ihm diese nun übergeben. Dabei gibt es aber einige Stolperfallen, welche der Programmierer beachten muss.

Um eine Textur zu nutzen, muss diese im Material des Shaders registriert werden. Dies geschieht durch die Erzeugung eines `TextureUnitStates`<sup>30</sup>.

```
void Shader::setTexture(std::string name,std::string parameterName)
{
    TextureUnitState* texture = getPass()->createTextureUnitState(name);
    texture->setTextureNameAlias(parameterName);
    texture->setName(parameterName);
}
```

Text 16: Erzeugung eines TextureUnitStates

Der Parameter `name` muss dabei der Name einer Textur sein, z.B. „cloud.png“.

<sup>29</sup> `Ogre::GpuProgramParameters::AutoConstantType`:

[http://www.ogre3d.org/docs/api/html/classOgre\\_1\\_1GpuProgramParameters.html#a155c886f15e0c10d2c33c224f0d43ce3](http://www.ogre3d.org/docs/api/html/classOgre_1_1GpuProgramParameters.html#a155c886f15e0c10d2c33c224f0d43ce3)

<sup>30</sup> `Ogre::TextureUnitState`: [http://www.ogre3d.org/docs/api/html/classOgre\\_1\\_1TextureUnitState.html](http://www.ogre3d.org/docs/api/html/classOgre_1_1TextureUnitState.html)

Die Texturen werden dabei ähnlich einem Stack im Material gespeichert und können somit über einen Index erreicht werden. Die Reihenfolge, wie die Texturen geladen werden, spielt somit eine Rolle. Um die Texturen korrekt dem Shader zu übergeben, muss diesem mitgeteilt werden, welchen Index er auf welche Texturvariable setzen muss.

```
this->setTexture("sun_color.png");
this->setTexture("ocean_floor_color.png");

this->getFragment()->setNamedConstant("LightMap",0);
this->getFragment()->setNamedConstant("GroundMap",1);
```

*Text 17: Laden von Texturen und setzen des Index in einem Shader*

Der Shader kann nun die benötigten Texturen korrekt anwenden.

### 4.1.3 Abstufungen

In drei Stufen sollten die durch die Shader erzielten Effekte kontinuierlich verbessert werden. Dafür sind die Levels Basic, Intermediate und Advanced vorgesehen. Die Implementierung konnte bis zur Intermediate-Stufe realisiert werden.

## 4.2 Erde-Shader

### 4.2.1 Basic

In dieser ersten Version des Shaders wird nur eine Textur übergeben. Dazu wird zuerst im Vertex-Shader die Farbe vom Vertex mit `gl_FrontColor = gl_Color` übernommen und somit weitergegeben. Danach wird die Variable `gl_TexCoord[0]` berechnet und dem Fragment-Shader übergeben. Dies geschieht mit dem Befehl mit der Multiplikation von der Textur-Matrix mit den Textur-Koordinaten.

Man beachte, dass man sowohl mehrere Texturen, als auch mehrere Textur-Koordinaten verwenden in einem Shader verwenden kann. Diese werden als Stack abgespeichert. Wenn nur eine Textur verwendet wird, wie im vorliegenden Fall wird diese mit der Angabe des Wertes 0 angesprochen.

Schliesslich muss noch die Position berechnet werden und in die Variable `gl_Position` geschrieben werden. Dies ist für einen Vertex-Shader zwingend erforderlich. Die Funktion `ftransform` übernimmt die Aufgabe, welche ansonsten von der Fixed-Function-Pipeline. Diese Funktion ist seit GLSL 1.40 deprecated. Anstelle dieser Funktion zu verwenden ist der Programmierer in neueren Versionen selbst dafür verantwortlich Projektion- und ModelView-Matrixen als uniforms zu übermitteln. Dies würde mit `gl_Position = projection_matrix * modelview_matrix * vec4(vertex, 1.0)` geschehen, vorausgesetzt die Applikation übermittelt dem Shader die beiden mat4 Matrizen.

```
void main()
{
    gl_FrontColor = gl_Color;
    gl_TexCoord[0] = gl_TextureMatrix[0] * gl_MultiTexCoord0;
    gl_Position    = ftransform();
}
```

*Text 18: Basic Earth Vertex Shader*

Die Textur wird im Fragment-Shader als `uniform sampler2D dirt_color` deklariert. Der Name `dirt_color` entspricht hier (der Verständlichkeit halber) auch dem Dateinamen, das muss aber nicht so sein. Schliesslich wird die Variable `gl_FragColor` mit dem Texel, einem `vec4` bestehend aus den Bildkoordinaten von `dirt_color`, sowie den Texture-Mapping Koordinaten beschrieben.

```
uniform sampler2D dirt_color;
void main()
{
    gl_FragColor = texture2D(dirt_color, gl_TexCoord[0].st);
}
```

Text 19: Basic Earth Fragment Shader

Damit die Texturierung eines Modells mit einem Shader, wie dem oben gezeigten überhaupt möglich ist wird ein UV-Koordinaten-System (UV-Mapping) verwendet, wobei `u` und `v` die Texturkoordinaten beschreiben. Dieses Koordinaten-System muss bei der Modellierung des Polyeders ebenfalls erstellt werden.

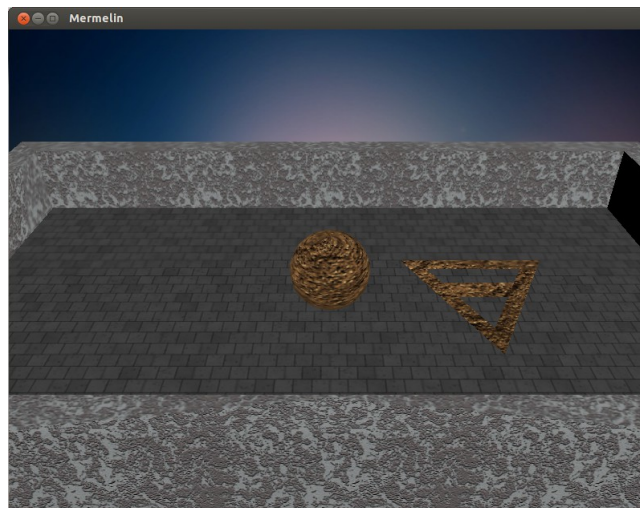


Abb. 12: Earth-Shader: Basic Stufe

## 4.2.2 Intermediate

In der zweiten, etwas fortgeschritteneren Variante des Shaders wird Normal Mapping (manchmal auch Bumpmapping genannt) angewandt, um den Eindruck zu vermitteln die Kugel sei unregelmässig. Es werden dabei in der Regel zwei Texturen verwendet; eine herkömmliche Textur mit den Farbwerten und eine zweite Textur welche die Normalen-Information enthält. Man kann entweder mit Height-Maps (in Graustufen) oder Normal-Maps (RGB) arbeiten. Die Grundidee bleibt in beiden Fällen dieselbe: Die Verfälschung der Normalen an einer bestimmten Stelle.



Aus einer glatten Oberfläche wird so eine gewölbte, unregelmässige Fläche. Diese Technik kommt oft zum Einsatz, wenn Objekte mit hoher Polygonzahl in Echtzeit gerendert werden müssen. Die Konturen des Objektes bleiben durch das Normal Mapping unverändert.



Abb. 13: Farb-Textur

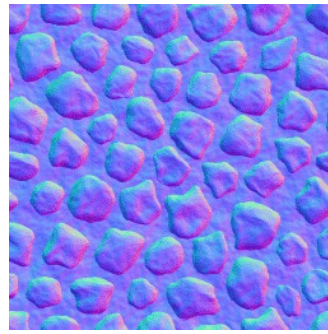


Abb. 14: Normal-Map

Im Vertex Shader werden die Vertex-Attribute `tangent`, `binormal`, `vertex` und `normal` von OGRE übermittelt und stehen dort zur Verfügung. Sie entsprechen den folgenden Vektoren<sup>31</sup>  $\mathbf{n}$ ,  $\mathbf{t}$  und  $\mathbf{b}$ :

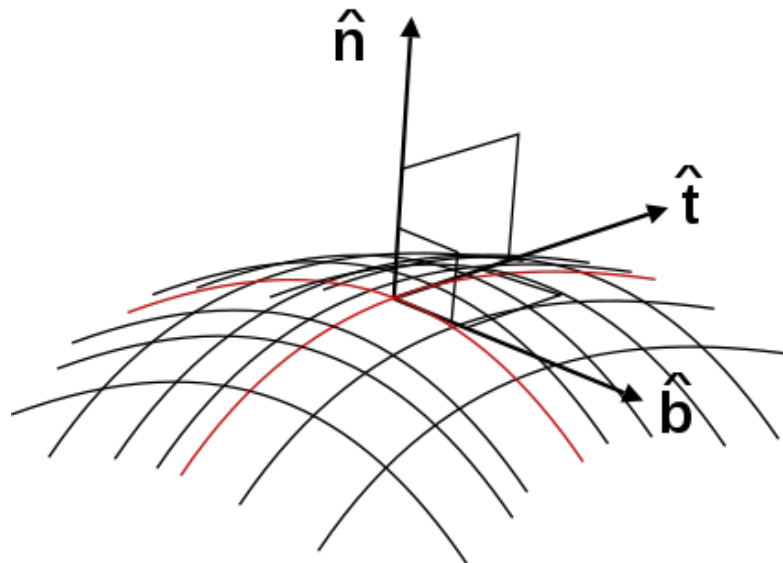


Abb. 15: Normale, Tangente und Bi-Normale eines Vertices

Das Phong-Beleuchtungsmodell<sup>32</sup> wird bei dieser Variante des Shaders angewandt. Dabei werden vereinfacht ausgedrückt drei verschiedene Beleuchtungs-Arten zueinander addiert:

- **Ambient:** Das Umgebungs-Licht oder die Grundbeleuchtung (ein konstanter Wert)
- **Diffuse:** Die diffuse Reflektion. Sie hängt vom Winkel zwischen Licht und der Oberflächen-Normalen ab. In diesem Shader die `lightDirection` Variable.
- **Specular:** Die spiegelnde Reflektion. Sie ist vom Winkel zwischen Betrachter und der Normalen der Oberflächen abhängig. Die Variable `viewDirection` wird diese Information beinhalten.

<sup>31</sup> Informationen und Bild-Quelle: [http://en.wikipedia.org/wiki/Unit\\_vector](http://en.wikipedia.org/wiki/Unit_vector)

<sup>32</sup> Siehe <http://de.wikipedia.org/wiki/Phong-Beleuchtungsmodell>

Zur Veranschaulichung ein Beispiel aus Wikipedia:

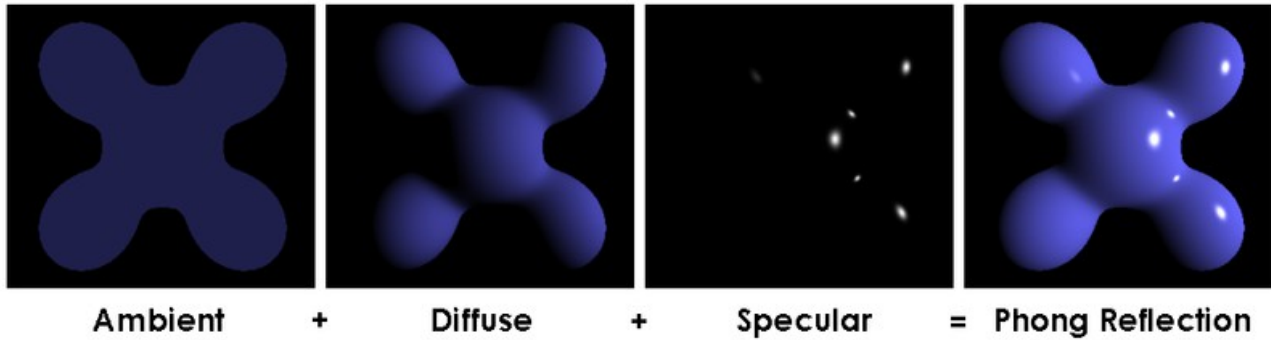


Abb. 16: Das Phong-Beleuchtungsmodell

Es wird also die Position einer Lichtquelle und der Kamera benötigt, um den Effekt zu erzielen. Beides wird hier als uniform übermittelt. Um die beiden Output-Variablen `viewDirection` und `lightDirection` zu berechnen werden nun Normale, Tangente und Binormale mit der Normalen-Matrix multipliziert. `gl_NormalMatrix` ist die inverse, transponierte Matrix, des links-oberen 3x3 Teils der Model-View-Matrix.

```
uniform vec3 light;
uniform vec3 camera;

in vec3 tangent;
in vec3 binormal;
in vec4 vertex;
in vec3 normal;

out vec2 texCoord;
out vec3 viewDirection;
out vec3 lightDirection;

void main(void)
{
    gl_Position = ftransform();
    texCoord = gl_MultiTexCoord0.st;

    vec3 n = gl_NormalMatrix * normal;
    vec3 t = gl_NormalMatrix * tangent;
    vec3 b = gl_NormalMatrix * binormal;

    vec3 object = vec3(vertex);
    mat3 rotation = mat3(t, b, n);

    lightDirection = rotation * (light - object);
    viewDirection = rotation * (camera - object);
}
```

Text 20: Intermediate Earth Vertex Shader

Im Fragment-Shader wird das Skalarprodukt aus der normalisierten `lightDirection` und dem Wert aus der Normalen-Textur wird in der Variable `dotLight` abgespeichert und entspricht dem Cosinus. Ebenso wird die Variable `dotView` berechnet, welche dem Cosinus zwischen Betrachter und Normale entspricht.

```
uniform sampler2D dirt_color;
uniform sampler2D dirt_normal;

in vec2 texCoord;
in vec3 viewDirection;
in vec3 lightDirection;

void main( void )
{
    float shininess = 2.0;

    vec3 lightDir = normalize(lightDirection);
    vec3 normal = normalize((texture2D(dirt_normal, texCoord).xyz*2.0) - 1.0);
    float dotLight = dot(normal, lightDir);

    vec3 reflection = normalize((normal * dotLight) - lightDir);
    vec3 viewDir = normalize(viewDirection);
    float dotView = max(0.0, dot(reflection, viewDir));

    vec4 color = texture2D(dirt_color, texCoord);

    vec4 ambient = gl_LightSource[0].ambient;
    vec4 diffuse = gl_LightSource[0].diffuse * dotLight;
    vec4 specular = gl_LightSource[0].specular * (pow(dotView, shininess));

    gl_FragColor = color * clamp(ambient + diffuse + specular, 0.0, 1.0) ;
}
```

Text 21: Intermediate Earth Fragment Shader

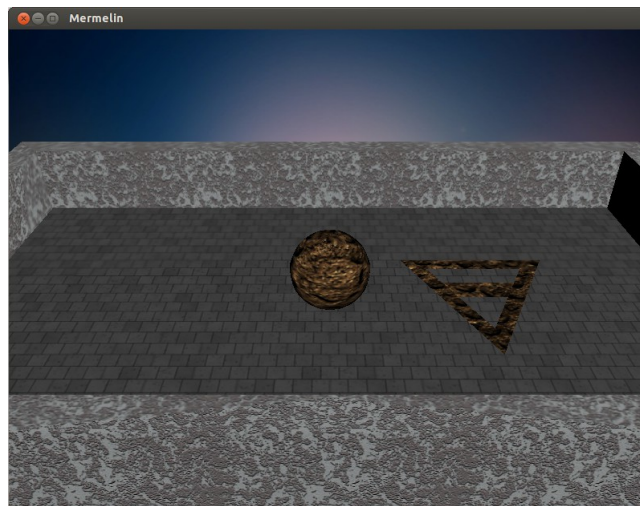


Abb. 17: Earth-Shader: Intermediate Stufe

### 4.2.3 Advanced

Eine Mögliche Weiterentwicklung des Shaders könnte darin bestehen das Abbröckeln von Material zu simulieren.



## 4.3 Feuer-Shader

### 4.3.1 Basic

Der Basic-Feuershader soll den Eindruck einer glühenden Oberfläche erzielen. Um diesen Effekt zu erreichen wird zwischen zwei Farben, mittels linearer Interpolation, gewechselt.

Im Vertex-Shader erzeugt ein Pseudo-Zufallsgenerator einen Wert zwischen 0 und 1, welcher später im Fragment-Shader als Interpolationswert verwendet wird.

```
out float noise;

float rand(vec4 co){
    return fract(sin(dot(co.xy ,vec2(12.9898,78.233))) * 43758.5453);
}

void main()
{
    gl_Position = ftransform();
    noise = rand(gl_Vertex);
}
```

Text 22: Basic Fire Vertex Shader

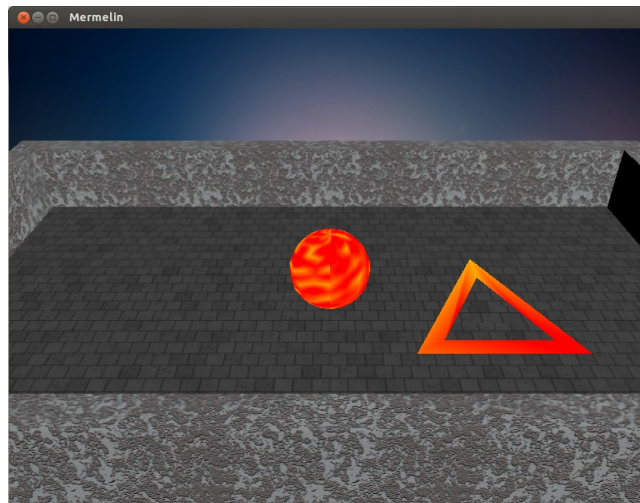


Abb. 18: Fire-Shader: Basic Stufe

Der Wert noise wird nun im im Fragment-Shader noch basierend auf der Zeit verändert, da der Wert nicht nur pro Vertex, sonder auch pro Texel variieren soll.

```
uniform vec4 startColor;
uniform vec4 endColor;
uniform float time;

in float noise;
out vec4 color;

void main()
{
    float lerpValue = abs( sin(time) + noise );
    color = mix(startColor, endColor, lerpValue);
}
```

Text 23: Basic Fire Fragment Shader



startColor und endColor werden manuell vom Programm definiert.

Die Funktion mix errechnet einen Wert zwischen den angegebenen Werten, startColor und endColor, aus, basierend auf dem lerpValue.

### 4.3.2 Intermediate

Lava ist eine fließende Masse aus geschmolzenem Gestein. Dabei gibt es bereits auf einer kleinen Fläche grosse Temperaturunterschiede. Sichtbar wird dies durch dunklere Färbung an Orten mit tieferer Temperatur.

Das Ziel des Intermediate-Fire Shader ist es, den Lava-Effekt noch zu verstärken, sowie ihn realistischer aussehen zu lassen. Hierbei wird nun auf eine Noise-Funktion verzichtet und stattdessen eine Alpha-Map verwendet. Des weiteren wird eine Textur verwendet, welche bereits einer Lavaoberfläche ähnelt.

Die Hauptarbeit des Shaders wird dabei vom Fragment-Shader übernommen.

Die Alpha-Map simuliert dabei die Temperaturunterschiede auf einer Fläche. Damit diese nicht statisch wird, werden die beiden Texturen unabhängig voneinander bewegt.

```
vec4 noise = texture2D(Texture1, Texcoord);
vec2 T1 = Texcoord + vec2(1.5, -1.5) * time * 2 * modifier;
vec2 T2 = Texcoord + vec2(-0.5, 2.0) * time * modifier;

T1.x += (noise.x) * 2.0;
T1.y += (noise.y) * 2.0;
T2.x -= (noise.y) * 0.2;
T2.y += (noise.z) * 0.2;
```

*Text 24: Intermediate Fire Fragment Shader*

Anschliessend wird der Oberfläche die Temperatur „gemessen“. Dazu werden die Farben der beiden Texturen zusammengerechnet. Dabei wird durch die Alpha-Map die Temperatur angegeben, was bedeutet, je grösser der Alpha-Wert bei einem Texel ist, desto tiefer ist die Temperatur/dunkler ist das geometrische Objekt an dieser Stelle.

```
float p = texture2D(Texture1, T1 * 2.0).a;
vec4 color = texture2D(Texture2, T2 * 2.0);
vec4 temp = color * (vec4(p,p,p,p) * 2.0) + (color * color - 0.1);
```

*Text 25: Intermediate Fire Fragment Shader, Temperaturbestimmung*

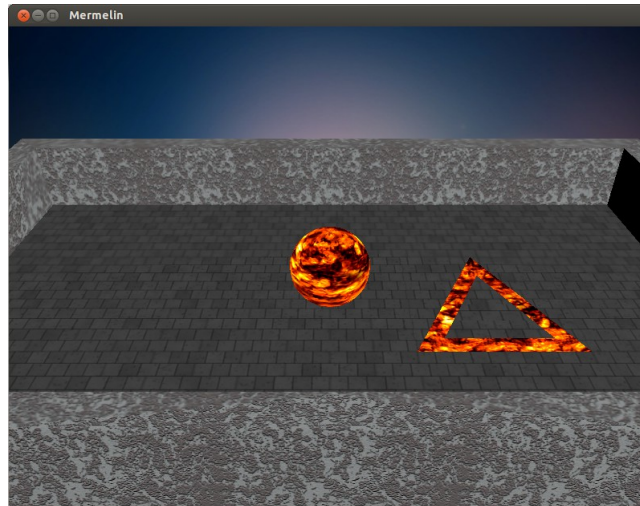


Abb. 19: Fire-Shader: Intermediate Stufe

### 4.3.3 Advanced

Das Element Feuer könnte man anstatt mit Lava auch mit Flammen realisieren, welche Rauch entwickeln. Dazu könnten Particles eingesetzt werden. Dazu wäre der Einsatz eines Geometry-Shaders sinnvoll.

## 4.4 Luft-Shader

### 4.4.1 Basic

Für den Basic-Luftshader soll ein einfacher Effekt erzielt werden, welcher die Bewegung von Wolken nachahmen soll. Dies wird erreicht durch die Rotation von 3 übereinander gezeichneten Texturen.

Die Rotation wird mittels einer Rotationsmatrix erzielt. Diese wird direkt in GLSL berechnet.

```
mat4 rotationMatrix(float Angle){
    return mat4(
        cos( Angle ), -sin( Angle ), 0.0, 0.0,
        sin( Angle ),  cos( Angle ), 0.0, 0.0,
        0.0,          0.0,          1.0, 0.0,
        0.0,          0.0,          0.0, 1.0);
}
```

Text 26: Basic Air Vertex Shader

Der Einfachheit halber wird nun dieselbe Textur drei mal verwendet. Dies spart nicht nur Speicher, es ergibt auch einen überzeugenden Effekt.

Im Fragment Shader werden nun 3 Positionen ausgelesen, wobei jeweils ein Texel von dieser Position aus der Textur gelesen werden. Zusammengerechnet ergeben sie die Farbe für das entsprechende Fragment.

```
vec4 tex1 = texture2D(clouds_color, gl_TexCoord[0].st);
vec4 tex2 = texture2D(clouds_color, gl_TexCoord[1].st);
vec4 tex3 = texture2D(clouds_color, gl_TexCoord[2].st);

gl_FragColor = tex1 * 0.25 + tex2 * 0.25 + tex3 * 0.25;
```

Text 27: Basic Air Fragment Shader

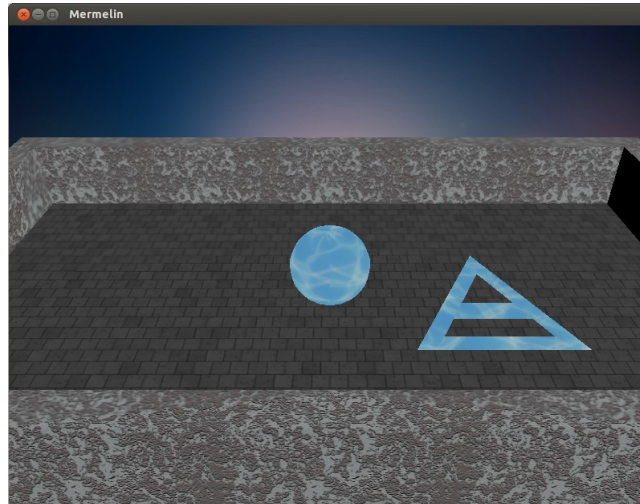
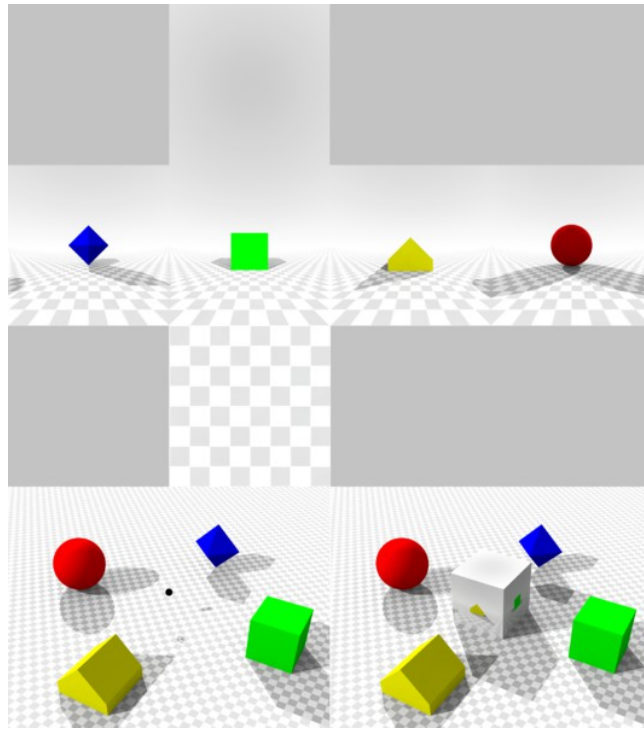


Abb. 20: Air-Shader: Basic Stufe

#### 4.4.2 Intermediate

Für die Intermediate-Stufe des Luft-Shaders wurde ein Glas-Effekt mit Spiegelung der Umgebung gewählt. Um dies zu erreichen ist dynamisches Cubemapping notwendig. Dabei wird die Umgebung in eine Textur gerendert und im Fragment-Shadert über das Objekt gespannt. OpenGL bietet dazu den Typ `samplerCube` an. Folgende Grafik aus Wikipedia zeigt, was damit gemeint ist:



*Abb. 21: Cube-Mapping*

Damit die Cube-Map nicht immer dasselbe Bild zeigt, sondern sich mit der Bewegung der Kugel anpasst, wird eine Funktion benötigt, welche die Cube-Map aktualisiert. Damit dies geschehen kann erbt die `AirShader`-Klasse zusätzlich von `Ogre::RenderTargetListener` und implementiert die Methoden `preRenderTargetUpdate` und `postRenderTargetUpdate`. Die Funktion `createCubeMap` schafft die Voraussetzungen, welche für das Cube-Mapping benötigt werden. Dazu gehört in erster Linie eine Kamera, die der Entität angehängt wird. Diese wird dann in beim Aufruf `preRenderTargetUpdate` so gedreht, dass für jede Richtung ein Bild entsteht. Zuvor wird noch das Objekt selbst unsichtbar gemacht.

```

...
void AirShader::createCubeMap()
{
    Ogre::String name = entity->getName() + Ogre::String("_cubeCam");

    if (ENGINE->getSceneManager()->hasCamera(name)) {
        cubeCam = ENGINE->getSceneManager()->getCamera(name);
    } else {
        cubeCam = ENGINE->getSceneManager()->createCamera(name);
        cubeCam->setFOVy(Degree(30));
        cubeCam->setNearClipDistance(0.1);
        entity->getNode()->attachObject(cubeCam);

        // creating the dynamic cube map texture
        TexturePtr tex = TextureManager::getSingleton().createManual("cubeMap",
            ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME,
            TEX_TYPE_CUBE_MAP, 128, 128, 0, PF_R8G8B8, TU_RENDERTARGET);

        // assigning the camera to all 6 render targets of the texture
        // (1 for each direction)
        for (unsigned int i = 0; i < 6; i++) {
            targets[i] = tex->getBuffer(i)->getRenderTarget();
            targets[i]->addViewport(cubeCam)->setOverlaysEnabled(false);
            targets[i]->addListener(this);
        }
    }
}

void AirShader::preRenderTargetUpdate(const RenderTargetEvent& evt)
{
    entity->getNode()->setVisible(false);
    cubeCam->setOrientation(Quaternion::IDENTITY);
    if (evt.source == targets[0]) cubeCam->yaw(Degree(-90));
    else if (evt.source == targets[1]) cubeCam->yaw(Degree(90));
    else if (evt.source == targets[2]) cubeCam->pitch(Degree(90));
    else if (evt.source == targets[3]) cubeCam->pitch(Degree(-90));
    else if (evt.source == targets[5]) cubeCam->yaw(Degree(180));
}

void AirShader::postRenderTargetUpdate(const RenderTargetEvent& evt)
{
    entity->getNode()->setVisible(true);
}
...

```

Text 28: Auszug aus der AirShader Klasse

```

uniform mat4 world;
uniform vec3 camera;

void main()
{
    vec4 worldSpace = world * gl_Vertex;
    vec3 worldSpaceNormal = normalize(mat3(world) * gl_Normal );
    vec3 eyeSpace = normalize(worldSpace.xyz - camera.xyz );

    gl_TexCoord[0] = gl_MultiTexCoord0;
    gl_TexCoord[1].xyz = reflect(eyeSpace, worldSpaceNormal);

    gl_Position = ftransform();
}

```

Text 29: Intermediate Air Vertex Shader

```

uniform samplerCube cubeMap;
uniform sampler2D colorMap;
uniform float time;

void main (void)
{
    float reflectivity = max(abs(sin(time)) / 1.5, 0.5);
    vec3 baseColor = texture2D(colorMap, gl_TexCoord[0].xy).rgb;
    vec3 cubeColor = textureCube(cubeMap, gl_TexCoord[1].xyz).rgb;
    gl_FragColor = vec4(mix(baseColor, cubeColor, reflectivity), 0.6);
}

```

Text 30: Intermediate Air Fragment Shader

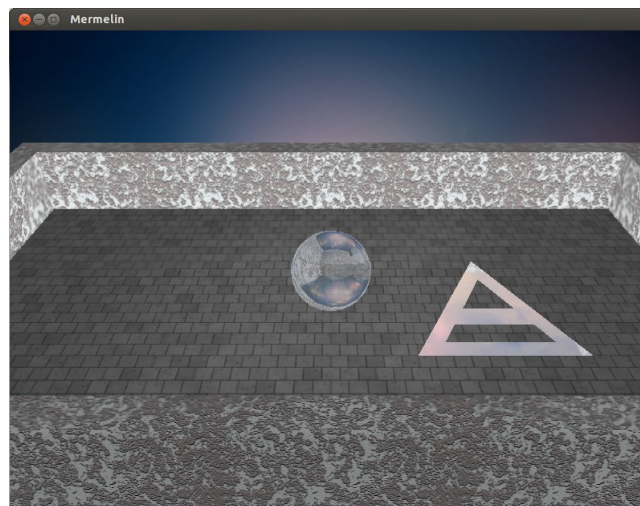


Abb. 22: Air-Shader: Intermediate Stufe

### 4.4.3 Advanced

Eine andere Idee um das Element Luft darzustellen wäre es die Kugel in eine kleine Wolke zu verwandeln.

## 4.5 Wasser-Shader

### 4.5.1 Basic

Das Ziel des Basic-Wasser Shader war einen ersten simplen aber auch überzeugenden Effekt zu erhalten. Dazu sollte die Bewegungen der Wellen an der Wasseroberfläche simuliert werden.

Dies wurde einfach durch die Überlagerung von zwei Texturen erreicht.

Im Vertex-Shader wird die Bewegung der Wasseroberfläche berechnet.

```
// simple vertex shader
#version 150 compatibility

uniform float time;
in vec2 waveSpeed;
out vec3 vTexCoord;

void main()
{
    //get multitexturing coords
    gl_TexCoord[0] = gl_MultiTexCoord0;
    gl_TexCoord[1] = gl_MultiTexCoord1;

    //Move the water
    gl_TexCoord[0].x += time * waveSpeed.x;
    gl_TexCoord[0].y += time * waveSpeed.y;

    gl_TexCoord[1].x -= time * waveSpeed.x;
    gl_TexCoord[1].y -= time * waveSpeed.y;

    gl_Position = ftransform();
}
```

*Text 31: Vertex Water Basic Shader*

Die Texturkoordinate des aktuellen Vertex ist in `gl_TexCoord[n]` gespeichert, wobei `gl_TexCoord[n]` sowohl im Vertex- als auch im Fragmentshader verfügbar ist. Dies ist nötig für die Texturierung mit mehreren Texturen.

Somit ist auch der nächste GLSL Befehl wenig überraschend, `gl_MultiTexCoord<n>`. Werden Texturkoordinaten durch OpenGL geschleust, so werden diese von OpenGL auf die jeweils zuständige `gl_MultiTexCoord<n>` gelegt.

Die Texturkoordinate wird anschliessend durch eine einfache Addierung bewegt, wodurch der Welleneffekt entsteht. Der Fragment Shader greift darauf hin auf diese Koordinaten zu.



```

#version 150 compatibility

uniform sampler2D waveTextureId;
uniform sampler2D waveTextureIdRef;

uniform float time;
in vec3 vTexCoord;

void main()
{
    vec4 color1 = texture2D(waveTextureId, vec2(gl_TexCoord[0]));
    vec4 color2 = texture2D(waveTextureIdRef, vec2(vTexCoord));

    gl_FragColor = 0.6 * vec4(color1 + color2) * vec4(0.0, 1.0, 1.0, 0.50);
}

```

Text 32: Fragment Water Basic Shader

Die finale Farbe für das entsprechende Pixel wird in der letzten Zeile gesetzt, indem die Farben aus den Texturen übereinander gelegt werden.

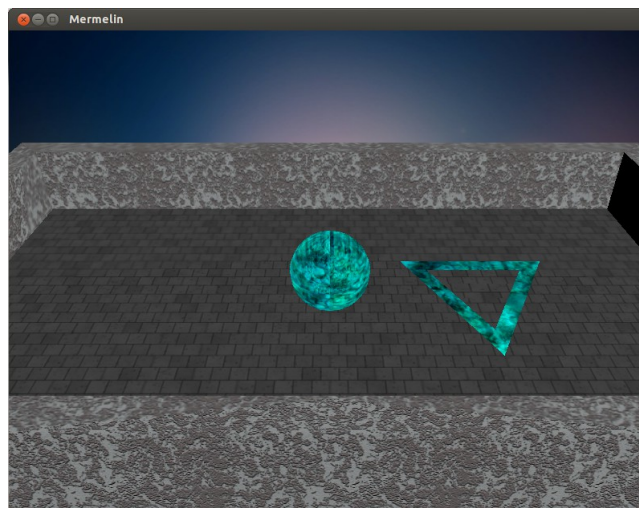


Abb. 23: Water-Shader: Basic Stufe

## 4.5.2 Intermediate

Am Strand oder am Ufer eines Sees kann man, bei richtigem Lichteinfall ein Phänomen names Diakaustik<sup>33</sup> beobachten. Diese Lichtstreifen entstehen durch die Brechung des Lichts, wobei die Wellen auf der Wasseroberfläche die Linse bilden.



Abb. 24: Diakaustik im Fischbecken

<sup>33</sup> Diakaustik: [http://de.wikipedia.org/w/index.php?title=Datei:A\\_fish\\_and\\_caustic.jpg&filetimestamp=20070921012749](http://de.wikipedia.org/w/index.php?title=Datei:A_fish_and_caustic.jpg&filetimestamp=20070921012749)



Es gibt verschiedene Verfahren um solche Kaustiken in Echtzeit zu zeichnen. Ein Ansatz dazu wurde für den Intermediate-Shader ausgewählt.

Um die Kaustiken korrekt zu berechnen, müsste jedes einzelne Photon von seiner Entstehung bis zu seinem Auftreffen mit dem Boden berechnet werden. Dies ist in einer Echtzeit-Anwendung mit den bestehenden Ressourcen nicht machbar. Daher werden für den folgenden Ansatz einige Annahmen getroffen:

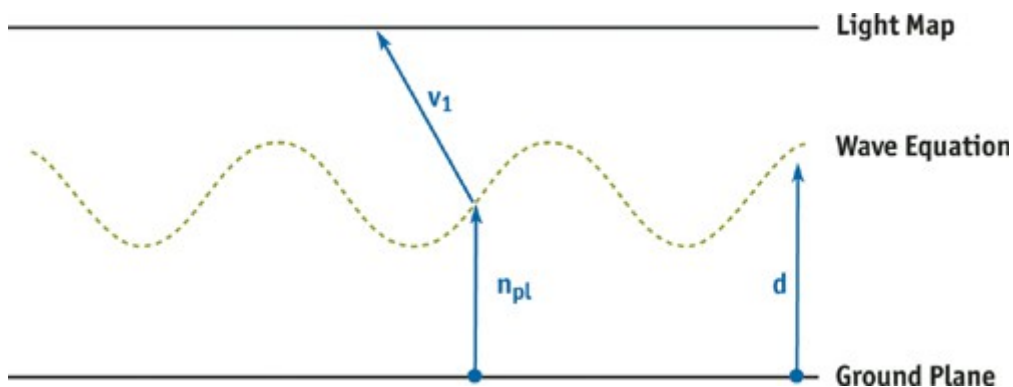
- Die Berechnung der Kaustiken findet am Erd-Äquator um 12 Uhr mittags statt. Dies bedeutet, dass die Sonne jederzeit komplett senkrecht über der zu berechnenden Fläche steht.
- Nur Sonnenstrahlen welche vertikal auf die Wasseroberfläche und auf den Boden treffen werden berechnet. Dies ist eine wichtige Annahme für den Erfolg des Algorithmus

Zusammengefasst arbeitet der Algorithmus folgendermassen:

1. Der Grund wird gezeichnet
2. Mit einer Wellenfunktion wird die Position und Intensität der Kaustiken errechnet.

Der Effekt wird somit folgendermassen erreicht:

An einem Punkt X, Y wird die Normale der Wellenfunktion, zum Zeitpunkt t, berechnet. Der daraus resultierende Strahl wird dabei auf die Light-Map geworfen und es wird entsprechend der Intensität der Light-Map die Kaustik bestimmt.



Dazu werden 2 grundsätzliche Funktionen benötigt. Eine um die Wellenfunktion zu berechnen und eine um den Schnittpunkt auf der Light-Map zu bestimmen.

```

vec2 gradwave(float x, float y, float timer)
{
    float dZx = 0.0f;
    float dZy = 0.0f;
    float octaves = OCTAVES;
    float factor = FACTOR;
    float d = sqrt(x * x + y * y);

    do {
        dZx += d * sin(timer * SPEED + (1/factor) * x * y * WAVESIZE) *
                y * WAVESIZE - factor *
                cos(timer * SPEED + (1/factor) * x * y * WAVESIZE) * x/d;
        dZy += d * sin(timer * SPEED + (1/factor) * x * y * WAVESIZE) *
                x * WAVESIZE - factor *
                cos(timer * SPEED + (1/factor) * x * y * WAVESIZE) * y/d;

        factor = factor/2;
        octaves--;
    } while (octaves > 0);

    return vec2(2 * VTXSIZE * dZx, 2 * VTXSIZE * dZy);
}

```

Text 33: Wellenfunktion

Die Funktion um die Welle zu berechnen ist oben dargestellt.

Hat man nun die Normale an einem Punkt auf der Welle erhalten ist als nächstes der Schnittpunkt auf der Light-Map an der Reihe. Dazu wird folgende Funktion benötigt:

```

vec3 line_plane_intercept(vec3 lineP, vec3 lineN, vec3 planeN, float planeD)
{
    float distance = (planeD - lineP.z) / lineN.z;
    return lineP + lineN * distance;
}

```

Es ist lineP die Position des aktuellen Vertex, lineN die Normale auf der Welle, planeN stellt den Boden dar, und planeD die Distanz des Bodens zur Wasseroberfläche. Somit wird ein Punkt errechnet, welcher auf der Light-Map anschliessend genutzt wird um die Intensität auszulesen.

Zusammengenommen sieht die main-Funktion des Shaders dann folgendermassen aus:

```

void main( void )
{
    vec2 dxdy = gradwave(vert.x, vert.y, time);
    vec4 vertex = vert;
    vertex.w = clamp(vertex.w, 0.0, 1.0);
    vec3 saturated = vec3(dxdy.x, dxdy.y, vertex.w);
    vec3 intercept = line_plane_intercept(vertex.xyz, saturated,
                                          vec3(0, 0, 1), -0.8);
    colour = texture(LightMap, intercept.xy * 0.8);
    colour += texture(GroundMap, vertex.xy);
    colour.a = 1;
}

```

Text 34: Fragment Water Intermediate Shader, Main-Funktion

Die Position der Welle wird in  $dx dy$  gespeichert. Der Schnittpunkt der Welle mit dem Lichtstrahl wird in die Variable `intercept` eingelesen. Mit dieser wird auf die Textur `Light-Map` zugegriffen, wobei eine Intensität ausgelesen wird. Diese wird nun einfach mit der `GroundMap`, also die Textur des Bodens, addiert, wodurch der Effekt der Kaustik zusammenkommt.

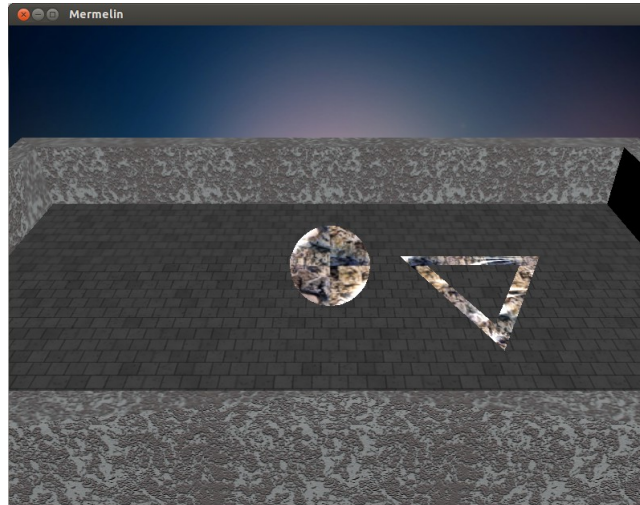


Abb. 25: Water-Shader: Intermediate Stufe

### 4.5.3 Advanced

Eine Erweiterung für diesen Shader wären noch sogenannte God-Rays, welche die Kaustiken nicht nur auf der Kugel, sondern auch noch auf den darunterliegenden Boden werfen.

## 4.6 Rotation-Shader

Dieser Shader ermöglicht es ein Objekt um eine der drei Hauptachsen zu rotieren. Er kommt bei der Rotor Entität zum Einsatz. Die Funktionsweise ist dieselbe wie beim Basic-Air Shader. Es wird eine Textur um eine Achse gedreht. Es gibt jedoch noch einige zusätzliche Parameter, welche eingestellt werden können. Diese sind Geschwindigkeit und Richtung einerseits (negative Werte lassen in die Gegenrichtung rotieren), sowie die Hauptachse andererseits.

## 5 Testing

Da die Cotopaxi-Engine auf vielen verschiedenen Bibliotheken aufbaut, hängt die Stabilität auch stark von diesen ab. Ein Fehler in einer der Libraries wirkt sich automatisch auf das implementierende Modul und somit auf die Engine aus. Im Rahmen dieser Arbeit ist es unmöglich gewesen alle benötigten Funktionen geschweige denn auf allen unterstützten Plattformen zu testen.

### 5.1 Use-Cases Spiel

Daher wurde der Ansatz gewählt eine kleine Anzahl von Use-Cases zu definieren und diese zu prüfen. Folgende Anwendungsfälle wurden für das Spiel definiert:

Nr.	Use-Case	Beschreibung / Indikatoren
1	Laden des Levels aus einer Text-Datei	Prüfung der Anzahl der erzeugten Objekte mittels eines Test-Levels.
2	Verwendung eines fehlerhaftes Levels	Exceptions müssen in bei falscher Syntax geworfen werden.
3	Wechsel der Zustände	Bestimmte Ereignisse versetzen die Engine in bestimmte States. Diese gilt es zu prüfen. Durch ein simuliertes Spiel kann festgestellt werden, ob die Engine sich jeweils im korrekten Zustand befindet.
4	Sturz der Kugel in den Abgrund	Stürzt die Kugel ab, so muss sie auf dem Startpunkt repositioniert werden. Der Startpunkt darf nicht im leeren Raum liegen.

Table 2: Use-Cases Tabelle

### 5.2 Shader-Testing

Die Shader wurden, wo es möglich war auch mit dem Programm ShaderMaker<sup>34</sup> visuell geprüft. Der Effekt sollte sowohl im Spiel, als auch im Tool gleich sein.

### 5.3 Log-Files

Vielerorts im Engine-Code werden Warnungen und Fehlermeldungen ins OGRE-Logfile geschrieben. Dieses Logfiles wurden bei Bedarf konsultiert um Fehler zu beheben, oder Optimierungsmassnahmen zu bestimmen.

<sup>34</sup> ShaderMaker: [http://cg.in.tu-clausthal.de/teaching/shader\\_maker/index.shtml](http://cg.in.tu-clausthal.de/teaching/shader_maker/index.shtml)

## 6 Projektmanagement

Die Planung des Projektes ist im Pflichtenheft (Kapitel 6 und folgende) definiert worden. Nachfolgend wird darauf eingegangen was davon umgesetzt werden konnte und ob der geplante Zeitplan eingehalten werden konnte.

### 6.1 Erfüllung der Anforderungen

Anforderung	Erfüllung in %
<i>Höchste Priorität (Muss-Kriterien)</i>	
Die definierten Shader müssen mindestens in der Stufe Intermediate mit GLSL implementiert werden.	100 %
Mindestens ein spielbares Level muss vorhanden sein.	100 %
Levels müssen editierbar sein	100 %
Tastatur als Eingabegerät	100 %
Soundeffekte und Audio	100 %
<i>Mittlere Priorität (Soll-Kriterien)</i>	
Menü: <i>Es gibt ein Menu, wobei es eher bescheiden ausgefallen ist.</i>	75 %
Verstellbare Optionen: <i>Optionen lassen sich über Dateien (der OGRE Library) einstellen.</i>	25 %
Game-Story: <i>Background-Geschichte vorhanden.</i>	40 %
<i>Tiefe Priorität (Kann-Kriterien)</i>	
Advanced-Shader implementieren: <i>Aus Zeitgründen nicht implementiert</i>	0 %
Portierung auf Mobile Devices: <i>Erste Versuche das Spiel in XCode auf dem iPhone Emulator zum Laufen zu bringen</i>	10 %
Level-Editor: <i>Ein Level-Editor der auf Swing (Java) aufbaut war geplant</i>	0 %

Table 3: Erfüllung der Anforderungen

## 6.2 Zeitplan

Die folgende Tabelle zeigt auf wo, dass Zeit-Margen vorhanden waren und welche Schritte länger dauerten als geplant war.

Tätigkeit	Start	Ende	Einhaltung
1. Pflichtenheft erstellen	21.09.2012	28.10.2012	Zeitplan eingehalten
2. Design-Dokument	21.09.2012	28.09.2012	Verspätet
3. Einrichtung der IDE / Repository. Teilweise Übernahme aus Vorprojekt.	21.09.2012	28.09.2012	Vor dem Zeitplan
4. Doxygen Lektüre und Installation	21.09.2012	27.09.2012	Vor dem Zeitplan
5. Wichtigste 3D Modelle erstellen	29.09.2012	05.10.2012	Vor dem Zeitplan
6. Spezifikation des Level-Formats	29.09.2012	05.10.2012	Zeitplan eingehalten
7. Implementation des Level-Parsers	06.10.2012	12.10.2012	Zeitplan eingehalten
8. Definition der 4 Haupt-Shader	13.10.2012	18.10.2012	Zeitplan eingehalten
9. Implementation der Shader auf Stufe Basic	19.10.2012	02.11.2012	Leicht verspätet
10. Implementation der Shader auf Stufe Intermediate	05.11.2012	14.12.2012	Verspätet
11. Gameplay-Element: Gitter	19.11.2012	30.11.2012	Nicht umgesetzt
12. Gameplay-Element: Brennbare Wände	03.12.2012	14.12.2012	Zeitplan eingehalten
13. Soll-Kriterien	17.12.2012	11.01.2013	Zeitplan eingehalten
14. Kann-Kriterien (wenn Zeit vorhanden ist)	11.12.2012	18.01.2013	Nicht umgesetzt
15. Überarbeiten der Dokumentation	29.10.2012	18.01.2013	Verspätet

Table 4: Zeitplan

## 6.3 Zusammenarbeit und Tools

Für die Zusammenarbeit haben sich die Autoren mindestens einmal pro Woche zusammengefunden und gemeinsam die weiteren Schritte besprochen und auch gemeinsam entwickelt. Dabei wurden die folgenden Werkzeuge verwendet:

- GIT als Repository
- Premake4 für die Erstellung von Makefiles und Solutions
- LibreOffice 3.6 für die Erstellung dieses Dokumentes
- Blender 2.63 zur Modellierung
- Gimp 2.8 für die Bearbeitung von Grafiken
- Audacity für die Sound-Dateien
- Netbeans 7.2 unter Linux, Visual Studio 2010 unter Windows
- Doxygen zur Generierung der Code-Dokumentation

## 7 Fazit

---

Das Einbinden der unterschiedlichen Libraries in die Engine war ein zeitaufwändiges Unterfangen. Es wurde mehr Zeit dafür beansprucht, als ursprünglich geplant war.

Unter anderem mussten die folgenden Herausforderungen bewältigt werden:

- Physik- und Kollisionserkennung
- Plattformunabhängigkeit (Unterschiedliche Compilers)
- Erstellung der Shaders (Parameterübergabe)
- Asset-Loading (Audio, GUI, Level)

Manchmal waren einzelne Code-Zeilen oder sogar ein einzelner falscher Parameter in einer Funktion die Ursache für mehrere Tage der Recherche und der Fehlersuche.

Auch wenn hie und da in gewissen Details etwas von der geplanten Umsetzung abgewichen wurde, so ist doch die Grund-Architektur seit dem Projekt 2 unverändert geblieben. Das damals gewählte Design hat sich gut bewährt.

In den Bereichen C++ Programmierung, sowie 3D-Modellierung konnte viel Erfahrung gesammelt werden.

### 7.1 Ausblick

Die Weiterentwicklung des Projekts nach Abschluss der Thesis ist vorgesehen. Aus zeitlichen Gründen konnten bei weitem nicht alle Ideen umgesetzt werden. Die Cotopaxi-Engine und Mermelin haben noch viel Potential. Dabei stehen folgende Erweiterungen im Vordergrund:

#### 7.1.1 Mermelin

- Erstellung von vielen kniffligen Levels
- Ein Intro-Video, Zwischensequenzen, sowie ein finales Video
- Verletzbarkeit der Kugel mittels eines Gesundheits-Zustandes
- Neue Objekte
  - Wasser
  - Gegner
  - Sich bewegende, schwebende Böden
  - Stacheln
  - Eis

### 7.1.2 Cotopaxi-Engine

- AI-Modul für Gegner
- Erhöhung der Konfigurierbarkeit
  - Tasten können vom Benutzer selber gewählt werden
  - Auflösung und Detailierungsgrad via Menu möglich
- Unterstützung weiterer Input-Geräte
- Überarbeitung des Level-Formats
  - Individuelle Hintergrund-Musik pro Level
  - Hintergrundbild pro Level definierbar
  - Themen (Gruppierung von Materials und Texturen)
  - Bessere Prüfung der Level-Dateien
- Level-Editor
  - Einfache Version mit Qt oder mit Java
  - WYSIWYG-Version direkt in der Engine
- Scores und High-Score-Liste (später online)
- Portierung auf iOS und Android
- Netzwerk
  - Globale High-Scores
  - Multiplayer
- Verbesserung der Shader
  - Stufe Advanced Implementieren
  - Wahl des Shader-Levels auf Grund der vorhandenen Hardware
  - Schatten mit Shader darstellen und nicht mehr über die Fixed-Function-Pipeline
- Wetter (Regen, Sonnenschein)



## 8 Quellen und Verzeichnisse

---

### 8.1 Literatur

#### 8.1.1 OpenGL und Shader

Urs Künzler

**CG- CG -06 - GPU Programming Version 2.0**, 2012

Berner Fachhochschule Technik und Informatik, Biel, Schweiz

Ricardo Marroquim, André Maximo

**Introduction to GPU Programming with GLSL**, 2009

Istituto di Scienza e Tecnologie dell'Informazione CNR Pisa, Italien

Laboratório de Computação Gráfica COPPE – UFRJ, Rio de Janeiro, Brasilien

<https://code.google.com/p/glsl-intro-shaders/>

John Kessenich, LunarG

**The OpenGL Shading Language**, 2012

<http://www.opengl.org/registry/doc/GLSLangSpec.4.30.6.pdf>

Mark Segal, Kurt Akeley

**OpenGL Core Specification 4.3**, 2012

<http://www.opengl.org/registry/doc/glspec43.core.20120806.pdf>

Dave Shreiner, Mason Woo , Jackie Neider, Tom Davis

**OpenGL Programming Guide** , 6. Edition , 2007

Addison-Wesley, ISBN-13 978-0-321-48100-9

Randi J. Rost , Bill Licea-Kane

**OpenGL Shading Language** , 3. Edition , 2009

Addison-Wesley, ISBN 978-0-321-63763-5

David Wolff

**OpenGL 4.0 Shading Language Cookbook**, 2011

Packt Publishing Ltd, ISBN 978-1-849514-76-7

Mike Bailey

**Tessellation Shaders, GLSL Geometry Shaders, OpenGL Compute Shaders**, 2012

Oregon State University, Corvalis, USA

<http://web.engr.oregonstate.edu/~mjb/>

### 8.1.2 Engine-Design und OGRE

Jason Gregory (bearbeitet von Jeff Lander und Matt Whiting)

**Game Engine Architecture**, 2011

A K Peters Ltd., ISBN 978-1-568814-13-1

Felix Kerger

**OGRE 3D 1.7 Beginner's Guide**, 2010

Packt Publishing Ltd, ISBN 978-1-849512-48-0

Ilya Grinblat, Alex Peterson

**OGRE 3D 1.7 Application Development Cookbook**, 2012

Packt Publishing Ltd, ISBN 978-1-84951-456-9

Gregory Junker

**Pro OGRE 3D Programming**, 2006

Apress, ISBN 978-1-59059-710-1

### 8.1.3 Mathematik und Game-Design Planung

Fletcher Dunn

**3D Math Primer for Graphics and Game Development**, 2002

Wordware Publishing, ISBN 978-1556229114

Jesse Schell

**The Art of Game Design; A book of Lenses**, 2008

Elsevier Inc, ISBN 978-0-12-369496-6

## 8.2 Web-Ressourcen

### 8.2.1 OpenGL

Song Ho Ahn

#### **OpenGL**

<http://www.songho.ca/opengl/index.html>  
(abgerufen am 10.01.2013)

Egon Rath

#### **OpenGL & 3D Graphics**

<http://www.matrix44.net/cms/notes/opengl-3d-graphics/coordinate-systems-in-opengl>  
(abgerufen am 10.01.2013)

## 8.3 Abbildungen

Abb. 1: Endstand des Projekts 2.....	5
Abb. 2: Kugellabyrinth.....	6
Abb. 3: D&D Dungeon Kacheln.....	8
Abb. 4: 1. Feuer 2. Erde 3. Wasser 4. Luft.....	9
Abb. 5: Architektur der Anwendung.....	11
Abb. 6: Das OGRE-Koordinatensystem.....	13
Abb. 7: SceneGraph Beispiel.....	13
Abb. 8: Graphische Darstellung der Physikpipeline.....	17
Abb. 9: GUI von Mermelin.....	20
Abb. 10: Erstellung der Komponenten.....	21
Abb. 11: Entfernung einer Komponente.....	22
Abb. 12: Earth-Shader: Basic Stufe.....	32
Abb. 13: Farb-Textur.....	33
Abb. 14: Normal-Map.....	33
Abb. 15: Normale, Tangente und Bi-Normale eines Vertexes.....	33
Abb. 16: Das Phong-Beleuchtungsmodell.....	34
Abb. 17: Earth-Shader: Intermediate Stufe.....	35
Abb. 18: Fire-Shader: Basic Stufe.....	36
Abb. 19: Fire-Shader: Intermediate Stufe.....	38
Abb. 20: Air-Shader: Basic Stufe.....	39
Abb. 21: Cube-Mapping.....	40
Abb. 22: Air-Shader: Intermediate Stufe.....	42
Abb. 23: Water-Shader: Basic Stufe.....	44
Abb. 24: Diakustik im Fischbecken.....	45
Abb. 25: Water-Shader: Intermediate Stufe.....	47
Abb. 26: Logo der OpenGL.....	58
Abb. 27: Linke-, bzw Rechte-Hand Regel.....	59
Abb. 28: Graphische Darstellung der Transformationsmatrizen.....	59
Abb. 29: Würfel im Object Space.....	60
Abb. 30: Darstellung der ModelView Matrix.....	61
Abb. 31: View-Frustum.....	62
Abb. 32: Variablen des Vertex-Shaders.....	67
Abb. 33: Einfluss der Tessellation auf Kanten und Flächen.....	68
Abb. 34: OpenGL Primitiven.....	69
Abb. 35: Variablen des Fragment-Shaders.....	71

## 8.4 Tabellen

Table 1: Entity-Tabelle.....	26
Table 2: Use-Cases Tabelle.....	48
Table 3: Erfüllung der Anforderungen.....	49
Table 4: Zeitplan.....	50
Table 5: Übersicht über die Rendering-Pipeline.....	63
Table 6: Übersicht OpenGL/GLSL Versionen.....	64
Table 7: Swizzeling-Tabelle.....	65
Table 8: Qualifiers-Tabelle.....	65
Table 9: Adjazenz-Primitiven.....	70

## 8.5 Code-Texte

Text 1: InputModule: Beenden des Spiels.....	14
Text 2: InputModule: Umschalten zwischen Running- und Debug-Modus.....	15
Text 3: Audio: Laden einer Audio-Datei vom File-System.....	15
Text 4: AudioModul: Aktualisieren des globalen Listeners.....	16
Text 5: PhysicsModule: Aktualisieren der physikalischen Welt.....	17
Text 6: PhysicsModul: Abarbeitung der ContactManifolds.....	18
Text 7: PhysicsModule: Abarbeitung der Ghost-Objekts.....	18
Text 8: GhostComponent: Pseudo-Code zur Kollisionsbearbeitung.....	19
Text 9: Level-Datei: Beispiel eines Logik-Abschnitts.....	24
Text 10: Level-Datei: Beispiel eines Object-Abschnitts.....	27
Text 11: Auszug aus premake4.lua.....	28
Text 12: Erstellung eines Materials.....	29
Text 13: Laden des Shader-Code.....	29
Text 14: Setzen von AutoConstants.....	30
Text 15: Setzen von Constants.....	30
Text 16: Erzeugung eines TextureUnitStates.....	30
Text 17: Laden von Texturen und setzen des Index in einem Shader.....	31
Text 18: Basic Earth Vertex Shader.....	31
Text 19: Basic Earth Fragment Shader.....	32
Text 20: Intermediate Earth Vertex Shader.....	34
Text 21: Intermediate Earth Fragment Shader.....	35
Text 22: Basic Fire Vertex Shader.....	36
Text 23: Basic Fire Fragment Shader.....	36
Text 24: Intermediate Fire Fragment Shader.....	37
Text 25: Intermediate Fire Fragment Shader, Temperaturbestimmung.....	37
Text 26: Basic Air Vertex Shader.....	38
Text 27: Basic Air Fragment Shader.....	39
Text 28: Auszug aus der AirShader Klasse.....	41
Text 29: Intermediate Air Vertex Shader.....	42
Text 30: Intermediate Air Fragment Shader.....	42
Text 31: Vertex Water Basic Shader.....	43
Text 32: Fragment Water Basic Shader.....	44
Text 33: Wellenfunktion.....	46
Text 34: Fragment Water Intermediate Shader, Main-Funktion.....	46

## 9 Anhang

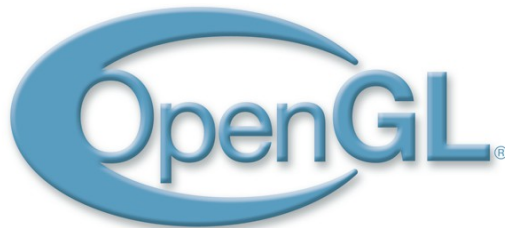
---

Es folgt noch eine kurze Einführung in die für die vorliegende Arbeit verwendeten Grundlagen. Des weiteren dienen dieser Arbeit zwei weitere, unabhängige Dokumente als Anhang. Es handelt sich dabei um

- Pflichtenheft
- Code-Dokumentation (mit Doxygen aus dem Quellcode generiert)

### 9.1 Grundlagen

#### 9.1.1 OpenGL



*Abb. 26: Logo der OpenGL*

Von der Mehrheit der Spiele-Konsolen und Smartphones verwendet, ist diese Bibliothek gegenwärtig nicht mehr wegzudenken: Die Open Graphics Library, kurz OpenGL. Seit der Einführung vor zwei Jahrzehnten ist sie zur führenden Umgebung für die Entwicklung interaktiver 2D- und 3D-Grafik-Anwendungen avanciert. OpenGL ist weitgehend unabhängig vom Betriebssystem und der Programmiersprache. Dies ist wohl der wichtigste Vorteil gegenüber dem mittlerweile einzigen Konkurrenten Direct3D, welcher Bestandteil von Microsoft DirectX ist.

Damit OpenGL verwendet werden kann, benötigt es je nach Betriebssystem ein unterschiedliches Interface, welches die OpenGL Spezifikationen implementiert. Auf den traditionellen Computern sind dies GLX (OpenGL Extension to the X Window System) bei Unix/Linux, WGL bei Windows und CGL (Core OpenGL) bei Mac. Fenstersystem-unabhängig kann auch GLUT (OpenGL Utility Toolkit), Freeglut, SDL (Simple Direct Media Layer), oder ein anderes Windowing-Toolkit, welches auf den vorgängig erwähnten Implementationen aufbaut, eingesetzt werden.

#### 9.1.2 Mathematische Begriffe in OpenGL

Zur Berechnung von Bildern am Computer (Rendering) ist im Hintergrund viel Mathematik notwendig. Im Speziellen bei der Verwendung von Shadern sind gute Kenntnisse in Vektor- und Matrizenrechnung erforderlich.

### 9.1.2.1 Koordinatensysteme

OpenGL benutzt ein kartesisches Koordinatensystem mit 3 Achsen, wobei dieses ein Rechtssystem ist. Dies ist einfach mit der rechten Hand zu veranschaulichen. Streckt man Daumen, Zeige- und Ringfinger aus, formt man dadurch automatisch das entsprechende Koordinatensystem. Der Daumen stellt dabei die x-Achse, der Zeigefinger die y-Achse und der Ringfinger die z-Achse dar.

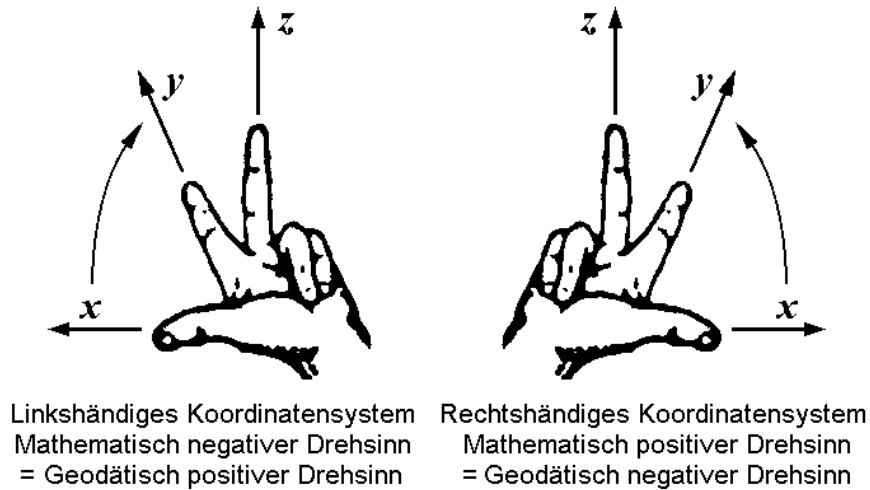


Abb. 27: Linke-, bzw Rechte-Hand Regel

Die normale mathematische Darstellung zeigt meist die Z-Koordinate welche nach oben zeigt. In OpenGL ist es jedoch die Y-Koordinate welche die Vertikale Achse belegt. Die Z-Achse erstreckt sich dabei zur Kamera (zum Bildschirm) hin. Dies ist ebenfalls mit der rechten Hand darstellbar, man zeigt einfach mit dem Zeigefinger nach oben und lässt den Ringfinger nach vorne zeigen.

Innerhalb von OpenGL wird nun mit verschiedenen Koordinatensystemen gerechnet. Jedes dieser Koordinatensysteme trägt andere Informationen, welche schlussendlich von der Rederingpipeline für die korrekte Platzierung der gewünschten geometrischen Objekte benötigt werden.

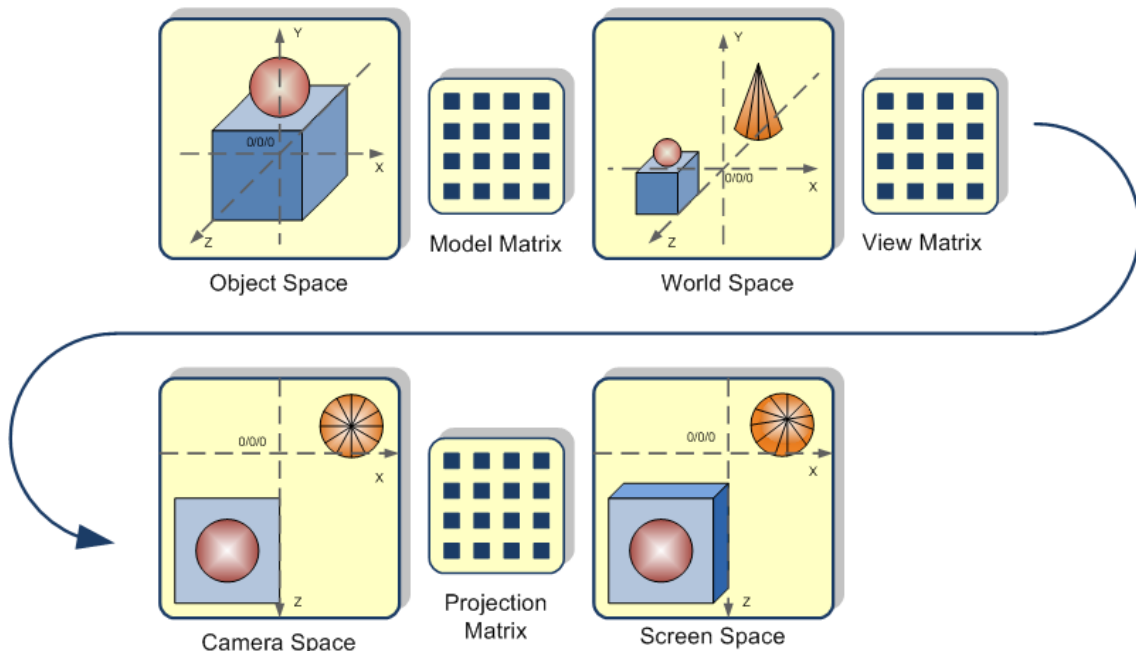


Abb. 28: Graphische Darstellung der Transformationsmatrizen

Die einzelnen Stadien sollen hier in kürze beschrieben werden.

#### 9.1.2.1.1 Object Space und Model Matrix

Der Object Space ist das lokale Koordinatensystem jedes Geometrischen Objekts. Jedes geometrische Objekt (Würfel, Kugel, Türe, Menschen) werden in einer 3D-Grafiksoftware, erstellt. Die Koordinaten jedes Vertex werden dabei im Object Space abgespeichert, also relativ zum Nullpunkt des Koordinatensystems des jeweiligen 3D-Models.

Stellt man sich einen Würfel vor, so hat jeder seiner 8 Vertices eine relative Koordinate.

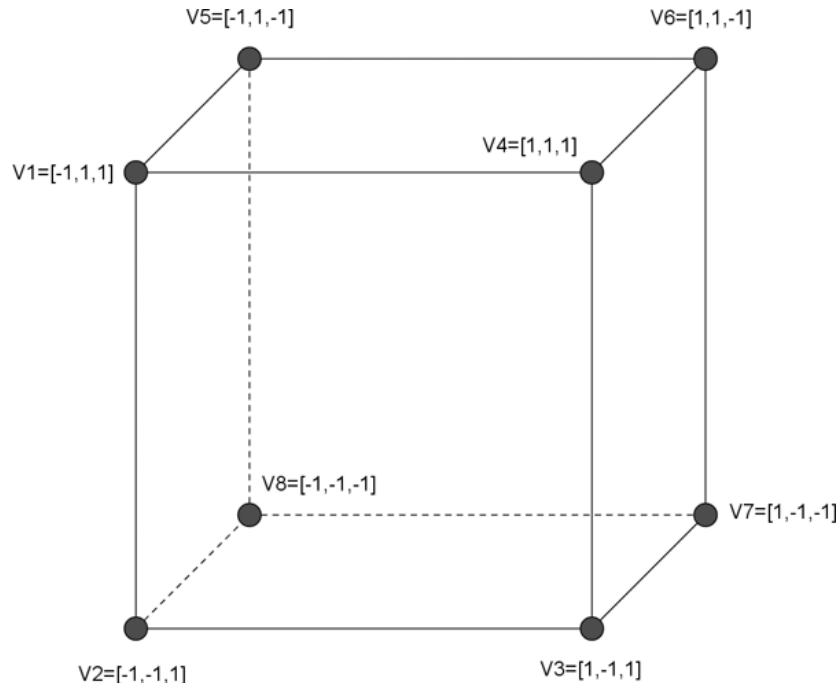


Abb. 29: Würfel im Object Space

Aus dem Object Space wird nun also die erste Matrix ersichtlich, die Model Matrix, welche später in die ModelView(Projection)Matrix eingebunden wird. In dieser Matrix sind Transformationen gespeichert, welche auf alle Vertices eines geometrischen Objekts angewendet werden sollen. So ist es z.B. Möglich den Würfel zu skalieren, also seine Grösse zu ändern.

#### 9.1.2.1.2 World-Space und View Matrix

Ein geometrisches Objekt muss natürlich irgendwo in einer globalen Umgebung platziert werden um gerendert zu werden. Dies ist das zweite Koordinatensystem, der World-Space.

Die Koordinaten die während der Modellierung erstellt wurden müssen nun von ihrem Object Space in den World-Space transformiert werden. Dies geschieht mithilfe einer Transformationsmatrix. Jeder Vertex in dem zum transformierenden geometrischen Objekt wird mit dieser Matrix multipliziert, wodurch der Vertex seine Koordinaten im World-Space erhält.

Der World-Space ergibt die zweite benötigte Matrix, die View Matrix. Diese wird später in die ModelView(Projection) Matrix eingebunden.

#### 9.1.2.1.3 Camera Space und Projection Matrix

Nachdem ein geometrisches Objekt nun im World-Space platziert wurde, ist es an der Zeit etwas auf dem Bildschirm darzustellen. Dazu müssen Objekte, welche sich im Bereich der Kamera befinden erfasst und gerendert werden.

Eine Kamera im klassischen Sinne existiert in OpenGL nicht. Die Relation zwischen Objekten und Kamera in OpenGL ist gerade umgekehrt als man sich das aus der Realität gewohnt ist. Um ein Objekt mit der Kamera zu erfassen wird nicht die Kamera auf das Objekt gerichtet, vielmehr wird das Objekt so



verschoben, das es im Bereich der Kamera liegt.

Dazu wird jeder Vertex eines Objekts mit der View-Matrix multipliziert, wodurch der Vertex seine endgültige globale Position erhält.

#### 9.1.2.1.4 Ablauf

Zusammengefasst ist der Ablauf, dargestellt in Abb. 3, nun also:

Die Vertices

1. ... des zu zeichnenden Object sind im Object Space
2. ... werden in den World-Space transformiert, durch multiplizierung mit der Model Matrix.
3. ... werden in den Camera Space transformiert, durch multiplizierung mit der View Matrix
4. ... werden in den Screen Space transformiert durch multiplizierung mit der Projection Matrix
5. ... befinden sich nun im Screen Space – was der Benutzer später auf seinem Bildschirm sehen wird.

#### 9.1.2.2 Model-View Matrix

Wie der Name schon vermuten lässt, besteht die Model-View Matrix aus zwei Teilen. Der eine stellt Rotationen, Skalierungen und ähnliche weniger genutzte Transformationen dar. Dies sind Informationen über das Model.

Der andere enthält die Translation im Dreidimensionalen Raum, also den View-Teil der Matrix.

Bisher musste sich der Programmierer nicht um die Berechnung dieser Matrix kümmern. Jedoch werden diese Matrizen in moderneren Versionen von OpenGL, sowie OpenGL ES <sup>35</sup> und WebGL <sup>36</sup> nicht mehr automatisch den Shadern übergeben. Dies macht es nötig selbst die Berechnungen durchzuführen.

OpenGL benutzt eine 4x4 Matrix um die ModelView-Matrix darzustellen.

In folgender Darstellung sind nun die beiden Teile der ModelView Matrix ersichtlich:

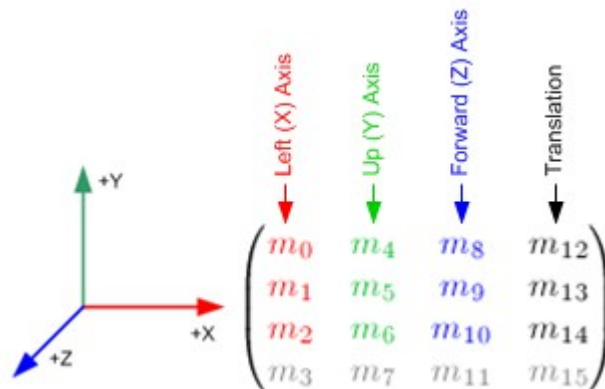


Abb. 30: Darstellung der ModelView Matrix

Der Aufbau ist einfach zu verstehen. Die linken 3 Spalten stellen den Object-Space dar, also alle euklidischen und affinen Transformationen, die auf ein geometrisches Objekt angewendet werden sollen.

Die letzte Spalte, auf der rechten Seite, stellt nun die Translation im World-Space dar.

Es wäre theoretisch möglich diese Matrix auch in einem Shader zu berechnen. Jedoch macht dies wenig Sinn, da die Matrix für ein geometrisches Objekt immer gleich bleibt und sich nicht von Vertex zu Vertex verändert.

<sup>35</sup> OpenGL for Embedded Systems: <http://www.khronos.org/opengles/>

<sup>36</sup> WebGL - OpenGL ES 2.0 for the Web: <http://www.khronos.org/webgl/>

### 9.1.2.3 Projection-Matrix

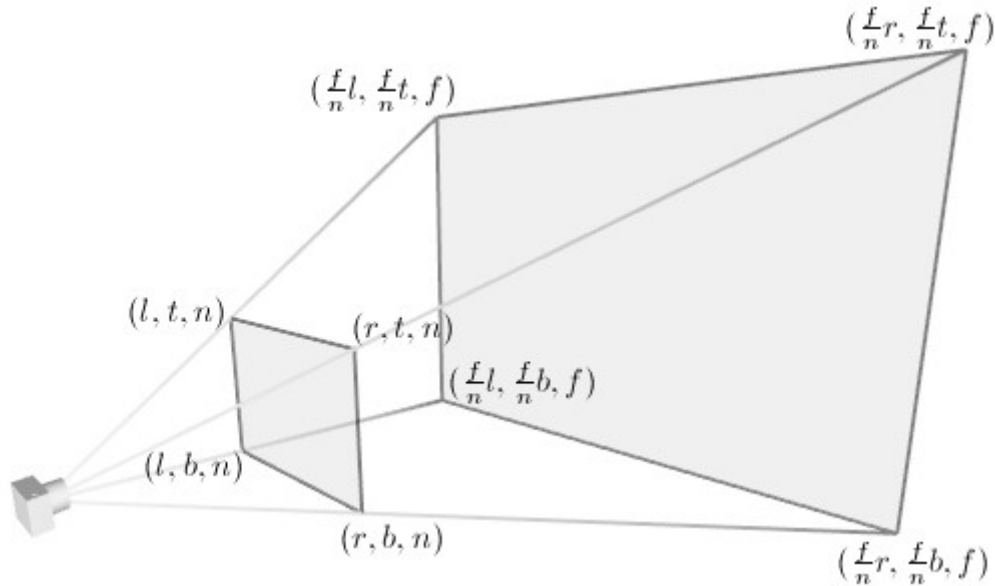


Abb. 31: View-Frustum

In der Projection-Matrix ist das View-Frustum gespeichert, also das eigentliche Sichtfeld der Kamera. Das Frustum ist ein Pyramidenstumpf mit 8 Vertices.

### 9.1.3 Shader

Shader werden von OpenGL seit Version 2.0 unterstützt. Sie erweitern die Fixed-Function Pipeline durch die Programmierfähigkeit und lösen diese in den heutigen Versionen fast vollständig ab. Es handelt sich um Programme welche im Gegensatz zu gewöhnlichen Programmen nicht auf der CPU, sondern auf der GPU laufen. Sie haben vieles mit herkömmlichen Programmen gemeinsam. Shader müssen, ebenso wie herkömmliche CPU-Programme für einen bestimmten Prozessor kompiliert werden, nur dass es sich in dem Fall um spezifische Prozessoren handelt, welche sich auf der Grafikkarte befinden.

Es gibt unterschiedliche Shader-Sprachen. Die folgenden drei sind gegenwärtig verbreitet:

- GLSL (OpenGL Shading Language) für OpenGL
- HLSL (High Level Shading Language) von Microsoft für DirectX
- CG (C for Graphics) von Nvidia sowohl für OpenGL, als auch für DirectX

Für die vorliegende Arbeit wurde GLSL verwendet.

Es gilt zu beachten, dass der simultane Einsatz von Shadern und der Fixed-Function Pipeline für denselben Teilprozess nicht möglich ist. Dies bedeutet, dass sämtliche Aufgaben, welche von dem betreffenden Teil der Pipeline zur Verfügung gestellt werden vom entsprechenden Shader selbst bewältigt werden müssen. Es ist aber möglich nur für bestimmte Teile Shader zu verwenden. Es ist beispielsweise möglich nur einen Fragment-Shader ohne einen dazugehörigen Vertex-Shader einzusetzen. In diesem Fall wird dann einfach die Fixed-Function für den Vertex-Processing-Teil ausgeführt. Das nächste Kapitel versucht aufzuzeigen, was es mit der Rendering-Pipeline auf sich hat. Shader haben gegenüber der Fixed-Function den Vorteil, dass sie dem Entwickler viel mehr Spielraum bieten und damit flexibler sind. Sie haben aber auch den Nachteil, dass sie langsamer sind als in Hardware realisierte Verarbeitungsschritte.

### 9.1.4 Die Rendering-Pipeline

Die nachfolgende Tabelle beschreibt in vereinfachter Weise den Ablauf des Rendering-Prozesses. Eine ausführlichere Beschreibung der jeweiligen Teilprozesse erfolgt später.

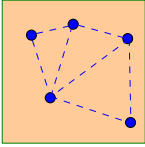
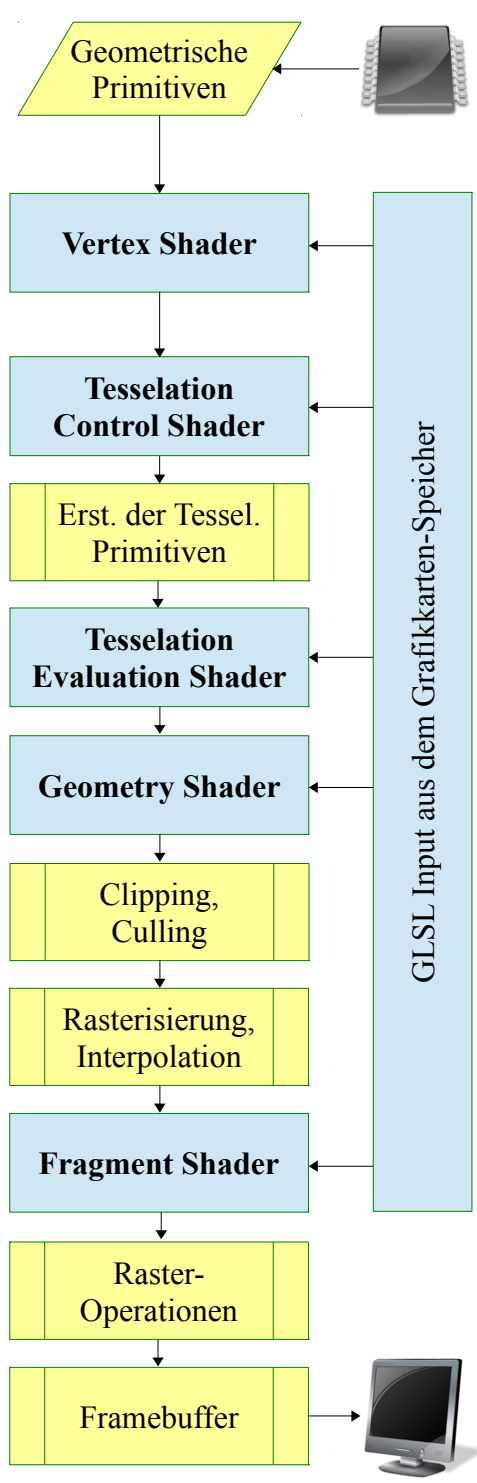
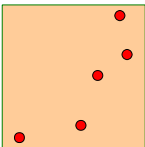
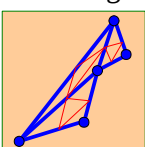
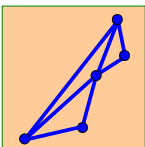
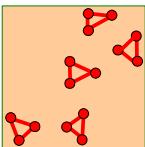
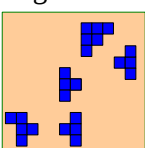
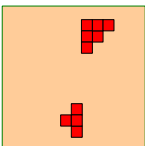
Teilprozess	Beschreibung	Flussdiagramm
<b>Eingabe</b> 	Am Anfang der Rendering-Pipeline wird die Information aus dem Speicher (RAM) geladen. Es handelt sich dabei um geometrische Primitiven. Diese sind eine Sammlung von Punkten im Raum (Vertices) und repräsentieren das virtuelle Modell.	
<b>Punkte</b> 	Im nächsten Schritt werden diese vom Vertex-Prozessor verarbeitet. Er ist zuständig für die Transformation (Translation, Rotation, etc.) der eingegangenen Punkte. Diese werden nun wieder zu geometrischen Primitiven zusammengefasst und als sogenannte Patches weitergegeben.	
<b>Kachelung</b> 	Der Tessellation Control Shader wandelt die erhaltenen Patches in reguläre Oberflächen um und berechnet den gewünschten Tessellationsgrad. Anschliessend werden in einem Fixed-Function Schritt U-V-W Koordinaten erzeugt und an den Tessellation Evaluation Shader weitergegeben, welcher die neue Oberfläche berechnet.	
<b>Geometrie</b>  	An dieser Stelle wird die Erstellung der geometrischen Primitiven gesteuert. Danach erfolgt das Clipping, wobei - vereinfacht ausgedrückt - nicht sichtbare Bereiche entfernt werden. Im Anschluss darauf folgt die Umsetzung der kontinuierlichen geometrischen Objekten in diskrete Pixelbilder, auch Rasterisierung genannt. Für jede Primitive resultieren daraus Fragmente.	
<b>Fragmente</b> 	Der Fragment-Shader, in anderen Sprachen auch Pixel-Shader genannt verarbeitet diese dann völlig unabhängig voneinander. Ziel dieses Schrittes ist das Setzen der Farbe oder das vollständige Verwerfen des jeweiligen Fragmentes.	
<b>Ausgabe</b> 	Abschliessend wird noch Tiefenprüfung (Z-Buffering) ausgeführt und das Bild an den Framebuffer übergeben. Damit ist das Rendering abgeschlossen und die Darstellung am Bildschirm kann erfolgen.	

Table 5: Übersicht über die Rendering-Pipeline

Legende:   Programmierbar   Konfigurierbar ● Input ● Output

Bis zur Version 2.0 von OpenGL war die Rendering Pipeline nur konfigurierbar. Je nach Version von OpenGL sind mehr oder weniger Teile der Pipeline programmierbar. Im Verlauf der Zeit wurden immer wie mehr Teile der Rendering-Pipeline programmierbar. Die folgende Tabelle<sup>37</sup> gibt eine chronologische Übersicht dazu:

GLSL Version	OpenGL Version	Jahr	Erweiterung um programmierbare Bestandteile
1.10.59	2.0	2004	Vertex- und Fragment-Shader
1.20.8	2.1	2006	
1.30.10	3.0	2008	
1.40.08	3.1	2009	
1.50.11	3.2	2009	Geometry-Shader
3.30.6	3.3	2010	
4.00.9	4.0	2010	Tessellation-Control- und Tessellation-Evaluation-Shader
4.10.6	4.1	2010	
4.20.6	4.2	2011	
4.30.6	4.3	2012	Compute-Shader

Table 6: Übersicht OpenGL/GLSL Versionen

## 9.1.5 GLSL Sprachmerkmale

Shaders werden in OpenGL mittels der Shading Language (GLSL) implementiert. Die Syntax von GLSL ist mit derjenigen von C nahezu identisch, was das Erlernen der Sprache vereinfacht. Es existieren auch Konstruktoren für bestimmte Datentypen, Strukturen, Präprozessor-Direktiven und vieles mehr. Gegenüber C fehlen der Shadersprache jedoch Zeiger, jegliche Art von Strings oder die dynamische Speicherallozierung. Auch sind rekursive Funktionsaufrufe nicht gestattet.

Alle Shader müssen über eine `main` Funktion verfügen.

### 9.1.5.1 Primitive Datentypen

Es werden die folgenden primitiven Datentypen unterstützt:

- `bool`
- `int` (signed) und `uint` (unsigned)
- `float` und `double`

### 9.1.5.2 Arrays

Mehrdimensionale Arrays, wie sie in den meisten Programmiersprachen häufig zur Anwendung kommen können nicht verwendet werden. Nur 1-Dimensionale Array sind zulässig.

Für vorgegebene Datentypen sind Vektoren in vorgegebenen Grössen definiert. Dabei wird die Länge als Zahl angehängt:

- Fließkomma: `vec2`, `vec3`, `vec4` bei `float`, resp. `dvec2`, `dvec3`, `dvec4` bei `double`
- Ganzzahlig: `ivec2`, `ivec3`, `ivec4`, resp. `uvec2`, `uvec3`, `uvec4`

<sup>37</sup> Quelle: [http://www.opengl.org/wiki/History\\_of\\_OpenGL](http://www.opengl.org/wiki/History_of_OpenGL)

Ein sehr praktisches Merkmal dieser Vektoren ist, dass man auf ihre Komponenten nicht nur mit den eckigen Klammern zugreifen kann, sondern auch mit verschiedenen Zugriffsnamen (welche Synonyme für die Positionen von 0 bis 3 darstellen). Dies wird auch als Swizzling bezeichnet:

Positionen				Verwendung
[0]	[1]	[2]	[3]	Sinnvoll wenn Iterationen stattfinden müssen (in Schleifen)
x	y	z	w	Nützlich für Vektoren, welche Punkte im Raum oder einer Fläche darstellen
r	g	b	a	Praktisch für Vektoren, welche einen Farbwert repräsentieren
s	t	p	q	Wird bei Textur-Koordinaten verwendet

Table 7: Swizzling-Tabelle

Beispiele:      `vec4 a = vec4(1.0, 0.0, 0.0, 0.5)`      *halbtransparentes Rot*  
                  `vec3 b = vec3(3, 7, 11)`                      *implizite Konvertierung von int zu float*  
                  `vec4 c = vec4(a.ba, b.tp)`                      *Winzling (Vertauschen) und Konstruktion*

a.g ist dasselbe wie a[1] oder a.y und hat hier den Wert 0.0  
b.a ist unzulässig

### 9.1.5.3 Matrizen

Um geometrische Berechnungen, besonders im 3-Dimensionalen Raum effizient durchführen zu können sind Matrizen unumgänglich. Daher bietet GLSL für Fliesskommazahlen alle Kombinationen von Matrizen mit 2 bis 4 Spalten resp. Zeilen an. Es reicht `mat2`, `mat3`, `mat4` für quadratische float Matrizen, ansonsten ist die genaue Angabe der Dimension notwendig: `mat2x2`, `mat2x3`, ... `mat4x3`, `mat4x4`. Soll für höhere Präzision double verwendet werden, muss ein `d` vorangestellt werden, zum Beispiel `dmat3x4`.

### 9.1.5.4 Kennzeichnungen

Variablen und Funktionen können mit Qualifiers gekennzeichnet werden. Die wichtigsten sind in der nachfolgenden Tabelle, inkl. dem entsprechenden Geltungsbereich aufgeführt.

Qualifier	Anwendung	Bedeutung
<code>const</code>	Variablen	Konstante zur Kompilierungszeit oder Read-only Funktionsparameter
<code>in</code>	Variablen	Abstammung aus einem vorgängigen Verarbeitungsschritt
<code>out</code>	Variablen	Übergabe aus dem aktuellen Shader in den nächsten Verarbeitungsschritt
<code>uniform</code>	Variablen	Die Herkunft der Variable ist die Anwendung
<code>smooth</code>	Interpolation	Perspektivisch-korrekte Interpolation
<code>flat</code>	Interpolation	Keine Interpolation
<code>noperspective</code>	Interpolation	Lineare Interpolation
<code>in</code>	Funktionen	Funktionsparameter, die in die Funktion einfließen (Grundzustand)
<code>out</code>	Funktionen	Uninitialisierte Parameter, die dem Aufrufer zurückgegeben werden
<code>inout</code>	Funktionen	Initialisierte Parameter, welche dem Aufrufer zurückgegeben werden

Table 8: Qualifiers-Tabelle

### 9.1.5.5 Operatoren

Die Operatoren, sowie die Priorität in Ausdrücken sind für GLSL gleich wie in C/C++.

### 9.1.5.6 Integrierte Typen und Funktionen

Die GLSL Sprache kennt fünf verschiedene sogenannte Built-In Typen. Es handelt sich dabei um:

- **Konstanten:** Es handelt sich dabei um Hardware-abhängige Konstanten, wie beispielsweise die maximale Anzahl der Lichter: `gl_MaxLights`.
- **Uniforms:** Variablen welche vom OpenGL Programm an den Shader übermittelt werden. Zum Beispiel `gl_ProjectionMatrix`.
- **Attribute:** Es handelt sich dabei um Vertex-Attribute, welche im Vertex-Shader verwenden können, wie die Normale eines Vertices oder die Farbe mit `gl_Color`. Das `attribute` Keyword ist in neuen GLSL Versionen deprecated.
- **Varyings:** Es sind die Variablen welche von einem Shader zum anderen übermittelt werden. Mit der Syntax der Versionen bis 1.2 mit musste das Keyword `varying` verwendet werden. `gl_Front_Color` ist ein Beispiel für ein solches Built-In.
- **Funktionen:** GLSL bietet eine Reihe von eingebauten Funktionen, wie Sinus (`sin`), Modulo (`mod`), Matrix-Inversion (`inverse`) unter vielen anderen.

Eine gute und vollständige Übersicht über die existierenden Datentypen, Qualifiers, Befehle, etc. findet man im OpenGL 4.3 & GLSL Quick Reference Guide<sup>38</sup>.

### 9.1.6 GLSL-Shader

Im Gegensatz zu einem traditionellen Programm laufen grafische Programme nicht ausschliesslich auf dem Hauptprozessor, sondern zum grössten Teil auf der Grafikkarte. Das steuernde Programm, welches aus einem oder auch mehreren Betriebssystem-Prozessen besteht, muss jedoch, wie jedes andere Programm auch, auf der CPU (oder den CPUs) laufen und aus dem Arbeitsspeicher gelesen werden. Es ist verantwortlich für die Delegation von Verarbeitungsschritten an die Grafikkarte. Dazu gehören auch die Shader, denn sie sind reine GPU-Programme. Sie haben keinerlei Zugriff auf RAM oder Geräte. Das CPU-Programm kompiliert den Shader-Code und übergibt das ausführbare Programm der Grafikkarte.

Dieser Vorgang der Shader-Erstellung verläuft nach einem fest vorgegebenen Ablauf und lässt sich wie folgt zusammenfassen:

1. Zuerst wird das GPU-Programm deklariert mit `glCreateProgram`.
2. Anschliessend wird mit `glCreateShader` definiert, welcher Shader-Typ<sup>39</sup> erstellt werden soll. Das in Schritt 1 erstellte Programm, als auch soeben definierte Shader werden über einen Integer (`GLuint`) identifiziert.
3. Dann wird der, meistens in externen Dateien gespeicherte, Shader-Code geladen. Es ist auch möglich den Shader-Code direkt im Quelltext als `GLchar*` einzubetten und zu übergeben. Dies geschieht mit dem Befehl `glShaderSource`.
4. Nun kann der Shader mit dem Aufruf von `glCompileShader` kompiliert werden.
5. Im nächsten Schritt kann der nun kompilierte Shader dem GPU-Programm angehängt werden. `glAttachShader` wird dafür verwendet.
6. Jetzt muss das Programm noch gelinkt werden mittels `glLinkProgram`.
7. Schliesslich kann das Programm als Teil der Pipeline mit dem Befehl `glUseProgram` verwendet werden.

Analog zur Erstellung stehen auch für das Aufräumen die Befehle `glDetachShader`, `glDeleteShader` und `glDeleteProgram` zur Verfügung.

All diese Vorgänge werden vom später vorgestellten Grafik-Modul ausgelöst. Es wird dabei nicht direkt mit den OpenGL Befehlen gearbeitet, sondern mit einer Zwischenschicht (einer C++ Bibliothek, welche auf

<sup>38</sup> Quick Reference Card: <http://www.opengl.org/documentation/glsl/>

<sup>39</sup> Mögliche Shader-Typen sind, je nach Version von GLSL: `GL_VERTEX_SHADER`, `GL_FRAGMENT_SHADER`, `GL_GEOMETRY_SHADER`, `GL_TESS_EVALUATION_SHADER`, `GL_TESS_CONTROL_SHADER` oder `GL_COMPUTE_SHADER`

welche im entsprechenden Kapitel noch näher eingegangen wird). Im Hintergrund müssen aber auch da genau dieselben Abläufe stattfinden.

Abhängig von ihrer Position im Rendering-Prozess unterscheiden sich die Shader in ihren Standard-Eingabe- und Ausgabe-Attribute. Grundsätzlich kann jeder Shader jede Art von zulässigen, benutzerdefinierten Parametern entgegennehmen oder ausgeben. Bei jedem Shader können also Variablen von ausserhalb der Rendering-Pipeline einfließen. Je nach Aufgabe benötigt es mehr oder weniger dieser benutzerdefinierten Variablen.

### 9.1.6.1 Vertex-Shader

Der Vertex-Prozessor verarbeitet vom Konzept her jeweils jeden Vertex einzeln und hat keine Kenntnisse über anliegende oder sonstige Vertices. Ein Vertex ist ein Eckpunkt einer geometrischen Form. Während dieser Etappe der Rendering Pipeline findet in der Fixed Function Pipeline unter anderem die Transformation, Normalen-Berechnung für die Beleuchtung, sowie die Textur-Koordinaten-Generierung statt. Der Vertex-Shader kann diese oder auch weniger oder mehr Vertex-Basierte Operationen durchführen. Er muss jedoch mindestens die Variable `gl_Position` ausgeben, welche die Position des entsprechenden Vertex nach der Verarbeitung durch den Shader beinhaltet.

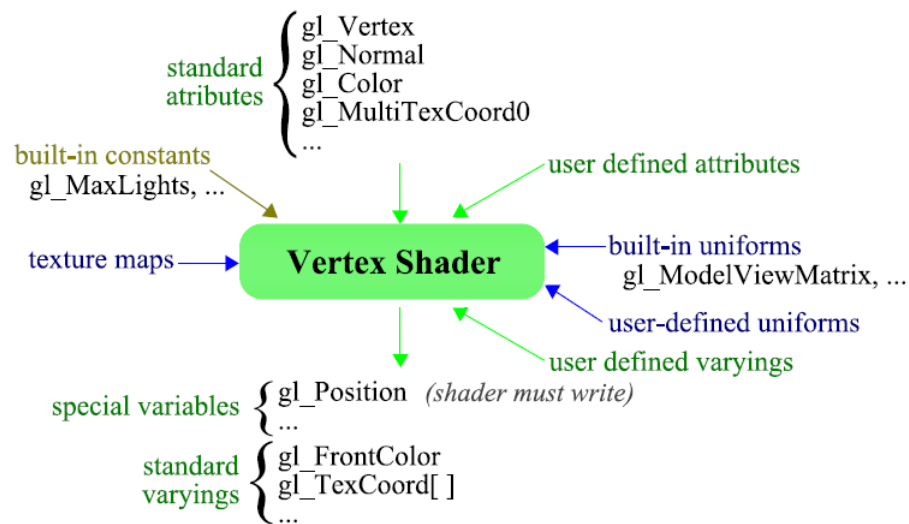


Abb. 32: Variablen des Vertex-Shaders

Da mit dem Vertex-Shader Eckpunkte verarbeitet werden, lässt sich damit auch die Form des Objektes verändern. Eine Anwendung dafür ist beispielsweise eine wehende Fahne.



### 9.1.6.2 Tessellation-Shaders

Ab Version 4.0 von GLSL ist auch der Schritt der Parkettierung (Tessellation) zur Erstellung der geometrischen Primitiven programmierbar.

Tessellation ist eine Etappe in der Grafik-Pipeline, welche Patches als Input erhält und neue Primitiven generiert, wobei es sich um Punkte, Linien oder Dreiecke handeln kann.

Ein Patch ist ein Array von Vertices deren Attribute vom Vertex-Shader berechnet wurden. Die Tessellation-Shaders erhalten diese transformierten Vertices und unterteilen diese in der Regel in kleinere Primitiven. Man unterscheidet drei Typen von Patches: Linen (Isolines)-, Dreieck- und Viereck-basierte Patches. Im Gegensatz zu den OpenGL Primitiven haben Patches eine benutzerdefinierte Anzahl von Punkten.

Die Sub-Pipeline der Tessellation besteht aus drei Schritten: Tessellations-Steuerung, Primitivengenerierung und Tessellations-Evaluation. Die Primitivengenerierung ist nicht vorgegeben, die beiden anderen Teile können programmiert werden.

#### 9.1.6.2.1 Tessellation Control Shader

Der Tessellation Control Shader erhält einen Patch als Eingabe. Seine Aufgabe ist es primär zu bestimmen wie stark die Tessellation zur Anwendung kommen soll. Es werden auch Attribute per-Patch geschrieben, welche den Unterteilungsgrad des Patches definieren. Dieser kann von 0 bis `GL_MAX_TESS_GEN_LEVEL` (meistens 64) gehen.

Er steuert die Unterteilung von jeder Seite des Patches, als auch das Innere. Es gibt mindestens vier äussere Unterteilungen beim Viereck (quad) und drei beim Dreieck (triangle). Jeweils eine pro Seite. Dann benötigt es mindestens zwei innere Unterteilungsstufen (Länge, Breite) beim Quad-Patch und eine beim Triangular-Patch.

#### 9.1.6.2.2 Primitiven-Generierung

Der nächste Schritt, die Primitiven-Generierung ist ein Fixed-Function-Schritt. Er ist für die Erstellung der Punkte innerhalb des Patches zuständig. Die Rendering-Pipeline verwendet dafür das folgende Muster<sup>40</sup>:

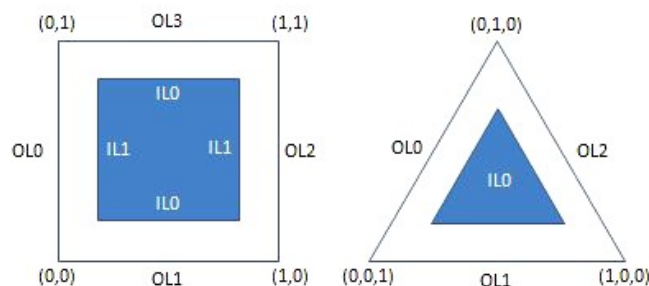


Abb. 33: Einfluss der Tessellation auf Kanten und Flächen

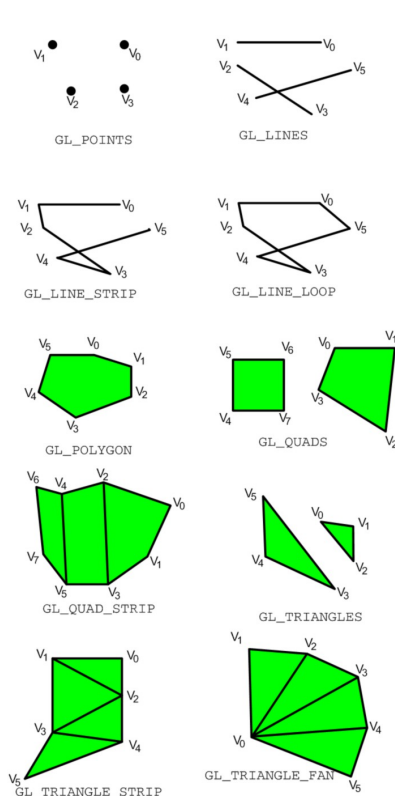
#### 9.1.6.2.3 Tessellation Evaluation Shader

Der letzte Schritt bei der Tessellation ist die Berechnung der interpolierten Positionen und anderen Attributen. Der Tessellation Evaluation Shader nimmt die vorgängig erstellte, abstrakte Tessellations-Primitive und die aktuellen Vertex-Daten des gesamten Patches entgegen und erzeugt einen Punkt daraus. Jeder Aufruf des Shaders erzeugt einen einzelnen Vertex. Er wird mindestens einmal für jeden Tessellations-Punkt aufgerufen.

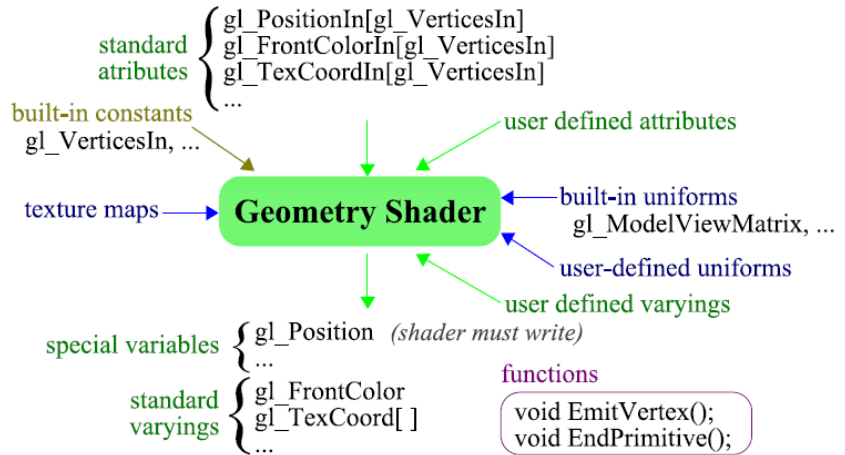
40 Quelle: <http://www.lighthouse3d.com/tutorials/glsl-core-tutorial/glsl-core-tutorial-tessellation/>



### 9.1.6.3 Geometry-Shader



Dieser Shader hat viel Ähnlichkeit mit den vorher beschriebenen Tessellation-Shaders. Er wurde etwas früher in die OpenGL-Sprache eingeführt. Er kann verarbeiten und erstellt neue Primitiven.



In der Praxis wird er meistens eingesetzt um Schattenvolumen zu erzeugen, die Geometrie auf die Seiten einer Cube-Map zu verteilen oder Haargeometrie zu produzieren.

Abb. 34: OpenGL Primitiven

Aus der OpenGL-Anwendung werden die in Abb.39 gezeigten geometrische Primitiven<sup>41</sup> generiert und in die Pipeline übertragen. Der Treiber wandelt diese dann in Adjazenz-Primitiven um, welche im nächsten Abschnitt noch genauer beschrieben werden. Schliesslich werden dann vom Geometry-Shader GL\_POINTS, GL\_LINE\_STRIP oder GL\_TRIANGLE\_STRIP erzeugt.

Es gibt keine Abhängigkeit vom Input-Typ zum Ausgabe-Typ. Ein Shader der Punkte entgegennimmt kann beispielsweise Dreiecke erzeugen.

41 Bild: <http://www.cs.uregina.ca/Links/class-info/405/WWW/Lab2/>

### 9.1.6.4 Adjazenz-Primitiven

Adjazenz ist ein Synonym für Nachbarschaft. Sie kann in vier verschiedenen Primitiven eingesetzt werden.

Die folgenden Typen existieren:

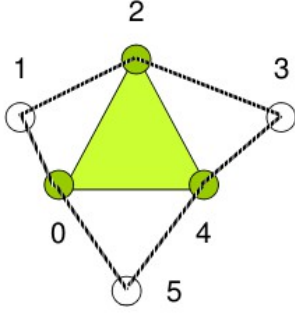
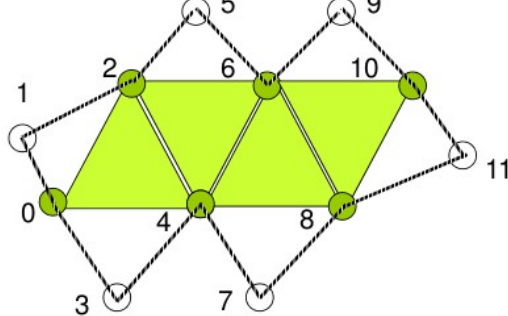
<b>GL_LINES_ADJACENCY</b> 0      1      2      3 ○      ● → ●      ○	<b>GL_LINE_STRIP_ADJACENCY</b> 0      1      2      3      4      5 ○      ● → ● → ● → ●      ○
<b>GL_TRIANGLES_ADJACENCY</b> 	<b>GL_TRIANGLE_STRIP_ADJACENCY</b> 

Table 9: Adjazenz-Primitiven

Dabei sind die weissen, angrenzenden Punkte diejenigen, welche für die Nachbarschafts-Bestimmung verwendet werden.

Der Geometry-Shader übernimmt die Variablen `gl_Position`, `gl_PointSize` und `gl_Layer` vom Vertex-Shader, falls diese geschrieben werden und liest sie als `gl_PositionIn[<Typ>]`, `gl_PointSizeIn[<Typ>]` und `gl_LayerIn[<Typ>]` ein.

### 9.1.6.5 Funktionen

Wenn die Funktion `EmitVertex` aufgerufen wird, wird der Satz von definierten Variablen zwischengespeichert und schliesslich mit `EndPrimitive` (es gibt keine `BeginPrimitive`-Funktion, dies wird beim Start des Shaders impliziert) zusammengestellt für die Weiterverarbeitung. `EndPrimitive` wird am Ende des Shaders automatisch aufgerufen, so dass der Befehl oft weggelassen wird.

### 9.1.6.6 Fragment-Shader

Der Fragment-Shader ist der letzte programmierbare Teil der Rendering-Pipeline und verarbeitet die interpolierten Werte, gibt Zugriff auf die Texturen und ermöglicht Farb-Manipulationen für das jeweilige Fragment. Ein Fragment ist der Datensatz welcher durch die Rasterisierung entsteht und für die Aktualisierung eines Framebuffer-Pixels benötigt wird. Es beinhaltet unter Unser Projektmanagement haben wir über das Managementtool „Redmine“ abgewickelt. Der Einsatz von Redmine hat sich für uns bewährt. Wir hatten die Phasen und einzelnen Tasks gut im Griff und waren uns über den Fortschritt des Projektes jederzeit bewusst.

Die meisten Meilensteine konnten wir halten. Was jedoch die Haptic betraf, so brauchten wir mehr Zeit als geplant. Rückblickend hätten wir die ganze Haptic im Vorfeld besser Analysieren müssen. I3D als Framework ist ein wenig eingesetztes Framework. Diesen Umstand haben wir in Form von Schwierigkeiten (Fehler konzeptionell oder Programmfehler) zu spüren bekommen. Diesen Faktor hätten wir mehr Beachtung schenken sollen. anderem die Fenster-Koordinaten, Farbe, Tiefe und Textur-Koordinaten. Im Fragment-Processing finden auch die Berechnungen für die Erstellung von Nebel-Effekten statt.

Im Fragment-Shader-Code kann man mehrmals auf Texturen zugreifen und die Werte nach Belieben kombinieren. Es gilt aber ähnlich wie beim Vertex-Shader, dass keine Operationen durchgeführt werden können, welche mehrere Fragmente gleichzeitig betreffen.

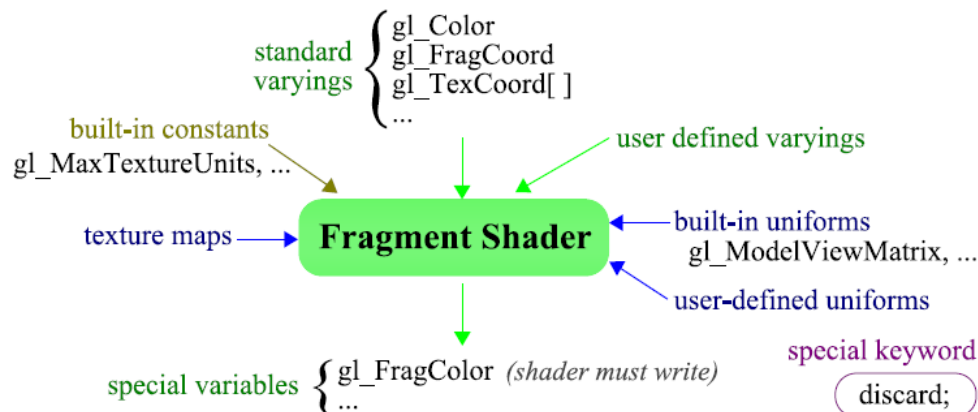


Abb. 35: Variablen des Fragment-Shaders

### 9.1.6.7 Compute-Shader

Im Gegensatz zu den bisher vorgestellten Shadern welche in einer strikt vorgegebenen Reihenfolge verwendet werden müssen, können Compute-Shader zu einem beliebigen Zeitpunkt zum Einsatz kommen. Sie werden also auch nicht automatisch im Rendering-Prozess ausgeführt, wie die anderen Shader, sondern müssen im Programm explizit mit `glDispatchCompute` aufgerufen werden. Sie finden vor allem Einsatz für nicht-grafische, parallelisierbare Programmabläufe, wobei die Grafikkarte im Sinne einer GPGPU (General Purpose Computation on Graphics Processing Unit) eingesetzt wird. Wie auch bei der Verwendung von OpenCL oder CUDA wird hierbei die Gegebenheit ausgenutzt, dass Grafikkarten für das effiziente und parallele Berechnen von Matrizenoperationen ausgelegt sind. Aber es lassen sich natürlich auch grafische Operationen damit realisieren.

Damit der Compute-Shader verwendet werden kann, muss die Grafikkarte OpenGL in der Version 4.3 unterstützen.