

Fakultät Informatik



OSTBAYERISCHE  
TECHNISCHE HOCHSCHULE  
REGENSBURG

## Bachelorarbeit

Erstellung einer würfelbasierten 3D-Grafikbibliothek mit Java & OpenGL

eingereicht von: Tobias Zink  
Matrikelnummer: 2764045  
Studiengang: Allgemeine Informatik  
OTH Regensburg

betreut durch: Prof. Dr. rer. nat Carsten Kern  
OTH Regensburg  
Prof. Dr. rer. nat Klaus Volbert  
OTH Regensburg

Regensburg, der 11. September 2016

# **Ehrenwörtliche Erklärung**

Ich erkläre hiermit ehrenwörtlich, dass ich die vorliegende Arbeit selbstständig angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Es wurden keine anderen als die angegebenen Quellen und Hinweise verwendet.

Die vorliegende Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Regensburg, der 11. September 2016

Tobias Zink

# **Kurzfassung**

Im Rahmen dieser Bachelorarbeit wurde eine 3D Grafikbibliothek zur Darstellung einer Würfelwelt entwickelt, welche aus einer einfachen Datenstruktur ein komplexes Gittermodell auf dem Bildschirm anzeigt. Mit dieser Bibliothek soll Studenten ein leichterer Einstieg in die grafische Programmierung rund um OpenGL mit Java ermöglicht werden.

Nach einer Gegenüberstellung der verfügbaren Technologien werden die Grundlagen der 3D-Programmierung unter Java mithilfe der OpenGL-Schnittstelle LWJGL dargestellt. Des Weiteren werden die Vor- und Nachteile eines Gittermodells erläutert und wie man dieses optimieren kann.

Nach der Vermittlung dieser Grundlagen wird anhand von Quellcode-Beispielen noch auf einzelne Elemente der 3D Grafikbibliothek eingegangen. Zu guter Letzt wird überprüft ob das optimierte Gittermodell tatsächlich einen Vorteil bringt.

# Inhaltsverzeichnis

<b>Glossar</b>	<b>1</b>
<b>1 Einführung</b>	<b>3</b>
1.1 Motivation & Zielsetzung . . . . .	3
1.2 Gliederung der Arbeit . . . . .	4
<b>2 Grundlagen</b>	<b>5</b>
2.1 OpenGL . . . . .	5
2.2 LWJGL . . . . .	6
2.3 Mathematik . . . . .	8
<b>3 Design</b>	<b>12</b>
3.1 Architektur . . . . .	12
3.2 Grafikpipeline . . . . .	14
3.3 Beispielprojekt . . . . .	20
<b>4 Realisierung</b>	<b>24</b>
4.1 Game Loop . . . . .	24
4.2 Definition eines Würfels . . . . .	25
4.3 Umsetzung mit Entities . . . . .	27
4.4 Aufbau eines Gittermodells . . . . .	29
4.5 Optimierung des Gittermodells . . . . .	30
4.6 Shader . . . . .	34
<b>5 Performance</b>	<b>36</b>
<b>6 Fazit</b>	<b>39</b>
6.1 Zusammenfassung . . . . .	39
6.2 Ausblick . . . . .	40
<b>7 Anhang</b>	<b>41</b>
Inhalt der beigefügten CD . . . . .	41
Bilder . . . . .	42
<b>Literaturverzeichnis</b>	<b>48</b>

# Abbildungsverzeichnis

2.1	Transformationen visualisiert.	10
2.2	Zusammenhang der Koordinatensysteme.	11
3.1	Chunks bei Sichtweite 2.	13
3.2	Vereinfachte Grafikpipeline.	14
3.3	Weißes Dreieck: ein Dreieck im „immediate mode“	17
3.4	Zusammenhang VAO & VBO.	18
3.5	Dreieck mit VAO.	18
3.6	Beispielprojekt: Schachbrett.	21
3.7	Quellcode: Würfel manipulieren.	22
3.8	Quellcode: eigene Spielklasse.	23
3.9	Quellcode: eigene Welt.	23
4.1	Ein sehr einfacher 'Game Loop'	24
4.2	Ein besserter 'Game Loop', welcher die verstrichene Zeit beachtet	25
4.3	Quadrat aus zwei Dreiecken	26
4.4	Quellcode um ein Entity zu erstellen.	28
4.5	Ein Würfel als Entity	28
4.6	UML: Entity Klasse	29
4.7	Ein „worst-case“ Gittermodell	32
4.8	Methode zum Hinzufügen von Würfelseiten.	33
4.9	Ein optimales Gittermodell	34
4.10	Quellcode: Laden des Shaders.	35
7.1	Stanford-Drache	42
7.2	Gittermodell nicht optimiert	43
7.3	Gittermodell optimiert	44
7.4	Gittermodell optimiert ohne Backface-Culling	45
7.5	Klassendiagramm Renderer	46
7.6	Klassendiagramm Weltklasse	47

# Tabellenverzeichnis

3.1	Weltspeicherbedarf . . . . .	12
4.1	Würfel aus acht Punkten . . . . .	26
4.2	Speicherverbrauch Gittermodell . . . . .	30
5.1	Systemkonfiguration Entwicklungsrechner . . . . .	36
5.2	Performance optimiertes Gittermodell . . . . .	38
5.3	Performance nicht optimiertes Gittermodell . . . . .	38

# Glossar

API	Eine Programmierschnittstelle zur Anbindung von eigenen Programmen an fremde Implementierungen. (eng.: application programming interface)
Backface Culling	Eine Technik, um nicht sichtbare Dreiecke zu entfernen. Benötigt Normalvektoren um Vorder- bzw. Rückseiten zu erkennen.
Chunk	Ein Teil einer Welt. Besteht aus einer festgesetzten Anzahl von Würfeln. In der umgesetzten Bibliothek immer kubisch.
Frame	Ein Frame ist ein einzelnes Bild. Idealerweise hat man in etwa so viele Frames pro Sekunde, wie der Monitor anzeigt.
Framebuffer	Der Bildspeicher auf der Grafikkarte. Jedem Bildschirmpixel kann genau ein Bereich des Framebuffers zugewiesen werden.
Index	Um aus mehreren Vertices ein Polygon zu zeichnen, wird durch indices die Reihenfolge angegeben, in der diese verbunden sind. (Mehrzahl: indices)
JAR-File	Eine Archiv-Datei zur Verteilung von Java-Klassenbibliotheken.
LWJGL	Lightweight Java Game Library kurz LWJGL ist eine Implementierung der OpenGL API in der Programmiersprache Java.
Normals	Ein Vektor, der rechtwinklig auf einer Fläche steht, um z.B. festzustellen, ob eine Fläche dem Benutzer zugewandt ist oder nicht und sie von der Bilderstellung auszuschließen ist. Außerdem werden sie zur Licht- und Reflexionsberechnung benötigt.
OpenGL	Open Graphics Library, ein Standard, der zur Darstellung von 3D-Grafiken genutzt wird.
Pixel	Ein einzelner Bildpunkt auf dem Bildschirm.
PNG	Ein Dateiformat für Bilder. (Englisch: Portable Network Graphics)

Rendern	Die grafische Ausgabe über die Grafikkarte von einzelnen Objekten oder kompletten Szenen. Findet im Idealfall ungefähr so oft statt wie die Bildwiederholfrequenz.
Szenengraph	Der Szenengraph beschreibt ganz allgemein eine Datenstruktur, die Elemente einer virtuellen Welt beinhaltet. Beispielsweise: eine Sammlung von Objekten, Beleuchtung und Animationen.
UML	Unified Modeling Language
VAO	Vertex Array Object. Ein OpenGL Objekt, das Verweise zu allen Buffern (VBOs) enthält, die zum Darstellen eines Objektes benötigt werden.
VBO	Vertex Buffer Object, verweist auf einzelne Buffer im Grafikspeicher.
Vertex	Mit Vertex ist ein Punkt mit einer X, einer Y und einer Z Koordinate gemeint. (Mehrzahl: vertices)

# 1 Kapitel 1

# Einführung

## 1.1 Motivation & Zielsetzung

Minecraft ist zwar nicht das erste Spiel, das eine Grafik aus Würfeln benutzt, aber unbestritten das Bekannteste. Da Minecraft aber nicht quelloffen ist und sich dadurch für den universitären Einsatz nur begrenzt eignet, existiert der Wunsch nach Alternativen. Mit „Minetest“<sup>1</sup> gibt es eine quelloffene Implementierung, welche auch eine größere Entwicklergemeinschaft vereint, aber da es komplett in C++ geschrieben ist, eignet es sich für den universitären Einsatz an der OTH auch nur begrenzt.

Die einzige quelloffene Minecraft Alternative in Java, die aufzufinden war, ist Terasology<sup>2</sup>. Dieses baut zwar auf die gleiche Technologie wie das Projekt der Bachelorarbeit auf, ist aber sehr viel komplexer und dementsprechend für die Verwendung an der Hochschule auch nicht ohne Weiteres geeignet.

Es ist nicht das Ziel dieser Abschlussarbeit, eine vollständige „3D-Engine“ zu schreiben, welche meist das Herzstück in Computerspielen darstellt. Vielmehr soll im Umfang dieser Abschlussarbeit eine technische Basis für eine würfelbasierte Grafikbibliothek für den universitären Einsatz geschaffen werden. Dabei ging es nicht nur darum, Blöcke möglichst einfach auf dem Bildschirm anzuzeigen, sondern es lag auch ein besonderes Augenmerk darauf, dies im Hinblick auf die Leistung zu optimieren.

Außerdem sollte der Quellcode der praktischen Arbeit sauber und ausführlich dokumentiert sein, damit folgende Absolventen auf diesen - zumindest in Teilen - aufbauen können.

Die Bibliothek soll in seinem Aufbau relativ einfach gehalten werden und eine gut dokumentierte API besitzen, damit Benutzer (vorrangig Studenten der OTH Regensburg) diese ohne längere Einarbeitung nutzen können. Um dies zu gewährleisten, sollte die Bibliothek zudem ausschließlich in Java programmiert werden, wodurch eine Einarbeitung in den vorhandenen Quellcode einfach ermöglicht wird.

Die minimalen Anforderungen für die Bibliothek sehen wie folgt aus:

- Mindestens 500.000 Würfel gleichzeitig im Speicher zu halten (aber nicht alle auf dem Bildschirm darzustellen).

<sup>1</sup>Minetest veröffentlicht unter der LGPL auf [minetest.net](http://minetest.net)

<sup>2</sup>Terasology veröffentlicht unter der Apache-Lizenz auf [terasology.org](http://terasology.org)

- Eine Welt, bestehend aus 500.000 Würfeln, so zu optimieren, dass sie dargestellt werden kann.
- Einfach zu erweiternder Quellcode.
- Eine dokumentierte API.
- Eine JAR-File, die einfach in anderen Arbeiten nutzbar ist.

## 1.2 Gliederung der Arbeit

Zu Beginn dieser schriftlichen Ausarbeitung werden die technischen Grundlagen auf Seiten der Grafikkarte (OpenGL) und die verwendete Schnittstelle in Java (LWJGL) beschrieben, und ausgeführt, warum diese Technologie verwendet wird. Unter dem Punkt Mathematik werden die mathematischen Verfahren, welche für die 3D-Berechnungen nötig sind erklärt und wie die einzelnen Matrizen aussehen.

Im Kapitel Design wird auf die Architektur und die Struktur der Bibliothek eingegangen. Danach wird die Grafikpipeline vorgestellt und wie diese mit OpenGL funktioniert. Am Ende des Kapitels werden die Schnittstellen der Bibliothek erläutert, und wie man diese anhand eines Beispielprojektes für eigene Projekte einsetzen kann.

Im Hauptteil wird als Erstes das wichtigste Entwurfsmuster für Spiele - der 'Game Loop' - erklärt und wie dieser in der Arbeit umgesetzt ist. Danach werden die einzelnen umgesetzten Elemente in technologisch sinnvoller Reihenfolge beschrieben und die vorgenommenen Optimierungen erläutert.

Nach der Optimierung wird auf die gewonnene Performance eingegangen und diese exemplarisch durch „worst-“ bzw. „best-cases“ gezeigt.

Schlussendlich gibt es im Fazit noch eine Zusammenfassung und einen Ausblick darauf, wie die Umsetzung weiter optimiert werden könnte und welche Erweiterungen der Bibliothek möglich wären.

Im Anhang befinden sich großformatige Bilder. In einzelnen Kaptieln wird Quellcode genutzt, um zu demonstrieren, wie es in der Beispieldokumentation umgesetzt wurde. Alle Quellcode-Beispiele sind soweit wie möglich gekürzt.

# 2 Kapitel 2

# Grundlagen

## 2.1 OpenGL

Da die umgesetzte Implementierung einer würfelbasierten Grafikbibliothek auf wenig externe Komponenten und Konfigurationen angewiesen sein soll, werden, wenn möglich, wenige Abstraktionsebenen verwendet. OpenGL hat dabei folgende Vorteile:

1. es muss weniger Overhead kommentiert werden, da OpenGL sich dicht an der Grafikpipeline befindet und
2. die Schnittstellen für OpenGL sind plattformübergreifend und herstellerunabhängig.

„Der hardwarenahe Entwurf ermöglicht es, dass OpenGL Befehle effizient und schnell an der CPU vorbei, fast ausschließlich von der Grafikkarte ausgeführt werden. Hierin liegt auch einer der Hauptgründe dafür, dass gerade über einen OpenGL-Test die Performance-Unterschiede zwischen einzelnen Grafikkarten deutlich werden.“ (Burggraf2003 [1], S. 22).

Der OpenGL-Standard enthält dabei über 500 unterschiedliche Befehle, um Objekte und Bilder bzw. drei-dimensionale Computergrafik zu beschreiben. OpenGL selbst ist dabei als hardwareunabhängige Schnittstelle entworfen, welche keine Funktionen enthält, um z.B. Eingaben zu verarbeiten oder mit dem Fenstermanager zu interagieren. Um Objekte darzustellen, muss man diese aus simplen geometrischen Primitiven wie Punkten, Linien und Dreiecken konstruieren (vgl. Shreiner2013 [2], S. 2).

OpenGL ist ein Zustandsautomat, das bedeutet einmal gesetzte Einstellungen bleiben erhalten und gelten für die folgenden Operationen.

Das „open“ in OpenGL steht hierbei nicht für „open-source“ also quelloffen. „Open“ ist im Namen enthalten, da es sich um einen offenen Standard handelt, was bedeutet, dass viele Firmen an der Entwicklung beteiligt sind. Einzelne Implementierungen von OpenGL könnten zwar durchaus auch quelloffen sein, aber die meisten Treiber sind es nicht.

## **Alternative zu OpenGL**

Es gibt außer OpenGL noch eine weitere Schnittstelle: DirectX von Microsoft. Diese Schnittstelle ist ebenfalls sehr mächtig und bietet außerdem Unterstützung für zusätzliche Elemente, welche früher oder später auch benötigt werden, wie z.B. Sound, Maus und Tastatur. DirectX kann ebenfalls mit modernen Anforderungen umgehen, wie der häufige Einsatz in der Spieleindustrie beweist.

DirectX hat aber den großen Nachteil, dass die API nur auf dem Betriebssystem Windows funktioniert, im Gegensatz zu OpenGL, das auf jedem beliebigen System eingesetzt werden kann. Da die implementierte Bibliothek aber eben gerade plattformunabhängig sein soll, macht es mehr Sinn, die Bibliothek auf dem betriebssystemübergreifenden Java in Kombination mit OpenGL aufzubauen.

## **2.2 LWJGL**

Da OpenGL nicht direkt in Java verfügbar ist, wird eine Programmbibliothek benötigt, welche es ermöglicht, auf die in C programmierten Schnittstellen von OpenGL zuzugreifen. Hierbei fiel die Wahl schlussendlich auf die „lightweight java game library“ (LWJGL). Der Schwerpunkt liegt hier, wie aus dem Namen schon unschwer zu erkennen, auf Computerspielen. LWJGL bietet keinen sogenannten Szenengraphen<sup>1</sup>, sondern macht nur die OpenGL Funktionen in Java verfügbar. LWJGL funktioniert hier also tatsächlich nur als Schnittstelle (API) zu den von OpenGL bereitgestellten Funktionen. Eines der bekanntesten Programme, das mit LWJGL entwickelt wurde, ist Minecraft.

## **Alternativen zu LWJGL**

Es gibt zwei Alternativen zu LWJGL: Java 3D und JOGL. Ersteres hat ein gänzlich anderes Konzept und verfolgt auch ein anderes Ziel.

Java 3D bietet einen eigenen Szenengraph an, welcher einen Einstieg in die Entwicklung einfacher macht und bietet generell ein höheres Abstraktionslevel als z.B. LWJGL an. Durch diese Abstraktion ist ein Einstieg zwar einfacher, aber es schränkt auch die Möglichkeiten ein, eigene Datenstrukturen zu nutzen. Ein direkter Zugriff auf OpenGL Funktionen ist nicht vorgesehen. Da die Bibliothek möglichst flexibel sein soll, ist diese vorgegebene Struktur durch Java 3D bereits ein Ausschlusskriterium.

Java Bindings for OpenGL (JOGL) ist ähnlich wie LWJGL eine Sammlung von Java-Wrapperklassen, welche den nativen Zugriff auf die Funktionen von OpenGL erlaubt. Zum aktuellen Zeitpunkt ist es so, dass LWJGL die aktuellen OpenGL Versionen

---

<sup>1</sup>Der Szenengraph beschreibt ganz allgemein eine Datenstruktur, welche Elemente einer virtuellen Welt beinhaltet.

schneller unterstützt. JOGL ist teilweise auch weniger hardwarenah und hält sich mehr an die Standard Java Objektorientierung. LWJGL hingegen hält sich strikter an den prozeduralen OpenGL-Standard.

JOGL ist genau wie LWJGL in der Lage, die gestellten Anforderungen an eine OpenGL-Schnittstelle zu erfüllen. Ob man JOGL oder LWJGL bevorzugt, ist Geschmackssache. Der offensichtlichste Unterschied ist der statische Zugriff auf die OpenGL Funktionen bei LWJGL, gegenüber dem Zugriff über einen OpenGL-Kontext-Object bei JOGL.

Sollte in der Zukunft der Aktualitätsvorteil von LWJGL sich zu JOGL verschieben oder die prozedurale Umsetzung des OpenGL-Standards in der Bibliothek nicht mehr gewünscht bzw. benötigt sein, wäre es möglich, die Bibliothek mit relativ geringem Aufwand auf JOGL zu portieren.

## **Warum keine vorhandene Spielengine nutzen?**

Inzwischen gibt es für Java auch diverse Engines, welche die Entwicklung stark beschleunigen können. Beispiele wären hier die jMonkeyEngine, die auf LWJGL und JOGL aufbaut und die libGDX, welche ebenfalls LWJGL nutzt. Eine vorhandene Spielengine bietet oftmals viele Vorteile wenn man ein Spiel, oder zumindestens eine 3D-Anwendung entwickeln möchte:

- Multiplattform Ausrichtung,
- diverse Editoren und Tools,
- Abstraktion der Hardwareschnittstellen,
- physikalische Berechnungen,
- Scriptingmöglichkeiten,
- ausführliche Dokumentation.

Während man bei der Entwicklung von Spielen oft ein fertiges Produkt als Ziel hat, und der Weg dahin möglichst schnell gehen soll, da er direkt mit Kosten verbunden ist, ist das Ziel bei der Bachelorarbeit kein fertiges Produkt. Aus diesem Grund müssen einzelne Vor- und Nachteile also anders abgewogen werden, als dies bei einer reinen Kosten-Nutzen-Analyse der Fall wäre.

Eine Multiplattform Ausrichtung ist für die kommerzielle Spieleentwicklung unerlässlich um zu gewährleisten, dass man seine Spiele ohne großen Aufwand auf Plattformen wie iOS und Android portieren kann. Für die umgesetzte Bibliothek wäre eine Multiplattform Unterstützung zwar nicht direkt hinderlich, aber sie wird zum aktuellen Zeitpunkt, und vermutlich auch später, nicht benötigt.

Vorhandene Editoren, wie z.B. ein Level Editor oder Tools zum Erstellen von 3D-Modellen, stellen nicht nur keinen Mehrwert für diese Bibliothek dar, sondern schränken

den Entwicklungsprozess zusätzlich ein. Java & LWJGL ermöglichen eine Entwicklung komplett unabhängig vom System und unterstützen zur Laufzeit praktisch alle größeren Betriebssysteme.

Hardwareschnittstellen sind oftmals komplex oder unhandlich zu bedienen. Diese zu verstecken klingt im ersten Moment natürlich sehr reizvoll, da dann „das Rad nicht neu erfunden“ werden muss. Das bedeutet aber auch, dass, wenn eine Schnittstelle direkt benutzt werden muss, diese im besten Fall von der Engine auch direkt zu Verfügung gestellt wird. Im schlechtesten Fall ist ein Zugriff mit den vorhandenen Möglichkeiten überhaupt nicht möglich, und man muss sich mit alternativen Lösungen begnügen.

Um diesen Problemen von Anfang an aus dem Weg zu gehen, wurde die minimalste Lösung mit den meisten Freiheiten gewählt. Da die vorhandenen Spielengines alle LWJGL nutzen, ist es einfacher gleich LWJGL selber zu nehmen. LWJGL bietet einen direkten und schnellen Zugriff auf die OpenGL-Funktionalität an.

## GLFW

OpenGL selbst bietet keine Mechanismen an, um Fenster oder Eingaben zu verwalten, weshalb es mehrere Bibliotheken gibt, welche diesen Zweck erfüllen. Das prominenteste Beispiel ist hier das OpenGL Utility Toolkit (GLUT), das aber nur eine Schnittstelle für C, C++, Fortran und Ada anbietet, und seit längerem nicht mehr weiterentwickelt wird. LWJGL unterstützt GLFW, eine kleine, in C geschriebene Bibliothek für OpenGL, die es mir ermöglicht, Fenster mit Java zu erzeugen, zu verwalten und Eingaben zu verarbeiten.

## 2.3 Mathematik

Jedes Objekt in der Spielwelt wird in OpenGL durch Punkte repräsentiert, die in einem dreidimensionalen Koordinatensystem liegen. Dieses Koordinatensystem ist ein sogenanntes kartesisches Koordinatensystem. Das kartesische Koordinatensystem ist orthogonal, was bedeutet, dass alle Achsen rechtwinklig sind: die x-Achse ist der untere Bildschirmrand, die y-Achse ist der linke Bildschirmrand und in den Monitor blickt man entlang der negativen z-Achse (vgl. McShaffry2012 [3], S. 478 ff.).

Um die Koordinaten eines Objekts in ein Weltkoordinatenystem bzw. schlussendlich in ein 2D-Koordinatensystem umzuwandeln, sind Transformationen in der Computergrafik außerordentlich wichtig. Dies trifft vor allem auf Spiele oder andere Echtzeitanwendungen mit grafischer Ausgabe zu.

Mit diversen Transformationen wird es ermöglicht, Objekte an unterschiedlichen Positionen in der Spielwelt zu platzieren, oder dessen Größe und Form zu verändern. Die folgenden Transformationen werden in der Bibliothek unterschieden:

- Die wichtigste Transformation ist die Verschiebung. Durch diese wird ein Objekt in der Spielwelt auf einer oder mehreren Achsen um einen frei wählbaren Wert in einer beliebigen Richtung verschoben.
- Eine weitere Transformation ist die Skalierung. Unter Skalierung versteht man das Ändern der Größe eines Objektes in Relation zum Weltkoordinatensystem.
- Die letzte Transformation ist die Drehung. Diese bestimmt, wie das Objekt anhand einzelner Achsen rotiert werden kann.

Alle diese Transformationen werden durch Matrixmultiplikationen ermöglicht und finden im Shader (siehe Grafikpipeline) statt. In Abb. 2.1 sind alle drei unterschiedlichen Transformationen visualisiert.

Unter a) sieht man eine Verschiebung auf der x-Achse. Wenn also ein Würfel an einer anderen Stelle angezeigt werden soll, muss er auf den jeweiligen Achsen um den Wert verschoben werden an dem er dargestellt werden soll. Diesen Vorgang nennt man Translation.

Danach sieht man auf der Abbildung eine Skalierung (b). Wenn einzelne Objekte größer oder kleiner dargestellt werden sollen, kann man diese skalieren, also ihre Größe im fertigen Bild ändern. In einer Würfelwelt werden meistens alle Würfel auf den gleichen Wert skaliert um zu gewährleisten, dass sie in der gleichen Größe dargestellt werden bzw. von vornherein gleich groß sind.

Zuletzt ist auf der Abbildung eine Rotation um die x-Achse dargestellt (c). Rotationen werden für die Würfel in einer Würfelwelt zwar nicht benötigt, da hier alle Würfel orthogonal zueinander sein sollen, können aber für andere Objekte in der Spielwelt durchaus sinnvoll sein.

Ein Punkt eines Dreiecks, das auf dem Bildschirm angezeigt werden soll, durchläuft dabei folgende Schritte (siehe Abb. 2.2):

1. Der Punkt wird in der Model-View Matrix transformiert (von *local space* nach *world space* und von *world space* nach *eye space*).
2. In der Projektionsmatrix wird das Ergebnis auf eine zweidimensionale Scheibe projiziert (von *eye space* nach *homogeneous clip space*).
3. Es wird die View Frustum (der „sichtbare“ Bereich) berechnet. (von *homogeneous clip space* nach *normalised device space*)
4. Danach werden die Koordinaten des projizierten Punktes in Koordinaten für den FrameBuffer umgerechnet. (von *normalised device space* nach *viewport space*)

Im Folgenden wird nur auf die Schritte 1. und 2. genauer eingegangen, da das Berechnen der View Frustum und die Umrechnung für den FrameBuffer von OpenGL übernommen werden.

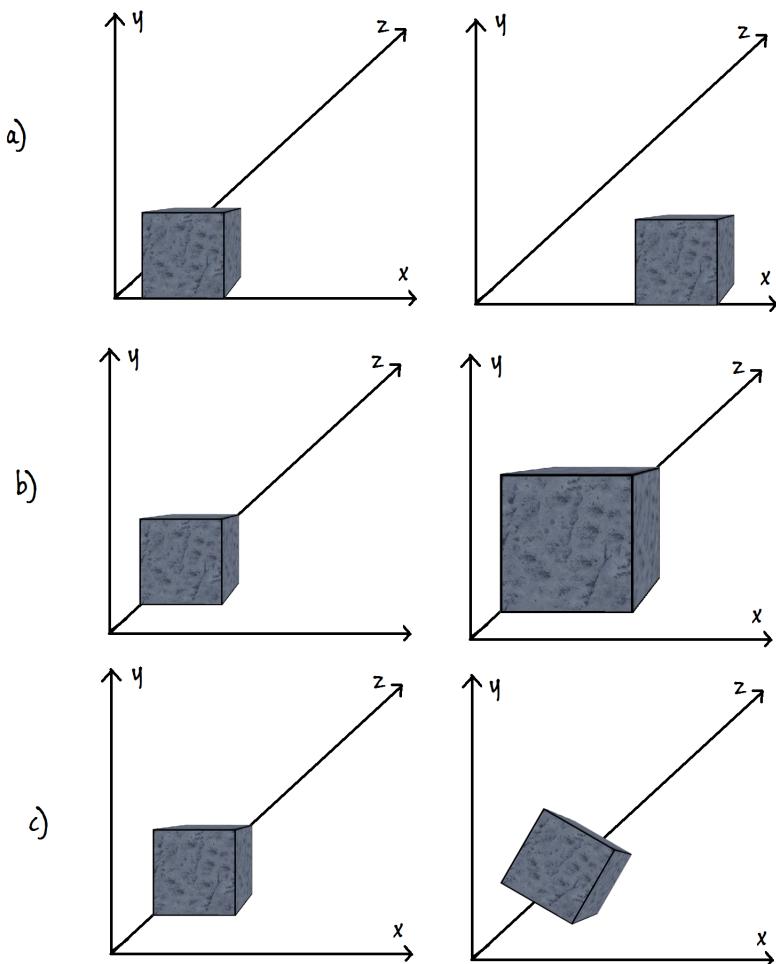


Abb. 2.1: Die unterschiedlichen Transformationen am Beispiel eines Würfels.

Um die sogenannte Weltkoordinate eines Punktes zu berechnen, erstellt man die passende Transformationsmatrix aus der Translationsmatrix (zur Verschiebung), der Skalierungsmatrix (zur Änderung der Größe) und den Rotationsmatrizen (zur Drehung). Diese so gewonnene Transformationsmatrix multipliziert man mit dem Vektor und erhält so die sogenannte *transformed vertex position*.

Die Weltkoordinaten multipliziert man dann mit der Kameramatrix (auch *view matrix* genannt) und erhält dadurch einen Positionsvektor in Relation zur Kamera. Um diesen in Relation zur Größe des Fensters bzw. des Bildschirms zu setzen, multipliziert man den erhaltenen Vektor mit der sogenannten Projektionsmatrix (auch *projection matrix* genannt).

Den so erhaltenen Vektor übergibt man an OpenGL, welches dann eine „perspective division“ durchführt, in der der Vektor in einen genormten Vektor überführt wird, der sich zwischen -1 und 1 befindet. Durch diese Division werden Vektoren, die sich weiter hinten in der Szene befinden (negative z-Achse) näher zur Mitte des Bildschirms transferiert. Diesen Vektor kann OpenGL dann anhand der z-Koordinate in Relation

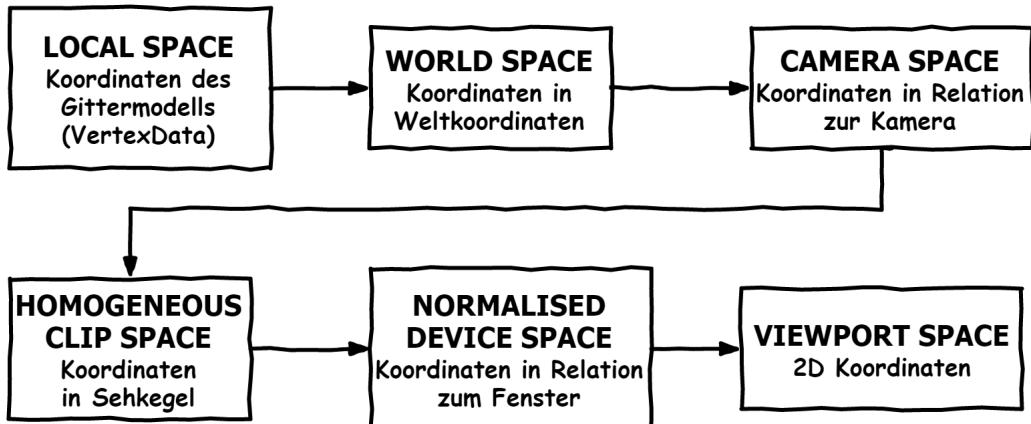


Abb. 2.2: Der Weg vom Vertex bis zum auf dem Bildschirm ausgegebenen Koordinaten (vgl. Gerdelen2014 [4]).

zu anderen Vektoren sortieren, um nur diejenigen zu zeichnen, die nicht von anderen Punkten überlagert werden.

# 3 Kapitel 3

# Design

## 3.1 Architektur

### Verwaltung der Welt

Da mit der Grafikbibliothek möglichst viele Würfel aus einer potentiell unendlich großen Welt darstellbar sein sollen, ist das Speichern und Verwalten der „Welt“ nicht trivial. Eine quadratische Welt umfasst dabei je nach Größe schnell einige Millionen Würfel (vgl. Tabelle 3.1).

Da eine der Anforderungen darin besteht, mindestens 500.000 Würfel gleichzeitig zu laden (nicht darzustellen), war sehr schnell klar, dass nicht einfach für jeden Würfel ein neues Objekt angelegt werden kann, sondern eine Datenstruktur benötigt wird, die es ermöglicht, eine Welt zu verwalten.

n	$n^3$	Würfel gesamt	Speicher
1	1	1	4 Byte
10	$10^3$	1.000	~3,9 Kb
40	$40^3$	64.000	250 Kb
100	$100^3$	1.000.000	~3,8 MB
500	$500^3$	125.000.000	~476 MB
1000	$1000^3$	1.000.000.000	~3,8 GB

Tab. 3.1: Würfel in einer quadratische Welt, angenommen ein Würfel benötigt 4 Byte (Enum).

Um also eine potentiell unendlich große Welt verwalten zu können, muss diese in kleine Teile (bei Minecraft und im Folgenden „Chunks“ genannt) zerlegt werden, welche dann nur jeweils wenige Blöcke<sup>1</sup> enthalten. Wenn diese Teile nun nur jeweils aus  $16 * 16 * 16$  Würfeln bestehen, werden pro Chunk nur wenige KB benötigt und sobald man sich weit genug von einem Chunk entfernt, kann dieser komplett aus dem Speicher entfernt werden.

Wenn man nun eine Sichtweite von 96 Blöcken erreichen möchte, muss man sechs Chunks mit jeweils 16 Blöcken in jede Richtung und den Chunk, auf dem man aktuell

<sup>1</sup>Wenig ist hier relativ, bei Minecraft handelt es sich um 65.536 Würfel

steht, laden. Da man dies in jede Richtung tun muss, ergibt sich folgende Formel:  $(6 + 6 + 1)^3 = 2.197$  Chunks oder allgemein:  $(n + n + 1)^3$ . Wenn man quadratische Chunks der Größe 64 benutzt, werden für eine Sichtweite von 100 Blöcken nur noch  $(2 + 2 + 1)^3 = 125$  Chunks benötigt. In Abb. 3.1 ist dieses Beispiel zweidimensional visualisiert. Das Feld S stellt hierbei den Spieler bzw. die Kamera dar. Die grauen Felder sind die Chunks, welche im Speicher gehalten werden müssen.

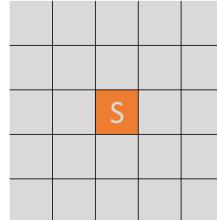


Abb. 3.1: Jedes Quadrat repräsentiert einen Chunk, wenn man eine Sichtweite von zwei Chunks hat. Die Chunks werden hier nur zweidimensional dargestellt. In der Praxis müssen diese Chunks auch noch für die dritte Dimension geladen werden.

Um Teile der Welt im Speicher zu halten, wurden zwei unterschiedliche Verfahren gefunden:

1. ein dreidimensionales Array, in dem jeder Würfel durch einen Integer (bzw. Enum) repräsentiert wird.
2. Ein 'Octree', ein Baum, in dem jeder Knoten genau acht bzw. keinen direkten Nachfolger hat.

Das dreidimensionale Feld hat hierbei den großen Vorteil, dass es sehr intuitiv ist und die Zugriffszeiten konstant sind. Der Nachteil besteht darin, dass große (quadratisches) Flächen aus nicht vorhandenen Würfeln (Luft) trotzdem Speicher benötigen und beim Durchlaufen des Feldes mit Schleifen eventuell sogar Rechenzeit während der grafischen Darstellung verbrauchen.

Der Octree ist in der Nutzung weniger intuitiv, aber macht es leichter, ganze Bereiche, die nur aus nicht vorhandenen Würfeln (oder dem gleichen Material) bestehen, einfach wegfällen zu lassen bzw. mit einem Knoten im Baum zu repräsentieren.

Da der Speicherbedarf aber durch die Umsetzung von Chunks nicht mehr problematisch ist, und die konstanten Zugriffszeiten beim Optimieren des Gittermodells sehr nützlich sind, wurde die Bibliothek mit einem dreidimensionalen Array umgesetzt.

Die Chunkgröße wirkt an dieser Stelle noch relativ willkürlich. Man könnte beispielsweise auf die Idee kommen, die Größe des Chunks einfach nach der gewünschten Sichtweite zu wählen, dann müsste man in jede Richtung nur noch einen Chunk laden, zusätzlich zu dem, auf dem man sich befindet. Bei geringen Sichtweiten stellt dies auf modernen Rechnern tatsächlich eine mögliche Lösung dar.

Da aber, aus später ausführlicher erklärten Gründen, ein Chunk bei jeder Änderung komplett neu aufgebaut werden muss, ist die Chunkgröße durchaus relevant. Hier gilt es die Größe so zu wählen, dass Änderungen am aktuellen Chunk ohne Probleme möglich sind, also schnell genug, dass die Ausgabe flüssig bleibt.

Außerdem müssen Chunks schnell genug nachgeladen werden, wenn der Spieler sich bewegt und klein genug sein, um Änderungen schnell speichern zu können.

## 3.2 Grafikpipeline

Der Begriff „rendern“, welcher bisher benutzt wurde, ohne ihn genau zu definieren, meint, wenn Bilder aus Objekten generiert werden. Objekte bestehen hierbei aus einfachen geometrischen Primitiven: Linien, Punkte und Dreiecke. Diese Primitiven werden durch Koordinaten angegeben, welche im Folgenden *Vertices* (*singular: Vertex*) genannt werden.

Um diese Vertices bzw. die daraus zusammengesetzten Objekte in den Speicher der Grafikkarte zu laden, gibt es mehrere Möglichkeiten mit unterschiedlichen Vor- und Nachteilen, welche im weiteren Verlauf erläutert werden. Mit „Grafikpipeline“ ist der komplette Prozess gemeint, wie er in Abb. 3.2 dargestellt ist.

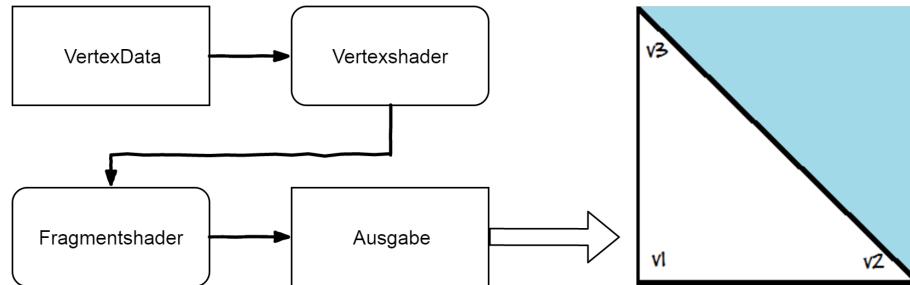


Abb. 3.2: Die von mir genutzte Grafikpipeline, vereinfacht dargestellt. Alle Zwischen- schritte zwischen Vertex- und Fragmentshader wurden nicht explizit genutzt.

Folgende Schritte werden bei der Berechnung eines Bildes mit der OpenGL-Technologie durchlaufen. Der Ablauf ist weitaus identisch mit der Arbeitsweise anderer Grafik-Bibliotheken:

1. OpenGL wird initialisiert.
  2. Es wird ein Array erstellt, in welchem Objekte (in meinem Fall ein Gittermodell der Welt) aus Polygonen abgelegt werden. In weiteren Arrays können Texturkoordinaten und die Normalen des Polygons abgelegt werden. Normale (auch Normalvektoren) werden genutzt, um festzustellen, ob eine Fläche dem Benutzer zugewandt ist oder nicht, um letztere dann von der Bildberechnung auszuschließen

(Back-Face Culling, siehe Optimierungen des Gittermodells). Des Weiteren werden sie zur Berechnung von Lichteinfall und Reflexionen benötigt. Ein weiteres Array ordnet die Indizes des Vertex-Arrays zu Dreiergruppen zusammen und gibt die Reihenfolge an, in der die Punkte zu Dreiecken zusammengesetzt werden. Diese Arrays werden dann zusammengefasst als `VertexArrayObject` (VAO) bezeichnet.

3. Die erstellten Arrays werden dann an die Grafikkarte übergeben. Für jeden Vertex wird ein Shaderprogramm, der so genannte Vertex-Shader, aufgerufen, der Operationen auf jedem einzelnen Vertex ausführen kann. Hier können Manipulationen an den Formen des Objekts vorgenommen werden. Außerdem wird mit Hilfe einer Matrixmultiplikation die Projektion des dreidimensionalen Punkts auf die zweidimensionale Fläche zur Anzeige auf dem Bildschirm berechnet.
4. Anschließend wird in der Grafikkarte das Bild rasterisiert. Dabei wird die zweidimensionale Projektion mit einem Pixelraster der Größe des Bildschirms überlagert und jeder Punkt einem Pixel zugeordnet.
5. Für jedes berechnete Pixel wird nun ein weiteres Unterprogramm aufgerufen: Der Fragment-Shader kann die Farbwerte des jeweiligen Pixels verändern und sorgt somit für die Darstellung von Texturen, Oberflächenmaterialien, Lichtern und Schatten.
6. Schlussendlich wird das errechnete Bild im Frame-Buffer der Grafikkarte abgelegt.

Im Folgenden wird auf die einzelnen Punkte nun genauer eingegangen und erläutert, wie diese umgesetzt wurden bzw. welche Alternativen es zu den gewählten Umsetzungen gibt.

## Initialisierung

Bevor etwas mit OpenGL dargestellt werden kann, müssen wie in Schritt 1 angekündigt einzelne Parameter für OpenGL konfiguriert werden:

- `glEnable(GL_DEPTH_TEST)`

Der Z-Buffer (Tiefe) wird benutzt, damit Pixel in der richtigen Reihenfolge gezeichnet werden. Wenn er nicht aktiviert wäre, würden z.B. Hinterseiten des Würfels nach den Vorderseiten gezeichnet werden wodurch die Vorderseite nicht oder nur zum Teil sichtbar wäre.

- `glEnable(GL_CULL_FACE) & glCullFace(GL_BACK)`

Mit dieser Einstellung werden Dreiecke, welche komplett verdeckt werden, nicht gezeichnet. Hinterseiten von Würfeln müssen z.B. nicht mehr gezeichnet werden (siehe Bilder des Gittermodells im Anhang).

- `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)`

Durch diesen Befehl werden einzelne Buffer wieder gelöscht. Hier werden der color und der depth buffer gelöscht. Da es sich bei OpenGL um einen Automaten handelt, ist es üblich, in einem sauberen Zustand zu starten.

- `glClearColor(255.0f, 255.0f, 255.0f, alpha)`

Hier wird festgelegt, mit welcher Farbe der color buffer beschrieben werden soll, wenn er gelöscht wird. Da alle Farben (rot, grün, blau) hier auf Maximum gesetzt sind, handelt es sich um weiß.

## Erstellen der Polygone

Um nun die Daten der Würfel speichern zu können, müssen Buffer mit der Java Methode `BufferUtils.createFloatBuffer(int size)` für Vertices, Indices, Normalen und Texturkoordinaten erstellt werden. Diese Buffer werden dann für jeden Würfel mit den entsprechenden Daten gefüllt (hierzu auch Abb. 4.8 im Kapitel Gittermodell).

In weiteren Verlauf dieses Kapitels kann der Einfachheit halber davon ausgegangen werden, dass der erstellte Buffer mit `buffer.put(float)` befüllt wurde und nun die darzustellenden Vertices enthält. Wie diese Vertices „erzeugt“ werden, wird in den späteren Kapiteln sehr ausführlich beschrieben.

## Übergabe an die Grafikkarte

Die folgenden Möglichkeiten bietet OpenGL an:

1. Immediate Mode (veraltet)
2. Display Lists (sehr schnell)
3. Vertex Arrays und Vertex Buffers (das gewählte Verfahren)

Im Folgenden wird nun jeweils kurz skizziert, wie die unterschiedlichen Möglichkeiten funktionieren, und warum die Entscheidung auf Vertex Arrays und Vertex Buffers fiel.

## Immediate Mode

Das unmittelbare Darstellen (immediate mode) ist seit OpenGL 3.0 veraltet und wurde in 3.1 komplett entfernt. Das alleine disqualifiziert diesen Modus für dieses Projekt bereits, wenn mit der aktuellen LWJGL Version gearbeitet werden soll. Um den Unterschied zu der schlussendlich umgesetzten Lösung aufzuzeigen, folgt aber trotzdem ein kurzes (sehr einfaches) Beispiel. Dieses einfache Beispiel ist in Abb. 3.3 zu sehen. Hier werden drei Vertices zu einem Dreieck verbunden und dann als weißes Dreieck gezeichnet.

Die dafür benötigten Befehle sind relativ gering, da jeder Vertex direkt angegeben, und an die Grafikkarte übertragen wird. Dadurch ist die Bedienung des „immediate

```

1 glBegin(GL_TRIANGLES);
2     glColor3f(255.0f, 255.0f, 255.0f) //Farbe: Weiß
3     glVertex3f(0.0, 0.0, 0.0); //v1
4     glVertex3f(1.0, 0.0, 0.0); //v2
5     glVertex3f(0.0, 1.0, 0.0); //v3
6 glEnd();

```

Abb. 3.3: Weißes Dreieck: ein Dreieck im „immediate mode“.

mode“ ohne viel Einarbeitung in die OpenGL Funktionalität möglich. Um hier größere Mengen an Dreiecken zu übertragen, könnte man z.B. einfach mit einer Schleife durch die Vertices laufen, um diese nacheinander zu übertragen.

Der „immediate mode“ hat aber auch einen großen Nachteil: Vertex-Informationen müssen während der Darstellung jedesmal neu in den Grafikspeicher geladen werden. Dieser Vorgang benötigt während jedem Frame Zeit und eignet sich dementsprechend nicht für meinen Einsatz.

## Display Lists

*Display Lists* lösen dieses Problem, indem im Grunde eine Sequenz von `glBegin` bis `glEnd` auf dem Grafikspeicher effizient zwischengespeichert wird. Dadurch handelt es sich um eine der schnellsten Möglichkeiten zur Darstellung, solange die genutzten Daten statisch sind.

Dies ist aber auch der größte Nachteil: sobald eine Display List auf der Grafikkarte gespeichert wurde, kann sie nicht mehr verändert werden.

## Vertex Arrays und Vertex Buffers

Statt einzelne Vertices wie im „immediate mode“ zwischen `glBegin()` und `glEnd()` zu definieren, kann man diese auch in einzelnen Arrays speichern, welche dann die jeweiligen Daten (Vertex, Index, Normale und Texturkoordinaten) enthalten. Diese Arrays kann man dann an OpenGL weitergeben, um sie zu zeichnen.

Vertex Buffer Objects (kurz VBOs) sind mehr oder weniger eine Erweiterung der Vertex Arrays. Vertex Buffer Objects besitzen aber den Vorteil, dass ihre Vertexdaten serverseitig, also im schnellen Speicher der Grafikkarte, statt wie bei den VAs im Hauptspeicher abgelegt werden. Das ist bei einer Displayliste zwar (fast) genauso der Fall, allerdings können die Daten nicht so einfach dynamisch geändert werden, was bei einem VBO aber bei Bedarf sehr einfach ist.

VBOs vereinen also die Flexibilität eines Vertexarrays mit der Geschwindigkeit von Display Lists und eignen sich daher besonders für das Rendern aufwändiger Geometrie, egal ob statisch oder dynamisch (vgl. Shreiner2013 [2], S. 19).

In einem VAO wird gespeichert, welches Datenformat die Vertices haben, die gerendert werden sollen (siehe Abb. 3.4). Also z.B., dass ein Vertex aus 3 floats für die Position, 3 floats für den Normalenvektor und 2 floats als Texturkoordinaten besteht.

Außerdem merkt sich das VAO, woher (aus welchem VBO) diese Daten genommen werden sollen. Den unter Umständen verwendeten Indexbuffer merkt sich das VAO ebenfalls, falls dieser angegeben wurde.

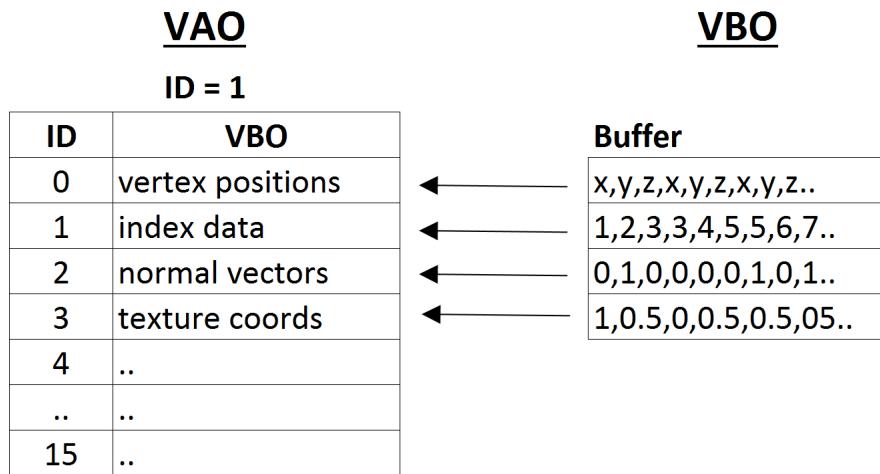


Abb. 3.4: Vertex Array Objects enthalten die ID mehrere Vertex Buffer Objects in welchen wiederum die Daten zur Darstellung liegen.

Wenn man nun das Dreiecks-Beispiel aus dem Abschnitt „immediate mode“ mit Vertex Array Objekten erstellen will, benötigt man zwar initial einen größeren Aufwand um die Buffer alle zu erstellen und an die Grafikkarte zu übertragen, kann dafür aber einen schnelleren und einfacheren `glDraw()` Befehl im rendering nutzen.

In Abb. 3.5 ist das Erstellen und Befüllen in der Methode `setup()` sichtbar und der `glDraw()` Befehl in der Methode `render()`. Die genutzten Methoden und deren OpenGL Funktionen werden nun im Weiteren erläutert.

```

1 int vao, vbo;
2 void setup(){
3     vao = glGenVertexArrays(); //ID des VAO von OpenGL anfordern
4     glBindVertexArray(vao); //VAO nutzen
5     vbo = glGenBuffers(); //VBO erstellen
6     glBindBuffer(vbo); //VBO nutzen
7     glBufferData(vertices); //Buffer an OpenGL schicken
8     glVertexAttribPointer(..); //Struktur an OpenGL übertragen
9     glEnableVertexAttribArray(vbo); //Buffer aktivieren
10 }
11
12 void render(){
13     glBindVertexArray(vao);
14     glDrawElements(...);
15 }
```

Abb. 3.5: Erstellen und Rendern eines Dreiecks mit VAOs und VBOs.

Initial muss OpenGL einen Vertex Array erstellen. Die ID (ein Integer) des Arrays wird dann gespeichert, da mit dieser immer wieder auf das Vertex Array zugegriffen werden kann. Direkt danach wird dieser Array „gebunden“, also zur Nutzung markiert. Da OpenGL ein Zustandsautomat ist, werden alle weiteren Änderungen bis zum nächsten „binden“ an diesem Array vorgenommen.

Um Daten im Speicher der Grafikkarte zu speichern, wird ähnlich wie beim Vertex Array Objekt nun ein Vertex Buffer bzw. die ID eines Vertex Buffers angefordert. Dieser Buffer wird nun ebenfalls „gebunden“ wodurch der folgende `glBufferData(vertices)` Aufruf sich auf den erstellten VBO bezieht. Vertices ist ein `FloatBuffer` von Java, in welchem die einzelne Vertices der Reihenfolge nach enthalten sind, welche mit diesem Befehl in den Grafikspeicher geladen werden.

Damit OpenGL weiß, um welche Daten es sich handelt, muss für den Buffer auch noch angegeben werden, aus wie viele Variablen ein Punkt besteht (in meinem Beispiel drei für x,y und z) und um was für Variablen es sich handelt (in meinem Beispiel Floats).

Mit  `glEnableVertexAttribArray(vbo)` wird der Buffer schließlich zur Darstellung „aktiviert“. Dies könnte man auch während des Renderns machen, falls man z.B. nicht alle oder unterschiedliche Buffer des Vertex Array nutzen möchte (z.B. unterschiedliche Texturkoordinaten).

Wie man an diesem Beispiel sieht, ist der Aufwand zum Erstellen der einzelnen Objekte deutlich höher als im „immediate mode“. Warum es sich trotzdem lohnt, sieht man in der Rendermethode: es reicht den Vertex Array zu aktivieren und ihn zeichnen zu lassen. Die Daten liegen bereits alle im schnellen Speicher der Grafikkarte, und es müssen keine einzelnen Vertices mehr zeit- und codeaufwändig übertragen werden.

Das Ergebnis beider Codefragmente ist also das Gleiche, allerdings führt die Nutzung von VBOs aus den oben genannten Gründen zu einer schnelleren Visualisierung. Dies ist vor allem bei großen Datenmengen oder häufiger Aktualisierung der Grafik von Vorteil und der Grund, warum die Entscheidung auf dieses Verfahren gefallen ist.

Wie zu Anfang des Kapitels erklärt, gibt es in der Grafikpipeline zwei wichtige Shader: Den Vertexshader und den Fragmentshader. Es gibt zwar noch weitere Shader, welche aber nicht genutzt wurden (z.B. Tessellationshader und Geometryhader).

OpenGL Shaders sind kleine „mini“ Programme, welche in der OpenGL Shading Language (kurz GLSL) geschrieben werden. GLSL weist viele Ähnlichkeiten mit C auf, wodurch der Code sehr einfach zu lesen ist, wenn man mit C vertraut ist (vgl. Sellers2013 [5], S. 17).

Wenn man etwas zeichnet, führt die Grafikkarte diese Shader der Reihe nach aus, und nutzt die Ausgabe des jeweils vorherigen Shaders als Eingabe für den nächsten Shader (siehe hierzu Kapitel Shader).

## **Vertexshader**

Für jeden zu zeichnenden Vertex, wird der Vertexshader aufgerufen, um diesen Vertex zu verarbeiten. Im Vertexshader werden alle Transformationen vorgenommen und die neue Position des Vertex über die Kameramatrix (auch *view matrix* genannt) errechnet.

## **Fragments shader**

Der zweite wichtige Shader in der Grafikpipeline ist der Fragments shader. Dieser wird genutzt, um für jeden Pixel auf dem Bildschirm die Farbe zu bestimmen. Deswegen wird er auch *Pixelshader* genannt. Der umgesetzte Fragments shader ist sehr einfach und die einzige Aufgabe des Shaders ist es, Texturen an der richtigen Stelle zu zeichnen, bzw. in späteren Versionen der Bibliothek den Lichteinfall zu berechnen.

Auch wenn es durchaus vorstellbar ist, mehrere Vertexshader oder Fragments shader zu nutzen, kann zu jedem Zeitpunkt immer nur ein Einziger aktiv sein. Man könnte aber z.B. einen anderen Shader für das Interface, den Himmel oder andere Objekte nutzen. Es wäre auch vorstellbar, den Shader bei einer anderen Systemkonfiguration auszutauschen, um eine höhere Leistung bzw. anspruchsvollere Effekte zu erzielen.

## **3.3 Beispielprojekt**

Um mit der Bibliothek zu arbeiten, liegt diese als Archiv vor. Dieses Archiv erlaubt es z.B. Studenten, die gerade erst anfangen, Java zu lernen, ihre Ausgabe nicht nur in der Komandozeile zu bewundern, sondern die grundlegenden Funktionen von Programmiersprachen (Schleifen, Abfragen etc.) zu visualisieren. In diesem Kapitel wird erklärt, wie man dies unter Eclipse bewerkstelltig, und einfache Beispiele gezeigt, welche die erstellte Bibliothek nutzen.

Da der Einsatz der Bibliothek in jedem Schritt nachvollziehbar sein soll, enthält dieses Kapitel sehr viele Quellcode Beispiele.

Um die Bibliothek in einem eigenen Projekt zu nutzen, muss die JAR-Datei als Library in Eclipse hinzugefügt werden, dies geht über: *Properties → Java Build Path → Libraries → Add JAR*. Nachdem man sich ein neues Gameobjekt erstellt hat, kann man über dieses die Configuration anpassen, die Welt initialisieren und die Ausgabe dann mit `start()` starten.

Im Folgenden werden zwei Möglichkeiten vorgestellt, um ein Schachbrett zu erzeugen, im Speicher abzulegen und dieses dann auszugeben. Die Ausgabe soll dabei immer der in Abb. 3.6 dargestellten Ausgabe gleichen.

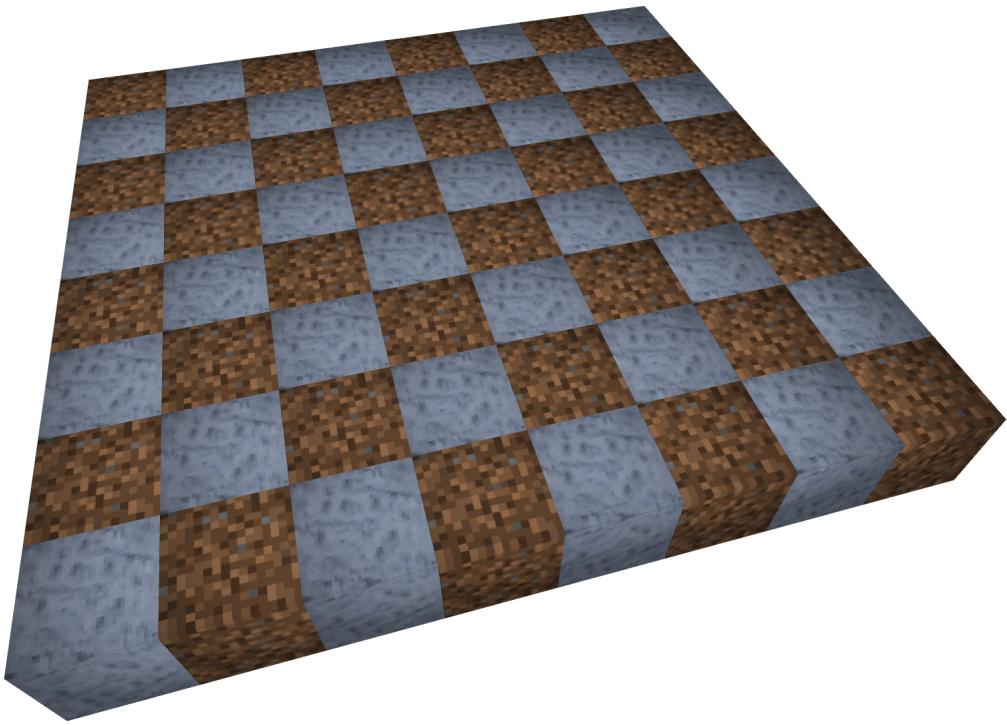


Abb. 3.6: Eine Welt, bestehend aus einem Schachbrett, durch zwei „for“ Schleifen erstellt.

### Würfel von außen manipulieren

Die einfachste Möglichkeit, um eine eigene Welt zu erstellen und mit der erstellten Bibliothek anzuzeigen ist in Abb. 3.7 zu sehen. Hierbei wird als Erstes eine neue Konfiguration erstellt und die maximale Weltgröße auf 8 Blöcke in jede Richtung begrenzt ( $8^3 = 512$ ). Danach wird ein neues Game-Object erstellt, welches die erstellte Konfiguration nutzt.

Mit `new World(size)` wird eine Welt aus 512 Blöcken erstellt, in welcher das Schachbrett erzeugt werden soll. Diese Welt ist komplett leer bzw. enthält bisher noch keine Blöcke.

In den folgenden Schleifen werden die Blöcke nun durch ein `world.setBlock(x,y,z,type)` geändert bzw. erstellt, um das typische Muster eines Schachbrettes zu erzeugen.

Um die so erstellte und modifizierte Welt schlussendlich anzuzeigen, wird sie initialisiert und die Ausgabe dann gestartet. Da man hier keinen Zugriff auf den sogenannten *Main Loop* hat, ist es nicht möglich, während der Anzeige die Welt zu verändern (mehr siehe Kapitel Main Loop).

```

1 import de.oth.blocklib.*;
2 public class main {
3     public static void main() {
4         Configuration config = new Configuration();
5         config.setWorldSize(8);
6         Game game = new Game(config);
7         World w = new World(config.getWorldSize());
8         boolean chess = true;
9         for (int i = 0; i < config.getWorldSize(); i++) {
10             for (int j = 0; j < config.getWorldSize(); j++) {
11                 if (chess) {
12                     w.setBlock(i, 0, j, BlockType.Stone);
13                     chess = !chess;
14                 } else {
15                     w.setBlock(i, 0, j, BlockType.Dirt);
16                     chess = !chess;
17                 }
18             }
19             chess = !chess;
20         }
21         game.initWorld(w);
22         game.start();
23     }
24 }
```

Abb. 3.7: Ein Beispiel für die Nutzung der Bibliothek in der nicht von Klassen geerbt wird. Alle Würfel werden in den „for“ Schleifen gesetzt, um die Ausgabe aus Abb. 3.6 zu erzeugen.

### Abgeleitete Klasse nutzen

Um auch während der Laufzeit Änderungen an der eigenen Welt vornehmen zu können, reicht es nicht aus, die Bibliothek einzubinden. Hierfür muss man zusätzlich von den jeweiligen Klassen erben, um eine eigene Spielklasse zu nutzen. Ein Beispiel hierzu ist in Abb. 3.8 zu sehen.

Die Klasse *ExampleGame* erbt hier von der Klasse *Game*. Nun kann man eine Instanz dieser Klasse starten. Um Änderungen während des Ablaufs vorzunehmen, muss man diese nur in der überschriebenen *update()* Methode vornehmen.

In dieser eigenen *update()* Methode könnte man z.B. eine Welt in Relation zur vergangenen Zeit verändern, oder eine externe API abfragen, um die Welt anzupassen.

Wichtig ist hierbei, dass das Gittermodell, nachdem man Blöcke hinzugefügt oder entfernt hat, mit *recreateForm()* aktualisiert wird.

### Eigene Weltklasse

Wenn man nicht nur Kontrolle über den Spielverlauf haben möchte, sondern auch über die Welt selber, kann man auch von dieser erben oder von der Klasse, um das

```

1 public class ExampleGame extends Game {
2     public ExampleGame(Configuration config) {
3         super(config);
4     }
5     public final void update(final float delta) {
6         if (!getWorld().isMeshLatest()) {
7             getWorldMesh().recreateMesh();
8         }
9         // Eigene modifizierungen am Spielverlauf
10    }
11 }
```

Abb. 3.8: Ein Beispiel für die Nutzung der Bibliothek, wobei die Kontrolle über die Spielverlauf übernommen wird, indem von den jeweiligen Klassen geerbt wird. Hier wird von der Game Klasse geerbt, um die Kontrolle über den Spielverlauf zu erhalten.

Gittermodell selber zu erstellen. Ein Beispiel hierzu ist unter Abb. 3.9 zu sehen. In diesem Beispiel wird wieder das bekannte Schachbrett aus dem ersten Beispiel erzeugt.

Wenn man diese neue Weltklasse nun renderet, ist die Ausgabe wieder dieselbe wie in Abb. 3.6 dargestellt.

Durch eine eigene Klasse zum Erstellen des Gittermodells könnte man z.B. eigene Optimierungen an Diesem vornehmen.

```

1 // Klasse eigene Welt
2 public class ExampleWorld extends World {
3     public ExampleWorld(int worldSize) {
4         super(worldSize);
5     }
6     public void fillWorldWithChessboard() {
7         boolean chess = true;
8         for (int i = 0; i < worldSize; i++) {
9             for (int j = 0; j < worldSize; j++) {
10                 if (chess) {
11                     world[i][0][j] = BlockType.Stone;
12                     chess = !chess;
13                 } else {
14                     world[i][0][j] = BlockType.Dirt;
15                     chess = !chess;
16                 }
17             }
18             chess = !chess;
19         }
20     }
21 }
```

Abb. 3.9: Ein Beispiel für die Nutzung der Bibliothek, wobei die Kontrolle über den Spielverlauf übernommen wird, indem von den jeweiligen Klassen geerbt wird. Hier wird von der Weltklasse geerbt.

# 4 Kapitel 4 Realisierung

## 4.1 Game Loop

Der sogenannte Game Loop ist das Entwurfsmuster, welches von praktisch jedem Computerspiel benutzt wird, und welches außerhalb von Computerspielen bzw. 3D-Anwendungen nur sehr selten genutzt wird. Der Game Loop läuft, solange das Spiel läuft. In jedem Durchlauf wird der Input verarbeitet, der Zustand des Spiels verändert und die Ausgabe erstellt. Außerdem behält er den Überblick über die verstrichene Zeit, um den Spielverlauf anzupassen. (vgl. Nystrom2014 [6], S. 150 f.)

In diesem Abschnitt wird der Game Loop nun nach und nach aufgebaut. Beginnend mit dem einfachsten vorstellbaren Loop wird dieser dann nach und nach um die nötigen Elemente erweitert.

Ein Beispiel für einen einfachen Game Loop ist in Abb. 4.1 zu sehen. Hier erkennt man schon, dass die Schleife (bis zum Ende des Spiels) dauerhaft laufen soll. Auch müssen eventuelle Änderungen am Zustand des Spiels vorgenommen werden (hier unter `updateGameLogic()`). Außerdem wird eine Ausgabe auf dem Bildschirm (hier `render()`) erzeugt.

```
1 while (true)
2 {
3     updateGameLogic();
4     render();
5 }
```

Abb. 4.1: Ein sehr einfacher 'Game Loop'

Dieser einfache Loop (aus Abb. 4.1) erfüllt zum Teil schon die gestellten Anforderungen. Er läuft zweifelsfrei, solange das Spiel läuft. Der Input wird aktuell noch nicht verarbeitet. In jedem Durchlauf wird der Zustand des Spiels verändert und die Ausgabe erstellt.

Über die verstrichene Zeit behält dieser sehr simple Game Loop allerdings noch keine Kontrolle. Genau genommen macht er sogar das Gegenteil: die Schleife läuft, so oft sie kann und auf schnelleren Rechnern dementsprechend schneller. Wenn man nun beispielsweise Physikberechnungen machen würde (Fallgeschwindigkeit:  $y = y - 1$ ) würde ein Objekt auf einem schnelleren Rechner schneller fallen als auf einem langsamen, da `updateGameLogic()` öfter ausgeführt wird als auf einem langsamen Rechner.

Da Änderungen am Zustand des Spiels aber auch mit zeitintensiven Rechenvorgängen einher gehen können<sup>1</sup>, kann es auch sein, dass `render()` nur wenige Mal pro Sekunde aufgerufen wird. Dadurch hätte der Anwender das Gefühl, die Ausgabe würde ruckeln bzw. hängen bleiben.

Um dieses Problem zu beheben, kann man den Game Loop wie in Abb. 4.2 anpassen. Die Schleife läuft nun immer noch, so oft sie kann, aber der Zustand des Spiels bekommt nun die verstrichene Zeit (delta) seit dem letzten Aufruf mitgeteilt. Durch dieses Delta können Berechnungen nun z.B. einfach mit dem Delta multipliziert werden, wodurch sie in Relation zur Zeit und nicht mehr zur Geschwindigkeit des Rechners berechnet werden.

In der Methode `render(double delta)` kann man nun ebenfalls die verstrichene Zeit kontrollieren und das Bild z.B. nur noch 60 mal pro Sekunde erzeugen (frames per second, FPS). Dies ist in den meisten Fällen ausreichend und auf vielen Bildschirmen ohnehin die maximale Bildwiederholfrequenz, wodurch zusätzliches rendern verschenkte Rechenzeit wäre.

```

1 float last = GetTime();
2 while (true)
3 {
4     float now = getTime();
5     float delta = now - last;
6     last = now;
7     updateGameLogic(delta);
8     render(delta);
9 }
```

Abb. 4.2: Ein verbesserter 'Game Loop', welcher die verstrichene Zeit beachtet

## 4.2 Definition eines Würfels

Die wichtigste geometrische Figur für diese Arbeit ist der Würfel. Einen Würfel konstruiert man aus sechs Seiten und jede Seite wiederum aus zwei Dreiecken (4.3). Ein Dreieck entsteht aus drei „Punkten“ bzw. Vertices, die man durch Striche verbindet.

Jeder Punkt wird durch drei Koordinaten ( $x, y, z$ ) definiert, welche im Folgenden einfach Vertex genannt werden. Daraus ergibt sich, dass für einen einzelnen Würfel zwölf Dreiecke benötigt werden. Da sich die Eckpunkte der Dreiecke überlagern, würden theoretisch acht Vertices ausreichen, um einen Würfel zu beschreiben.

In einem dreidimensionalen Koordinatensystem könnten die Vertices wie in Tabelle 4.1 aussehen, wobei x hier die Horizontale, y die Vertikale und z die Tiefe darstellen würde.

---

<sup>1</sup>Es könnte beispielsweise ein Pathfinding stattfinden, welches mehrfach durch eine große Anzahl an Blöcken iterieren muss.

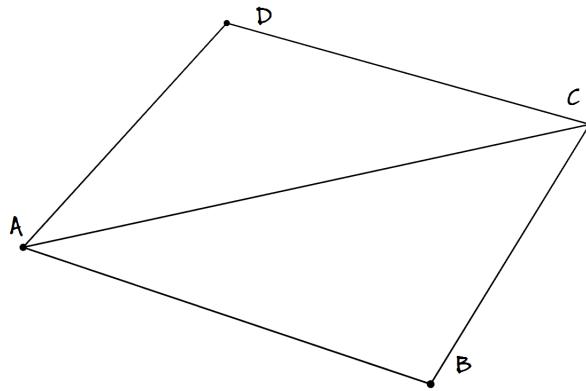
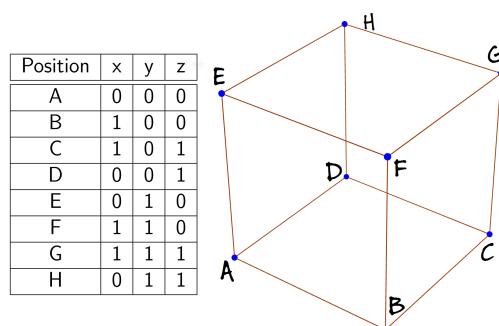


Abb. 4.3: Ein Quadrat welches aus zwei Dreiecken besteht, welche beide jeweils aus drei Vertices erstellt wurden.

Aus diesen Vertices baut man sich nun Dreiecke zusammen, so dass jede Seite des Würfels jeweils aus zwei Dreiecken entsteht. Die Punkte  $A \rightarrow B \rightarrow C \& C \rightarrow D \rightarrow A$  stellen die ersten zwei Dreiecke auf der Unterseite dar (siehe Abb. 4.3 und 4.1). Hierbei muss man einzelne Vertices mehrfach benutzen. Wenn man diesen Würfel nun ausgeben würde sähe dies wie in Abb. 4.1 aus.

Wenn man diesen Würfel mit dem im Kapitel Renderpipeline beschriebenen Verfahren darstellt, enthält der VertexBuffer den Inhalt der Tabelle in der Reihenfolge, in der man die Daten hinzufügt, hier also: 0, 0, 0, 1, 0, 0, ..., 0, 1, 1. Im IndexBuffer werden dann die Dreiecke „zusammengebaut“, indem die Reihenfolge angegeben wird, in der diese aus den Vertices erstellt werden: aus  $A \rightarrow B \rightarrow C$  würde also 0, 1, 2 und aus  $C \rightarrow D \rightarrow A$  würde 2, 3, 0 im Buffer.

Dieser sehr einfache Würfel bildet im weiteren Verlauf die Grundlage für die Umsetzung mit Entities. Im nicht optimierten Gittermodell, aus den späteren Kapiteln, wird ebenfalls der gleiche Würfel (bis auf verschobene Koordinaten) verwendet. Für jeden enthaltenen Würfel werden diese *vertices* und *indices* erstellt und dem Vertex- bzw. IndexBuffer des Gittermodells hinzugefügt.



Tab. 4.1: Acht Punkte aus denen ein Würfel mit Seitenlänge '1' erstellbar wird.

## 4.3 Umsetzung mit Entities

Um eine Welt, bestehend aus vielen tausend Würfeln anzuzeigen, muss der im vorherigen Kapitel erstellte Würfel nun also vervielfacht werden. Da es sich hier meistens um viele tausende (oder sogar millionen) Würfel handelt, sollte dies soweit abstrahiert werden, dass nur noch die Position und der Typ des Würfels angegeben werden müssen, um diesen zu erzeugen.

Für diese Abstraktion gibt es mehrere Verfahren, wobei nicht alle geeignet sind, mit der geforderten Anzahl an Würfeln zu arbeiten bzw. diese darzustellen. Im folgenden wird ein Ansatz erklärt, welcher zwar sehr elegant ist, aber schlussendlich nicht zum Ziel geführt hat.

Um nun also nicht mehr mit einzelnen Dreiecken arbeiten zu müssen, wurde einen Würfel als Modell gespeichert. Dieses Modell (im weiteren RawModel genannt) enthält nur noch ein „Vertex Array Object“ (siehe hierzu Kapitel Grafikpipeline), in dem Vertices, Indices, Normals und Texturkoordinaten für einen Würfel mit Kantenlänge 1 gespeichert sind.

Dieses Modell wird nun inklusive einer Textur (z.B. Stein) als sogenanntes Textured-Model gespeichert (siehe Abb. 4.6). Ein Entity bekommt nun ein TexturedModel zugewiesen, welches die Textur und ein RawModel enthält. Dies hat den Vorteil, dass das RawModel für den Würfel wieder benutzt werden kann.

Das so erstellte Entity enthält nun außer dem RawModel noch einen „Punkt“ (dreidimensionale Koordinaten), drei Variablen für die Rotation um die X-, Y- bzw. Z-Achse und schlussendlich noch eine Variable für die Skalierung. Dies macht es sehr einfach, unterschiedliche Würfel zu erzeugen, indem neue Entities, an unterschiedlichen Stellen, mit der passenden Textur erzeugt werden.

Der große Vorteil hierbei ist, dass die primitiven geometrischen Punkte und Dreiecke (das RawModel) nur einmalig an die Grafikkarte übertragen werden müssen. Dies benötigt also während des Darstellens keine Rechenzeit mehr, und ein einzelner Würfel benötigt auch fast keinen Grafikspeicher mehr. Durch die hohe Abstraktion, muss man nicht mehr mit einzelne Buffern aus Variablen arbeiten, sondern kann ein Entity Objekt benutzen. Mit diesem Entity Objekt ist es sehr einfacher zu arbeiten.

Man könnte statt einem Würfel auch ein beliebiges anderes Objekt (z.B. Gegner, Waffen oder andere Objekte, siehe Bild im Anhang: Abb. 7.1 Stanford Drache) laden und es ebenfalls genauso einfach verwalten wie einen Würfel.

In Abb. 4.4 wird exemplarisch ein Würfel mit einer Steintextur erstellt. Die Daten für den Würfel (vertices, indices, texture coordinates und normals) werden aus der Datei cube.obj geladen. Danach erstellt der „Loader“ ein RawModel welches die ID eines VertexArray enthält. Dieser VertexArray wiederum enthält Verweise auf die die jeweiligen Buffer, welche der Loader ebenfalls erstellt und mit den geladenen Daten befüllt.

Außerdem lädt der Loader eine Textur für den Stein in den Grafikspeicher und gibt eine „ModelTexture“ mit der von OpenGL generierten ID der Textur zurück.

Wenn nun mit `new Entity()` ein neues Entity erzeugt wird, übergibt man diesem nur noch das TexturedModel, einen Vector für die Position (im Beispiel `0,0,0`) und eine eventuell Rotation (hier keine) bzw. Skalierung (hier `1.0`). Der daraus resultierende Würfel ist in Abb. 4.5 zu sehen.

Wenn nun ein weiterer Würfel angezeigt werden soll, kann man sich ein neues Entity mit `new Entity(texturedModel, new Vector3f(1,0,0), rotX, rotY, rotZ, scale)` erstellen, wobei hierzu die Position, Rotationen und Skalierung angegeben werden muss. Dieser Würfel benutzt das gleiche RawModel und die gleiche Textur, hat aber eine andere Position.

```
1 // Load the cube from .obj file
2 ModelData m = OBJLoader.loadOBJ("cube");
3 // Load the array onto opengl
4 RawModel model = loader.loadToVAO(
5     m.getVertices(), m.getTextureCoords(),
6     m.getNormals(), m.getIndices());
7 // Load texture
8 ModelTexture stone = new ModelTexture(loader.loadTexture("stone"));
9 TexturedModel texturedModel = new TexturedModel(model, stone);
10 // Create entity with model & texture
11 Entity e = new Entity(texturedModel, new Vector3f(0,0,0), 0,0,0,1.0f);
```

Abb. 4.4: Quellcode um ein Entity zu erstellen.



Abb. 4.5: Das in Abb. 4.4 erstellte Entity wenn man es mit dem EntityRenderer ausgibt.

Es gibt aber auch einen gewaltigen Nachteil: bevor ein Entity gerendert werden kann, muss eine passende Transformationsmatrix auf die Grafikkarte übertragen werden, damit der Shader das Objekt dementsprechend „umrechnen“ kann, da es sich ja um die gleichen Vertices handelt, welche den Würfel beschreiben.

Im obigen Beispiel wird im Grunde zweimal nacheinander der gleiche Würfel gezeichnet. Vor dem Zeichnen wird aber die Transformationsmatrix geändert, so dass der zweite Würfel neben dem ersten gezeichnet wird.

Der in diesem Kapitel vorgestellte Ansatz zum Darstellen von Würfeln ist zwar sehr elegant und für wenige Objekte (< 10.000) in Tests noch unproblematisch gewesen, wird aber bei der benötigten Anzahl an Würfeln für eine komplette Welt unweigerlich zum Flaschenhals<sup>2</sup>.

Um später andere Objekte anzuzeigen, welche nicht dem im folgenden Kapitel vorgestellten Gittermodell hinzugefügt werden sollen, eignen sich die Entities aber trotzdem hervoragend. Wenn z.B. ein Baum in der Welt angezeigt werden soll, kann man hierfür einfach ein komplexes Modell laden und aus diesem dann ein Entity erstellen.

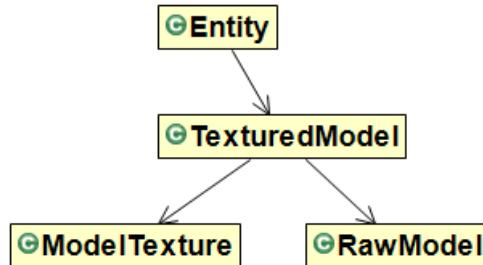


Abb. 4.6: Die Entity Klasse vereint ein Modell und eine Textur.

## 4.4 Aufbau eines Gittermodells

Um die im vorherigen Kapitel erklärten Probleme mit einzelnen Entities zu lösen, wird ein komplett anderer Ansatz für die Darstellung von Würfeln benötigt. Während der Entity-Ansatz es erlaubt, wenige geometrische Objekte an die Grafikkarte zu übertragen, aber dafür bei der Darstellung jedes Mal Daten übertragen werden müssen, sollen nun möglichst wenig Daten (im besten Fall keine) während der Darstellung übertragen werden.

Als Lösung werden nun nicht mehr einzelne Objekte (Würfel) an die Grafikkarte übertragen, sondern es wird ein größeres Objekt aus Würfeln bzw. aus Dreiecken für diese Würfelflächen erstellt. Es entsteht ein Gittermodell, welches mehrere Würfel zusammenfasst und dadurch mit einem Befehl an die Grafikkarte übertragen wird.

Dies funktioniert logischerweise nur, solange keine Würfel hinzugefügt oder entfernt werden. Wenn Elemente aus dem Gittermodell entfernt oder hinzugefügt werden sollen, muss das Gittermodell neu erstellt und in den Grafikspeicher übertragen werden.

Der Vorteil ist nun, dass dieses Gittermodell mit einem einzigen „draw call“ gezeichnet wird, statt wie bei den Entities für jeden Würfel extra die Transformationsmatrix zu ändern und ihn dann zu zeichnen.

---

<sup>2</sup>Auf meinem Rechner sind bis zu 80.000 Würfel möglich, bevor es zu starken Performanceeinbrüchen kommt.

Um aus einem Chunk ein Gittermodell zu erzeugen, muss einmal durch das komplette Feld mit Würfel-Enums iteriert werden und überall dort, wo ein Würfel entstehen soll, die entsprechenden Punkte dem VerticesBuffer hinzugefügt werden. Die so erstellten Vertices müssen dann im IndicesBuffer zu Flächen verbunden werden, die den entsprechenden Würfel erzeugen.

## 4.5 Optimierung des Gittermodells

Das Gittermodell aus dem vorherigen Kapitel lässt sich zwar schnell darstellen, ist aber immer noch viel zu umfangreich. Wenn man mit diesem sehr naiven Ansatz ein Gittermodell für einen Chunk ( $16 * 16 * 16$ ) erstellt, welcher komplett mit Würfeln ausgefüllt ist, werden dem Gittermodell 4.096 einzelne Würfel hinzugefügt. In Tabelle 4.2 sieht man, dass hierfür insgesamt 442.368 einzelne Variablen benötigt werden.

Diese Anzahl ist nicht nur für den Grafikspeicher in einer größeren Welt problematisch, sondern es werden auch sehr viele für die Kamera nicht sichtbare Würfel dauerhaft gezeichnet bzw. durch die komplette Grafikpipeline geschickt, was ebenfalls sehr uneffektiv ist.

Wenn man für eine Gleitkommazahl 4 Bytes im Speicher benötigt und für einen Integer ebenfalls von 4 Bytes ausgeht, braucht ein quadratischer Chunk der Größe 128 bereits knapp ein Gigabyte an Speicher. Um mehr als 100 Blöcke weit zu sehen, benötigt man aber bereits fünf dieser Chunks<sup>3</sup>. Fünf Gigabyte Grafikspeicher sind selten möglich und für die Bibliothek als Anforderung absolut nicht akzeptabel. Um dieses Problem zu lösen, müssen also unausweichlich die Anzahl der Würfel (bzw. der Dreiecke) im Gittermodell reduziert werden.

Chunk	Vertices	Floats	Indices	Integer	Speicher
$1^3$	24	72	12	36	0,432 kB
$16^3$	98.304	294.912	49.152	147.456	~1,7 MB
$32^3$	786.432	2.359.296	393.216	1.179.648	~14,2 MB
$64^3$	6.291.456	18.874.368	3.145.728	9.437.184	~114 MB
$128^3$	50.331.648	150.994.944	25.165.824	75.497.472	~906 MB

Tab. 4.2: Speicherverbrauch eines nicht optimierten Gittermodells.

Um die Anzahl der Dreiecke im Gittermodell zu verringern, wurden folgende Möglichkeiten gefunden, welche nun näher erläutert werden:

- Würfel aus dem Gittermodell entfernen, welche nicht dargestellt werden,
- Würfel nicht mehr komplett anzeigen, sondern nur noch die Seiten, die sichtbar sind,

---

<sup>3</sup>siehe 3.1 Verwaltung der Welt:  $(n + n + 1)^3$ .

- keine einzelnen Würfel mehr im Gittermodell, sondern größere Flächen.

## **Würfel aus dem Gittermodell entfernen**

Bisher befinden sich im Gittermodell alle im Chunk vorhandenen Würfel. Da ein Chunk im „worst-case“ (voll mit Blöcken) dadurch eben diese  $n^3$  Würfel enthält, fallen die in Tabelle 4.2 dargestellten Vertices an, bzw. der Speicher wird benötigt. Da der Spieler aber nur einen Bruchteil dieser Würfel tatsächlich „sieht“, kann man hier offensichtlich viel optimieren.

Um Würfel aus dem Gittermodell zu entfernen, kann man folgende Möglichkeiten nutzen:

- Würfel, welche auf allen sechs Flächen von anderen Würfeln umgeben sind, nicht hinzufügen.
- Würfel, welche sich „hinter“ dem Spieler bzw. der Kamera befinden, nicht hinzufügen (nicht im Beispielprojekt umgesetzt).
- Würfel, welche sich hinter anderen Würfeln befinden: z.B. Berge oder Höhlen nicht dem Gittermodell hinzufügen (nicht im Beispielprojekt umgesetzt, vermutlich nicht lohnend umzusetzen, siehe unten).

Von diesen drei Ansätzen wurde nur für den Ersten eine sinnvolle Lösung gefunden. Im Folgenden wird diese erklärt, und warum die beiden anderen Ansätze nicht möglich waren.

Wenn man Würfel dem Gittermodell hinzufügt, ist es relativ einfach, vorher zu prüfen ob dieser Würfel auf jeder Seite von einem anderen Würfel umgeben ist. Wenn dies der Fall ist, muss der Würfel nicht mehr dem Gittermodell hinzugefügt werden, da er logischerweise auch für den Spieler nicht sichtbar sein kann.

Sobald ein Würfel an einer der jeweiligen sechs Seiten allerdings an Luft grenzt, bedeutet dies nicht, das der Spieler diesen Würfel auch tatsächlich sehen kann. Dem Gittermodell werden also nur noch „Oberflächen-Würfel“ hinzugefügt.

Ein Beispiel hierfür sind z.B. verschlossene Gewölbe unter der Erde, welche für den Spieler nicht sichtbar sind. Die Würfel welche die Höhle begrenzen, sind zwar teilweise von Luft umgeben, aber trotzdem nicht sichtbar für den Spieler.

Obwohl diese Optimierung sehr gering wirkt, lässt sich der Umfang des Gittermodells hierdurch in viele Fällen schon drastisch reduzieren. Im „best-case“, wenn ein Chunk komplett aus Würfeln besteht, müssen dem Gittermodell überhaupt keine, oder nur die Würfel am Rand des Chunks hinzugefügt werden.

Im „worst-case“ allerdings, wenn ein Chunk immer abwechselnd aus einem Würfel und dann wieder aus Luft besteht, ändert diese Methode nichts am Gittermodell und ist damit überflüssig (siehe Abb. 4.7).

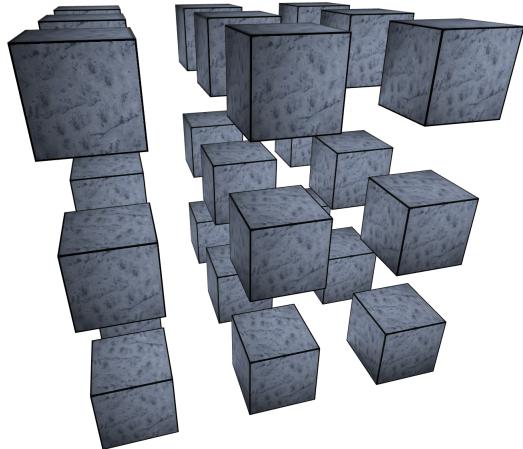


Abb. 4.7: Ein Chunk, immer abwechselnd aus einem Würfel und dann wieder aus Luft, kann nicht mehr optimiert werden.

Die zweite Möglichkeit, Würfel „hinter“ dem Spieler zu entfernen, eignet sich nicht für einen einzelnen Chunk. Wenn man dies umsetzen würde, müsste man das Gittermodell bei jeder Kameradrehung neu berechnen und würde den durch das Gittermodell so aufwändig erarbeiteten Geschwindigkeitsvorteil wieder verlieren.

Wenn man allerdings mehrere Chunks nutzt, kann man ein ähnliches Verfahren nutzen und nur jene Chunks tatsächlich rendern, welche im Sichtbereich des Spielers liegen (sogenanntes *frustum culling*). Berechnet und in den Grafikspeicher geladen werden, müssen diese Gittermodelle aber trotzdem, um auf Kameradrehungen schnell reagieren zu können. Hier spart man sich also keinen Grafikspeicher, sondern nur während des renders kostbare Zeit.

Für die letzte Möglichkeit, Würfel hinter verdeckten Würfeln zu entfernen, gilt ein ähnliches Problem wie für die zweite Möglichkeit. Man könnte diese zwar durchaus errechnen, aber müsste den Chunk dann bei Bewegungen ebenfalls neu berechnen, was auch hier nicht lohnend wäre.

### **Würfel nicht mehr komplett anzeigen**

Wenn für einen Würfel jeweils 24 Vertices bzw. zwölf Dreiecke erstellt und gezeichnet werden müssen, ist dies in vielen Fällen überflüssig. Da man nie alle Seiten eines Würfels gleichzeitig sehen kann, könnte man theoretisch jeden Würfel im Gittermodell um einzelne Seiten kürzen.

Während der Darstellung übernimmt dies zum Teil OpenGL selber, im sogenannten *Backface Culling* (siehe Anhang: Abb. 7.3 und 7.4). Dabei entscheidet OpenGL selbstständig, über die Normalvektoren, ob ein Vertex sichtbar ist und zeichnet diesen nur dann. Auf den Speicher hat dies allerdings noch keine Auswirkungen, da trotzdem alle Vertices übertragen werden müssen.

Man kann aber im Gittermodell bereits einzelne Seiten auslassen. Hier kann man analog zum Verfahren des Würfel-entfernens vorgehen: wenn eine Seite von einem anderen Würfel verdeckt ist, muss diese dem Gittermodell nicht mehr hinzugefügt werden. Auch für dieses Verfahren gelten ähnliche Vor- und Nachteile: unter Umständen spart man sich enorm viele Würfel ein und im schlechtesten Fall überhaupt keine.

In Abb. 4.8 ist der Quellcode aufgeführt, um einzelne Seiten bzw. die Daten für diese den jeweiligen Buffern hinzuzufügen. Aufgerufen wird die Methode addCube(), welcher ein Array aus Booleans übergeben wird. Diese Booleans geben an, welche Seiten dem Buffer hinzugefügt werden sollen (hier nur für die Hinterseite des Würfels aufgeführt).

Die Methode addBackSide() fügt nun für die Rückseite die jeweiligen Vertices, Indices, Normalvektoren und Texturkoordinaten den jeweiligen Buffern hinzu. Dies wird für alle benötigten Seiten gemacht, so dass im Buffer am Ende alle Seiten enthalten sind, die sichtbar sind.

```

1 public int addCube( float tx , float ty , float tz , boolean[] sides ){
2     if (sides[0]){
3         addBackSide(tx , ty , tz , type);
4     }
5     // Hier fehlen zur Übersichtlichkeit die anderen Seiten
6 }
7 private int addBackSide( float tx , float ty , float tz ){
8     vertexPositionData.put(new float []{
9         tx , 1+ty , tz ,
10        tx , ty , tz ,
11        1+tx , ty , tz ,
12        1+tx , 1 + ty , tz
13    });
14     indicesData.put(new int []{
15         1 , 0 , 3 ,
16         3 , 2 , 1
17    });
18     normalsData.put(new float []{
19         0 , 1 , 0 ,
20         0 , 0 , 0 ,
21         1 , 0 , 0 ,
22         1 , 1 , 0 ,
23    });
24     textureCoords.put(new float []{
25         0 , 1 ,
26         1 , 0 ,
27         1 , 1 ,
28         0 , 1
29    });
30 }
```

Abb. 4.8: Methode, um einen Würfel dem Gittermodell hinzuzufügen. Hier wird exemplarisch nur die Methode zum Hinzufügen der Rückseite eines Würfels aufgeführt, die anderen Seiten sehen aber dementsprechend gleich aus.

### Keine einzelnen Würfel mehr anzeigen

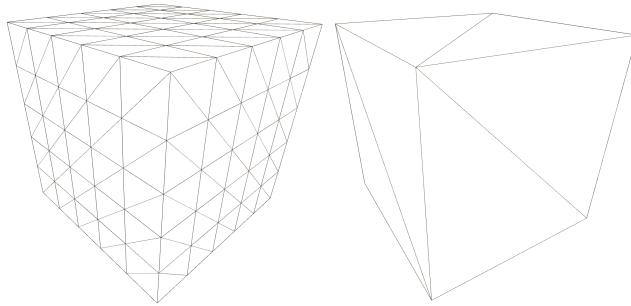


Abb. 4.9: Links ein Gittermodell mit den Optimierungen der Bibliothek und rechts ein optimales Gittermodell.

In einer Würfelwelt kommen oft größere rechteckige Vorkommen desselben Blocks vor, z.B. ein großer Würfel, welcher aus vielen kleinen Würfel besteht. Hier werden dem Gittermodell für jeden dieser Würfel, wenn er sichtbar ist, eigene Vertices hinzugefügt. Stattdessen könnte man aber auch mehrere Würfel zusammenfassen und diese mit einer sich wiederholenden Textur belegen.

Für den Nutzer wirkt dies, durch die sich wiederholende Textur, als gäbe es viele Würfel, aber im Gittermodell müssen nur einzelne Rechtecke erstellt werden, welche mehrere Würfel repräsentieren. Wenn einzelne Würfel entfernt oder hinzugefügt werden, müssen die „großen Rechtecke“ dementsprechend neu berechnet und angepasst werden.

Hierdurch kann man die Anzahl der Vertices, wenn es sich um gleiche Blöcke handelt, noch einmal stark reduzieren, da dieses Verfahren immer ein optimaleres Gittermodell liefert, als das von mir Umgesetzte (vgl. Ofps2012 [7]).

In der aktuellen Version der Bibliothek ist dieses Verfahren noch nicht implementiert aber in Abb. 4.9 skizziert. Auf der linken Seite ist ein  $5 * 5 * 5$  Chunk zu sehen, wie er in der Bibliothek optimiert wird und auf der rechten Seite, wie der selbe Chunk mit dieser Optimierung aussehen würde.

## 4.6 Shader

Im Gegensatz zur „fixed function pipeline“ unterstützt OpenGL auch die sogenannte „programmable pipeline“. Mit dieser programmierbaren Pipeline können nicht mehr nur die Parameter auf unterschiedlichen Ebenen der Grafikkartenverarbeitung gesteuert, sondern eigene Programme geschrieben werden, welche direkt auf der Grafikkarte ausgeführt werden.

Diese kleinen Programme werden Shader genannt und (unter OpenGL) in der OpenGL Shading Language (GLSL) geschrieben. GLSL orientiert sich dabei stark an der C-Syntax, wodurch die Programme sehr ähnlich aussehen (vgl. McShaffry2012 [3], S. 499).

OpenGL hängt seit der Version 3.1 sehr stark von Shadern ab. Die einzige Anzeige, welche ohne Shader möglich ist, ist das Löschen eines Fensterinhaltes (glClear). Bis inklusive Version 3 von OpenGL war es möglich, die „fixed-function pipeline“ zu nutzen, welche Geometrie- und Pixeldaten selbstständig verarbeitet hat, ohne eigene Shader zu nutzen.

Seit Version 2.0 werden Shader offiziell von OpenGL unterstützt (vgl. Shreiner2013 [2], S. 34).

Um einen Shader zu nutzen, muss dieser ähnlich wie ein C-Programm von einem Compiler analysiert, auf Fehler überprüft, in „object code“ übersetzt und schlussendlich kompiliert werden. In OpenGL stehen diese Operationen durch die API zur Verfügung (vgl. Shreiner2013 [2], S. 70).

In meinem Beispiel findet dies in der Klasse „ShaderProgram“ statt (siehe Abb. 4.10). Hierbei wird der Methode `createShader(string)` ein String mit dem Inhalt des Shader übergeben. Dann wird von OpenGL mit `glCreateShader(type)` eine ID angefordert und mithilfe dieser ID der Code des Shaders an OpenGL geschickt.

OpenGL kompiliert nun den Shader. Um ihn zu verwenden, wird er einem „program“ (`glCreateProgram()`) hinzugefügt, welchem auch weitere Shader hinzugefügt werden. OpenGL kann hier nun auch validieren, ob der Vertexshader zum Fragmentshader kompatibel ist. Wenn man die geladenen Shader während der Darstellung nutzen möchte, benötigt man nur noch die „program id“ (`glUseProgram(programID)`).

```
1 int loadShader(String shaderSource, int type){  
2     int id = glCreateShader(type);  
3     glShaderSource(id, shaderSource);  
4     glCompileShader(id);  
5     return id;  
6 }  
7 createShader(String shaderString){  
8     shaderId = loadShader(shaderString, GL_VERTEX_SHADER);  
9     programID = glCreateProgram();  
10    glAttachShader(programID, shaderId);  
11    bindAttributes();  
12    glLinkProgram(programID);  
13    glValidateProgram(programID);  
14 }
```

Abb. 4.10: Quellcode: Laden des Shaders.

# 5 Kapitel 5 Performance

Um zu überprüfen, wie effektiv die einzelnen Verfahren zum Rendern der Welt sind, und ob die vorgenommenen Optimierungen sich gelohnt haben, wurde während der Entwicklung immer wieder getestet, wie viel Leistung die Bibliothek benötigt.

Da die ersten Verfahren mit einzelnen Entities nicht die gestellten Anforderungen erfüllt haben, sind hier direkte Vergleiche nur schwer möglich. Zwischen den einzelnen Gittermodellen kann man aber sehr direkt vergleichen.

Das Entwicklungssystem war ein sehr schnelles System, so dass die Anforderungen der Bibliothek am besten immer übertroffen werden sollten, da viele Anwender höchstwahrscheinlich mit langsameren Systemen arbeiten. Da selbst das Erfassen der einzelnen Messdaten die Ergebnisse verfälschen kann, sind drastische Unterschiede gewünscht, um die Effektivität sicher beurteilen zu können.

Entwicklungssystem	
CPU	3,40 GHz
RAM	16 GB
Grafikspeicher	3 GB
GPU	950 MHz
Speichertyp	GDDR 5
Speichertaktfrequenz	1250 MHz
OpenGL Version	6.14

Tab. 5.1: Konfiguration des Systems, auf dem entwickelt wurde.

## Rendern einzelner Entities

Wenn jeder Würfel als Entity und damit tatsächlich als Objekt erstellt wird, ist ein Geschwindigkeitsvergleich sinnlos. Selbst auf dem sehr starken Entwicklungssystem ist die Renderzeit schon bei 27.000 Blöcken auf mehr als 0.02 Sekunde pro Frame gestiegen, wodurch es zu FPS einbrüchen kam.

Eine Messung mit *Java Monitor* hat ergeben, dass der Zeitbedarf in Relation zu den jeweiligen Transformationsmatrizen liegt, welche übertragen werden müssen.

## Rendern eines Gittermodells

Ein nicht optimiertes Gittermodell muss jeden Würfel, welcher in der Welt existiert, hinzugefügen. Dadurch steigt die Anzahl an Vertices enorm schnell. Das „worst-case“ ist in diesem Fall eine Welt, welche ausschließlich aus Blöcken besteht, da dann die meisten Vertices benötigt werden. Hierbei kam es ab ca. 729.000 Würfeln zu den ersten Performanceeinbrüche. Dies entspricht 17.496.000 Vertices.

## Rendern eines optimierten Gittermodells

Theoretisch ist das „best-case“ eines Gittermodells eine Welt, in der überhaupt keine Würfel sind. Da dies aber sinnfrei zur Messung der Performance wäre, wurde ein sinnvolles „best-case“ gewählt. Eine Welt welche komplett aus Blöcken besteht, ist ähnlich häufig wie eine Welt ohne Blöcke, aber Messungen lassen sich durchführen.

In diesem sinnvollen „best-case“ waren durch die Optimierungen Welten nicht mehr durch die Anzahl der Vertices beschränkt, sondern durch den verfügbaren Speicher. Eine Welt aus 729.000.000 Würfeln benötigt dabei nur 19.440.000 Vertices, aber bereits ca. 3 GB Speicher um sie im Arbeitsspeicher zu halten

Aber selbst im „worst-case“ war mit diesem Gittermodell noch eine Welt mit 5.832.000 möglichen Blöcken, in der 729.000 Würfel dargestellt werden müssen, ohne größere FPS-Einbrüche darstellbar. Da im „worst-case“ von allen Würfeln auch jede Seite dargestellt werden muss, handelt es sich hierbei um  $729.000 * 6 = 4.374.000$  Seiten. Dies entspricht 17.496.000 Vertices.

Im Gegensatz zum „worst-case“ des nicht optimierten Gittermodells müssen hier aber explizit immer abwechselnd Blöcke und keine Blöcke sein, um diese Anzahl an Würfeln im Gittermodell überhaupt zu erreichen.

Um eine realistischere Messung durchführen zu können, als mit dem jeweiligen „best-“ bzw. „worst-case“, wurde ein Generator für eine Welt aus octavenförmigen Gebilden genutzt. Hier waren die Ergebnisse sehr positiv, wie man in Tabelle 5.2 sehen kann. Hierbei ist zu beachten, das sich im Array mehr Würfel befinden, als im Gittermodell, da eben nicht mehr alle dargestellt werden müssen.

Als Vergleichswert die Werte für ein nicht optimiertes Gittermodell der gleichen Welt in Tabelle 5.3. Hierbei ist zu beachten, dass die Darstellung mit diesem Gittermodell bereits nicht mehr flüssig war.

	Wert
Weltgröße	8.000.000
Würfel im Array	1.988.894
Würfel im Gittermodell	224.508
Würfelseiten im Buffer	351.808
Vertices im Buffer	1.407.232
Zeit für Bildberechnung	$< 0,015$ Sekunden
Speicherbedarf (RAM)	93 MB

Tab. 5.2: Messergebnisse einer realistischeren Welt, in der Optimierungen am Gittermodell vorgenommen wurden.

	Wert
Weltgröße	8.000.000
Würfel im Array	1.988.894
Würfel im Gittermodell	2.028.894
Würfelseiten im Buffer	12.173.364
Vertices im Buffer	48.693.456
Zeit für Bildberechnung	ca. 0,08 Sekunden
Speicherbedarf (RAM)	567 MB

Tab. 5.3: Messergebnisse einer realistischeren Welt, **ohne** das Optimierungen im Gittermodell vorgenommen wurden.

# 6 Kapitel 6

# Fazit

## 6.1 Zusammenfassung

Das Ziel, eine technische Grundlage zur Darstellung von Blöcken zu schaffen, konnte trotz einiger Schwierigkeiten erreicht werden. Die anfängliche Vorgabe, eine halbe Millionen Würfel im Speicher zu halten, und das Gittermodell ausreichend zu optimieren, um dieses auch tatsächlich darstellen zu können, wurde nicht nur erreicht, sondern sogar um ein Vielfaches übertroffen.

Eine große Herausforderung hat zu Anfang die Grafikpipeline von OpenGL dargestellt. Da im Studium weder die Mathematik hinter 3D-Berechnungen, noch die Funktionsweise von OpenGL Gegenstand einer eigenen Vorlesung waren, musste hier zu Beginn sehr viel Zeit investiert werden. Dieser initiale Aufwand hat sich aber spätestens dann gelohnt, als der Ansatz, die Welt aus eigenen Entities zu rendern, gescheitert ist und die Grafikpipeline manuell durchlaufen werden musste.

Durch das darauf folgende Umschreiben der Grafikpipeline konnte nicht alles, was gewünscht war, umgesetzt werden. Für die Lichtberechnungen sind zwar die Grundlagen gelegt, aber fertig implementiert wurden sie nicht. Auch beim Einbinden der Texturen wurde noch keine optimale Technik verwendet, sondern es wird eine externe Bibliothek benutzt, welche nur PNG-Dateien laden kann.

Um Chunks (mehr als einen) zu implementieren, hat die Zeit gegen Ende leider nicht mehr ausgereicht. Dies ist besonders ärgerlich, da hier schon während der Grundlagen sehr viel Zeit aufgewendet wurde. Da die Welt aber komplett vom Renderer getrennt ist, kann man hier, mit genug Zeit für Implementierung und Testen, ohne größere Änderungen an der API die Funktionalität ergänzen. Hier wäre eine bessere Zeitplanung und Priorisierung der einzelnen Features hilfreich gewesen.

Es hat sich, wenn auch wenig überraschend, herausgestellt, dass die verwendete Technologie die gestellten Anforderungen ohne Probleme erfüllen kann. Da es sich aber aktuell noch um eine sehr rudimentäre Funktionalität handelt, wäre es sehr spannend die Bibliothek noch einmal in C oder C++ zu programmieren und zu überprüfen, ob und wie groß der Geschwindigkeitsvorteil hier wäre.

## 6.2 Ausblick

Aktuell lassen sich mit der Bibliothek nur Blöcke anzeigen und man kann die Kamera bewegen, um dies zu überprüfen. Dies ist sicherlich eine Grundlage, aber eben auch noch nicht sehr „hübsch“ oder „spannend“. In, auf diese Arbeit aufbauenden, anderen Arbeiten, wären praktisch alle Features einer vollwertigen Spieleengine denkbar. Exemplarisch werden im Folgenden ein paar mögliche Erweiterungen vorgestellt:

- Weitere Optimierungen im Gittermodell, wie sie in dieser Arbeit vorgestellt wurden.
- Mehr als einen Chunk erstellen, diese verbinden und den WorldRenderer zu einem ChunkRenderer umbauen, damit Welten nahezu unendlich groß sein können.
- Multithreading für Änderungen im Gittermodell. Dadurch könnten Änderungen unabhängig vom Game Loop vorgenommen werden, wodurch es vor allem bei größeren Chunks zu weniger „Rucklern“ kommt.
- Kollisionsprüfung zwischen Kamera und Spieler, damit man nicht mehr durch die Welt fliegt, sondern tatsächlich laufen kann.
- Eine Skybox, welche den Himmel darstellt.
- Einen FontRenderer, um Text oder ein Interface anzuzeigen.
- Entfernen & Hinzufügen einzelner Blöcke aus dem Gittermodell per Maus (Abbau und Aufbauen von Blöcken).
- Weltgenerierung einzelner Biome nach dem Vorbild von Minecraft.
- Umschreiben der Bibliothek auf JOGL, oder neu Implementieren in C, um Performance Unterschiede zu messen.

# **7** Kapitel 7

# Anhang

## **Inhalt der beigefügten CD**

Der Bachelorarbeit ist eine CD-Rom beigefügt, auf welcher sich alle, während der Bachelorarbeit erstellten Dateien befinden. Im Einzelnen handelt es sich um folgenden Inhalte:

- Die erstellte Bibliothek als Quellcode für den Import nach Eclipse.
- Eine einzeln lauffähige JAR-File der Bibliothek, welche nur noch Java zum Ausführen benötigt und in andere Projekte eingebunden werden kann.
- Die Dokumentation der API der Bibliothek im HTML-Format.
- Die Ausarbeitung der Bachelorarbeit als PDF (dieses Dokument).

## Bilder



Abb. 7.1: Der Stanford-Drache, ein Modell der Stanford-Universität welcher aus 1,132,830 Dreiecken besteht. Hier gerendert mit dem EntityRenderer der Bibliothek und mit einer Textur aus Steinen, Wiese und Erde belegt.

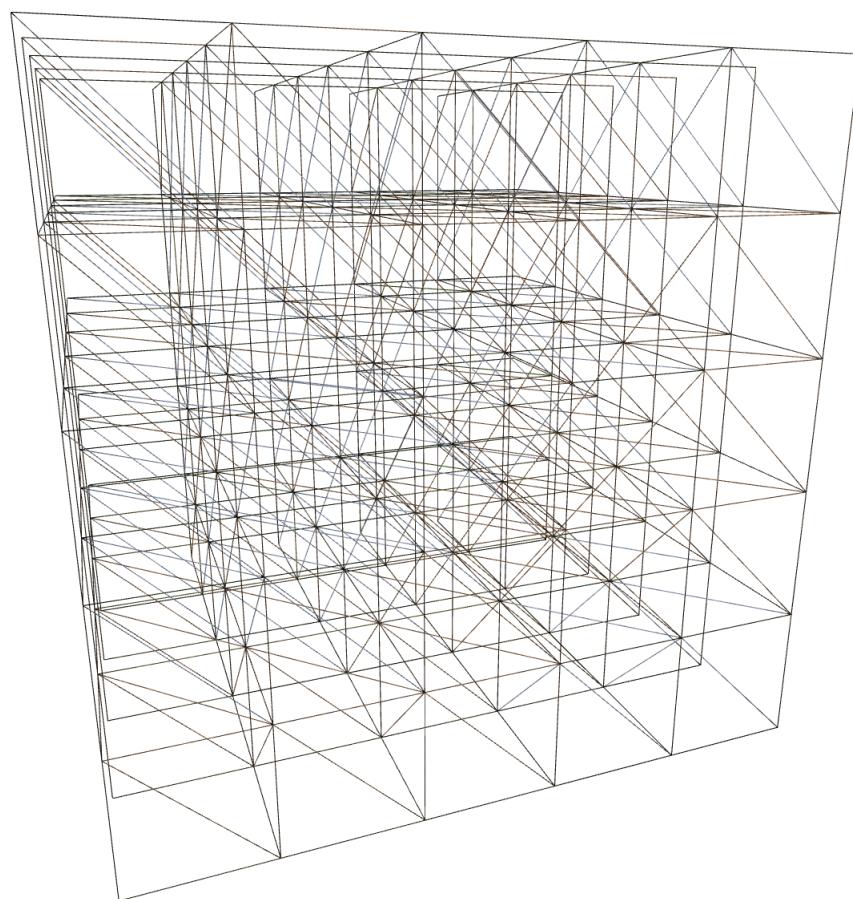


Abb. 7.2: Ein  $5 \times 5 \times 5$  Chunk in dem das Gittermodell nicht optimiert wurde.

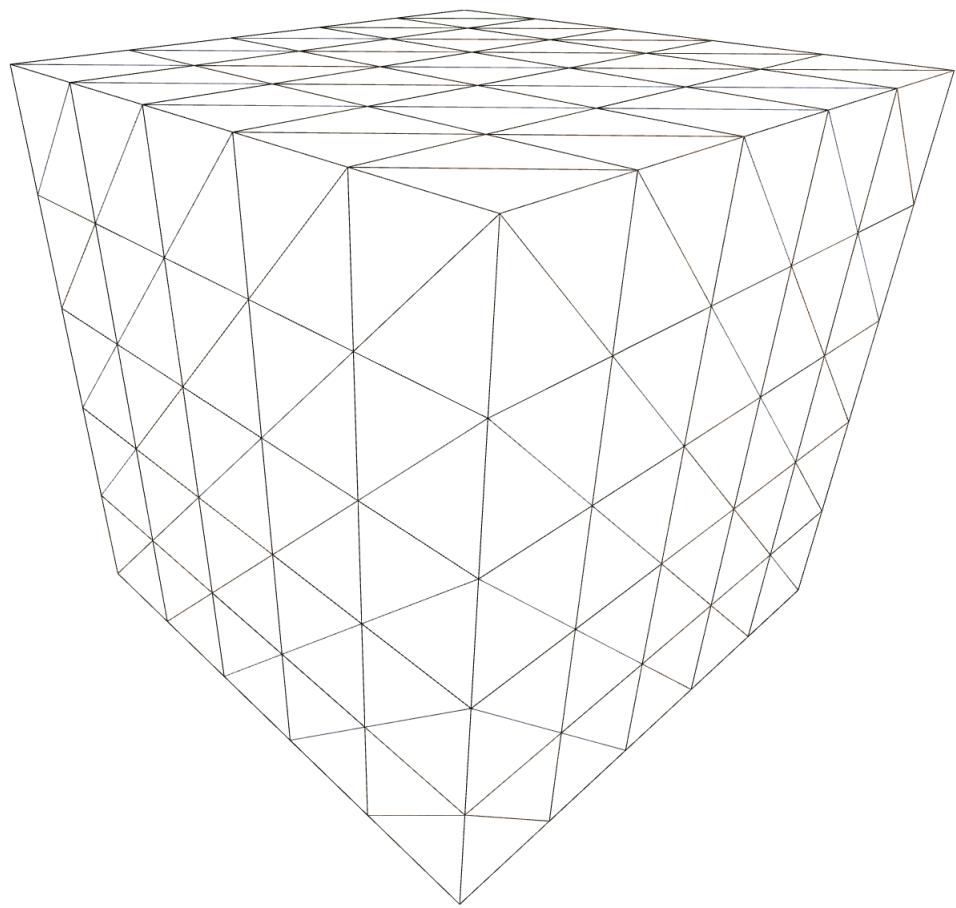


Abb. 7.3: Ein  $5 \times 5 \times 5$  Chunk in dem das Gittermodell optimiert wurde. Alle Würfel werden nur noch mit ihren Außenseiten gerendert.

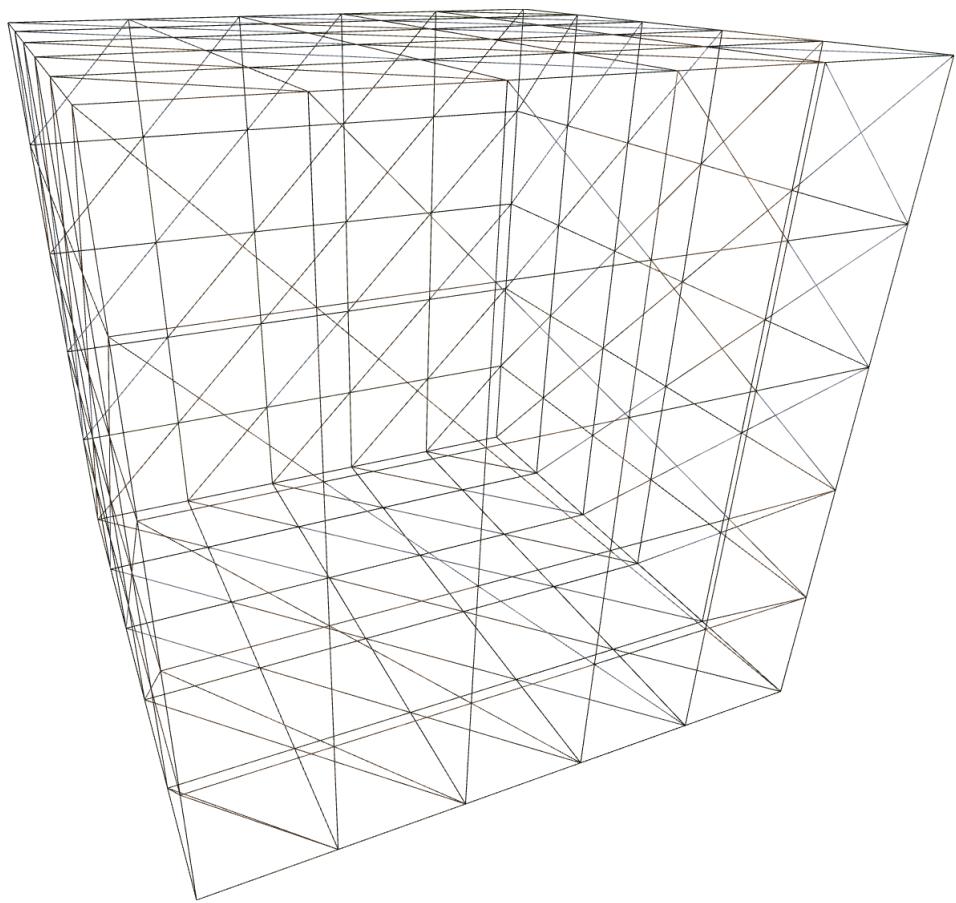


Abb. 7.4: Ein  $5 \times 5 \times 5$  Chunk in dem das Gittermodell optimiert wurde. Ohne Backface-Culling werden die nicht sichtbaren Dreiecke trotzdem gerendert.

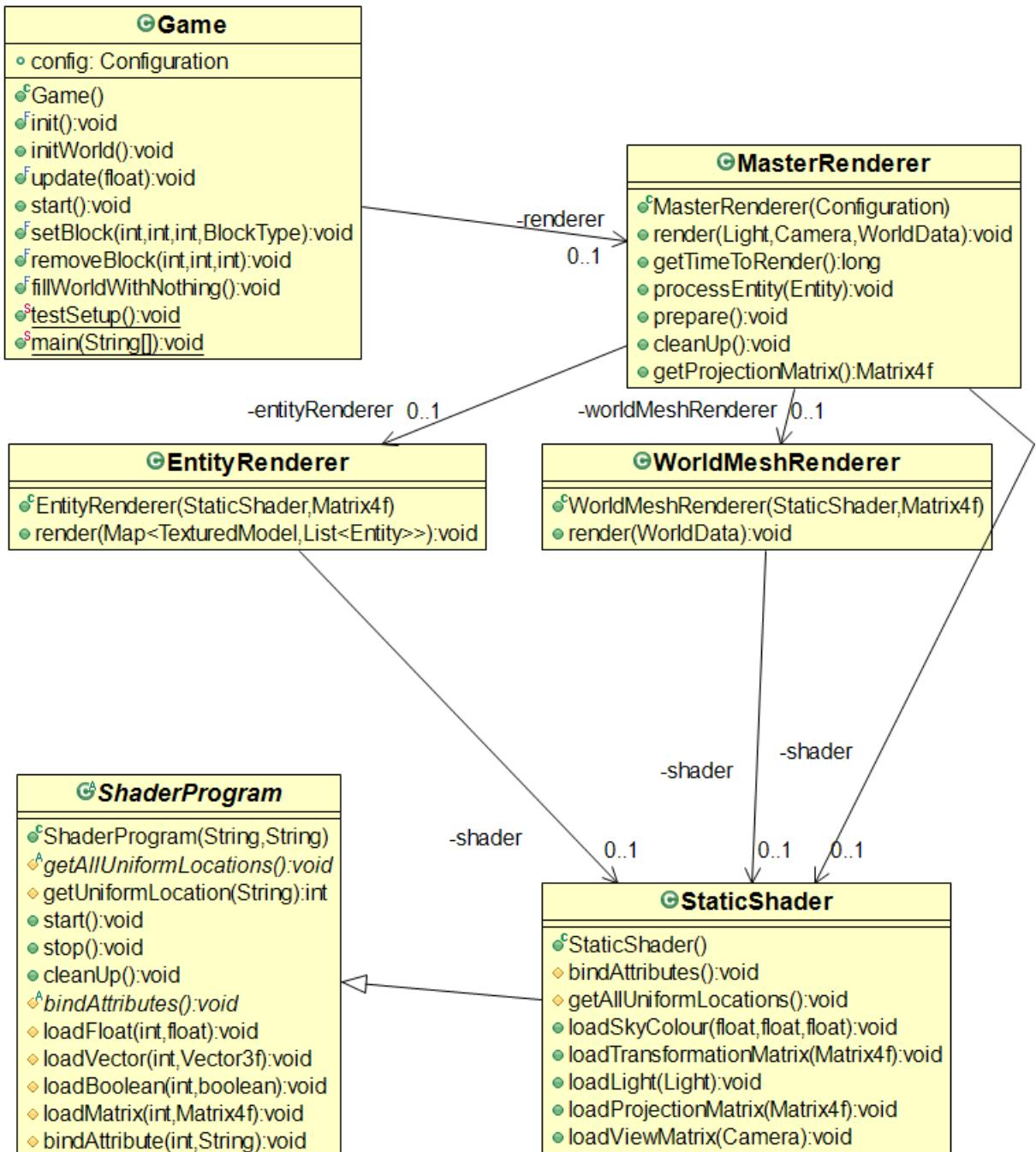


Abb. 7.5: Das Klassendiagramm der Grafikpipeline.

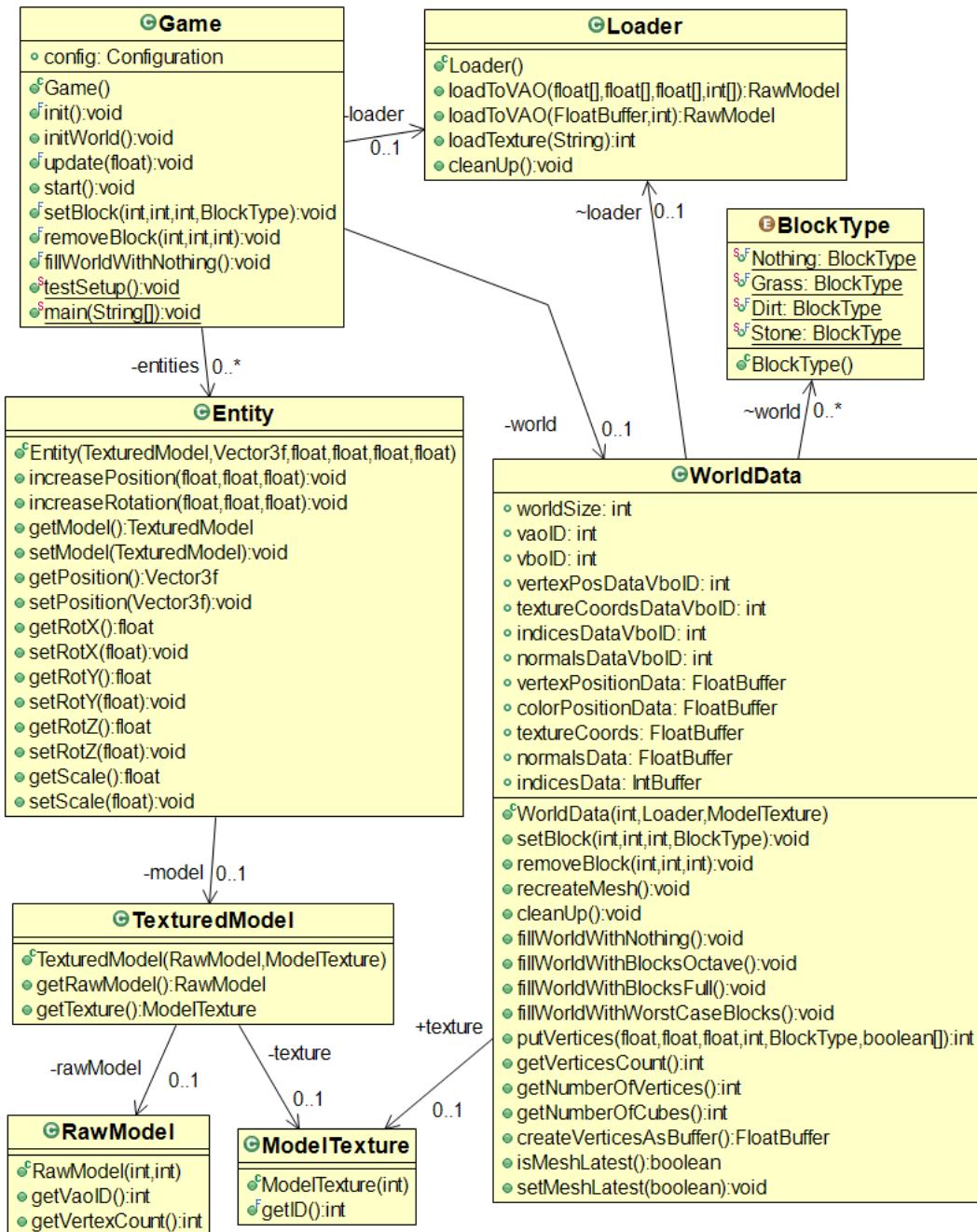


Abb. 7.6: Das Klassendiagramm der Entities und der Welt.

# Literaturverzeichnis

- [1] BURGGRAF, Lorenz: *Jetzt lerne ich OpenGL*. 1. Aufl. MÃŒnchen : Pearson Deutschland GmbH, 2003. – ISBN 978-3-827-26237-0
- [2] SHREINER, Dave ; SELLERS, Graham ; KESSENICH, John M. ; LICEA-KANE, Bill: *OpenGL Programming Guide - The Official Guide to Learning OpenGL, Version 4.3*. 8. Aufl. Amsterdam : Addison-Wesley, 2013. – ISBN 978-0-132-74843-8
- [3] MC SHAFFRY, Mike: *Game Coding Complete, Fourth Edition* -. Clifton Park, NY : Cengage Learning, 2012. – ISBN 113-3-776-582-
- [4] GERDELAN, Anton: *Anton's OpenGL 4 Tutorials*. 7. Auflage. 2014
- [5] SELLERS, Graham ; WRIGHT, Richard S J. ; HAEMEL, Nicholas: *OpenGL SuperBible - Comprehensive Tutorial and Reference*. 6. Aufl. Amsterdam : Addison-Wesley, 2013. – ISBN 978-0-133-36508-5
- [6] NYSTROM, Robert: *Game Programming Patterns*. 1. Aufl. Genever | Benning, 2014. – ISBN 978-0-990-58290-8
- [7] LYSENKO, Mikola: *Meshing in a Minecraft Game*. <https://0fps.net/2012/06/30/meshing-in-a-minecraft-game/>. – [Online; Stand 03. September 2016]
- [8] VIRAG, Gerhard: *Grundlagen der 3D-Programmierung - Mathematik und Praxis mit OpenGL*. München : Open Source Press, 2012. – ISBN 978-3-941-84175-8
- [9] GREGORY, Jason: *Game Engine Architecture*. 2. Aufl. Taylor & Francis Ltd., 2014. – ISBN 978-14665600178