Fakultät Informatik

Bachelorarbeit

Implementation of a block based graphics engine

eingereicht von:	Tobias Zink Matrikelnummer: 2764045 Studiengang: Allgemeine Informatik OTH Regensburg
betreut durch:	Dipl.-Inform. Dr. rer. nat. Carsten Kern OTH Regensburg Dipl.-Inform. Dr. rer. nat. Klaus Volbert OTH Regensburg

Regensburg, der 16. August 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 16. August 2016

Tobias Zink

Kurzfassung

Meine Bachelorarbeit beschäftigt sich mit dem Thema eine „block-basierte“ Grafikbibliothek in Java mithilfe von LWJGL zu implementieren. TODO: Zusammenfassung sobald Text fertig!

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

1 Einführung

1.1 Motivation & Zielsetzung

Minecraft ist zwar nicht das erste Spiel welches eine Grafik aus Würfeln besitzt, aber unbestritten das bekannteste. Da Minecraft aber nicht quelloffen ist und sich dadurch für den universitären Einsatz nur begrenzt eignet, ist der Wunsch nach Alternativen hier gewaltig. Mit „Minetest“[1] gibt es zwar 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 ich gefunden habe, ist Terasology[2]. Dieses baut zwar auf die gleiche Technologie wie das Projekt meiner Bachelorarbeit auf, ist aber sehr viel komplexer und dementsprechend für die Verwendung an der Hochschule auch nicht ohne weiteres geeignet.

Das Ziel der Arbeit war es eine technische Basis für eine würfelbasierte Grafikbibliothek zu schaffen. Dabei ging es nicht nur darum, Blöcke möglichst einfach auf den Bildschirm zu bekommen, sondern es lag auch ein besonderes Augenmerk darauf, dies im Hinblick auf auf die Leistung möglichst optimal zu machen. Außerdem sollte der Quellcode meiner Arbeit sauber und ausführlich dokumentiert sein, damit folgende Absolventen auf diesen zumindestens in Teilen aufbauen können.

1.2 Gliederung der Arbeit

Zu Beginn meiner schriftlichen Ausarbeitung werde ich die technischen Grundlagen auf Seiten der Grafikkarte (OpenGL) und die verwendete Schnittstelle in Java (LWJGL) beschreiben und abgrenzen, warum ich mich für die verwendete Technologie entschieden habe. Außerdem werde ich noch erklären, wie das wichtigste Entwurfsmuster für Spiele - der 'Game Loop' - in meiner Arbeit umgesetzt ist. Danach gehe ich auf die Architektur und die Struktur der Arbeit ein, und wie man diese anhand eines Beispielprojektes für eigene Übungen einsetzen kann.

[1] Minetest veröffentlicht unter der LGPL auf: http://www.minetest.net
[2] Terasology veröffentlicht unter der Apache-Lizenz auf http://terasology.org

Im Hauptteil werde ich die einzelnen von mir umgesetzten Elemente in technologisch sinnvoller Reihe beschreiben. Schlussendlich gibt es im Fazit noch eine Zusammenfassung und einen Ausblick darauf, wie meine Umsetzung weiter optimiert werden könnte und welche mögliche Erweiterungen der Bibliothek ich sehe.

Kapitel 2

2 Grundlagen

2.1 OpenGL

Da meine Implementierung einer würfelbasierten Grafikbibliothek auf wenig externe Komponenten und Konfigurationen angewiesen sein soll, verwende ich, wenn möglich, wenig Abstraktionsebenen. Die beiden großen Schnittstellen Open Graphics Library (OpenGL) und Direct X sind technisch in der Lage meine Anforderungen zu erfüllen. Für OpenGL habe ich mich aufgrund folgender Punkte entschieden:

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 OpenGL-Standart beschreibt 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 Objekten wie Punkten, Linien und Dreiecken konstruieren (vgl. Shreiner, Dave, 2013).

2.2 LWJGL

Was ist die „lightweight java game library“, warum nutze ich genau diese Programmbibliothek (hardwarenah, Minecraft?) und welche Alternativen gibt es?

LWJGL steht für „lightweight java game library“ und ist eine Bibliothek für Java. Der Schwerpunkt liegt hier, wie aus dem Namen schon unschwer zu erkennen auf Computerspielen. LWJGL bietet keinen sogenannten Szenengraphen[1], sondern macht nur die OpenGL API[2] in Java verfügbar. LWJGL funktioniert hier also tatsächlich nur als Schnittstelle zu den von OpenGL bereitgestellten Funktionen.

[1] Der Szenengraph beschreibt ganz allgemein eine Datenstruktur, welche Elemente einer virtuellen Welt beinhaltet.

[2] API steht für application programming interface, Programmierschnittstelle

Warum nicht Java 3D?

weil

Das wichtigste: keine Shader-Integration (falsch!!)

Warum nicht JOGL?

weil

Warum keine vorhandene 'game engine' nutzen?

weil

Beispiele:

jMonkeyEngine

libgdx

2.3 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 wird verändert und die Ausgabe wird erstellt. Außerdem behält er den Überblick über die verstrichene Zeit, um den Spielverlauf anzupassen. (vgl. [1])

Der mehr oder weniger simpelste Game Loop ist in Abb. 2.1 zu sehen. Hier erkennt man schon, dass die Schleife (bis zum Ende des Spiels) dauerhaft laufen soll. Außerdem müssen eventuelle Änderungen am Zustand des Spiels vorgenommen werden (hier unter **updateGameLogic()**). Außerdem wird eine Ausgabe auf dem Bildschirm (hier **rendern()**) erzeugt.

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

Abb. 2.1: Ein sehr einfacher 'Game Loop'

Dieser einfache Loop (aus Abb. 2.1) erfüllt zum Teil schon meine 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 (Gravitation: y = y-1) würde ein Objekt auf einem schnelleren Rechner schneller fallen als auf einem langsamen, da **updateGameLogic()** öfter ausgeführt wird.

Da Änderungen am Zustand des Spiels aber auch mit zeitintensiven Rechenvorgängen einher gehen können[3], kann es auch sein dass **rendern()** 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, habe ich meinen Game Loop wie in Abb. 2.2 angepasst. 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 Physikberechnungen nun z.B. einfach mit dem Delta multipliziert werden, wodurch sie in Relation zur Zeit laufen und nicht mehr zur Geschwindigkeit des Rechners.

In der Methode **rendern(delta)** kann man nun ebenfalls die verstrichene Zeit kontrollieren und das Bild nur 60 mal pro Sekunde erzeugen[4]. Dies ist auf den meisten 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         rendern(delta);
9 }
```

Abb. 2.2: Ein verbesserter 'Game Loop', welcher die verstrichene Zeit beachtet

[3] Es könnte beispielsweise ein Pathfinding stattfinden, welches mehrfach durch eine große Anzahl an Blöcken iterieren muss.

[4] frames per second, FPS

Kapitel 3

3 Design

3.1 Architektur

Verwaltung der Welt

Da mit meiner Grafikbibliothek möglichst viele Würfel aus einer annähernd 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 tausend Würfel (vgl. Tabelle 3.1). Da eine meiner Anforderungen daraus besteht mindestens 500.000 Würfel gleichzeitig zu laden (nicht darzustellen), war sehr schnell klar, dass ich nicht einfach für jeden Würfel ein neues Objekt anlegen kann, sondern eine Datenstruktur benötige, die es möglich macht eine Welt zu verwalten.

n	**n^3**	**Würfel gesamt**	*Speicher*
1	1	1	4 Byte
10	10^3	1.000	0,004 MB
100	100^3	1.000.000	4 MB
500	500^3	125.000.000	500 MB
1000	1000^3	1.000.000.000	4 GB

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

Um also eine unter Umständen 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[1] enthalten (MEHR SIEHE MESHING - TRADEOFF - VERWEIS). Wenn diese Teile nun nur jeweils aus 16 x 16 x 16 Würfeln bestehen, werden pro Chunk nur wenige MB 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 steht, laden. Da man dies in jede Richtung tun muss, ergibt sich folgende Formel: $(6 + 6 + 1)^3 = 2.197$ Chunks oder allgemein: $(n + n + 1)^3$.

Um die Welt bzw. einzelne Teile im Speicher zu halten, bin ich auf zwei unterschiedliche Verfahren gekommen:

[1] Wenig ist hier relativ, bei Minecraft handelt es sich um 65.536 Würfel

1. ein dreidimensionales Feld 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 das 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 wegfallen zu lassen bzw. mit einem Knoten zu repräsentieren.

HIER VERGLEICH ZWISCHEN ARRAY UND OCTREE (evtl. BENCHMARK? auf jedenfall O(n))!!

Klassendiagram

asdf

Komponentendiagram

asdf

3.2 Dateistrukturierung

Ein kurzer Überblick wie die Arbeit strukturiert ist, wie die Pakete aufgebaut sind, wie die Bibliotheken eingebunden sind. Evtl. noch etwas zur Kommentierung des Quellcodes.

3.3 Weltgenerierung

Evtl. unter Realisierung? Falls nicht mehr umgesetzt dann in Ausblick?

3.4 Grafikpipeline

Eventuell weiter nach hinten? Anderer Titel. Hier beschreiben das es in opengl 3.0 kein begin –> draw mehr gibt. Dann die „großen“ Drei Verfahren und warum letzendlich für VBO / VBA entschieden. Maybe verweis zu instanced rendering?

3.5 Beispielprojekt

Ähnlich wie in overview.html aus Javadoc!

Kapitel 4

4 Realisierung

4.1 Definition eines Würfels

Die wichtigste geometrische Figur für meine Arbeit ist der Würfel. Einen Würfel konstruiere ich aus sechs Seiten und jede Seite wiederum aus zwei Dreiecken (4.1). Ein Dreieck entsteht aus drei „Punkten“ welche ich durch Striche verbinde. Jeder „Punkt“ wird durch drei Koordinaten (y, x, z) definiert welche ich im folgenden einfach Punkt nenne. Daraus ergibt sich, dass ich für einen einzelnen Würfel zwölf Dreiecke benötige. Da die Eckpunkte der Dreiecke sich überlagern, würden theoretisch acht Punkte ausreichen um einen Würfel zu beschreiben.

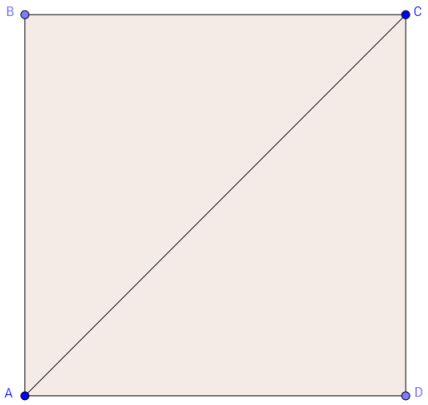

Abb. 4.1: Ein Quadrat welches aus zwei Dreiecken besteht, welche beide jeweils aus drei Punkten erstellt wurden.

In einem dreidimensionalem Koordinatensystem könnten die Punkte wie in Tabelle 4.1 aussehen, wobei x hier die Horizontale, y die Vertikale und z die Tiefe darstellen würde. Wenn man diesen Würfel nun ausgeben würde sähe dies wie in Abb. 4.2 aus.

test

test

Position	x	y	z
A	0	0	0
B	1	0	0
C	1	0	1
D	0	0	1
E	0	1	0
F	1	1	0
G	1	1	1
H	0	1	1

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

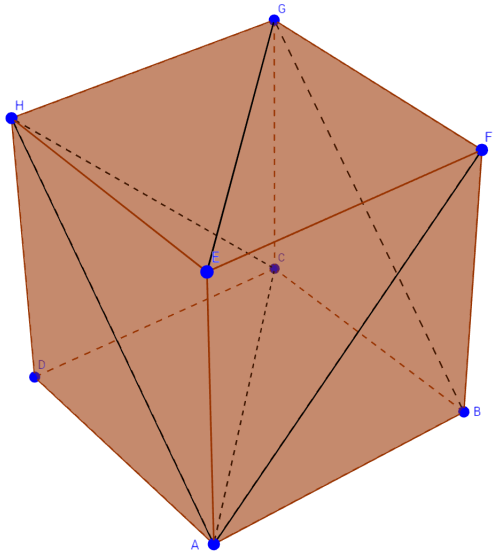

Abb. 4.2: Ein Würfel aus den in Tabelle 4.1 definierten Punkten.

4.2 Umsetzung mit Entities

„single cube“ Ansatz, warum es gescheitert ist und warum der umgesetzte EntityRenderer trotzdem nötig ist für die Arbeit!!

4.3 Aufbau eines Gittermodells

Wie ist das mesh aufgebaut und wie erstelle ich es in einem naiven Ansatz in dem einfach alle Würfel erstellt werden.

4.4 Optimierung des Gittermodells

Welche Möglichkeiten zur Optimierung des mesh existieren, welche habe ich umgesetzt und wie sähe ein theoretisch optimales Gittermodell aus. Warum ein „tradeoff“ zwischen

„draw-calls“ und Arbeitsspeicher?

Möglichkeiten zur Optimierung des Gittermodells:

- Würfel aus dem Gittermodell entfernen, welche nicht dargestellt werden.
  - Würfel, welche auf allen sechs Seiten von anderen Würfeln umgeben sind
  - Würfel, welche sich „hinter“ dem Spieler bzw. der Kamera befinden
  - Würfel, welche sich hinter anderen Würfeln befinden (z.B. Berge oder Höhlen)
- Würfel nicht mehr komplett anzeigen sondern nur noch die Seiten, welche auch dargestellt werden bzw. sichtbar sind.
- Würfel nicht mehr als einzelne Würfel dem Gittermodell hinzufügen, sondern große Würfel mit sich wiederholender Textur belegen, so dass die Illusion von mehreren Würfeln entsteht.

4.5 Shader

Wie sehen meine Shader aus, wie werden sie an openGL weitergegeben und welche Möglichkeiten gibt es hier um rechenintensive Vorgänge auf die Grafikkarte auszulagern.

4.6 Kamera

Gibt es eine Kamera? Das Modell der Kamera in der Computergraphik und meine Umsetzung. Erklärungen zu (meinen) View-Matrizen und deren Aufbau.

4.7 Texturen

Wie werden meine Texturen geladen, wie werden sie auf die Würfel projiziert und warum ich mich für einen Texturatlas entschieden habe. Ausblick auf Array Texture.

4.8 Eingabeverarbeitung

Wie fange ich Tastatureingaben ab, wie werden Änderungen im shader benutzt. Wie wird die Mauseingabe umgesetzt und wie werden einzelne Würfel mit der Maus ausgewählt (FALLS NOCH UMGESETZT!).

4.9 Entfernen von Würfeln

TODO: Umsetzung im Programmcode

Kapitel 5

5 Fazit

5.1 Zusammenfassung

- Kontext zum Studium herstellen
- Schwierigkeiten und Fehleinschätzungen?

5.2 Ausblick

- Weitere Optimierungen im Mesh
- Vielleicht chunks
- Collision Detection?

Glossar

API	Eine Programmierschnittstelle zur Anbindung von eigenen Programmen an fremde Implementierungen. (eng.: application programming interface)
Chunk	Ein Teil einer Welt. Besteht aus einer festgesetzten Anzahl von Würfeln.
LWJGL	Lightweight Java Game Library kurz LWJGL ist eine Implementierung der OpenGL API in der Programmiersprache JAVA.
OpenGL	Open Graphics Library, ein Standard, welcher zur Darstellung von 3D-Grafiken genutzt wird.
Rendern	Die grafische Ausgabe über die Grafikkarte von einzelne Objekten oder kompletten Szenen. Findet im Idealfall ungefähr 60 mal pro Sekunde statt.
Szenengraph	Der Szenengraph beschreibt ganz allgemein eine Datenstruktur, welche Elemente einer virtuellen Welt beinhaltet. Beispielsweise: eine Sammlung von Objekten, Beleuchtung und Animation.
Vertex	Mit Vertex ist ein Punkt mit einer X einer Y und einer Z Koordinate gemeint. (Mehrzahl: vertices)

Literaturverzeichnis

[1] Nystrom, Robert: *Game Programming Patterns* -. 1. Aufl. Genever | Benning, 2014. – ISBN 978–0–990–58290–8

[2] 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

[3] 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

[4] Virag, Gerhard: *Grundlagen der 3D-Programmierung - Mathematik und Praxis mit OpenGL*. München : Open Source Press, 2012. – ISBN 978–3–941–84175–8

[5] Gregory, Jason: *Game Engine Architecture*. 2. Aufl. Taylor & Francis Ltd., 2014. – ISBN 978–14665600178