Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Chris Michael Marquardt

Automatisierte Generierung von Baustein-Prototypen aus 3D-Modellen

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Chris Michael Marquardt

Automatisierte Generierung von Baustein-Prototypen aus 3D-Modellen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Philipp Jenke
Zweitgutachter: Prof. Dr. Stefan Sarstedt

Eingereicht am: 12. Dezember 2014

Chris Michael Marquardt

Thema der Arbeit
Automatisierte Generierung von Baustein-Prototypen aus 3D-Modellen

Stichworte
Informatik, Computergrafik, Voxel, Bausteine

Kurzzusammenfassung
Um schnell und kostengünstig Prototypen von Produkten herzustellen wurde ein Algorithmus implementiert, der aus einer 3D-Vorlage ein auf Bausteinen basierendes Modell generiert. Dieses soll, am Ende des Prozesses, aus nur einer zusammenhängenden Komponente bestehen und so stabil wie möglich sein.

Chris Michael Marquardt

Title of the paper
Automatic generation of a brick prototype from a 3D model

Keywords
Computer Science, computer graphics, voxel, bricks

Abstract
An algorithm, which generates brick models from a 3D template, was implemented to produce fast and cost efficient prototypes. At the end of the process the model should be made of a single connected component and be as stable a possible.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

1 Einleitung

1.1 Motivation

Bei der Entwicklung von neuen Produkten wird häufig Rapid Prototyping angewendet. Dabei werden, in einem sich wiederholenden Prozess, Prototypen erstellt, die getestet und weiterentwickelt werden. Deswegen müssen diese relativ schnell und günstig gebaut werden können, um diesen kontinuierlichen Arbeitsprozess aufrecht zu erhalten.

Eine Möglichkeit ist die Verwendung von 3D-Druckern, da diese ohne größeren Aufwand ein schon bestehendes Modell direkt in einen Prototyp verwandeln können. Allerdings benötigen die Drucker derzeitig noch sehr lange. Für ein Produkt, welches die Größe eines Buches besitzt, beläuft sich die Dauer auf über 12 Stunden. (Mueller u. a. (2014)) Bei komplexeren Modellen sogar noch länger. Diesem Umstand ist es geschuldet, dass derzeitig keine schnelleren Iterationen in der Entwicklung möglich sind.

Diese Bachelorarbeit stellt einen Ansatz vor, wie man zu noch schnelleren Iterationen kommen kann, ohne dabei den Aufwand deutlich zu vergrößern. Die Idee hinter diesem Ansatz ist, dass die Prototypen heutzutage vor allem am Computer modelliert werden und deshalb schon ein 3D-Modell des Produktes existiert. Dieses wird, mit dem hier vorgestellten Algorithmus, in ein Baustein-Modell umgewandelt. Je nach Auflösung dauert dieser Schritt maximal ein paar Minuten. Danach muss man nur noch das Produkt, aus zum Beispiel Lego-Bausteinen, zusammenbauen.

Vor allem in der frühen Phase der Entwicklung, bei der es noch nicht auf die genauen Abmessungen und Details ankommt, kann mit dieser Lösung, im Gegensatz zu einem 3D-Drucker, viel Zeit gespart werden.

1.2 Problemstellung

Es gibt insgesamt drei Aufgaben, die gelöst werden müssen, um aus einem 3D-Modell ein darstellbares und stabiles Baustein-Modell zu generieren. Ein Beispiel zeigt die Abbildung 1.1, in welcher der Stanford-Hase konvertiert wurde.

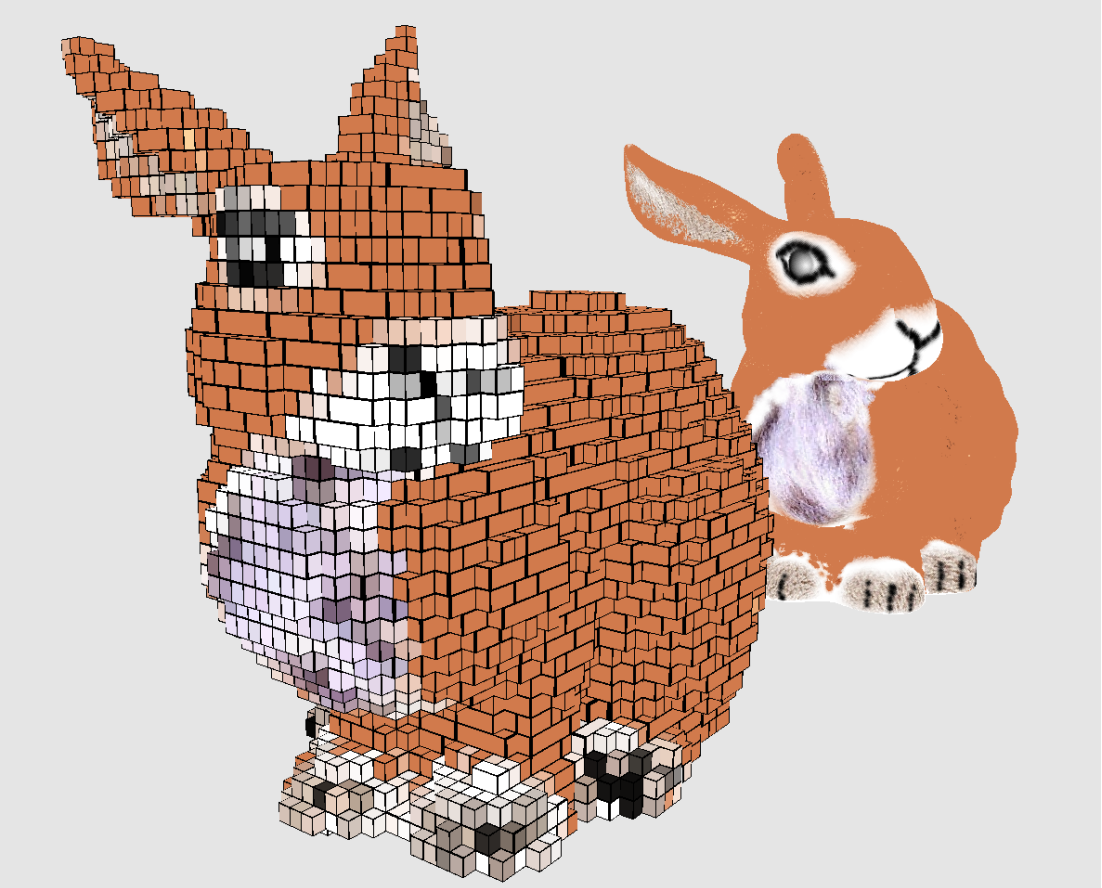

Abbildung 1.1: Der Stanford-Hase als 3D- und Baustein-Modell mit Beibehaltung der Textur

Die erste Hürde ist die Umwandlung eines 3D-Modells in eine Datenstruktur, die es erlaubt Bausteine aus dieser zu generieren. Deswegen wird das gegebene Dreiecksnetz in ein Voxelgitter umgewandelt. Dabei findet eine Diskretisierung statt, durch die Informationen verloren gehen. In diesem Falle wird die daraus resultierende Voxelmenge, die eine vordefinierte Auflösung besitzt, nicht mehr so genau wie das zu Grunde liegende 3D-Modell.

Als zweite Aufgabe müssen die Voxel zu Bausteinen zusammengefügt werden. Dabei soll das Baustein-Modell die folgenden zwei Anforderungen erfüllen. Das Modell sollte erstens nur

aus einem Stück bestehen, so dass Teile nicht einfach abfallen und zweitens möglichst stabil sein. Stabilität bedeutet in diesem Falle, dass einzelne Bausteine, beziehungsweise Teilstücke, mit möglichst vielen anderen Bausteinen verbunden sind.

Als dritte Problemstellung gilt es das generierte Baustein-Modell wieder in ein Dreiecksnetz und damit in ein 3D-Modell zurück zu transformieren. Dadurch kann zum Beispiel eine Anleitung zum Bau des Ergebnisses angezeigt werden.

1.3 Ziel dieser Arbeit

Das Ziel dieser Bachelorarbeit ist es zunächst die Algorithmen zur Lösung der Problemstellungen, wie in Kapitel 1.2 beschrieben, genau zu erklären und in das vorgegebene Framework cgResearch umzusetzen.

Das Framework stellt dabei schon die Implementierung von Dreiecksnetzen zur Verfügung. Dadurch kann direkt mit 3D-Modellen gearbeitet werden, ohne sich Gedanken machen zu müssen, wie man ein solches Modell lädt oder speichert.

Um das spätere Arbeiten mit der Software zu erleichtern, soll für die zwei selbst entwickelten Datenformate, welche Zwischenergebnis und Endergebnis repräsentieren, die Möglichkeit bestehen, diese in Dateien zu speichern und wieder zu laden.

Außerdem soll, für texturierte 3D-Modelle, das programmierte Softwaremodul die Bausteine so färben, dass die Textur grob auch über die Diskretisierung erhalten bleibt.

1.4 Gliederung der Arbeit

Die Arbeit gliedert sich in folgende sieben Kapitel.

Nach dem derzeitigen Kapitel folgt eine Einführung in die Grundlagen des Themas und der benötigten Begriffe, die in dieser Bachelorarbeit vorkommen.

Im dritten Kapitel werden die benutzten Algorithmen vorgestellt, die zur Lösung der gegebenen Problemstellung benutzt werden.

Das darauf folgende Kapitel befasst sich mit der Implementierung und gibt einen Überblick über die erstellten Klassen und Methoden des Softwaremoduls.

Das fünfte Kapitel evaluiert die Implementierung anhand der gemessenen Performance. Außerdem wird geprüft, ob die vorher in Kapitel 1.3 aufgestellten Ziele eingehalten wurden.

Kapitel 6 beschreibt weitere mögliche Erweiterungen der benutzten Algorithmen und gibt einen Ausblick auf deren Implementierung.

Das letzte Kapitel enthält abschließend eine Zusammenfassung und ein Fazit für die gesamte Arbeit.

2 Grundlagen

2.1 Allgemeine Begriffe

Punkt:
Ein Punkt ist ein genau definierter Ort im dreidimensionalen Raum.

Ecke:
Eine Ecke ist ein Objekt ohne definierten Ort.

Kante:
Eine Kante ist eine Verbindung zwischen zwei Punkten oder Ecken.

Dreiecksnetz:
Ein Dreiecksnetz besteht aus Punkten und Kanten. Genau drei Kanten zwischen drei verschiedenen Punkten bilden ein Dreieck.

Auflösung:
Die Auflösung beschreibt im Falle der Computergrafik die Dichte von Punkten in einem vorher definiertem Raum.

Voxel:
Ein Voxel ist eine Sammlung von Daten an einem gewissen Punkt. In dieser Arbeit hat ein Voxel immer einen Typ, eine Größe und möglicherweise einen Farbwert.

Voxelgitter:
Ein Voxelgitter ist ein Gitternetz mit einer bestimmten Auflösung n in dem sich n^3 Voxel befinden.

Graph (ungerichtet):
Ein Graph besteht aus einer Menge von Ecken und Kanten. Dabei verbindet eine Kante immer zwei Ecken (in beide Richtungen) miteinander.

Teilgraph:
Ein Teilgraph besitzt eine Untermenge von Ecken und Kanten eines anderen Graphens.

Zusammenhang:
In der Graphentheorie beschreibt der Zusammenhang, ob jede Ecke mit jeder anderen Ecke über eine beliebige Folge von Kanten verbunden ist.

Gelenkpunkt:
Ein Gelenk- oder Artikulationspunkt ist eine Ecke, durch dessen Entfernung ein Graph nicht mehr zusammenhängend ist.

Bounding Box:
Eine Bounding Box ist ein Würfel, der ein Objekt genau einschließt, ohne dass sich Teile des Objekts außerhalb dieses Würfels befinden.

2.2 Bisherige Arbeiten

Im Februar 2001 publizierte die LEGO A/S die Problemstellung, wie man jedes mögliche 3D-Modell mit LEGO-Bausteinen nachbauen kann. Bis zu diesem Zeitpunkt gab es nur theoretische Annäherungen an das Problem ohne praktischen Bezug wie zum Beispiel die Kostenfunktion für das Simulated Annealing (Gower u. a. (1998)).

Nach der Veröffentlichung gab es im Laufe der Jahre verschiedene Ansätze zur Lösung des Problems angefangen bei evolutionären Algorithmen (Petrovic (2001)), über Beam Search Algorithmen (Winkler (2005)), bis hin zu zellularen Automaten (Van Zijl und Smal (2008)). Diese Arbeiten zielten vor allem auf den Aspekt der minimalen Anzahl an Bausteinen im Ergebnis-Modell ab.

Erst neuere Publikationen wie die Paper (Testuz u. a. (2013)) und (Mueller u. a. (2014)) berücksichtigen dabei die Stabilität und damit auch die tatsächliche Möglichkeit das Modell zu bauen.

Ein Teil dieser Arbeit benutzt den beschriebenen Algorithmus zur Generierung eines Baustein-Prototyps von (Testuz u. a. (2013)).

3 Beschreibung des Algorithmus

Wie bereits im Kapitel 1.2 beschrieben, unterteilt sich der gesamte Algorithmus in insgesamt drei Teile, die folgend beschrieben werden.

3.1 Voxelisierung

Für den Voxelisierungsprozess wird der „Parity Count"-Algorithmus benutzt, entwickelt von Fakir S. Nooruddin und Greg Turk und publiziert im „Simplification and Repair Polygonal Models Using Volumetric Techniques" Paper (Nooruddin und Turk (2003)).

Der Algorithmus durchsticht das 3D-Modell aus mehreren Richtungen mit Strahlen, anhand derer berechnet wird, ob ein Voxel innerhalb oder außerhalb des Modells liegt.

Es werden insgesamt N^2 parallele Strahlen bei einem N^3 Voxelgitter benutzt. Diese durchlaufen dabei die Zentren aller Voxel. Es wird eine vorher definierte Anzahl an unterschiedlichen Richtungen benutzt, um das Modell zu durchwandern. Normalerweise werden dazu die X-, Y- und Z-Achsen des Raumes genutzt.

Trifft dabei ein Strahl auf eine Oberfläche des Modells erhöht sich der Parity-Zähler. Alle Voxel entlang dieser Geraden werden anhand dieses Zählers bewertet, ob sie sich innerhalb oder außerhalb befinden. Ist der Zähler bei dem Voxelzentrum gerade, befindet er sich außerhalb des Modells. Bei ungerader Zahl innerhalb des Modells.

Diese Bewertungen werden sich für jeden Voxel gemerkt und anhand der Bewertungen aller Strahlen wird per Mehrheitsentscheidung gewählt, ob sich ein Voxel innerhalb oder außerhalb des 3D-Modells befindet. Je nachdem wird das Voxel gefüllt oder leer gelassen. Bei einem Gleichstand gilt ein Voxel als leer.

Um ein Unentschieden zu vermeiden, wird immer eine ungerade Anzahl an Richtungen gewählt, die die Strahlen verfolgen. Allerdings kann es trotzdem passieren, dass ein Unentschieden auftritt. So werden Strahlen für ungültig erklärt, die nach dem Durchwandern des Raumes, in dem sich das Modell befindet, einen ungeraden Parity-Zähler besitzen. Dadurch werden Strahlen, die sich durch ein Loch im Modell bewegen, nicht berücksichtigt. Allerdings kann es passieren, dass ein Strahl durch genau zwei solcher Löcher im Modell wandert und dabei viele

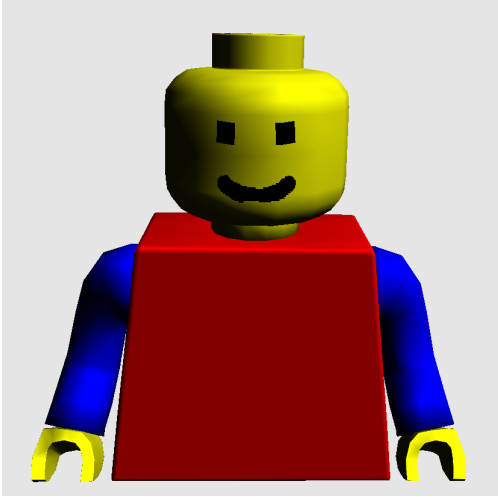

Abbildung 3.1: 3D-Modell einer Lego-Figur

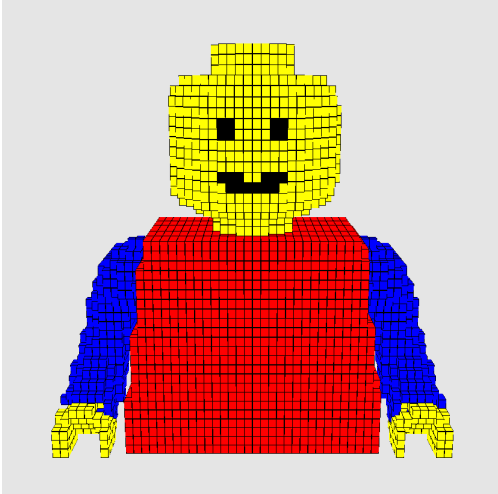

Abbildung 3.2: Ergebnis der Voxelisierung anhand einer Lego-Figur

Voxel fehlerhaft markiert. Diese Auswirkungen werden jedoch durch den Mehrheitsentscheid zwischen allen Bewertungen erheblich reduziert.

Um die Anzahl der letztendlichen Voxel (und damit der Bausteine) zu verringern, kann nach dem Voxelisierungsalgorithmus das Voxelgitter ausgehöhlt werden. Gefüllte Voxel, die sich neben einem Leeren befinden, werden als Oberfläche markiert. Voxel, deren Distanz zum nächsten Oberflächen-Voxel niedriger ist, als zuvor festgelegt, werden als Hülle markiert. Die

übrig gebliebenen Innen-Voxel können dann im Lego-Algorithmus ignoriert werden. So fallen viele unnötige Voxel im Inneren des Modells weg, ohne die Stabilität stark zu beeinflussen.

3.2 Lego-Algorithmus

Der Lego-Algorithmus wird im Paper „Automatic Generation of Constructable Brick Sculptures" von Romain Testuz, Yuliy Schwartzburg und Mark Pauly beschrieben (Testuz u. a. (2013)).

Generell lässt sich dieser in zwei Teile unterscheiden. Im ersten Schritt werden die Voxel, bei denen im Folgenden von 1x1 Bausteine ausgegangen wird, per Zufall zu größeren Bausteinen zusammengefügt. Das Ergebnis wird höchstwahrscheinlich aber noch nicht komplett aus nur einem Stück bestehen und noch nicht sehr stark miteinander verbunden sein. Deswegen versucht der zweite Schritt dies auszubessern.

3.2.1 Zufalls-Kombinieren

Die 1x1 Bausteine werden als Erstes nacheinander mit ihren Nachbarn kombiniert. Dabei wird versucht möglichst große Bausteine herzustellen, damit die Konnektivität zwischen den einzelnen Schichten erhöht und die generelle Anzahl an Bausteinen gesenkt wird. Ebene für Ebene wird dabei das Modell von unten nach oben durchlaufen und folgender, wie in Abbildung 3.3 dargestellter, Algorithmus durchgeführt.

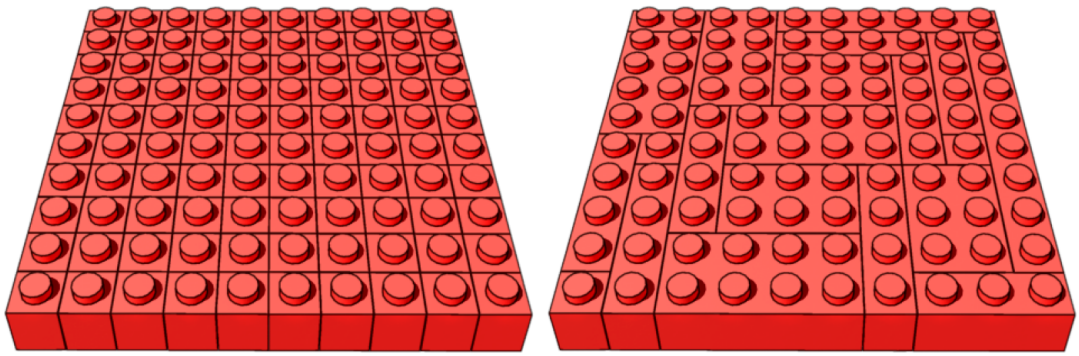

Abbildung 3.3: Beispiel einer Schicht vor und nach dem Zufalls-Kombinieren (Quelle: Testuz u. a. (2013))

Per Zufall wird ein Stein aus der Schicht gewählt und es werden alle möglichen Nachbarn untersucht, mit denen ein neuer Baustein erzeugt werden kann. Dabei werden nur Nachbarn in Betracht gezogen, die einen Stein aus der Menge der möglichen Bausteine mit dem Ausgewählten bilden können. Aus dieser Menge an möglichen Kombinationen wird die genommen,

welche die größte Anzahl an neuen Verbindungen zwischen den Steinen herstellt. Sollte dies für mehrere Nachbarn zutreffen, wird einer davon wiederum per Zufall ausgewählt.

Dieser Vorgang wird so lange wiederholt, bis es keine Möglichkeit mehr gibt auf einer Ebene Bausteine miteinander zu kombinieren.

Wurden alle Schichten bearbeitet, ist die Wahrscheinlichkeit relativ groß, dass Teile des Modells nicht mit dem Rest verbunden sind. Zur Ausbesserung solcher Stellen folgt der nächste Algorithmus.

3.2.2 Optimierung der Stabilität

Bei dieser Problemstellung steht die Stabilität in direkter Verbindung mit der Anzahl an Verbindungen zwischen den einzelnen Bausteinen im Modell. Das bedeutet, umso mehr Verbindungen ein Baustein mit anderen Steinen in den Schichten darüber und darunter hat, desto stabiler wird insgesamt das Bauwerk.

Aus diesem Grund werden die Verbindungen zwischen den einzelnen Bausteinen auf einen Graphen gemappt, so dass ein Stein eine Ecke und eine Verbindung eine Kante darstellt. Dabei spielt es keine Rolle, mit wie vielen einzelnen „Knöpfen“ ein Stein mit einem Anderen in Kontakt steht. In Abbildung 3.4 werden zwei Beispiele von solchen Graphen gezeigt.

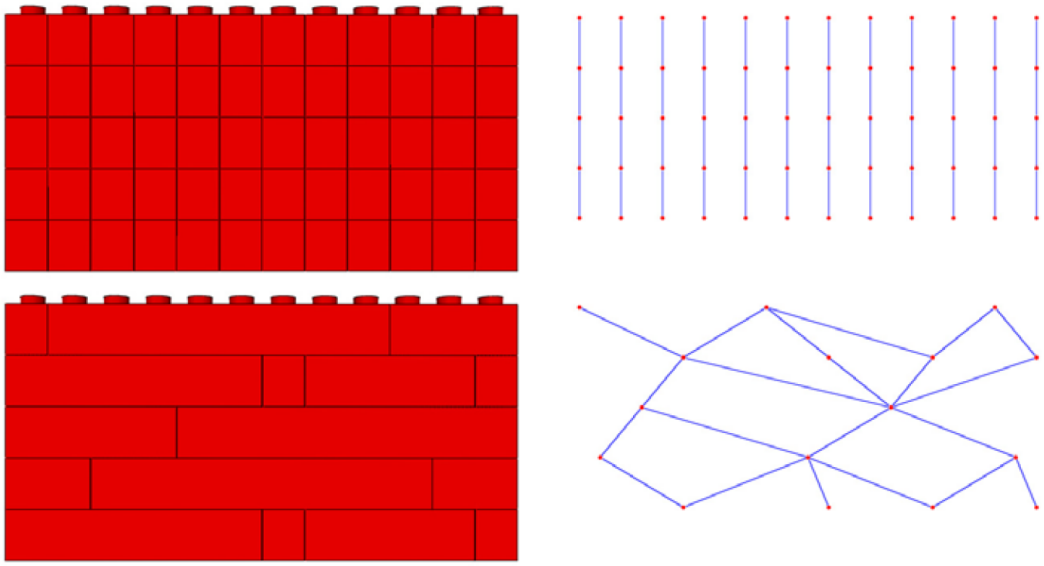

Abbildung 3.4: Beispiele von Baustein-Modellen und deren dazugehöriger Graph (Quelle: Testuz u. a. (2013))

Der entstehende Graph kann dann auf gewisse Merkmale untersucht werden. Zum einen, ob der Graph tatsächlich zusammenhängend ist, denn wenn nicht, sind Teile des Modells nicht mit dem Rest verbunden und können einfach abfallen. Zum anderen wird nach sogenannten schwachen Artikulations- oder Gelenkpunkten gesucht. Im Kapitel 2.1 wurden diese bereits beschrieben. Ein schwacher Gelenkpunkt ist dabei die Erweiterung, dass er zwei Teilgraphen mit je einer Größe von mehr als 1 verbindet.

Nachdem so alle Schwachstellen des Modells ausgemacht wurden, wird nun versucht diese auszubessern. Durch die Zufalls-Kombinierung aus Kapitel 3.2.1 können die Bausteine nicht mehr mit Anderen verbunden werden. Deswegen zerteilen wir die Bausteine an der Schnittstelle zwischen zwei Zusammenhangskomponenten, die Gelenkpunkte, sowie deren direkte Nachbarn, wieder in 1x1 Blöcke auf. An diesen Stellen wird wiederum der Kombinierungsalgorithmus aus Kapitel 3.2.1 angewendet. Allerdings wird nicht mehr der Stein mit den meisten Verbindungen gewählt, sondern es wird per Zufall entschieden. Abbildung 3.5 zeigt ein Beispiel wie der rot markierte Gelenkpunkt, sowie seine Nachbarn zerlegt und wieder miteinander kombiniert werden.

Dieser Prozess wird dabei so lange wiederholt, bis es keine signifikante Änderung im Zusammenhang des Graphen und der Anzahl der Gelenkpunkte gibt.

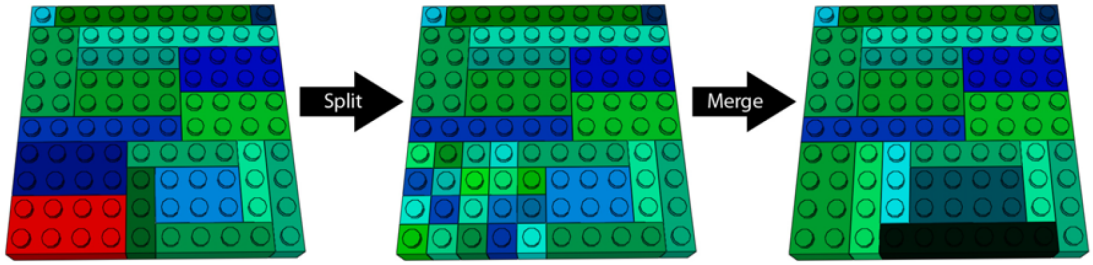

Abbildung 3.5: Beispiel der Zerlegung und erneuten Kombination zur Vermeidung von Gelenkpunkten (rot) (Quelle: Testuz u. a. (2013))

3.3 Visualisierung

Die gewählte Variante zur Visualisierung des fertigen Baustein-Modells ist wegen seiner Einfachheit gewählt worden.

Für jeden Stein im Modell wird ein in der Länge angepasster Würfel erzeugt. Dabei besteht jede Seite des Würfels aus genau zwei Dreiecken. Anhand der Position des Steines wird dieser Würfel an die richtige Stelle im Raum transformiert.

Insgesamt wird dadurch für ein Modell mit n Bausteinen ein Dreiecksmodell mit 12n Dreiecken erstellt.

Um diese Zahl weiter zu senken, könnten zum Beispiel nur die Steine angezeigt werden, die sich tatsächlich an der Oberfläche befinden. Ein anderer Schritt wäre es, Bausteine miteinander zu verschmelzen, die eine größere Fläche bilden. So könnten weitere Dreiecke eingespart werden. Allerdings müsste eine Textur verwendet werden, die die eigentlichen Bausteine innerhalb einer solchen Fläche wieder darstellt.

4 Implementierung

4.1 Entwicklungsumgebung und genutzte Software

Die verwendete Programmiersprache und Sprache der Bibliotheken ist Java 7. Als Entwicklungsumgebung wurde Eclipse Juno und zur Versionisierung Git benutzt.

Das Modul baut dabei auf dem Framework CgResearch von Prof. Dr. Philipp Jenke auf, sowie auf eine externe Bibliothek von Reality Interactive um das Bildformat TGA benutzen zu können. Hinzu kommt die JGraphT Library um mit Graphen arbeiten zu können. CgResearch selbst stellt bereits Klassen und Methoden bereit, um mit grundlegenden Elementen wie Vektoren und Matrizen oder mit komplexen 3D-Modellen zu arbeiten. Dabei wandelt das Framework solche Modelle in eine eigene Dreiecksnetz-Datenstruktur um, auf der man weitere mathematische Operationen durchführen kann. Des Weiteren kann mit Texturen und Shadern gearbeitet werden. Alle Elemente werden in einem Szenengraphen als Nodes gespeichert. Mit Hilfe der Bibliothek JOGL (Java OpenGL) oder einer anderen Engine kann CgResearch diesen Graphen dann rendern.

4.2 Übersicht der Klassen

4.2.1 Das VoxelCloud Paket

Im VoxelCloud Paket befinden sich alle Klassen für das Arbeiten mit einem Voxelgitter. Eine Übersicht des Paketes zeigt Abbildung 4.1.

Die Grundklasse VoxelCloud stellt dabei die allgemeine Datenstruktur zur Verfügung und beschriebt die Eigenschaften eines Voxelgitters, sowie alle sich darin befindlichen Voxel. Ein einzelnes Voxel wird dabei nach einer der vier Typen Außen, Innen, Hülle oder Oberfläche unterschieden. Außen-Voxel sind leer und schneiden mit keinem Teil das Modell. Alle folgenden Typen sind ausgefüllte Voxel und unterscheiden sich nur durch die Position relativ zur Oberfläche des 3D-Modells. Oberflächen-Voxel sind alle, die einen Teil der Oberfläche berühren und deswegen auch einen Farbwert zugewiesen bekommen haben. Je nach gewählter

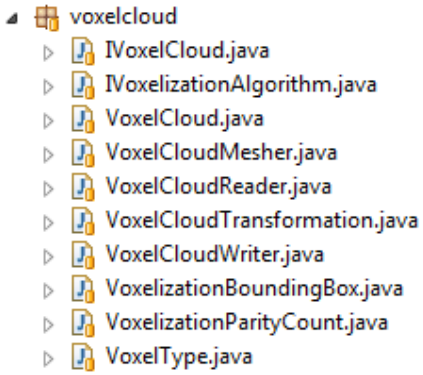

Abbildung 4.1: Übersicht des VoxelCloud Paketes

Hüllengröße füllen Hüllen-Voxel den Raum nach den Oberflächen-Voxeln aus. Innen-Voxel sind alle restlichen Voxel, die sich innerhalb des 3D-Modells befinden.

Wird ein Voxel als Innen bewertet, dann wird gleichzeitig der nächste Punkt vom Voxelmittelpunkt mit der Oberfläche berechnet und dessen Farbwert gespeichert.

Die VoxelCloudTransformation Klasse bietet die Möglichkeit, die Voxel anhand ihrer Position im Voxelgitter zusätzlich als Oberfläche und Hülle zu markieren.

Mit den VoxelCloudWriter und VoxelCloudReader Klassen können die Voxelgitter gespeichert und geladen werden. Das Format ist selbstentwickelt und wird wie folgt beschrieben:

```
// Header (String):
// Lower left corner location (double)
loc x y z
// Voxel dimensions (double)
dim x y z
// Resolution (int)
res x y z
// Number of voxel with a color
col number

// Data (Binary):
// Start of the data section
data
// Color of the voxel at x,y,z (byte/short/int + byte[3])
x1 y1 z1 c1 x2 y2 z2 c2 ...
// Voxel type (byte)
// Order (xyz): 000 .. 100 .. N00 .. 010 .. NN0 .. 001 .. NNN
```

```
18 type1 type2 ...
```

Listing 4.1: .vox File Format

Im Header werden die allgemeinen Daten des Voxelgitters gespeichert. Im Datenteil werden zuerst alle Farben von Voxeln gespeichert und dann der Typ jedes einzelnen Voxels.

Mit der VoxelCloudMesher Klasse kann ein Voxelgitter in ein Dreiecksnetz umgewandelt werden, so dass dieses dann dargestellt werden kann.

4.2.2 Voxelisierung

Zur Umwandlung eines Modells in ein Voxelgitter wird die Klasse VoxelizationParityCount benutzt, die den, wie in Kapitel 3.1 erläutert, „Parity Count“ Algorithmus (Nooruddin und Turk (2003)) verwendet.

Wie erwähnt werden aus drei verschiedenen Richtung Strahlen durch das Modell gestochen, um die einzelnen Voxel zu bewerten. Um Zeit zu sparen werden die verschiedenen Richtungen in unterschiedlichen Threads abgearbeitet. Nur die letztendliche Bewertung jedes einzelnen Voxels wird nicht parallelisiert.

Um herauszufinden, ob ein Strahl das 3D-Modell schneidet, muss jedes Dreieck untersucht werden. Um diesen Vorgang zu beschleunigen wird für alle Dreiecke eine Bounding Box berechnet. Anhand dieser kann schnell bestimmt werden, ob sich ein Strahl in der Nähe eines Dreiecks befindet oder nicht. Wenn ein Dreieck nah ist, wird mit dem Ray/Triangle Intersection Algorithmus (Möller und Trumbore (1997)) der Treffpunkt ermittelt. Wurden alle Treffpunkte des Strahls berechnet werden die Voxel entweder als Innen oder Außen bewertet.

4.2.3 Das BrickCloud Paket

Das BrickCloud Paket, wie in Abbildung 4.3 dargestellt, enthält alle Klassen für das Arbeiten mit einem Baustein-Modell.

Die Klassen RootBrick, ComposedBrick und SpecialBrick beschreiben die verschiedenen Baustein-Typen, die das Modell haben kann. Der RootBrick ist der Baustein, auf dem alle anderen aufbauen. Im Falle von Lego wäre das zum Beispiel der 1x1 Baustein. Ein ComposedBrick dagegen ist ein Stein, der sich aus einer gewissen Anzahl von 1x1 Steinen zusammensetzt. Auch der SpecialBrick setzt sich wie der Composed aus einer gewissen Anzahl des Roots zusammen, kann aber nicht mit anderen Steinen im Lego-Algorithmus verbunden werden. SpecialBricks werden nicht durch den Algorithmus erzeugt, sondern müssen per Hand gesetzt werden.

Aus einer Anzahl von solchen Baustein-Typen setzt sich dann das BrickSet zusammen. Es enthält alle möglichen Typen und merkt sich aus welchem Typ sich andere Typen ableiten.

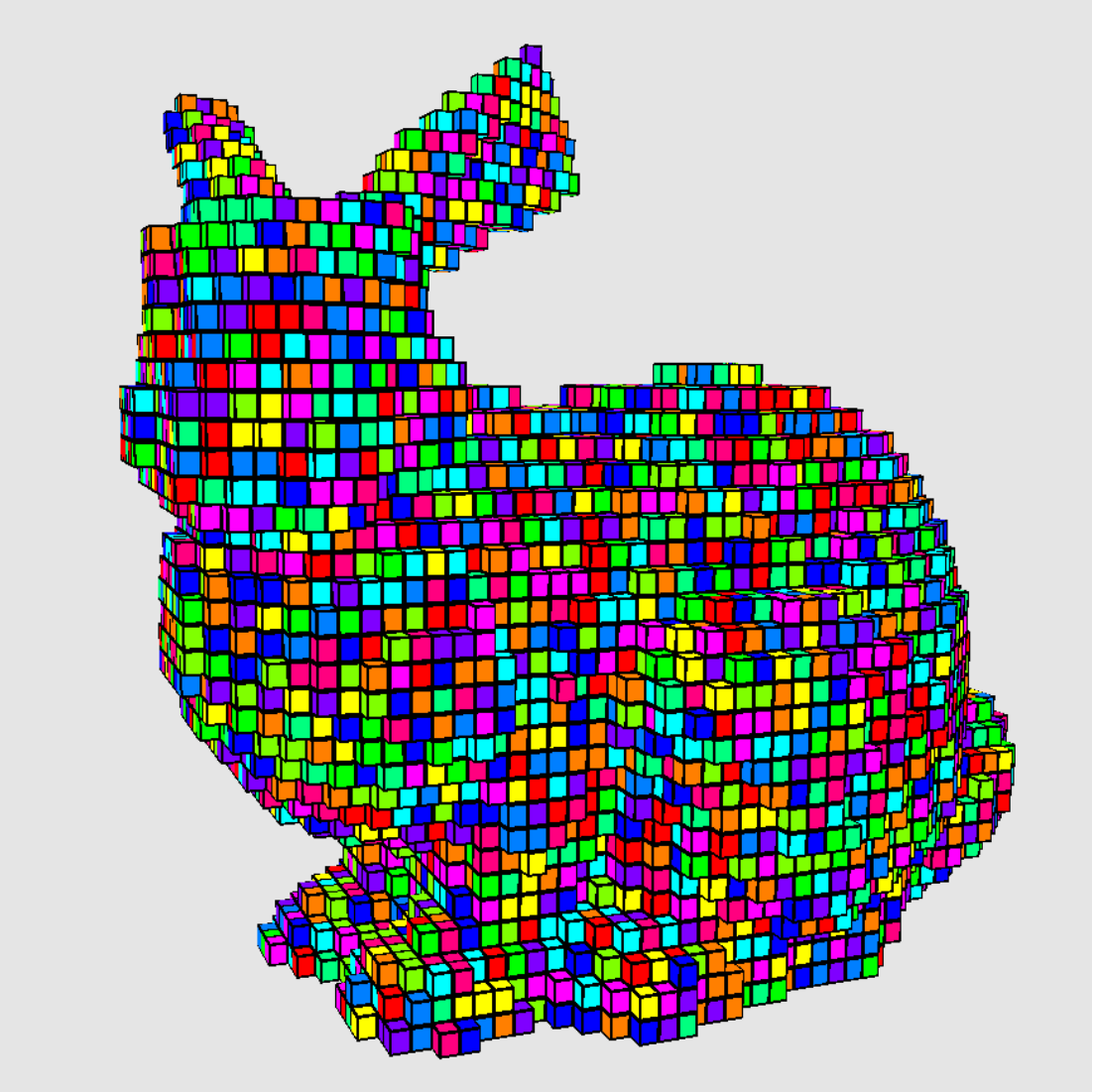

Abbildung 4.2: Der Stanford-Hase als Voxel-Modell

Daraus ergibt sich eine Baumstruktur durch die der Algorithmus entscheiden kann, mit welchen Nachbarn sich ein Baustein verbinden kann und was das Ergebnis davon ist. Das BrickSet enthält dabei auch alle möglichen Farben, die ein Baustein annehmen kann.

Die BrickCloud Klasse stellt das eigentliche Baustein-Modell da. Wie bei der VoxelCloud wird von einem Gitter ausgegangen. In diesem Falle sind dabei die einzelnen Zellen ein Rootbaustein. Um ein Modell zu initialisieren werden eine Position, eine Auflösung des Gitters, sowie ein BrickSet benötigt. Die Klasse erstellt dabei einen Graphen, der die Verbindungen zwischen den

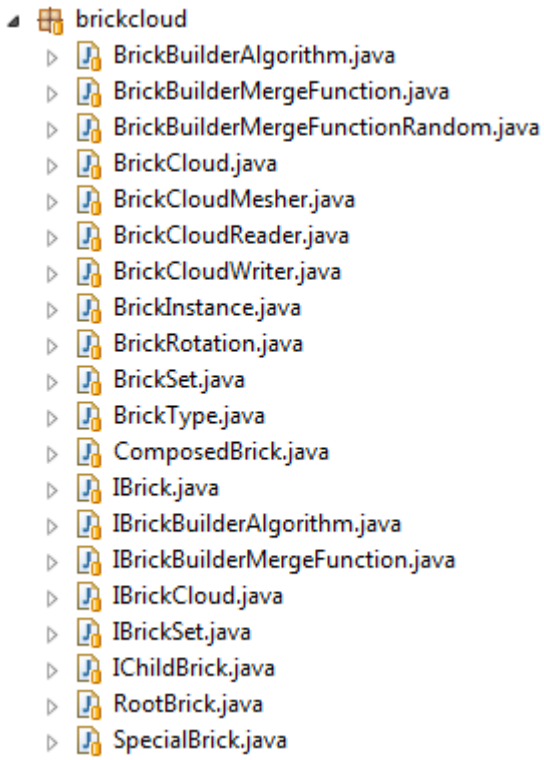

Abbildung 4.3: Übersicht des BrickCloud Paketes

Bausteinen darstellt. Per addBrick Methode können mit den Parametern Position, Rotation, Farbe und Steintyp ein Baustein hinzugefügt werden.

Wie auch bei der VoxelCloud können mit den BrickCloudReader und BrickCloudWriter die Baustein-Modelle geladen und gespeichert werden. Das Format ist wiederum selbstentwickelt und zielt auf eine einfache und platzsparende Speicherung ab.

```
// Header (String):
// Lower left corner location (double)
loc x y z
// Resolution (int)
res x y z
// Number of brick types in the set (without root)
bri number
// Number of brick colors in the set
col number
// Number of bricks in the cloud
cnt number
```

```
12
13 // Data (Binary):
14 // Start of the data section
15 data
16 // Brick types (byte) (1st needs to be the root brick)
17 type ...
18 // when a root brick (3x double)
19         dimX dimY dimZ ...
20 // when a composed brick (3x byte)
21         resX resY resZ ...
22 // when a special brick (byte short 3x byte)
23         specialType unitCount unitX1 unitY1 unitZ1 ...
24 // Brick type colors (int)
25 color1 color2 ...
26 // Every brick (int 3x byte/short/int byte int)
27 type posX posY posZ rot col ...
```

Listing 4.2: .brick File Format

Im Header werden die allgemeinen Daten des Bausteinmodells hinterlegt. Im Datenteil folgen dann erst die einzelnen Bausteintypen. Danach werden alle möglichen Farben des BrickSets abgespeichert und zuletzt folgen der Typ, Position, Rotation und Farbe jedes einzelnen Bausteins.

Die BrickCloudMesher Klasse dient dazu das Modell in ein Dreiecksnetz umzuwandeln, welches dann dargestellt werden kann.

4.2.4 Lego-Algorithmus

Um aus einem Voxelgitter ein Baustein-Modell generieren zu lassen wird die Klasse BrickBuilderAlgorithm genutzt. Wie im Kapitel 3.2 beschrieben ist der erste Schritt die vorhanden Voxel im Gitter zu Root-Bausteine umzuwandeln und gegebenenfalls einzufärben. Im zweiten Schritt werden zuerst die einzelnen Ebenen per Zufall in der mergingAtHeight Funktion kombiniert. Dabei wird die Kostenfunktion BrickBuilderMergeFunction benutzt, die als Kriterium die Anzahl der möglichen Verbindungen zu anderen Bausteinen heranzieht.

```
1 /**
2  * Value of a brick. Higher = better.
3  * @param brickCloud    brick cloud
4  * @param brick         brick type to test
5  * @param pos           position of the brick
```

```
6   * @param rot           rotation of the brick
7   * @return              connections to other bricks above and under
8   */
9  public int valueFunction(IBrickCloud brickCloud, IBrick brick,
10                 IVectorInt3 pos, BrickRotation rot) {
11         // costs = connections / more = better
12         return brickCloud.getBrickConnections(brick, pos, rot);
13 }
```

Listing 4.3: BrickBuilderMergeFunction

Im letzten Schritt wird nun versucht die Stabilität zu verbessern. Dafür wird der Graph der BrickCloud benutzt um die Anzahl der Zusammenhangskomponenten zu bestimmen.

Nach der Bestimmung der größten Komponente werden die Steine aller Anderen in Rootsteine umgewandelt. Ist dies auf allen Schichten passiert, werden diese, mittels der bereits vorher eingesetzten mergingAtHeight Methode und der Kostenfunktion BrickBuilderMergeFunctionRandom, wieder kombiniert. Nach einem solchen Durchlauf wird wiederum die Anzahl der Komponenten ermittelt. Sollte sich diese erhöht haben wird die BrickCloud auf den Stand davor zurückgesetzt. Bei keiner Änderung wird ein Zähler erhöht, der bei einem vorher definierten Wert den Algorithmus abbricht. Durch eine Verringerung der Komponentenanzahl wird dieser Zähler auf Null zurückgesetzt.

Dieser Vorgang wird so lange wiederholt, bis es entweder nur noch eine Zusammenhangskomponente gibt, der Zähler die Schleife abbricht oder die definierte Anzahl an maximalen Durchläufen erreicht wurde.

Nach der Verringerung der Anzahl der Komponenten, wird dieser Prozess, in leicht abgewandelter Form, zur Vermeidung von Gelenkpunkten benutzt. Zuerst werden die Gelenkpunkte mit Hilfe einer Tiefensuche ermittelt. Nacheinander werden diese Steine, sowie deren Nachbarn, in Rootsteine zerlegt. Per mergingAtHeight Methode werden diese Bausteine durch Zufall wieder kombiniert. Nach jedem Gelenkpunkt wird überprüft, ob sich die Anzahl der Komponenten oder Gelenkpunkte erhöht hat. Wird dies festgestellt, so wird die BrickCloud zurückgesetzt. Nachdem alle Gelenkpunkte bearbeitet wurden, wiederholt sich der Vorgang bis die Anzahl auf Null sinkt oder die vorher definierte Menge an Durchläufen erreicht wurde.

Ein mögliches Ergebnis des Algorithmus, angewandt auf den Stanford-Hasen, zeigt die Abbildung 4.4.

Besitzt das 3D-Modell eine Textur, kann diese im Lego-Algorithmus berücksichtigt werden. Nach der Voxelisierung besitzen Oberflächen-Voxel eine Farbe, die auf die 1x1 Rootsteine übertragen wird. Steine aus Hüllen- und Innen-Voxeln werden als farblos markiert. Ist dies

gewünscht, so berücksichtigt die mergingAtHeight Methode bei der Kombination nur noch Bausteine der gleichen Farbe oder welche die farblos sind. Abbildung 1.1 zeigt den Stanford-Hasen unter Berücksichtigung der Textur.

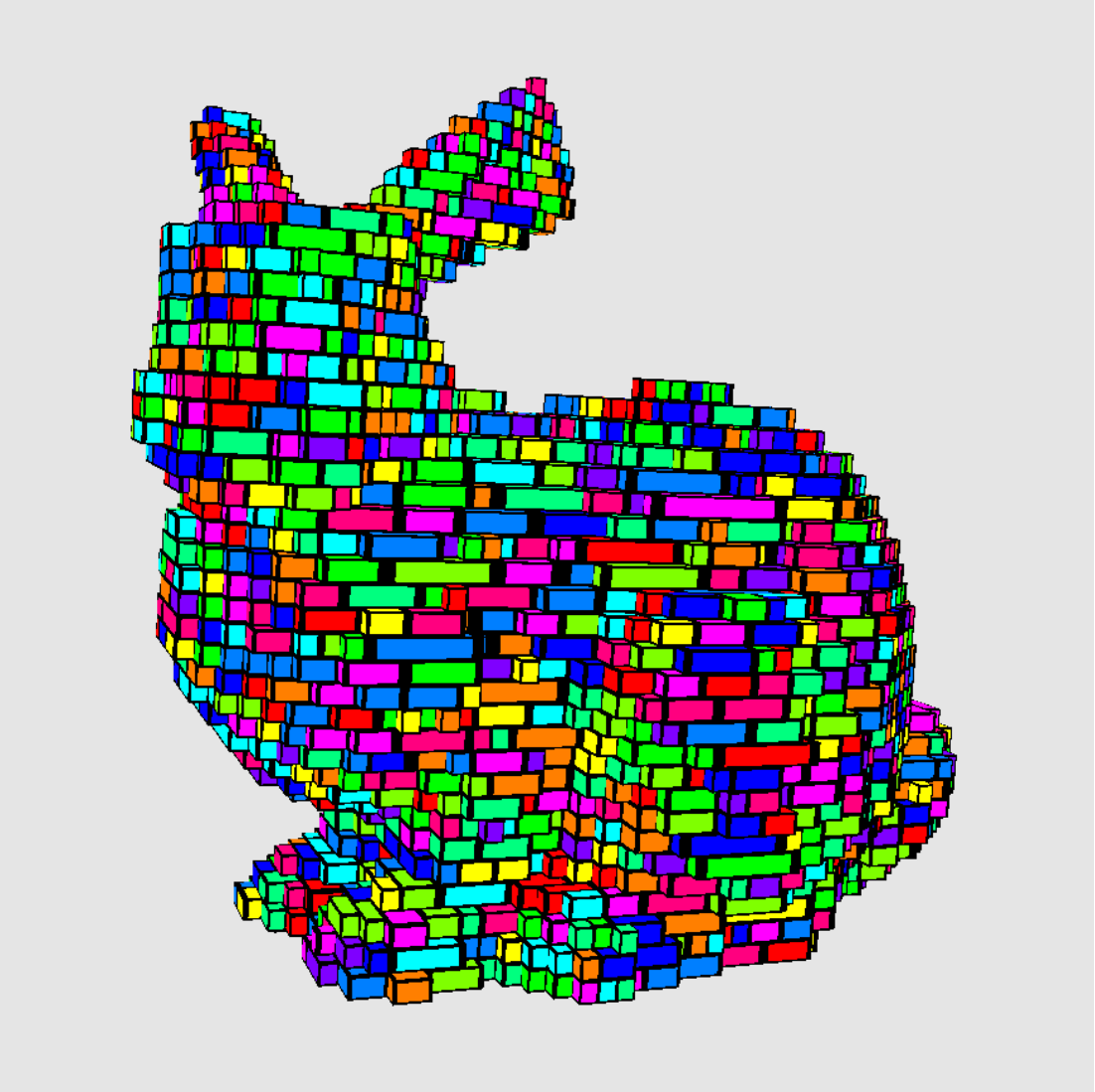

Abbildung 4.4: Der Stanford-Hase als Baustein-Modell

4.3 Schwierigkeiten

Bei der Entwicklung des Lego-Algorithmus hingegen wurde die relativ vage Formulierung der Optimierung der Stabilität ein Problem. Leider gab es zum Zeitpunkt dieser Arbeit keinen frei zugänglichen Sourcecode, der hätte helfen können. So lieferte der Algorithmus lange Zeit nicht annähernd so gute Ergebnisse wie sie im Paper beschrieben sind. Vor allem gab es starke Abweichungen bei der Menge an Zusammenhangskomponenten und schwachen Gelenkpunkten. Erst durch eine stetige Weiterentwicklung konnte ein Ergebnis erzielt werden, das den gegebenen Werten nahe kommt. Ein solches Beispiel war die Veränderung, dass Gelenkpunkte und Komponenten nicht gleichzeitig zerlegt und wieder kombiniert werden, sondern erst nacheinander, wie in Kapitel 4.3 beschrieben.

5 Evaluation

In diesem Kapitel soll die Leistung der Implementierung dargestellt werden.

Zum Testen wurden die drei Modelle (Stanford-Hase, Lego-Figur und Piraten-Yoshi), zu sehen in der Abbildung 5.1, benutzt. Die drei Modelle variieren in der Anzahl der Dreiecke und damit in der Komplexität wie in der Abbildung 5.2 dargestellt.

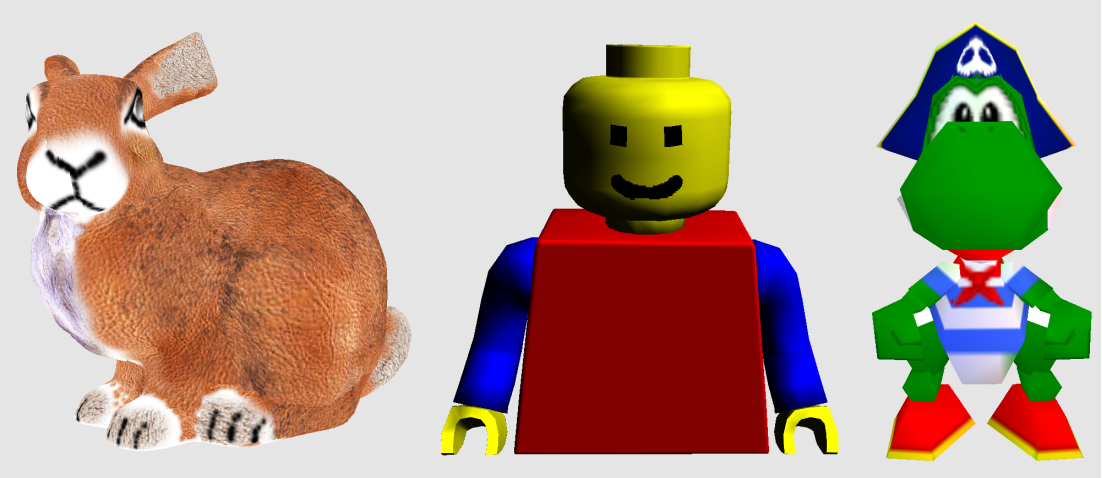

Abbildung 5.1: Die drei Testmodelle

Mesh	Ecken	Dreiecke
Bunny	34834	69451
LegoMan	1288	2520
YoshiPirate	167	230

Abbildung 5.2: Anzahl der Ecken und Dreiecke der Testmodelle

Zuerst wurde der Voxelisierungs-Algorithmus getestet. Dabei wurden die drei Modelle in verschiedenen Auflösungen von 10, 25, 50 und 100 Voxel pro Achse konvertiert. Es wurde insgesamt der Durchschnitt von 10 Tests für jede Auflösung und Modell berechnet.

	Durschnittliche Zeit pro Auflösung (ms)			
Mesh	10	25	50	100
Bunny	555	2620	9266	36692
LegoMan	66	88	164	605
YoshiPirate	4	15	89	1006

Abbildung 5.3: Durchschnittliche Zeit (in Millisekunden) die benötigt wird das 3D-Modell in ein Voxelgitter umzuwandeln

Das Ergebnis aus Abbildung 5.3 zeigt, dass, mit dem Anstieg der Auflösung und der Anzahl der Dreiecke, die Berechnungszeit erheblich steigt. Für ein Modell mit knapp 70.000 Dreiecken wird für eine Auflösung von 50 Voxeln pro Achse (insgesamt 125.000 Voxel) unter 10 Sekunden Zeit benötigt.

Wie beim ersten Test wurde auch der Lego-Algorithmus mit den selben Modellen getestet. Wieder wurde der Durchschnitt von insgesamt 10 Tests für die Auflösungen 10, 25 und 50 Voxel pro Achse berechnet. Aus Zeitgründen, sowie der Möglichkeit eines StackOverflows bei zu geringem Arbeitsspeicher, wurde die Auflösung 100 weggelassen.

	Durchschnittswerte von 10 Tests			
Mesh (Auflösung)	Zeit (ms)	Steine	Komponenten	Gelenkpunkte
Bunny (10)	104	123.7	1.0	0.1
Bunny (25)	881	1844.9	1.0	0.3
Bunny (50)	8632	14745.8	1.0	0.8
LegoMan (10)	48	120.3	1.0	0.1
LegoMan (25)	855	1832.8	1.0	0.5
LegoMan (50)	8747	14784.6	1.0	0.7
YoshiPirate (10)	290	582.3	1.0	0.3
YoshiPirate (25)	3881	8110.0	1.0	0.5
YoshiPirate (50)	42286	65017.5	1.0	0.7

Abbildung 5.4: Durchschnittliche Werte der Konvertierung eines Voxelgitters in ein Baustein-Modell

Bei allen Tests, wie in der Abbildung 5.4 zu sehen, wurde es geschafft, dass das Modell aus nur einem Stück besteht und somit keine Teile abfallen würden. Auch die Anzahl der schwachen Gelenkpunkte liegt im Durchschnitt zwischen Null und Eins. Somit sollte das Modell stabil sein und könnte mit zum Beispiel Lego-Steinen problemlos nachgebaut werden. Die Zeiten der Konvertierung sind ebenfalls im Rahmen. Die hohe Durchschnittszeit von 42 Sekunden beim YoshiPirate-Modell hängt vor Allem mit der hohen Anzahl an Steinen zusammen.

Mit den Werten aus dem „Automatic Generation of Constructable Brick Sculptures“ Paper (Testuz u. a. (2013)) kann man die eigenen Ergebnisse leider nicht vergleichen. Die Anzahl der Bausteine variiert zu stark zwischen den Werten dieser Arbeit und dem Paper. Es ist auch nicht auszumachen, ob die Werte mit oder ohne dem Aushöhlen vor dem Lego-Algorithmus zu Stande kommen.

6 Mögliche Erweiterungen

6.1 Besondere Bausteine

Mit der Klasse SpecialBrick besteht schon die Möglichkeit besondere Bausteine in ein Modell hinzuzufügen. Allerdings ist die derzeitige Implementierung relativ simpel gehalten. So werden diese vom Lego-Algorithmus einfach komplett ignoriert. Außerdem gibt es keine Überprüfung von möglichen Überschneidungen, wenn zum Beispiel der SpecialBrick 3 Mal so hoch wie ein RootBrick ist.

Darüber hinaus könnten diese besonderen Steine Gelenke oder ähnliche bewegliche Teile darstellen, die dann, mit erhöhtem Aufwand in der Visualisierung, Animationen am Modell erzeugen könnten.

6.2 Aushöhlung des fertigen Modells

Wie im „Automatic Generation of Constructable Brick Sculptures“ Paper (Testuz u. a. (2013)) vorgeschlagen kann nach dem Lego-Algorithmus das Modell noch einmal ausgehöhlt werden um Bausteine einzusparen. Im Gegensatz zum Aushöhlen vor dem Algorithmus müsste jedes einzelne Entfernen eines Steines überprüft werden. Dabei dürfte die Anzahl der Zusammenhangskomponenten und die der schwachen Gelenkpunkte nicht erhöht werden. Erst dann kann ein Stein sicher entfernt werden ohne das Modell zu beeinflussen.

6.3 Bausteintyp Limitierungen

Um gewisse Bausteintypen zu limitieren wird im bereits erwähntem Paper (Testuz u. a. (2013)) vorgeschlagen Bausteine nach dem Durchführen des Lego-Algorithmus umzuwandeln. Würde dies während des Algorithmus geschehen, könnte die Stabilität des Modells beeinträchtigt werden, weswegen ein ähnlicher Prozess wie bei der Aushöhlung verwendet werden könnte. Bausteine, dessen Typ über der Limitierung, ist müssten beim Umwandeln in andere Steine auf eine stabile Anzahl von Zusammenhangskomponenten und Gelenkpunkten überprüft werden. Falls dem nicht so ist, würde der Prozess einfach mit dem nächsten Baustein weiter arbeiten.

6.4 Multithreading

Im Voxelisierungs-Algorithmus werden bereits mehrere Threads zur Berechnung des Voxelgitters benutzt, jedoch nicht im Lego-Algorithmus. Hier könnte eine Möglichkeit gefunden werden gewisse Abläufe zu parallelisieren um die benötigte Zeit weiter zu optimieren. Eine Schwierigkeit besteht darin, dass Threads durch das Bearbeiten von angrenzenden Ebenen sich nicht gegenseitig bei der Kombinierung von Bausteinen beeinflussen.

6.5 Schwerpunktberechnung

Das „Automatic Generation of Constructable Brick Sculptures“ Paper (Testuz u. a. (2013)) schlägt außerdem vor, dass das Gewicht jedes einzelnen Steines verrechnet werden könnte umso den Massenschwerpunkt des Modells zu errechnen. Sollte der Schwerpunkt dabei zu einer Seite abweichen könnten Steine, die vorher durch das Aushöhlen entfernt wurden, wieder hinzugefügt werden. Mit diesem Prozess könnte gewährleistet werden, dass das Modell stehen bleibt und nicht auf eine Seite kippt.

7 Zusammenfassung

Mit der vorgestellten Technik lassen sich Prototypen kostengünstig und schnell herstellen. Damit gibt es eine Alternative zu den heute sehr häufig eingesetzten 3D-Druckern. Mit der möglichen Zeitersparnis beim Produzieren der Prototypen kann die Anzahl der Iterationen während der Entwicklung erhöht werden. Dies ist vor allem bei Verfahren wie dem Rapid Prototyping interessant.

Ziel dieser Arbeit war es ein Softwaremodul zu entwickeln, welches ein 3D-Modell in einen aus Bausteinen bestehenden Prototypen konvertiert. Dabei sollte auch die Möglichkeit bestehen, dass die Textur eines Modells mit in die Farben der Bausteine einfließt.

Mit den vorgestellten Algorithmen konnten diese Ziele erreicht werden und für das CgResearch Framework eine Software entwickelt werden. Mit den Möglichkeiten zum Speichern und Laden der Zwischen- und Endergebnisse kann auch unabhängig von diesem Projekt an diesen gearbeitet werden. Wie im Kapitel 6 bereits aufgezeigt, gibt es noch mögliche Erweiterungen für die Weiterentwicklung.

Literaturverzeichnis

[Gower u. a. 1998] Gower, R. ; Heydtmann, A. ; Petersen, H.: LEGO: Automated Model Construction. In: *European Study Group with Industry. Study Report* (1998), S. 81–94

[Möller und Trumbore 1997] Möller, T ; Trumbore, B.: Fast, Minimum Storage Ray/Triangle Intersection. In: *Journal of graphics tools 2.1* (1997), S. 21–28

[Mueller u. a. 2014] Mueller, S. ; Mohr, T. ; Guenther, K. ; Frohnhofen, J. ; Baudisch, P.: faBrickation: Fast 3D Printing of Functional Objects by Integrating Construction Kit Building Blocks. In: *CHI 2014* (2014), S. 3827–3834

[Nooruddin und Turk 2003] Nooruddin, F. ; Turk, G.: Simplification and repair of polygonal models using volumetric techniques. In: *Visualization and Computer Graphics, IEEE Transactions, Vol. 9, No. 2* (2003), S. 191–205

[Petrovic 2001] Petrovic, P.: Solving the LEGO brick layout problem using evolutionary algorithms. In: *Tech. rep., Norwegian University of Science and Technology* (2001)

[Testuz u. a. 2013] Testuz, R. ; Schwartzburg, Y. ; Pauly, M.: Automatic Generation of Constructable Brick Sculptures. In: *Eurographics 2013* (2013), S. 81–84

[Van Zijl und Smal 2008] Van Zijl, L. ; Smal, E.: Cellular automata with cell clustering. In: *Automata 2008* (2008), S. 425–441

[Winkler 2005] Winkler, D.: Automated brick layout. In: *BrickFest* (2005)

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 12. Dezember 2014 Chris Michael Marquardt