HAW
HAMBURG

BACHELORTHESIS

Christian Dorn

Constructive Solid Geometry mit einer blockbasierten Benutzeroberfläche

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Christian Dorn

Constructive Solid Geometry mit einer blockbasierten Benutzeroberfläche

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Technische Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Philipp Jenke
Zweitgutachter: Prof. Dr. Andreas Meisel

Eingereicht am: 22. März 2021

Christian Dorn

Thema der Arbeit

Constructive Solid Geometry mit einer blockbasierten Benutzeroberfläche

Stichworte

CSG, constructive, solid, geometry, gui, block, benutzeroberfläche, swing, jMonkey

Kurzzusammenfassung

Visual Programming ermöglicht es, die Interaktion mit Benutzeroberflächen einfach und intuitiv zu halten. Es lassen sich Konzepte durch Blöcke abstrahieren, sodass ein Nutzer kein tieferes Verständnis für die Software benötigt und effizient arbeiten kann. Insbesondere Modellierungswerkzeuge könnten von einer solchen Vereinfachung profitieren, da diese häufig sehr komplex sind. Es gilt daher, die Möglichkeit zu erkunden, die Modellierungstechnik Constructive Solid Geometry mit Visual Programming zu kombinieren und dadurch ein einfach zu bedienendes Modellierungswerkzeug zu implementieren.

Christian Dorn

Title of Thesis

Constructive solid geometry with a block-based user interface

Keywords

CSG, constructive, solid, geometry, gui, block, ui, user, interface, swing, jMonkey

Abstract

Visual Programming enables the possibility to simplify interactions with a user interface by keeping it intuitive and easy to use. By using block-like abstractions the user does not need to have a deep understanding of the application helping him to work efficently. Especially modeling tools can profit from such a simplification as they can get quite complex in usage. Therefore the possibility to combine visual programming and the modeling technique Constructive Solid Geometry should be explored and a proof of concept modeling tool implemented.

Inhaltsverzeichnis

Abbildungsverzeichnis **viii**

Abkürzungen **x**

1 Einleitung **1**

1.1 Motivation 1

1.2 Zielsetzung 2

1.3 Aufbau der Arbeit 2

2 Grundlagen und Stand der Technik **4**

2.1 Constructive Solid Geometry 4

2.1.1 Alternative Repräsentationen 5

2.1.2 Vorteile von CSG 6

2.1.3 Nachteile von CSG 7

2.1.4 Anwendungsbereiche 8

2.1.5 CSG Darstellung 8

2.1.6 Stand der Technik 10

2.2 Visual Programming 12

2.2.1 Beispiele VPLs 12

2.2.2 Stand der Technik 15

2.3 3D-Modell Speicherungsformat 17

2.3.1 Wavefront OBJ 17

3 Analyse **20**

3.1 Requirements 20

3.2 Softwarestack 21

3.2.1 3D Darstellung 21

3.2.2 Blockbasierte Benutzeroberfläche 22

4 Konzept **25**
4.1 Blocktypen 25
4.2 Design der Benutzeroberfläche 26
4.3 Software-Architektur 27
4.3.1 Architektur 27
4.4 Erstellung einer eigenen, blockbasierten Oberfläche 28
4.4.1 Zeichnen eines Blocks 28
4.4.2 Organisation von Blöcken 29
4.4.3 Andocken 29
4.4.4 Gleichzeitiges Bewegen 30
4.4.5 Dynamisches Wachsen 30
4.5 Baum-Datenstruktur 31
4.6 3D-Darstellung 31
4.6.1 Bedingungen zur Darstellung 32
4.6.2 Hervorheben der aktuellen Selektion 32
4.6.3 Veränderung der Maße, Translation und Rotation der 3D-Modelle . 33
4.6.4 Kamerasteuerung 34
4.7 CSG-Generierung 35

5 Implementierung **37**
5.1 Software Engineering 37
5.1.1 Komponenten 37
5.1.2 Dependency Injection Entwurfsmuster 38
5.1.3 Single-Responsibility-Prinzip 38
5.1.4 Sequenzen 39
5.2 Benutzeroberfläche 41
5.3 Hindernisse 42
5.3.1 Rechteckige Bounding Boxen in Swing 42
5.3.2 Leere Meshes in jMonkey 43
5.3.3 Versionen höher als Java 8 44
5.3.4 Transfer von Blöcken zwischen Containern 44
5.4 Werkzeuge 45
5.4.1 Build-System Gradle 45
5.4.2 Versionskontrolle git 45
5.4.3 Swing Explorer 45

6 Evaluation **46**
6.1 Methodik 46
6.1.1 Probandentests mit Umfrage 46
6.1.2 Heuristiken von Kölling und McKay 47
6.2 Auswertung 48
6.2.1 Interaktion 48
6.2.2 Modellierung 51

7 Fazit **52**
7.1 Weiterentwicklung 53

Literaturverzeichnis **54**

Selbstständigkeitserklärung **57**

Abbildungsverzeichnis

2.1 Ein einfacher CSG-Baum. 4
2.2 Zwei CSG-Objekte, dessen Roth Diagramm mit Mengenoperationen. (Angelehnt an Roth [18]) 9
2.3 Bild von CSG-Modell eines Helikopters, generiert auf einem CDC 6600. Quelle: [5]. 10
2.4 Einfaches *Hello World* in Scratch 3.0 13
2.5 Node Editor von Blender - Einfach Anordnung für ein Ziegelsteinmaterial 14
2.6 Ergebnis des Node Editors aus 2.5. 14
2.7 Einfaches NXT-G Programm - Roboter wartet eine Sekunde, dreht Motor B und C vier Umdrehungen. 15
2.8 Ergebnis der Frage, ob Schüler einen weiteren Informatikkurs belegen würden. Quelle: [21] 16
2.9 Beispielhaftes Objekt, erstellt mit den Parametern aus Abschnitt 2.3.1 und in Blender importiert. 19

4.1 GUI-Mockup 26
4.2 Datenfluss des Prototypen 27
4.3 Blöcke mit Konnektoren erstellt in Swing. 28
4.4 Andocken von Blöcken. 29
4.5 Sich überlappende Blöcke, wenn Blockgröße sich nicht ändert. 30
4.6 In die Breite wachsende Blöcke. 31
4.7 Zwei Modelle: Der Zylinder ist selektiert und opaque, während der Würfel transparent ist. 32
4.8 Transformationsmanipulatoren in Blender. 34

5.1 Komponentendiagramm der Anwendung. 37
5.2 Klassendiagramm einiger Event-Handler. 39
5.3 Sequenz vom Hinzufügen eines Blockes. 39
5.4 Sequenz vom Verbinden zweier Blöcke. 40

5.5 GUI des Prototypen mit dem gebauten Beispiel aus Abbildung 2.1. 41
5.6 Konfigurationsfenster des Prototypen für einen Zylinder. 42
5.7 Blöcke mit eingezeichneten Bounding-Boxen. Ein Operatorblock verdeckt einen Primitivblock 42

Abkürzungen

BSP Binary Space Partition.

CAD Computer Aided Design.

CGI Computer Generated Imagery.

CSG Constructive Solid Geometry.

GUI Graphical User Interface.

JLWGL Java Lightweight Game Library.

JME jMonkey-Engine.

JOGL Java OpenGL.

MVC Model-View-Controller.

VPL Visual Programming Language.

1 Einleitung

1.1 Motivation

Im Laufe der Geschichte der Informatik hat diese in vielen Fachbereichen massiv an Präsenz gewonnen und dadurch mit vielen neuen Möglichkeiten und Erweiterungen bereichert. Neue Werkzeuge erleichtern die Entwicklung von Produkten erheblich und bilden sogar manchmal neue Branchen und Berufe. Dieses Wachstum hat auch zur Folge, dass die benutzten Entwicklungswerkzeuge zunehmend komplexer werden und es Neueinsteigern schwierig machen, sich in einen Bereich einzuarbeiten. Es muss erst genügend Erfahrung gesammelt werden, um effizient arbeiten zu können.

Um dagegen anzukommen, werden interaktive Lernplattformen oder spezielle Benutzeroberflächen genutzt, in denen durch Abstraktion Konzepte vereinfacht werden. Diese Abstraktionen äußern sich meistens durch vereinfachte Bedienelemente, mit denen Nutzer durch simple Interaktionen, wie dem logischen Anordnen von Komponenten, komplexe Konstrukte bauen können und dadurch ohne tieferes Verständnis der Thematik zu einem Ergebnis kommen. Diese Idee des Editierens durch Anordnung von grafischen Elementen wird als *Visual Programming* bezeichnet.

Insbesondere Modellierungwerkzeuge für das Erstellen von Objekten in der Computergrafik sind häufig von komplexen Benutzeroberflächen geprägt und machen den Prozess des Modellierens oft langwierig und aufwändig. Diese Werkzeuge sind zwar sehr umfangreich und mächtig, bieten allerdings alles andere als einen schnellen Einstieg und selbst für einfache Modelle müssen unerfahrene Nutzer sich lange einarbeiten [10].

Der Bereich der Modellierung ist sehr weitreichend, vom *Computer Aided Design* (CAD) in der Fertigungsindustrie, über Simulationen bis hin zu *Computer Generated Imagery* (CGI) in Film und Fernsehen. Es ist daher wünschenswert, eine Möglichkeit zu finden, einen einfachen Einstieg für diese Werkzeuge zu bieten und den bestehenden Prozess zu vereinfachen.

1.2 Zielsetzung

Ziel dieser Arbeit ist es, eine Applikation zu entwickeln, welches es ermöglicht 3D-Modelle mit dem Konzept von Visual Programming zu erstellen. Es soll eine blockbasierte Oberfläche entwickelt werden, um die Komplexität aus der Modellierung zu entfernen und mit simplen Interaktionen einfache Modelle zu erstellen. Dabei liegt Hauptaugenmerk auf der Benutzerfreundlichkeit. Es soll einer Person möglich sein, ohne jegliche Vorkenntnisse modellieren zu können.

Da die Modellierung und Darstellung von dreidimensionalen Objekten sehr umfangreich ist, wird sich im Rahmen der Arbeit nur auf die Modellierung mithilfe von *Constructive Solid Geometry* (CSG) konzentriert.

1.3 Aufbau der Arbeit

Die Arbeit besteht aus den folgenden 7 Kapiteln:

Einleitung

Grundlagen + Stand der Technik Es werden die Grundlagen von Constructive Solid Geometry erläutert und mit anderen Darstellungsformen verglichen. Ebenfalls wird das Konzept von Visual Programming erläutert und anhand von Beispielen gezeigt. Für beide Themen wird der Stand der Technik präsentiert.

Analyse Es werden Anforderungen für die zu entwickelnde Applikation bestimmt und anhand der Ergebnisse passende Technologien ausgesucht. Diese werden untereinander verglichen und letztendlich die am geeignetsten ausgewählt.

Konzept Basierend auf dem vorangegangenen Kapitel werden Designentscheidungen für die zu entwickelnde Applikation getroffen und bestimmte Funktionen genauer erläutert. Zusätzlich wird eine passende Architektur gewählt.

Implementierung Hier wird das fertige Endprodukt präsentiert. Es wird auf Details der Architektur und die Kommunikation der einzelnen Komponenten eingegangen. Benutzte Werkzeuge und während der Entwicklung entstandene Probleme werden beschrieben.

Evaluation Es wird ein passendes Schema zur Evaluation der entwickelten Software gewählt und schließlich der Prototyp anhand dessen bewertet.

Fazit Zusammenfassung aller Ergebnisse und ein Ausblick auf mögliche Erweiterungen.

2 Grundlagen und Stand der Technik

In diesem Kapitel werden die Grundlagen für die Thesis dargelegt. Es wird die Bedeutung und Funktionweise von Constructive Solid Geometry erläutert und auf das Konzept von Visual Programming eingegangen und Beispiele gezeigt. Zuletzt wird das Speicherformat .obj präsentiert.

2.1 Constructive Solid Geometry

Constructive Solid Geometry ist eine Repräsentationsform und Modellierungstechnik für Festkörper in der Computergrafik, bei der durch geschickte Kombination von sogenannten *Primitiven* (Körper wie Würfel, Kugeln und Zylinder) komplexe Volumenmodelle erstellt werden können. Die Primitive werden durch Mengenoperationen (Union, Differenz und Schnittmenge) miteinander verknüpft und bilden dadurch neue Körper, welche wiederum miteinander kombiniert werden können [6].

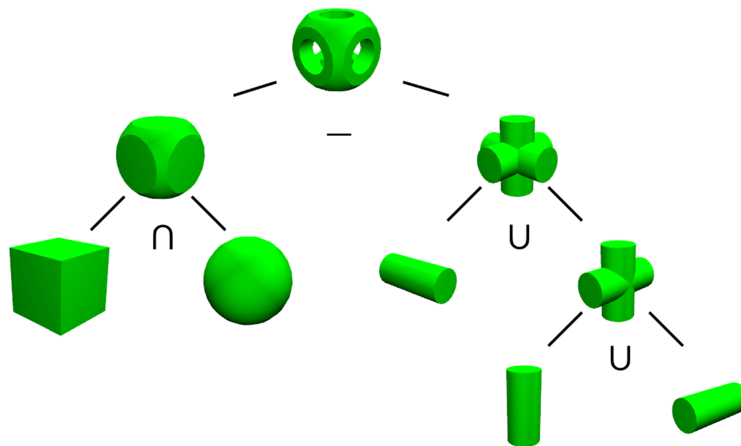


Abbildung 2.1: Ein einfacher CSG-Baum.

Die Verkettung der Primitive bilden dann eine Baumstruktur, wie sie in Abbildung 2.1 zu sehen ist. Es kann also beispielsweise ein Würfel und eine Kugel der gleichen Größe als Primitiv genommen und mit einer Schnittmenge verknüpft werden. Das gemeinsame überlappende Volumen bildet dann die Schnittmenge und resultiert in einem neuen Würfel mit abgerundeten Ecken, ähnlich wie bei einem Spielwürfel. Dieses Beispiel ist im linken Teilbaum von Abbildung 2.1 erkennbar. Dieser neu entstandene Würfel wird nun weiter verknüpft, um ein noch komplexeres Modell zu erstellen.

2.1.1 Alternative Repräsentationen

Constructive Solid Geometry ist nur eine mögliche Modellierungsform von vielen. Andere Formen haben andere Eigenschaften und jeweilige Vor- und Nachteile.

Boundary Representation

Als Alternative Repräsentation gibt es die *Boundary Representation* (B-Rep). Anders als bei CSG wird ein Modell nicht durch Primitive beschrieben, sondern besteht aus einer topologischen Beschreibung in Form von zusammenhängenden Punkten (Polygonen) und einer geometrischen Beschreibung zwischen diesen, bestehend aus mathematischen Gleichungen für Kurven und Flächen. Die Vertices der Polygonen beschreiben die Eckpunkte einer Kante, oder Fläche und die geometrische Beschreibung die Form zwischen den Punkten. Durch diese Beschreibungen ist es möglich, sehr akkurat und mit hoher Präzision Kurven und gewölbte Flächen zu beschreiben [6].

Polygonale Repräsentation

Modelle können auch als Polygonnetze repräsentiert werden. Ein Polygonnetz besteht aus vielen einzelnen Knoten (Vertices), welche über gerade Kanten miteinander verbunden sind. Diese Sammlung aus Knoten und Kanten bilden wiederum eine Fläche (Polygon), welche an vielen weiteren Flächen angrenzen und einen Polyeder bilden. Diese Netze können beliebig groß sein. Durch eine Erhöhung der Anzahl von Polygonen lässt sich ein höherer Detailgrad erzielen, mit entsprechend erhöhter Rechenzeit von Algorithmen und Darstellung. Allerdings ist es gerade bei gleichmäßig konkaven oder konvexen Flächen unmöglich an die Präzision von B-Rep oder CSG heranzukommen, da Polygonnetze ohne hohe Polygonanzahl keine perfekt runden Flächen darstellen können.

Dennoch werden meistens zum Darstellen von B-Reps und CSG-Modellen Polygonnetze genutzt. Moderne Grafikkartenarchitektur ist auf das Darstellen von Polygonnetzen optimiert und ermöglicht es, sehr schnell Operationen mit diesen durchzuführen. Daher werden B-Reps und CSG-Modelle für die Darstellung innerhalb von Programmen in angenäherte Polygonnetze konvertiert, um dem Nutzer ein Bild des Modells zu verschaffen, während intern weiterhin mit anderen Repräsentationen gerechnet wird [17].

Es ist wichtig in diesem Kontext zu erwähnen, dass nicht jede Anwendung mit CSG Unterstützung auch intern eine CSG Repräsentation nutzt. In vielen Fällen werden CSG Algorithmen auf polygonaler Basis genutzt, die direkt die CSG Operationen auf Polygonnetzen durchführen.

2.1.2 Vorteile von CSG

Nachvollziehbarkeit

Wird ein Modell mithilfe von CSG erstellt, so liegt am Ende ein CSG-Baum vor. Dieser Baum beschreibt jeden Schritt, um von allen beinhalteten Primitiven zum Endergebnis zu kommen. Wird die Baum-Datenstruktur geeignet visualisiert, so lässt sich mit Leichtigkeit jeder einzelne Schritt nachvollziehen. Auch können Teile des Baumes verändert, entfernt oder erweitert werden, ohne dabei die anderen Komponenten zu beeinflussen, da jedes Primitiv weiterhin in seiner Ursprungsform im Baum existiert.

Bei anderen Repräsentationsformen ist die Entstehungshistorie meistens nach Beendigung des Modellierungswerkzeuges verloren und kann nicht mehr eingesehen werden.

Wasserdichtigkeit

Wasserdichtigkeit beschreibt bei 3D-Modellen, ob diese ein endliches Volumen beschreiben, von einer Fläche vollständig umschlossen sind und damit Innen- und Außenraum voneinander getrennt sind. Da die in CSG genutzten Primitive alle wasserdicht sind, sind auch die Kombination aus diesen ebenfalls wasserdicht. Dadurch muss nicht nach jeder Operation überprüft werden, ob das erstellte Modell noch gültig ist. Wegen dieser klaren Definition des Volumens der Primitive ist es auch einfach zu bestimmen, ob sich ein Punkt innerhalb des Volumens befindet [6].

Anders als bei CSG ist es möglich, bei B-Rep und anderen Repräsentationsformen nicht wasserdichte Modelle zu erzeugen, etwa wie beim Weglassen einer Seite bei einem Würfel. Dies hat zur Folge, dass die Modelle nachträglich algorithmisch auf Wasserdichtigkeit geprüft werden müssen.

Kompakte Datenstruktur

Da ein CSG-Modell nur aus Positions-, Rotations- und Skalierungsdaten von Primitiven und dessen Relation zueinander besteht, ist dieses sehr speichereffizient.

B-Rep und Polygonnetze verbrauchen dabei deutlich mehr Speicher, da diese jedes Polygon und damit deren Eckpunkte und dessen Zusammengehörigkeit untereinander speichern müssen. Eine einfache Kugel kann in CSG-Repräsentation nur aus einer Position und einem Radius bestehen, während bei einem Polygonnetz tausende Polygonen gespeichert werden müssen. Je detaillierter das Modell ist, desto größer ist der Speicherverbrauch.

2.1.3 Nachteile von CSG

Bearbeiten von Kanten

CSG ist zwar leicht zu verwenden, kommt jedoch auch mit Einschränkungen. Es ist unter anderem nicht möglich Ecken und Kanten separat zu bearbeiten. Stattdessen muss versucht werden, durch geschicktes Kombinieren der Primitive eine Alternative zu finden.

Da dies nicht immer möglich ist, werden andere Repräsentationen benutzt, oder hybride entwickelt, die CSG gleichzeitig mit anderen Repräsentationen unterstützen.

Wiederholtes Auswerten des CSG-Baumes

Da ein CSG-Baum nur aus durch Mengenoperationen kombinierte Primitiven besteht, lässt sich ohne weitere Auswertung des Baumes keine Operation mit einem CSG-Modell durchführen. Es muss für jede Operation der gesamte Baum verarbeitet werden, um geometrische Operationen mit diesem durchführen zu können. Dies kann besonders bei großen Bäumen zu großen Leistungsproblemen führen.

Inverses CSG

Es können CSG-Modelle in andere Repräsentationsformen konvertiert werden, allerdings ist das umgekehrte Verfahren deutlich komplexer. Eine direkte Konvertierung ist nicht möglich. Stattdessen muss eine Annäherung gefunden werden, bei der eine Kombination von Primitiven möglichst nah an das originale Modell herankommt. Es ist zwar möglich eine Lösung zu finden, allerdings ist dies eine mögliche Lösung von unendlich vielen und meistens nicht der kleinstmögliche Baum.

2.1.4 Anwendungsbereiche

Die vielen Vorteile von CSG machen es sehr attraktiv für das Computer Aided Design (CAD). Insbesondere die Eigenschaft der Wasserdichtigkeit ist im CAD begehrenswert, da reale Objekte für die Produktion und Herstellung modelliert werden und dessen Modelle möglichst akkurat sein sollen. Die meisten gängigen CAD-Programme unterstützen daher auch CSG Operationen. Allerdings hat auch B-Rep viele Vorteile, besonders die exakte mathematische Beschreibung ist gerade beim CAD sehr wichtig. Daher werden beide Repräsentationen miteinander gemischt um die jeweiligen Nachteile auszugleichen und Vorteile beider weiterhin nutzen zu können. Solche Programme werden als *dual-representation Modeler* bezeichnet [6].

Auch in der Spieleentwicklung wird CSG oft benutzt. Hier hat insbesondere die einfache Nutzung der CSG Operationen den Vorteil, dass nicht nur Designer, sondern auch Entwickler schnell prototypische Umgebungen und Objekte erstellen können. Die Open-Source Game Engine *Godot*[1] hat CSG implementiert, um es Entwicklern leichter zu machen, die keine Erfahrung in der Modellierung haben [11]. Auch Karteneditoren wie *Unreal Editor* der *Unreal Engine*[2] [3], oder Valve's *Hammer Editor* für die Goldeye- und Source Engine, unterstützen CSG-Operationen.

2.1.5 CSG Darstellung

Soll ein CSG Modell, ohne Konvertierung in ein Polygonnetz, dargestellt werden, so kann ein Raytracing Algorithmus genutzt werden. Wie bereits in Abschnitt 2.1.2 beschrieben, hat CSG den großen Vorteil, mit Leichtigkeit bestimmen zu können, ob sich ein Punkt

[1] https://godotengine.org/
[2] https://www.unrealengine.com/en-US/

im Volumen des Körpers befindet oder nicht. Da bei Raytracing die Bewegung von Lichtstrahlen und Kollision mit Objekten simuliert wird, eignet sich es hervorragend, um ein CSG Modell darzustellen [4].

Beim Raytracing müssen die Schnittpunkte der Strahlen bei einem Festkörper bestimmt werden. Darstellbar sind diese Schnittpunkte mithilfe eines *Roth Diagramms* [4],[18].

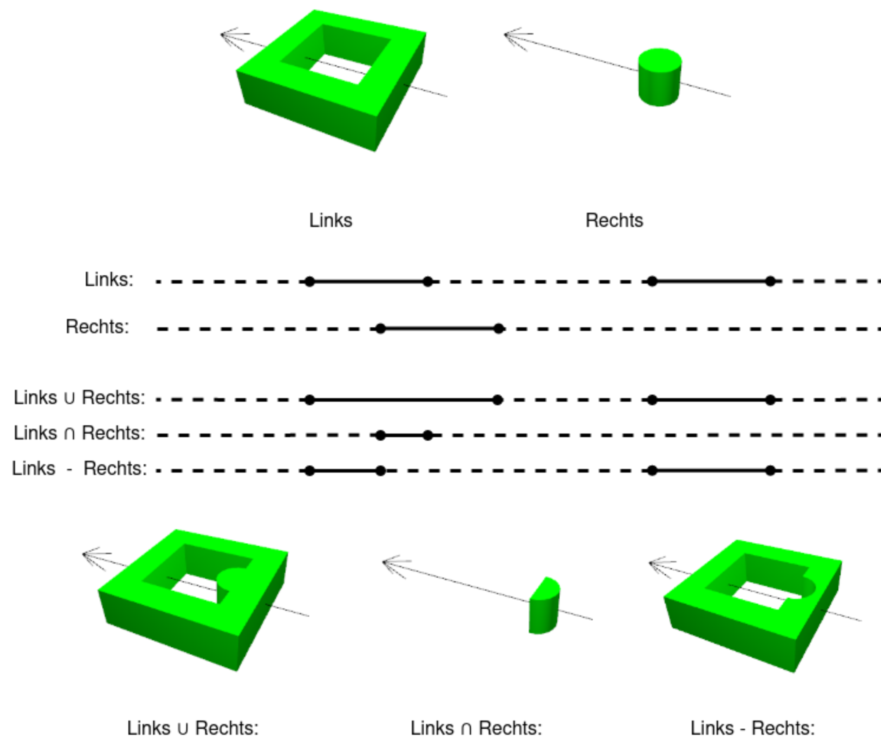


Abbildung 2.2: Zwei CSG-Objekte, dessen Roth Diagramm mit Mengenoperationen. (Angelehnt an Roth [18])

Abbildung 2.2 zeigt Roth Diagramme von zwei CSG-Objekten und dessen jeweiligen Ergebnisse nach den Mengenoperationen. Ein Roth Diagramm zeigt einen Strahl, welcher jeden Wert des Parameters t innerhalb einer parametrisierten Geradengleichung darstellt (Die Geradengleichung ist hierbei der Raytracing Lichtstrahl). Dieser gibt die Intervalle an, in welchen sich der Raytracing-Strahl innerhalb eines Körpers befindet. Der Beginn und das Ende des jeweiligen Intervalls gibt dabei den Schnittpunkt mit dem Körper an.

Um das komplette Intervall eines zusammengesetzten CSG-Objektes zu erhalten, muss für jedes Primitiv das Schnittintervall bestimmt werden. Die berechneten Intervalle werden dann, wie im zugehörigen CSG-Baum, mit Mengenoperationen verknüpft.

Als Resultat ist nur noch ein Intervall übrig, welches den Einfalls- und Austrittspunkt des Raytracing Strahles für das CSG-Objekt angibt. Dies muss für alle Strahlen des Raytracing Algorithmus wiederholt und entsprechend für die Darstellung verarbeitet werden.

2.1.6 Stand der Technik

Erste Entwicklung von CSG

CSG hat eine lange Geschichte in der Informatik. Die ersten Anfänge von CSG lassen sich in der Arbeit von Goldstein und Nagel in [5] erkennen. Hier noch als *Combinatorial Geometry* bekannt, haben Goldstein und Nagel mithilfe von einfachem Raytracing Bilder von CSG-Objekten, wie in Abbildung 2.3 zu sehen ist, generiert.

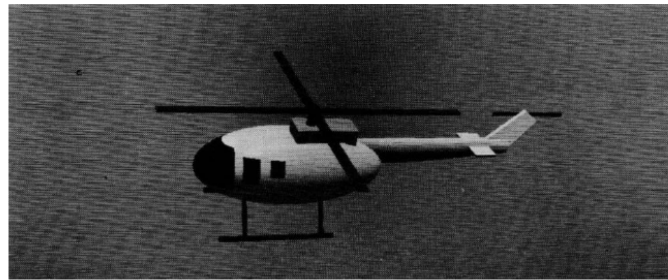


Abbildung 2.3: Bild von CSG-Modell eines Helikopters, generiert auf einem CDC 6600. Quelle: [5].

CSG auf Basis von Polygonen

Im Laufe der Jahre wurde diese Grundidee weiter ausgearbeitet und optimiert, wie beispielsweise durch Laidlaw et al. Implementierung aus [9]. Hier wird kein Raytracing Algorithmus benutzt, sondern es werden CSG Operationen direkt auf Polyedern durchgeführt und diese mithilfe eines Renderers dargestellt.

Der Algorithmus iteriert über alle Polygonen beider miteinander zu verknüpfenden Netze und unterteilt diese in kleinere Polygonen, wenn sie sich mit einem weiteren Polygon des anderen Objektes schneiden. Im nächsten Schritt werden alle Polygone beider Netze klassifiziert, ob diese innerhalb, auf der Kante oder außerhalb des jeweiligen Netzes liegen. Nach der Klassifizierung können dann, je nach Mengenoperation der beiden Objekte, die entsprechenden Polygonen gelöscht oder beibehalten werden.

Konvertierung zu BSP

Weitere Optimierungen sind unter anderem Konvertierung von CSG-Objekten in andere Formate. Naylor und Thibault hat in [15] ein Konzept entwickelt, bei dem *Binary Space Partition* (BSP) zur Optimierung von Raytracing und CSG-Objekten verwendet werden kann.

BSP-Bäume bieten eine mögliche Lösung für das Sichtbarkeitsproblem in der Computergrafik. Es wird ein Raum über Hyperebenen in kleinere Unterräume unterteilt. Diese Unterteilung wird in einem Binärbaum gespeichert. Jeder Knoten stellt dabei eine Hyperebene dar und die linken und rechten Kinder jeweils die Objekte auf den jeweiligen Seiten der Hyperebene. Somit wird eine räumliche Unterteilung durchgeführt, mit welcher es durch traversieren des Baumes möglich ist, alle sichtbaren Objekte in der korrekten Reihenfolge darzustellen.

Naylor und Thibault präsentieren einen Algorithmus, der ein beliebiges CSG-Modell in eine äquivalenten BSP-Baum konvertiert. Hierfür wird von dem zu konvertierenden CSG-Modell nach einer Heuristik eine Facette eines Primitives ausgewählt. Auf dieser Fläche der Facette wird eine Hyperebene gespannt, die das CSG-Modell in zwei Hälften teilt. Diese erste Hyperebene bildet die Wurzel des neuen BSP-Baumes. Als nächstes wird der CSG-Baum mithilfe der Hyperebene in zwei Teile getrennt, sodass zwei neue CSG-Bäume entstehen, dessen Inhalte sich jeweils im linken und rechten Unterraum der Hyperebene befinden. Diese Schritte werden wiederholt für alle Facetten durchgeführt.

Geschwindigkeits- und Genaugkeitsverbesserungen

Auch heute wird noch an Optimierungen gearbeitet. Lu et al. haben in ihrer Arbeit [12] ebenfalls CSG Operationen auf Basis von Polyedern implementiert, allerdings mit einem Fokus auf Geschwindigkeit und Genauigkeit. Erzielt wurde dies mithilfe von einer Kombination aus *Octrees* und lokalisierten CSG-Bäumen. Ferner wurde durch die Integration von Ebenengeometrie eine höhere Präzision erzielt, da sich ansonsten relative Fehler aufsummieren.

2.2 Visual Programming

Anders als bei der normalen textbasierten Programmierung wird bei der visuellen Programmierung auf grafische Elemente gesetzt. Diese Elemente können einfache Blöcke, Puzzleteile oder auch Graphen mit Pfeilen sein, welche vom Nutzer beliebig verschoben und angeordnet werden können. Dadurch fallen Schwierigkeiten durch Syntax und teilweise auch Semantik weg, da die Elemente durch die vorgegebenen Formen nur in speziellen Weisen angeordnet werden können. Manche Elemente können auch zusätzlich konfiguriert oder untereinander verschachtelt werden. Dadurch können Systembeschreibungen, Abläufe und Algorithmen von Nutzern ohne Vorkenntnisse mit Leichtigkeit erstellt werden. Erstellte Konstrukte sind meistens sehr anschaulich und schnell zu verstehen. Der Fokus vom Visual Programming ist hierbei meistens auf Verständlichkeit gelegt, anstatt auf Umfang oder Komplexität [20].

Problematisch wird es erst bei größeren Projekten, da durch die hohe Anzahl von Komponenten die Übersicht fehlt und das Bearbeiten und Lesen zunehmend schwerer wird. Auch ist das Platzieren von vielen Blöcken sehr zeitaufwändig, da wiederholt ein Block in einem Menü gesucht, gezogen, platziert und eventuell mit Parametern konfiguriert werden muss [2].

2.2.1 Beispiele VPLs

Es folgen nun bekannte Beispiele von *Visual Programming Languages*

Lernplattform Scratch

Scratch[3] bietet Kindern, aber auch Erwachsenen, einen spielerischen, anschaulichen und erleichterten Einstieg in die Programmierwelt. In *Scratch* ist es möglich, einfach Programme zu erstellen und auszuführen. Hierfür werden einfache Puzzleteile, welche jeweils verschiedene Variablen, Kontrollstrukturen oder Funktionen repräsentieren, durch Klicken und Ziehen verschoben und aneinander gesteckt. Dabei ist jedes Teil speziell geformt, sodass nur bestimmte Teile zusammenpassen, wodurch syntaktische Fehler nicht auftreten können. Diese Teile bilden am Ende eine Reihe von Anweisungen, ähnlich wie beim textbasierten Programmieren [13].

[3] https://scratch.mit.edu/

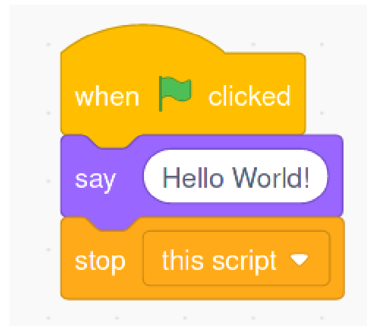


Abbildung 2.4: Einfaches *Hello World* in Scratch 3.0

In Abbildung 2.4 wurde ein einfaches *Hello World!* Beispiel erstellt. Beim Ausführen des Skripts (in Scratch durch Anklicken der Grünen Flagge) erscheint eine Sprechblase mit den Worten *Hello World!*.

Blender Node Editor

Die freie Modellierungs- und Animationssoftware *Blender*[4] hat einen speziellen knotenbasierten Editor. Dieser ermöglicht es dem Nutzer, Materialien, Texturen und Bild-Kompositionen zu erstellen. Dies erfolgt durch das geschickte Anordnen von Knotenelementen, welche jeweils einen Eingang und Ausgang besitzen. Knoten können über diese Schnittstellen miteinander verbunden werden und bilden dadurch eine logische Sequenz, die am Ende ein fertiges Ergebnis ausgibt. Durch dieses System können komplexe prozedural generierte Materialien und Texturen erstellt oder verbessert werden. [1]

[4]https://www.blender.org/

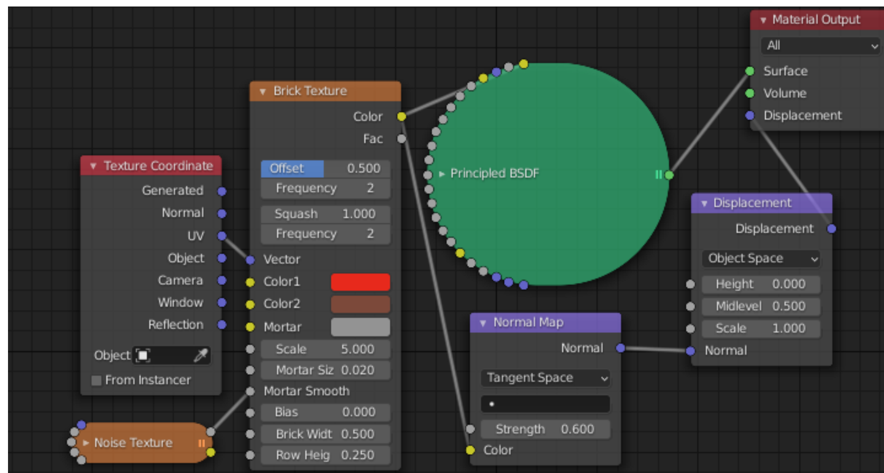


Abbildung 2.5: Node Editor von Blender - Einfach Anordnung für ein Ziegelsteinmaterial

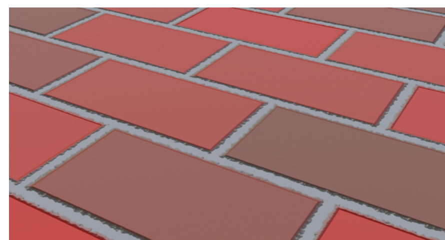


Abbildung 2.6: Ergebnis des Node Editors aus 2.5.

In Abbildung 2.5 wurde ein Material für eine Ziegelsteinwand erstellt. Hierfür wurde die fertige *Brick Texture* Vorlage von Blender mit dem standardmäßigen *Principled BSDF* Material verbunden und Werte sowie Farben angepasst. Um mehr Tiefe zu erzielen wurde noch Rausch-Knoten für den Mörtel eingebracht, damit dieser ungleichmäßiger auftritt. Zuletzt wurde noch ein Displacement Knoten eingehängt, welcher den Spalten zwischen Ziegelsteinen mehr Tiefe verleiht. Das Endergebnis der Knotensequenz ist in Abbildung 2.6 zu sehen.

LEGO™ Mindstorms NXT-G

Das Unternehmen *LEGO™* hat im Rahmen der Produktserie *LEGO™ Mindstorms* die Entwicklungsumgebung NXT-G veröffentlicht, welche es ermöglicht, den EV3-Steuerungscomputer zu programmieren und damit mit *LEGO™*gebaute Roboter zu steuern. Ähnlich wie bei *Scratch* können auch hier Blöcke durch Klicken und Ziehen miteinander verbunden wer-

den. Diese Blöcke können Bewegungsanweisungen, Sensormessungen oder programmiertypische Kontrollstrukturen darstellen. Dadurch können situationsbedingte Handlungsstränge entwickelt und anschließlich auf den Roboter übertragen werden [7].

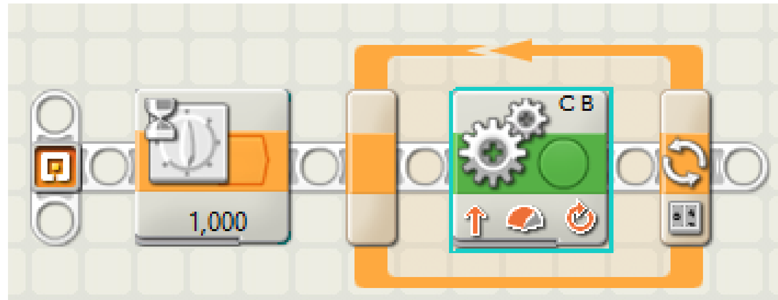


Abbildung 2.7: Einfaches NXT-G Programm - Roboter wartet eine Sekunde, dreht Motor B und C vier Umdrehungen.

Ein einfaches Beispielprogramm in NXT-G ist in Abbildung 2.7 zu sehen. Hier wurde eine Sequenz aus drei Bausteinen gebaut, welche den Roboter zuerst eine Sekunde warten lässt, dann eine Schleife von vier Durchläufen startet, in welcher die Motoren B und C jeweils eine Umdrehung vollführen.

2.2.2 Stand der Technik

Weintrop und Wilensky

In [21] haben Weintrop und Wilensky in einer Studie die Anwendungsmöglichkeiten von VPLs im Informatikunterricht in Schulen erkundet. Es wurde eine fünfwöchige Studie innerhalb zweier Schulklassen durchgeführt. Eine Klasse hat ihr Curriculum im Rahmen eines Programmierkurses für Einsteiger mit einer textbasierten Programmierumgebung durchgearbeitet, während die andere eine blockbasierte Umgebung benutzt hat. Die Schüler wurden jeweils vor und nach den fünf Wochen mit einem Fragebogen getestet. Zusätzlich wurden im Verlauf des Experiments mit den Schülern Interviews geführt und Anmerkungen sowie Fortschritte dokumentiert.

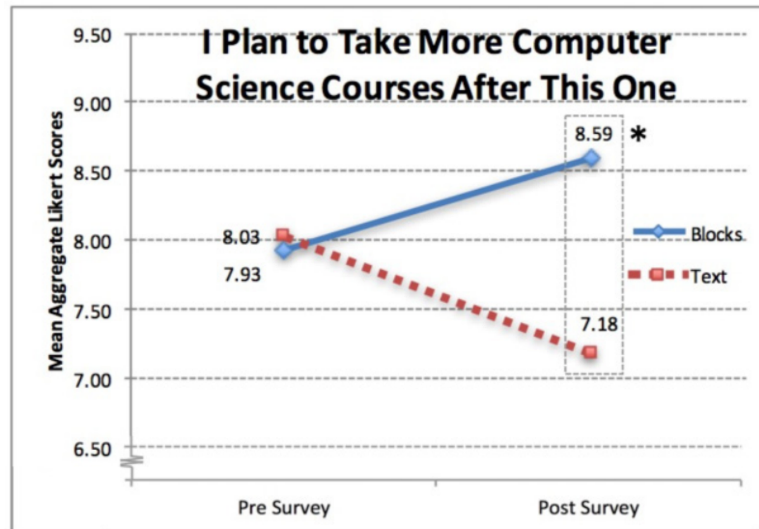


Abbildung 2.8: Ergebnis der Frage, ob Schüler einen weiteren Informatikkurs belegen würden. Quelle: [21]

Als Endergebnis hat die Klasse mit der blockbasierten Programmierumgebung minimal besser abgeschnitten als die textbasierte Klasse. Auch in Faktoren wie Zuversicht und Spaß am Programmieren hat die blockbasierte Klasse bessere Werte erzielt. Besonders deutlich lässt sich dies an Abbildung 2.8 erkennen. Hier wurden die Schüler gefragt, ob sie einen weiteren Informatikkurs belegen würden. Hier lässt sich eine positive Tendenz von Schülern mit weiterem Interesse an Informatikkursen erkennen, nachdem sie erste Schritte mit einer blockbasierten Programmierumgebung gemacht haben, während im Gegensatz dazu Schüler mit textbasierten Umgebungen eher Interesse verloren haben.

Zuvor haben Weintrop und Wilensky bereits in [20] Umfragen zum Thema blockbasierte Programmiersprachen in Schulklassen durchgeführt. Diese Klassen hatten als Einstieg *Snap!*[5] genutzt und sind nach fünf Wochen auf Java umgestiegen und wurden zu ihren Eindrücken befragt. Auch hier wurde die blockbasierte Benutzeroberfläche für Intuitivität und einfache Benutzbarkeit gelobt. Allerdings wurde auch bemängelt, dass die blockbasierte Oberfläche einschränkend wirkte und sich größere Projekte nur mit Mühe umsetzen ließen.

[5]https://snap.berkeley.edu/

Kölling und McKay

Anders als Weintrop und Wilensky setzen Kölling und McKay in [8] nicht auf eine Studie, sondern entwickeln für die Bewertung von blockbasierten Programmiersprachen Heuristiken. Diese heuristischen Kriterien sollen bestmöglich alle Bereiche einer blockbasierten Umgebung umfassen und bewertbar machen. Diese Heuristiken basieren auf der Arbeit von Nielsen und Molich, welche sich allgemein auf Benutzeroberflächen beziehen und auf blockbasierte Oberflächen angepasst und optimiert wurden. Getestet wurden die entwickelten Heuristiken mit drei Anfänger Programmierwerkzeugen von den Autoren selbst und ein weiteres Werkzeug von einer Gruppe Probanden. Dabei wurden die Heuristiken auf *thoroughness*, *validity* und *reliablity* (basierend auf der Arbeit von [19]) geprüft und ausgewertet. Verglichen mit den alten Heuristiken von Nielsen und Molich haben die neuen Heuristiken in der Problemidentifikation besser abgeschnitten und bieten das Potential für weitere Analysen verwendet zu werden. In der Evaluation in Kapitel 6 wird auf die Heuristiken noch einmal zurückgegriffen.

2.3 3D-Modell Speicherungsformat

Wird mithilfe von Werkzeugen etwas aufwändig erstellt, so besteht meistens der Wunsch den Inhalt zu speichern und woanders verwenden zu können. Es ist daher wünschenswert, im Prototypen eine Funktion zu haben, die es ermöglicht, erstellte Modelle exportieren und in anderen Werkzeugen weiterverwenden zu können. Hierfür wird konkret das Speicherungsformat *.obj* (Wavefront OBJ) betrachtet.

2.3.1 Wavefront OBJ

Das Wavefront OBJ Format wurde im Rahmen des Grafikpakets *The Advanced Visualizer* von *Wavefront Technologies* entwickelt und wird von den meisten 3D-Grafik-Applikationen unterstützt. Das Format ist in ASCII codiert und beschreibt einzelne geometrische Objekte [14]. Dies wird mithilfe der Beschreibung von Vertices und deren Relation zueinander realisiert. Auch sind Beschreibungen von Freiformflächen möglich, sind im Kontext aber nicht weiter relevant. Wegen der weiten Adaption in anderen Werkzeugen und der einfachen ASCII-Codierung bietet sich Wavefront OBJ am besten für den Export innerhalb der Applikation.

Um eine gültiges Modell zu beschreiben, werden drei verschiedene Beschreibungen benötigt.

Vertices

Es müssen alle Vertices des Models aufgelistet werden. Pro Vertex wird eine Zeile benötigt und wird jeweils mit dem Buchstaben *v* eingeleitet. Danach folgt die Koordinate des Vertex. Dies sieht in der Datei wie folgt aus:

```
v 1 0 -1
v 1 0 1
v -1 2 -1
v -1 0 -1
v -1 2 1
v -1 0 1
```

Normalen

Die Normalen sind nicht zwingend nötig, allerdings kann es ohne diese zu Problemen in der Beleuchtung von Objekten kommen. Hier handelt es sich um einfache Vektoren, die wie die Vertices einfach angegeben werden. Die Normalen werden mit dem Buchstaben *vn* eingeleitet:

```
vn 0 0 1
vn -1 0 0
vn 0 -1 0
vn 0 0 -1
vn 0.7071 0.7071 0
```

Facetten

Zuletzt müssen die einzelnen Vertices miteinander verknüpft werden. Dies folgt durch eine Auflistung der zusammengehörenden Vertices mit einem Index. Der Index bildet sich durch die Reihenfolge der Definition in der Datei. Zusätzlich muss der Index der zugehörigen Normalen mit angegeben werden. Die beiden Indizes werden über Schrägstriche

getrennt. Zwischen den Vertexindex und Normalenindex gehört der Texturekoordinatenindex. Dieser wird hier aber nicht benötigt und weggelassen. Die Zeilen mit Facetten werden mit einem f eingeleitet:

```
f 5//1 6//1 2//1
f 5//2 4//2 6//2
f 1//3 6//3 4//3
f 4//4 3//4 1//4
f 1//5 3//5 2//5
f 2//5 3//5 5//5
f 5//2 3//2 4//2
f 1//3 2//3 6//3
```

Resultat

Kombiniert man die zuvor genannten Beispiele, so ergibt sich das Modell in Abbildung 2.9, welches in Blender importiert wurde.

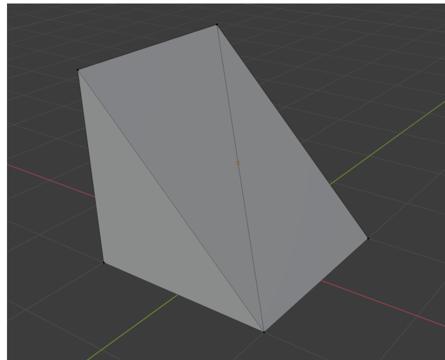


Abbildung 2.9: Beispielhaftes Objekt, erstellt mit den Parametern aus Abschnitt 2.3.1 und in Blender importiert.

3 Analyse

In diesem Kapitel werden Anforderungen an den zu entwickelnden Prototypen ermittelt und analysiert. Anhand dieser Ergebnisse werden mögliche Darstellungen und Grafikframeworks verglichen und letztendlich eines ausgewählt.

3.1 Requirements

Der Prototyp soll eine Beispielimplementierung sein und die Möglichkeit zeigen, eine blockbasierte Oberfläche mit der 3D Modellierung zu kombinieren. Es wird daher ein Minimalumfang der Software benötigt.

Klassische Modellierungswerkzeuge haben meistens eine 3D-Darstellung von dem Modell, was gerade bearbeitet wird. In dieser lässt sich die Kamera frei verschieben und rotieren, um das Modell aus allen Perspektiven betrachten zu können. Da der Prototyp ein vereinfachtes Modellierungswerkzeug sein soll, wird ebenfalls eine solche Ansicht benötigt.

Zusätzlich muss eine Fläche vorhanden sein, auf welcher mit Blöcken interagiert werden kann. Auf der Blockarbeitsfläche soll es möglich sein, Primitiv- und Operatorenblöcke durch Klicken und Ziehen zu platzieren, zu kombinieren und dessen Ergebnisse ebenfalls für weitere Operationen zu benutzen. Jeder Block, oder kombinierte Blöcke, soll durch ein Modell in der 3D-Ansicht parallel dargestellt werden und bei entsprechender Kombination miteinander ein CSG-Modell generieren.

Aus diesen Bedingungen lassen sich folgende Requirements ableiten:

Block-GUI

REQ01: Der Nutzer muss Blöcke durch Klicken und Ziehen verschieben können.

REQ02: Blöcke müssen sich durch Rein- und Rausziehen aus einem Bereich löschen und erstellen lassen.

REQ03: Blöcke müssen geeignet darstellen, inwiefern sie sich miteinander verbinden lassen.

REQ04: Blöcke müssen sich, wenn sie nah genug aneinander und kompatibel sind, miteinander wie Puzzleteile verbinden.

REQ05: Es muss Primitive-Blöcke und Operator-Blöcke geben, die sich jeweils miteinander verbinden lassen. Die Verbindung aus diesen muss sich wiederum mit anderen Operator-Blöcken verbinden lassen.

REQ06: In mehreren verbundenen Blöcken muss eine Baumstruktur erkennbar sein.

3D-Darstellung

REQ08: Alle momentan platzierten Blöcke und Blockverbindungen müssen in der 3D-Ansicht dargestellt werden.

REQ09: Verbundene Blöcke mit einem Operator müssen in der 3D-Ansicht ein CSG-Modell mit dem benutzen Operator generieren und darstellen.

REQ10: Die Kamera der 3D-Ansicht muss frei verschieb- und rotierbar sein, um die Modelle aus allen Perspektiven betrachten zu können.

3.2 Softwarestack

3.2.1 3D Darstellung

Um die repräsentativen Modelle der Blöcke geeignet darzustellen, wird ein Grafikframework benötigt. Dieses muss ermöglichen, 3D-Modelle bei Bedarf hinzuzufügen und wieder entfernen zu können, sowie mit der Kamera interagieren zu können.

jMonkey

jMonkey[1] ist eine Open-Source Game-Engine geschrieben in Java. Da es sich um eine fertige Game-Engine handelt, sind Komponenten wie Renderer und Szenengraph bereits implementiert und entsprechend abstrahiert, sodass sich nur mit Logik beschäftigt werden muss. Dadurch lässt sich mit Leichtigkeit mit Modellen, Kameras und Licht interagieren.

Da jMonkey Open-Source ist, gibt es von anderen Entwicklern Addons. Eins dieser Addons[2] ermöglicht es auf polygonaler Basis innerhalb von jMonkey CSG-Operationen durchzuführen.

libGDX

LibGDX ist ein Open-Source Java-Framework zur Entwicklung von Spielen. Anders als bei jMonkey handelt es sich hier nicht um eine fertige Game-Engine. Viele in jMonkey abstrahierte Konzepte müssen bei libGDX beachtet werden. Unter anderem wird sehr nah an OpenGL gearbeitet, wodurch sehr viel *Boilerplate code* entsteht. Andererseits ist dadurch eine feinere Kontrolle über einzelne Komponenten möglich. Ein weiterer Vorteil ist die Plattformunabhängigkeit. Es ist mit Leichtigkeit möglich ein Projekt auf Windows, Linux, Android, iOS oder einem Webbrowser zu portieren. LibGDX bietet leider keine Möglichkeit CSG-Operationen durchzuführen, weswegen diese Funktionalität selber implementiert werden müsste.

jMonkey bietet sich für das Projekt insgesamt besser an. Es wird keine große Kontrolle wie in libGDX benötigt und die einfache Benutzbarkeit von jMonkey und dessen fertigen CSG-Implementierung bietet eine ideale Grundlage um einen Prototypen zu erstellen.

3.2.2 Blockbasierte Benutzeroberfläche

Um es dem Nutzer möglichst leicht zu machen, mit der Software zu interagieren, muss eine geeignete Benutzeroberfläche gefunden oder entwickelt werden. Dabei soll die Oberfläche möglichst ohne extra Fenster oder Kontextmenüs auskommen und gleichzeitig so intuitiv wie möglich sein.

[1] https://jmonkeyengine.org/
[2] http://jmonkeycsg.sourceforge.net/

OpenBlocks

OpenBlocks[3] ist eine Java-Bibliothek zur Erstellung von eigenen blockbasierten Sprachen. Blöcke können nach Belieben mithilfe einer XML-Datei selber erstellt und konfiguriert werden. Es ist unter anderem möglich Blockform, Beschreibung, Farben und Verbindungsteile zu anderen Blöcken anzupassen. Auch die Umgebung zum Erstellen und Löschen, sowie Klicken und Ziehen von Blöcken ist durch die Bibliothek gegeben. Ebenfalls sind entsprechende Schnittstellen, um auf Interaktionen und Ereignisse der Blockkombinationen zu reagieren, vorhanden.

OpenBlocks bietet damit eine gute Basis für einen Prototypen, allerdings wurde wegen zwei Problemen sich bewusst gegen OpenBlocks entschieden. Das erste Problem ist die mangelnde Dokumentation. Es gibt zwar Javadoc und ein kurzes Spezifikationsblatt zu der XML-Konfiguration, aber insbesondere das Spezifikationsblatt ist sehr kurz gehalten und übergeht viele Parameter, was das Entwickeln einer eigenen Blocksprache erschwert.

Das zweite Problem ist das Laufzeitverhalten. Die XML-Konfiguration wird zum Programmstart einmal geladen und die generierten Blöcke lassen sich zur Laufzeit nicht mehr ändern. Dies ist problematisch, da es Ziel dieses Projektes ist, mithilfe der Blöcke eine Baumstruktur interaktiv aufzubauen. Also muss ein Mechanismus implementiert werden, der bei entsprechender Verschachtelungstiefe die Größe der Blöcke korrekt anpasst, sodass alle Blöcke gleichmäßig in den Baum passen (dieses Verhalten wird in Sektion 4.4.5 noch einmal genauer erläutert). Da die Blöcke sich nachträglich nicht mehr verändern lassen, muss eine Alternative Bibliothek verwendet werden.

Eigene Implementierung

Eine andere Alternative für die Oberfläche ist es, von Grund auf eine eigene Implementierung zu entwickeln, mit der es möglich ist, Blöcke nach Belieben zu verschieben und zusammenzustecken. Da das Projekt in Java implementiert wird, stehen zwei GUI-Toolkits zur Verfügung: *Swing* und *JavaFX*.

Beide bieten einen ähnlichen Funktionsumfang und eine einfach zu benutzende Schnittstelle, um plattformunabhängige Benutzeroberflächen zu erstellen. JavaFX ist der Nachfolger von Swing und bietet viele Optimierungen und Verbesserungen, allerdings sind die

[3] https://github.com/mikaelhg/openblocks

meisten für die zu entwickelnde Software nicht weiter relevant. Weiterhin ist die Integration der jMonkey Engine in das gleiche GUI-Fenster wichtig. jMonkey bietet für Swing Applikationen eine extra Schnittstelle, die ohne weitere Probleme funktioniert. JavaFX dagegen wird nicht unterstützt und lässt sich nur über Umwege verwenden. Wegen diesen Umständen wird Swing als GUI-Toolkit gewählt.

4 Konzept

In diesem Kapitel werden Designentscheidungen basierend auf der vorangegangen Analyse bestimmt und erläutert. Zusätzlich werden zu implementierende Mechanismen der Software besprochen.

4.1 Blocktypen

Es gibt zwei Blocktypen: Primitive und Operatoren. Diese haben jeweils eine andere Form und Farbe, um sie leicht unterscheiden zu können. Um eine Verbindungsmöglichkeit zu signalisieren, haben die beiden Blocktypen jeweils entgegengesetze Konnektoren. Diese sind T-förmig und passen wie Puzzleteile zusammen.

Von den Primitiven gibt es drei Versionen: Würfel, Zylinder und Kugeln. Diese Blöcke haben das gleiche Aussehen und unterscheiden sich nur von der textuellen Beschreibung und dem zugehörigen 3D-Modell. Ein Primitiv-Block mit der Aufschrift Cube repräsentiert also einen Würfel, Sphere eine Kugel usw.

Für die Operatoren werden die typischen CSG-Operatoren wie Differenz, Schnittmenge und Union benutzt. Es gibt für jeden Operator einen eigenen Block, der wie bei den Primitiv-Blöcken sich nur von dem Text innerhalb des Blockes unterscheiden lässt. Bei dem Union-Operator werden zwei Modelle miteinander addiert und werden zu einem großen Modell. Bei der Differenz wird ein Modell von dem anderen abgezogen, sodass der überlappende Bereich entfernt wird. Zuletzt wird bei der Schnittmenge nur der überlappende Bereich behalten.

4.2 Design der Benutzeroberfläche

Um ein ungefähres Bild für die Benutzeroberfläche und Struktur zu haben, wurde aus den zuvor bestimmten Requirements beispielhaft ein Mockup, zu sehen in Abbildung 4.1, erstellt.

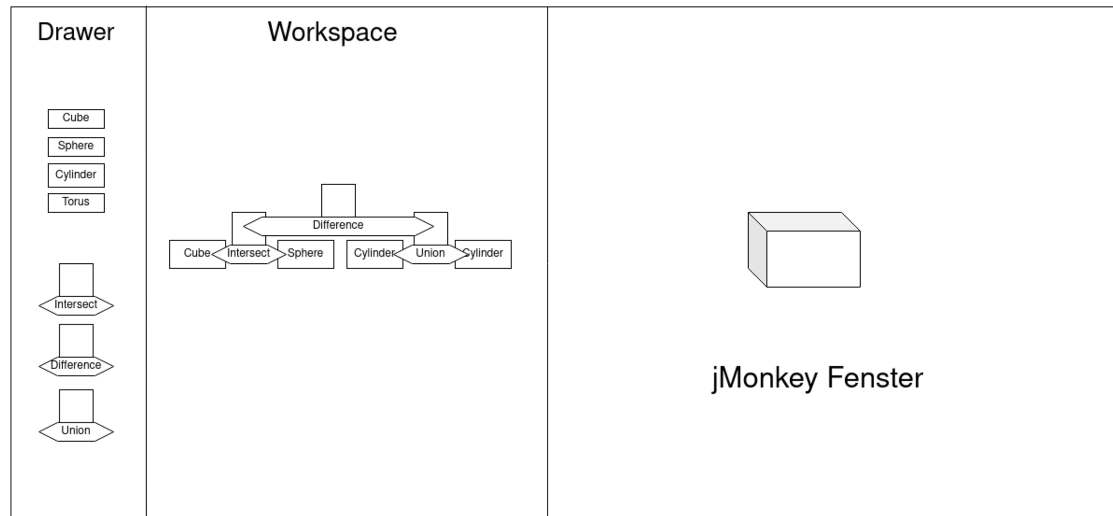


Abbildung 4.1: GUI-Mockup

Das Fenster wird in drei Teile aufgeteilt: Drawer, Workspace und der 3D-Darstellung mit jMonkey. Der Drawer und Workspace bilden den blockbasierten Teil der Oberfläche. Hier kann der Nutzer Blöcke durch Klicken und Ziehen verschieben und frei im Workspace platzieren. Soll ein neuer Block erstellt werden, so kann aus dem Drawer ein neuer Block herausgezogen werden. Zum Löschen kann ein Block einfach aus dem Workspace herausgezogen werden.

An einen Operator können jeweils nur zwei Primitive links und rechts über die Steckverbindung angeschlossen werden. Um zwei verknüpfte Primitive weiterverwenden zu können, muss an den oberen Anbau eines Operators ein weiterer Operator angeschlossen werden. Dadurch bildet sich eine Baumstruktur, welche beliebig geschachtelt werden kann (ebenfalls in Abbildung 4.1 zu sehen).

Werden Primitive in den Workspace gezogen, so tauchen diese auch als 3D-Modell in jMonkey parallel auf. Jede CSG-Operation wird dann dort durchgeführt und zeigt den aktuellen Stand eines Baumes an.

4.3 Software-Architektur

4.3.1 Architektur

Als Architektur-Pattern wurde das *Model-View-Controller* (MVC) Pattern gewählt. Dies eignet sich für die Anwendung sehr gut, da die Anwendung vom Aufbau klein bleibt und zwischen den Komponenten ein klarer Datenfluss erkennbar ist.

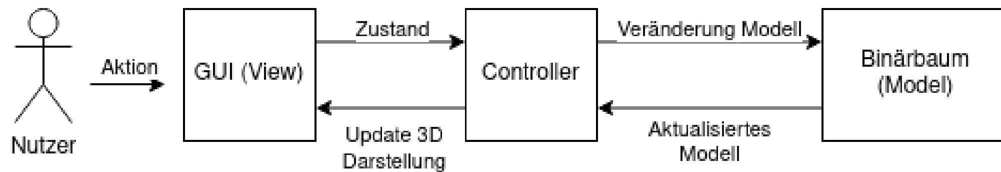


Abbildung 4.2: Datenfluss des Prototypen

Der Nutzer führt eine Aktion in der Oberfläche aus. Die View leitet diese Aktion an den Controller weiter und wird von diesem verarbeitet. Bei Bedarf, wird das Datenmodell verändert und die neue Änderung in der View wieder dargestellt. Dies ist vereinfacht in Abbildung 4.2 dargestellt.

View

Die View erstellt und verwaltet alle GUI-Elemente, sowie Benutzereingaben. Zum Start der Applikation konstruiert die View das nötige Fenster und initialisiert alle Event-Handler. Bei Benutzereingaben leitet die View entsprechende Funktionsaufrufe an den Controller weiter und aktualisiert entsprechend das Fenster um Rückmeldung für die Interaktion zu geben.

Controller

Der Controller bildet die zentrale Verwaltungskomponente und beinhaltet die Logik der Applikation. Er startet die Applikation und initialisiert das Model und die View. Es werden Aufrufe von der View entgegen genommen und bei Bedarf das Modell entsprechend bearbeitet.

Model

Das Datenmodell besteht in dieser Applikation aus Binärbaumen. Die Binärbäume stellen CSG-Bäume dar und ermöglichen es, durch rekursives traversieren CSG-Modelle zu generieren. Das Modell bietet eine Schnittstelle für typische Operationen zur Bearbeitung einer Baumstruktur, wie das Hinzufügen, Löschen und Suchen von Elementen.

4.4 Erstellung einer eigenen, blockbasierten Oberfläche

Das GUI-Toolkit Swing ermöglicht es, Benutzeroberflächen zu erstellen und diese mit typischen Komponenten (Components) wie Knöpfen, Texten und Eingabefeldern zu befüllen. Dabei wird zwischen Containern und Komponenten unterschieden. Ein Container kann beliebig viele Komponenten beinhalten und mithilfe eines *Layout-Managers* automatisch positionieren.

4.4.1 Zeichnen eines Blocks

Im ersten Schritt muss eine geeignete Komponente erstellt werden, die Blöcke darstellt. Mithilfe von Vererbung ist es möglich, innerhalb von Swing eigene Komponenten hinzuzufügen. Es muss nur eine *render()* Methode implementiert werden, in der dann mithilfe von Turtle-Grafik ein geeigneter Block gezeichnet werden kann. Abbildung 4.3 zeigt die Blockdesigns mit Konnektoren aus dem fertigen Prototypen.

Als Grundform der Blöcke wurden der Einfachheit halber Rechtecke gewählt. Der Operator muss speziell geformt sein, da einerseits Primitive Blöcke andocken können sollen und andererseits weitere Operatoren, um einen Baum bilden zu können.

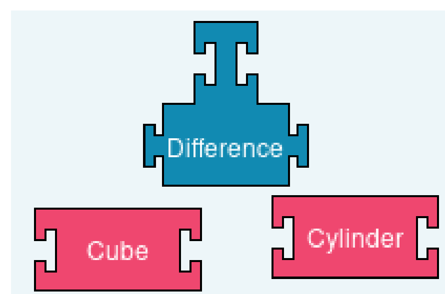


Abbildung 4.3: Blöcke mit Konnektoren erstellt in Swing.

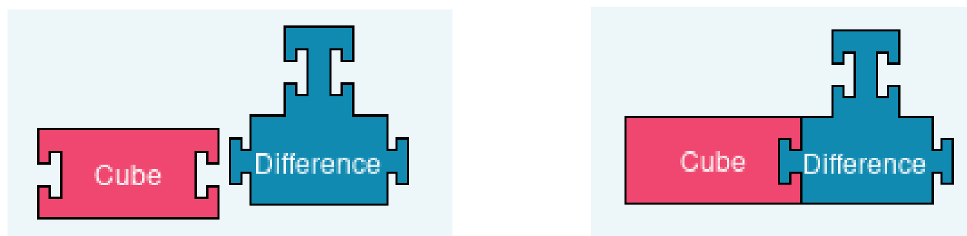
4.4.2 Organisation von Blöcken

Wurden die Komponenten erstellt, so müssen diese in einem Container untergebracht werden. Hierfür wird eine einfache, leere Fläche als Arbeitsfläche und ein scrollbarer Bereich für die Blockauswahl genutzt. In Swing wird normalerweise innerhalb eines Containers ein Layout-Manager verwendet. Da in diesem Kontext der Nutzer selber die Blöcke verschieben können soll, wird auf der Arbeitsfläche keiner benutzt (auch bekannt als *Null-Manager*).

In dem Drawer wird eine Instanz von jedem Block mit einem Layout-Manager listenförmig platziert und so konfiguriert, dass beim Anklicken dieser eine neue Kopie des Blockes erstellt werden kann. Dadurch kann beliebig oft aus dem Drawer ein Block herausgezogen werden.

4.4.3 Andocken

Das Verbinden von Blöcken soll dem Nutzer auch visuell vermittelt werden. Wird ein Block in der Nähe eines anderen Blocks losgelassen, so soll der zuvor bewegte Block an dem anderen Block einrasten. Um dies umzusetzen, muss jeder Block eine Liste von Einrastpositionen führen. Diese Positionen befinden sich jeweils an einem Konnektor. Wird ein Block losgelassen, so wird für jeden Konnektor des bewegten Blocks geprüft, ob sich in einem bestimmten Radius ein weiterer passender Konnektor eines weiteren Blocks befindet. Ist dies der Fall, so wird der Block an die richtige Position bewegt, sodass die Konnektoren zusammengesteckt werden. Das beschriebene Verhalten lässt sich anhand von den Abbildungen 4.4a und 4.4b nachvollziehen.



(a) Losgelassener Block vor dem Andocken. (b) Losgelassener Block nach dem Andocken.

Abbildung 4.4: Andocken von Blöcken.

Dabei ist zu beachten, dass auf beiden Seiten des Blocks Konnektoren sein müssen. Primitive müssen jeweils links und rechts an einem Operator angebracht werden können.

Auch Operatoren müssen auf beiden Seiten mit weiteren Operatoren verbunden werden können. Allerdings darf ein Primitiv oder die obere Seite eines Operatorblocks niemals auf beiden Seiten gleichzeitig verbunden sein, da sonst die Baumstruktur verloren geht. Sobald ein Block an einem Konnektor angedockt wird, muss dem Nutzer vermittelt werden, dass der Konnektor auf der anderen Seite nicht mehr benutzbar ist. Dies lässt sich, beispielsweise wie in Abbildung 4.4b, durch einfaches Verschwinden des Konnektors signalisieren.

4.4.4 Gleichzeitiges Bewegen

Um das Bewegen von Blockgruppen möglichst einfach zu gestalten, sollen durch Klicken und Ziehen eines Blocks, der gleichzeitig die Wurzel eines Baumes ist, alle Kinder von diesem Block ebenfalls mit verschoben werden. Dieser gezogene Baum kann dann ebenfalls an weitere Bäume oder einzelne Blöcke angedockt werden.

4.4.5 Dynamisches Wachsen

Wird ein zu großer Baum mithilfe von Blöcken gebaut, so kann es passieren, dass die Blöcke anfangen sich zu überlappen, zu sehen in Abbildung 4.5. Um dies zu Verhindern muss die Breite des Operatorblocks abhängig von der Verschachtelungstiefe des zugehörigen Baumes sein und bei jeder Änderung aktualisiert werden. Die Wurzel bildet damit den breitesten Block des Baumes und jede weitere Verschachtelungsebene ist nur halb so breit wie die zuvor, bis das Ende des Baumes und die Minimalgröße erreicht ist. Die korrekte Version ist in Abbildung 4.6 zu sehen.

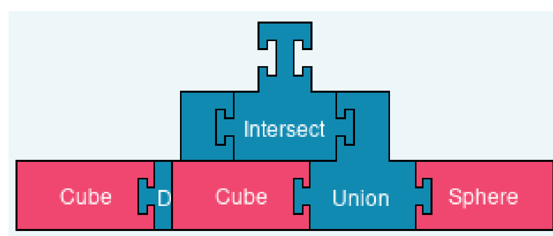


Abbildung 4.5: Sich überlappende Blöcke, wenn Blockgröße sich nicht ändert.

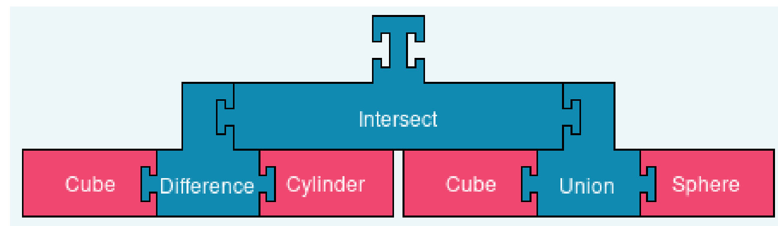


Abbildung 4.6: In die Breite wachsende Blöcke.

4.5 Baum-Datenstruktur

Um die CSG-Modelle korrekt zu generieren und gleichzeitig unabhängig von der Oberfläche zu bleiben, muss im Hintergrund eine Baum-Datenstruktur gehalten werden. Hierbei muss es sich um einen unsortierten Binärbaum handeln, welcher es ermöglicht, an beliebigen anderen Knoten weitere Knoten anzubringen. Hierfür wird eine eigene generische Implementierung von einem Binärbaum erstellt, welche typische Operationen wie Einfügen, Löschen, Suchen von Elementen und Berechnung der Verschachtelungstiefe unterstützt. Konkret muss beim Einfügen und Löschen eines Elements immer der Baum durchsucht werden, da einerseits beim Einfügen der Wurzelknoten des anzubringenden Elements und andererseits beim Löschen das zu entfernende Element gesucht werden muss.

In Betracht auf Laufzeitverhalten sollte das ständige Suchen kein Problem darstellen, da die Bäume nicht groß werden und von der Knotenzahl eher im geringen zweistelligen Bereich bleiben.

Die Oberfläche kann allerdings auch mehrere Bäume beinhalten. Wenn zwei Blöcke auf die Arbeitsfläche gezogen und nicht verbunden werden, so müssen zwei separate Bäume erstellt werden. Sobald zwei Blöcke oder Blockgruppen zusammengesteckt werden, müssen die Bäume ebenfalls an der gleichen Stelle verbunden werden. Es wird also eine Komponente benötigt, die mehrere Bäume verwaltet und jederzeit zuordnen kann, in welchem Baum sich momentan ein Block befindet.

4.6 3D-Darstellung

jMonkey bietet mehrere vorgefertigte Konfigurationen, die durch Vererbung bestimmter Klassen genutzt werden können. Eine dieser Konfigurationen, *SimpleApplication*, bietet einen Schnellstart in die jMonkey-Engine und stellt alle nötigen Objekte bereit. Unter

anderem bietet jMonkey in SimpleApplication Zugriff auf den Wurzelknoten des Szenengraphen für das Entfernen und Hinzufügen von Modellen, eine fertig konfigurierte Kamera und einen Input-Manager, der das Hinzufügen von Maus- und Tastatureingaben ermöglicht.

4.6.1 Bedingungen zur Darstellung

Wird ein Block innerhalb der Arbeitsfläche platziert, so muss parallel ein repräsentatives 3D-Modell erstellt und in den Szenengraphen eingehängt werden. Werden zwei Blöcke miteinander verknüpft, so müssen die beiden zugehörigen 3D-Modelle aus dem Szenengraphen entfernt, die CSG Operation durchgeführt und das Endergebnis wieder in den Szenengraphen eingehängt werden.

Wird ein Block durch Rausziehen aus der Arbeitsfläche gelöscht, so muss das dazugehörige Modell ebenfalls aus dem Szenengraphen ausgehängt und gelöscht werden.

4.6.2 Hervorheben der aktuellen Selektion

Um eine gute Übersicht bei der Darstellung von vielen 3D-Modellen zu behalten, soll das momentan ausgewählte Objekt markiert werden. Die anderen Objekte werden leicht transparent dargestellt, sodass sich immer leicht erkennen lässt, welches Primitiv oder CSG-Objekt welchen Teil des Baumes ausmacht. Dies ist beispielhaft in Abbildung 4.7 zu sehen. Parallel dazu wird auch der relevante Teilbaum in der Block-Ansicht anders eingefärbt, um erkennen zu können, welcher Block gerade ausgewählt ist..

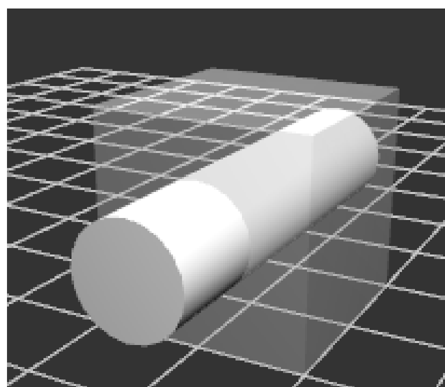


Abbildung 4.7: Zwei Modelle: Der Zylinder ist selektiert und opaque, während der Würfel transparent ist.

4.6.3 Veränderung der Maße, Translation und Rotation der 3D-Modelle

Ein größeres Problem stellt das Verändern von Parametern wie Position, Rotation oder Maßen dar. Da der Nutzer einfache 3D-Modelle erstellen können soll, muss es möglich sein, einzelne Primitive im Raum zu verschieben und zu rotieren. Es muss hierfür eine oder mehrere geeignete Methoden gefunden werden, die dies leicht und intuitiv umsetzen.

Rotations- und Translationsblöcke

Eine Möglichkeit wäre es, ebenfalls Blöcke für die Verschiebung und Rotation zu benutzen. Diese würden dann an die Primitive angeschlossen werden und mithilfe einer Textbox innerhalb des Blocks konfiguriert werden. Diese Idee wäre vermutlich die intuitivste und würde keine neuen Konzepte hinzufügen.

Auf der anderen Seite ist es schwierig, dies mit der vorangegangenen Planung umzusetzen. Durch die Architektur ist eine klare Trennung von Baumstruktur und 3D-Darstellung gegeben. Werden jetzt Verschiebungs- und Rotationsblöcke hinzugefügt, so werden diese Komponenten gemischt, da dann Eigenschaften der 3D-Modelle im Baum auftauchen.

Der Platzmangel stellt ein weiteres Problem dar. Primitiv-Blöcke haben bereits links und rechts jeweils einen Konnektor für die Operatoren, also bliebe nur oben oder unten als Verbindungsmöglichkeit. Also müssten die Operatoren auch in der Höhe angepasst werden, um Platz für die neuen Konfigurationsblöcke freizuhalten. Auch wird der sichtbare Baum bestehend aus Blöcken recht unübersichtlich, wenn jedes Primitiv jeweils ein Rotations- und Translationsblock angeschlossen hat.

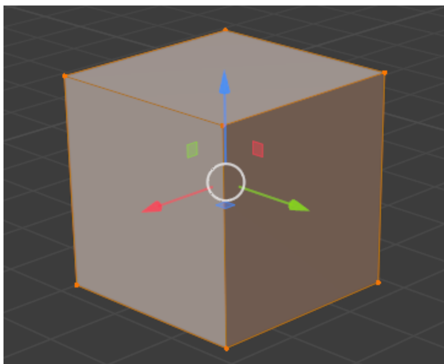
Seperates Fenster

Eine besonders leicht zu implementierende Alternative wäre ein einfaches Popup-Fenster, in welchem man Position, Rotation und Maße angeben kann. Dieses Menü könnte über ein einfachen Rechtsklick auf den betreffenden Block geöffnet werden. Die Lösung ist einfach und erfordert keine weiteren Implementierungen, da das GUI-Toolkit bereits alles anbietet.

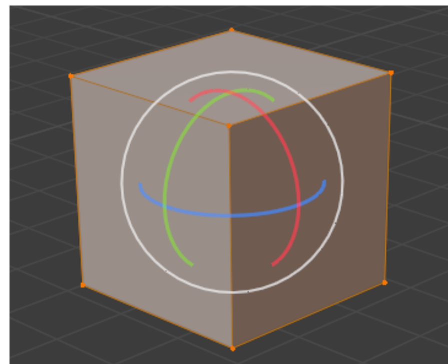
Allerdings muss der Nutzer auf diese Funktionalität hingewiesen werden oder er findet es per Zufall heraus, da nichts darauf deutet, dass es so ein Pop-up gäbe.

Interaktion mit 3D-Darstellung

Bisher soll die 3D-Darstellung nur zur Visualisierung dienen. Es gäbe allerdings die Möglichkeit, direkt im Fenster die Objekte zu bewegen. Typische Modellierungswerkzeuge bieten meistens sogenannte *Transformationsmanipulatoren* in Form von drei Pfeilen und drei Ringen (siehe Abbildung 4.8a und 4.8b). Diese lassen sich jeweils ziehen um das gewünschte Objekt auf einer Achse zu verschieben oder zu rotieren.



(a) Pfeile zu Translation auf drei Achsen.



(b) Ringe zum Rotieren auf drei Achsen.

Abbildung 4.8: Transformationsmanipulatoren in Blender.

Dieses Konzept ist ebenfalls sehr benutzerfreundlich, da es einfach durch Ausprobieren getestet werden kann. Nur die Veränderung der Maße ist hiermit nicht möglich.

Daher sollte idealerweise das Konzept mit dem Pop-up-Fenster mit dem der Transformationsmanipulatoren kombiniert werden. Es ließen sich dann grobe Anpassungen innerhalb der 3D-Darstellung mit Transformationsmanipulatoren durchführen und feine Anpassungen innerhalb des Pop-up-Fensters.

4.6.4 Kamerasteuerung

Damit der Nutzer die dargestellten Objekte betrachten kann, wird der Input-Manager von jMonkey benutzt. Dieser wird so konfiguriert, dass durch Klicken und Ziehen innerhalb der 3D-Darstellung die Kamera um das Zentrum rotiert.

Der Input-Manager wird für die Rotation der Kamera um ein Objekt benutzt, um dem Nutzer das Betrachten des Models aus allen Richtungen zu ermöglichen.

4.7 CSG-Generierung

Sobald zwei Blöcke innerhalb der Oberfläche verbunden werden, muss ein neues CSG-Modell erstellt werden. Innerhalb des CSG-Addons für jMonkey werden die repräsentativen 3D-Modelle der jeweiligen Blöcke zu einem Objekt mit einem Operator hinzugefügt. Danach kann die Generierung gestartet werden.

Die Generierung erfolgt rekursiv und wird mit der Wurzel des betroffenen Baumes gestartet. Um das CSG-Objekt zu generieren, werden die Modelle des linken und rechten Teilbaumes benötigt. Bestehen diese auch aus Operatoren, so wird auf den jeweiligen Teilbäumen ebenfalls eine CSG-Generierung gestartet, bis auf ein Primitiv gestoßen wird, welches immer ein verfügbares Modell hat und für die Generierung genutzt werden kann. Listing 4.1 beschreibt diese rekursive Funktion mit Pseudocode für vollständige Bäume.

```
def generateCSG(block):
    if(block is operator):
        left = getLeftChild().getModel()
        right = getRightChild().getModel()
        csgBlender = csgBlender()

        csgBlender.add(generateCSG(left))
        csgBlender.add(generateCSG(right))
        return csgBlender.generate(block.operator)
    else: # is primitive
        return model
```

Listing 4.1: Pseudocode für rekursive Funktion zur CSG-Generierung

Auch beim Greifen von Blöcken, die zuvor verbunden waren, müssen die CSG-Modelle neu generiert werden. Einerseits muss das CSG-Modell des gegriffenen Blocks neu generiert werden, da dieses nicht mehr Teil des ursprünglichen CSG-Modells ist und andererseits muss auch der alte verbundene Baum neu generiert werden, da ein Block aus diesem entfernt wurde.

Hierbei ist zu beachten, dass auch bei einem nicht vollständigen Baum ein CSG-Modell generiert werden kann. Beispielsweise ist es möglich, dass der Nutzer die Schnittmenge

mit keinem weiteren Block bildet. Dieser Baum ist aber dennoch gültig, da die Schnittmenge mit einer leeren Menge ebenfalls eine leere Menge ergibt und damit ein gültiges Ergebnis ist. Das System muss also robust implementiert sein, sodass in solchen Situationen keine Fehler auftreten.

5 Implementierung

Es folgt eine genauere Beschreibung von Implementierungsdetails. Es werden geeignete strukturelle Diagramme präsentiert und erläutert, sowie manche Features im Detail besprochen. Zuletzt wird auf verwendete Werkzeuge und aufgetretene Probleme innerhalb der Entwicklung eingegangen.

5.1 Software Engineering

5.1.1 Komponenten

Abbildung 5.1 zeigt das Komponentendiagramm der Anwendung. Es wurde, wie im Konzept festgelegt, das MVC Entwurfsmuster implementiert.

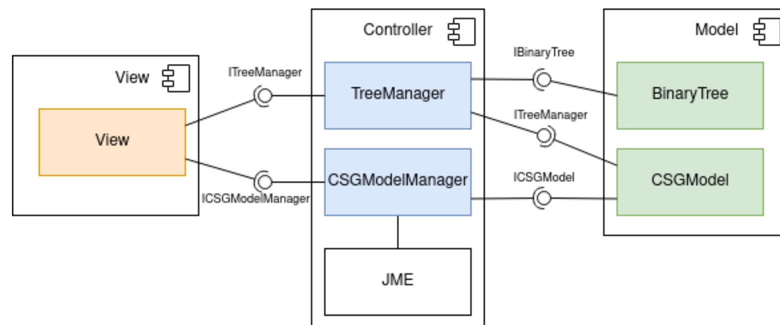


Abbildung 5.1: Komponentendiagramm der Anwendung.

Es folgt eine kurze Erläuterung der wichtigsten Klassen aus dem Komponentendiagramm aus Abbildung 5.1.

BinaryTree Generische Klasse eines Binärbaums, die die typischen Funktionalitäten wie Einfügen, Suchen und Löschen bietet.

CSGModel Repräsentiert das 3D-Modell eines Blocks. Unter anderem kann sie mithilfe des TreeManagers rekursiv den unter sich liegenden CSG-Baum auswerten.

TreeManager Verwaltet alle erstellten Bäume. Ist im Stande, bei Bedarf Bäume zu erstellen, zu löschen oder Knoten innerhalb eines Baumes zurückzuliefern. Wird eine Operation auf einem Baum benötigt, so sucht der TreeManager den richtigen Baum raus und führt die Operation durch.

CSGModelManager Verwaltet alle Instanzen von CSGModel. Bei Erstellung eines Blockes wird ein zugehöriges CSGModel erstellt und in den Szenengraph eingehängt. Interagiert bei Bedarf mit der jMonkey-Engine.

JME Kapselt alle relevanten jMonkey-Engine Komponenten.

View Erstellt und verwaltet die gesamte Benutzeroberfläche.

Es gibt zusätzlich in der View Komponente mehrere Event-Handler Klassen, die sämtliche Funktionalitäten der Interaktion mit den Blöcken innerhalb der GUI übernehmen. Diese wurden aus Platzgründen hier weggelassen.

5.1.2 Dependency Injection Entwurfsmuster

Um die Komponenten sauber voneinander trennen zu können, wird das *Dependency Injection* Entwurfsmuster genutzt. Komponenten kennen sich dabei nur über Interfaces, wodurch Objekterstellung und Benutzung getrennt werden. Dies ermöglicht einen leichten Austausch der einzelnen Komponenten. Beispielsweise ist es mit Leichtigkeit möglich, eine andere Implementierung eines Binärbaums einzubinden, ohne große Veränderungen im Quellcode. Auch ist ein optimaleres Testen möglich, da jede Komponente isoliert werden kann.

5.1.3 Single-Responsibility-Prinzip

Im Rahmen der Entwicklung wurde verstärkt darauf geachtet, dass *Single-Responsibility-Prinzip* umzusetzen. Jede Klasse soll dabei konkret eine Aufgabe übernehmen. Dies hat die spätere Entwicklung sehr vereinfacht, da durch das modulare Design sehr leicht neue Features / Klassen implementiert werden konnten. Insbesondere bei der Implementierung von Event-Handlern war dies hilfreich, da jeder Handler genau eine Aufgabe übernimmt

und bei Bedarf hinzufügt oder entfernt werden kann. Zu sehen ist dies beispielhaft in dem Klassendiagramm 5.2.

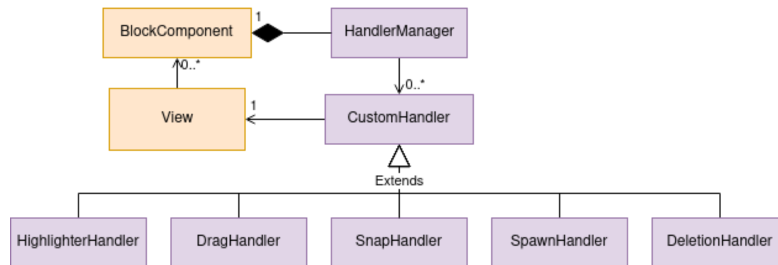


Abbildung 5.2: Klassendiagramm einiger Event-Handler.

Es wurde zum Sammeln aller Blockfunktionalitäten ein HandlerManager geschrieben, der alle anderen Handler verwaltet und Events an diese weiterleitet. Jeder einzelne Block verfügt dabei über einen eigenen HandlerManager.

5.1.4 Sequenzen

Es folgen Sequenzdiagramme zu den wichtigsten Interaktionen innerhalb des Prototypen, neues Erstellen von Blöcken und das Verbinden dieser. Hier handelt es sich nur um eine Übersicht, daher wurden Hilfsklassen, die fürs weitere Verständnis nicht benötigt werden, weggelassen.

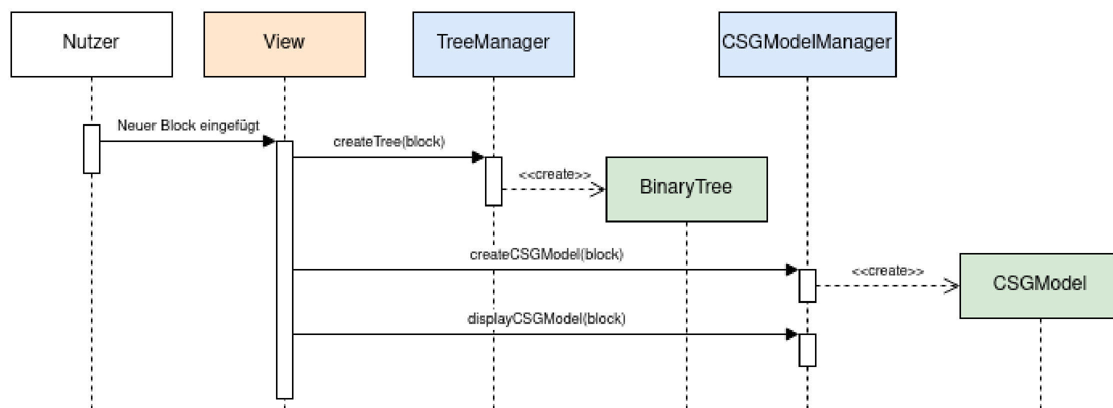


Abbildung 5.3: Sequenz vom Hinzufügen eines Blockes.

Abbildung 5.3 zeigt das Verhalten bei dem Hinzufügen eines Blockes in der Oberfläche. Es wird vom TreeManager ein neuer Baum erstellt, an dessen Wurzel sich der

neu hinzugefügt Block befindet. Zusätzlich wird von dem neuen Block ein 3D-Modell vom CSGModelManager erstellt und in den Szenengraphen eingehängt. Der Tree- und CSGModelManager können nun bei Änderungen des Datenmodells entsprechende Änderungen an den korrekten/es Baum/Model weiterleiten, wie beispielsweise das Verbinden von zwei Blöcken (siehe Abbildung 5.4).

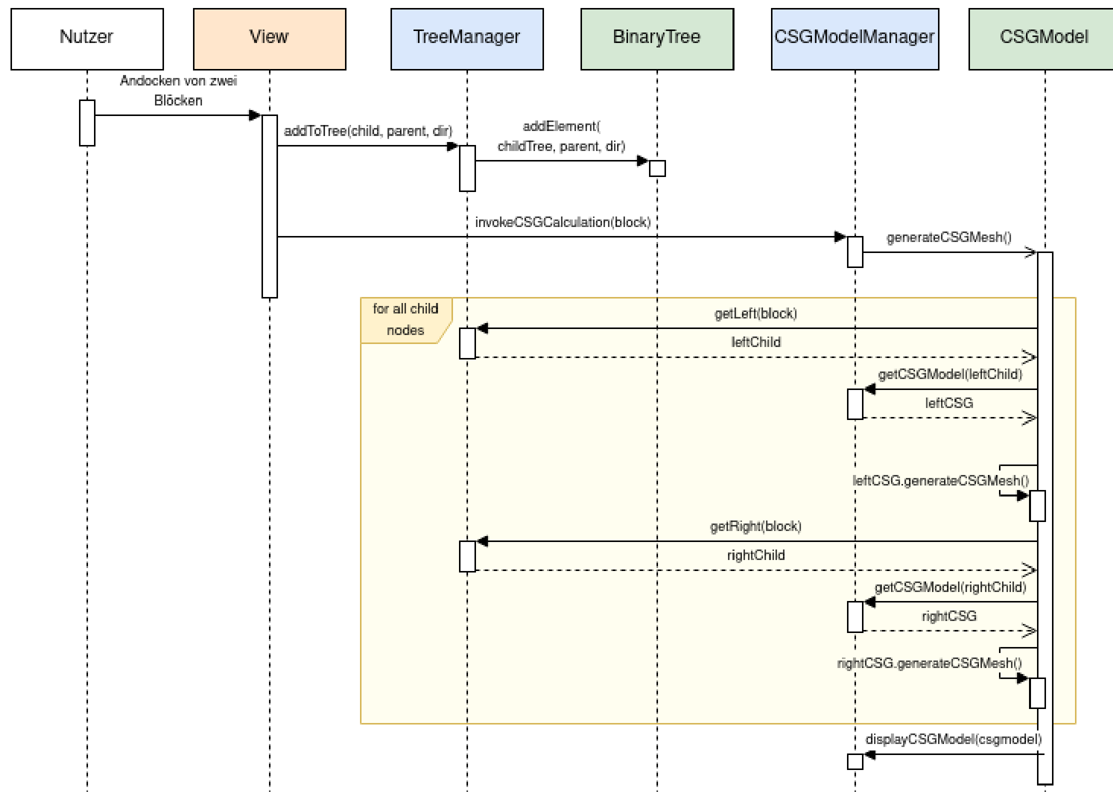


Abbildung 5.4: Sequenz vom Verbinden zweier Blöcke.

Wird ein Block an einen anderen angedockt, so muss das Datenmodell aktualisiert werden. Dies geschieht durch einen Aufruf zum TreeManager. Dieser verknüpft die beiden vorhandenen Bäume der Blöcke entsprechend. Zusätzlich muss die 3D-Darstellung erneuert werden, da eine neue Blockkomposition gebildet wurde. Daher wird die CSG-Generation an der Wurzel des interagierten Baumes gestartet. Von der Wurzel aus wird rekursiv von allen Kindknoten des Baumes das CSG-Model generiert. Dabei werden jeweils vom Tree- und CSGModelManager die relevanten Objekte herangeholt. Ist die Generation abgeschlossen, so wird das neue Modell in den Szenengraphen eingehängt.

Die gesamte CSG-Generierung wird in einem seperatem Thread durchgeführt, da ansonsten die Benutzeroberfläche während der Generierung nicht benutzbar wäre.

5.2 Benutzeroberfläche

Die fertige Benutzeroberfläche, zu sehen in Abbildung 5.5, ähnelt dem Mockup aus der Konzeption (Abbildung 4.1). Allerdings wurde die Anordnung der Komponenten leicht abgeändert.

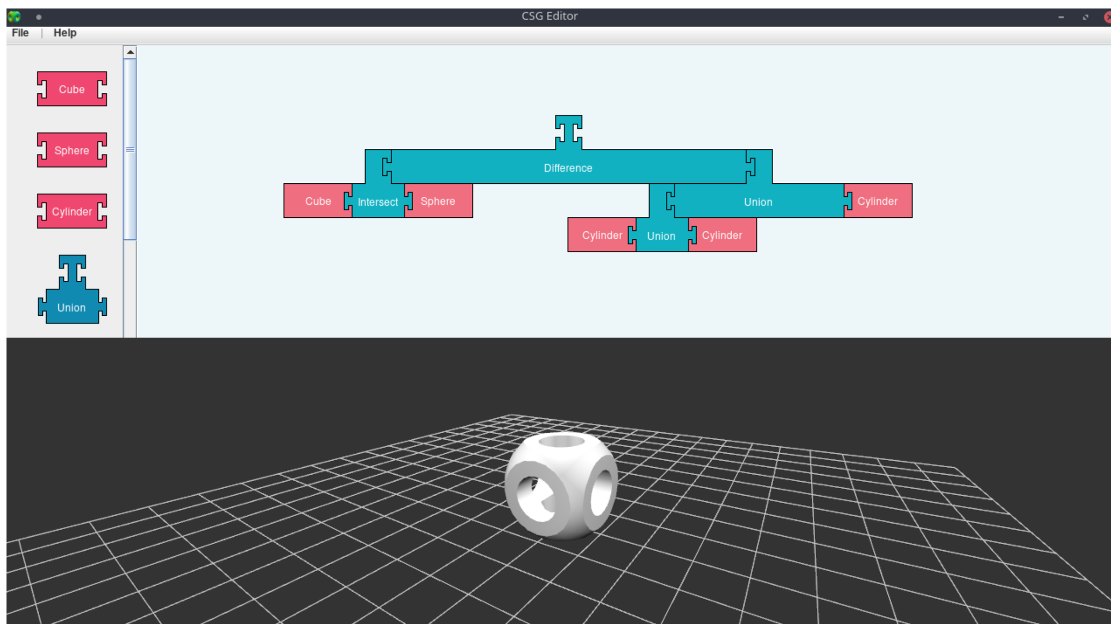


Abbildung 5.5: GUI des Prototypen mit dem gebauten Beispiel aus Abbildung 2.1.

Die Arbeitsfläche und die 3D-Darstellung befinden sich nun übereinander, anstatt nebeneinander, da durch das dynamische Wachsen der Blöcke es sehr schnell zu Platzproblemen gekommen ist. Wird das Beispiel aus Abbildung 2.1 gebaut, so sieht man, dass bereits bei einer Schachtelungstiefe von drei der Baum stark in die Breite gewachsen ist.

Zusätzlich wurde auch noch eine einfache Menüleiste hinzugefügt. Ziel der GUI ist es, mit möglichst wenig Elementen auszukommen und gleichzeitig intuitiv zu sein. Da Menüleisten in fast jedem gängigen Programm vorhanden sind, sollten diese ein vertrautes Konzept sein. Dort wurde ein Export-Feature sowie eine kleine Bedienungsanleitung hinterlegt.

Um Blöcke geeignet von der Größe her zu konfigurieren und dessen Position, sowie Rotation einzustellen, wurde ein einfaches extra Fenster hinzugefügt, wie in Abbildung 5.6 dargestellt. Dieses kann über einen Rechtsklick auf den gewünschten Block aufgerufen werden. In diesem kann der Nutzer alle für die Form relevante Parameter anpassen und eine Position sowie Rotation angeben.

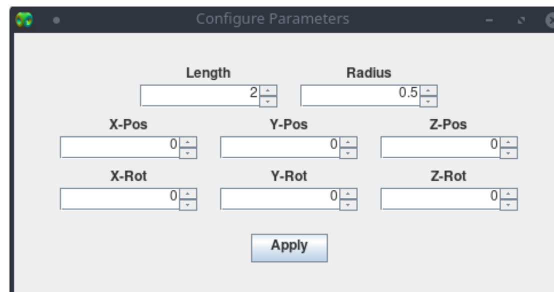


Abbildung 5.6: Konfigurationsfenster des Prototypen für einen Zylinder.

5.3 Hindernisse

5.3.1 Rechteckige Bounding Boxen in Swing

Während der Entwicklung der GUI hat sich recht schnell ein Problem mit den Formen der Operatorblöcke herausgestellt. Diese sind nicht rechteckig, sondern bestehen aus zwei unterschiedlich großen Rechtecken. Dies stellte sich als problematisch heraus, da in Swing alle Komponenten eine große rechteckige Bounding-Box haben. Die Bounding-Box beschreibt die Grenze einer Komponente und damit dessen interagierbare Fläche. Abbildung 5.7 zeigt beispielhaft drei Blöcke mit ihren Bounding-Boxen.

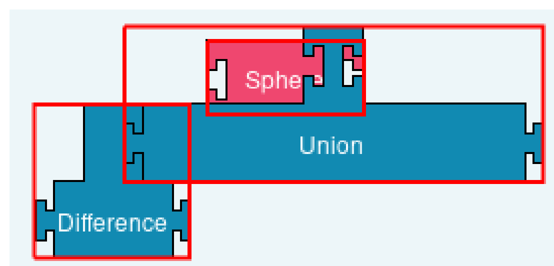


Abbildung 5.7: Blöcke mit eingezeichneten Bounding-Boxen. Ein Operatorblock verdeckt einen Primitivblock

Die obere Fläche des Union-Blockes hat viel freie Fläche, die immernoch Teil der Bounding-Box ist. Der sich unter der Union befindene Sphere-Block lässt sich nun nicht mehr bewegen, ohne zuvor den darüberliegenden Block mit zu großer Bounding-Box zu entfernen.

Um dies zu Lösen, muss eine eigene *contains()* Methode implementiert werden, die zurückliefert, ob sich ein Punkt innerhalb der Fläche einer Komponente befindet. Herausforderung hierbei war es, diese contains() Methode mit dem dynamischen Wachsen der Blöcke zu kombinieren. Bei jedem Klick eines Blockes muss nun geprüft werden, ob der Klick auf der tatsächlich gezeichneten Fläche stattgefunden hat oder nur auf der leeren Fläche innerhalb der Bounding-Box. Ist Letzteres der Fall, so muss durch alle anderen Blöcke iteriert werden, um zu prüfen, ob sich ein Block unter dem fälschlicherweise angeklickten Block befindet. Wird einer gefunden, so muss jedes Maus-Event das durchgeführt wird an den darunterliegenden Block weitergeleitet werden.

Ein einfaches Weiterleiten reicht dabei nicht aus. Stattdessen muss das Event korrekt konvertiert werden, da es relative Positionsdaten zur angeklickten Komponente beinhaltet. Hierfür bietet Swing die Methode *convertMouseEvent()* an, welche das Maus-Event so verändert, dass das Event relativ zur angegebenen Komponente umwandelt.

5.3.2 Leere Meshes in jMonkey

Um die Generation der CSG-Modelle einfach zu gestalten, wurde festgelegt, dass jeder Block ein repräsentatives Mesh haben sollte, einschließlich der Operatoren. Die Operatoren würden dann ein leeres Mesh haben, bis ein ein valides CSG-Modell aus den angeschlossenen Blöcken generiert werden kann. Dieses leere Mesh würde durch das neu generierte Mesh ersetzt werden. Leere Meshes lassen sich auch ohne Probleme in den Szenengraphen einhängen und sind nicht weiter sichtbar.

Das CSG-Addon für jMonkey hat mit leeren Meshes allerdings Probleme, da es bei der CSG-Berechnung mit einem leeren Mesh zu Exceptions kommt und die Generierung fehlschlägt. Innerhalb der Rekursion für die Generierung muss also umständlich und häufig auf Null-Pointer und leere Meshes geprüft werden.

Im gleichen Kontext gibt es noch den Fehler, dass die CSG-Generation fehlschlägt, wenn das resultierende CSG-Modell leer sein wird. Bisher war es nicht möglich, diesen Fehler zu verhindern, da es nicht ohne Weiteres möglich ist, im Vorhinein zu wissen, ob eine CSG-Berechnung ein leeres Ergebnis haben wird.

5.3.3 Versionen höher als Java 8

jMonkey bietet durch eine Abstraktionsschicht die Möglichkeit, verschiedene Rendering-Engines zu benutzen. Standardmäßig wird die *Java Lightweight Game Library* (JLWGL) verwendet, aber es kann auch alternativ *JOGL* oder direkt *OpenGL* genutzt werden. Konkret benutzt jMonkey JLWGL 2.9.3, was leider in Kombination mit Java Versionen höher als 8 zu Abstürzen führt. Eine genaue Ursache konnte nicht ermittelt werden. Leider lässt sich JLWGL nicht weiter updaten, da neuere Versionen die Integration in eine Swing-GUI nicht unterstützen.

5.3.4 Transfer von Blöcken zwischen Containern

Die erste Implementierung des Verschiebens einzelner Blöcke war sehr schnell einsatzbereit. Allerdings war es nur möglich, diese Blöcke innerhalb eines Swing-Containers zu verschieben. Die Oberfläche besteht allerdings aus mehreren Containern, wie dem Drawer und der Arbeitsfläche. Wurde beispielsweise versucht, einen Block aus dem Drawer in die Arbeitsfläche zu ziehen, so blieb der Block innerhalb des Drawers und wurde außerhalb des sichtbaren Bereiches bewegt.

Um dies zu verhindert, wurde ein transparenter Container mithilfe einer Layered-Pane genutzt, welche sich über das gesamte Fenster erstreckt. Wird ein Block zum Bewegen angeklickt, so wird dieser zur transparenten Ebene hinzugefügt und aus dem alten Container gelöscht. Da die transparente Ebene alle anderen überdeckt, kann der Block beliebig auf der Oberfläche verschoben werden. Wird der Block losgelassen, so muss der Block wieder aus der transparenten Ebene entfernt werden und zur alten Ebene wieder hinzugefügt werden. Beim Wechsel zwischen den Ebenen musste die relative Position der Container immer umgerechnet werden, da die Container sich an unterschiedlichen Positionen innerhalb des Fensters befinden.

5.4 Werkzeuge

5.4.1 Build-System Gradle

Um die Applikation leicht zusammen mit den benötigten Bibliotheken auszuliefern wurde *Gradle*[1] benutzt. Gradle ist ein Build-System, welches es ermöglicht, automatisiert abhängige Bibliotheken aus Repositories herunterzuladen und korrekt in Applikationen einzubinden. Mit nur einem Befehl ist es möglich, die gesamte Applikation zu kompilieren und alle nötigen Abhängigkeiten aufzulösen.

5.4.2 Versionskontrolle git

Für die Versionsverwaltung wurde git 2.30.1 benutzt. Das Repository befindet sich in der HAW GitLab. Dabei wurden zwei Branches benutzt. Auf dem Master-Branch befand sich immer ein funktionsfähiger Stand, der erst aktualisiert wird, wenn ein Feature fertig implementiert wurde. Auf dem Dev-Branch wurden alle sonstigen Zwischenstände gespeichert.

5.4.3 Swing Explorer

Während der GUI-Entwicklung kam es oft zu Schwierigkeiten. Oftmals waren Komponenten nicht zu sehen oder hatten ein nicht erklärliches Verhalten. Um die Fehlersuche zu erleichtern wurde das Werkzeug *Swing Explorer*[2] genutzt. Dies ermöglicht eine bessere Einsicht in die einzelnen Swing Komponenten und hilft, Werte wie Größen und Positionen besser nachzuvollziehen.

[1] https://gradle.org/
[2] https://github.com/swingexplorer/swingexplorer

6 Evaluation

In diesem Kapitel wird ein passendes Bewertungsschema ausgewählt und der Prototyp anschließend bewertet.

6.1 Methodik

Blockbasierte Systeme werden in den meisten Fällen verwendet um einen bestehenden Prozess einfacher zu gestalten und nicht vertrauten Nutzern einen leichteren Einstieg zu gewähren. Um nun zu ermitteln, ob denn der ermittelte Prototyp benutzerfreundlich ist, werden geeignete Bewertungsschemata benötigt.

6.1.1 Probandentests mit Umfrage

Eine Möglichkeit die Qualität der User-Experience zu testen ist es, mehrere unerfahrene Nutzer im Bereich der Modellierung den Prototypen testen zu lassen. Diese erhalten beispielsweise ein Bild von einem Modell und müssen versuchen, dieses nachzumodellieren. Dabei wird die Funktionsweise des Prototypen nicht näher erläutert, sodass die Probanden durch ausprobieren und mithilfe von Intuition arbeiten. Anschließend ließe sich mit einer Umfrage oder einem Interview die Erfahrungen der Nutzer sammeln und auswerten.

Dieser Ansatz bringt jedoch viele Probleme mit sich. Es muss ein Fragenkatalog entwickelt werden, der möglichst viele Aspekte der Oberfläche abdeckt. Der kritischste Punkt ist Bias bei den Probanden. Die Probanden werden wahrscheinlich aus dem näheren Umfeld gewählt und dadurch die Applikation eher subjektiv bewerten. Daher kommt diese Ansatz eher nicht in Frage.

6.1.2 Heuristiken von Kölling und McKay

Alternativ lassen sich die zuvor angesprochenen Ergebnisse aus Abschnitt 2.2.2 aus der Arbeit von Kölling und McKay anwenden. Anhand der dort entwickelten Heuristiken soll der entwickelte Prototyp möglichst objektiv bewertet werden. Die Heuristiken wurden übersetzt und in der Tabelle 6.1 abgebildet.

Name	**Beschreibung**
Engagement	Das System sollte einladend und motivierend für Nutzer wirken.
Non-threatening	Das System sollte dem Nutzer die Sicherheit geben, dass er ohne Konsequenzen experimentieren kann.
Minimal language redundancy	Innerhalb des Systems sollten möglichst keine Redundanzen auftreten, um den Nutzer nicht unnötig zu verwirren.
Learner-appropriate abstractions	Das Konzept hinter der blockbasierten Sprache sollte möglichst weit abstrahiert sein, sodass es für Anfänger verständlich ist.
Consistency	Die Benutzeroberfläche sollte im Design konsistent und einheitlich sein, sowie Konzepte richtig darstellen.
Visiblity	Der Nutzer sollte jederzeit den Status des Systems erkennen und einfach navigieren können.
Secondary notations	Das System sollte auf mehrere Weisen eine Information darstellen (Form, Farben).
Clarity	Die Benutzeroberfläche sollte simpel aufgebaut und strukturiert sein.
Human-centric syntax	Syntaktische Elemente sollten einfach gehalten und komplexe Terminologie vermieden werden.
Edit-order freedom	Das System sollte keine Bearbeitungsreihenfolge von Elementen vorgeben.
Minimal viscosity	Einfaches und schnelles Bearbeiten von Elementen sollte jederzeit möglich sein.
Error-avoidance	Das System sollte möglichst so aufgebaut sein, dass keine Fehler durch den Nutzer entstehen können.
Feedback	Das System sollte jederzeit Feedback zu Aktionen liefern und bei Problemen Lösungen oder Alternativen vorschlagen.

Tabelle 6.1: Heuristiken für blockbasierte Programmiersprachen. Quelle: [8]

Die Heuristiken bieten einen sehr guten Überblick über die meisten Aspekte einer blockbasierten Benutzeroberfläche und werden daher für die Evaluation eingesetzt. Dabei kann

es sein, dass manche Punkte schwieriger zu bewerten sind als andere, da diese Heuristiken eher auf große vollständige Lernplattformen zugeschnitten sind. Bei dem entwickelten Prototypen handelt es sich eher um einen kleinen Demonstration, weswegen auf manche Punkte mehr eingegangen werden kann, als auf andere. Auch überschneiden sich viele Punkte in manchen Aspekten und Argumente würden sich wiederholen. Daher werden manche Punkte im gleichen Kontext diskutiert.

Die Heuristiken beziehen sich allerdings nur auf die Interaktion mit der GUI. Inwiefern sich mit dem Prototypen 3D-Modelle erstellen lassen, muss separat evaluiert werden.

6.2 Auswertung

6.2.1 Interaktion

Engagement

Leider ist Engagement mit am Schwierigsten zu bewerten. Der entwickelte Prototyp hat keine Features oder Inhalte, die diesen Aspekt betreffen. Systeme wie Scratch haben in diesem Punkt meistens ein Maskottchen und werben aktiv mit Nutzungsmöglichkeiten und zeigen diese innerhalb der Applikation anhand von Beispielen. Dadurch wird der Nutzer dazu motiviert, selber Dinge auszuprobieren und bekommt durch Beispiele eigene Ideen.

Non-threatening

Mit dem Start der Applikation kann der Nutzer direkt beginnen zu modellieren. Die Benutzeroberfläche ist minimalistisch und hat nur sehr wenige interagierbare Schaltflächen. Es ist mit wenigen Schritten möglich, ein einfaches Modell zusammenzustecken. Hat der Nutzer einen Block falsch platziert, so kann er diesen einfach wieder wegbewegen oder löschen. Es können ohne Konsequenzen Blöcke immer wieder neu erstellt, gelöscht oder kombiniert werden. Der Prototyp wirkt insgesamt nicht überfordernd.

Das Rechtsklickmenü eines Blockes besteht im Gegensatz zum Rest der Applikation aus sehr vielen Eingabefeldern für Größen, Position und Rotation. Dieses ist eher überladen und könnte den Nutzer im ersten Moment überfordern und abschrecken.

Minimal language redundancy

Da der Prototyp nur einen sehr kleinen Funktionsumfang bietet, gibt es daher auch keine Redundanzen, die den Nutzer unnötig verwirren. Es gibt nur zwei Blocktypen, die sich jeweils verbinden lassen.

Höchstens lassen sich die beidseitigen Konnektoren kritisieren, wovon aber nur einer immer benutzbar ist. Um dies zu verhindern, müsste man zweimal den gleichen Block einfügen, mit jeweils verschiedenen Konnektoren. Dies wäre aber eine deutlich größere Redundanz.

Learner-appropriate abstractions, Human-centric syntax

CSG zeichnet sich durch Mengenoperationen an Modellen aus und an der daraus entstehenden Baumstruktur. Durch die Oberfläche wird beides vermischt und abstrahiert, sodass der Nutzer nichts von CSG-Bäumen wissen muss, um den Prototypen zu nutzen. Die Baumstruktur entsteht von alleine und ist durch die Blockformen automatisch gegeben.

Bezogen auf menschlich zentrierte Syntax ist die Namensgebung der Operationsblöcke eher schlecht gewählt. Jemand ohne Vorwissen über Mengenoperationen könnte Schwierigkeiten haben, diese auf Anhieb ohne Ausprobieren zu verstehen.

Consistency, Visibility und Clarity

Die Benutzeroberfläche ist sehr einfach gehalten und besteht zum Programmstart nur aus drei Flächen, wodurch der Nutzer nicht überfordert wird und direkt anfangen kann mit der Anwendung zu interagieren. Die Bereiche sind logisch angeordnet und in Arbeitsfläche und 3D-Darstellung getrennt. Diese sind jeweils farblich mit verschiedenen Grautönen getrennt und leicht erkennbar. Auch die verschiedenen Blocktypen haben klar erkennbare verschieden Farben. Zu jedem Zeitpunkt kann der Nutzer den Status des Systems erfassen, da es nur das Hauptfenster gibt und keine Änderungen innerhalb ohne die Interaktion des Nutzers innerhalb der GUI auftreten.

Das Einzige, was der Nutzer nicht einsehen kann, ist die CSG-Generierung. Dauert diese sehr lange, etwa weil der Nutzer ein sehr komplexes Modell gebaut hat, so ist das Modell zwischenzeitlich nicht im Szenengraphen und taucht erst als vollständiges Modell auf,

wenn es fertig berechnet wurde. In der Zwischenzeit erhält der Nutzer keine Information darüber, dass gerade ein Modell generiert wird.

Secondary notations

Die Oberfläche bietet mehrere visuelle Hinweise. Es wird jederzeit der zuletzt ausgewählte Block leuchtend angezeigt und parallel in der 3D-Ansicht markiert. Wird dieser Block mit einem weiteren verbunden, so wird der gesamte, entstandene Baum ausgewählt.

Auch das Andocken der Blöcke wird visuell unterstützt. Beim Verbinden bewegen sich die Blöcke bei der richtigen Distanz automatisch in die richtige Position.

Eine Notation, die fehlt, ist die Referenzierung zwischen einem Block und der repräsentativen 3D-Darstellung. Sind viele Blöcke des gleichen Typs innerhalb der Arbeitsfläche, so ist es unmöglich zu bestimmen, welcher Block zu welchem Modell gehört.

Edit-order freedom

Da innerhalb der Anwendung Binärbäume mit Blöcken gebaut werden, kann der Nutzer frei entscheiden, ob er diese von unten nach oben oder umgekehrt aufbaut. Auch die Reihenfolge, welcher Block an welchen angeschlossen wird ist nicht vorgegeben und kann frei gewählt werden.

Minimal viscosity

Zum Programmstart kann der Nutzer direkt loslegen, Blöcke zu platzieren. Das Blockauswahlmenü ist jederzeit sichtbar und kann sofort durch Klicken und Ziehen benutzt werden. Auch platzierte Blöcke können auf die gleiche Art sofort neu angeordnet werden. Um Blöcke zu bearbeiten, muss nur ein Rechtsklick auf den gewünschten Block gemacht werden.

Error-avoidance und Feedback

Durch die Konnektoren der Blöcke wird eine syntaktische Ordnung vorgegeben, wodurch es unmöglich ist, einen ungültigen Baum zu erstellen. Trotz dieser Ordnung ist es weiterhin möglich, einen unvollständigen Baum zu bauen, welcher beispielsweise nur aus Operatoren besteht. Die Applikation verhindert in diesen Fällen bestmöglich, dass so ein Modell generiert wird. Dass so ein Baum fehlerhaft ist, wird dem Nutzer nicht mitgeteilt. Generell gibt es kein Feedback-System, was den Nutzer über Ereignisse informieren kann.

6.2.2 Modellierung

Innerhalb der Anwendung ist es möglich, einfache CSG-Modelle zu erstellen. Es gelten die typischen CSG-Limitationen im Kontext der Modellierung. Allerdings bringt die Benutzeroberfläche ebenfalls eigene Limitierungen mit sich. Eine davon ist der Platz innerhalb der Arbeitsfläche. Durch das dynamische Wachsen der Blöcke kommt es bei zu hoher Verschachtelungstiefe schnell zu Platzproblemen, wodurch Blöcke breiter werden als Platz innerhalb des Fensters ist.

Auch könnte die Bearbeitung von komplexeren Modellen hinderlich werden. Durch die hohe Anzahl von Blöcken steigt auch die Anzahl an einzelnen CSG-Berechnungen. Dies kann den Modellierungsfluss hindern, da manchmal mehrere Sekunden gewartet werden muss, bis das Modell generiert wurde.

Ein weiteres Problem ist die Bedienung. Es ist zwar möglich, fast beliebige Modelle zu erstellen, allerdings ist das Bewegen und Rotieren über das Pop-up-Fenster der Primitive sehr sperrig und nicht sonderlich intuitiv. Es müssen oft Positionsangaben geschätzt und korrigiert werden. An dieser Stelle ist die Implementierung von Transformationsmanipulatoren wünschenswert.

7 Fazit

Ziel war es, eine blockbasierte Benutzeroberfläche mit CSG zu kombinieren. Resultat davon ist der entstandene Prototyp, welcher das Potential zeigt, die beiden Themen zu verknüpfen. Die entwickelte Applikation ermöglicht es, ohne große Umstände, Blöcke durch Klicken und Ziehen auf einer Fläche frei zu bewegen. Werden zwei Blöcke mit kompatiblen Verbindungsstücken nebeneinander gelegt, so verbinden sich diese. Bei Verbinden von Primitiv-Blöcken mit einem Operatorblock wird ein neues 3D-Modell mit der entsprechenden CSG-Operation erstellt. Durch diese Grundfunktion lassen sich geschachtelte CSG-Bäume erstellen. Die Blöcke stellen dabei selber einen CSG-Baum dar, wodurch sich jederzeit der Aufbau nachvollziehen lässt.

Wegen der blockbasierten Benutzeroberfläche, war es besonders wichtig, diese möglichst benutzerfreundlich aufzubauen. Der entwickelte Prototyp hat trotz des geringen Funktionsumfangs bereits viele, mithilfe der von Kölling und McKay aufgestellten Heuristiken, positive Eigenschaften aufgezeigt. Obwohl nur die nötigsten Funktionen implementiert wurden, ist das Gesamtpaket vom Konzept der blockbasierten Oberflächen her leicht bedienbar und veranschaulicht Constructive Solid Geometry.

Es werden allerdings sehr schnell Limitationen innerhalb des Prototyps erreicht. Die maximale Verschachtelungstiefe eines Baumes ist durch die Fenstergröße der Oberfläche begrenzt, weswegen es nicht möglich ist, unbegrenzt große Bäume zu erstellen. Betrachtet man außerdem die Transformationen der einzelnen Modelle innerhalb der 3D-Darstellung, so muss diese noch verbessert werden. Das Bearbeiten dieser ist zwar durch das Pop-up-Fenster möglich, allerdings sehr umständlich und wirkt fehl am Platz in der sonst intuitiven Umgebung.

Durch Weiterentwicklung kann die entwickelte Software durchaus einen benutzbaren Stand erreichen, der es einem ermöglicht, schnell simple und kleine Modelle zu erstellen.

7.1 Weiterentwicklung

Der Prototyp, der im Rahmen dieses Arbeit enwickelt wurde, lässt sich noch um viele weitere Facetten erweitern. Es müssten deutlich mehr helfende und erleichternde Funktionen implementiert werden. Beispielsweise müsste die Perspektive der Arbeitsfläche veränderbar sein, sodass bei Bedarf rein- und raußgezoomed werden kann. Dadurch ließe sich das Platzproblem beheben und insgesamt größere Bäume erstellen.

Auch die Kamerabewegung in der 3D-Darstellung müsste überarbeitet werden. Diese ist bisher sehr rudimentär und sollte zukünftig ebenfalls Operationen wie Verschiebung und Abstandsveränderungen unterstützen.

Bisher wird nur das Exportieren von fertigen Modellen in Wavefront OBJ Dateien unterstützt. Diese lassen sich allerdings nicht mehr in den Prototypen importieren und nach Beendigung ist die Entstehungsgeschichte verloren. Denkbar wäre hierfür ein einfaches Speichern und Laden der Arbeitsfläche, sodass innerhalb des Prototypen Modelle wieder geladen werden können. Im gleichen Sinne sind auch Vorlagen denkbar, welche immer wieder geladen werden können, um weitere Kopien zu erstellen.

Das geplante Feature mit den Transformationsmanipulatoren wurde leider aufgrund von Zeitmangel nicht implementiert, wird aber sehr dringend benötigt und würde die Bearbeitung von Modellen massiv erleichtert und zur Benutzerfreundlichkeit beitragen.

Um das Problem der verlängerten Auswertungszeiten bei großen Bäumen zu addressieren, muss die rekursive Iteration angepasst werden. Hier wäre eine einfache Markierung für alle Teilbäume denkbar, die sich in der Zwischenzeit nicht verändert haben, sodass deren CSG-Modell nicht erneut ausgewertet werden muss und direkt verwendet werden kann.

Literaturverzeichnis

[1] Blender Foundation: *Introduction to Nodes — Blender Manual*. – URL https://docs.blender.org/manual/en/2.79/render/blender_render/materials/nodes/introduction.html. – Zugriffsdatum: 2021-02-08

[2] Brown, Neil C. C. ; Kolling, Michael ; Altadmri, Amjad: Position paper: Lack of keyboard support cripples block-based programming. In: *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*, IEEE, 2015, S. 59–61. – URL http://ieeexplore.ieee.org/document/7369003/. – Zugriffsdatum: 2021-01-20. – ISBN 978-1-4673-8367-7

[3] Epic Games Inc.: *UDN - Two - BspBrushesTutorial*. – URL https://docs.unrealengine.com/udk/Two/BspBrushesTutorial.html. – Zugriffsdatum: 2021-02-04

[4] Glassner, Andrew S. (Hrsg.) ; Computing Machinery, Association for (Hrsg.): *An introduction to ray tracing*. digital print. Kaufmann, 2007. – Meeting Name: SIGGRAPH annual meeting OCLC: 838426863. – ISBN 978-0-12-286160-4

[5] Goldstein, Robert A. ; Nagel, Roger: 3-D Visual simulation. 16 (1971), Nr. 1, S. 25–31. – URL http://journals.sagepub.com/doi/10.1177/003754977101600104. – Zugriffsdatum: 2021-02-08. – ISSN 0037-5497, 1741-3133

[6] Hoffmann, Christoph M.: *Geometric and solid modeling: an introduction*. Morgan Kaufmann Publishers Inc., 1989. – ISBN 978-1-55860-067-6

[7] Kelly, James F.: *LEGO mindstorms NXT-G programming guide*. 2nd ed. Apress ; Distributed to the book trade worldwide by Springer-Verlag, 2010. – ISBN 978-1-4302-2976-6

[8] Kölling, Michael ; McKay, Fraser: Heuristic Evaluation for Novice Programming Systems. 16 (2016), Nr. 3, S. 1–30. – URL https://dl.acm.org/doi/10.1145/2872521. – Zugriffsdatum: 2021-01-20. – ISSN 1946-6226, 1946-6226

[9] Laidlaw, David H. ; Trumbore, W. B. ; Hughes, John F.: Constructive solid geometry for polyhedral objects. In: *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, Association for Computing Machinery, 1986 (SIGGRAPH '86), S. 161–170. – URL https://doi.org/10.1145/15922.15904. – Zugriffsdatum: 2020-11-02. – ISBN 978-0-89791-196-2

[10] Lee, Ghang ; Eastman, Charles M. ; Taunk, Tarang ; Ho, Chun-Heng: Usability principles and best practices for the user interface design of complex 3D architectural design and engineering tools. 68 (2010), Nr. 1, S. 90–104. – URL https://linkinghub.elsevier.com/retrieve/pii/S1071581909001451. – Zugriffsdatum: 2021-03-18. – ISSN 10715819

[11] Linietsky, Juan: *Godot Engine - Godot gets CSG support*. 2018. – URL https://godotengine.org/article/godot-gets-csg-support. – Zugriffsdatum: 2021-02-03

[12] Lu, Ping ; Jiang, Xudong ; Lu, Wei ; Ran, Wei ; Sheng, Bin: Fast, Exact and Robust Set Operations on Polyhedrons Using Localized Constructive Solid Geometry Trees. (2015)

[13] Maloney, John ; Resnick, Mitchel ; Rusk, Natalie ; Silverman, Brian ; Eastmond, Evelyn: The Scratch Programming Language and Environment. 10 (2010), Nr. 4, S. 1–15. – URL https://dl.acm.org/doi/10.1145/1868358.1868363. – Zugriffsdatum: 2021-01-07. – ISSN 1946-6226, 1946-6226

[14] Murray, James D. (Hrsg.) ; VanRyper, William (Hrsg.): *Encyclopedia of graphics file formats: the complete reference on CD-ROM with links to Internet resources; for PC, Macintosh and UNIX platforms. Buch: ...* 2. ed. O'Reilly, 1996. – OCLC: 833645820. – ISBN 978-1-56592-161-0

[15] Naylor, Bruce ; Thibault, William: Application of BSP Trees to ray-tracing and CSG evaluation. (1986)

[16] Nielsen, Jakob ; Molich, Rolf: Heuristic evaluation of user interfaces. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, Association for Computing Machinery, 1990 (CHI '90), S. 249–256. – URL https:

//doi.org/10.1145/97243.97281. – Zugriffsdatum: 2021-02-02. – ISBN 978-0-201-50932-8

[17] ROSSIGNAC, Jarek R.: Solid and Physical Modeling. (2007)

[18] ROTH, Scott D.: Ray casting for modeling solids. 18 (1982), Nr. 2, S. 109–144. – URL https://linkinghub.elsevier.com/retrieve/pii/0146664X82901691. – Zugriffsdatum: 2021-02-05. – ISSN 0146664X

[19] SEARS, Andrew: Heuristic Walkthroughs: Finding the Problems Without the Noise. (1997), S. 23

[20] WEINTROP, David ; WILENSKY, Uri: To block or not to block, that is the question: students' perceptions of blocks-based programming. In: *Proceedings of the 14th International Conference on Interaction Design and Children - IDC '15*, ACM Press, 2015, S. 199–208. – URL http://dl.acm.org/citation.cfm?doid=2771839.2771860. – Zugriffsdatum: 2020-10-28. – ISBN 978-1-4503-3590-4

[21] WEINTROP, David ; WILENSKY, Uri: Comparing Block-Based and Text-Based Programming in High School Computer Science Classrooms. 18 (2017), Nr. 1, S. 1–25. – URL https://dl.acm.org/doi/10.1145/3089799. – Zugriffsdatum: 2020-10-28. – ISSN 1946-6226, 1946-6226

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „— bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] — ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen.“

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: ____________________

Vorname: ____________________

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Constructive Solid Geometry mit einer blockbasierten Benutzeroberfläche

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

____________	____________	____________________
Ort	Datum	Unterschrift im Original