**Vorwort**

In meiner Maturaarbeit habe ich es geschafft, zwei meiner Interessen in einem Projekt zu vereinen. Das eine ist der Rubik’s Cube – ein faszinierendes farbiges Knobelspiel – und das andere ist die Informatik, die für einen Technikbegeisterten eine grosse Anziehungskraft hat.

Bereits seit vier Jahren fasziniert mich der Rubik’s Cube. Mein jüngerer Bruder erhielt damals zu Weihnachten einen solchen. Seine Begeisterung hielt sich in Grenzen. Als der Würfel erst verdreht war, verlor er das Interesse daran.

Ich versuchte mich ebenfalls mit dem neuen Spielzeug, da es mein Neugier geweckt hatte. Meine ersten Lösungsversuche scheiterten kläglich. Weil mich der Ehrgeiz gepackt hatte, suchte ich auf dem Internet nach Lösungswegen. Nach kurzer Zeit hatte ich eine Dokumentation gefunden, mit der ich den Würfel erfolgreich lösen konnte. Mit der Zeit lernte ich die Algorithmen auswendig.

Die Informatik zog mich bereits in der Volksschule in ihren Bann. Wir hatten in der sechsten Klasse einige Computer in unserem Zimmer, was an unserer Schule eine Ausnahme bildete. Die Sekundarschule bot mir die Möglichkeit, einen Informatikkurs zu absolvieren. Am Gymnasium besuchte ich das Fakultativfach Informatik und ab der Sekunda dasselbe als Ergänzungsfach.

Mein Projekt wäre niemals so erfolgreich verlaufen, wenn ich nicht von mehreren Leuten in den unterschiedlichen Phasen unterstützt worden wäre:

Mein erster Dank gilt meinem Betreuer, Tobias Bäumlin, der mich durch das ganze Projekt begleitet hat und mir immer wieder mit konstruktiven Lösungen weiterhalf.

Mein zweiter Dank widme ich Werner Randelshofer, Informatikstudent an der ETH Zürich, der mir den Einstig in die dreidimensionale Welt ermöglicht hat. Die beiden Bücher, die er mir auslieh, haben mir geholfen, die Komplexität der virtuellen Welt zu verstehen.

Ebenfalls danke ich Moritz Kobel, …, für seine nützlichen Ratschläge. Er half mir aus der Sicht des … meine Probleme zu analysieren und mit mir mögliche Lösungsansätze zu besprechen. Zusätzlich hat er mir gezeigt, wie ich mein Projekt mit GitHub [xy] organisieren kann.

**Abstract**

**Einleitung**

Das Ziel dieser Arbeit ist die Erstellung eines lauffähigen Programms in der Sprache Java, das den Rubik’s Cube in einem dreidimensionalen Raum darstellt. Dabei soll der Anwender die gleichen Drehbewegungen durchführen können wie am realen Rubik’s Cube (siehe Angang Disposition).

Die ersten Sorgen bereitete mir die Erstellung eins virtuellen dreidimensionalen Raumes. Dieses Problem löste sich jedoch schnell. Das Hauptproblem waren die unterschiedlichen Drehmöglichkeiten. Denn die meisten Steine können um mehrere Achsen gedreht werden.

Zu Beginn der Arbeit waren mir die unterschiedlichen Eigenschaften des Rubik’s Cube bereits gut bekannt. Auch die Literatur besass ich schon. Die Informatik war dagegen eher unbekannt. Obwohl ich bereits zwei Jahre in Java unterrichtet wurde, stellte sich heraus, dass ich eine ganze Bibliothek neu lernen musste.

Zeitplan erwähnen für Anhang

**Theoretischer Teil**

**Rubik’s Cube**

**Begriffe**

Um den Würfel zu beschreiben, erkläre ich in diesem Unterkapitel meine verwendeten Begriffe. Die Namen habe ich selbst ausgewählt.

**Würfel**

Rubik’s Cube mit allen seinen Elementen. Wenn es nicht explizit erwähnt wird, handelt es sich um den gelösten Rubik’s Cube. Das heisst, dass jede Wand nur eine Farbe hat.

**Wand**

Eine Seite des Würfels. Jede Seite ist dabei mit einer anderen Farbe koloriert. Normalerweise besitzt er folgende Farben: Weiss, Gelb, Rot, Orange, Blau, Grün.

**Stein**  
Er bezeichnet eine der neun Einzelteile, die jede Wand enthält. Es ist zu beachten, dass die meisten Steine in zwei oder drei Wänden gleichzeitig vorkommen.

Drehung

Der Würfel lässt sich insgesamt auf neun Arten verdrehen. Dabei lassen sich die Rotationen in drei Gruppen einteilen. Die eine Gruppe beinhaltet die Drehung von den drei horizontalen Ebenen um die y-Achse. Die zweite Gruppe beinhaltet drei Drehungen um die x-Achse und die letzte Gruppe enthält die Drehung von drei Ebenen um die z-Achse.

**Steintypen**

Jeder Steintyp kennzeichnet sich durch die gleiche Anzahl an farbigen Oberflächen. Sie können (mit Ausnahme der Mittelsteine) ihre Position untereinander austauschen.

Es gibt drei verschiedene Steintypen:

**Mittelstein**

Es handelt sich um einen Steintypen, der in der Mitte jeder Wand vorkommt. Es gibt pro Würfel insgesamt sechs.

**Kantenstein**

Es handelt sich um den Steintypen, der immer zwischen zwei Mittelsteinen liegt und zwei farbige Oberflächen hat. Es gibt insgesamt zwölf solcher Steine pro Würfel.

**Eckstein**

Es handelt sich um den Steintypen, der drei farbige Flächen besitzt. Er befindet sich, wie der Name schon sagt, in den Ecken des Würfels. Es gibt acht solcher Steine pro Würfel.

|  |
| --- |
| Abb. xy: Der gelöste Würfel |
| Bild mit Text ergänzen |
| Der gelöste Würfel hat drei verschiedene Steintypen. Jeder Steintyp besitzt andere Eigenschaften. Der Begriff Wand bezieht sich auf die komplette Seite des Würfels. |

**Funktionsweise des Würfels**

Der Aufbau des Würfels hat etwas Faszinierendes. Obwohl man jede Seite beliebig oft und in beliebiger Reihenfolge verdrehen kann, fällt er (fast) nie auseinander.

Um das Phänomen zu verstehen, muss man das Innenleben des Würfels kennen.

* Die erste wichtige Erkenntnis ist, dass sämtliche Mittelsteine miteinander verbunden sind. Dadurch behält jeder Mittelstein stets seinen gegenüberliegenden Mittelstein; das heisst der weisse gegenüber dem gelben, der rote gegenüber dem orangen und der blaue gegenüber dem grünen. Durch die Fixierung lassen sich die Mittelsteine nur um ihre eigene Achse drehen.
* Damit die Flexibilität ermöglicht wird – die garantiert, dass nach einer Drehung jede andere Drehung möglich ist – ist die innere Struktur kugelartig geformt (siehe Bild unten). Die einzelnen Steine halten sich zusammen, da sie sich gegenseitig verkannten. Sobald ein Stein fehlt, beginnt der Würfel auseinanderzufallen.

|  |
| --- |
| Abb. xy: Querschnitt durch den Würfel auf der Höhe des ersten Ebene |
| runde Innenstruktur.png |
| Durch den Querschnitt ist die kugelartige Innenkonstruktion klar zu erkennen. Sie sorgt dafür, dass die Steine sich gegenseitig zusammenhalten. Fehlt ein Stein, beginnt der Würfel auseinander zu fallen. |

**Java 3D Bibliothek**

Beim Erstellen meines Projekts habe ich vorwiegend mit der Bibliothek *javax.media.j3d* gearbeitet. Es handelt sich um eine Java-Bibliothek, die dreidimensionale Objekte erstellen und verändern kann. Zurzeit ist Java 3D noch nicht teil der *Java Fondation Class* (*JFC*).

Geschichtliche Entwicklung von Java 3D

Im Jahr 1997 begann die Entwicklung von Java 3D durch Sun Microsystems. Die Version 1.0 erschien im Jahr 1998. Die Entwicklung wurde von Sun zwischenzeitlich eingestellt. Seitdem ist die Bibliothek als Open Source dreigegeben. Zurzeit ist Version 1.5 die aktuelle Version.

(Quelle: <https://www.matse.rz.rwth-aachen.de/dienste/public/show_document.php?id=7223>)

**Aufbau der Bibliothek**

Die Bibliothek besteht aus mehreren Klassen. Zum grössten Teil gehören diese zum Klassenverband S*ceneGraphObject*. Da die Bibliothek sehr umfassend ist (siehe Anhang xy), habe ich mich entscheiden, nur die selbst verwendeten Klassen zu beschreiben.

|  |
| --- |
| Abb. xy: Klassenhierarchie der *javax.media.j3d* Bibliothek |
| verwendete Klassen in Hierarchie_1.png |
| Die Klassen haben eine klare Hierarchie. Um mit der Biblithek arbeiten zu können, ist das Verständnis der Hierarchie entscheidend. Die gelb markierten Klassen habe ich in meinem Programm selbst verwendet. Es gibt noch eine Menge weiterer Klassen, die auf der Grafik nicht zu sehen sind. (Gesamtansicht siehe Anhang xy) |

Das *VirtualUniverse* bildet einen Container, in welchen sämtliche eingesetzten *SceneGraphObjects* eingebunden werden müssen. Die meisten Programme enthalten nur ein *VirtualUniverse*, obwohl es denkbar ist, mehrere zu verwenden. Das *VirtualUniverse* stellt im Programm einen virtuellen Raum dar.

Das *Locale* definiert einen Ursprung im *VirtualUniverse* und stellt ein kartesisches Koordinatensystem dar.

Die *SceneGraphObjects* werden in zwei Hauptgruppen unterteilt: Der *NodeComponent* und der *Node*.

* Im *NodeComponent* werden Geometrie, Aussehen und Texturen des *LeafNode* spezifiziert.
  + Der *QuadArray* enthält Arrays, in welchen Koordinaten, Farben und Texturen gespeichert werden können. Dabei bilden immer vier Array-Elemente die Eckpunkte eines Vierecks.
* Der *Node* enthält Objekte, die zum Aufbau eines Programms benötigt werden. Er gliedert sich in *Group* und *Leaf* unterteilt.
  + *Group* beinhaltet *BranchGroup* und *TransformGroup*.
    - *BranchGroup* sind die einzigen Objekte, die direkt hinter einem *Locale* stehen können. Wenn eine *BranchGroup* zu einem *Locale* oder einem *SimpleUniverse* hinzugefügt wird, werden es und alle seine Child-Objekte lebendig. Durch das Entfernen der *BranchGroup* wird dieser Effekt rückgängig gemacht.
    - Alle Objekte die in der *TransformGroup* enthalten sind, können dadurch transformiert, skaliert und rotiert werden.
  + Der *Leaf* enthält *Shape3D*, welches eine oder mehrere geometrische Komponente(n) enthält und eine einzelne Komponente für das Aussehen.

*Transform3D* nimmt einen Wert auf, welcher der *TransformGroup* übergeben wird. Dadurch werden Manipulationen (Transformation, Skalierung, Rotation) der *TransformGroup* erreicht.

***Capabilities***

Sie kontrollieren den Schrib- und Lesezugriff. Solange ein *Node* nicht kompiliert oder lebendig ist, können seine Attribute verändert werden. Die *Capabilities* kontrollieren den Zugriff, sobald ein *Node* kompiliert oder lebendig ist.

Capabilities sollte sparsam eingesetzt werden, damit bei der Kompilierung mehr optimiert werden kann. Die Capabilities müssen direkt nach dem erstellen eines Objekts gesetzt werden. Sobald das Objekt kompiliert oder lebendig ist, können keine Capabilities mehr gesetzt werden.

**Aufbau des *SceneGraph* Modells**

Um ein Projekt mit der Java 3D Bibliothek zu realisieren, muss man sich an eine klare Struktur halten. Es handelt sich um den *SceneGraph*, eine hierarchische Baumstruktur. Alle Objekte, die verwendet werden, müssen dem *SceneGraph* hinzugefügt werden. Wichtig ist, dass es immer genau einen Weg vom hierarchisch höchsten Objekt bis zu jedem anderen gibt.

|  |
| --- |
| Abb. xy: Symbole für den *SceneGraph* |
| Symbole für SceneGraph.png |
| Um ein erstelltes Programm grafisch darzustellen ist es Sinnvoll für bestimmte Objekte eigene geometrische Formen zu verwenden. In meiner Arbeit werde ich jeden mit den oben gezeigten Grafiken darstellen. Zu beachten ist, dass *Group* und *Leaf* nicht ein bestimmtes Objekt meinen, sondern eine Gruppe von Objekten. |

Die Symbole auf der linken Seite stellen Objekte dar, die im *SceneGraph* Verwendung finden. Die ersten beiden bilden Objekte spezifischer Klassen. Das dritte, vierte und fünfte Objekt beinalten die *BranchGroup* / *TransformGroup* bzw. *Shape3D* bzw. *QuadArray*. Das letzte Symbol wird verwendet, um Klassen zu beschreiben, die in keine der obigen Objekte fallen.

Auf der rechten Seite sind die Symbole für die Beziehungen zwischen den Objekten dargestellt. Diese heissen *Arcs*. Die *Parent-Child*- und die *Reference*-Beziehung wird durch eine durchgezogenen bzw. gestrichelten Pfeil abgebildet.

|  |
| --- |
| Abb. xy: Beispiel für einen einfachen *SceneGraph* |
| SceneGraph Beispiel.png |
| An diesem *SceneGraph* sind sämtliche bereits erklärten Symbole vorhanden. Diese Symbole, die eine Gruppe von Objekten darstellen kann, sind weiter mit Buchstaben gekennzeichnet, damit keine Verwechslungen entstehen können. |

Das *VirtualUniverse* bildet die Wurzel des *SceneGraph*. Unterhalb des *VirtualUniverse‘s* folgt das *Locale*. Danach verzweigt sich der *SceneGraph* zum ersten Mal in zwei *BranchGroups*. Jede der beiden hat eine andere Aufgabe.

* Die *ContentBranchGroup* ist für sämtlichen Inhalt zuständig (Geometrien, Darstellung, Bewegungen, Beleuchtungen).
* Die *ViewBranchGroup* erzeugt die Ansicht (Kameraposition, Kameraausrichtung).

In der *ContentBranchGroup* können nun die Objekte eingefügt werden, die im vorherigen Kapitel beschrieben wurden. Die *Nodes* bilden dabei Knotenpunkte im *SceneGraph*. Zu beachten ist, dass *GroupNode* immer ein *Child*-Objekt hat und *LeafNode* immer am Ende eines Astes steht; mit anderen Worten, kein *Child-*Objekt hat.

|  |
| --- |
| Abb. xy: Das *SimpleUniverse* übernimmt die rechte Seite des *SceneGraph*’s |
| (Bild des SceneGraph mit SimpleUniverse) |
| Das *SimpleUniverse* erleichtert das Erstellen eines Java 3D Programmes, weil es die gesamte rechte Seite selbst erstellt. |

Da das *VirtualUniverse*, das *Locale* und die *ViewBranchGroup* in den meisten Programmen die gleichen Eigenschaften enthalten sollen, gibt es eine Hilfsklasse, die sich *SimpleUniverse* nennt. Sie erstellt die drei oben genannten Objekte und setzt Standartwerte ein. Ich habe diese Hilfsklasse verwendet und gehe deshalb nicht weiter auf ihre Objekte ein.

**Praktischer Teil**

**Vorgehen**

**Planung**

Um den Ablauf für meine Arbeit grob zu strukturieren, habe ich mich entschieden einen Zeitplan (siehe Anhang xy) zu erstellen. Die Teilarbeiten, bei denen ich den grössten Zeitaufwand vermutete, setzte ich in meine Frühlings-, Sommer- und Herbstferien 2012, da ich dort am wenigsten unvorhergesehene Ablenkungen vermutete. Die Rücksprachen mit meiner Betreuungsperson terminierte ich nach längeren Arbeiten, damit sie mir bei Schwierigkeiten aufzeigen konnte, wie ich mein Programm weiterentwickeln kann.

Die Einhaltung des Planes stellte sich jedoch schwierig heraus, da ich die zeitaufwendigen Etappen im Voraus nicht genau abschätzen konnte.

**Informationssuche**

Meine erste Informationssuche startete ich auf dem Internet. Gleichzeitig halfen mehrere Lehrpersonen allen Schülerinnen und Schüler meiner Klasse Informationen für unsere Maturaarbeiten zu suchen, indem sie uns in die Schulhaus Mediothek und die UNI-Bibliothek führten, um uns Hinweise für die Informationsbeschaffung zu geben. Ich fand weder auf dem Internet noch in den Bibliotheken geeignete Dokumente, da ich vermutlich zu diesem Zeitpunkt noch nicht nach den richtigen Quellen suchte.

Einen ersten Fortschritt erlebte ich am Besuchstag der ETH Zürich im Gymnasium Neufeld. Ich erhielt von einem Informatikstudenten den Hinweis, dass ein Studienkollege von ihm, Werner Randelshofer, sich im Rahmen seiner Masterarbeit intensiv mit dem Rubik’s Cube auseinandersetze. Nach der ersten Kontaktaufnahme lieh er mir zwei Bücher aus, welche sich mit Java und der Herausforderungen des Raumes in der Informatik beschäftigen.

Der Durchbruch gelang mir dank meiner Betreuungsperson, die mir riet, die Java 3D Bibliothek für mein Projekt zu verwenden. Um die Java 3D Bibliothek zu verstehen, beschaffte ich mir im Internet eine Tutorial-Reihe und die dazugehörige Dokumentation.

**Informationsverarbeitung**

Während meiner Arbeit habe ich mich intensiv mit dem Rubik’s Cube und der Java 3D Bibliothek auseinandergesetzt. Meine wichtigsten Erkenntnisse dazu sind im Kapitel „Theoretischer Teil“ dargestellt.

**Programmerstellung**

Bevor ich mich meinem Programm gewidmet habe, versuchte ich mich mit Beispielprogrammen einzuarbeiten. Während dieser Zeit konnte ich die gelesenen Inhalte anwenden. Ich habe eine Menge solcher Beispiele erstellt. Die für meine Arbeit am Wichtigsten möchte ich im folgenden Kapitel erklären.

**1. Programm**

Als ich mit dem ersten Programm begann, wusste ich nicht genau, wie ich Anfangen sollte. Also habe ich mit dem Teil Begonnen, den ich bereits vor Java 3D kannte: Ich erstellte ein *JFrame*, in welchem mein Programm später laufen sollte. Ich habe mich von Beginn an, dazu entschieden sämtliche Programme in einem *JFrame* zu schreiben. An dessen Grundeinstellungen habe ich nichts mehr verändert.

In das *JFrame* fügte ich ein *Canvas3D*- und ein *SimpleUniverse*-Objekt ein. Sobald ich die beiden in das *JFrame* einband, besass ich bereits die Grundlage. Das *SimpleUniverse* war jedoch noch leer. Nachdem ich eine *BranchGroup* mit einem ColorCube [xy] hinzufügte, war ich bereits zufrieden mit meinem ersten Programm.

Zu erkennen ist, dass bereits dieses einfache Programm alle wesentlichen Teile eines *SceneGraph’s* enthält.

[xy]: Der ColorCube ist aus der Hilfsklasse *com.sun.j3d.utils.geometry.* Das Objekt stellt einen Würfel dar, mit sechs unterschiedlich eingefärbten Seiten

|  |
| --- |
| Abb. xy: Erstes Programm mit der Verwendung der Java 3D Bibliothek, welches eine Seite eines Würfels zeigt. |
| Rubik's Cube Version 1.png |
| Das Programm, welches zu sehen ist, war mein erstes lauffähiges Programm in Java 3D. Es zeigt einen Würfel aus der Richtung der z-Achse. Die anderen Seiten sind nicht zu sehen, weil der Würfel parallel zum Betrachter steht. |

**2. Programm**

Im zweiten Programm habe ich zwischen den *ColorCube* und die *BranchGroup* einen weiteren *GroupNode* eingefügt. Es handelt sich um eine TransformGroup. Zusätzlich ist ein Objekt der Klasse *Transform3D* hinzugefügt worden. Mit diesen zwei Änderungen sind bereits zwei Seiten des *ColorCube* sichtbar. Zuerst hatte ich Probleme mit der Transformation, da ich den Winkel in Grad, statt im Bogenmass angegeben habe. Seither verwende ich immer die Math.toRadiant(). Um zwei Transform3D-Objekte zusammen zu rechnen wird auf das *Transform3D*-Objekt die mul(Transform3D td) angewendet.

|  |
| --- |
| Abb. xy: Zweite Java 3D Programm mit dem ColorCube |
| Rubik's CUbe Vefrsion 2.0.png |
| Nach dem hinzufügen eines Transformgroup und einer Transform3D Objekts ist der Würfel bereits als solches zu erkennen. Wichtig ist, dass der Winkel im Bodenmass angegeben wird. |

**Programm 3**

In diesem Programm habe ich noch tiefer in das *SceneGraph*-Modell eingegriffen und habe eine eigene Geometrie von der Klasse QuadArray eingefügt. Dieser Schritt war einer der wichtigsten, da er mir erlaubt, eigene Geometrien zu verwenden. Ich habe mich zuerst etwas schwer getan, da ich nicht genau wusste, welche der vielen Subklassen ich aus der GeometryArray-Klasse für mein Programm am besten geeignet ist.

|  |
| --- |
| Abb. xy: Drittes Java 3D Programm mit einer eigenen Geometrie |
| Rubik's CUbe Vefrsion 3.0.png |
|  |

Programm 4

Im vierten und letzten Programm habe ich aus mehreren QuadArray einen eigenen Würfel zusammengestellt. Hierbei ist zu beachten, dass die Punkte gegen den Uhrzeigersinn aufgerufen werden, da Java 3D immer nur die Vorderseite darstellt. Als Vorderseite wird diese Seite betrachtet von der die Reihenfolge der Punkt gegen den Uhrzeigersinn verlaufen.

|  |
| --- |
| Abb. xy: |
| Rubik's CUbe Vefrsion 4.0.png |
|  |

Erstellung des Hauptprogramms

Nach diesen Programmen habe ich mit der eigentlichen Programmierung anzufangen. In den nächsten zwei Unterkapiteln beschreibe ich zwei wichtige Lösungsansätze. Sie sind von grosser Bedeutung für mein Programmergebnis.

Erstellung der Steine

Im Kapitel „Theoretischer Teil“ habe ich bereits erklärt, dass der Würfel aus drei verschiedenen Steintypen besteht. Zu Beginn der Programmierung hatte ich die Idee drei Klassen zu erstellen. Jede davon sollte einen Steintypen repräsentieren. Im Verlauf der Arbeit habe ich mich dagegen entschieden, weil der Aufwand die Vorteile übersteigt. Stattdessen habe ich mich für eine Klasse entschieden, die alle Steine aus *QuadArrays* erstellt. Um die Steine richtig zu platzieren, hat der Konstruktor je einen Parameter für jede Raumachse (insgesamt drei Parameter). Diese Zahlen werden verwendet, um ach Punkte zu erstellen, die als Eckpunkte eines Steines dienen. Die Stone-Objekte werden in der Rubiks\_Cube Klasse erstellt. Ich habe mich dabei für ein dreidimensionales Array entschieden

|  |
| --- |
|  |
| for(int x = 0; x < stone.length; x++)  for(int y = 0; y < stone.length; y++)  for(int z = 0; z < stone.length; z++)  {  stone[x][y][z] = new Stone (x - 1, y - 1, z - 1);  } |
|  |

1. Als Erstes habe ich festgestellt, dass es nicht möglich ist einige *Stone*-Objekte in mehrere *TransformGroups* zu setzten, da die Java 3D Bibliothek nicht unterstütz, dass ein *Child*-Objekt mehr als ein *Parent*-Objekt hat. Aus diesem Grund habe ich mich entschieden jedes einzelne *Stone*-Objekt in eine eigene *TransformGroup* zu setzen. Um im Verlauf des Programms die *TransformGroup* und das *Stone*-Objekt ansprechen zu können, habe ich beide in einem Array abgelegt. Ich habe ein dreidimensionales Array gewählt, weil dadurch die Position eines Objekts schnell ermittelt werden kann.

|  |
| --- |
| Abb. xy: Erster Ansatz zur Erstellung der *Stone*-Objekte |
| for(int x = 0; x < stone.length; x++)  for(int y = 0; y < stone.length; y++)  for(int z = 0; z < stone.length; z++)  {  stone[x][y][z] = new Stone (x - 1, y - 1, z - 1);  stoneTransform[x][y][z] = new TransformGroup();  stoneTransform[x][y][z].addChild(stone[x][y][z].getTransformGroup());  objTransform.addChild(stoneTransform[x][y][z]);  } |
| Bereits bei der Erstellung der *Stone*-Objekte wird der *SceneGraph* so aufgebaut, dass ermöglicht wird, dass später Rotationen möglich sind. |

1. Kurz darauf ist mir aufgefallen, dass nach einer Rotation eines *Stones*-Objekts sein aktueller Punkt nicht als neuer Ursprung gespeichert wird. Daher konnte ich immer nur eine Drehung ausführen. Zuerst löste ich dieses Problem, indem ich den Winkel bei jeder Rotation manuell erhöhte. Später kam ich zum Ergebnis, dass ich eine neue Rotation mit den bereits ausgeführten Rotationen multiplizieren muss. Mit diesen Erkenntnissen habe ich dem Algorithmus zur Erstellung der *Stone*-Objekte ein *Transform3D*-Objekt hinzugefügt. Dieser wird auch als Array gespeichert, um in im Programm-Verlauf wieder aufzurufen.

|  |
| --- |
| Abb: xy: Überarbeiteter Algorithmus der die *Stone*-Objekte erstellt. |
| for(int x = 0; x < stone.length; x++)  for(int y = 0; y < stone.length; y++)  for(int z = 0; z < stone.length; z++)  {  stone[x][y][z] = new Stone (x - 1, y - 1, z - 1);  stoneTransform[x][y][z] = new TransformGroup();  stoneTransform[x][y][z].addChild(stone[x][y][z].getTransformGroup());  rotate[x][y][z] = new Transform3D();  objTransform.addChild(stoneTransform[x][y][z]);  } |
|  |

**Endprogramm-Dokumentation**

Ich beschreibe zuerst die Gesamtheit des Projekts. Danach gehe ich auf die einzelnen Klassen ein und beschreibe die verwendeten Objekte und Methoden. Zum Schluss erkläre ich den Zusammenhang der Klassen als Ganzes.

Ich kennzeichne Methoden, indem ich hinter ihren Namen ein rundes Klammerpaar anfüge.

**Gesamtheit des Projekts**

(Programmierstruktur)

Abb. xy: Aufbau der selbst erstellten Klassen

Mein Projekt besteht aus vier Klassen (Main, GUI, Rubiks\_Cube, Stone). Der Hauptteil der Arbeit befindet sich in der Rubiks\_Cube Klasse.

**Main-Klasse**

Die Main-Klasse bildet die Hauptklasse und den Programmstart (Entry Point). In der main() wird ein neues Objekt der GUI-Klasse erstellt. Ansonsten beinhaltet die Klasse keine weiteren Methoden.

**GUI-Klasse**

In der GUI() wird das Fenster-Gerüst aufgebaut, in welchem später der Rubik’s Cube zu sehen ist. In der createPanel() wird ein Objekt initialisiert, welches darauf wartet, dass ein Knopf vom Benutzer gedrückt wird. Sobald ein Knopf gedrückt wird, ermittelt *actionPerformed(ActionEvent e)*, um welchen Knopf es sich handelt und ruft dementsprechend eine andere Methode in der Rubiks\_Cube-Klasse auf.

Drei Knöpfe

**Rubiks\_Cube-Klasse**

Die Klasse hat zwei Aufgaben. Zum einen erstellt sie mit createViewBranchGraph() die Ansicht auf den Rubik’s Cube, zum anderen wird der Inhalt des Programms generiert.

Die erste Aufgabe wird in der createViewBranchGraph() ausgeführt; die Standard-Konfiguration eingestellt und die Kameradistanz zum Würfel festgelegt.

Die zweite Aufgabe wird in der createContentBranchGraph() erledigt. Der Würfel wird durch die getStart() gedreht, damit zu Beginn der Applikation drei Seiten sichtbar sind. Danach fügt der Aufruf der createMouseBehavior() dem SceneGraph ein neues Objekt hinzu, das dem Endnutzer erlaubt, die Kameraposition um den Rubik’s Cube zu bewegen. Die nächste Schleifengruppe fügt der Szene die Bestandteile des Rubik’s Cube hinzu. Die einzelnen Steine werden, bevor sie in den SceneGraph eingefügt werden, noch weiter verschachtelt. Die Verschachtelung ermöglicht, dass sie rotieren können. Zum Schluss des Aufbaus werden mit der createCoordSystem() der Szene noch drei Linien hinzugefügt, die das Koordinatensystem visuell darstellen.

Weiter werden die Interaktionen aus der GUI-Klasse verarbeitet. Dies ist der Fall, wenn der Endbenutzer einer der drei Knöpfe drückt, die sich am oberen Bildschirmrand des Programms befinden. Alle drei Methoden haben die Aufgabe eine Seite zu rotieren. Je nachdem welcher Knopf gedrückt wird, wird die Methode aufgerufen, welche die entsprechende Seite dreht. Die drei Methoden führen ähnliche Befehle aus.

Wenn die Methode aufgerufen wird, werden alle Objekte, die für die Rotation benötigt werden, erstellt und danach auf die getTransform() zurückgegriffen.

Wenn die Methode aufgerufen wird, dreht sie eine Seite um eine Koordinatenachse. Da jedes Stein-Objekt sein eigenes Koordinatensystem hat, muss nicht jeder Stein um die gleiche Achse gedreht werden. Deshalb wird jeder Stein einzeln gedreht. Danach wird die changeArrayPosition() aufgerufen, welche die Steine im Array ihrer neuen Position zuweist.

**Stone-Klasse**

In dieser Klasse werden in der *createGeometry()* aus den drei übergebenen Parametern acht Punkte berechnet. Jeder Punkt bezeichnet eine der Ecken, die jeder Stein hat. Sechs *QuadArrays* erstellen anhand der Punkte je ein Quadrat. Jedem Quadrat, das später an der Oberfläche des Würfels zu sehen sein wird, wird in der *createAppearnace()* eine Farbe zugewiesen. Alle Quadrate werden in einem Shape3D-Objekt zu einem dreidimensionalen Stein zusammengefügt und der Rubiks\_Cube-Klasse durch die *getTransformGroup()* zurückgegeben.

Die weiteren Methoden werden für die Interaktion des Endbenutzers benötigt.

Endbenutzerdokumentation

Diskussion/ Resultat

(Wie bin ich auf die Lösung gekommen, Weshalb kein andere?)

Das erste Programm hat einen Würfel gezeigt.

**Quellen**

**Zwei Bücher von Werner Randelshofer**

Bell, Douglas und Parr, Mike. 2002. Java for Students. Person Education Limited. Harlow.

Plenge Axel. Amiga 3D-Grafik und Animation. 1988. Markt & Tech Verlag. Haar bei München

**Bilder für theoretischer Teil „Rubik’s Cube“**

selbst erstellt mit dem Programm „Google SketchUp“ Version 8

**Text für theoretischer Teil Kapitel „Java 3D Bibliothek“**

Java 3D API Version 1.5.0:

[download.java.net/media/java3d/javadoc/1.5.0/index.html](http://download.java.net/media/java3d/javadoc/1.5.0/index.html) (Stand 01.10.2012)

Java Tutorial 1

[cs.stir.ac.uk/courses/ITNP3B/Java3D/Tutorial/j3d\_tutorial\_ch1.pdf](http://www.cs.stir.ac.uk/courses/ITNP3B/Java3D/Tutorial/j3d_tutorial_ch1.pdf) (Stand 01.10.12)