Bachelorarbeit Technische Informatik

Eine Zauberwürfel-Applikation unter Android mit OpenGL ES

vorgelegt von

Jason Oltzen

Erstgutachter: Prof. Dr. Carsten Vogt (Technische Hochschule Köln)

Zweitgutachter: Prof. Dr. René Wörzberger (Technische Hochschule Köln)

Dezember 2022

Fakultät für
Informations-, Medien-
und Elektrotechnik

Technology
Arts Sciences
TH Köln

Bachelorarbeit

Titel: Eine Zauberwürfel-Applikation unter Android mit OpenGL ES

Gutachter:

- Prof. Dr. Carsten Vogt (TH Köln)
- Prof. Dr. René Wörzberger (TH Köln)

Zusammenfassung: Das Ziel dieser Bachelorarbeit war es, eine Zauberwürfel-Applikation auf der Basis einer bereits existierenden Android Open GL-Bibliothek zu erstellen. Die Zauberwürfel-Applikation implementiert sowohl alle Funktionen, die einem Benutzer bei einem realen physischen Zauberwürfel zur Verfügung stehen, als auch weitere Zusatzfunktionen, wie zum Beispiel das Zurücksetzen einzelner Schritte sowie das automatische Vermischen des Zauberwürfels.

Stichwörter: Zauberwürfel, Android, OpenGL ES

Datum: 22.12.2022

Bachelors Thesis

Title: A Magic Cube application under Android with OpenGL ES

Reviewers:

- Prof. Dr. Carsten Vogt (TH Köln)
- Prof. Dr. René Wörzberger (TH Köln)

Abstract: The aim of this bachelor thesis was to create a Magic Cube application based on an existing Android Open GL library. The Magic Cube application implements all functions of a real physical Magic Cube, as well as additional functions such as resetting individual steps and automatically scrambling the Magic Cube.

Keywords: Magic Cube, Android, OpenGL ES

Date: 22.12.2022

Inhaltsverzeichnis

1 Einleitung

Der Zauberwürfel, auch bekannt als Rubik's Cube, wurde 1974 von dem ungarischen Bauingenieur und Professor der Architektur Ernő Rubik erfunden und sollte ursprünglich das räumliche Denk- und Vorstellungsvermögen seiner Studenten verbessern. Mit diesem Ziel erbaute er den allerersten Zauberwürfel aus 27 kleinen Holzblöcken und beklebte die einzelnen Segmente mit farbigen Papierquadraten. Nach einigen Drehversuchen merkte Ernő Rubik, dass er, ohne es zu wissen, mehr als nur ein Lehrmittel erfunden hatte, sondern vielmehr eine neue Art von Knobelspielen. Seit dem Produktionsbeginn Anfang der 1980er-Jahre gehört der Zauberwürfel zu den beliebtesten Puzzlespielen weltweit und gewann 1980 den Sonderpreis als „Bestes Solitärspiel". [StRot22]

Zahlreiche Menschen auf der Welt versuchen seither den Zauberwürfel zu lösen, indem der Zauberwürfel wieder in seinen ursprünglichen Ausgangszustand zurückgesetzt wird. Seit 2004 veranstaltet die „World Cube Association" einen jährlichen Wettbewerb, bei dem versucht wird, den aktuellen Weltrekord für das Lösen des Zauberwürfels zu schlagen. Der aktuelle Rekord (Stand November 2022) liegt bei 3,47 Sekunden und wurde am 24. November 2018 von dem chinesischen „Speedcuber" Yusheng Du aufgestellt. [StRot22]

Unter „Speedcubing" versteht man die Tätigkeit, den Zauberwürfel in möglichst kurzer Zeit zu lösen. Während der klassische Zauberwürfel eine Matrix von 3x3x3 besitzt, wurden im Laufe der Zeit verschiedene Variationen und Abwandlungen herausgebracht, wobei die offiziell zu lösenden Varianten von dem 2x2x2 Würfel bis hin zu dem 7x7x7 Würfel reichen. [StRot22]

1.1 Zielsetzung

Das Ziel dieser Bachelorarbeit ist die Erstellung einer mobilen Zauberwürfel-Applikation in Android mit OpenGL ES unter Verwendung einer bereitgestellten Android OpenGL ES-Bibliothek, mit dem Ziel, diese weiterzuentwickeln.

Der Benutzer soll mithilfe eines mobilen Endgerätes einen virtuell simulierten traditionellen 3x3x3 Zauberwürfel bedienen können. Hierbei liegt der Hauptfokus auf der Rotation des kompletten Zauberwürfels um seine eigenen Achsen sowie dem

Verdrehen einzelner Ebenen des Zauberwürfels, welche das Vermischen oder Lösen eines Zauberwürfels erreichen soll. Zusätzlich sollen dem Benutzer weitere Funktionen zur Verfügung gestellt werden, um der Zauberwürfel-Applikation einen unterhaltsamen Nutzen hinzuzufügen.

Bei der Ermittlung möglicher Anforderungen an die Implementierung werden außerdem mögliche Anwendungsfälle entwickelt.

Die Benutzung der Zauberwürfel-Applikation soll ausschließlich über den Bildschirm des mobilen Endgerätes erfolgen, sodass der Benutzer keinerlei zusätzliche Hardware für die Nutzung der Applikation benötigt.

1.2 Aufbau der Arbeit

Zu Beginn dieser Abschlussarbeit wird im zweiten Kapitel die Grundstruktur des Zauberwürfels näher erläutert, da dieses Verständnis einen wichtigen Grundstein für das Verständnis der Implementierung darstellt.

Im dritten Kapitel werden die genauen Anforderungen für die Implementierung der Zauberwürfel-Applikation aufgezeigt und die daraus resultierenden Anwendungsfälle werden mit entsprechenden Anwendungsfalldiagrammen und Anwendungsfalltabellen detaillierter untermauert.

Im Anschluss an die Erstellung der spezifischen Anforderungen wird, basierend auf den aus Kapitel 3 erlangten Erkenntnissen, ein erster Entwurf der Benutzeroberfläche dargestellt, auf den im vierten Kapitel näher eingegangen wird.

Auf der Basis der Anforderungen sowie des Benutzeroberflächenentwurfs wird im fünften Kapitel die schrittweise Umsetzung der Implementierung der Zauberwürfel-Applikation unter Android OpenGL ES erläutert.

Am Ende dieser Abschlussarbeit werden im sechsten Kapitel unter anderem die Ergebnisse dieser Arbeit zusammengefasst und es werden zusätzlich mögliche Erweiterungsmöglichkeiten beschrieben, die in dieser Arbeit jedoch nicht umgesetzt wurden.

2 Grundlagen

2.1 Der Zauberwürfel

Bei der Beschreibung der grundlegenden Struktur eines klassischen Zauberwürfels liegt der Fokus auf der richtigen Notation und dem Aufbau der einzelnen Rotationen eines Zauberwürfels. Mithilfe des in diesem Kapitel beschriebenen Vorwissens kann die Implementation eines physischen Zauberwürfels in eine virtuell simulierte Anwendung nachvollzogen und somit korrekt umgesetzt werden.

2.1.1 Struktur des Zauberwürfels

Der klassische 3x3x3-Zauberwürfel besteht aus insgesamt sechs Seiten mit verschiedenen festgelegten Farben: rot, orange, grün, blau, gelb und weiß. Der hier zur Demonstration verwendete Zauberwürfel besteht aus der traditionellen Farbanordnung, bei der die rote Seite gegenüber von der orangenen, die gelbe gegenüber von der weißen und die blaue gegenüber von der grünen Seite liegt. Der Zauberwürfel ist zusätzlich in neun verschiedene Ebenen unterteilt, die um 90, 180, 270 und 360 Grad gedreht und somit verstellt werden können. Durch das wiederholte Drehen mehrerer unterschiedlicher Ebenen kann der Zauberwürfel vermischt werden (Abbildung 1). [ZaWi22]

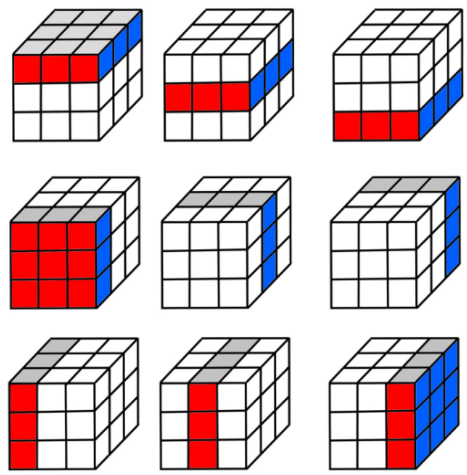

Abbildung 1: Hier sind die einzelnen Ebenen des Zauberwürfels dargestellt. Die Ebenen gehen von links oben nach rechts unten in der folgenden Reihenfolge: oben, mittlere Y-Ebene, unten, vorne, mittlere Z-Ebene, hinten, links, mittlere X-Ebene, rechts.

Ein klassischer 3x3x3-Zauberwürfel besteht aus insgesamt 27 einzelnen Steinen, wobei bei einem physischen Zauberwürfel nur 26 Steine zu sehen sind, da sich an der Stelle des 27. Steines eine Diagonalachse befindet. Diese 26 Steine werden in drei verschiedene Arten von Steinen unterteilt: Mittelsteine, Ecksteine und Kantensteine (Abbildung 2). [ZaAn22]

Die sechs zentralen Mittelsteine des Zauberwürfels sind mit einer unbeweglichen Diagonalachse verbunden und können nicht verdreht werden. Dadurch wird sichergestellt, dass sich die Grundstruktur des Zauberwürfels nach dem Mischen nicht verändert und jede Seite ihre feste Position behält. Diese Steine in der Mitte markieren somit die Farbe der zu lösenden Seite und sichern die Grundstruktur des Zauberwürfels. [ZaAn22]

Die restlichen 20 Steine werden in zwölf Ecksteine und acht Kantensteine aufgeteilt. Diese Steine wechseln nach jeder Rotation einer Ebene ihre Position und werden durch gleichartige Steine ersetzt. Generell kann jeder Mittelstein mit einem Mittelstein, jeder Eckstein mit einem Eckstein und jeder Kantenstein mit einem Kantenstein getauscht werden, andere Tauschmöglichkeiten sind ausgeschlossen. [ZaAn22]

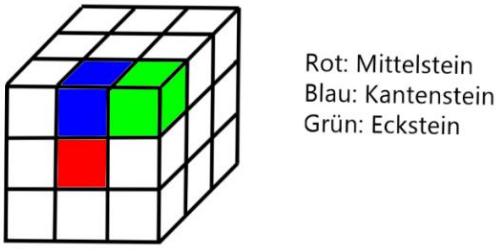

Abbildung 2: Darstellung der drei verschiedenen Steine in einem Zauberwürfel

Jede Seite des Zauberwürfels besteht aus neun Quadraten mit der entsprechenden Farbe. Der in dieser Arbeit relevante 3x3x3-Zauberwürfel besteht aus insgesamt 54 verschiedenen Quadraten, die durch Aufkleber ihre Farbe repräsentieren.

Der Zauberwürfel kann prinzipiell in jedem Zustand gelöst werden, was bedeutet, dass der ursprüngliche Zustand des Zauberwürfels immer wiederhergestellt werden kann. Voraussetzung dafür ist, dass die physische Form des Zauberwürfels nicht verändert wird, indem zum Beispiel einzelne Steine herausgenommen oder

einzelne Farbaufkleber abgelöst werden, ohne dass diese wieder in die korrekte Position zurückgesetzt oder womöglich sogar vertauscht werden.

In Abbildung 3 wird dieser Unterschied noch einmal bildlich verdeutlicht. Links ist ein Zauberwürfel abgebildet, bei dem nur noch die richtige Zugkombination fehlt, um die letzten beiden Ecksteinen an ihren Ursprungsplatz zu befördern.

Sollte der Benutzer zuvor jedoch einen Stein rotieren, ohne diese Rotation durch das Verdrehen der Ebenen zu erzielen, entsteht ein sogenannter nicht lösbarer Zustand des Zauberwürfels, der auf der rechten Seite der Abbildung 3 dargestellt wird. Statt einer Rotation der Ebenen, wurde hier einer der Ecksteine (rot, gelb, grün) physisch verdreht, wodurch der Zauberwürfel in diesem Zustand nicht mehr gelöst werden kann.

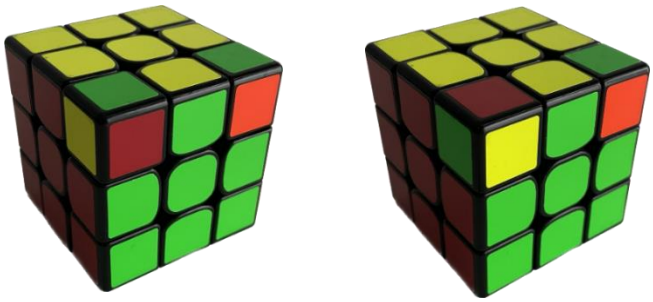

Abbildung 3: (links) Lösbarer Zauberwürfel, (rechts) Unlösbarer Zauberwürfel

2.1.2 Notationen

Unter dem Begriff der Notation versteht man in Bezug auf den Zauberwürfel die Zuweisung eines bestimmten Buchstabens für jede 90-Grad-Drehung einer Ebene. Mit den Notationen lassen sich mehrere Züge zu einer Zugkombination, einem sogenannten Algorithmus, zusammenstellen. Diese Zugkombinationen werden häufig beim Erlernen einer Lösungsmethode verwendet. [ZaWi22]

Auch bei Wettbewerben werden die Zauberwürfel nach einer bestimmten Zugkombination vermischt, um sicherzustellen, dass auch tatsächlich alle Zauberwürfel entsprechend vermischt sind und ausreichend Züge zum Lösen erfordern und nicht wie im wenigsten komplexen Fall nur zwei Züge dafür benötigen. [ZaWi22]

Um den Zauberwürfel lösen zu können, muss der Benutzer wissen, welche Zugkombination angewendet werden muss, um einen oder mehrere Steine in die richtige Position zu bringen. Um das Lösen des Zauberwürfels zu erlernen, gibt es verschiedene Lösungsansätze. Bei jeder dieser Lösungsmethoden muss der Benutzer die algorithmischen Folgen in der richtigen Kombination ausführen. Dabei kann der Benutzer entweder die Zugkombination auswendig lernen oder mithilfe einer ausführlichen Anleitung versuchen, den Zauberwürfel zu lösen.

In der folgenden Tabelle ist die Standardnotation für einen 3x3x3-Zauberwürfel abgebildet. Bei einem physischen Würfel ist es zu Beginn nicht von Bedeutung, welcher Farbe beziehungsweise welcher Ebene die entsprechende Notation zugewiesen ist, jedoch orientiert sich diese Arbeit an den zu sehenden Farbzuweisungen in der vierten Spalte. [ZaWi22]

Notation	**Englisch**	**Deutsch**	**Farbe**
F	front	vorne	rot
B	back	hinten	orange
L	left	links	grün
R	right	rechts	blau
U	up	oben	weiß
D	down	unten	gelb

Tabelle 1: Standardnotationen für die Drehungen der einzelnen Ebenen des Zauberwürfels um 90 Grad im Uhrzeigersinn

Mit den angegebenen Notationen besteht nun die Möglichkeit, eine erste Zugkombination zusammenzustellen. Allerdings notieren diese Notationen nur die Drehungen der Ebenen um 90 Grad im Uhrzeigersinn. Die Ebenen-Rotation um 90 Grad gegen den Uhrzeigersinn wird in der Regel mit demselben und einem anhängenden Apostrophe notiert. Zum Beispiel würde ein (F) eine 90-Grad-Drehung im Uhrzeigersinn und ein (F‘) eine 90-Grad-Drehung gegen den Uhrzeigersinn beschreiben. [RoFr12]

Eine weitere Notationsschreibweise gibt es für mehrere Bewegungen einer Ebene hintereinander. Wenn zum Beispiel die vordere Ebene dreimal hintereinander um 90 Grad im Uhrzeigersinn gedreht werden soll, gibt es hierfür drei Möglichkei-

ten dies zu symbolisieren. Entweder wird die Notation einfach als drei Buchstaben hintereinandergeschrieben (FFF), als Potenzzahl (F^3) oder die Zahl der auszuführenden Drehungen hinter die Notation der zu rotierenden Ebene (F3). Alle drei Fälle indizieren dasselbe, und der Benutzer weiß somit welche Aktion er ausführen muss, um den Algorithmus korrekt auszuführen. In der Regel wird diese Notationsart jedoch nicht bei mehr als zwei Rotationen hintereinander notiert. Der Grund hierfür ist, dass es in den Fällen, wo eine Ebene mehr als zwei Mal hintereinander in dieselbe Richtung gedreht werden soll, durch eine andere Notierung vereinfacht beschrieben werden kann.
Bei dem oben beschriebenen Beispiel wäre die Notation vereinfacht (F3 = F‘), da eine dreifache Rotation einer Ebene nur eine einzige Drehung um 90 Grad gegen den Uhrzeigersinn bedeuten würde. Weiterhin würde die Notation (F4) bedeuten, dass die Ebene an den Anfangszustand vor dem Ausführen der Aktion zurückgekehrt wäre und somit ist diese Notationsschreibweise überflüssig und kann weggelassen werden. [RoFr12]

Zusammengefasst lässt sich sagen, dass drei verschiedene Notationsarten in einer Zugkombination zu finden sind: groß geschriebener Buchstabe (zum Beispiel: F) = 90-Grad-Drehung einer Ebene im Uhrzeigersinn; groß geschriebener Buchstabe mit Apostroph (zum Beispiel F‘) = 90-Grad-Drehung einer Ebene gegen den Uhrzeigersinn; und ein Buchstabe mit einer angehängten zwei (zum Beispiel F2) = zweifache Drehung einer Ebene.

Aus den gelernten Notationen können jetzt beliebig viele Zugkombinationen oder Algorithmen zusammengestellt werden. Ein Beispiel für eine solche Zugkombination mit allen drei Notationsarten wäre: (R F2 U L R‘ B F‘ R).

Zusätzlich zu den Notationen für die Drehungen der Ebenen gibt es auch die Notationen „X“, „Y“ und „Z“ für die Drehungen des kompletten Würfels. „X“ ist die Rotation des Würfels in dieselbe Richtung wie (R); „Y“ in die Richtung von (U) und „Z“ in die Richtung von (F). Das Endstadium der Drehungen des kompletten Würfels wird in Abbildung 4 dargestellt. Obwohl diese Rotationen in der offiziellen Notierung des Zauberwürfels existieren, werden sie selten in den Algorithmen berücksichtigt. Ursächlich dafür ist das angestrebte Ziel, dass der Benutzer den Zauberwürfel so wenig wie möglich um seine eigene Achse bewegen und ihn am besten immer in der gleichen Position halten soll. [RoFr12]

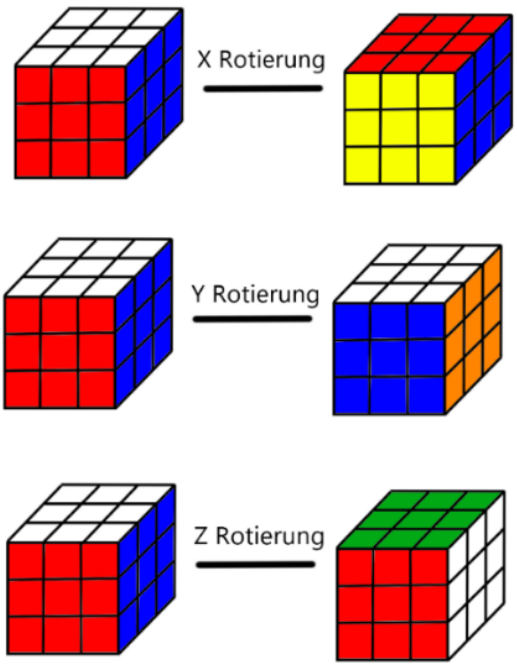

Abbildung 4: Rotationen des kompletten Zauberwürfels

2.2 Android

Android ist eine freie Open-Source-Software-Plattform, die primär in mobilen Endgeräten wie zum Beispiel Smartphones und Tablets zum Einsatz kommt. Das Android System wurde 2007 von der Open Handset Alliance gegründet und stand seither unter der Leitung von der Google LLC. Das erste mobile Endgerät, welches mit dem Android-Betriebssystem ausgestattet war, wurde 2008 auf den Markt gebracht. [KüTh21a]

Seit der Veröffentlichung des Android-Betriebssystems im September 2008 hat sich sowohl die Software-Plattform als auch ihr primäres Einsatzgebiet stark ausgebreitet. In der heutigen Zeit findet man weitaus mehr Endgeräte, die mit einem Android-Betriebssystem ausgestattet sind. Dazu gehören vor allem Fernseher (Android TV), Smartwatches (Android Wear OS by Google), Autos (Android Automotive) und Notebooks. [KüTh21a]

2.2.1 OpenGL ES

OpenGL für Eingebettete Systeme (englisch: OpenGL for Embedded Systems, kurz: OpenGL ES) ist ein 3D-Grafikstandard, der 1992 von der Firma Silicon Graphics entwickelt wurde und auf der OpenGL-Bibliothek basiert. OpenGL ES wird in weiten Bereichen der Branche verwendet, von mobilen Endgeräten, wie zum Beispiel Smartphones, Tablets und Videospielkonsolen, bis hin zu den Supercomputern der NASA, um beispielsweise Strömungssimulationen auszuführen. [ProGL12]

Zum Zeitpunkt der Entwicklung der Zauberwürfel-Applikation (Oktober 2022) existieren fünf Versionen von Android OpenGL ES. Die Versionen „OpenGL ES 1.0“ und „1.1“ unterstützen alle Android-Versionen ab Android 1.0. „OpenGL ES 2.0“ unterstützt alle Android-Versionen ab Android 2.2 ab dem API-Level 8. „OpenGL ES 3.0“ unterstützt alle Android-Versionen ab Android 4.3 ab dem API-Level 18 und die letzte und auch aktuellste Version „OpenGL ES 3.1“ unterstützt alle Android-Versionen ab Android 5.0 ab dem API-Level 21. [AnDe22]

Da die für diese Abschlussarbeit bereitgestellte Android OpenGL ES-Bibliothek auf der OpenGL ES Version 2.0 basiert, wird bei der Implementierung auch diese Version verwendet.

3 Anforderungsanalyse

In diesem Kapitel werden die Anwendungsfälle der zu erstellenden Applikation vorgestellt und im Anschluss analysiert. Hierdurch soll ein erster Überblick über den Programmablauf erschaffen werden und erste Überlegungen für die graphische Benutzeroberfläche entstehen.

3.1 Anwendungseinsatz

Die Zauberwürfel-Applikation soll grundsätzlich für jeden Benutzer entwickelt werden, der über ein mobiles Endgerät mit dem Android-Betriebssystem verfügt. Es ist nicht erforderlich, zusätzliche Selektionskriterien für die Anwendungsnutzung anzugeben. Die Anwendung soll hauptsächlich dem Unterhaltungszweck des Benutzers dienen und sollte keine zusätzlichen Vorkenntnisse oder Erfahrungen für die reine Verwendung der Zauberwürfel-Applikation voraussetzen.

3.2 Anwendungsfunktionalität

Im Folgenden werden einige Grundfunktionen vorgestellt, die für die Zauberwürfel-Applikation implementiert werden sollen. In Abbildung 5 ist ein typischer Anwendungsfall dargestellt, der bei der Verwendung der Zauberwürfel-Applikation auftritt.

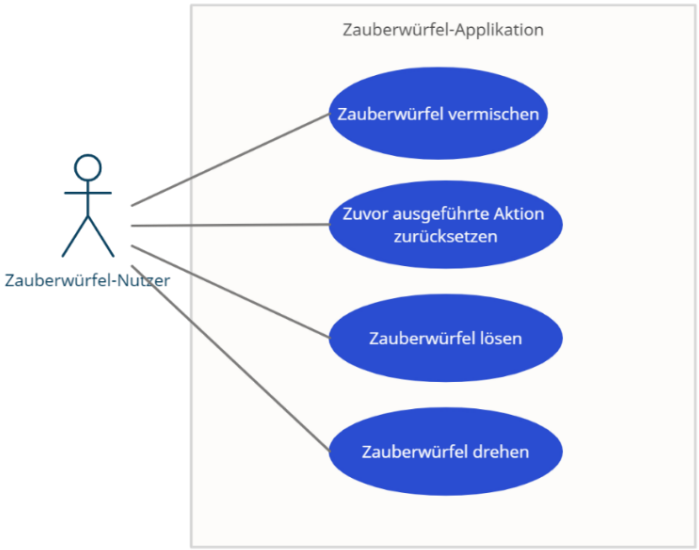

Abbildung 5: Anwendungsfall-Diagramm - Allgemeine Funktionsübersicht

Der einzige Akteur in der Zauberwürfel-Applikation ist der Benutzer selbst. Gemäß den Anforderungen kann der Benutzer die folgenden Anwendungsfälle aus der Zauberwürfel-Applikation heraus auslösen:

1. Rotieren des kompletten Zauberwürfels
2. Rotieren einer einzelnen Ebene des Zauberwürfels
3. Zurücksetzen des Zauberwürfels
4. Zufälliges automatisches Vermischen des Zauberwürfels
5. Schrittweises Zurücksetzen von zuvor ausgeführten Aktionen

In den folgenden Abschnitten werden die aufgezählten Anwendungsfälle im Einzelnen beschrieben.

3.2.1 Rotieren des kompletten Zauberwürfels

Der Anwendungsfall zum Rotieren des kompletten Zauberwürfels soll dem Benutzer ermöglichen, den Zauberwürfel, um seine eigenen Achsen drehen und somit diesen von allen Seiten betrachten zu können. Diese Funktion ist für die korrekte Verwendung der Applikation unerlässlich, da der Benutzer ohne diese Option nicht in der Lage wäre, die aktuelle genaue Position zu lokalisieren, an der sich der ausgewählte Zauberwürfelstein momentan befindet. Er wüsste daher nicht, welche Drehbewegung(en) der Ebene(n) er als nächstes ausführen müsste, um den Zauberwürfelstein an die gewünschte Position zu bringen.

Die Abbildung 6, im Zusammenhang mit der Tabelle 2, stellt den Anwendungsfall detailliert dar. Dieser Anwendungsfall soll ausgelöst werden, sobald der Benutzer den Bildschirm des mobilen Endgerätes berührt und weder eine berührungsempfindliche Schaltfläche noch der virtuell simulierte Zauberwürfel selbst berührt wurden.

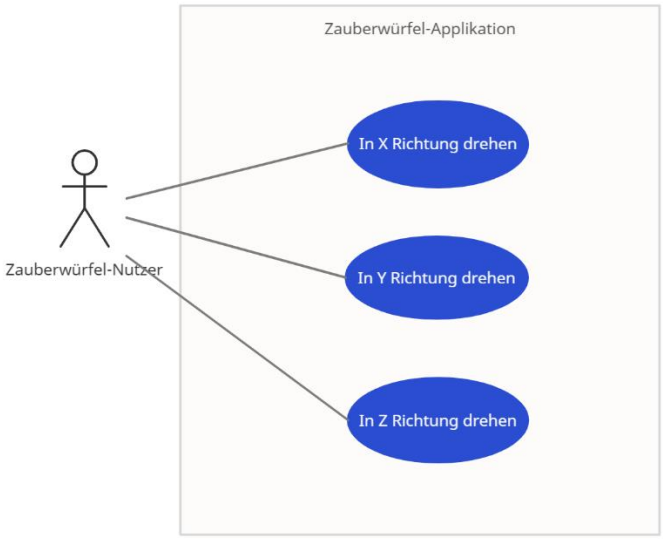

Abbildung 6: Anwendungsfall-Diagramm - Rotieren des kompletten Zauberwürfels.

Anwendungsfall	Rotieren des kompletten Zauberwürfels
Beteiligte Akteure	Zauberwürfel-Nutzer
Kurzbeschreibung	Der Benutzer dreht den Zauberwürfel um seine eigenen Achsen.
Auslöser	Berühren des Bildschirmes unter der Bedingung, dass weder der Zauberwürfel selbst noch eine berührungsempfindliche Schaltfläche zu Beginn berührt wurden.
Vorbedingung	Applikation wurde gestartet
Eingehende Informationen	Berührungspunkte des Bildschirmes
Standardablauf	1. Bildschirm wird an einer Position berührt 2. Prüfung, ob eine andere Aktion vom Benutzer gewollt ist 3. Benutzer führt entsprechende Wischbewegungen aus 4. Der Zauberwürfel folgt der Wischbewegung des Benutzers und dreht sich somit um die eigenen Achsen
Ergebnis	Die aktuelle Rotationsposition und der aktuelle Rotationswinkel werden abgespeichert.

Tabelle 2: Anwendungsfall - Rotieren des kompletten Zauberwürfels

3.2.2 Rotieren einer einzelnen Ebene des Zauberwürfels

Um die beiden Anwendungsfälle *„3.2.1 Rotieren des kompletten Zauberwürfels“* und *„3.2.2 Rotieren einer einzelnen Ebene des Zauberwürfels“* voneinander zu separieren, muss eine Fallunterscheidung der beiden Funktionen definiert werden. Der Benutzer soll mit dieser Funktionalität die Möglichkeit haben, die einzelnen Ebenen des Zauberwürfels durch entsprechende Berührungs- und Wischaktionen in die gewünschte Richtung zu verdrehen. Durch das wiederholte Ausführen dieser Funktionalität soll entweder das manuelle Vermischen oder das manuelle Lösen des Zauberwürfels ermöglicht werden.

Die Abbildung 7 im Zusammenhang mit der Tabelle 3 beschreibt den genannten Anwendungsfall detaillierter. Der Anwendungsfall soll ausgelöst werden, sobald der Benutzer den virtuell simulierten Zauberwürfel berührt. Auch hier ist die Ausgangsposition der Berührung ausschlaggebend für die Ausführung der Funktion. Sobald der Bildschirm vom Benutzer losgelassen wird, wird die Ebene in die gewünschte Richtung gedreht. In dem Anwendungsfall hat der Benutzer die Wahl zwischen zwei verschiedenen möglichen Richtungen, in die die Ebene des Zauberwürfels gedreht werden kann.

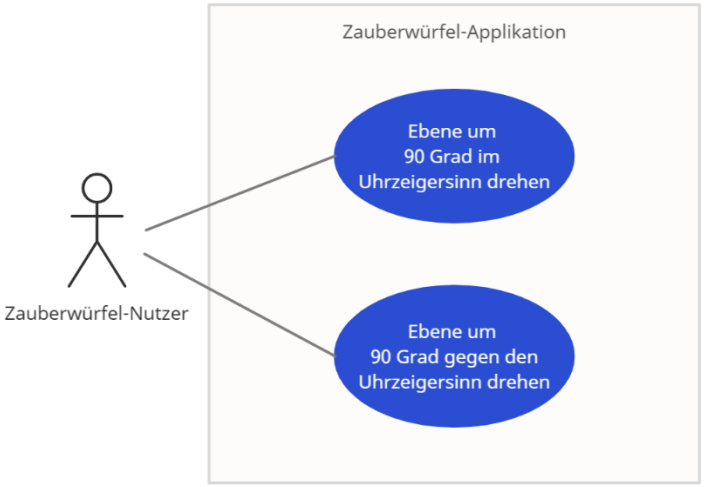

Abbildung 7: Anwendungsfall-Diagramm - Rotieren einer einzelnen Ebene des Zauberwürfels

Anwendungsfall	Rotieren einer einzelnen Ebene des Zauberwürfels
Beteiligte Akteure	Zauberwürfel-Nutzer
Kurzbeschreibung	Der Benutzer dreht die berührte Ebene in die gewünschte Richtung um 90 Grad im oder gegen den Uhrzeigersinn.
Auslöser	Berühren des Bildschirmes unter der Bedingung, dass der Zauberwürfel selbst zu Beginn berührt wurde.
Vorbedingung	Applikation wurde gestartet
Eingehende Informationen	Berührungspunkte des Bildschirmes
Standardablauf	1. Bildschirm wird an einer Position berührt 2. Prüfung, ob eine andere Aktion vom Benutzer gewollt ist 3. Benutzer führt entsprechende Wischbewegungen aus 4. Die Ebene folgt der Wischbewegung des Benutzers
Ergebnis	Der Zauberwürfel wird vermischt oder gelöst.

Tabelle 3. Anwendungsfall - Rotieren einer einzelnen Ebene des Zauberwürfels

Das Anwendungsfalldiagramm in Abbildung 8 ist identisch für die letzten drei Anwendungsfälle: *„3.2.3 Zufälliges automatisches Vermischen des Zauberwürfels“*, *„3.2.4 Schrittweises Zurücksetzen von zuvor ausgeführten Aktionen“* und *„3.2.5 Zurücksetzen des Zauberwürfels“*. Dieses Anwendungsfalldiagramm stellt die drei Anwendungsfälle in detaillierter Form dar.

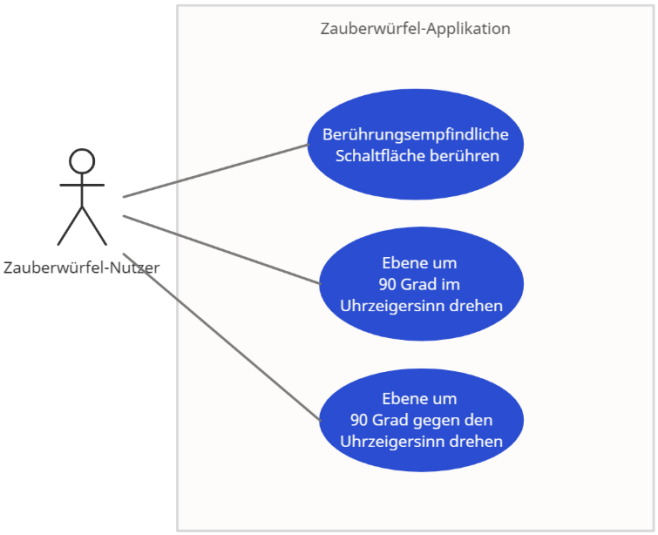

Abbildung 8: Anwendungsfall-Diagramm - "Zurücksetzen des Zauberwürfels", "Zufälliges automatisches Vermischen des Zauberwürfels" und "Schrittweises Zurücksetzen von zuvor ausgeführten Aktionen"

3.2.3 Zufälliges automatisches Vermischen des Zauberwürfels

Bei diesem Anwendungsfall soll der Benutzer die Möglichkeit haben, das manuelle Mischen des Zauberwürfels mithilfe einer berührungsempfindlichen Schaltfläche zu überspringen. Eine Ebene des Zauberwürfels soll sich nach dem Ausführen dieser Funktion durch zufällig generierte Parameter verdrehen. Nachdem die Funktion beendet wurde, kann der Benutzer den Zauberwürfel manuell lösen.

Der Anwendungsfall wird ausgelöst, sobald der Benutzer den Bildschirm berührt hat und das Programm erkennt, dass die entsprechende berührungsempfindliche Schaltfläche vom Benutzer berührt wurde. Der Zauberwürfel wird durch das Ausführen des Anwendungsfalles automatisch nach dem Zufallsprinzip gemischt, sobald der Bildschirm vom Benutzer losgelassen wurde.

Anwendungsfall	Zufälliges automatisches Vermischen des Zauberwürfels
Beteiligte Akteure	Zauberwürfel-Nutzer
Kurzbeschreibung	Verschiedene Ebenen des Zauberwürfels werden zufällig automatisch gedreht.
Auslöser	Berühren des Bildschirmes unter der Bedingung, dass der Benutzer die berührungsempfindliche Schaltfläche für die entsprechende Aktion berührt.
Vorbedingung	Applikation wurde gestartet.
Eingehende Informationen	Berührungspunkte des Bildschirmes
Standardablauf	1. Bildschirm wird an einer Position berührt 2. Prüfung, ob eine andere Aktion vom Benutzer gewollt ist 3. Ausführen der Aktion
Ergebnis	Der Zauberwürfel wird automatisch nach dem Zufallsprinzip vermischt und kann nach Abschluss der Aktionsausführung vom Benutzer gelöst werden.

Tabelle 4: Anwendungsfall - Zufälliges automatisches Vermischen des Zauberwürfels

3.2.4 Schrittweises Zurücksetzen von zuvor ausgeführten Aktionen

Mit dieser Funktion soll dem Benutzer eine weitere berührungsempfindliche Schaltfläche zur Verfügung gestellt werden. Hierbei soll sich die Ebene, die zuletzt verdreht wurde, bei jeder einzelnen Berührung der Schaltfläche wieder zurückdrehen. Diese Funktion soll dem Benutzer die Applikations-Nutzung erleichtern, da dieser bei einer nicht gewollten oder falsch ausgeführten Drehung der Ebene die Möglichkeit hat, die Drehung wieder rückgängig zu machen. Solange sich der Zauberwürfel nicht in seinem Anfangszustand befindet, können mit dieser Funktion alle durchgeführten Drehungen der Zauberwürfelebenen wieder zurückgesetzt werden. Dabei sollen die restlichen nicht zurückgesetzten Drehaktio-

nen bestehen bleiben und die folgenden Drehaktionen, die nach dem schrittweisen Zurücksetzen des Zauberwürfels durchgeführt wurden, hinzugefügt werden.

Der Anwendungsfall wird ausgelöst, sobald der Benutzer den Bildschirm berührt und das Programm erkennt, dass die berührungsempfindliche Schaltfläche vom Benutzer berührt wurde.

Die Anzahl an zuvor ausgeführten Aktionen, die zurückgesetzt werden sollen, wird durch die Häufigkeit der Berührungen an der entsprechenden Schaltfläche bestimmt. Die zuvor ausgeführten Aktionen können maximal so oft zurückgesetzt werden, bis der Zauberwürfel wieder sein Anfangsstadium im gelösten Zustand erreicht hat.

Anwendungsfall	Schrittweises Zurücksetzen von zuvor ausgeführten Aktionen
Beteiligte Akteure	Zauberwürfel-Nutzer
Kurzbeschreibung	Zuvor ausgeführte Aktionen werden zurückgesetzt, bis der Zauberwürfel sein Anfangsstadium nach dem Starten der Applikation erreicht hat.
Auslöser	Berühren des Bildschirmes unter der Bedingung, dass der Benutzer die berührungsempfindliche Schaltfläche für die entsprechende Aktion berührt.
Vorbedingung	Applikation wurde gestartet und es wurde zuvor mindestens eine zurücksetzbare Aktion vom Benutzer ausgeführt.
Eingehende Informationen	Berührungspunkte des Bildschirmes und die Parameter der zuvor ausgeführten Aktionen.
Standardablauf	1. Bildschirm wird an einer Position berührt 2. Prüfung, ob eine andere Aktion vom Benutzer gewollt ist 3. Ausführen der Aktion
Ergebnis	Durch das einmalige Berühren der berührungsempfindlichen Schaltfläche wird die zuletzt ausgeführte Aktion zurückgesetzt. Dies wird so oft wiederholt, bis der Zauberwürfel das Anfangsstadium nach dem Starten erreicht hat.

Tabelle 5: Anwendungsfall - Schrittweises Zurücksetzen von zuvor ausgeführten Aktionen

3.2.5 Zurücksetzen des Zauberwürfels

Die Funktionalität zum Zurücksetzen des Zauberwürfels soll durch eine weitere berührungsempfindliche Schaltfläche realisiert werden. Mit der Berührung der Schaltfläche soll sich der Zauberwürfel wieder in seinen Anfangszustand zurücksetzen und die restlichen gespeicherten ausgeführten Drehaktionen der Funktion *„3.2.4 Schrittweises Zurücksetzen von zuvor ausgeführten Aktionen“* gelöscht werden.

Der Anwendungsfall wird ausgelöst, sobald der Benutzer den Bildschirm berührt und das Programm erkennt, dass die berührungsempfindliche Schaltfläche zum Zurücksetzen des Zauberwürfels vom Benutzer berührt wurde.

Anwendungsfall	Zurücksetzen des Zauberwürfels
Beteiligte Akteure	Zauberwürfel-Nutzer
Kurzbeschreibung	Der Zauberwürfel wird per Knopfdruck in seinen Anfangszustand zurückgesetzt.
Auslöser	Berühren des Bildschirmes unter der Bedingung, dass der Benutzer die berührungsempfindliche Schaltfläche für die entsprechende Aktion berührt.
Vorbedingung	Applikation wurde gestartet.
Eingehende Informationen	Berührungspunkte des Bildschirmes
Standardablauf	1. Bildschirm wird an einer Position berührt 2. Prüfung, ob eine andere Aktion vom Benutzer gewollt ist 3. Die einzelnen Würfel werden in den Startzustand zurückgesetzt
Ergebnis	Der Zauberwürfel befindet sich wieder im Anfangszustand wie nach dem Starten der Applikation und kann wieder vermischt und gelöst werden.

Tabelle 6: Anwendungsfall - Zurücksetzen des Zauberwürfels

4 Oberflächenentwurf

Aus den Anwendungsfällen im Abschnitt *„3.2 Anwendungsfunktionalität“* lässt sich bereits eine erste Überlegung für den Entwurf der Benutzeroberfläche ableiten. Die Benutzeroberfläche soll ausschließlich über den Bildschirm des mobilen Endgerätes bedient werden können. Da der Benutzer während der Benutzung der Zauberwürfel-Applikation keine Text-Eingaben vornehmen muss, sind weder eine Hardware-Tastatur noch eine Android-Touchscreen-Tastatur erforderlich, um die Applikation zu verwenden.

Die Steuerelemente müssen groß genug sein, um die Bedienung der Applikation zu erleichtern, und sollten so sinnvoll wie möglich auf dem Bildschirm positioniert beziehungsweise angeordnet werden. Zusätzlich sollte für den Benutzer ersichtlich sein, welches Bedienelement für eine entsprechende Aktion berührt werden muss.

Abgeleitet von den geforderten Kriterien, werden folgende Komponenten für die Realisierung benötigt:

- Eine berührungsempfindliche Schaltfläche für das automatische Vermischen des Zauberwürfels nach dem Zufallsprinzip.
- Eine berührungsempfindliche Schaltfläche für das schrittweise Zurücksetzen von zuvor ausgeführten Aktionen.
- Eine berührungsempfindliche Schaltfläche für das Zurücksetzen des Zauberwürfels.
- Der virtuell simulierte Zauberwürfel.

Das Rotieren des kompletten Zauberwürfels und das Rotieren der einzelnen Ebenen sollen über entsprechende Gesten auf dem Multitouch-Bildschirm des mobilen Endgerätes erfolgen, weshalb für diese Funktionalitäten keine eigenen Schaltflächen benötigt werden.

Die Abbildung 9 enthält einen ersten Entwurf der möglichen Benutzeroberfläche mit den oben identifizierten Komponenten.

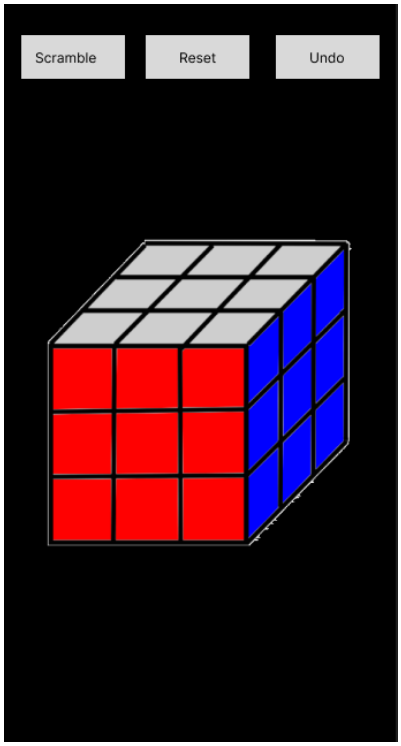

Abbildung 9: Prototyp der Benutzeroberfläche der Zauberwürfel-Applikation

Die Benutzeroberfläche besteht aus zwei Komponenten. In der Mitte befindet sich der virtuell simulierte dreidimensionale Zauberwürfel, der durch den Benutzer mit entsprechenden Berührungen im Ganzen oder ebenenweise um seine eigenen Achsen gedreht werden kann.

Im oberen Teil des Bildschirmes befinden sich drei berührungsempfindliche Schaltflächen, die für das automatische Vermischen des Zauberwürfels, das Zurücksetzen des Zauberwürfels in seinen Anfangszustand sowie für das schrittweise Zurücksetzen von zuvor ausgeführten Rotationsaktionen zuständig sind.

5 Implementierung

In diesem Kapitel werden die wichtigsten Implementierungen der Zauberwürfel-Applikation unter Android OpenGL ES erläutert. Bei der Implementierung wurden die Anforderungen aus Kapitel 3 umgesetzt. Das folgende Kapitel ist in zwei Teile gegliedert, wobei im ersten Teil der eigentliche Implementierungsprozess erläutert und im zweiten Teil die Benutzeroberfläche beschrieben wird.

5.1 Entwicklungsumgebung

Bei der Entwicklung wurde die Entwicklungsumgebung Android Studio mit der Version Bumblebee 2021.1.1 Patch 3 verwendet.

Android Studio basiert auf der IntelliJ IDEA Community Edition der Firma JetBrains. Die Verwendung von Android Studio erspart die weiteren Installationsschritte für das Android Software Development Kit (Android SDK) und das Java Development Kit (OpenJDK), da diese seit der Android Studio Version 2.2 direkt mitinstalliert werden. [KüTh21b]

Zusätzlich bietet Android Studio einen weiteren Vorteil in der Android-Entwicklung, indem die Applikation in einem Android-Emulator gestartet und getestet werden kann. Somit benötigt der Entwickler kein physisches mobiles Endgerät zum Testen der Android-Applikation, wobei die Leistung des Emulators, je nach Hardware, auf der die Applikation entwickelt wird, stark variieren kann.

Für die Entwicklung der Zauberwürfel-Applikation wurde ein Emulator des Google Pixel 4 mit der Android Version 12.0 und dem API-Level 31 verwendet.

5.2 Klassendiagramm

Das in Abbildung 10 dargestellte UML-Klassendiagramm konzentriert sich auf die Klassen und Assoziationen zur Darstellung ihrer Beziehungen untereinander. Die Funktionsweisen der einzelnen Klassen werden im Abschnitt *„5.2.1 Kurzbeschreibung der Klassen“* und *„5.2.2 Anpassungen der Klassen in der Bibliothek“* näher erläutert.

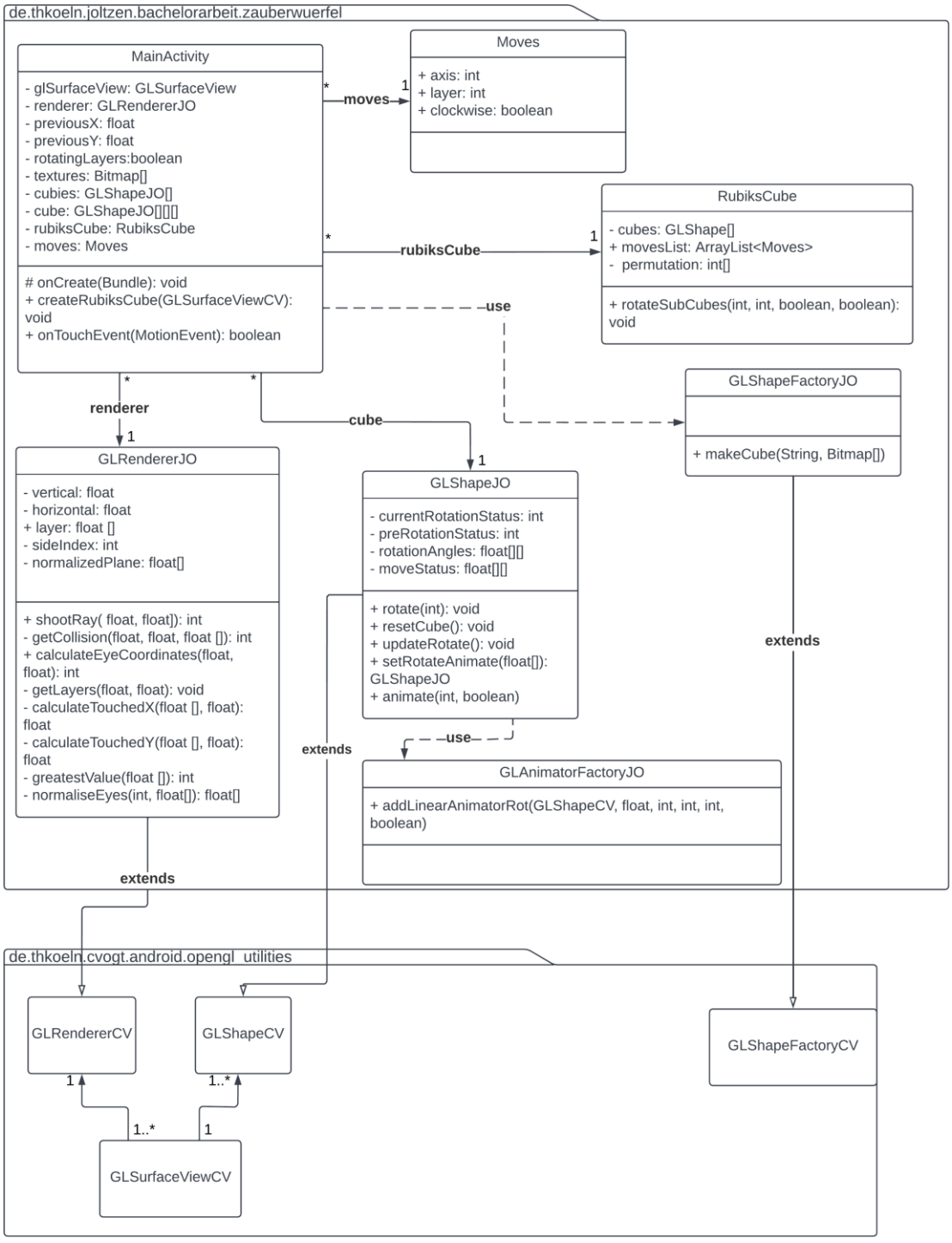

Abbildung 10: Klassendiagramm der Zauberwürfel-Applikation

5.2.1 Beschreibung der Klassen

In diesem Abschnitt werden die Funktionsweisen der einzelnen Klassen erläutert. Die Abbildungen 11 bis 17 stellen die Ausschnitte des Klassendiagramms aus Abbildung 10 dar.

MainActivity

Die Klasse `MainActivity` besteht aus insgesamt drei Methoden. Mit der Methode `onCreate` wird die geerbte Activity, der Renderer und der SurfaceView der Applikation initialisiert. Zusätzlich werden hier die berührungsempfindlichen Schaltflächen generiert und der Benutzeroberfläche hinzugefügt.

In der `createRubiksCube`-Methode wird von der `onCreate`-Methode aufgerufen. Hier werden mit sechs Bitmap Dateien 27 einzelne Würfel generiert, in eine dreidimensionale Matrix transformiert und der Benutzeroberfläche hinzugefügt.

Mit der `onTouchEvent`-Methode werden alle ausgeführten Berührungsaktionen des Benutzers, die sich auf den Zauberwürfel beziehen, gesteuert. Je nach Art der Berührung werden unterschiedliche Funktionen der Applikation aufgerufen und ausgeführt.

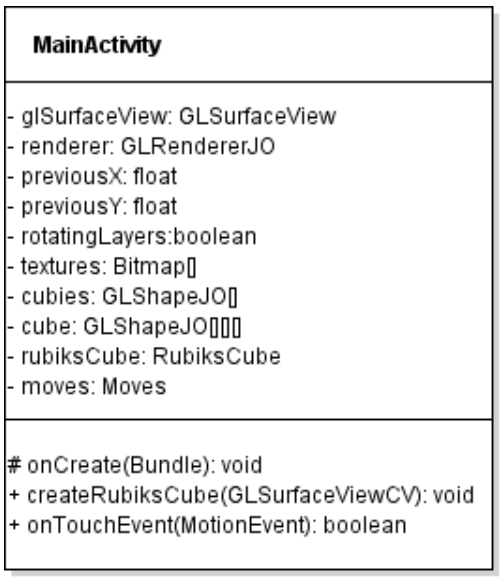

Abbildung 11: Ausschnitt des Klassendiagramms - MainActivity-Klasse der Zauberwürfel-Applikation

GLRendererJO

Die Klasse `GLRendererJO` wird von der Klasse `GLRendererCV` der bereitgestellten Android OpenGL ES-Bibliothek abgeleitet. Hierdurch wird der Funktionsumfang der Bibliothek durch die Funktionen der Objektrotation mithilfe von Berührungsaktionen um seine eigenen Achsen und der Bestimmung der berührten Zauberwürfel-Seite und Zauberwürfel-Ebene erweitert.

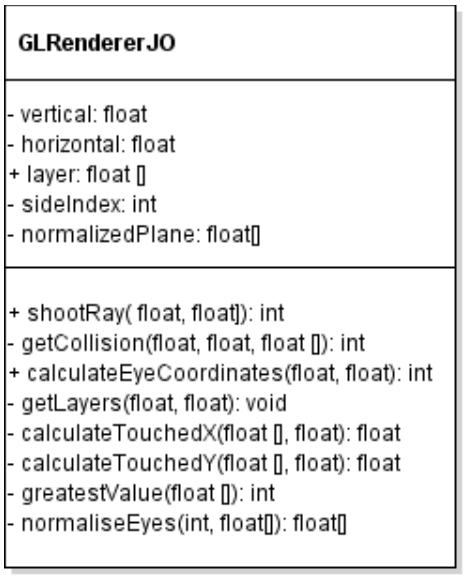

Abbildung 12: Ausschnitt des Klassendiagrams - GLRendererJO-Klasse der Zauberwürfel-Applikation

GLShapeJO

Die Klasse `GLShapeJO` wird von der Klasse `GLShapeCV` der bereitgestellten Android OpenGL ES-Bibliothek abgeleitet. Diese Klasse ist für die Zuweisung der Rotationsstadien der einzelnen Steine im virtuellen Zauberwürfel zuständig, dessen Funktionalität in Kapitel *„5.5 Zauberwürfel Status Maschine“* detailliert beschrieben wird. Zudem werden die entsprechenden Rotationsanimationen der einzelnen Ebenen des Zauberwürfels innerhalb dieser Klasse ausgeführt.

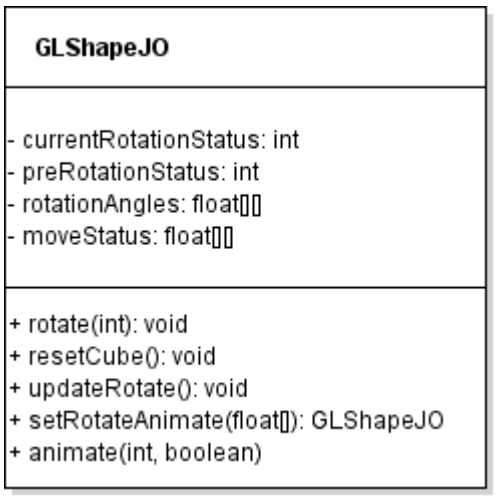

Abbildung 13: Ausschnitt des Klassendiagrams - GLShapeJO-Klasse der Zauberwürfel-Applikation

Moves

Mit der `Moves`-Klasse wird ein Objekt erzeugt, welches nach dem Ausführen einer Rotationsaktion einer Ebene generiert wird. Mithilfe dieses Objektes können die ausgeführten Aktionen zwischengespeichert werden und im späteren Verlauf beliebig oft beziehungsweise bis der Zauberwürfel wieder seinen Anfangszustand erreicht hat, vom Benutzer rückgängig gemacht werden.

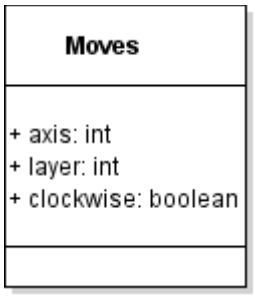

Abbildung 14: Ausschnitt des Klassendiagrams - Moves-Klasse der Zauberwürfel-Applikation

RubiksCube

Die Hauptaufgabe der `RubiksCube`-Klasse ist die korrekte Zuweisung der Parameter für das Ausführen einer Rotationsaktion. Die Methode `rotateSubCubes` bestimmt hierbei, welche Würfel zu der berührten Ebene gehören und ruft die `rotate`-Methode und die `animate`-Methode der `GLShapeJO`-Klasse, für die entsprechenden Teilwürfel im kompletten Zauberwürfel auf. Des Weiteren wird

die Rotationsaktion entsprechend des `revert`-Parameters entweder in die ArrayList `movesList` hinzugefügt oder herausgelöscht.

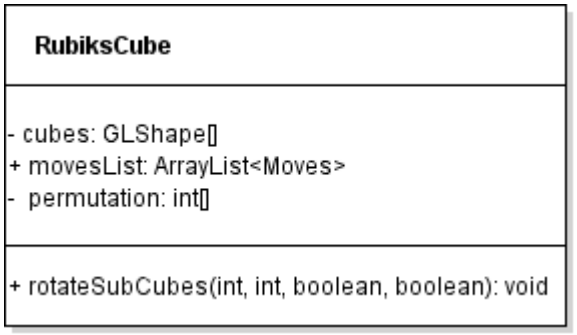

Abbildung 15: Ausschnitt des Klassendiagrams - RubiksCube-Klasse der Zauberwürfel-Applikation

GLShapeFactoryJO

Die Klasse `GLShapeFactoryJO` enthält lediglich eine Methode `makeCube`, die der Erstellung eines Würfels von der Klasse `GLShapeJO` dient. Diese Methode ist eine Kopie der gleichnamigen Methode aus der Bibliotheksklasse `GLShapeFactoryCV` und musste implementiert werden, um ein neues Objekt der erweiterten `GLShapeJO` Klasse erzeugen zu können.

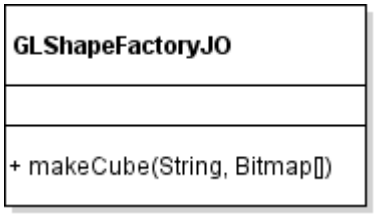

Abbildung 16: Ausschnitt des Klassendiagrams - GLShapeFactoryJO-Klasse der Zauberwürfel-Applikation

GLAnimatorFactoryJO

Die Klasse `GLAnimatorFactoryJO` dient zur Umsetzung der Animationen für die Rotationen einer einzelnen Ebene des Zauberwürfels.

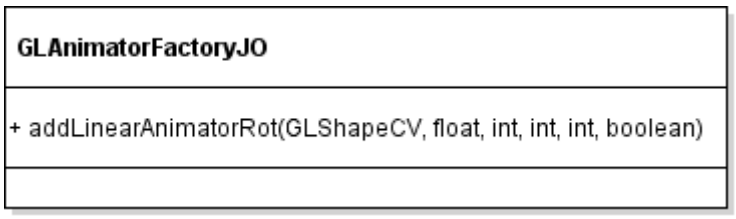

Abbildung 17: Ausschnitt des Klassendiagrams - GLAnimatorFactoryJO-Klasse der Zauberwürfel-Applikation

5.2.2 Anpassungen der Klassen in der Bibliothek

GLRendererCV

Die Klasse `GLRendererCV` der bereitgestellten Android OpenGL ES-Bibliothek musste um vier Getter-Methoden und eine weitere Setter-Methode erweitert werden, um die benötigten privaten Attribute `eyeX`, `eyeY`, `eyeZ` und `projectionMatrix` innerhalb der abgeleiteten Klasse `GLRendererJO` benutzen und anpassen zu können. Hierdurch wurde die minimale direkte Anpassung der Bibliothek gewährleistet und alle weiteren Erweiterungen außerhalb der Bibliothek vorgenommen.

TextureBitmapsCV

Die letzte direkte Anpassung der Bibliothek wurde in der `get`-Methode der `TextureBitmapsCV` Klasse implementiert. Hier wurden die sechs Bitmaps-Ressourcen hinzugefügt, um diese in der Zauberwürfel-Applikation für die Erstellung der einzelnen Würfel benutzen zu können.

5.3 Grundstruktur des Zauberwürfels

Nach dem Starten der Applikation werden zunächst in der `onCreateRubiksCube`-Methode innerhalb der `MainActivity`-Klasse 27 einzelne Teilwürfel erstellt, in eine dreidimensionale Matrix transformiert und der Benutzeroberfläche hinzugefügt. Aus diesen Teilwürfeln wird ein Objekt der `RubiksCube`-Klasse erstellt. Hierbei werden jedem Teilwürfel des Zauberwürfels ein Indexwert zwischen 0 und 26 zugewiesen und in einem Array gespeichert. Durch diese Zuweisung entsteht eine Anordnung von Objekten beziehungsweise eine Permutation für den Zauberwürfel, die für den weiteren Implementierungsprozess ausschlaggebend ist. Die resultierende Permutation des Zauberwürfels im gelösten Zustand ist in Abbildung 18 dargestellt.

Mithilfe dieser Permutation kann bestimmt werden, welche Teilwürfel zu welcher Ebene des Zauberwürfels gehören, um bei der Rotation einer Ebene zu identifizieren, welche Teilwürfel gedreht werden müssen. Da bei dem Vermischen des Zauberwürfels nicht nur eine Seite des Zauberwürfels rotiert wird, sondern eine Ebene, besitzen die Kantensteine und Ecksteine, die zu mehr als einer Seite gehören, denselben Permutations-Index. Jeder Index repräsentiert somit genau ein Zauberwürfel-Element.

Sobald eine Ebene vom Benutzer rotiert wird, muss die Permutation des Zauberwürfels entsprechend angepasst werden. Dieses Vorgehen wird innerhalb der `rotateSubCubes`-Methode der `RubiksCube`-Klasse realisiert. Mithilfe von verschiedenen Fallunterscheidungen der Achsen und der Ebenen, die gedreht werden sollen, werden die Positionen der Teilwürfel innerhalb des Permutations-Index bestimmt. Wie in der Abbildung 18 ersichtlich ist, besteht die vordere Ebene zu Beginn aus den Teilwürfeln mit den folgenden Indizes: [2, 5, 8, 11 ,14, 17, 20, 23, 26].

Nachdem die vordere Ebene um 90 Grad im Uhrzeigersinn gedreht wurde, wird die Permutation der vorderen Ebene wie folgt angepasst: [20, 11, 2, 23, 14, 5, 26, 17, 8]. Die neue Permutation des Zauberwürfels nach dem Ausführen dieser Aktion ist in Abbildung 19 dargestellt. Die neue Zuweisung ist notwendig, damit das Programm bei der darauffolgenden Drehung einer weiteren Ebene identifizieren kann, welche Teilwürfel aktuell zu der Ebene, die rotiert werden soll, gehören.

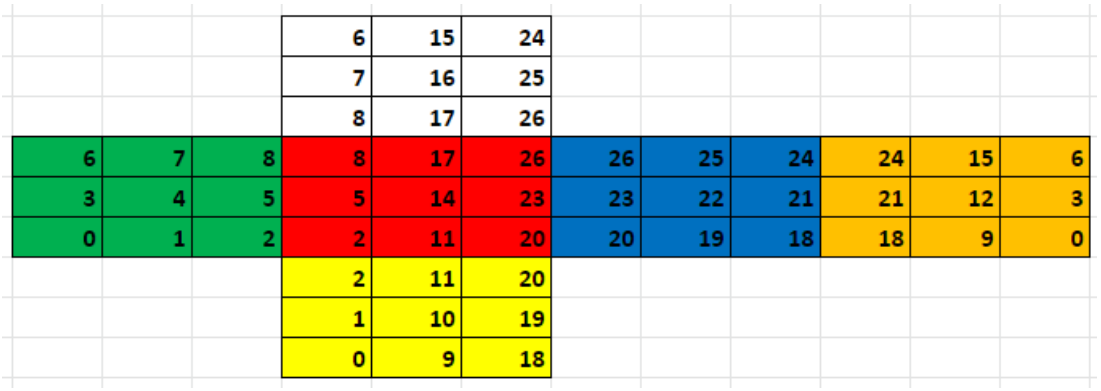

Abbildung 18: Die Permutation des virtuellen Zauberwürfels im gelösten Zustand

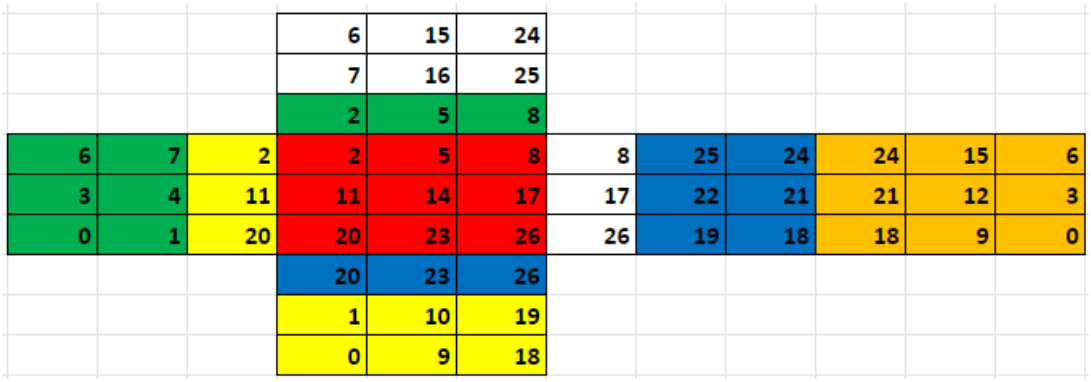

Abbildung 19: Die Permutation des virtuellen Zauberwürfels nach einer Drehung der vorderen Ebene um 90 Grad im Uhrzeigersinn

5.4 Berührungsaktionen des Zauberwürfels

Sobald der Benutzer den Bildschirm berührt, wird die `onTouchEvent`-Methode innerhalb der `MainActivity`-Klasse aufgerufen. Aus dem übergebenen Objektparameter kann zum einen die Art der Berührung, zum Beispiel ob der Finger den Bildschirm berührt, der Finger über den Bildschirm bewegt wurde oder ob der Finger vom Bildschirm abgehoben wurde, als auch die aktuelle Position der Berührung ermittelt werden.

Für die Identifikation, ob der komplette Zauberwürfel um seine eigenen Achsen gedreht werden soll oder nur eine bestimmte Ebene des Zauberwürfels, wird das Prinzip des sogenannten *Ray Castings* oder auch *Ray Tracings* angewendet. Mit diesem Verfahren kann identifiziert werden, ob ein Objekt vom Benutzer berührt

wurde oder nicht. Um dies zu realisieren, wird ein Strahl (Ray) von einem Startpunkt aus in eine bestimmte Richtung projiziert, und geprüft, ob dieser Strahl mit einem Objekt kollidiert. Anhand dieses Kollisionspunktes kann im Anschluss bestimmt werden, welche Seite des Zauberwürfels vom Benutzer berührt wurde. [AnGe16]

In einer gerenderten Szene wird jedes Objekt im Allgemeinen durch eine Matrix für die Betrachtung und durch eine Matrix für die Projektion transformiert. Während die Betrachtungs-Matrix die Richtung und die Position beschreibt, von welcher die Szene betrachtet wird, beschreibt die Projektions-Matrix die Kartierung von den dreidimensionalen Punkten einer Szene zu den zweidimensionalen Punkten von dem Betrachtungsstandpunkt, aus der Sicht des Benutzers. [LoGL22]

In OpenGL wird zwischen fünf verschiedenen Arten von Koordinatensystemen differenziert: Objektkoordinatensystem (Local space), Weltkoordinatensystem (World space), Betrachtungskoordinatensystem (View space), Clip-Koordinatensystem (Clip space) und Bildschirmkoordinatensystem (Screen space). [LoGL22]

Das Objektkoordinatensystem beschreibt hierbei die Koordinaten von dem Objekt, die relativ zu dessen lokalen Ursprungspunkt (0,0,0) stehen. Durch die Transformation dieser Koordinaten mithilfe der Model-Matrix wird das Objekt in das Weltkoordinatensystem gesetzt.

„Die Modelmatrix ist eine Transformationsmatrix, die ihr Objekt transformiert, skaliert und/oder rotiert, um es in der Welt an einem Ort oder in einer Ausrichtung zu platzieren, zu der dieses gehört.“ [LoGL22]

In einem Betrachtungskoordinatensystem werden die räumlichen Koordinaten in die Koordinaten konvertiert, die sich vor dem Benutzer befinden. Das Sichtfeld ist also der Raum, der aus der Sicht des Benutzers gesehen wird. Um dies zu erreichen, werden verschiedene Verschiebungen und Drehungen einer Szene so ausgeführt, dass sich das Objekt auf der Vorderseite der Kamera befindet. Aus diesem Grund wird der Betrachtungsraum auch im Zusammenhang mit OpenGL als *Betrachtungsraum der Kamera* bezeichnet. [LoGL22]

Im Anschluss transformiert die Projektions-Matrix die Koordinaten aus dem Betrachtungskoordinatensystem in das Clip-Koordinatensystem. Da sich der normalisierte Raum von Open GL zwischen den Koordinaten (-1,-1,-1) und (1,1,1) befindet, müssen die resultierenden Koordinaten auch zwischen diesen beiden Punkten liegen, ansonsten wären die Koordinaten nicht mehr für den Benutzer sichtbar und würden somit verworfen werden. Die hieraus hervorgehenden Koordinaten werden *Normalized Device Coordinates* genannt und der Prozess des Verwerfens der nicht sichtbaren Koordinaten wird in Bezug auf OpenGL als Clipping bezeichnet. [HaSe15]

Die Implementierung dieser Funktionsweise in der Applikation stützt sich auf die folgende Anleitung und Beschreibung der mathematischen Hintergründe des Ray Castings. [AnGe16]

Für die Umsetzung werden im ersten Schritt die vom Benutzer berührten zweidimensionalen x- und y-Koordinaten aus der Methode `onTouchEvent` der `MainActivity`-Klasse normalisiert und als Parameter an die `shootRay`-Methode der `GLRendererJO`-Klasse übergeben. Der Startpunkt des Strahls wird im Anschluss durch die Transformierung der normalisierten Koordinaten auf der Ebene (z = -1) und der invertierten Projektions-Matrix berechnet.

Die Richtung, in die der Strahl gerichtet werden soll, wird durch eine weitere Transformation der normalisierten Koordinaten auf der Ebene (z = -1) und der invertierten Betrachtungs-Matrix bestimmt. Der resultierende Vektor wird im Anschluss normalisiert und an die `getCollision`-Methode der `GLRendererJO`-Klasse übergeben.

Im nächsten Schritt wird mithilfe von sechs verschiedenen normalisierten Vektoren, die die Blickrichtung der Seite auf der Achse implizieren, bestimmt, welche der sechs Seiten des Zauberwürfels berührt wurde.

Sobald der Strahl eine der Zauberwürfel-Seiten berührt hat, wird der Index der Blickrichtung, (rechts = 0, links = 1, oben = 2, unten = 3, vorne = 4, hinten = 5) mit der geringsten Distanz vom Startpunkt aus zurückgegeben. Falls keine der Flächen berührt wurde, wird der Seiten-Index auf -1 gesetzt. Aus dem zurückgegeben Seiten-Index wird dann in der `onTouchEvent`-Methode der `MainActivity`-Klasse geprüft, ob eine Seite des Zauberwürfels berührt wurde oder nicht.

Falls eine Seite des Zauberwürfels berührt wurde, wird die Variable `rotatingLayers`, die für die Fallunterscheidung zwischen dem Rotieren des kompletten Zauberwürfels und dem Rotieren einer einzelnen Ebene des zuständig ist, auf den Wert *true* gesetzt. Sobald der Benutzer den Bildschirm nicht mehr berührt, wird die `shootRay`-Methode erneut aufgerufen, um die zu rotierende Ebene zu identifizieren.

Bei der Rotation einer einzelnen Ebene des Zauberwürfels sind zwei Faktoren ausschlaggebend für die korrekte Ausführung der Rotation. Zum einen muss bestimmt werden, welche der insgesamt neun Ebenen des Zauberwürfels rotiert werden soll und in welche Richtung der Benutzer die Ebene drehen möchte.

Anhand der Distanz zwischen der Startposition und der Endposition der Berührungsaktion kann bestimmt werden, in welche Richtung die Wischbewegung vom Benutzer ausgeführt worden ist. Dieser Wert wird im späteren Verlauf als Parameter an eine der entsprechenden Rotationsfunktionen übergeben.

Zur Identifikation, welche Ebene des Zauberwürfels der Benutzer rotieren möchte, wird die `getLayers`-Methode innerhalb der `GLRendererJO`-Klasse aufgerufen. Da der Zauberwürfel an sich nicht rotiert wird, sondern sich lediglich die Kamera um den Zauberwürfel bewegt, kann so der Blickwinkel auf den Zauberwürfel angepasst werden. Anhand der aktuellen Kameraposition kann so identifiziert werden, welche Seite aus der Sicht des Benutzers betrachtet wird. Mithilfe der `greatestValue`-Funktion wird dann im Anschluss festgestellt, welcher Wert der Kamerapositionen am größten ist und der entsprechende Index zurückgegeben.

Für die Berechnung des Berührungspunktes auf dem Zauberwürfel werden die beiden Funktionen `calculateTouchedX` und `calculateTouchedY` anhand der berührten x- und y-Koordinaten auf dem Bildschirm aufgerufen. Die lineare Berechnung innerhalb dieser beiden Funktionen normalisiert die Kanten des virtuellen Zauberwürfels, wodurch eine Unterteilung des Zauberwürfels in neun Ebenen auch bei einer dreidimensionalen Betrachtung ermöglicht werden kann.

Jede der drei Rotationsachsen x, y und z sind, wie in den Tabellen 7, 8 und 9 dargestellt, insgesamt drei Ebenen zugewiesen. Aufgrund der Tatsache, dass eine Zauberwürfel-Seite nur um zwei Rotationsachsen rotiert werden kann, besitzt jede der Zauberwürfel-Seiten jeweils zwei Ebenen. Während die vordere und hintere Zauberwürfel-Seite um die Rotationsachsen x und y rotiert werden kön-

nen, können die linke und rechte Zauberwürfel-Seite um die y- und z-Achse sowie die obere und untere Seite um die x- und z-Achse rotiert werden. Durch die vorherige Normalisierung der Zauberwürfel-Kanten kann die berührte Seite sowohl in der x-Achse als auch in der y-Achse durch drei geteilt werden, wodurch der Ebenen-Index der berührten Seite gespeichert wird. Nachdem der korrekte Ebenen-Index identifiziert worden ist, wird in Kombination mit der Richtungsbestimmung der Wischaktion ausgewählt, welche Zauberwürfel-Ebene in welche Richtung rotiert werden soll und im Anschluss entsprechend animiert.

Das Sequenzdiagramm in Abbildung 20 stellt den detaillierten Funktionsaufruf der oben beschriebenen Implementation dar.

Ebene	**Ebenen-Index**
Links	0
Mitte X	1
Rechts	2

Tabelle 7: Rotationsebenen der X-Achse und der jeweiligen Index-Zuweisung

Ebene	**Ebenen-Index**
Unten	0
Mitte Y	1
Oben	2

Tabelle 8: Rotationsebenen der Y-Achse und der jeweiligen Index-Zuweisung

Ebene	**Ebenen-Index**
Vorne	0
Mitte Z	1
Hinten	2

Tabelle 9: Rotationsebenen der Z-Achse und der jeweiligen Index-Zuweisung

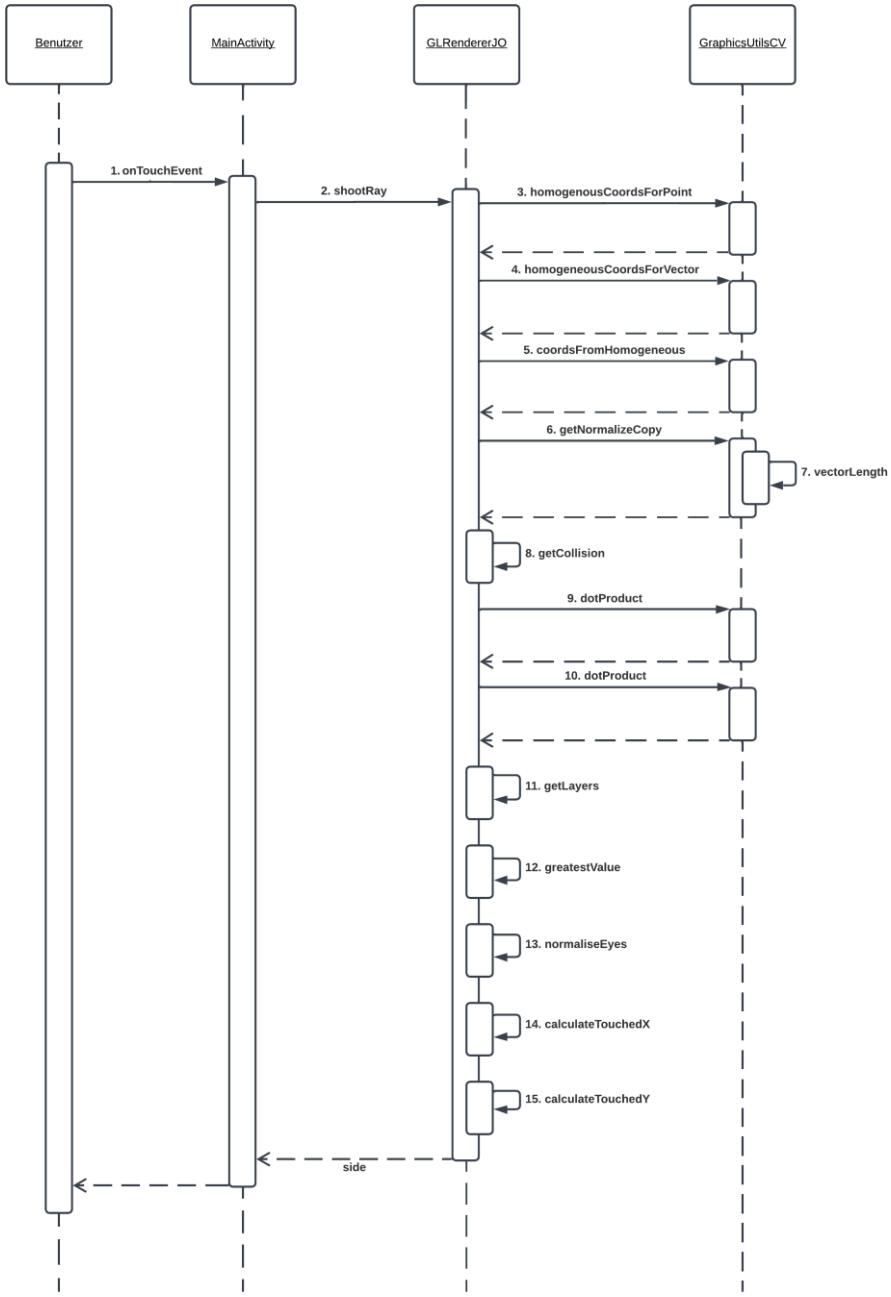

Abbildung 20: Sequenzdiagramm - Rotieren einer einzelnen Ebene des Zauberwürfels

Im Falle, dass keine Seite des Würfels berührt wurde, und der Seiten-Index somit gleich -1 ist, kann der Benutzer den Würfel mithilfe von anschließenden Wischbewegungen um seine eigenen Achsen rotieren. Hierzu wird eine weitere Methode `calculateEyeCoordinates` mit den Parametern für die Berechnung des Horizontalwinkels und des Vertikalwinkels in der `GLRendererJO` Klasse aufgerufen. Um einen besseren Überblick über die Berechnung des Horizontalwinkels (Formel 1) und des Vertikalwinkels (Formel 2) zu erschaffen, wird die Berechnung wie folgt zusammengefasst:

$$Horizontalwinkel = \left(\frac{distanzX}{Bildschirmbreite}\right) * \pi + Horizontalwinkel$$

Formel 1

$$Vertikalwinkel = \left(\frac{distanzY}{Bildschirmhöhe}\right) * \pi + Vertikalwinkel$$

Formel 2

Anhand dieser beiden Werte wird der Zusammenhang zwischen den Polarwinkeln und den kartesischen Koordinaten wie folgt berechnet (Formel 3). [AlRu22]

$$EyeCoordinates = \begin{pmatrix} eyeX \\ eyeY \\ eyeZ \end{pmatrix} = |r| * \begin{pmatrix} \cos(v) & * & \sin(h) \\ & \sin(v) & \\ \cos(v) & * & \cos(h) \end{pmatrix}$$

Formel 3

Die neu berechneten Werte für die Kamerapositionen werden im Anschluss an die Klasse `GLRendererCV` übergeben. Durch den Aufruf der `updateViewProjectionMatrix`-Methode innerhalb der `GLRendererCV`-Klasse wird die aktuelle `viewProjectionMatrix` anhand der neuen Kamerapositionen aktualisiert und der Würfel kann von dem Benutzer rotiert werden. Der detaillierte Funktionsaufruf ist in Abbildung 21 in Form eines Sequenzdiagramms dargestellt.

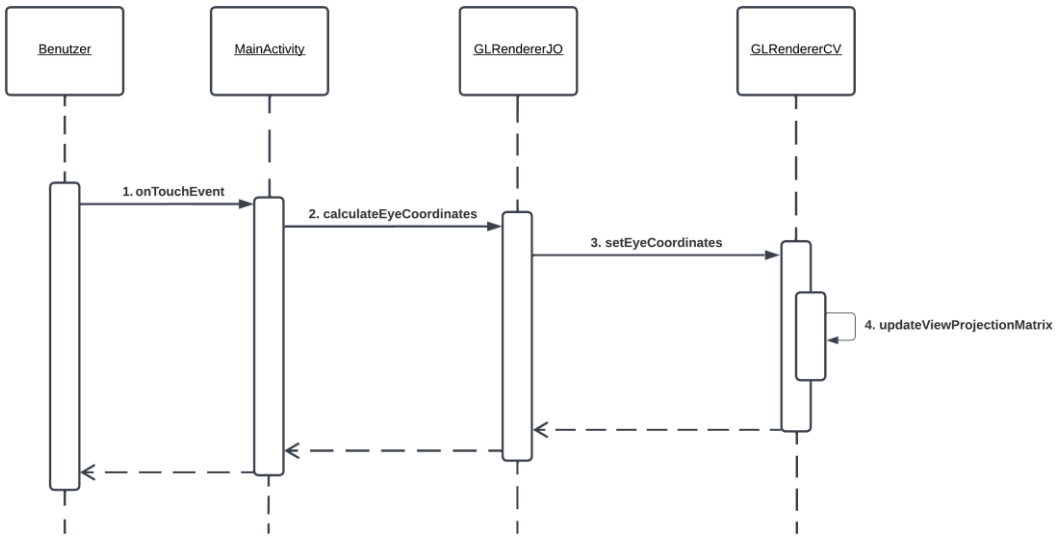

Abbildung 21: Sequenzdiagramm - Rotieren des kompletten Zauberwürfels

5.5 Zauberwürfel Status Maschine

Die Hauptfunktionalität eines Zauberwürfels ist die Möglichkeit, diesen durch verschiedene Rotationen der einzelnen Ebenen zu vermischen. Der nächste Schritt in der Implementierung war dementsprechend, dass sich die einzelnen Teilwürfel des Zauberwürfels so rotieren, dass die Teilwürfel nach einer bestimmten Drehung, die korrekte Farbkombination im Zauberwürfel anzeigen. Bei einer Rotation von der vorderen Ebene des Zauberwürfels, werden zum Beispiel die Würfel mit dem Permutations-Index, wie in Abbildung 18 ersichtlich, von [2, 5, 8, 11, 14, 17, 20, 23, 26] um 90 Grad um die z-Achse rotiert. Wenn der Benutzer eine weitere Drehung einer Ebene ausführen möchte, wie beispielsweise eine Drehung der rechten Ebene um 90 Grad, sind drei der Würfel in der Ebene bereits um 90 Grad um die z-Achse rotiert worden. Da eine Drehung der rechten Ebene eine Rotation um 90 Grad um die x-Achse bedeuten würde, müssten sich also die drei schon vorher rotierten Würfel, ausgehend von ihrer momentanen Rotation, erneut um 90 Grad um die x-Achse drehen.

Zu Beginn der Entwicklung wurde angenommen, dass die Ausführungsreihenfolge der Achsenrotationen nicht von Bedeutung sei, jedoch war diese Annahme nicht korrekt gewesen. Ein Würfel befindet sich, je nachdem um welche Achse der Würfel zuerst rotiert wurde, immer in einem anderen Zustand.

Zur Veranschaulichung, dass die Rotationen nicht kumulativ sind, wird angenommen, dass der Würfel zuerst um 90 Grad um die x-Achse rotiert wird und im Anschluss erneut um die y-Achse. Ausgehend von dem Ausgangszustand (rot befindet sich vorne und weiß befindet sich oben), ist nach dem Abschluss der Rotation der Würfel im folgenden Zustand (blau befindet sich vorne und rot befindet sich oben). Wenn die Rotationsreihenfolge nun vertauscht wird und der Würfel zuerst um die y-Achse und im Anschluss um die x-Achse rotiert wird, befindet sich der Würfel in einem anderen Zustand (gelb befindet sich vorne und blau befindet sich oben).

Daraus folgte während der Implementierung die wichtige Erkenntnis, dass die Rotationen eines Würfels in Android OpenGL ES unter Verwendung der bereitgestellten Bibliothek nicht kumulativ sind.

Dies bedeutet, dass es nicht möglich war, die darauffolgende Rotation, ausgehend von der aktuellen Rotationsausrichtung zu addieren. Sobald die Funktion `setRotation` in der Bibliotheksklasse `GLShapeCV` mehrfach hintereinander mit verschiedenen Parametern aufgerufen worden ist, wurde der einzelne zu rotierende Würfel bei der nächsten Rotation in seinen Ausgangszustand zurückversetzt und von dieser Position aus erneut rotiert. Dadurch wurde der Zauberwürfel nicht korrekt vermischt und wäre in diesem Zustand nicht lösbar gewesen (Abbildung 22). Der erwartete Zustand nach der Ausführung der oben genannten Zugkombination ist in Abbildung 23 dargestellt.

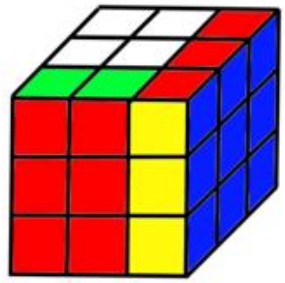

Abbildung 22: Unlösbarer Zustand - Ergebnis nach jeweils einer Drehung der vorderen und rechten Ebene um 90 Grad im Uhrzeigersinn

Abbildung 23: Lösbarer Zustand - Erwartetes Ergebnis nach jeweils einer Drehung der vorderen und rechten Ebene um 90 Grad im Uhrzeigersinn

Im Anschluss an die Erkenntnis, dass die Rotationen nicht kumulativ sind, wurde nach einer neuen Lösungsalternative gesucht. Während der Lösungsfindung wurde ersichtlich, dass sich die Rotationen zwar nicht addieren lassen, es jedoch möglich ist, dass sich ein Würfel durch bestimmte Winkelangaben in die korrekte Position rotieren lässt.

In Abbildung 24 sind die Rotationswinkel eines Würfels aufgelistet, die benötigt werden, um einen Würfel in den entsprechenden Zustand zu versetzen. Dabei kann ein einzelner Würfel insgesamt 24 verschiedene Rotationsstadien besitzen. Die Spalte *angleRotation* gibt hierbei den Status-Index an, in welchem Zustand sich der Würfel im Anschluss an die Rotation befindet. Zur Vereinfachung wurden nur die Farben der oberen und vorderen Seite des Würfels aufgelistet, da sich die Farborientierung eines einzelnen Würfels nach einer Rotation um die eigene Achse nicht verändert. Ein wichtiger Hintergrundaspekt ist hierbei, dass jeder einzelne der 27 Würfel die gleiche Farbanordnung besitzt, da sie einzeln generiert werden.
Im Anschluss an die initiale Generierung des Zauberwürfels befinden sich alle 27 Würfel in dem Zustand mit dem Index 0.

Durch die Implementierung der Zustandsmaschine können alle Würfel von einer bestimmten Rotation ausgehend von ihrem aktuellen Stadium rotiert werden.

Als Beispiel wird wieder angenommen, dass der Benutzer die vordere Ebene um 90 Grad im Uhrzeigersinn rotieren und im Anschluss an diese Rotation eine weitere Rotation der rechten Ebene ausführen möchte. Da sich alle Würfel zu Beginn in dem Rotationszustand 0 befinden und eine Rotation der vorderen Ebene eine Rotation um die z-Achse indiziert, werden die Würfel mit den Permutations-Indizes [2, 5, 8, 11, 14, 17, 20, 23, 26] auf den Status 15 gesetzt. Die Permutations-Indizes der rechten Ebene sind folgende: [18, 19, 20, 21, 22, 23, 24, 25, 26]. Daraus resultiert eine Schnittmenge von den beiden Permutations-Indizes aus vorne ∩ rechts = {20, 23, 26}. Da diese drei Elemente der Schnittmenge schon auf den neuen Status 15 gesetzt wurden, wird bei einer Rotation um die x-Achse dessen Status verwendet. Bei allen anderen Mengenelementen der rechten Ebene wird weiterhin der Status 0 benutzt, da sich diese seit der initialen Generierung noch nicht verändert haben. Aus diesem Grund bekommen die Elemente {18, 19, 21, 24, 25} der rechten Ebene den Rotationsstatus 10 und die Elemente {20, 23, 26} der rechten Ebene den Rotationsstatus 11. Jede weitere Rotation einer Ebene orientiert sich an diesem Prinzip und dadurch konnte das korrekte Vermischen des Zauberwürfels realisiert werden. In Abbildung 25 sind die Rotationsstadien dargestellt, die der Würfel im Anschluss an eine Rotation in eine bestimmte Richtung erreicht.

angleRotation	oben	vorne	x	y	z
0	weiß	rot	0	0	0
1	weiß	grün	0	90	0
2	weiß	orange	0	180	0
3	weiß	blau	0	270	0
4	blau	rot	0	0	90
5	blau	weiß	0	90	90
6	blau	orange	180	0	270
7	blau	gelb	0	270	90
8	rot	weiß	90	180	0
9	rot	grün	90	180	270
10	rot	gelb	90	180	180
11	rot	blau	90	180	90
12	grün	gelb	0	90	270
13	grün	orange	0	180	270
14	grün	weiß	0	270	270
15	grün	rot	0	0	270
16	gelb	blau	0	90	180
17	gelb	grün	0	270	180
18	gelb	rot	0	0	180
19	gelb	orange	0	180	180
20	orange	weiß	90	0	0
21	orange	blau	90	0	90
22	orange	gelb	90	0	180
23	orange	grün	90	0	270

Abbildung 24: Rotationswinkel eines Würfels in x-, y- und z-Richtung

			x	x	y	y	z	z
angleIndex	oben	vorne	90	-90	90	-90	90	-90
0	weiß	rot	10	20	3	1	15	4
1	weiß	grün	12	5	0	2	23	9
2	weiß	orange	22	8	1	3	6	13
3	weiß	blau	7	14	2	0	11	21
4	blau	rot	9	21	7	5	0	18
5	blau	weiß	1	16	4	6	20	8
6	blau	orange	23	11	5	7	19	2
7	blau	gelb	17	3	6	4	10	22
8	rot	weiß	2	18	9	11	5	14
9	rot	grün	13	4	10	8	1	17
10	rot	gelb	19	0	11	9	12	7
11	rot	blau	6	15	8	10	16	3
12	grün	gelb	16	1	15	13	22	10
13	grün	orange	21	9	12	14	2	19
14	grün	weiß	3	17	13	15	8	20
15	grün	rot	11	23	14	12	18	0
16	gelb	blau	5	12	18	19	21	11
17	gelb	grün	14	7	19	18	9	23
18	gelb	rot	8	22	17	16	4	15
19	gelb	orange	20	10	16	17	13	6
20	orange	weiß	0	19	21	23	14	5
21	orange	blau	4	13	22	20	3	16
22	orange	gelb	18	2	23	21	7	12
23	orange	grün	15	6	20	22	17	1

Abbildung 25: Status Tabelle der Würfel nach einer Rotation in die entsprechende Achse um 90 Grad oder -90 Grad

5.6 Animationen

Nachdem die Hauptfunktionalität eines realen Zauberwürfels, wie in Kapitel „*5.5 Zauberwürfel Status Maschine“* beschrieben, implementiert wurde, musste im letzten Schritt der Implementierung die visuelle Darstellung der Rotation einer Zauberwürfel-Ebene in Form einer Animation umgesetzt werden.

Die Rotationsanimation einer Zauberwürfel-Ebene ist hierbei in zwei wesentliche Teilschritte unterteilt. Zum einen wird eine Animationsfunktion benötigt, welche alle Teilwürfel der zu rotierenden Ebene um einen bestimmten Koordinatenpunkt, rotiert und zusätzlich eine weitere Animationsfunktion, in der sich die einzelnen Teilwürfel synchron um sich selbst rotieren. Durch die synchrone Ausführung dieser beiden Funktionen entsteht eine korrekte Rotationsanimation für eine bestimmte Ebene des Zauberwürfels.

Für die Rotation der Teilwürfel um einen gegebenen Achsenpunkt wurde die Methode `addAnimatorArcPathAroundAxis` der bereitgestellten Bibliotheksklasse `GLAnimatorFactoryCV` verwendet. Durch das Ausführen dieser Funktion werden alle Teilwürfel der entsprechenden Zauberwürfel-Ebene entlang eines Bogens um eine Achse bewegt, welche durch zwei Punkte in dem dreidimensionalen Raum bestimmt wird.

Da auch die Rotationen der Teilwürfel bezüglich der Rotationsanimationen, wie in Kapitel *„5.5 Zauberwürfel Status Maschine“* beschrieben ist, nicht kumulativ sind, musste für den zweiten Teilschritt der Animation die bereitgestellte Bibliothek entsprechend erweitert werden. Hierfür wurde in der `GLAnimatorFactoryJO`-Klasse die `addLinearAnimatorRot`- Methode implementiert. Der in dieser Methode erstellte `ObjectAnimator` greift mit dem propertyName „rotateAnimate“ auf die implementierte `setRotateAnimate` Methode in der Klasse `GLShapeJO` zu. Hierbei werden die zu der Ebene zugehörigen Teilwürfel zuerst um die Achse rotiert, in die der Benutzer die entsprechende Wischbewegung ausgeführt hat, und im Anschluss werden diese erneut einzeln, ausgehend von den entsprechenden Rotationswinkeln des vorherigen Zustands des Teilwürfels, rotiert. Durch diese Vorgehensweise wird verhindert, dass sich die Teilwürfel, die schon zuvor rotiert worden sind, gleichzeitig um mehrere Achsen drehen und somit eine falsche Rotationsanimation darstellen würden.

Die Abbildung 26 zeigt den vorher beschriebenen Funktionsaufruf in Form eines Sequenzdiagramms.

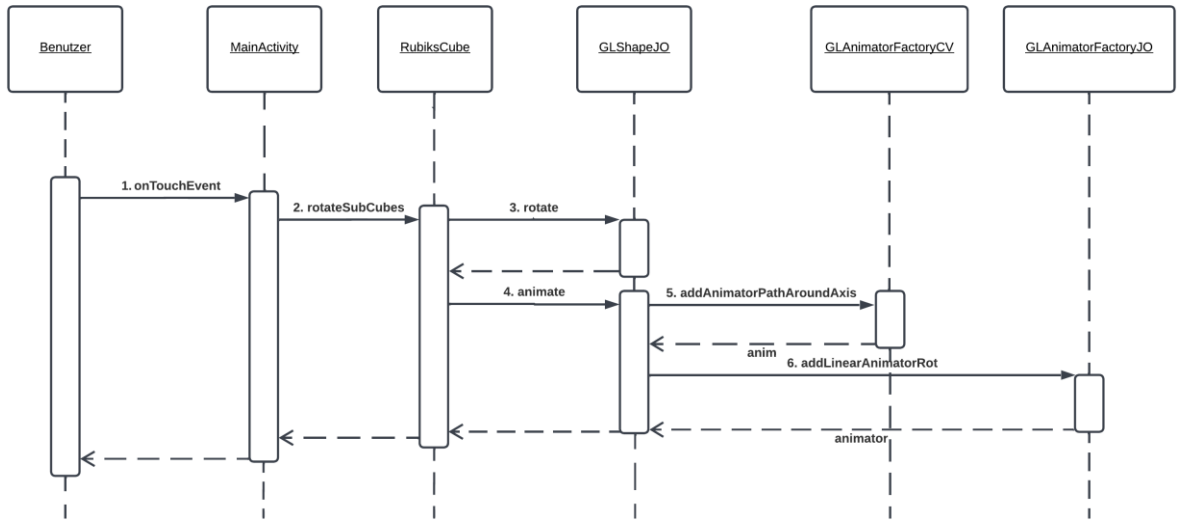

Abbildung 26: Sequenzdiagramm - Rotationsanimationen beim Rotieren einer einzelnen Ebene des Zauberwürfels.

5.7 Benutzeroberfläche

Im Folgenden wird beschrieben, wie die Benutzeroberfläche der Zauberwürfel-Applikation umgesetzt wurde. Bei der Umsetzung wurde auf ein übersichtliches Layout der Bildschirmelemente geachtet. Die Abbildung 27 zeigt die Benutzeroberfläche der Zauberwürfel-Applikation nach dem Starten der Applikation.

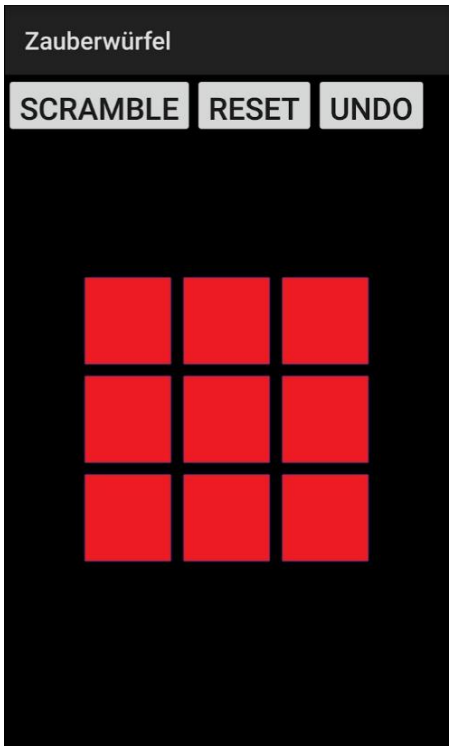

Abbildung 27: Benutzeroberfläche der Zauberwürfel-Applikation nach dem Starten der Applikation

Rotation des kompletten Zauberwürfels

Um die Rotation des kompletten Zauberwürfels um seine eigenen Achsen zu starten, muss die vom Benutzer ausgeführte Aktion zwei Kriterien erfüllen. Zum einen muss die Startposition der Berührung einen Berührungspunkt unterhalb des Zauberwürfels beinhalten, und zum anderen muss im Anschluss an die Bildschirmberührung eine Wischbewegung in die entsprechende Rotationsrichtung vom Benutzer ausgeführt werden.

Während der Ausführung der Aktion spielt es keine Rolle, ob der Benutzer diese Wischbewegung auf Höhe einer weiteren Benutzeroberflächen-Komponente ausführt. Das bedeutet, dass die Rotation des Zauberwürfels nicht abgebrochen wird, sobald der Benutzer eine der berührungsempfindlichen Schaltflächen be-

rührt, mit der Folge, dass dem Benutzer während der Ausführung der Wischbewegung die komplette Benutzeroberfläche zur Verfügung steht, um den Zauberwürfel um seine eigenen Achsen zu rotieren.

Die Rotationsaktion des kompletten Zauberwürfels wird so lange ausgeführt, bis der Benutzer die Berührungsaktion beendet, indem dieser den Finger vom Bildschirm hebt. Erst dann kann der Benutzer eine weitere Aktion ausführen und alle Funktionen stehen diesem wieder zur Verfügung.

In Abbildung 28 ist die Benutzeroberfläche nach dem Beenden der Rotationsaktion des Zauberwürfels um die eigene Achse abgebildet.

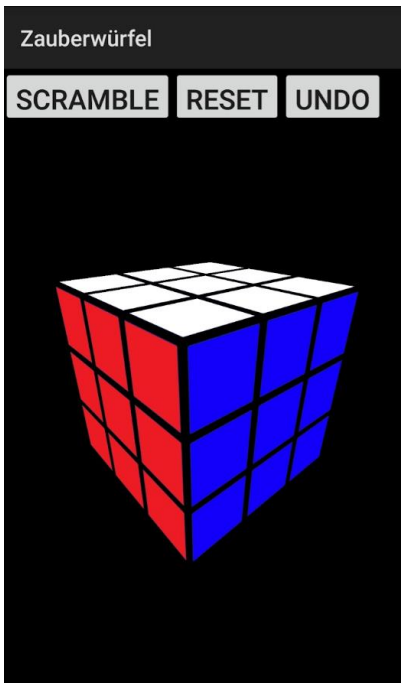

Abbildung 28: Rotierter Zauberwürfel um seine eigene Achse

Einzelne Ebenen des Zauberwürfels rotieren

In Abbildung 29 ist die Benutzeroberfläche nach dem Ausführen einer Zugkombination bestehend aus den Notationen (R‘ U L D‘ F) dargestellt. Um eine Ebene manuell zu rotieren, berührt der Benutzer die Ebene des virtuellen Zauberwürfels, die rotiert werden soll, und führt im Anschluss eine Wischbewegung in die Richtung aus, in die sich die Ebene rotieren soll. Hierbei erfolgt die Rotation in dieselbe Richtung, in die der Benutzer die Wischbewegung ausgeführt hat.

In Tabelle 10 sind die Notationen der Rotationszüge und die dazugehörigen Wischfunktionen aufgelistet. Die Rotationsaktionen lassen sich in drei verschiedene Rotationsrichtungen zusammenfassen. Diese sind identisch mit den Rotationsrichtungen des kompletten Zauberwürfels (Abbildung 4).

Da sich die Seiten-Notationen beim Rotieren des kompletten Zauberwürfels nicht verändern, wird hierbei davon ausgegangen, dass die vordere Ebene immer zum Benutzer hinzeigt. Die Rotation wird ausgeführt, sobald der Benutzer die Berührungsaktion beendet, indem dieser den Finger vom Bildschirm anhebt.

Notationen	**Rotationsrichtung**	**Wischbewegung von**	**Wischbewegung nach**
F, b	Z	links	rechts
f, B	-Z	rechts	links
u, D	Y	links	rechts
U, d	-Y	rechts	links
R, l	X	oben	unten
r, L	-X	unten	oben

Tabelle 10: Notationen und entsprechende Rotationsrichtungen mit Angabe der auszuführenden Wischbewegungen

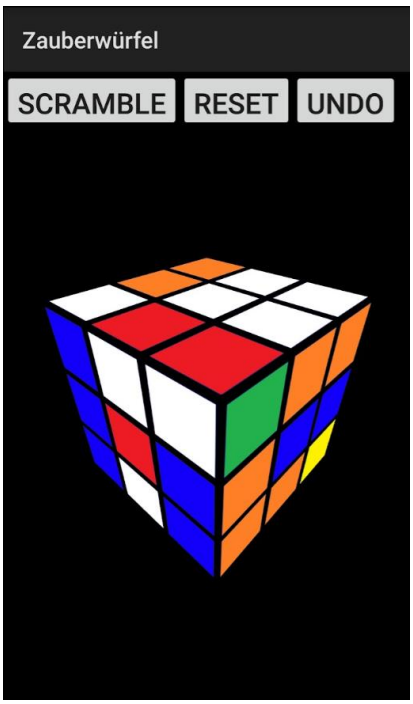

Abbildung 29: Zauberwürfel-Stadium nach Ausführen einer Zugkombination sowohl manuell als auch per Knopfdruck

Zusätzliche Funktionen

Im nächsten Schritt der Implementierung wurden die drei zusätzlichen Funktionen für die Zauberwürfel-Applikation implementiert. Diese sind zum einen das Zurücksetzen des Zauberwürfels, das automatische Vermischen des Zauberwürfels nach dem Zufallsprinzip sowie das schrittweise Zurücksetzen von zuvor ausgeführten Aktionen. Für diese Erweiterungen stehen dem Benutzer jeweils drei berührungsempfindliche Schaltflächen zur Verfügung.

Schrittweises Zurücksetzen des Zauberwürfels

Jede vom Benutzer ausgeführte Rotationsaktion wird im Anschluss in einer ArrayList mit dem Namen `movesList` abgespeichert.
Durch die Berührung der rechten berührungsempfindlichen Schaltfläche des Bildschirms mit der Beschriftung *undo*, wird die Aktion zum Zurücksetzen von zuvor rotierten Ebenen ausgeführt. Jede Rotationsaktion einer Ebene wurde zuvor wie oben beschrieben in einer Liste abgespeichert und wird entsprechend der Berührungsanzahl der Schaltfläche zurückgesetzt. Hierbei kann der Benutzer beliebig viele Aktionen zurücksetzen, bis der Zauberwürfel wieder seinen Anfangszustand erreicht hat. Nach dem Zurücksetzen von mehreren Aktionen, ohne

dass das Anfangsstadium des Zauberwürfels erreicht wurde, werden die neuen Rotationsaktionen zu den noch nicht zurückgesetzten Rotationsaktionen hinzugefügt.

Automatisches Vermischen des Zauberwürfels

Die linke berührungsempfindliche Schaltfläche mit der Beschriftung *scramble* führt die Aktion zum automatischen Vermischen des Zauberwürfels nach dem Zufallsprinzip aus. Hierbei wird eine zufällige Ebene automatisch rotiert und der Zauberwürfel wird somit vermischt. Der Benutzer hat im Anschluss an die Ausführung dieser Aktion die Möglichkeit, den Zauberwürfel manuell zu lösen oder eine weitere Aktion auszuführen.

Zurücksetzen des Zauberwürfels

Sobald die berührungsempfindliche Schaltfläche mit der Beschriftung *reset* vom Benutzer berührt wurde, wird die Methode `resetCube` in der `GLShapeJO` Klasse für jeden einzelnen der 27 Würfel aufgerufen. Innerhalb dieser Methode wird daraufhin der aktuelle Rotationsstatus des Würfels auf den Anfangswert 0 gesetzt und die Methode `updateRotate` zum Setzen des Rotationsstatus aufgerufen. Im Anschluss wird die Liste der zuvor ausgeführten Aktionen geleert. Der Zauberwürfel wird somit wieder in seinen Anfangszustand zurückgesetzt.

6 Zusammenfassung

In diesem Kapitel werden die Ergebnisse und die generelle Vorgehensweise der Abschlussarbeit zusammengefasst. Zum Schluss wird ein Ausblick auf eventuelle Erweiterungsmöglichkeiten der Zauberwürfel-Applikation erläutert.

Bei der Implementierung wurde die bereitgestellte Android OpenGL ES-Bibliothek als Grundlage für die Entwicklung der Zauberwürfel-Applikation verwendet und in entsprechenden Fällen erweitert und angepasst.

6.1 Ergebnisse

Der erste Schritt der Ausarbeitung war, sich mit der Struktur und der Funktionsweise der bereitgestellten Android OpenGL-Bibliothek vertraut zu machen, um zu verstehen, welche Funktionen bereits implementiert wurden und um welche diese gegebenenfalls ergänzt werden muss.

Im nächsten Schritt wurde zunächst die Grundstruktur des Zauberwürfels implementiert und dieser auf der Benutzeroberfläche positioniert. Damit der Zauberwürfel nicht mehr nur statisch von einer einzigen Seite aus betrachtet werden kann und die weiteren Funktionen leichter implementiert werden können, wurden die entsprechenden Berührungsaktionen auf dem Multitouch-Bildschirm eines mobilen Endgerätes hinzugefügt. Hierdurch hatte der Benutzer bereits die Möglichkeit, den Zauberwürfel, um seine eigenen Achsen zu rotieren und diesen somit von mehreren Seiten gleichzeitig zu betrachten.

Im weiteren Verlauf der Ausarbeitung wurden die Berührungsaktionen zum Rotieren des Zauberwürfels durch die Unterteilung der einzelnen Zauberwürfel-Seiten in neun Ebenen und der Richtungsbestimmung der Wischaktionen eines Benutzers erweitert. Durch diese Implementierung wäre es zu diesem Zeitpunkt theoretisch möglich gewesen, die entsprechenden Zauberwürfel-Ebenen zu rotieren.

Da während der Ausarbeitung festgestellt wurde, dass die Rotationen eines Objektes in Android OpenGL ES nicht kumulativ sind, wurde hier nach einer alternativen Möglichkeit gesucht. Aus dieser Lösungsfindung heraus wurde eine Zauberwürfel Status Maschine entwickelt, mit der anhand des vorherigen Zustandes

eines Objektes die anschließende Rotation bestimmt und das Objekt in die richtige Position rotiert werden konnte.

Im Anschluss an diesen Schritt der Implementierung wurden die korrekten Rotationsanimationen implementiert und die Anwendung wurde durch weitere zusätzliche Funktionalitäten, wie das vollständige Zurücksetzen des Zauberwürfels in den Anfangszustand, das schrittweise Zurücksetzen einer zuvor ausgeführten Rotationsaktion sowie das automatische Vermischen des Zauberwürfels erweitert.

6.2 Ausblick

Die Zauberwürfel-Applikation stellt eine gute Grundlage für weitere Erweiterungen zur Verfügung. Aktuell wurden ausschließlich die Grundfunktionalitäten eines Zauberwürfels implementiert. Eine denkbare Erweiterungsmöglichkeit wäre zum Beispiel eine Funktionalität, in der dem Benutzer die bestmögliche Zugkombination angezeigt wird, um den Zauberwürfel zu lösen. Denkbar wäre hier die Darstellung der bereits abgespeicherten ausgeführten Rotationsaktionen in negierter Reihenfolge in beispielsweise einem Textfeld oder in einer Benachrichtigungsfunktion.

Eine weitere interessante Möglichkeit wäre die Rotation einer Zauberwürfel-Ebene mithilfe von verschiedenen berührungsempfindlichen Schaltflächen, die entsprechend mit den erläuterten Notationen aus dem Kapitel *„2.1.2 Notationen“*, beschriftet sind. Durch diese Erweiterung könnte der Benutzer schneller eine gegebene Zugkombination ausführen, welche zum einen für das Vermischen des Zauberwürfels als auch für das Erlernen einer Lösungsmethode einen großen Anwendungszweck darstellen würde.

A Anhang

A1 Programmcode

GLAnimatorFactoryJO

```
package de.thkoeln.joltzen.bachelorarbeit.zauberwuerfel;

import android.animation.ObjectAnimator;
import android.animation.TypeEvaluator;
import android.animation.ValueAnimator;

import de.thkoeln.cvogt.android.opengl_utilities.GLShapeCV;

public class GLAnimatorFactoryJO {

    private static final float[] ZERO_ROTATION = new float[3];

    /**
     * Makes an animator to let the shape make a linear rotation around
a axis.
     * Adds the new animator to the animators of the given shape.
     *
     * @param shape           The shape to animate.
     * @param angleToTraverse The rotation angle to traverse.
     * @param axis            The rotation axis.
     * @param duration        The duration of the animation.
     * @param repeatCount     The number of times the animation shall be
repeated.
     * @param reverse         true if the animation shall be reversed.
     * @return The newly generated animator.
     */

    public static ObjectAnimator addLinearAnimatorRot(GLShapeCV shape,
float angleToTraverse, int axis, int duration, int repeatCount, boolean
reverse) {
        EvaluatorLinearTransformer eval = new EvaluatorLinearTransform-
er();

        float[] rotateAxisLast = new float[3];
        rotateAxisLast[axis] = angleToTraverse;
        ObjectAnimator animator = ObjectAnimator.ofObject(shape, "ro-
tateAnimate", eval, ZERO_ROTATION, rotateAxisLast);
        animator.setDuration(duration);
        animator.setRepeatCount(repeatCount);
        if (reverse)
            animator.setRepeatMode(ValueAnimator.REVERSE);
        shape.addAnimator(animator);
        return animator;
    }


    /**
     * Class that defines a TypeEvaluator for a linear rotation around a
axis
     */
    private static class EvaluatorLinearTransformer implements TypeEval-
```

```
uator<float[]> {
        public float[] evaluate(float f, float[] from, float[] to) {
            float[] currentAngle = new float[from.length];
            for (int i = 0; i < from.length; i++) {
                currentAngle[i] = from[i] * (1 - f) + to[i] * f;
            }
            return currentAngle;
        }
    }

}
```

GLRendererJO

```
package de.thkoeln.joltzen.bachelorarbeit.zauberwuerfel;

import android.opengl.Matrix;
import android.util.Log;

import java.util.Arrays;

import de.thkoeln.cvogt.android.opengl_utilities.GLRendererCV;
import de.thkoeln.cvogt.android.opengl_utilities.GraphicsUtilsCV;


public class GLRendererJO extends GLRendererCV {

    /**
     * vertical angle value to calculate the new camera position after
touching the screen.
     */
    private float vertical;

    /**
     * horizontal angle value to calculate the new camera position after
touching the screen.
     */
    private float horizontal;

    /**
     * Layer of the cube that should be rotated
     */
    public int[] layer = new int[2];

    /**
     * Side index of the cube that is the closest to the camera
     */
    private int sideIndex;

    /**
     * Normals of the plane that defines a facing direction.
     */
    private static final float[][] normalizedPlane = {
            {-1.1f, 0.0f, 0.0f},
            {1.1f, 0.0f, 0.0f},
            {0.0f, -1.1f, 0.0f},
            {0.0f, 1.1f, 0.0f},
            {0.0f, 0.0f, 1.1f},
            {0.0f, 0.0f, -1.1f},
    };
```

```
    /**
     * Shoots a ray based on the screen coordinates. If the ray hits a
cube side, the side and the collision point will be stored.
     * The calculation is based on the following article:
http://antongerdelan.net/opengl/raycasting.html
     */
    public int shootRay(float x, float y) {

        //Get the view projection matrix and the projection matrix from
the GLRendererCV class
        final float[] projectionMatrix = getProjectionMatrix();
        final float[] viewProjectionMatrix = getViewProjectionMatrix();

        final float[] invertedMatrix = new float[16];

        // create homogeneous clip coordinates which indicates the start
point of the ray
        float[] ray_clip = GraphicsUtilsCV.homogeneousCoordsForPoint(new
float[]{x, y, -1f});

        // calculate the start point of the ray by transforming the ho-
mogeneous coordinates with the inverse projection matrix
        Matrix.invertM(invertedMatrix, 0, projectionMatrix, 0);
        Matrix.multiplyMV(ray_clip, 0, invertedMatrix, 0,
ray_clip.clone(), 0);

        // set the direction of the line of sight into the view port
        float[] ray_eye = GraphicsUtilsCV.homogeneousCoordsForVector(new
float[]{x, y, -1f});

        // to convert from the view space to the world space, the homo-
geneous coordinates are transformed with the inverse view projection
matrix
        Matrix.invertM(invertedMatrix, 0, viewProjectionMatrix, 0);
        Matrix.multiplyMV(ray_eye, 0, invertedMatrix, 0,
ray_eye.clone(), 0);

        float[] ray_world =
GraphicsUtilsCV.coordsFromHomogeneous(ray_eye);

        //normalise the vector ray_world because z was set to -1, so the
ray needs to be normalised.
        ray_world = GraphicsUtilsCV.getNormalizedCopy(ray_world);

        return getCollision(x, y, ray_world);
    }

    /**
     * The method returns the collision point of the ray and the plane
and returns the side index of the touched cube side
     *
     * @param x
     * @param y
     * @param ray_point
     * @return
     */
    private int getCollision(float x, float y, float[] ray_point) {
        // get the current camera positions from the GLRendererCV class
        float[] eyes = new float[]{getEyeX(), getEyeY(), getEyeZ()};

        // initial touched side, -1 means no side is touched
        int side = -1;
```

```
        // Find intersection point with the planes
        for (int i = 0; i < normalizedPlane.length; i++) {
            //get the normal of the plane that defines the facing direc-
tion
            final float[] planeNorm = normalizedPlane[i];

            // angle between the ray and the plane
            final float dotRayPlane =
GraphicsUtilsCV.dotProduct(ray_point, planeNorm);
            // angle between the camera and the plane
            final float dotEyePlane = GraphicsUtilsCV.dotProduct(eyes,
planeNorm);

            //distance between the ray origin and the plane
            float distance = -(dotEyePlane - 1f) / dotRayPlane;

            final float[] intersectionPoint = new float[3];
            for (int j = 0; j < 3; j++) {
                intersectionPoint[j] = ray_point[j] * distance +
eyes[j];
            }

            //check if the ray points intersect with a object
            for (float v : intersectionPoint) {
                if (v >= 1.0f || v <= -1.0f) {
                    distance = 0;
                    break;
                }
            }

            // store the side of the cube with the shortest distance
            if (distance < 0) {
                //calculate the layer that should be rotated
                getLayers(x, y);
                side = this.sideIndex;
            }
        }
        return side;
    }

    /**
     * Calculates the layer of the cube that should be rotated
     *
     * @param x
     * @param y
     */
    private void getLayers(float x, float y) {
        //get the current camera positions from the GLRendererCV class
        float[] eyes = new float[]{getEyeX(), getEyeY(), getEyeZ()};
        // get the absolute value of the camera position
        float[] absEyes = new float[]{Math.abs(eyes[0]),
Math.abs(eyes[1]), Math.abs(eyes[2])};
        // get the index of the maximum value of the camera position
        int greatestEye = greatestValue(absEyes);
        // set the index of the side that is touched
        sideIndex = 2 * greatestEye + (eyes[greatestEye] < 0 ? 1 : 0);
        // normalize the camera position
        float[] normEyes = normaliseEyes(sideIndex, eyes);
        // calculate the layer of the cube that is touched
        /**
         * Touched position normalized by the current camera position
         */
        float[] touchPoint = new float[]{calculateTouchedX(normEyes, x),
```

```
calculateTouchedY(normEyes, y)};

        if (touchPoint[0] >= 0.3f) {
            layer[0] = sideIndex == 5 || sideIndex == 1 ? 0 : 2;
        } else if (touchPoint[0] < -0.3f) {
            layer[0] = sideIndex == 5 || sideIndex == 1 ? 2 : 0;
        } else {
            layer[0] = 1;
        }

        if (touchPoint[1] >= 0.3f) {
            layer[1] = sideIndex == 3 ? 0 : 2;
        } else if (touchPoint[1] < -0.3f) {
            layer[1] = sideIndex == 3 ? 2 : 0;
        } else {
            layer[1] = 1;
        }

    }

    /**
     * Calculates the x coordinate of the touched point on the cube
     *
     * @param normEyes
     * @param x
     * @return
     */
    private float calculateTouchedX(float[] normEyes, float x) {
        float deltaE = (float) Math.atan(normEyes[0] / normEyes[2]);
        float deltaT = (float) Math.asin(x / 3f);
        if (deltaE < 0) {
            return (deltaE / 0.77f) + ((-deltaE / 0.77f) * 2.5f + 4f) *
deltaT;
        } else {
            return (deltaE / 0.77f) - ((-deltaE / 0.77f) * 2.5f - 4f) *
deltaT;
        }

    }

    /**
     * Calculates the y coordinate of the touched point on the cube
     * +
     *
     * @param normEyes
     * @param y
     * @return
     */
    private float calculateTouchedY(float[] normEyes, float y) {
        float deltaE = normEyes[1] / normEyes[2];
        float deltaT = (float) Math.asin((y / 3f) * (7.4f / 4f));
        if (deltaE < 0) {
            return (deltaE / 0.77f) + ((-deltaE / 0.77f) * 2.5f + 4f) *
deltaT;
        } else {
            return (deltaE / 0.77f) - ((-deltaE / 0.77f) * 2.5f - 4f) *
deltaT;
        }

    }

    /**
     * Returns the index of the maximum value of the array
```

```
     *
     * @param values
     * @return
     */
    private int greatestValue(float[] values) {
        int greatest = 0;
        for (int i = 0; i < values.length; i++) {
            greatest = values[i] > values[greatest] ? i : greatest;
        }
        return greatest;
    }

    /**
     * Normalizes the camera position
     *
     * @param sideIndex
     * @param eyes
     * @return
     */
    private float[] normaliseEyes(int sideIndex, float[] eyes) {
        float[] normEyes = new float[3];
        int si = sideIndex & 0xE;
        switch (si) {
            case 0:
                normEyes[0] = -eyes[2];
                normEyes[1] = eyes[1];
                normEyes[2] = eyes[0];
                break;
            case 2:
                normEyes[0] = eyes[0];
                normEyes[1] = -eyes[2];
                normEyes[2] = eyes[1];
                break;
            case 4:
                normEyes = eyes;
        }
        if (sideIndex % 2 == 1) {
            for (int i = 0; i < 3; i++) {
                normEyes[i] = -normEyes[i];
            }
        }
        return normEyes;
    }

    /**
     * Calculates the camera position around the SurfaceView and sets
the values eyeX, eyeY and eyeZ in the GLRendererCV class.
     * Calculation according to
https://www.spieleprogrammierung.net/2010/02/mathematik-der-3d-
programmierung-teil-3_18.html
     */
    public void calculateEyeCoordinates(float x, float y) {
        this.vertical += Math.PI * y;
        this.horizontal += Math.PI * x;

        float eyeX = ((float) (-5.0f * Math.cos(vertical) *
(Math.sin(horizontal))));
        float eyeY = ((float) (5.0f * Math.sin(vertical)));
        float eyeZ = ((float) (5.0f * Math.cos(vertical) *
Math.cos(horizontal)));
        setEyeCoordinates(eyeX, eyeY, eyeZ);
    }
}
```

GLShapeFactoryJO

```
package de.thkoeln.joltzen.bachelorarbeit.zauberwuerfel;


import android.graphics.Bitmap;

import de.thkoeln.cvogt.android.opengl_utilities.GLShapeFactoryCV;


public class GLShapeFactoryJO extends GLShapeFactoryCV {

    /**
     * copy of the makeCube method from GLShapeFactoryCV to create a
cube of the GLShapeJO class
     *
     * @param id
     * @param textures
     * @return
     */
    public static GLShapeJO makeCube(String id, Bitmap[] textures) {
        if (textures == null || textures.length != 6) return null;
        float edgeLength = 1.0f;
        float leftFrontUpperCorner_X = -edgeLength / 2.0f;   // set val-
ues such that model coordinates (0,0,0) are the center of the cube
        float leftFrontUpperCorner_Y = edgeLength / 2.0f;
        float leftFrontUpperCorner_Z = edgeLength / 2.0f;
        GLShapeJO shape = new GLShapeJO(id, trianglesForTextured-
Cube(leftFrontUpperCorner_X, leftFrontUpperCorner_Y, leftFrontUpper-
Corner_Z, edgeLength, textures));
        return shape;
    }

}
```

GLShapeJO

```
package de.thkoeln.joltzen.bachelorarbeit.zauberwuerfel;


import android.animation.ObjectAnimator;
import android.opengl.Matrix;

import de.thkoeln.cvogt.android.opengl_utilities.GLAnimatorFactoryCV;
import de.thkoeln.cvogt.android.opengl_utilities.GLShapeCV;
import de.thkoeln.cvogt.android.opengl_utilities.GLTriangleCV;

public class GLShapeJO extends GLShapeCV {


    /**
     * Update the rotation matrix based on the rotation angles from the
angleRotation array
     */
    private int currentRotationStatus = 0;
```

```
    /**
     * Rotation status of the sub cube before it was rotated
     */
    private int preRotationStatus;

    /**
     * Rotation angles for the axes x, y and z
     */
    private final float[][] rotationAngles = {
            {0, 0, 0},
            {0, 90, 0},
            {0, 180, 0},
            {0, 270, 0},

            {0, 0, 90},
            {0, 90, 90},
            {0, 180, 90},
            {0, 270, 90},

            {90, 180, 0},
            {90, 180, 270},
            {90, 180, 180},
            {90, 180, 90},

            {0, 90, 270},
            {0, 180, 270},
            {0, 270, 270},
            {0, 0, 270},

            {0, 90, 180},
            {0, 270, 180},
            {0, 0, 180},
            {0, 180, 180},

            {90, 0, 0},
            {90, 0, 90},
            {90, 0, 180},
            {90, 0, 270},
    };

    /**
     * States for the rotation of the sub cube
     */
    private final int[][] moveStatus = {
            {10, 20, 3, 1, 15, 4},
            {12, 5, 0, 2, 23, 9},
            {22, 8, 1, 3, 6, 13},
            {7, 14, 2, 0, 11, 21},

            {9, 21, 7, 5, 0, 18},
            {1, 16, 4, 6, 20, 8},
            {23, 11, 5, 7, 19, 2},
            {17, 3, 6, 4, 10, 22},

            {2, 18, 9, 11, 5, 14},
            {13, 4, 10, 8, 1, 17},
            {19, 0, 11, 9, 12, 7},
            {6, 15, 8, 10, 16, 3},

            {16, 1, 15, 13, 22, 10},
            {21, 9, 12, 14, 2, 19},
            {3, 17, 13, 15, 8, 20},
```

```
            {11, 23, 14, 12, 18, 0},

            {5, 12, 18, 19, 21, 11},
            {14, 7, 19, 18, 9, 23},
            {8, 22, 17, 16, 4, 15},
            {20, 10, 16, 17, 13, 6},

            {0, 19, 21, 23, 14, 5},
            {4, 13, 22, 20, 3, 16},
            {18, 2, 23, 21, 7, 12},
            {15, 6, 20, 22, 17, 1},
    };


    /**
     * Constructor for the GLShapeJO class which creates a sub cube
     *
     * @param shapeId
     * @param textureId
     */
    public GLShapeJO(String shapeId, GLTriangleCV[] textureId) {
        super(shapeId, textureId);
    }

    /**
     * Set the rotation status of the sub cube based on the given move
index and the current rotation status of the sub cube
     *
     * @param moveIndex
     */
    public void rotate(int moveIndex) {
        preRotationStatus = currentRotationStatus;
        currentRotationStatus = moveSta-
tus[currentRotationStatus][moveIndex];
    }

    /**
     * The method resets all sub cubes to the initial rotation status
     */
    public void resetCube() {
        currentRotationStatus = 0;
        updateRotate();
    }


    /**
     * Update the rotation matrix based on the rotation angles from the
angleRotation array
     */

    public void updateRotate() {
        float[] rotationAngle = rotationAngles[currentRotationStatus];
        float[] rotationMatrix = this.getRotationMatrix();
        Matrix.setIdentityM(rotationMatrix, 0);

        // rotate for each single axis by the new angle
        for (int i = 0; i < 3; i++) {
            float[] axis = new float[3];
            axis[i] = 1;
            Matrix.rotateM(rotationMatrix, 0, rotationAngle[i], axis[0],
axis[1], axis[2]);
        }
        this.setRotationMatrix(rotationMatrix);
```

```
    }

    /**
     * Setter function to set the animated values. The function is
called by the ObjectAnimator inside the GLAnimatorFactoryJO class based
on the property name.
     *
     * @param rotationAngle
     * @return
     */
    public GLShapeJO setRotateAnimate(float[] rotationAngle) {
        float[] rotationMatrix = this.getRotationMatrix();
        Matrix.setIdentityM(rotationMatrix, 0);
        float[] start = rotationAngles[preRotationStatus];

        float[] axis = new float[3];
        for (int i = 0; i < axis.length; i++) {
            if (rotationAngle[i] != 0) {
                axis[i] = 1;
                Matrix.rotateM(rotationMatrix, 0, rotationAngle[i], ax-
is[0], axis[1], axis[2]);
                axis[i] = 0;
            }
        }
        for (int i = 0; i < axis.length; i++) {
            axis[i] = 1;
            Matrix.rotateM(rotationMatrix, 0, start[i], axis[0], ax-
is[1], axis[2]);
            axis[i] = 0;
        }

        this.setRotationMatrix(rotationMatrix);
        return this;
    }

    /**
     * Animate the rotation of the layer of the magic cube around a
point.
     *
     * @param axis
     * @param clockwise
     */
    public void animate(int axis, boolean clockwise) {
        int duration = 300;

        float[] fromRotationAxis = new float[3];
        float[] toRotationAxis = new float[3];
        fromRotationAxis[axis] = -1;
        toRotationAxis[axis] = 1;

        ObjectAnimator animPath = GLAnimatorFacto-
ryCV.addAnimatorArcPathAroundAxis(this, fromRotationAxis, toRotationAx-
is, 90f, clockwise, duration, 0);
        addAnimator(animPath);
        animPath.start();

        ObjectAnimator animRot = GLAnimatorFacto-
ryJO.addLinearAnimatorRot(this, clockwise ? -90f : 90f, axis, duration,
0, false);
        addAnimator(animRot);
        animRot.start();
```

```
    }
}
```

Moves

```
package de.thkoeln.joltzen.bachelorarbeit.zauberwuerfel;

public class Moves {
    /**
     * The axis of the previous action (0 = x, 1 = y, 2 = z)
     */
    public int axis;

    /**
     * The layer of the previous action
     */
    public int layer;

    /**
     * The direction of the previous action (true = clockwise, false =
counterclockwise)
     */
    public boolean clockwise;

    /**
     * Constructor for the Moves class which stores the axis, part and
direction of the previous done rotation
     *
     * @param axis
     * @param layer
     * @param clockwise
     */
    public Moves(int axis, int layer, boolean clockwise) {
        this.axis = axis;
        this.layer = layer;
        this.clockwise = clockwise;
    }
}
```

MainActivity

```
package de.thkoeln.joltzen.bachelorarbeit.zauberwuerfel;

import android.app.Activity;
import android.graphics.Bitmap;
import android.os.Bundle;
import android.util.Log;
import android.view.MotionEvent;
import android.widget.Button;
import android.widget.LinearLayout;

import java.util.Random;

import de.thkoeln.cvogt.android.opengl_utilities.GLSurfaceViewCV;
import de.thkoeln.cvogt.android.opengl_utilities.GraphicsUtilsCV;
import de.thkoeln.cvogt.android.opengl_utilities.TextureBitmapsCV;
```

```
public class MainActivity extends Activity {

    /**
     * The surface view to which the shape is currently attached.
     */
    private GLSurfaceViewCV glSurfaceView;

    /**
     * The registered renderer.
     */
    private GLRendererJO renderer;

    /**
     * previously touched x coordinate
     */
    private float previousX;

    /**
     * previously touched y coordinate
     */
    private float previousY;

    /**
     * Information whether the user has touched the Rubik's cube to ro-
tate it or not
     */
    private boolean rotatingLayers = false;

    /**
     * The textures for the Rubik's cube
     */
    private final Bitmap[] textures = new Bitmap[6];

    /**
     * The sub cubes of the Rubik's cube
     */
    private final GLShapeJO[] cubes = new GLShapeJO[27];

    /**
     * The 3 dimensional array of the sub cubes
     */
    public GLShapeJO[][][] cube = new GLShapeJO[3][3][3];

    /**
     * The 3x3x3 Rubik's cube
     */
    private RubiksCube rubiksCube;

    /**
     * The moves of the Rubik's cube
     */
    private Moves moves;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        TextureBitmapsCV.init(this);
        renderer = new GLRendererJO();
        glSurfaceView = new GLSurfaceViewCV(this, renderer, false);
        createRubiksCube(glSurfaceView);
        setContentView(glSurfaceView);
```

```
        LinearLayout buttonLayout = new LinearLayout(this);

        //Create a button to scramble the cube
        Button scramble = new Button(this);
        scramble.setText(R.string.scramble_action);
        scramble.setTextSize(25);
        scramble.setOnClickListener(v -> {
            // rotate one random layer of the cube
            Random random = new Random();
            rubiksCube.rotateSubCubes(random.nextInt(3), ran-
dom.nextInt(3), random.nextBoolean(), false);

        });

        //Create a button to undo the moves
        Button undo = new Button(this);
        undo.setText(R.string.undo_action);
        undo.setTextSize(25);
        undo.setOnClickListener(v -> {
            // delete the last move from the list
            if (rubiksCube.movesList.size() > 1) {
                moves = ru-
biksCube.movesList.get(rubiksCube.movesList.size() - 1);
                rubiksCube.rotateSubCubes(moves.axis, moves.layer,
!moves.clockwise, true);
            } else if (rubiksCube.movesList.size() == 1) {
                moves = rubiksCube.movesList.get(0);
                rubiksCube.rotateSubCubes(moves.axis, moves.layer,
!moves.clockwise, true);
            }
        });

        //Create a button to reset the cube
        Button reset = new Button(this);
        reset.setText(R.string.reset_action);
        reset.setTextSize(25);
        reset.setOnClickListener(v -> {
            // reset all shapes to their initial position
            for (GLShapeJO subCube : cubes) {
                subCube.resetCube();
            }
            //clear the moves list after reset
            while (rubiksCube.movesList.size() != 0) {
                rubiksCube.movesList.remove(rubiksCube.movesList.size()
- 1);
            }
        });

        //Add the buttons to the layout
        buttonLayout.addView(scramble);
        buttonLayout.addView(reset);
        buttonLayout.addView(undo);
        LinearLayout.LayoutParams params = new LinearLay-
out.LayoutParams(LinearLayout.LayoutParams.MATCH_PARENT, LinearLay-
out.LayoutParams.MATCH_PARENT);
        params.height = 150;
        this.addContentView(buttonLayout, params);
    }


    /**
     * Creates the Rubik's cube and adds it to the SurfaceView
     *
```

```
     * @param surfaceView
     */
    public void createRubiksCube(GLSurfaceViewCV surfaceView) {
        textures[0] = TextureBitmapsCV.get("red");
        textures[1] = TextureBitmapsCV.get("blue");
        textures[2] = TextureBitmapsCV.get("orange");
        textures[3] = TextureBitmapsCV.get("green");
        textures[4] = TextureBitmapsCV.get("white");
        textures[5] = TextureBitmapsCV.get("yellow");

        // Create 27 individual cubes
        for (int cubeIndex = 0; cubeIndex < 27; cubeIndex++) {
            cubes[cubeIndex] = GLShapeFacto-
ryJO.makeCube(Integer.toString(cubeIndex), textures);
        }

        // Sets the translation for the 27 individual cubes to a 3x3x3
cube and add it to the surfaceView
        int cubeIndex = 0;
        for (int x = 0; x < 3; x++) {
            for (int y = 0; y < 3; y++) {
                for (int z = 0; z < 3; z++) {
                    cube[x][y][z] = cubes[cubeIndex];
                    cube[x][y][z].setTrans(x - 1, y - 1, z - 1);
                    surfaceView.addShape(cube[x][y][z]);
                    cubeIndex++;
                }
            }
        }
        rubiksCube = new RubiksCube(cubes);
    }


    /**
     * This method is called when the user touches the screen.
     *
     * @param event
     * @return
     */
    @Override
    public boolean onTouchEvent(MotionEvent event) {

        // normalized coordinates from the touched x and y positions
        float normalizedX = ((2.0f * event.getX()) / glSur-
faceView.getWidth()) - 1.0f;
        float normalizedY = 1.0f - ((2.0f * event.getY()) / glSur-
faceView.getHeight() - 0.25f);

        float[] distance;
        switch (event.getAction()) {
            case MotionEvent.ACTION_DOWN: {
                // shoot a ray from the touch point into the scene to
get the position of the touched object
                int side = renderer.shootRay(normalizedX, normalizedY);

                //check if the cube is touched and set the start touch
position and rotateLayers to true
                rotatingLayers = (side != -1);

                // update the previously touched x and y positions
                previousX = event.getX();
                previousY = event.getY();
```

```
            }
            break;

            case MotionEvent.ACTION_MOVE: {
                if (!rotatingLayers) {
                    // distance between the current touch position and
the previous touch position
                    float deltaX = event.getX() - previousX;
                    float deltaY = event.getY() - previousY;

                    // calculate the camera view angles
                    renderer.calculateEyeCoordinates((deltaX / glSur-
faceView.getWidth()), (deltaY / glSurfaceView.getHeight()));
                    // update the previously touched x and y positions
                    previousX = event.getX();
                    previousY = event.getY();
                }


            }
            break;

            case MotionEvent.ACTION_UP: {
                if (rotatingLayers) {
                    // shoot a ray from the touch point into the scene
to get the position of the touched object
                    int side = renderer.shootRay(normalizedX, normal-
izedY);
                    //calculate the swipe direction
                    distance = new float[]{previousX - event.getX(),
previousY - event.getY()};
                    float[] absoluteDistance = {Math.abs(distance[0]),
Math.abs(distance[1])};

                    // 0 = right, 1 = left, 2 = top, 3 = bottom, 4 =
front, 5 = back
                    switch (side) {
                        case 0:
                            if (absoluteDistance[0] < absoluteDis-
tance[1]) {
                                rubiksCube.rotateSubCubes(2, render-
er.layer[0], distance[1] < 0f, false);
                            } else {
                                rubiksCube.rotateSubCubes(1, render-
er.layer[1], distance[0] >= 0f, false);
                            }
                            break;
                        case 1:
                            if (absoluteDistance[1] < absoluteDis-
tance[0]) {
                                rubiksCube.rotateSubCubes(1, render-
er.layer[1], distance[0] >= 0f, false);
                            } else {
                                rubiksCube.rotateSubCubes(2, render-
er.layer[0], distance[1] >= 0f, false);
                            }
                            break;
                        case 2:
                            if (absoluteDistance[1] < absoluteDis-
tance[0]) {
                                rubiksCube.rotateSubCubes(2, render-
er.layer[1], distance[0] < 0f, false);
                            } else {
```

```
                                rubiksCube.rotateSubCubes(0, render-
er.layer[0], distance[1] >= 0f, false);
                            }
                            break;
                        case 3:
                            if (absoluteDistance[1] < absoluteDis-
tance[0]) {
                                rubiksCube.rotateSubCubes(2, render-
er.layer[1], distance[0] >= 0f, false);
                            } else {
                                rubiksCube.rotateSubCubes(0, render-
er.layer[0], distance[1] >= 0f, false);

                            }
                            break;
                        case 4:
                            if (absoluteDistance[0] < absoluteDis-
tance[1]) {
                                rubiksCube.rotateSubCubes(0, render-
er.layer[0], distance[1] >= 0f, false);
                            } else {
                                rubiksCube.rotateSubCubes(1, render-
er.layer[1], distance[0] >= 0f, false);
                            }
                            break;
                        case 5:
                            if (absoluteDistance[1] < absoluteDis-
tance[0]) {
                                rubiksCube.rotateSubCubes(1, render-
er.layer[1], distance[0] >= 0f, false);
                            } else {
                                rubiksCube.rotateSubCubes(0, render-
er.layer[0], distance[1] < 0f, false);
                            }
                            break;

                    }
                }
                rotatingLayers = false;
            }
            break;
        }
        return true;
    }

}
```

RubiksCube

```
package de.thkoeln.joltzen.bachelorarbeit.zauberwuerfel;

import java.util.ArrayList;

public class RubiksCube {

    /**
     * The sub cubes of the Rubik's cube
     */
    private GLShapeJO[] cubes;
```

```
    /**
     * The moves of the Rubik's cube
     */
    public ArrayList<Moves> movesList = new ArrayList<>();

    /**
     * Permutation of the rubik's cube
     */
    int[] permutation = new int[27];

    /**
     * Constructor for the Rubik's cube
     *
     * @param cubes
     */
    public RubiksCube(GLShapeJO[] cubes) {
        this.cubes = cubes;
        for (int i = 0; i < 27; i++) {
            permutation[i] = i;
        }
    }

    /**
     * Method to update the permutation of the Rubik's cube and starts
the animation for the selected layer
     *
     * @param axis
     * @param layer
     * @param clockwise
     * @param revert
     */
    public void rotateSubCubes(int axis, int layer, boolean clockwise,
boolean revert) {
        int[] cubeIndex = null;
        int[] cubeIndexMove = null;
        int moveIndex = 0;

        if (axis == 0) {
            moveIndex = clockwise ? 0 : 1;
            switch (layer) {
                //left layer
                case 0:
                    cubeIndex = new int[]{0, 1, 2, 3, 4, 5, 6, 7, 8};
                    cubeIndexMove = clockwise ? new int[]{6, 3, 0, 7, 4,
1, 8, 5, 2} : new int[]{2, 5, 8, 1, 4, 7, 0, 3, 6};
                    break;
                //middleX layer
                case 1:
                    cubeIndex = new int[]{9, 10, 11, 12, 13, 14, 15, 16,
17};
                    cubeIndexMove = clockwise ? new int[]{15, 12, 9, 16,
13, 10, 17, 14, 11} : new int[]{11, 14, 17, 10, 13, 16, 9, 12, 15};
                    break;
                //right layer
                case 2:
                    cubeIndex = new int[]{18, 19, 20, 21, 22, 23, 24,
25, 26};
                    cubeIndexMove = clockwise ? new int[]{24, 21, 18,
25, 22, 19, 26, 23, 20} : new int[]{20, 23, 26, 19, 22, 25, 18, 21, 24};
                    break;
            }
        }
```

```
        if (axis == 1) {
            moveIndex = clockwise ? 2 : 3;
            switch (layer) {
                //bottom layer
                case 0:
                    cubeIndex = new int[]{0, 1, 2, 9, 10, 11, 18, 19,
20};
                    cubeIndexMove = clockwise ? new int[]{2, 11, 20, 1,
10, 19, 0, 9, 18} : new int[]{18, 9, 0, 19, 10, 1, 20, 11, 2};
                    break;
                //middleY layer
                case 1:
                    cubeIndex = new int[]{3, 4, 5, 12, 13, 14, 21, 22,
23};
                    cubeIndexMove = clockwise ? new int[]{5, 14, 23, 4,
13, 22, 3, 12, 21} : new int[]{21, 12, 3, 22, 13, 4, 23, 14, 5};
                    break;
                //top layer
                case 2:
                    cubeIndex = new int[]{6, 7, 8, 15, 16, 17, 24, 25,
26};
                    cubeIndexMove = clockwise ? new int[]{8, 17, 26, 7,
16, 25, 6, 15, 24} : new int[]{24, 15, 6, 25, 16, 7, 26, 17, 8};
                    break;

            }
        }
        if (axis == 2) {
            moveIndex = clockwise ? 4 : 5;
            switch (layer) {
                //front layer
                case 0:
                    cubeIndex = new int[]{2, 5, 8, 11, 14, 17, 20, 23,
26};
                    cubeIndexMove = clockwise ? new int[]{20, 11, 2, 23,
14, 5, 26, 17, 8} : new int[]{8, 17, 26, 5, 14, 23, 2, 11, 20};
                    break;
                //middleZ layer
                case 1:
                    cubeIndex = new int[]{1, 4, 7, 10, 13, 16, 19, 22,
25};
                    cubeIndexMove = clockwise ? new int[]{19, 10, 1, 22,
13, 4, 25, 16, 7} : new int[]{7, 16, 25, 4, 13, 22, 1, 10, 19};
                    break;
                //back layer
                case 2:
                    cubeIndex = new int[]{0, 3, 6, 9, 12, 15, 18, 21,
24};
                    cubeIndexMove = clockwise ? new int[]{18, 9, 0, 21,
12, 3, 24, 15, 6} : new int[]{6, 15, 24, 3, 12, 21, 0, 9, 18};
                    break;
            }
        }

        //if revert is true, the move is reversed. if not, the move is
added to the move list
        if (revert) {
            movesList.remove(movesList.size() - 1);
        } else {
            movesList.add(new Moves(axis, layer, clockwise));
        }

        if (cubeIndex != null) {
```

```
            for (int ci : cubeIndexMove) {
                GLShapeJO subCube = cubes[permutation[ci]];
                subCube.rotate(moveIndex);
                subCube.animate(axis, clockwise);
            }

            //Update the permutation after the rotation
            int[] pCopy = new int[permutation.length];
            System.arraycopy(permutation, 0, pCopy, 0, permuta-
tion.length);
            for (int i = 0; i < cubeIndex.length; i++) {
                permutation[cubeIndex[i]] = pCopy[cubeIndexMove[i]];
            }

        }
    }
}
```

A2 Programmcode Anpassungen in der Bibliothek

GLRendererCV

```
public class GLRendererCV implements GLSurfaceView.Renderer {


    /**
     * JASON OLTZEN added this method on the 13th of November, 2022.
     * Get a copy of the current view projection matrix.
     */

    public float[] getProjectionMatrix() {
        return projectionMatrix.clone();
    }


    /**
     * JASON OLTZEN added this method on the 13th of November, 2022.
     * Set the current Position of the camera based.
     */
    public void setEyeCoordinates(float eyeX, float eyeY, float eyeZ) {
        this.eyeX = eyeX;
        this.eyeY = eyeY;
        this.eyeZ = eyeZ;
        updateViewProjectionMatrix();
    }

    /**
     * JASON OLTZEN added this method on the 13th of November, 2022.
     * Get the current Position of the camera: X Value.
     */
    public float getEyeX() {
        return eyeX;
    }

    /**
     * JASON OLTZEN added this method on the 13th of November, 2022.
     * Get the current Position of the camera: Y Value.
```

```
     */
    public float getEyeY() {
        return eyeY;
    }

    /**
     * JASON OLTZEN added this method on the 13th of November, 2022.
     * Get the current Position of the camera: Z Value.
     */
    public float getEyeZ() {
        return eyeZ;
    }
}
```

TextureBitmapsCV

```
public class TextureBitmapsCV {


  public static Bitmap get(String name) {
    Bitmap bitmap = textureBitmaps.get(name);
    // if bitmap has already been created, return it
    if (bitmap != null)
        return bitmap;
    // create the bitmap and add it to the HashMap
    switch (name) {
        //JASON OLTZEN added the following cases for the Rubik's Cube on
the 19th of October, 2022.
        case "yellow":
            textureBitmaps.put("yellow", BitmapFacto-
ry.decodeResource(context.getResources(), R.raw.yellow));
            break;
        case "white":
            textureBitmaps.put("white", BitmapFacto-
ry.decodeResource(context.getResources(), R.raw.white));
            break;
        case "blue":
            textureBitmaps.put("blue", BitmapFacto-
ry.decodeResource(context.getResources(), R.raw.blue));
            break;
        case "green":
            textureBitmaps.put("green", BitmapFacto-
ry.decodeResource(context.getResources(), R.raw.green));
            break;
        case "red":
            textureBitmaps.put("red", BitmapFacto-
ry.decodeResource(context.getResources(), R.raw.red));
            break;
        case "orange":
            textureBitmaps.put("orange", BitmapFacto-
ry.decodeResource(context.getResources(), R.raw.orange));
            break;
    }
    return textureBitmaps.get(name);
}

}
```

Abkürzungsverzeichnis

API Application Programming Interface

NASA National Aeronautics and Space Administration

Verwendete Ressourcen

Sketchbook Pro

Für die Erstellung der Zauberwürfel Zeichnungen wurde die Applikation Sketchbook Pro verwendet.
Url: https://www.sketchbook.com/

Creatly

Die Anwendungsfalldiagramme wurden mit dem Webtool Creatly erstellt.
Url: https://creately.com/

Google Spreadsheets

Für die Erstellung der Tabellen der Zauberwürfel Status Maschine wurde die Webapplikation Google Spreadsheets verwendet.
Url: https://docs.google.com/spreadsheets/

Lucid

Die Sequenzdiagramme und das Klassendiagramm wurden mit dem Webtool Lucid erstellt.
Url: https://lucid.app/

Abbildungsverzeichnis

Tabellenverzeichnis

Quellenverzeichnis

[AlRu22] Rudolph, A. (kein Datum). *3D-Programmierung (Mathematik) Teil 03: Polarkoordinaten*. Abgerufen am 15. 11 2022 von https://www.spieleprogrammierung.net/2010/02/mathematik-der-3d-programmierung-teil-3_18.html

[AnDe22] Developers, A. (kein Datum). *OpenGL ES*. Abgerufen am 10. 23 2022 von https://developer.android.com/develop/ui/views/graphics/opengl/about-opengl

[AnGe16] Gerdelan, A. (2. 10 2014). *Anton's OpenGL 4 Tutorials.* Kindle Amazon. Seite 140-153

[HaSe15] Serrano, H. (1. 11 2015). *What is Clipping in OpenGL*. Abgerufen am 6. 11 2022 von https://www.haroldserrano.com/blog/what-is-clipping-in-opengl

[KüTh21a] Künneth, T. (2021). *Android 11 Das Praxisbuch für App-Entwickler.* Bonn: Rheinwerk Verlag. Seite 22-23

[KüTh21b] Künneth, T. (2021). *Android 11 Das Praxisbuch für App-Entwickler.* Bonn: Rheinwerk Verlag. Seite 33

[LoGL22] Vries, J. d. *Learn OpenGL*. Abgerufen am 06. 11 2022 von https://learnopengl.com/Getting-started/Coordinate-Systems

[ProGL12] Smithwick, M., & Verma, M. (2012). *Pro OpenGL ES for Android.* Berlin: Springer Verlag. Seite 2.

[RoFr12] Frisch, R. (18. 01 2012). *Rolandsroids Allerlei*. Abgerufen am 10. 11 2022 von https://rolandroid.wordpress.com/2012/01/18/zauberwuerfel-notation/

[StRot22] Rothneiger, S. (kein Datum). *SteMaRo-Magic.de*. Abgerufen am 11. 11 2022 von https://www.stemaro-magic.de/Infoseite/Die-interessante-Geschichte-des-legendaeren-Zauberwuerfels.html

[ZaAn22] Pochmann, S. (kein Datum). *Zauberwürfel-Anleitung*. Abgerufen am 18. 10 2022 von https://ppedv.de/Images/Zauberw%C3%BCrfel_Anleitung_CubeSolution.pdf

[ZaWi22] Wölk, M. (kein Datum). *Zauberwürfel*. Abgerufen am 8. 12 2022 von https://dewiki.de/Lexikon/Zauberw%C3%BCrfel

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorgelegte Abschlussarbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Köln, den 22.12.2022

Unterschrift

(Vorname, Nachname)

TH Köln
Gustav-Heinemann-Ufer 54
50968 Köln
www.th-koeln.de

Technology
Arts Sciences
TH Köln