4d visualisierungs Engine mit einer 3d Benutzeroberfläche als Hilfestellung beim Lernen und Experimentieren

Maturitätsarbeit

https://github.com/NandorKovacs/4dvisualizer

MNG Rämibühl

January 9, 2023

Abstract

Ziel dieser Arbeit ist, vierdimensionale Würfel im dreidimensionalen Raum darzustellen. Die Projektionsmethode, um den vierdimensionalen Raum in einem dreidimensionalen Raum darzustellen, kann man sich folgenderweise vorstellen: Nimmt man einen Kuchen und schneidet diesen in die Hälfte, wird die Schnittfläche eine zweidimensionale Fläche. Anders ausgedrückt, nehme ich einen Körper im dreidimensionalen Raum, welche ich mit einem zweidimensionalen Raum (also einer Ebene) schneide. In dieser Arbeit nehme ich eine 'Hyperebene', also einen dreidimensionalen Raum, und schneide damit den vierdimensionalen Raum. Der Schnittköper wird mit Hilfe des Koordinatensystems des Schnittraumes auf dem Bildschirm visualisiert.

Inhaltsverzeichnis

1 Vorwort

> *... diese Nullen tatsächlich rätselhafte, unverständliche Dinger waren. ... So eine Null bestand im Grunde bloß aus zwei Kupferscheiben von der Größe einer Untertasse und einer Dicke von fünf Millimetern — der Abstand zwischen den Scheiben betrug ungefähr vierzig Zentimeter. Außer diesem Abstand aber gab es nichts zwischen ihnen, absolut nichts, nur Leere. Man konnte die Hand in den Zwischenraum stecken, sogar den Kopf, wenn man übergeschnappt war — nichts als Luft. Und doch musste was zwischen den Scheiben existieren, irgendeine geheimnisvolle Kraft, wenn ich recht verstehe, denn noch niemandem war es bisher gelungen, sie zusammenzudrücken oder auseinanderzuziehen." (Arkadi und Boris Strugazki: Picknick am Wegesrand)*

Es ist spannend sich auszumalen, wie das Leben in einer zweidimensionalen Welt wäre. Man kann sich das vorstellen, wie kleine Lebewesen auf der Tischoberfläche, die nur ihre eigene Ebene wahrnehmen können. Sie sehen nichts ausserhalb dieser Ebene, und können sich auch nur in ihrem eigenen, unendlichen zweidimensionalen Raum bewegen. Bemerken sie womöglich einige Anhaltspunkte, dass rund um sie herum ein grösserer Raum existiert?

Eines Tages lässt jemand einen Bleistift neben dem Tisch fallen, der durch die Ebene dieser Lebewesen fällt. Eines der Tierchen wird Zeuge dieses Vorfalls: Es sieht zuerst einen Kreis der immer grösser wird, zuerst schwarz, dann holzfarben. Dann sieht es ein rotes Sechseck, und etwas später verschwindet dieses gänzlich. Das Phänomen ist unerklärbar, aber kurz. Vergebens erzählt es seinen Kameraden davon, doch niemand glaubt ihm.

Einige Tage später legt jemand eine Gabel auf die Tischkante, so dass dessen Spitzen nicht auf dem Tisch liegen. Die Spitzen krümmen sich aber nach unten, und gehen durch die Ebene wo unsere Lebewesen leben. Diesesmal haben sie alle genügend Zeit um das Phänomen zu beobachten. Wenn sie sich alle anstrengen, können sie die Gabel sogar etwas bewegen. Alle beobachten gespannt wie die vier kleinen metallenen Objekte, die sichtbar von nichts zusammengehalten werden, trotzdem immer gleichweit voneinander bleiben, egal wie oder welches der Vier sie bewegen.

Es hat mich schon immer beschäftigt, mir Etwas vorstellen zu können, das eigentlich recht unvorstellbar ist. Für meine Maturarbeit habe ich den kleins-

ten unvorstellbaren Raum gewählt, den vierdimensionalen Raum, und darin einen gut bekannten, einfachen Körper, den Würfel. Nur eine Kugel wäre einfacher gewesen, aber genau wie alle zweidimensionale Schnitte einer dreidimensionalen Kugel Kreisflächen sind, sind alle dreidimensionale Schnitte von vierdimensionalen Kugeln normale Kugeln. Das macht die Aufgabe nicht so interessant wie ein Würfel.

An dieser Stelle möchte ich mich bei meinem Vater bedanken, der immer für mich da war, und mir geholfen hat wenn ich beim Programmieren stecken blieb. Er hat mir ausserdem geholfen die Technologien und Methoden kennen zu lernen, und die richtigen auszuwählen. Ich bedanke mich bei meiner Mutter, die mich stets ermutigt hat; und bei meinem Bruder, Lorant, der oft schlafen musste während ich noch im Zimmer am arbeiten war. Mein Bruder Albert hat mir gezeigt wie ich die dreidimensionale Darstellungen erstellen kann. Ohne ihn wären die Bilder nicht so übersichtlich und schön. Ich möchte mich ausserdem bei meiner Begleitperson, Herr Pietro Gilardi, bedanken, für seine Unterstützung und dass er die Aufgabe angenommen hat meine Arbeit zu betreuen, nachdem meine vorherige Begleitperson die Schule verlassen hat und mich nicht weiter betreuen konnte.

Ich möchte mich ausserdem bei meinen Freunden und Bekannten bedanken die mir beim Korrekturlesen geholfen haben: Dimitrij, Kyril und Nina. Sie haben mir etliche grammatische Fehler korrigiert, und mich auf schwer verstehbare Abschnitte hingewiesen.

2 Einleitung

2.1 Zielsetzung

Die zweidimensionale Schnitte von dreidimensionalen Körpern kennen wir gut. Schauen wir beispielsweise den Einheitswürfel an. Viele seiner Schnitte sind quadratisch oder rechteckig. Wir finden aber auch schnell dreieckige Schnitte wenn wir eine der Ecken abschneiden. Etwas schwieriger zu sehen, ist, dass wir auch fünf, und sechseckige Schnitte bekommen können. Wie sehen wohl verschiedene vierdimensionale Schnitte aus? Diese Arbeit beschäftigt sich genau damit: mithilfe einer Computersimulation werden solche Schnitte dargestellt.

Ein wichtiges Ziel ist, dass man die dreidimensionalen Resultate der Schnit-

te gut von allen Seiten begutachten kann. Dass heisst, dass sie drehbar und bewegbar sind, sowie verständlich gefärbt dargestellt werden.

3 Hauptteil

3.1 Theorie

3.1.1 Was ist ein vierdimensionaler Würfel, und wie kann man sich so einen Würfel vorstellen?

Um zu definieren, wie ein vierdimensionaler Würfel aufgebaut ist, werfe ich einen Blick auf den Unterschied zwischen einem dreidimensionalen Würfel und einem zweidimensionalen Würfel, wie auch einem zweidimensionalen Würfel und einem eindimensionalen Würfel. Ein- und zweidimensionale Würfel gibt es bekanntlich nicht. Was equivalent ist mit einem zweidimensionalen Würfel, ist ein Quadrat.

Wie mache ich einen Würfel aus einem Quadrat? Anfänglich liegt das Quadrat einfach auf der Ebene $F(x, y) = 0$. Um daraus einen Würfel zu machen, dupliziere ich das Quadrat, und bewege es entlang der neuen Koordinatenachse, bis die zwei Quadrate gleich weit voneinander entfernt sind, wie auch die Seitenlänge der Quadrate ist. Nun verbinde ich die Ecken der Quadrate, die übereinander liegen, sie werden zu einem Würfel.

Genau gleich können wir aus einem eindimensionalen Würfel einen Quadrat machen. Jedes eindimensionale Objekt ist eine Strecke. So auch ein Würfel. Nehme ich jetzt eine Strecke, setze sie in einen zweidimensionalen Raum auf die Gerade $F(x) = 0$, dupliziere sie, und bewege die zweite Strecke auf der neuen y Koordinate bis die zwei Strecken so weit voneinander sind wie eine Strecke lang ist, und verbinde dann die Ecken, erhalte ich ein Quadrat.

So mache ich den Sprung auch eine Dimension höher. Ich nehme einen Würfel mit Seitenlänge 1, auf der Hyperebene $F(x, y, z) = 0$. Einen zweiten Würfel verschiebe ich entlang der w-Achse um 1. Verbindet man nun die Ecken der zwei Würfel, erhaltet man einen vierdimensionalen Würfel.

3.1.2 Die Definition eines vierdimensionalen Würfels

Um einen vierdimensionalen Würfel zu definieren, müssen wir zuerst den vierdimensionalen Raum definieren. Dies definieren wir ähnlich zu den zwei-

und drei dimensionalen Räumen:

$$\mathbb{R}^2 = \{(x1,x2) \mid x1,x2 \in \mathbb{R}\}$$
$$\mathbb{R}^3 = \{(x1,x2,x3) \mid x1,x2,x3 \in \mathbb{R}\}$$
$$\mathbb{R}^4 = \{(x1,x2,x3,x4) \mid x1,x2,x3,x4 \in \mathbb{R}\}$$

Es ist einfach zu sehen, dass es auf den Fall einer beliebigen $R^n, n \in \mathbb{N}$ ausehnbar ist.

Eine Hyperebene ist folgendes: In einem n-dimensionalen Raum beschreibt sie alle Punkte, die durch einen Stützvektor und n-1 Richtungsvektoren beschrieben werden können. Somit beschreibt sie einen n-1 dimensionalen Raum innerhalb eines n-dimensionalen Raumes.

Im folgenden beschäftigen wir uns mit vierdimensionalen Räumen. Die Koordinaten schreiben wir als (x, y, z, w). Unter einer Hyperebene verstehen wir die dreidimensionalen Hyperebenen des vierdimensionalen Raumes.

Definieren wir nun Würfel in verschiedenen Dimensionen:

- Der eindimensionale kanonische Würfel: $\{x \in \mathbb{R} \mid -1 \leq x \leq 1\}$
- Der zweidimensionale kanonische Würfel: $\{x, y \in \mathbb{R} \mid -1 \leq x, y \leq 1\}$
- Der dreidimensionale kanonische Würfel: $\{x, y, z \in \mathbb{R} \mid -1 \leq x, y, z \leq 1\}$
- der vierdimensionale kanonische Würfel: $\{x, y, z, w \in \mathbb{R} \mid -1 \leq x, y, z, w \leq 1\}$

Wir merken, dass im Fall des dreidimensionalen Würfels die Ecken genau die Punkte sind, bei denen alle drei Koordinaten den maximal oder minimal Wert haben. Das heisst, dass die Koordinaten -1 oder 1 sind. Solche Tripel hat es $2^3 = 8$, genau so viel, wie man auf einem Würfel sehen kann. Die Kanten bekommen wir folgenderweise: Aus den drei Koordinaten fixieren wir zwei beliebige auf einen der Grenzwerte (-1 oder 1), die dritte gleitet über den Intervall $(-1, 1)$. Zwei Koordinaten aus drei auszuwählen, und dann für die zwei Koordinaten alle grenzwert Kombinationen durchgehen ergibt $3 \times 2 \times 2 = 12$ Möglichkeiten. Genau so viele Kanten hat ein Würfel. Bei den Flächen gehen wir ähnlich vor, fixieren aber nur eine der Koordinaten auf -1 oder 1. Diesmal hat es $3 \times 2 = 6$ Möglichkeiten, so viele Flächen hat ein Würfel.

Was ist der Fall bei einem vierdimensionalen Würfel?

Eine Ecke nennen wir wieder eine Ecke, sofern alle Koordinaten entweder -1 oder 1 sind. Daraus bekommen wir $2 \times 2 \times 2 \times 2 = 16$ Ecken. Kanten, beziehungsweise eindimensionale Seiten bekommen wir mit drei fixierten Koordinaten. Dies ergibt $4 \times 2 \times 2 \times 2 = 32$ Kanten. Die Anzahl zweidimensionaler Seiten ist $\binom{4}{2} \times 2 \times 2 = 24$. Diese sind weiterhin Quadrate, genau wie in der eins tieferen Dimension. Vierdimensionale Würfel haben auch dreidimensionale Flächen. Diese bekommen wir, indem wir nur eine der Koordinaten bei -1 oder 1 fixieren. Fixieren wir beispielsweise die letzte Koordinate, bekommen wir den dreidimensionalen kanonischen Würfel, aus einer höheren Dimension betrachtet. Die dreidimensionalen Seiten sind alles reguläre Würfel. Davon hat es $4 \times 2 = 8$.

3.1.3 Drehungen in vier Dimensionen

Hier schauen wir uns Drehungen um den Ursprung an.

3.1.3.1 Die Grundrotationen In zwei Dimensionen kann man nur um Punkte drehen. So kann eine Drehung um den Ursprung mit einer einzigen Zahl; dem Winkel, angegeben werden.

In drei Dimensionen wird es schon spannender. Wir haben jetzt drei verschiedene Drehungen. Wir können um die X-, Y- oder Z-Achse drehen. Alle andere Drehungen können durch diese drei Drehungen zusammengesetzt werden (Satz vom Fussball, `https://de.wikipedia.org/wiki/Satz_vom_Fu%C3%9Fball`).

In vier Dimensionen drehen sich die Grundrotationen um keine Achsen, sondern rotieren um eine zweidimensionale Ebene. Um zu verstehen wieso das der Fall ist, müssen wir aufhören zu denken, dass Drehungen um etwas drehen, sondern einsehen das Drehungen immer eine zweidimensionale Ebene drehen. Deswegen hat es im 2d Raum nur einen Punkt (0d Objekt) der nicht bewegt wird bei einer Rotation. Im 3d Raum haben wir eine Fixachse (1d Objekt), die orhtogonale Gerade auf die drehende Fläche. So ist es intuitiver, dass es im 4d Raum bei einer Rotation eine Fixfläche (2d Objekt) existiert.

Aus diesem Grund existieren sechs Grundrotationen. Es hat vier Achsen, und je zwei drehen sich. Es gibt also $\binom{4}{2} = 6$ Kombinationen. Die sechs

Grunddrehungen sind die Drehungen um die XY, XZ, XW, YZ, YW und ZW Flächen.

3.1.3.2 Die Drehmatrizen Die Drehmatrizen in vier Dimensionen sind sehr ähnlich zu den dreidimensionalen Drehmatrizen.

Das sind alle Drehmatrizen mit dem Drehwinkel α:

$$R(XY,\alpha) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \cos\alpha & -\sin\alpha \\ 0 & 0 & \sin\alpha & \cos\alpha \end{pmatrix}$$

$$R(XZ,\alpha) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & 0 & -\sin\alpha \\ 0 & 0 & 1 & 0 \\ 0 & \sin\alpha & 0 & \cos\alpha \end{pmatrix}$$

$$R(XW,\alpha) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R(YZ,\alpha) = \begin{pmatrix} \cos\alpha & 0 & 0 & -\sin\alpha \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \sin\alpha & 0 & 0 & \cos\alpha \end{pmatrix}$$

$$R(YW,\alpha) = \begin{pmatrix} \cos\alpha & 0 & -\sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ \sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R(ZW,\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Orhtogonale Matrizen mit einer Determinante von 1 sind Drehmatrizen. Alle sechs Matrizen erfüllen diese Kriterien.

3.1.3.3 Drei Drehungen, um einen Einheitsvektor an einen beliebigen Ort auf der Einheitskugel zu drehen

$$R(ZW,\alpha)\cdot\begin{pmatrix}1\\0\\0\\0\end{pmatrix}=\begin{pmatrix}\cos\alpha\\\sin\alpha\\0\\0\end{pmatrix}$$

$$R(YW,\beta)\cdot R(ZW,\alpha)\cdot\begin{pmatrix}1\\0\\0\\0\end{pmatrix}=\begin{pmatrix}\cos\alpha\cdot\cos\beta\\\sin\alpha\\\cos\alpha\cdot\sin\beta\\0\end{pmatrix}$$

$$R(YZ,\gamma)\cdot R(YW,\beta)\cdot R(ZW,\alpha)\cdot\begin{pmatrix}1\\0\\0\\0\end{pmatrix}=\begin{pmatrix}\cos\alpha\cdot\cos\beta\cdot\cos\gamma\\\sin\alpha\\\cos\alpha\cdot\sin\beta\\-\cos\alpha\cdot\cos\beta\cdot\sin\gamma\end{pmatrix}$$

Jetzt drehe ich $\begin{pmatrix}1\\0\\0\\0\end{pmatrix}$ so, dass es $\begin{pmatrix}a\\b\\c\\d\end{pmatrix}$ wird für beliebige a, b, c und d, so dass $\sqrt{a^2+b^2+c^2+d^2}=1$ erfüllt ist. Das heisst, (a,b,c,d) liegt auf dem Einheitskreis.

Wenn entweder b, c, oder d 1 ergibt, sind alle anderen 0, da wir angenommen haben dass $\begin{pmatrix}a\\b\\c\\d\end{pmatrix}$ die Länge 1 hat. Dann braucht es nur eine einzige Drehung. Ansonsten:

$$\begin{aligned}\alpha&=\arcsin(b)\\\beta&=\arcsin(\frac{c}{\cos\alpha})\\\gamma&=\arccos(\frac{a}{\cos\alpha\cdot\cos\beta})\end{aligned}$$

Das dies lösbar ist im Normalfall, zeigen folgende Ungleichungen:

$$|b| \leq 1$$
$$|c| = \sqrt{1 - a^2 - b^2 - c^2} \leq \sqrt{1 - b^2} = \sqrt{1 - \sin^2 \alpha} = |\cos \alpha|$$
$$|a| \leq \sqrt{1 - c^2} \leq \sqrt{\cos^2 \alpha - c^2} = |\cos \alpha| \cdot \sqrt{1 - \frac{c^2}{\cos^2 \alpha}} = |\cos \alpha| \cdot \sqrt{1 - \sin^2 \beta} = |\cos \alpha| \cdot |\cos \beta|$$

Das zeigt das ein Körper mit einer arbiträren Drehung durch drei Drehungen beschrieben werden kann, und nicht alle sechs Grunddrehungen verwendet werden müssen dafür.

3.1.4 Schnittkörper eines 4d Würfels mit einer 3d Hyperebene

Der Schnittkörper eines 4d Würfels und einer 3d Hyperebene ist ein maximal dreidimensionales Volumen. Es ist weniger wie dreidimensional in entartetten Fällen, beispielsweise wenn die Hyperebene eine Ecke, Kante, oder zweidimensionale Seite nur knapp berührt.

Um einen Weg zu finden diesen Schnittkörper zu berechnen, schauen wir uns zuerst das Problem im dreidimensionalen Raum an. Wir sehen, dass wir im Normalfall die Schnittfläche eines 3d Würfels und einer 2d Ebene bekommen können, indem wir die Schnittpunkte der Ebene mit den Kanten des Würfels ausrechnen, und die konvexe Hülle dieser Schnittpunkte nehmen. (Figure **??**).

Im Normalfall hat es maximal einen Schnittpunkt mit jeder beliebigen Kante und der Schnittebene.

Es gibt aber einige entartende Fälle. Beispielsweise wenn einige Kanten des Würfels komplett auf der Schnittebene liegen.

Für diese Fälle bleibt der Algorithmus korrekt, denn wir nehmen einfach beide Enden der gegebenen Kante und betrachten diese Punkte als Schnittpunkte mit der Schnittebene und der Kante. (Es ist einfach zu sehen, dass alle Punkte zwischen diesen zwei Punkten in der konvexen Hülle sind von jedem Set von Punkten welche beide Endpunkte beinhalten).

Wir beweisen, dass der gleiche Algorithmus anwendbar ist, wenn wir einen 4d Würfel mit einer 3d Hyperebene schneiden: im Normalfall wird der Schnittköper die konvexe Hülle der Schnittpunkte der Kanten des Würfels und der Hyperebene.

Lemma 1. *Im Normalfall wird das Resultat eines Schnittes von einer zweidimensionalen Kante und einer dreidimensionalen Hyperebene ein einzelner Punkt.*

Beweis. Im 4d Raum existiert für jede 3d Hyperebene ein Vector $\overrightarrow{v}$, der orthogonal zur Ebene ist. Jede Gerade, die nicht orthogonal zu v ist, kann nur einen Schnittpunkt mit der Ebene haben. Wenn es beispielsweise 2 hätte, x_1 und x_2, dann wäre $\overrightarrow{x_2 - x_1}$ auf der einen Seite orthogonal zu $\overrightarrow{v}$ weil die Hyperebene orthogonal zu $\overrightarrow{v}$ ist, auf der anderen Seite aber nicht orthogonal, da wir angenommen haben, dass die Linie nicht orthogonal zu $\overrightarrow{v}$ ist. □

Lemma 2. *Im Normalfall ist der Schnitt von einer 3d Seite eines 4d Würfels und einer 3d Hyperebene zweidimensional.*

Beweis. Eine 3d Seite eines 4d Würfels ist ein 3d Würfel. Gemäss Lemma 1 sieht der Schnitt der Kanten wie Abbildung **??** aus. Der komplette Schnitt ist die konvexe Hülle dieser Punkte. Es ist eindeutig, dass sie einen mindestens zweidimensionalen Raum spannen. Im Normalfall müssen sie auf der selben 2d Ebene liegen. Um das zu beweisen, nehmen wir an, dass sie nicht auf der selben 2d Ebene liegen. Das würde heissen, dass wir einen Punkt auf einer Kante finden können, den wir via lineare Kombination von den anderen drei Punkten erreichen können. Was bedeutet, dass wir 2 Punkte auf der gleichen Kante im dreidimensionalen Raum haben, was gemäss Lemma 1 kein Normalfall ist. □

Das heisst, dass im Normalfall die äussere Oberfläche des Schnittes von einem 4d Würfel und einer 3d Hyperebene eine Union von 2d Flächen sein wird, was wir bekommen indem wir die 3d Seiten mit der 3d Hypereben schneiden, und das ist was wir beweisen wollten.

3.1.5 Rechenalgorithmus

Um einen praktischen Algorithmus implementieren zu können, müssen wir eine Repräsentation von einem 4d Würfel im 4d Raum finden, die wir verstehen und nachvollziehen können. Zu beachten ist, dass bereits die Nummerierung der 2d Seiten nicht trivial ist.

Um mit der Komplexität umzugehen, implementieren wir folgenden Algorithmus, um den Schnitt eines 4d Würfels und einer 3d Hyperbene zu bestimmen:

1. Wir wenden eine isometrische Transformation und eine einheitliche Skalierung an, dass den Würfel in den kanonischen Würfel transformiert.
2. Wir transformieren die 3d Ebene mit der gleichen Transformation.
3. Wir schneiden den kanonischen Würfel mit der transformierten Ebene.
4. Wir wenden die Inverse der Transformation am resultierenden Körper an.

Diese Vorgehensweise erlaubt uns eine einfache Implementation des Schneidealgorithmus.

3.2 Entwicklung

3.2.1 Tools und Frameworks

- Programming language: C++
- Cross-platform windowing library with OpenGL support: GLFW.
- Drawing: OpenGL
- OpenGL loader: GLEW
- Build system: CMake

3.2.1.1 C++ C++ ist meine präferierte Computersprache. Der gesamte Programmcode den ich geschrieben habe, ist in dieser Sprache, mit Ausnahme der Shaderprogramme, die in GLSL geschrieben werden müssen.

3.2.1.2 GLFW GLFW (Graphics Library Framework) ist eine Library die Anwendungsfenster und Eingabe über Maus und Tastatur verwaltet. Mithilfe von GLFW ist es möglich für uns, ein Fenster mit dem Programm zu öffnen, um eine Oberfläche zum Zeichnen zu erhalten.

3.2.1.3 OpenGL OpenGL ist im Prinzip das was uns erlaubt mit der Graphikkarte zu komunizieren. Wir rufen OpenGL Funktionen auf, und schicken damit unsere Daten an die Grafikkarte. Wir schicken ausserdem unser Schattierungsprogramme an die Grafikkarte. Die Grafikkarte benutzt diese Programme dann in der richtigen Reihenfolge, um zu berechnen welche Pixel auf dem Bildschirm welche Farbe haben sollen.

3.2.1.4 GLEW

OpenGL hat eine API (Das heisst Funktionen, die man benutzen kann im eigenen Code) die Öffentlich ist. Aber je nach Graphikkarten Hersteller und Operationssystem sind diese Funktionen anderst implementiert. Ausserdem gibt es viele Versionen von OpenGL, und manchmal möchte man, dass ein Programm auf mehreren Versionen funktioniert. Im Normalfall werden die Funktionen einer API während der Compilation in das eigene Programm compiliert. Mit OpenGL aber wird nur eine sehr kleine Anzahl der Funktionen direkt hineincompiliert, viele der Funktionen müssen abgefragt werden. Somit kann man beispielsweise nach einer Funktion 'Zeichnen' fragen, und falls diese auf mehrere OpenGL Versionen existiert, funktioniert das Programm auf all diesen Versionen. Damit man dieses Abfragen nicht selbst machen muss, exisitert GLEW. GLEW erledigt dies ganz automatisch, man muss nur einige Parameter angeben und dann funktioniert es von alleine.

3.2.2 Die Funktionsweise und das Zusammenspiel der Libraries, der Graphikkarte und des Operationssystems

Die Libraries, die ich benutzt habe, wurden oben beschrieben. Hier erläutere ich, wie sie genau funktionieren. Mit einem Programm ein Applikationsfenster zu öffnen und zu benutzen ist ganz einfach mit Libraries, die im Hintergrund alles komplizierte automatisch lösen. Dafür haben diese immer sehr viele Einschränkungen. Will man wirklich frei sein, zu tun und lassen was man will, muss man ganz Vieles selbst lernen und umsetzen.

Aus diesem Grund habe ich mich dazu entschieden zu lernen, wie man mit OpenGL umgeht. OpenGL ist eines der meistverbreiteteten API's für zwei- und dreidimensionale Vektorgraphiken, daher war mir sicher, dass es Sinn macht dies zu lernen.

Mit OpenGL lädt man Daten auf die Grafikkarte. GLEW dient dabei wie ein Übersetzer; es macht eigentich nichts anderes, als unsere Funktionsaufrufe an die richtige Funktionen zu binden. In unserem Fall lädt man eine Menge von Punkten in einem 3d Raum (repräsentiert durch Vektoren) sowie die Schattierungsprogramme auf die Grafikkarte. Alles andere, also die Berechnungen und die Verwaltung des Bildschirms, geschieht 'von alleine'. Es wird von der Grafikkarte übernommen, und ist nicht unsere Verantwortung. Die Schattierungsprogramme, sofern wir sie an den richtigen Ort geladen haben, verändern und interpretieren unsere Daten, sodass aus Punkten, Farben und Vektoren eine Menge Pixel entstehen.

OpenGL selber ist aber nur ein Werkzeug, um mit der Grafikkarte zu komunizieren. Um ein Fenster zu öffnen und Tastendrücke zu erkennen und verwenden im Programm, brauche ich auch eine Schnittstelle mit dem Operationssystem. Dafür verwende ich GLFW. GLFW hilft bei der Kommunikation mit dem Operationssystem. Mithilfe von GLFW kann ich das Applikationsfenster erstellen und verwalten, Tastendrücke wahrnehmen und die Mausbewegung benutzen.

Was noch nicht ganz klar ist, ist wie die Grafikkarte weiss, wo genau die berechneten Pixel hingehören. Dies wird gelöst, indem wir durch GLFW einen sogenannten Kontext beim Erstellen des Fensters erhalten. Wenn wir die Daten und Programme an die Grafikkarte schicken, schicken wir zudem die Information welchen Kontext wir benutzen.

Je nach Operationssystem bekommt GLFW so einen Kontext auf unterschiedlicher Weise für uns. Auf Windows werden die Fenster vom Operationssystem verwaltet. Das heisst, das GLFW mithilfe des Operationssystems ein Fenster erstellt, was widerum auf der Grafikkarte einen Kontext erstellt, und an GLFW die Nummer des Kontextes auf der Karte mitteilt. Auf Linux, zumindest auf der Verteilung die ich benutze, gibt es einen sogenannten Compositor. Dieser sorgt dafür, dass alle Fenster richtig funktionieren, macht das man bei transparenten Fenster das sieht, was unter dem Fenster ist und verwaltet allgemeint alles, was Fenster angeht. Somit spricht GLFW nicht nur mit dem Operationsystem, sondern auch mit dem Compositor. Der Compositor erstellt mit OpenGL einen Kontext und gibt dann auch die Nummer des Kontextes an GLFW zurück.

3.2.3 Meilensteine

3.2.3.1 Meilenstein 1: Ein 3d Würfel dargestellt, mit Navigation

Key results

- Der 3d Würfel ist gerendert.
- Das Bewegen des Kameras ist möglich sowohl mit der Tastatur, als auch mit der Maus.

Value Wir haben eine Arbeitsumgebung aufgebaut, und haben ein Verständnis für die Frameworks und Libraries erarbeitet.

Prozess Dieser Meilenstein sieht nicht sehr schwierig aus, war aber einer der zeitaufwändigsten Teile des Projekts. Ich habe zuerst aus einem Buch und dem Internet gelernt, wie OpenGL funktioniert, und was man für Libraries braucht und musste zuerst lernen, wie diese zusammenspielen. Erst dann konnte ich einen einzelnen blauen Punkt auf dem Bildschirm erscheinen lassen. Dann konnte ich anfangen einen Würfel darzustellen. Hier war die grösste Schwiereigkeit den Code richtig zu strukturieren. Ich habe mir hier überlegt, was für Datenstrukturen ich brauchen werde, und was ich in Klassen abstrahieren möchte.

3.2.3.2 Meilenstein 2: Der Schnitt eines zentrierten, kanonischen 4d Würfels mit einer verstellbaren und beweglichen Hyperebene wird dargestellt.

Key results Ein C++ Programm, dass:

- den Schnitt eines zentrierten, kanonischen 4d Wüfel mit einer arbiträren Hyperebene berechnet.
- diesen Schnitt als dreidimensionales Objekt rendert.
- die Parameter der Hyperebene interaktiv ändern kann.
- den gerenderten Schnitt bei Änderungen der Hyperebene in Echtzeit anpasst.
- weiterhin im dreidimensionalen Raum navigieren kann wie im vorherigen Meilenstein.

Value Es zeigt sich, dass die Theorie für die Berechnung des Schnittes korrekt ist. Wir erhalten einen ersten Eindruck vom 4d Würfel.

Prozess Hier habe ich zuerst lange nichts programmiert, sondern habe mir mit Papier und Stift überlegt, wie das funktionieren soll. Ich habe beschlossen, dass ich die Hyperebene mit zwei vierdimensionalen Vektoren repräsentiere. Eins ist ein Normalvektor, und der Andere beschreibt die Verschiebung der Hyperebene vom Ursprung. Hier habe ich mir auch ausgedacht, wie der Algorithmus funktionieren soll, sodass ich einen 3d Schnittkörper erhalte. Ich habe schnell einen Algorithmus erfunden, um die Eckpunkte des

Schnittkörpers zu finden, aber das hat noch lange nicht alles gelöst. Um einen Körper mit OpenGL darzustellen, muss man diesen in Dreiecke zersetzen und anschliessend an OpenGL schicken. Dafür musste ich wissen welche der Ecken benachbart sind. Dies war keine triviale Aufgabe. Ich dachte zuerst daran einen klassischen Algorithmus für dreidimensionale konvexe Hüllen zu implementieren. Dies wäre aber recht komplex und zeitaufwändig gewesen. Darum habe ich mir lieber eine einfacher implementierbare Methode überlegt, die beim Schneiden des Würfels die Information ausnutzt, auf welcher 2d Seite sich ein Schnittpunkt befindet, um zu bestimmen, welche Punkte benachbart sein können. Der genaue Algorithmus, wie die Dreiecke zusammengestellt werden, folg später im Teil 3.2.4.2.

3.2.3.3 Meilenstein 3: Sichtbarkeitseffekte

Key Results Objektschattierung und Beleuchtung wird implementiert.

Value Das Programm erlaubt uns mit Parametern zu experimentieren, und verschiedene Formen und Gestalten von Schnitten des 4d Würfels kennen zu lernen.

Mit den Lichteffekten und der Schattierung wird das gerenderte Bild einfacher zu sehen und zu verstehen für den Benutzer. Jetzt erst sind die Kanten und Ecken sichtbar. Ohne Schattierung konnte man nur die Umrisse sehen, und die Kanten und Ecken durch Bewegung erahnen.

Prozess Damit man überhaupt einen Punkt erscheinen lassen kann, benötigt man sogenannte Schattierungsprogramme. OpenGL funktioniert so, dass jeder Punkt eine sogenannte Rendering-Pipeline durchläuft. Vereinfacht sieht das so aus, dass zuerst jeder Punkt durch den Vertexshader geht, dann alle Primitive (in unserem Fall Dreiecke, aber es könnten auch Linien sein) durch den Tesselationshader und dem Geometryshader gehen. Dann geschieht die Rasterisation, das heisst, dass jetzt die einzelnen Pixel auf dem Bildschirm berechnet werden. Schlussendlich geht jeder Pixel noch durch den sogenannten Fragment Shader. Uns interessiert nur der Vertexshader und der Fragmentshader. Für dieses Projekt ist es nicht nötig die anderen zu benutzen.

Wie man solche Shattierungsprogramme (Shader) schreibt, habe ich schon im Verlauf des Arbeitsprozesses gelernt. Doch erst in diesem Schritt habe ich

genau lernen und verstehen müssen, wie man sie am besten benutzt und was es für Tricks gibt. Bis zu diesem Moment habe ich nur primitive Shader gehabt (Beispielsweise habe ich nur jedem Punkt eine Farbe gegeben, und darauf vertraut, dass während dem Rasterizationsverfahren durch Interpolation auf dem Würfel ein Farbverlauf entstehen wird). Jetzt habe ich aus dem OpenGL Buch gelernt, wie man Lichteffekte machen kann, indem man Informationen wie den Ursprung des Lichtes, Materialeigenschaften, Lichtintensität und Farbe und weiteres an die Shader schickent und mit Formeln so die richtige Schattierung für jeden Pixel ausrechnen kann.

3.2.3.4 Meilenstein 4: Der Schnitt eines arbiträren 4d Würfels

Key results Der Würfel ist nicht mehr zwingend zentriert beim Ursprung. Er kann sich jetzt an einem arbiträren Ort im 4d Raum, mit einer arbiträren Drehung befinden. Dies ist hier aber vorerst nur im Code beeinflussbar, man kann den Würfel nicht einstellen oder bewegen.

Value Wir haben ein Skelett vom Kern des Programms.

Prozess Am wichtigsten war einen Weg zu finden, die Transformation von einem arbiträren Würfel in den kanonischen Würfel zu repräsentieren. Da die grösste Matrix in der standard OpenGL mathematik Library eine 4×4 Matrix ist, habe ich mich entschieden nicht die ganze Transformation mit einem Matrix zu beschreiben, sondern eine Klasse zu erstellen die zuerst eine Drehung, dann eine Skalierung, und als letztes eine Translation durchführt.

Das wir die Transformation mit diesen drei Komponenten beschreiben, wird etwas später einen weiteren grossen Vorteil haben: der Benutzer wird sehr einfach so eine Transformation angeben können. Man kann alle drei Komponente unabhängig voneinander beliebig einstellen, ohne das Probleme entstehen können.

Angenommen wir haben eine Transformation T, welche den kanonischen Würfel in den arbiträren Würfel verschiebt. Es hat die folgende drei Komponente: $T = T_{\text{translation}} \circ T_{\text{skalierung}} \circ T_{\text{rotation}}$. Wir wollen jetzt eine Transformierte Hyperebene haben, so dass das Schneiden des kanonischen Würfels mit dieser Hyperebene, und dann die zurück Transformierung des Schnittes genau den gesuchten Schnitt des arbiträren Würfels ergibt.

Dafür müssen wir einfach nur den Ursprung der Hyperebene mit T^{-1} transformieren. Das machen wir, indem wir die Inverse der Komponenten in der umgekehrten Reihenfolge anwenden: $T^{-1} = T^{-1}_{\text{rotation}} \circ T^{-1}_{\text{skalierung}} \circ T^{-1}_{\text{translation}}$. An den Einheitsvektoren der Hyperebene wenden wir nur die Inverse der Rotation an. Um zu verstehen wieso das funktioniert, muss man einfach überlegen was T macht: es transformiert den kanonischen Würfel in den gewählten Würfel. Logischerweise macht T^{-1} also genau das Umgekehrte. Und wenn wir diese inverse Transformation auch an die Hyperebene anwenden, 'geht sie mit dem Würfel mit'. In anderen Worten, T^{-1} ist eine Operation, dass ein Koordinatensystem erstellt, in dem der gewählte Würfel der kanonische Würfel ist.

Dies zu programmieren war recht unkompliziert. Ich habe ein separates Modul erstellt, das die vierdimensionale Transformationen für mich macht. Dann habe ich kleine Änderungen am bestehenden Code vorgenommen: im Renderer rufe ich den Intersektror mit der Transformierten Hyperebene auf, und Transformiere die Resultate auch zurück.

Um die resultierenden 3d Vektore richtig zu Transformieren, müssen wir sie zuerst in 4d Vektore zurückkonvertieren mithilfe der Hyperebene. Dann können wir T an ihnen anwenden, und sie dann wieder in 3d Vektore zurückwandeln.

Es muss immer gut aufgepasst werden, ob ein Vektor ein Ortsvektor ist oder nicht. Für Vektore die keine Ortsvektore sind, müssen wir den Ursprung der Hyperebene subtrahieren, und nur die Rotation anwenden, ohne den anderen zwei Komponenten der Transformation.

3.2.3.5 Meilenstein 5: Ein Format, dass den Aufbau einer Welt beschreibt

Key results

- Ein Fileformat wird definiert, dass 4d Objekte im Raum beschreibt.
- Ein Parser ist implementiert.

Value Wir sind bereit das Endprodukt zusammenzustellen.

Prozess Um einfach mit dem Programm spielen und experimentieren zu können brauchte ich einen Konfigurationsfileformat das leserlich ist. Ich

habe mich für den beliebten Yaml Format entschieden, da es sehr leserlich ist, und simple Libraries verfügbar sind die es für mich einlesen können.

Der folgende Codestück demonstriert wie das Fileformat funktioniert:

```
- cube:
  rotation:
    XY: 10
  scale: 0.8
  shift: [1.5, 0, 0, 0]
- cube:
  shift: [-1.5, 0, 0, 0]
  rotation:
    ZY: 20
    YW: 5
```

Man muss einfach den Programm aus einem Command-Terminal starten, und ein Konfigurationsfile als Parameter angeben:

```
$ cd directory_with_binary/
$ ./multicube path_to_config_file
```

3.2.3.6 Meilenstein 6: Die Renderung einer Welt

Key results Das Programm kann eine Welt, definiert durch ein configurations File, rendern. Die Schnitt-Hyperebene und Kamera sind weiterhin dreh und bewegbar. Auf einer HUD (Heads-Up Display) werden die Parameter der Hyperebene und der Kamera präsentiert.

Value Dies ist das erwartete Endprodukt von der ursprünglichen Projektbeschreibung.

Prozess Nachdem das Fileformat implementiert war, hatte ich eine Datenstruktur mit einer Liste von Würfeln, repräsentiert durch Transformationen. Ich habe im Renderer eine Schleife hinzugefügt die diese Liste durchläuft, und für jeden Würfel den Intersektor aufruft. Der Programm war schon so strukturiert, dass keine weitere Änderungen nötig waren. Alles andere funktionierte von alleine.

3.2.4 Die wichtigste Module im Programm

3.2.4.1 Die main() Funktion Die `main()` Funktion ist die Funktion die gestartet wird, wenn das Programm startet. Sie stellt die Ausgangswerte aller Module ein und startet GLEW und GLFW. Danach startet sie eine unendliche Schleife bis das Fenster geschlossen wird. Ein Zyklus in dieser Schleife nennen wir einen Tick. In einem Tick geschieht folgendes:

- Falls sich die Fenstergrösse verändert hat, wird dies im Renderer angepasst.
- Der Renderer rendert ein Bild.
- Alle Tastatur- und Mausevente werden abgefragt und vom Inputmodul gehandhabt.
- Es wird ausgerechnet wie viel Zeit seit dem letzen Tick verstrichen ist.
- Die `tick()` Funktion vom Kameramanager wird gerufen.
- Die `tick()` Funktion vom Hyperebenmanager wird gerufen.

Die zwei `tick()` Funktionen erhalten die verstrichene Zeit als Parameter, damit sie ausrechnen können, was sich wie viel bewegen muss. Sonst kann es vorkommen, das die Kamera schneller ist, sofern der Computer schneller ist, da mehr Ticks pro Sekunde gemacht werden können.

3.2.4.2 Der Intersektor Die Aufgabe des Intersektors ist das Schneiden von einem kanonischen Würfel mit einer 3d Hyperebene.

4d Würfel haben 3d und 2d Seiten. Im folgenden Abschnitt ist mit einer Seite immer eine 2d Seite gemeint.

Die Nachbarschaftsverhältnisse der Seiten Jede Kante im Würfel ist Teil von drei Seiten. Diese drei Seiten nennen wir benachbart.

Zuerst erstellen wir manuell eine Liste der Kanten. Für die Kanten rechnen wir dann alle Seiten aus, die diese Kante beinhalten. Dies geschieht nur ganz am Anfang vom Programm, bevor das erste Bild erscheint. Die Kanten werden von zwei Vektoren repräsentiert, die jeweils die zwei Endpunkte der Kanten beschreiben. Bei der Vorberechnung geben wir jeder Seite eine Identifikationszahl. Wir müssen nur wissen, welche Seite zu welcher Kante gehört und brauchen für den Algorithmus keine weiteren Angaben einer Seite.

Das Schneiden der Kanten Nun schneiden wir alle Kanten mit der Hyperebene. Dabei erhalten wir entweder null, ein oder zwei Ergebnisse. Keine Ergebnisse, wenn die Kante ganz ausserhalb der Hyperebene liegt, ein Ergebniss falls die Kante die Hyperebene kreuzt, und zwei Ergebnisse, falls die Kante mit der Hypereben in einer Ebene liegt. In diesem letzten Fall würde es eigentlich unendlich viele Resultate geben, aber wir nehmen statt dem die zwei Enden der Kanten, da wir schlussendlich nur die Punkte brauchen, die die konvexe Hülle des Volumens ausmachen. Das Schneiden ist trivial; Die Punkte A und B sind die Endpunkte der Kante. Wir betrachten die Distanz $d(A)$ zwischen Punkt A und der Hyperebene, und die Distanz $d(B)$ zwischen Punkt B und der Hyperebene. Falls $d(A) \cdot d(B) > 0$ dann sind beide auf der gleichen Seite der Hyperebene, und es gibt keinen Schnittpunkt. Falls $d = 0$ für eine der Punkte, wird dieser Punkt als Schnittpunkt akzeptiert. Falls $d(A) = 0 \wedge d(B) = 0$ muss die Kante auf der Hyperebene liegen. Somit geben wir beide Endpunkte A und B zurück. Ansonsten berechnen wir aus dem Verhältniss der beiden Distanzen wo auf der Kante der Schnittpunkt liegt und geben das zurück. So erarbeiten wir eine Liste mit allen Schnittpunkten. Gleichzeitig bauen wir eine Datenstruktur auf, die für jede Seite eine Liste von Schnittpunkten beinhaltet. Dies ist ganz einfach, da wir vom Schnittpunkt den wir gerade gefunden haben immer wissen, zu welcher Kante er gehört und wir schon im Voraus berechnet haben, welche Seiten welche Kanten beinhalten. Sobald ein Schnittpunkt gefunden wird, wird es bei allen 3 Seiten die dessen Kante beinhalten, als Schnittpunkt auf der Oberfläche der Seite abgespeichert.

Während diesem Prozess rechne ich auch den Durchschnitt aller Schnittpunkte aus. Dieser wird sich innerhalb des Lösungskörpers befinden und wird sich als Hilfreich erweisen, um Normalvektore zu finden, die gegen Aussen zeigen.

Zwei Schnittpunkte die sich auf der gleiche Würfelseite befinden, werden im entstehenden Schnittvolumen immer eine gemeinsame Kante haben. Aus diesem Grund können wir eine Liste von Nachbarpunkten für jeden Schnittpunkt aufbauen. Dies geschieht aus der Datenstruktur, die für jede Seite die Punkte auf dessen Oberfläche beinhaltet.

Die Zusammensetzung der konvexen Hülle aus Dreiecken Wir könnten mit einer einfachen Tiefensuche alle Dreiecke finden, wo zwei Kanten auch Kanten des Schnittvolumens sind. Daher machen wir eine Tiefensuche

auf dem Nachbarschaftsgraph, behalten immer die vorherigen zwei Punkte im Kopf und immer wenn wir zu einem Punkt gelangen, zeichnen wir das Dreieck mit den vorherigen 2 Ecken. Das habe ich zunächst auch getan. Es treten zwei Probleme auf:

- **Löcher in der Oberfläche** Wenn man eine Fläche hat mit mehr als 4 Ecken, und die Dreiecke auf diese Weise zeichnet, gibt es ein Loch in der Mitte. Das ist auf der Abbildung **??** zu sehen. Die Fläche kann nicht durch Dreiecke abgedeckt sein, die nur Punkte beinhalten welche benachbart sind. Es braucht auch Dreiecke, die aus einem Punkt, und zwei zusätzlichen Punkten, die auf der gegenüberliegenen Seite liegen, bestehen.

- **Z-Fighting** Wenn zwei Objekte und die Kamera in einer Linie sind, verdeckt das eine Objekt das andere mindestens zum Teil. OpenGL geht damit so um, dass im Verlauf des im Abschnitt 3.2.2 beschriebenen Rendering-Pipelines OpenGL doppelte Pixel aussortiert und immer die näher bei der Kamera liegenden behält. Da OpenGL aber mit Gleitkommazahlen arbeitet, die eine kleine aber existente Ungenauigkeit aufweisen, kommt es so, dass falls zwei Flächen sehr genau aufeinanderliegen, es sehr zufällig ist, welche der Flächen behalten, und welche weggeworfen wird. Dies führt zu einem vibrierenden Effekt: Wenn sich das Bild bewegt, ist bei jedem der etlichen Bilder pro Sekunde ganz zufällig eine der Flächen oben.

  Wenn wir so wie beschrieben die Dreiecke zeichnen, dann wird wie man in der Abbildung sieht, ein gewisser Teil immer überlappen. Dies führt zu eiem unschönem Effekt, das Z-Fighting genannt ist.

Um vorläufig eines der Probleme zu beheben, habe ich alle möglichen Kombinationen mit drei Schnittpunkten genommen und sie als Dreieck gerendert. Das löste natürlich das Problem der Löcher. Es führte aber zu noch viel mehr Z-Fighting, da nun viel mehr Flächen überlappten.

Von nun an nenne ich die gefundenen Schnittpunke Ecken oder Eckpunkte. Sie sind nämlich die Ecken des Lösungsobjekts.

Zuerst dachte ich daran, einen Algorithmus für dreidimensionale konvexe Hüllen zu implementieren. Diese sind aber sehr komplex und ich habe gedacht, dass ich mir sicher etwas Einfacheres überlegen kann. So bin ich auf folgenden den Algorithmus gestossen:

Wenn ich ein Dreieck auf einer Fläche des Lösungsobjekts habe, wo zwei der drei Kanten auch Kanten des Lösungsobjekts sind, kann ich diese ganze Fläche des Lösungsobjekts mit Dreiecken belegen, ohne Überlappungen oder Löcher. Das Funktioniert folgendermassen: Ich merke mir den Normalvektor des Dreiecks, das ausserhalb des Lösungsobjekts zeigt. Um dies zu berechnen, benutze ich den Durchschnitt aller Ecken des Lösungskörpers, den ich generiert habe als ich alle Ecken berechnete. Ich nehme einen der Punkte, der nicht in der Mitte liegt, als Ausgangspunkt. Der mittlere Punkt ist der Punkt, der beide anderen Punkte als Nachbar hat. Falls alle drei Punkte diese Eigenschaft besitzen, ist die ganze Fläche nur ein Dreieck, da die drei Punkte durch drei Kanten des Lösungsobjekts verbunden sind. Dann bricht der Algorithmus ab. Ansonten habe ich jetzt einen Ausgangspunkt, einen mittleren Punkt, und einen Endpunkt. Ich gehe jetzt alle Nachbarn vom Endpunkt durch. Ich kontrolliere immer, ob das Dreieck, das durch den Augangspunkt, dem Endpunkt, und dem neuen Punkt gebildet wird, den gleichen Normalvektor gegen Aussen hat, wie das ursprüngliche Dreieck. Falls ja, liegt es auf der gleichen Ebene, ist also Teil der gleichen Fläche. Dieses Dreieck zeiche ich. Der vorherige Endpunkt wird jetzt zum mittleren Punkt, und der neue Punkt zum Endpunkt. Ich widerhole diesen Prozess, bis der Endpunkt und der Augangspunkt der gleiche Punkt sind. Dann breche ich ab. Jedesmal wenn ich ein Dreieck einzeichne, merke ich mir, dass es eingezeichnet wurde. Ich speichere auch jedesmal ab, dass das Dreieck, gebildet durch den mittleren Punkt, dem Endpunkt, und dem richtigen neuen Punkt, bereits gezeichnet ist. Zwar ist dies nicht direkt schon gezeichnet, aber dessen Fläche wird nach dem Ende des Algorithmus eingefüllt sein.

Jetzt mache ich die Tiefensuche, die ich Oben beschrieben habe. Ich suche jedes Dreieck, wo zwei Kanten auch Kanten des Lösungsobjekts sind. Falls dieses Dreieck noch nicht ausgefüllt ist, lasse ich den Algorithmus laufen.

Hier zeige ich warum das immer eine richtige Lösung geben wird:

- Jede Fläche wird Dreiecke enthalten mit so einem Startdreieck, wo zwei Kanten Teil des Lösungsobjekts sind.
- Es wird von jedem Dreieck, das aufgebaut ist wie ein Startdreieck, versucht den Algorithmus zu starten. Nur wenn das Dreieck schon als ausgefüllt markiert ist, wird der Algorithmus doch nicht laufen gelassen.
- Wenn von einem Startdreieck der Algorithmus gestartet wird, wird die ganze Fläche ausgefüllt, ohne überlappende Dreiecke.

- Es wird nie zweimal die gleiche Fläche ausgefüllt, da alle mögliche Startdreiecke während dem Ausfüllen einer Fläche als ausgefüllt markiert werden.

3.2.4.3 Der Kameramanager Dieses Modul beinhaltet alles, was zur Kamera gehört. Es beinhaltet die Position und Winkel der Kamera, Konstanten die die Dreh- und Bewegungsgschwindigkeit beeinflussen, sowie die momentane Bewegungsrichtung. Es kann jederzeit eine Transformationsmatrix vom Kameramanager abgefragt werden, die alle Objekte in der Welt so bewegt, dass simuliert wird, dass die Kamera in einer anderen Position ist. Dies ist nötig, da in OpenGL die Kamera nicht bewegt werden kann. Stattdessen muss die ganze Welt so gedreht und verschoben werden, dass das simuliert werden kann. Jeden Tick macht das Kameramanager ausserdem folgende Aktionen:

- Es bewegt die Kamera in die Richtung angegeben durch die momentan gedrückten Tasten. Falls keine gedrückt sind, bewegt sich die Kamera nicht.
- Es berechnet die Transformationsmatrix neu.

3.2.4.4 Der Hyperebenenmanager Der Hyperebenenmanager speichert alle nötigen Parameter der Hyperebene. Dies ist ein Vektor, der dessen Position beschreibt, 3 Vektore, welche die drei Einheitsvektoren der dreidimensionalen Hyperebene bilden und ein Normalvektor. Wie der Kameramanger beinhaltet er auch Konstanten für die Dreh- und Bewegungsgschwindigkeit. Ausserdem beinhalten er einen Vektor für die momentane Drehrichtung und eine Zahl für die momentane Bewegungsrichtung. Hier braucht es nur eine Zahl, da es nur in die Richtung der eigenen Normalen verschoben werden kann, und somit nur angegeben werden muss, ob es in die positive oder negative Richtung bewegt wird. Die `tick()` Funktion des Hyperebenenmanagers macht folgendes:

- Die Hyperebene drehen, jenachdem welche Tasten gedrückt sind.
- Die Hyperebene bewegen, falls eine der Bewegungstasten gedrückt ist.

3.2.4.5 Inputmanager

Der Inputmanager ist nur eine Schnittstelle von GLFW zu meinem Programm. Es implementiert sogenannte 'listener'. Das sind Funktionen, die ich selber nicht rufen werde, sondern im Fall das eine Taste gedrückt wird oder die Maus bewegt wird, von GLFW gerufen werden.

Wenn ein sogenanntes 'Mausevent' wahrgenommen wird, ruft es ganz einfach die richtigen Funktionen vom Kameramanager.

Etwas komplizierter ist was bei einem Tastendruck geschieht. Es hat nämlich zwei 'Keyboardevent's, 'Press' und 'Release'. 'Press' wird genau einmal in dem Moment gerufen, wenn eine Taste gedrückt wird. Egal wie lange man die Taste gedrückt hält, es gibt nur einen Event. 'Release' wird gerufen, sobald eine Taste losgelassen wird.

Für mich ist aber interessant, ob eine Taste momentan gedrückt ist. Ich muss also selber abspeichern in welchem Zustand welche Tasten sind. Sobald eine Taste gedrückt wird, merke ich dass sie gedrückt ist, und sobald es losgelassen wird merke ich, dass sie nicht gedrückt ist. Ob eine Taste gedrückt ist oder nicht, kann dann jedes Modul, welches auf den Inputmanager zugriff hat, abfragen.

3.2.4.6 Der Renderer

Der Renderer ist dafür zuständing in jedem Tick mithilfe von OpenGL ein Bild zu zeichnen.

Ganz am Anfang beim starten des Programms, schickt es die Schattierungsprogramme durch OpenGL an die Grafikkarte und reserviert Speicher auf der Karte, für die Daten die es später schicken wird.

Er hat ausserdem eine weitere Funktion, die nicht die `render()` Funktion ist. Mit `set_size()` kann die grösse des Fensters angepasst werden falls es sich geändert hat. Das wird in der `main()` Funktion aufgerufen, wie bereits beschrieben.

Die `render()` Funktion macht folgendes, in dieser Reihenfolge:

- Sie lässt den Intersektor alle Dreiecke generieren.
- Sie berechnet die Matrix, die die Perspektive der Kamera simuliert.
- Sie berechnet mithilfe des Kameramanagers die Matrix, welche die Welt richtig dreht und verschiebt.
- Sie schickt diese Matrizen an die Grafikkarte.

- Sie schickt alle nötigen Daten für die Lichtberechnung wie Materialeigenschaften, Lichtintensität und Richtung und die Normalvektore der Dreiecke an die Grafikkarte.
- Sie schickt alle Dreiecke an die Grafikkarte.
- Sie schickt ein Signal, damit die Grafikkarte ein neues Bild generiert und an den Monitor schickt.

3.2.4.7 Shaderprogramme Die Shaderprogramme, oder Schattierungsprogramme auf Deutsch, sind die Programme, die auf die Grafikkarte geladen werden und dort mit den von mir angegebenen Daten die Farbe der einzelnen Pixel berechnen.

Ich habe zwei Shaderprogramme: Einen Vertex- und einen Fragmentshader. Zwischen diesen beiden Programmen durchlaufen die Daten den Rest des Rendering-Pipeline's, darunter auch die Rasterization. Während der Rasterization werden alle Daten die spezifisch sind für einzelne Punkte auf das Dreieck interpoliert.

Lichtberechnung Damit man Objekte gut erkennen kann braucht es Lichteffekte. Ohne Lichteffekte kann man schwer bis garnicht erkennen wo eine Fläche anfängt und wo sie endet, oder wo die Kanten und Ecken sind. Man sieht nur die Umrisse des Objekts, als wäre es ein farbiger Schatten. Wenn man sich bewegt kann man die Kanten und Ecken an den Veränderungen erahnen, aber auch das ist nicht verlässlich.

Also möchten wir Licht simulieren. Das ist aber nicht ganz einfach. In der echten Welt sehen wir Licht und Schatten, weil Photone von Energiequellen ausgeschieden werden, die dann an Objekten abspringen, und schliesslich in unser Auge gehen, das wir dann wahrnehmen. Heutzutage wäre die Simulation von diesem Verhalten nicht unmöglich; man stösst immer häufiger auf sogenannte Ray-Tracing Algorithmen. Diese sind aber sehr komplex, und brauchen oft sehr viel Rechenleistung. Aus diesem Grund wähle ich eine einfachere Methode. Solche Methoden, welche nicht die echte Physik simulieren, nennen wir oft Modelle.

Eines der weitverbreitetsten Modelle ist das 'ADS' Modell.

- **Ambiente Reflexion** ist überall gleich stark, die Richtung des Lichtes und die Position des Punktes auf dem Körper spielt keine Rolle.

- **Diffuse Reflexion** ist stärker oder schwächer, jenachdem woher das Licht kommt, wo sich der Punkt auf dem Körper im Verhältniss zum Licht befindet, und was der Einfallswinkel des Lichtes ist.

- **Spiegelnde Reflexion** macht einen Körper glänzend. Es rechnet kleine Flecken aus, wo das Licht besonders hell erscheint, und bildet glänzende Flecken auf dem Körper.

Es existieren viele verschiedene Arten von Belichtungselementen die man implementieren könnte. Ich werde aber zwei Elemente benutzen:

- **Ambiente Belichtung** , also Licht wo nur eine ambiente Komponente enthält. Dies dient als Minimum, nichts kann stockdunkel sein, da alles von diesem Licht etwas erhellt wird.

- **Richtungslicht** , Licht das nur eine Richtung hat, nicht aber einen Ursprungspunkt. Das Simuliert sehr weit entfernte Lichtquellen, wie zum Beispiel die Sonne.

Wie fest ein Objekt auf ambiente, diffuse und spiegelnde Reflexion reagiert, kommt auf die Materialeigenschaften an. Lichter und Materialien haben beide ADS Werte. Diese geben wir mit RGBA (Red Green Blue Alpha) an. Materialien haben ausserdem einen sogenannten Glanzwert n; wie Glänzend das Material ist.

Für jeden Pixel müssen wir nun die Intensität I der Lichtreflexion bestimmen, den wir wahrnehmen. Das ist gleich der Summe von der ambienten, diffusen und spiegelnden Reflexion den wir in diesem Pixel wahrehmen.

$$I_{beobachtet} = I_{ambient} + I_{diffus} + I_{spiegelnd}$$

Die ambiente Reflexion $I_{ambient}$ setzt sich folgendermassen zusammen:

$$I_{ambient} = Licht_{ambient} \cdot Material_{ambient}$$

Die diffuse Reflexion I_{diffus} ist schon etwas komplizierter, da es auch vom Einfallswinkel des Lichtes θ abhängt. Genauer, die Menge des Lichtes das vom Punkt reflektiert, ist proportional zu $\cos\theta$.

$$I_{diffus} = Licht_{diffus} \cdot Material_{diffus} \cdot \cos\theta$$

Um dies zu berechnen benutzen wir den Normalvektor der Fläche $\hat{N}$, und den Richtungsvektor des Lichtes $\hat{L}$. Das Skalarprodukt der beiden Vektoren gibt uns den Cosinus ihres Winkels; genau das brauchen wir. Ausserdem wollen brauchen wir diffuses Licht nur, wenn $-90 \leq \theta \leq 90$, da das Licht sonst gar nicht auf die Oberfläche scheint. Cosinus ist genau in diesem Bereich positiv. Also benutzen wir folgende Gleichung:

$$I_{diffus} = Licht_{diffus} \cdot Material_{diffus} \cdot \max(\hat{N} \cdot \hat{L}, 0)$$

Die spiegelnde Reflexion ist etwas spannender. Dies simuliert die überbelichtete Flecken, wo das Licht durch den Objekt genau in das Auge des Betrachters reflektiert wird. Wie viel spiegelnde Reflexion in einem Punkt ist, kommt darauf an, wie klein der Winkel ϕ vom austretenden Lichtvektor $\hat{R}$, und dem Vektor vom Punkt zum Auge $\hat{V}$ ist. Wir können die Cosinusfunktion als 'falloff' Funktion verwenden. Der Glanzwert des Materials ist der Exponent das die Cosinusfunktion bekommmt. So können wir die Falloff-Funktion stärker oder schwächer machen. So kommen wir zu der Gleichung:

$$I_{spiegelnd} = Licht_{spiegelnd} \cdot Material_{spiegelnd} \cdot \max((\hat{R} \cdot \hat{V})^{n}, 0)$$

Wieder nehmen wir nur die positiven Werte. Sonst können spezielle Artifakte entstehen mit inverser Glanzflecken, flecken wo dünkler sind.

Diese Methode Lichteffekte zu berechnen ist das Phong-Beleuchtungsmodell. Ich habe es aus dem OpenGL Buch, und Wikipedia gelernt. Es ist ein empirisches Modell; es achtet nicht darauf, das nicht mehr Licht entsteht durch die Reflexion wie es vor der Reflexion gegeben hat. Es ist aber ein einfaches Modell, und die Unterschiede zu anderen Modellen ist nicht gross.

Der Vertexshader Im Vertexshader bearbeitet man einen einzelnen Punkt. Man kann alles daran verändern, was ich auch ausnutze. Wie im Renderer beschrieben werden zwei Matrizen an die Grafikkarte geschickt, die die Welt so drehen und wenden, dass die Kameraposition und die Perspektive stimmen. Diese wenden wir auf den Punkt an.

Anfangs habe ich ausserdem die Position des Punktes genommen, und da die Farbe des Punktes auch als Vektor repräsentiert wird, es einfach auch als Farbe angegeben. Das resultierte in einem Objekt, das einen Farbverlauf hatte, da die Interpolation auch die Farben interpolierte. Dies ist ein

gutes Beispiel dafür, wie Daten durch die Rendering-Pipeline geschickt werden können, und dann auf der Fläche des Dreiecks interpoliert angewendet werden.

Für die Lichteffekte brauche ich aber komplizierterere Shader. Hier für rechne ich drei Vektore aus der position des Punktes, der Matrixtransformation der Kamera, und dem angegebenem Normalvektor der Fläche. Den Betrachtungsvektor $\overrightarrow{V}$, den Lichteinfallsvektor $\overrightarrow{L}$ und den Normalvektor $\overrightarrow{N}$. Ich schike die durch den Rasterizer, und werde sie erst im Fragmentshader wieder brauchen.

Der Fragmentshader Im Fragmentshader wird ein einzelner Pixel bearbeitet. Hier geben wir an welche Farbe der Pixel annehmen soll. Wir können Parameter verwenden wie die Koordinaten des Pixels, oder die Identifikationszahl des Dreiecks zu dem der Pixel gehört. Ausserdem können wir Daten verwenden die wir aus dem Vertexshader weitergeschickt haben, oder Konstanten die wir im Voraus auf die Grafikkarte geladen haben.

Vom Vertexshader wurden $\overrightarrow{V}$, $\overrightarrow{L}$ und $\overrightarrow{N}$ generiert, und weitergeschickt. Durch die Interpolation sind sie sogar genau dem Pixel entsprechend den ich jetzt bearbeite, obwohl ich sie nur für die drei Ecken des Dreiecks berechnet habe.

Hier rechne ich jetzt $\overrightarrow{R}$ aus. Das geht ganz einfach, ich spiegle $\overrightarrow{L}$ an $\overrightarrow{N}$.

Jetzt kann ich einfach die Formeln für ambiente, diffuse, und spiegelnde Reflexion verwenden, und dessen Summe als Farbe ausgeben. So erhalte ich den vollständigen Bild, der schlussendlich auf dem Bildschirm zu sehen ist.

3.3 Bedienungsanleitung

3.3.1 Installation

3.3.1.1 Windows

Die Binärdateien herunterladen

- Lade den Release-Zip-File von der Releases Seite auf GitHub herunter, und entzippe es. `https://github.com/NandorKovacs/4dvisualizer/releases`

Die Binärdateien befinden sich im resultierenden Ordner, und die Weltkonfigurationen in einem Ordner mit dem Namen `worlds`.

Um die neuste Binärdatei laufen zu lassen, benutze folgenden Command:
`multicube.exe ..\progs\multicube\worlds\two_rotated_cubes.world.yaml`

Den Quellcode kompilieren

- Installiere Build Tools for Visual Studio 2022 von `https://visualstudio.microsoft.com/downloads/`.
- Installiere CMake (>3.24) von `https://cmake.org/download/`.
- Lade den Quellcode-Zip-File von der Releases Seite auf HitHub herunter, und entzippe es. `https://github.com/NandorKovacs/4dvisualizer/releases`
- Im Start Menü, starte 'x64 Native Tools Command Prompt for VS 2022', wechsle zum Ordner des Quellcodes mithilfe von 'cd', und lasse folgende Commands laufen:

```
mkdir build
cd build
cmake ..
cmake --build . --config Release
```

Alle Libraries sollen automatisch heruntergeladen und kompiliert werden. Die resultierende Binärdateien befinden sich hier:
`build/progs/<program name>/Release/<program name>.exe`.

3.3.1.2 Linux Um den Quellcode auf Linux zu kompilieren, werden die gleichen Schritte wie für Windows benötigt. Build Tools for Visual Studio 2022 wird aber nicht benötigt.

Auf Ubuntu, benutze stattdessen diesen Command:

```
sudo apt-get install build-essentials
```

3.3.2 Konfigurations Format

In einem Yaml File kann man Würfel beschreiben. Für jeden Würfel können mehrere Rotationen, eine Grösse und ein Ursprungsvektor angegeben werden. Es wird aufgelistet um wie viel Grad um welche Fläche gedreht wird. Eine Zahl wird angegeben für die Grösse. Und ein Vektor wird angegeben als Ursprung.

```
- cube:
  rotation:
    XY: 10
  scale: 0.8
  shift: [1.5, 0, 0, 0]
- cube:
  shift: [-1.5, 0, 0, 0]
  rotation:
    ZY: 20
    YW: 5
```

3.3.3 Das Programm mit einer gegebenen Konfiguration starten

- Finde die installierte Binärdatei.
- Lasse die Binärdatei von einem Command-Terminal laufen, und gebe den konfigurations File als einzigen Parameter an.

3.3.4 Das Programm steuern

- Das Programm beenden: Q
- Die Kamera bewegen: A — links, D — rechts, W — forwärts, S — rückwärts, Shift — Hoch, Ctrl — Runter
- Die Kamera drehen: Maus
- Die Hyperebene auf seinem Normalvektor verschieben: R — positive Richtung, F — negative Richtung
- Die Hyperebene drehen: O, P, [und L, ö, ä. (Auf einer Deutsch-Schweizer Tastatur. Auf anderen Tastaturen sollten die Tasten funktionieren, wo sich sonst ö oder ä befinden)

4 Literaturverzeichnis

4.1 Bücher

- John L. Clevenger and V. Scott Gordon: Computer Graphics Programming in OpenGL with C++
- Bjarne Stroustrup: The C++ programming language

4.2 Internetseiten

- `https://www.khronos.org/opengl/wiki/`
  - `https://www.khronos.org/opengl/wiki/Data_Type_(GLSL)` (9.1.2023)
  - `https://www.khronos.org/opengl/wiki/Uniform_(GLSL)` (9.1.2023)
  - `https://www.khronos.org/opengl/wiki/Built-in_Variable_(GLSL)` (9.1.2023)
  - `https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview` (9.1.2023)
  - `https://www.khronos.org/opengl/wiki/Vertex_Shader` (9.1.2023)
  - `https://www.khronos.org/opengl/wiki/Fragment_Shader` (9.1.2023)
- `https://en.wikipedia.org/`
  - `https://en.wikipedia.org/wiki/Tesseract` (6.1.2023)
  - `https://en.wikipedia.org/wiki/Four-dimensional_space` (9.1.2023)
  - `https://en.wikipedia.org/wiki/Rotations_in_4-dimensional_Euclidean_space` (9.1.2023)

  - https://de.wikipedia.org/wiki/Drehmatrix (9.1.2023)
  - https://en.wikipedia.org/wiki/Phong_reflection_model (9.1.2023)
  - https://en.wikipedia.org/wiki/Phong_shading (9.1.2023)
  - https://en.wikipedia.org/wiki/Transformation_matrix (9.1.2023)
  - https://en.wikipedia.org/wiki/Z-fighting (8.1.2023)
  - https://en.wikipedia.org/wiki/GLFW (8.1.2023)
  - https://en.wikipedia.org/wiki/OpenGL (8.1.2023)
  - https://en.wikipedia.org/wiki/OpenGL_Shading_Language (8.1.2023)
  - https://de.wikipedia.org/wiki/Satz_vom_Fu%C3%9Fball (9.1.2023)
  -

- https://www.qfbox.info/4d/vis/10-rot-1 (8.1.2023)
- https://henders.one/2022/05/09/lost-4D-rotation/ (8.1.2023)
- https://www.glfw.org/docs/latest/ (8.1.2023)
  - https://www.glfw.org/docs/latest/intro_guide.html (9.1.2023)
  - https://www.glfw.org/docs/latest/window_guide.html (9.1.2023)
  - https://www.glfw.org/docs/latest/context_guide.html (9.1.2023)

  - https://www.glfw.org/docs/latest/input_guide.html (9.1.2023)
  - https://www.glfw.org/docs/latest/monitor_guide.html (9.1.2023)
- https://github.com/NandorKovacs/4dvisualizer/releases
- https://github.com/NandorKovacs/4dvisualizer/
- https://visualstudio.microsoft.com/downloads/
- https://cmake.org/download/

Der/die Unterzeichnete bestätigt mit Unterschrift, dass die Arbeit selbständig verfasst und in schriftliche Form gebracht worden ist, dass sich die Mitwirkung anderer Personen auf Beratung und Korrekturlesen beschränkt hat und dass alle verwendeten Unterlagen und Gewährspersonen aufgeführt sind.

Adliswil	9.1.2023	Nandor Kovacs
Ort	Datum	Unterschrift