HOCHSCHULE COBURG

Hochschule für angewandte Wissenschaften Coburg

Fakultät Elektrotechnik und Informatik

Studiengang: Informatik (Sem. 5)

Praxisbericht

Programmierung eines Rubik's Cube in JavaScript unter Verwendung der Programmierschnittstelle WebGL

Andreas Schwarzmann

Abgabe der Arbeit: 1. Februar 2018

Betreut durch:

Prof. Dr. Meyer Quirin, Hochschule Coburg

Inhaltsverzeichnis

Abbildungsverzeichnis

Codebeispielverzeichnis

1 Einleitung

1.1 Vorstellung

Wir leben in einer 3D-Welt. Diese nehmen wir wahr, indem wir uns in drei Dimensionen bewegen und dreidimensional denken. Auch der Großteil unserer Medien wird dreidimensional dargestellt, wie zum Beispiel Kinofilme, Videospiele oder online Dienste wie „Google Maps“. Damals benötigte man für das Erstellen von 3D-Grafiken nicht nur einen Hochleistungsrechner, sondern auch teure Software. Innerhalb des letzten Jahrzehnts hat sich das grundlegend verändert. Heutzutage wird jeder Computer und jedes Smartphone mit einem Grafikchip ausgeliefert, der leistungsstärker ist, als die professionellen Grafikrechner vor 15 Jahren. Die Software für das Ausführen der Anwendungen ist heutzutage universell verfügbar und kostenlos. Diese nennt sich Web Browser [Parisi 2014, S. 3].

Seit HTML5 besitzt der Web Browser mehr Funktionalität und Leistung als zuvor. Das macht es dem Browser möglich auch anspruchsvolle 3D-Anwendungen problemlos ausführen zu können. Darüber hinaus gibt nun es eine Reihe von Grafiktechnologien, die zum Teil eine wichtige Rolle in dieser Arbeit spielen. Eines der wesentlichen Neuheiten nennt sich *WebGL*. WebGL ist eine standardisierte Grafik-Programmierschnittstelle, die auf der OpenGL-API basiert und von fast allen Browsern unterstützt wird. Man verwendet diese Schnittstellen hauptsächlich für das hardwarebeschleunigte 3D-Zeichnen mit JavaScript [Parisi 2014, S. 5].

1.2 Zielsetzung

Im Rahmen des Wahlpflichtfachs Computergrafik wurden viele verschiedene Konzepte vorgestellt, wie das Zeichnen von Dreiecken, Transformationen oder die Berechnungen für Beleuchtungseffekte. Dieses umfangreiche Theoriewissen soll nicht nur in verschiedenen Praktikumsaufgaben Anwendung finden, sondern soll auch in einem selbst gewählten Projekt praktisch umgesetzt werden, dass möglichst viele der gelehrten Themen abdeckt.

Als Computergrafik-Projekt möchte ich einen dreidimensionalen Rubik’s Cube unter Verwendung der Programmiersprache JavaScript und der WebGL-Schnittstelle implementieren. Diese Zielsetzung besteht aus verschiedene Teilaufgaben, die im nächsten Kapitel vorgestellt werden.

1.3 Aufgabenstellung

Für das Erreichen der Zielsetzung wurde eine feste Aufgabenstellung definiert, die sich in verschiedene Teilaufgaben untergliedert. Am Ende des Projekts soll sich der Würfel von allen Seiten aus betrachten lassen. Jedes Würfelelement soll zunächst auswählbar sein und sich dann um die X-, Y- oder Z-Achse drehen lassen. Die Rotation um 90° soll mit einer Animation dargestellt werden. Der Rubik's Cube soll darüber hinaus in einer 3D-Umgebung schweben und von einer Lichtquelle beleuchtet werden. Außerdem wird eine sinnvolle Steuerung für alle Interaktionen, wie zum Beispiel für das Rotieren der Seiten oder das Verändern der Kameraposition, benötigt. Aufgrund des vorgestellten Umfangs werden weitere Features, wie zum Beispiel ein „Verdreh"- und „Löser"-Algorithmus nicht berücksichtigt.

1.4 Aufbau der Arbeit

Der Aufbau dieser Arbeit orientiert sich strikt nach den genannten Teilaufgaben, die das Projekt erfüllen soll. Hierzu bezieht sich jeder Oberpunkt der Gliederung auf genau einen Teilaspekt der Aufgabenstellung. In jedem dieser Kapitel wird dann auf die einzelnen Konzepte, Eigenschaften und Implementierungen eingegangen. Mit Hilfe von Abbildungen und Codesegmenten werden die wichtigsten Bereiche der Programmierung veranschaulicht. Auf allgemeine Konzepte von WebGL kann nicht an jeder Stelle detailliert eingegangen werden, daher wird dort auf entsprechende Referenzen verwiesen. Am Schluss folgt eine kurze Zusammenfassung.

Für den Großteil dieser Arbeit sollte ein grundlegendes Verständnis über JavaScript-Programmierung und Computergrafik vorhanden sein. Für das Projekt wurde außerdem die frei verfügbare JavaScript-Bibliothek „gl-matrix.js" verwendet.

2 Zeichnen des Würfels

In diesem Kapitel wird auf die wesentlichen Merkmale des Würfels eingegangen. Darüber hinaus wird die Vorgehensweise erklärt, wie der Würfel schrittweise aufgebaut werden soll.

2.1 Konzept und Logik

Der gesamte Rubiks Cube besteht aus 27 Teilwürfeln (3x3x3), die in einem dreidimensionalen Array abgespeichert werden. Jeder Teilwürfel soll einzeln mit seinen jeweiligen Positionen und weiteren Eigenschaften, wie z.B. den unterschiedlichen Seitenfarben gezeichnet werden. Das zentrale Element liegt dabei im Ursprung des Koordinatensystems bei [0, 0, 0].

Bevor die Würfel gezeichnet werden können, müssen zuerst deren Attribute festgelegt werden. Danach werden diese mit Hilfe von drei verschachtelten Schleifen nacheinander gezeichnet, indem für jeden Teilwürfel die Array- und Indexbuffer explizit neu geladen und gerendert werden. Die nachfolgende Abbildung soll das genannte Konzept veranschaulichen.

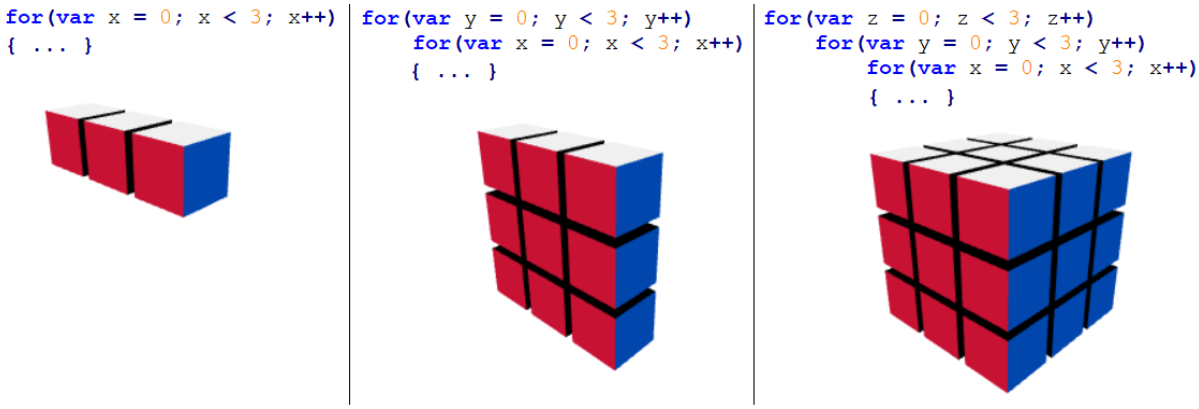

Abbildung 1: Zeichnen mit Schleifen

2.2 Eigenschaften der Teilwürfel

Für die Teilwürfel existiert eine eigene Klasse namens „SubCube“. Beim Anlegen eines Objekts dieser Klasse werden diesem eine Würfel-ID, ein Vertex- und ein Indexarray übergeben. Alle SubCube-Seiten werden durch das Zeichnen von zwei Dreiecken realisiert. Das Vertexarray enthält alle Eckpositionen die zum Zeichen dieser Dreiecke mit Hilfe des Array-Buffers notwendig sind. Das Vertex-Array beinhaltet neben den Positionen noch RGBA-Farbwerte und einen Normalenvektor für jede Seite.

Das Indexarray enthält Verweise auf die Eckpositionen im Vertexarray. Die erwähnten Attribute werden dann an den SubCube-Konstruktor weitergereicht, der die spezifischen Eigenschaften des Teilwürfels festlegt. Außerdem wird im Konstruktor eine zusätzliche feste Positions-ID und eine 4x4 Identitätsmatrix angelegt, die für die Rotationen später notwendig sind.

Die Position- und Indexarrays werden unter Verwendung der verschachtelten Schleifen (s. Abbildung 1) befüllt, indem für jede der sechs Teilwürfelseiten genau vier Koordinaten vergeben werden. In der nachfolgenden Zeichnung werden die vier Eckpunkte für die obere Seite eines Teilwürfels dargestellt. Damit zum Zeichnen der beiden oberen Dreiecke die Eckpunkte nicht doppelt im Vertex-Buffer liegen, gibt es den Indexbuffer, der für das Beispiel so aussieht: 0, 1, 2, 0, 2, 3.

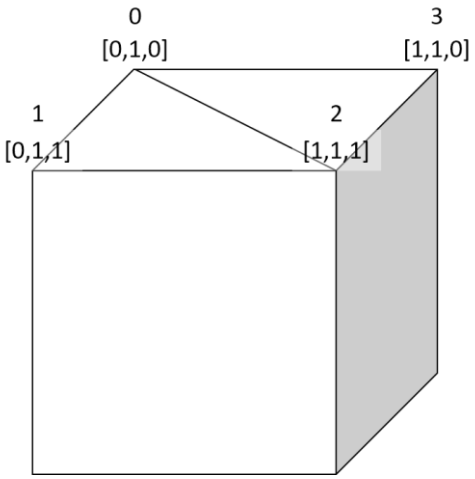

Abbildung 2: Indexpositionen

2.3 Vertex- und Fragmentshader

Für das Zeichnen der Vertex-Indizes mittels des Befehls „gl.drawElements(…)“, benötigt man sowohl einen Vertex- als auch einen Fragmentshader. Im Vertex-Shader werden die Positionsinformationen aus dem aktuell gebundenen Array- bzw. Index-Buffer geladen und mit einer besonderen Matrix multipliziert, um die es ausschließlich im nächsten Kapitel gehen soll.

Nach dieser Multiplikation wird die neue Vertex-Position der impliziten GLSL-Variable „gl_Position“ zugewiesen. Dadurch weiß die Grafikkarte an welcher Position gezeichnet werden soll. Die jeweiligen Seitenfarben des Teilwürfels werden mit einer „out“-Variable an den Fragment-Shader weitergegeben, der diese ebenfalls an die Hardware weitergibt.

Betrachtet man das bloße Rendern der Würfel, dann lässt sich der Vertex-Shader so darstellen:

```
in vec3 vertPosition;
in vec4 vertColor;
out vec4 outColor;
uniform mat4 mWorld;

void main()
{
     gl_Position = mWorld * vec4(vertPosition, 1.0);
     fragColor = vertColor;
}
```

Code 1: Vertex-Shader für die Würfelzeichnung

Der dazugehörige Fragmentshader schaut dann folgendermaßen aus:

```
precision mediump float;
in vec4 outColor;

void main()
{
     if(abs(outPosition.x) == 1.0
     || abs(outPosition.y) == 1.0
     || abs(outPosition.z) == 1.0)
     {
          fragColor = outColor;
     }
     else
     {
          fragColor = vec4(0.0, 0.0, 0.0, 1.0);
     }
}
```

Code 2: Fragment-Shader für die Würfelzeichnung

Die If-Anweisung wird benötigt, damit nur die Positionen an den Außenseiten des Würfels eine Farbe erhalten, da an diesen Stellen mindestens eine der X-, Y- oder Z-Koordinaten eine 1 bzw. -1 haben muss. Alle anderen Positionen, die innerhalb des gesamten Rubik's Cube liegen, werden dann folglich schwarz gezeichnet.

3 Blickwinkel und Kameraposition

Jede dreidimensionale Szene in der Computergrafik benötigt eine festgelegte Blickposition, aus der ein Betrachter die Zeichnung sehen kann. Hierzu werden zwei 4x4-Matrizen benötigt. Die Erste nennt sich View-Matrix und definiert die Position und Ausrichtung der Kamera im 3D-Schauplatz. Die Zweite heißt Projection-Matrix und übersetzt durch Multiplikation das dreidimensionale Sichtfeld der Kamera in eine 2D Zeichenebene, die schließlich von der Hardware zum Rendern verwendet werden kann [Parisi 2014, S. 13f].

3.1 View- und Projection-Matrix

Durch die initial beschriebene Matrix-Multiplikation der View- und Projection-Matrix erhält man eine neue Matrix, die als View-Projection-Matrix bezeichnet wird.

$$ViewProjectionMatrix = ViewMatrix \cdot ProjectionMatrix$$

Die folgenden Codezeilen zeigen die Initialisierung und Multiplikation der beiden Matrizen.

```
var viewMatrix = new Float32Array(16);
mat4.lookAt(viewMatrix, [0, 0, 8], [0, 0, 0], [0, 1, 0]);

var projMatrix = new Float32Array(16);
mat4.perspective(projMatrix, glMatrix.toRadian(45),
     canvas.clientWidth / canvas.clientHeight, 0.1, 1000.0);

mat4.multiply(viewProjMatrix, projMatrix, viewMatrix);
```

Code 3: Setup der View-Projection-Matrix

Die View-Projection-Matrix wäre nun ausreichend, um alle 27 Teilwürfel zu zeichnen. Da sich aber die einzelnen Teilwürfel bei Seitenrotationen drehen, verändern sich deren Vertexpositionen. Damit man das Vertex-Array nicht ständig neu aufbauen muss, besitzt jeder Teilwürfel eine Drehmatrix, die als Identitätsmatrix initialisiert wird. Diese wird dann für jeden Würfel nochmals mit der View-Projection-Matrix multipliziert, um korrekte Positionen im festgelegten 3D-Raum zu erhalten. Das Ergebnis wird dann als Uniform-Variable an den Vertex-Shader weitergegeben. Diese Variable wurde in Code 1 als „mWorld“ bezeichnet. Der nachfolgende Code zeigt dies geschilderte Multiplikation und den Upload an den Vertexshader.

```
mat4.multiply(uMatrix, viewProjMatrix,
subCubes[z][y][x].rotatedMatrix);

gl.uniformMatrix4fv(matWorldUniformLocationCube, gl.FALSE,
uMatrix);
```

Code 4: Multiplikation mit Teilwürfel-Matrix

3.2 Kamerasteuerung

Für die Steuerung der Kamera bzw. das Verändern des Sichtbereichs auf den Würfel werden verschiedene Interaktionen benötigt. Diese Interaktionen werden in Form von „ActionEvents“ mit der Maus ausgelöst und mit „EventListener“ abgefragt. Durch einen einfachen Mausklick auf die Umgebung werden die aktuellen Mauspositionen festgehalten. Bewegt man die Maus mit gehaltener Taste, wird der Abstand der neuen und der alten Position, in X- und Y-Richtung berechnet. Mit zunehmenden Abstand erhöht sich die jeweilige Gradzahl für die Rotation.

Dadurch lässt sich die View-Projection-Matrix entsprechend verändern, indem man diese mit die ermittelten Gradwerten um die X- und Y-Achse dreht.

```
mat4.rotateX(viewProjMatrix, viewProjMatrix, move_angle_y);
mat4.rotateY(viewProjMatrix, viewProjMatrix, move_angle_x);
```

Code 5: Verändern des Sichtbereichs

Außerdem löst das Mausrad ebenfalls ein ActionEvent aus, das eine globale Zoom-Variable je nach Drehrichtung des Rads vergrößert oder verkleinert. Diese Zoom-Variable wird dann mit der Z-Koordinate der Kameraposition in der „lookAt()“-Funktion multipliziert. Dadurch wird die Kameraposition beim erneuten Erstellen der View-Matrix durch den Zoom-Faktor beeinflusst und die Kamera befindet sich dann näher oder weiter weg vom Würfel.

```
mat4.lookAt(viewMatrix, [0, 0, 8*zoom], [0, 0, 0], [0, 1, 0]);
```

Code 6: Zoomen mit dem Mausrad

4 Rotation

In diesem Kapitel werden die einzelnen Komponenten, die für das Drehen der Seiten nötig sind, vorgestellt. Außerdem werden auch Steuerung und Animation der Drehung näher erläutert.

4.1 Auswahl eines Teilwürfels

Um eine Seitenrotation ausführen zu können, muss zuerst ein Teilwürfel durch einen Linksklick ausgewählt werden. Die Auswahl eines Teilwürfels wird mit einem durchdachten Trick realisiert, indem der Alphawert, beim Initialisieren der Vertex-Attribute jedes Würfels, einen besonderen Wert bekommt. Dieser erhält den SubCube-Index, der jeden Teilwürfel identifiziert.

Beim Klicken auf einen Teilwürfel wird im EventHandler folgender Ablauf ausgeführt:

1. Anhand einer Uniform-Variable wird dem Vertexshader mitgeteilt, dass er im Fragmentshader die Alphawerte mit den Würfel-Indizes verwenden soll. Es handelt sich hierbei nur um eine temporäre Zeichnung, die für den nächsten Schritt wichtig ist.
2. Nun werden die Bildschirmkoordinaten vom ActionEvent abgefragt, dass durch den Linksklick ausgelöst wurde. Diese Koordinaten werden dann der Funktion „gl.readPixels()“ übergeben. Diese Funktion liefert ein Array zurück, dass die RGBA-Werte des geklickten Pixels enthält. Durch das Abprüfen des Alpha-Wertes kenn man nun den Index des geklickten Teilwürfels.
3. Die Information über den ausgewählten Teilwürfel wird beim nächsten Zeichnen an den Vertexshader übergeben, der diesen dann speziell behandelt und heller zeichnet. Der Teilwürfel ist nun ausgewählt und somit wird auch die Drehsteuerung freigegeben.

4.2 Drehsystem und Steuerung

Für die Rotationen gibt es ein grundlegendes Prinzip, nachdem sich die Steuerung richtet: Ein ausgewählter Teilwürfel soll sich immer um alle drei Achsen drehen können. Daher ist es wichtig vor der Rotation eine Drehachse festzulegen. Diese lässt sich mit den Pfeiltasten der Tastatur (Hoch bzw. Runter) auswählen. Die eigentliche Drehung wird mit den Pfeiltasten nach links bzw. rechts ausgelöst. Dabei bewirkt die Pfeiltaste nach rechts eine Drehung um 90° im Uhrzeigersinn um die ausgewählte Achse und die Pfeiltaste nach links führt zu einer Drehung gegen den Uhrzeigersinn (s. Abbildung 3).

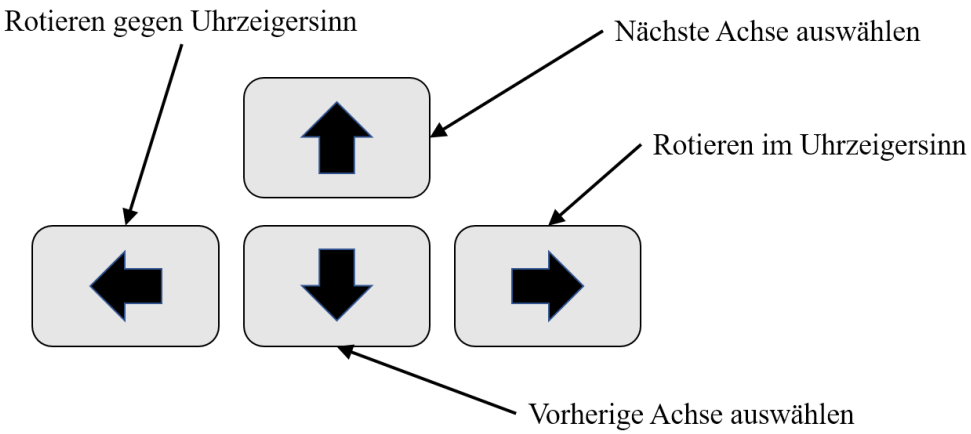

Abbildung 3: Dreh- und Achsensteuerung

Für das Ausführen einer Drehung besitzt die SubCube-Klasse eine extra Funktion namens „rotateSubCube()“. Diese Funktion erhält bei Aufruf zwei Parameter. Zum einen muss die Funktion wissen, um welche der drei ausgewählten Achsen der Teilwürfel gedreht werden soll und zum anderen wird die Richtung benötigt, da man im bzw. gegen den Uhrzeigersinn drehen kann. Je nach Interaktion durch Tastatur werden ActionEvents ausgelöst. Im entsprechenden EventListener wird dann geprüft, welche Tasten gedrückt wurden. Hierdurch wird dann sowohl Drehachse als auch Drehrichtung festgelegt und die Seiten-Drehfunktion ausgeführt.

4.3 Rotation der Seiten

Durch eine Seitenrotation müssen alle neun betroffenen Teilwürfel gedreht werden. Welche Würfel betroffen sind, lässt sich mit Hilfe von festen Position-IDs bestimmen. Jeder Teilwürfel besitzt dieses Attribut. Sollen nun beispielsweise die rechten neun Teilwürfel gedreht werden, muss man einen der entsprechenden Seitenwürfel und die X-Achse auswählen. Dadurch werden alle Teilwürfel gedreht, die einen der rechten Positions-IDs besitzen.

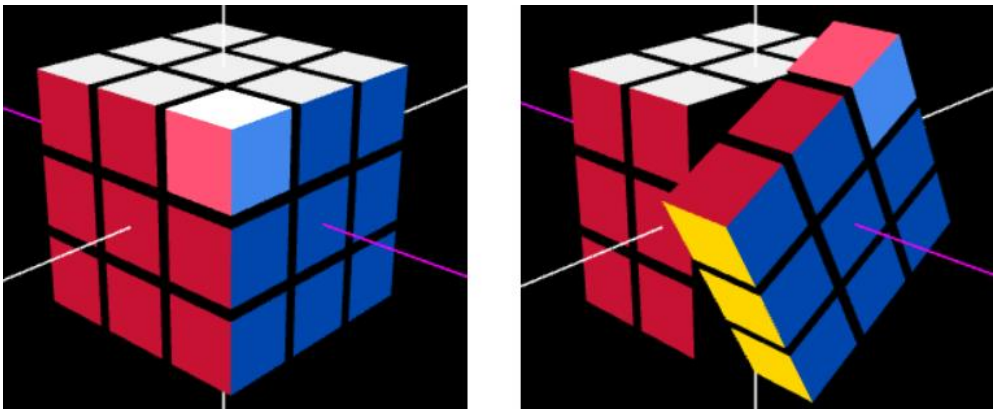

Abbildung 4: Auswahl und Drehung

Die Teilwürfel haben nun ihre Position visuell verändert. An dieser Stelle stößt man aber auf eine Problematik, wenn man eine andere Seite drehen möchte. Es entsteht ein Chaos, da sich nicht mehr die richtigen Seitenwürfel drehen werden. Dieses Problem entsteht, da die Teilwürfel mit Hilfe einer Schleife überprüft werden und anhand des 3x3x3 SubCube-Arrays dann gedreht werden. Das SubCube Array darf somit nach einer Drehung nicht unverändert bleiben.

Die Positionen müssen sich ebenfalls im Array entsprechend verändern. Für das Sortieren des Arrays ist eine Hilfsfunktion notwendig, die alle Teilwürfel-Attribute der alten Position in die neue kopiert. Diese Funktion ist notwendig, da die SubCube-Klasse über selbstverwalteten Speicher verfügt. Damit das erste Element beim Kopieren nicht verloren geht, muss dieses in eine Hilfsvariable auslagert und am Ende zurücksichert werden (s. Code 6). Das zentrale Element behält seine Position. Die nachfolgende Abbildung soll den Vorgang für das Neuanordnen des Arrays nach einer Rechtsdrehung veranschaulichen.

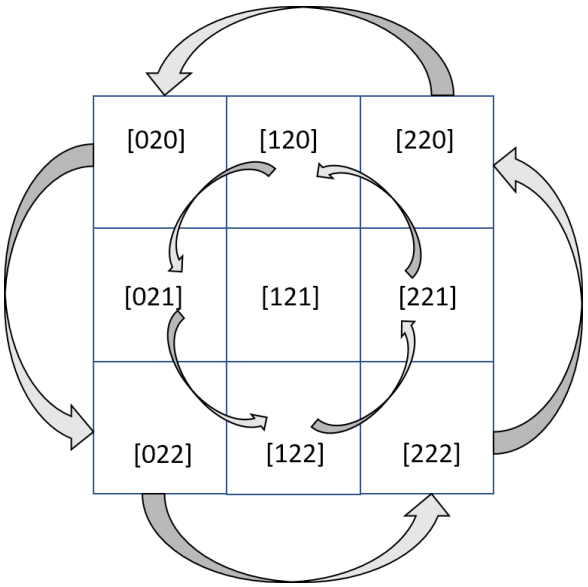

Abbildung 5: Neuanordnung des Arrays

```
subCubeHelp.copyValues(subCubes[0][2][2]);
subCubes[0][2][2].copyValues(subCubes[0][2][0]);
subCubes[0][2][0].copyValues(subCubes[2][2][0]);
subCubes[2][2][0].copyValues(subCubes[2][2][2]);
subCubes[2][2][2].copyValues(subCubeHelp);

subCubeHelp.copyValues(subCubes[0][2][1]);
subCubes[0][2][1].copyValues(subCubes[1][2][0]);
subCubes[1][2][0].copyValues(subCubes[2][2][1]);
subCubes[2][2][1].copyValues(subCubes[1][2][2]);
subCubes[1][2][2].copyValues(subCubeHelp);
```

Code 7: Neuanordnung des Arrays

4.4 Animation

Die Animation der Seitendrehungen wird realisiert, indem die Drehung in zehn Teildrehungen zerlegt wird. Eine vollständige Seitendrehung beträgt dabei 90°. Umgerechnet entspricht das dem Bogenmaß: $\frac{\pi}{4}$. Jede Teildrehung beträgt dann dem Bogenmaß $\frac{\frac{\pi}{4}}{10}$. Beim zehnmaligen Ausführen einer Rotation mit diesem Wert, entsteht die visuell sichtbare Dreh-Animation.

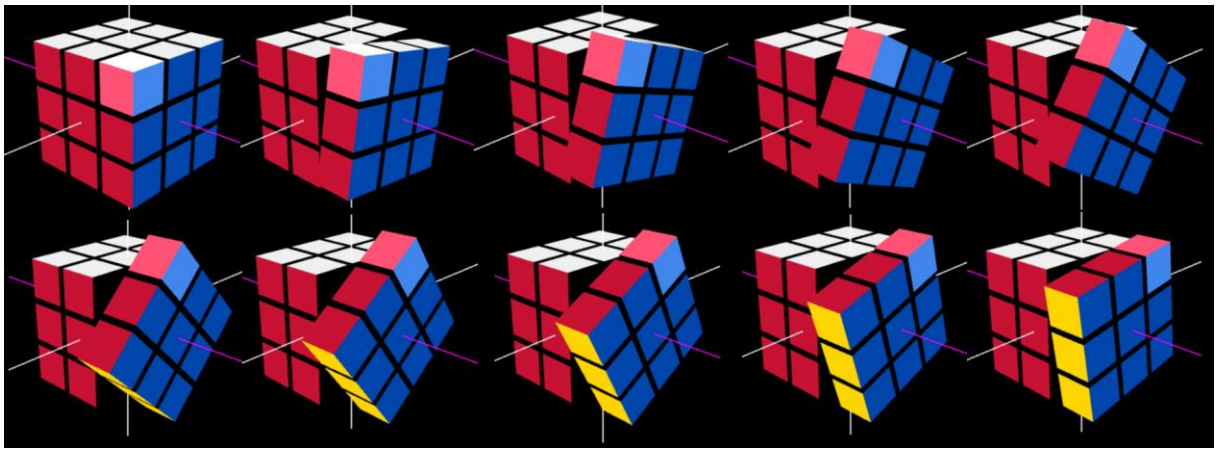

Abbildung 6: Animation der Seitenrotation

5 3D-Umgebung

Die Umgebung wird mit einer sogenannten *Cubemap* realisiert. Hierzu ist ein neuer Vertex- und Fragmentshader notwendig, da an dieser Stelle 2D -Texturkoordinaten verwendet werden. Die Implementierung der Cubemap wird im nachfolgenden Kapitel erläutert.

5.1 Cube-Mapping

Die Cubemap besteht aus sechs separaten Bildern, die zusammengesetzt wie ein aufgeklappter Würfel aussehen (s. Abbildung 8). Legt man diese Texturen nun passgenau um einen Würfel, werden die Übergange zwischen den einzelnen Texturen fließend. Befindet sich die Kameraposition innerhalb des Texturwürfels, wirken die Texturen wie ein dreidimensionales Umfeld.

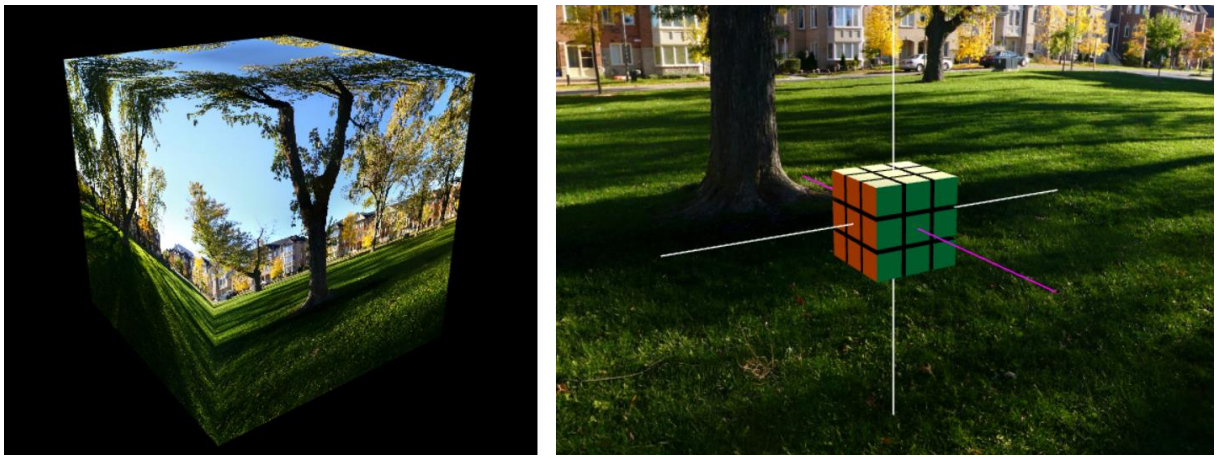

Abbildung 7: Cubemap

Um dieses Konzept in WebGL umsetzen zu können verwendet man die Funktion gl.texImage2D(…). Diese muss für jede der sechs Texturen einzeln aufgerufen werden, um die Texturen mittels spezifischen Parametern an den jeweiligen Cubemap-Seiten binden zu können.

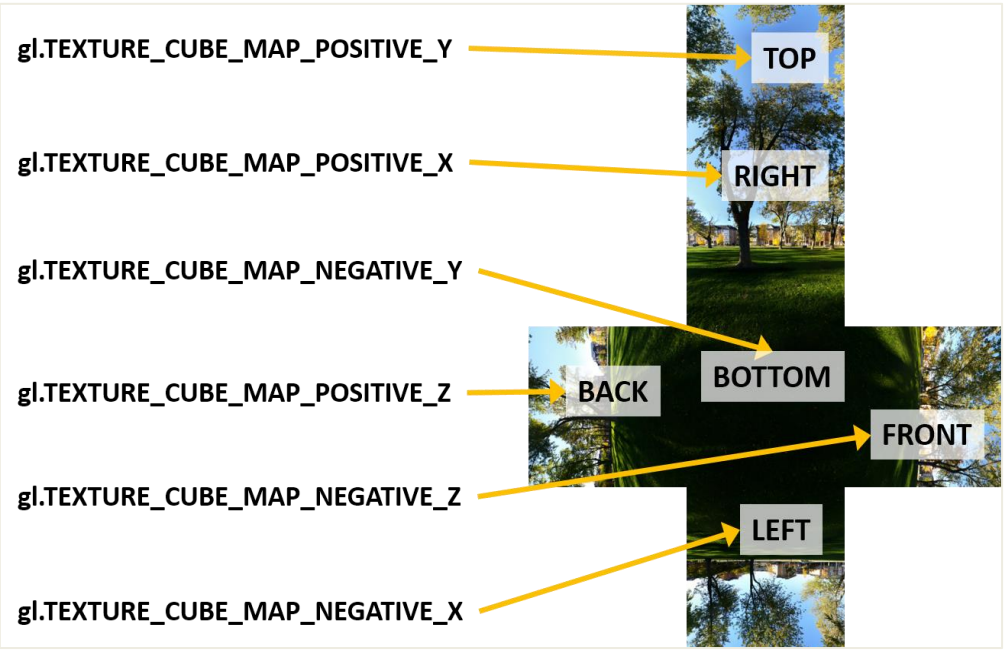

Abbildung 8: Parameter für Texturzuordnung

5.2 Vertex- und Fragmentshader

Für die Cubemap müssen eigene Shader angelegt werden. Diese arbeiten ausschließlich mit Texturkoordinaten und der View-Projection-Matrix, damit sich die Kameraperspektive an der richtigen Position innerhalb der Cubemap befindet.

```
in vec3 vertPosition;
out vec3 vertTexCoord;
uniform mat4 mWorld;

void main()
{
     vertTexCoord = vertPosition;
     gl_Position = mWorld * vec4(vertPosition, 1.0);
}
```

Code 8: Vertexshader für Cubemap

Der dazugehörige Fragmentshader der Cubemap sieht folgendermaßen aus.

```
precision mediump float;

in vec3 vertTexCoord;
out vec4 fragColor;
uniform samplerCube cubemap;

void main()
{
     fragColor = texture(cubemap, vertTexCoord);
}
```

Code 9: Fragmentshader für Cubemap

6 Beleuchtung

Als Beleuchtung wird eine einfache Punkbelichtung implementiert. Für diesen Beleuchtungseffekt müssen zusätzliche Veränderungen am Vertex- und Fragmentshader der Teilwürfel vorgenommen werden. Für die Berechnung des ankommenden Lichtanteils auf den Texturen werden nun zusätzlich eine Model- und eine Normalen-Matrix benötigt.

Außerdem benötigen die Shader zusätzliche Informationen, wie den Lichtanteil der Umgebung, oder die Intensität der Lichtquelle und deren Position. Die Teilwürfel erhalten ebenfalls ein weiteres Attribut, ihren Normalenvektor.

6.1 Lichtverhältnisse und Position

Wie bereits initial beschrieben, werden einige neue Variablen benötigt. Für die Berechnung des resultierenden Lichtanteils auf den Texturen des Würfels sind drei essentielle Informationen notwendig:

- Das Licht der Umgebung wird mit der Variable „AmbientLight“ initialisiert. Das Umgebungslicht beträgt hier: [0.6, 0.6, 0.6].
- Das gestreute Licht der Beleuchtungsquelle wird mit der Variable „DiffuseLight“ initialisiert. Das diffuse Licht beträgt hier : [1.5, 1.5, 0.7] und wirkt somit etwas gelblich.

- Die Position der Lichtquelle wird mit der Variable „LightPosition“ initialisiert. Die Position der Lichtquelle wird im 45° Winkel zu einem der oberen Ecken des Würfels bei [2.0, 2.0, 2.0] platziert.

Diese drei Variablen werden für die weitere Berechnung als Uniform-Variablen an den Fragmentshader übergeben.

6.2 Berechnung des Lichts

Das resultierende Licht auf dem Würfel wird mit dem „diffusen“ und „ambienten“ Lichtanteils im Fragmentshader berechnet. Zuerst werden ambientes und diffuses Licht einzeln für jeden Farbwert, auf dem das jeweilige Licht auftrifft, ermittelt und anschließend addiert.

Die folgenden Gleichungen zeigen die Berechnung für die beiden Lichtanteile. Für das diffuse Licht wird zusätzlich das Skalarprodukt aus Lichtposition und Normalenvektors des Teilwürfels benötigt. Diese Variable wird hier als „nDotL“ bezeichnet und entscheidet über den Anteil des ankommenden Lichts, abhängig von der Würfel- und Belichtungsposition.

$$nDotL = dot(uLightPosition, cubeNormal)$$
$$ambientLight = uAmbientLight \cdot fragColor.rgb$$
$$diffuseLight = uDiffuseLight \cdot fragColor.rgb \cdot nDotL$$

Durch Addition der Variablen „$ambientLight$" und „$diffuseLight$" erhält man die sichtbare Beleuchtung auf dem Würfel aus der Richtung der definierten Lichtposition.

$$Light = ambientLight + diffuseLight$$

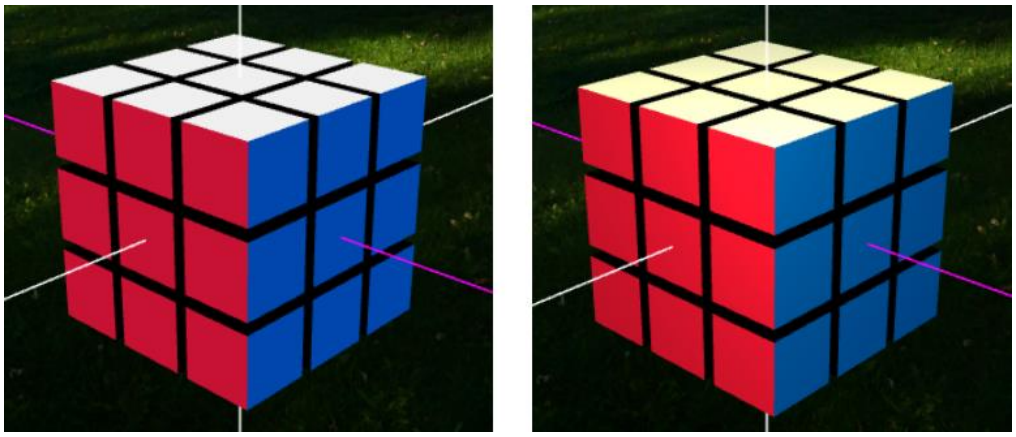

Abbildung 9: Punktbeleuchtung

7 Zusammenfassung

Das Wahlpflichtfach Computergrafik setzt sich aus einem Theorie- und einem Praxisteil zusammen. In der Theorie werden viele verschiedene Methoden aus der Computergrafik erläutert, während diese in einem Projekt praktisch umgesetzt werden sollen. Nach einigen Vorüberlegungen, fiel meine Entscheidung auf einen Rubik's Cube. Dieser Würfel sollte am Ende von allen Seiten betrachtbar sein und in einer 3D-Umgebung schweben. Außerdem soll sich jeder Teilwürfel bzw. jede Seite um 90° um eine ausgewählte Achse drehen können. Das Einführen einer sinnvollen Steuerung war ebenfalls ein Teil der festgelegten Aufgaben.

Die Programmierung des Rubik's Cube wurde mit der Programmiersprache JavaScript und der Grafikschnittstelle WebGL realisiert. Zu Beginn wurde ein Konzept aufgestellt, wie der Würfel im dreidimensionalen gezeichnet werden soll und welche Eigenschaften die Klasse der Teilwürfel besitzen muss. Danach wurde eine Kamerasteuerung implementiert, die es möglich gemacht hat den Würfel von allen Seiten zu betrachten. Darauffolgend wurde die Systematik für das Rotieren der Seiten aufgestellt, bei der man zuerst einen Teilwürfel und dann eine Drehachse festlegen muss. Zum Schluss wurde die 3D-Umgebung mit Hilfe einer Cubemap und ein Beleuchtungseffekt mit einer Punktbeleuchtung realisiert. Nach dem erfolgreichen Einbringen der Beleuchtung, wurden alle Teilaufgaben erfüllt und somit gilt auch das Projekt als abgeschlossen. Das Ergebnis lässt sich nun mit jedem Browser ansehen, der WebGL unterstützt.

Nichtsdestotrotz könnte man die Arbeit an verschiedenen Stellen fortsetzen, indem man eine Seitendrehung mit der Maus ermöglicht, oder verschiedene Texturen über die Teilwürfel legt, um dem gesamten Würfel ein neues Aussehen zu verleihen. Auch eine Auswahl zwischen verschiedenen Rubik's Cube Größen wäre denkbar, bei der man zum Beispiel zwischen einem 3x3x3 und einem 4x4x4 Würfel wählen kann.

Literaturverzeichnis

[Parisi 2014] Parisi, T.: Programming 3D Applications with HTML5 and WebGL, 1. Aufl., O'Reilly, Beijing, 2014

Eigenständigkeitserklärung

Ich versichere hiermit, dass ich meinen Praxisbericht mit dem Titel

---

Programmierung eines Rubik's Cube in JavaScript unter Verwendung der Programmierschnittstelle WebGL

---

selbständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie nicht an anderer Stelle als Prüfungsarbeit vorgelegt habe.

Ort

Datum

Unterschrift