Computergrafik WS 21/22
Praktikum Gruppe A
Daniel Sandbühler
OpenGL-Miniprojekt

Dokumentation

In dieser Dokumentation werden im Nachfolgenden die einzelnen Bestandteile des Projekts betrachtet. Es soll deren Aufbau und Zusammenhang deutlich gemacht werden.

GL.1

b) Umbenennung der Dateien

Nach Änderung der Dateinamen, wurde auch in der HTML-Datei in Zeile 220 folgende Änderung vorgenommen:

```
<script type="text/javascript" src="G19_A_1242267.js"></script>
```

c) Endlosschleife der Animation

Durch das Entfernen der Kommentarzeichen vor

```
requestAnimationFrame(render);
```

in Zeile 940, wird die Animation der Szene gestartet.

d) Starten und Stoppen der Animation auf Knopfdruck

Realisiert durch das Setzen der Flag `is_animated` (ein Boolean), die den aktuellen Status der Animation speichert. Durch das Drücken des Buttons wird diese entweder auf `true` oder `false` gesetzt.
Innerhalb der render function greift diese dann bei Zeile 903:

```
if(is_animated) {
     theta[axis] += 2.0;
} else {
     theta[axis] += 0.0;
}
```

Dort wird abgefragt, ob eine Animation durchgeführt werden soll. Ist dies der Fall, wird der Winkel der Rotation pro Frame auf 2 gesetzt. Anderfalls wird der Winkel auf 0 gesetzt.

f) ZUSATZ FPS Counter

Für den FPS Counter wird für jeden Frame die aktuelle Zeit in der Variable now gespeichert und auf Sekunden berechnet (Division durch 1000). Danach wird counter hochgezählt und die verstrichene Zeit zwischen dem letzen Frame und diesem Frame berechnet und in elapsedTime gespeichert.
In einer if-Abfrage wird überprüft, ob die vergangene Zeit bereits eine Sekunde überschritten hat. Dies stellt sicher, dass der FPS Counter nur jede volle Sekunde geupdatet wird. Ist dies der Fall, wird die fps Wiedergabevariable auf den Wert von counter gesetzt. FpsDegreeCounter spielt für die Darstellung der FPS keine Rolle und kommt erst später zum Einsatz. Danach wird die vergangene Zeit wieder heruntergesetzt und die Werte aus fps zum Anzeigen an das HTML-Dokument übergeben.
Ganz zum Schluss wird then noch einmal neu gesetzte, um im nächsten Frame mit genaueren Werten arbeiten zu können.

```
// FPS berechnen
var now = Date.now()/1000;
counter++;
elapsedTime += (now - then);
if(elapsedTime > 1){
           fps = counter;
           fpsDegreeCounter = counter;
           counter = 0;
           elapsedTime -= 1;
           document.getElementById("fpsCounter").innerHTML = fps;
}

then = Date.now()/1000;
```

In der HTML-Datei wird er wie folgt realisiert:

```
<label id="fps">FPS</label>
<span id="fpsCounter"></span>
```

GL.2

a) Verschiebung und Drehung des kleinen Würfels

Der ursprüngliche Würfel kann durch Einsatz folgender Code-Zeilen in Zeile 656 verschoben werden:

```
model = mult(model, translate(5.0, 0 ,1.0));
```

Dadurch wird eine Verschiebung um 5 auf der z- und um 1 auf der x-Achse realisiert.

Die Rotation um die eigene z-Achse wird mit Hilfe des FPS Counters realisiert. Dieser speichert den aktuellen FPS Wert in der Variablen `fpsDegreeCounter`.

Dieser wird wie folgt ab Zeile 933 verrechnet:

```
if(fpsDegreeCounter > 30) {
     zDegree += (36 / fpsDegreeCounter)
};
if(zDegree > 360) zDegree = 0.3;
```

Hier wird überprüft, ob der Degreecounter bereits weit genug hochgezählt hat. Ist dies nicht der Fall, wird die Rotation pro Frame standardmäßig auf 0.3 gesetzt. Sollte der Zähler aber hoch genug sein, wird die Variable zDegree für die Rotation berechnet.
Die Überlegung war folgende: Da sich der Würfel in 10 Sekunden eine komplette Rotation um seine eigene x-Achse vollziehen soll (360°), muss er folglich in einer Sekunde um 36° rotieren. Um dies auf einen Frame herunterzubrechen, werden diese 36° noch einmal durch die aktuelle FPS Zahl geteilt. Dieser Wert wird dann für die Rotation an den Würfel übergeben und mit folgendem Aufruf realisiert:

```
model = mult(model, rotate(zDegree, [0, 0, 1] ));
```

b) Zweiten Würfel einfügen, skalieren, verschieben, umfärben und um die eigene Achse rotieren

Um einen zweiten Würfel einzufügen und um die Übersicht zu behalten, werden alle Objekte in der Szene in eigene Funktionen gepackt. Für den zweiten Würfel wurde die Funktion `cube2` in Zeile 668 erstellt, die sämtliche Angaben zum Zeichnen des Würfels enthält. Dort wird in Zeile 685 `drawCube()` aufgerufen, um einen generischen Würfel zu erstellen. Dieser wird durch folgende Code-Zeilen den Angaben entsprechend verändert:

Skalierung um den Faktor 2
```
model = mult(model, scalem(2.0, 2.0, 2.0));
```

Rotation um die x-Achse. Selbes Prinzip wie beim ersten Würfel, dieses Mal wird der Grad Rotation verdoppelt (rotiert doppelt so schnell).
```
model = mult(model, rotate(2*zDegree, [1, 0, 0] ));
```

Zuletzt wird der Würfel an die richtige Position verschoben.
```
model = mult(model, translate(5.0, 0 ,-3.0));
```

Im Code sind diese im Sinne der Matrixmultiplikation in entgegengesetzter Reihenfolge angeordnet.

Farblich wird der Würfel in Zeile 681 verändert, wo die Variable `color` gesetzt wird und der Hilfsfunktion `lightingFunc` übergeben, die die Lichtberechnungen in eine einzelne Funktion auslagert.

```
var color = vec4(0.0, 1.0, 0.0, 1.0);
lightingFunc(true, color);
```

Gemäß der Vorgaben wird der Würfel über die RGBA-Werte 0, 1, 0, 1 auf grün gesetzt, ohne Transparenz.

c) Erste Pyramide zeichnen

Um die Pyramide zu zeichnen, wurde sich am Aufbau des Würfels orientiert. Analog dazu wurde in Zeile 357 die Funktion `drawPyramid()` erstellt. Darin wurden die fünf Vertices und die Farben dieser Vertices bestimmt.
Anschließend wurde die Hilfsfunktion `tris()` genutzt. Diese wurde zuvor in Zeile 252 erstellt und übergibt, wie die gegebene `quad()`, die Vertices von Dreiecken an die `vertices`, `normal` und `colors` Arrays. Anders als `quad()`, wird dieser Prozess aber nur noch für ein Dreieck durchgeführt. Da die Seiten der Pyramide nur aus einzelnen Dreiecken bestehen, konnte die `quad()` Funktion hierfür nicht genutzt werden. Sie kommt nur beim Boden der Pyramide zum Einsatz.

Damit kann in 732 in der Funktion `pyramide1` mit dem Aufruf von `drawPyramid()` eine Pyramide gezeichnet werden. Die Angaben für die Farbe wurde vom ersten Würfel übernommen. Auch die Transformationsmatrizen für die globale Drehung per Buttons wurde aus dem ersten Würfel übernommen.

d) Zweite Pyramide zeichnen, drehen, verschieben und umfärben

Die Funktion pyramide2 für die zweite Pyramide orientiert sich an `pyramide1`. Der Farbwert wird zu RGBA 1, 0, 0, 1 für eine rote, nicht durchsichtige Pyramide geändert. Danach wird die Lage über folgende Aufrufe verändert:

> Zuerst eine Drehung um 180° um die z-Achse, damit die Spitze nach unten zeigt.
> `model = mult(model, rotate(180, [0, 0, 1] ));`
>
> Dann eine Verschiebung um 8 auf der y-Achse.
> `model = mult(model, translate(0, 8 ,0));`

Im Code sind diese im Sinne der Matrixmultiplikation in entgegengesetzter Reihenfolge angeordnet.

Für die globale Drehung über die Buttons, wurden die Rotationsmatrizen aus der ersten Pyramide übernommen.

e) Dritte Pyramide zeichnen, drehen, verschieben und umfärben

Die dritte Pyramide wird analog zu den ersten beiden erstellt und in der Funktion pyramide3 realisiert. Die Farbe wird über den RGBA-Wert 0, 0, 1, 1 festgelegt.

Dann wird die Pyramide zuerst skaliert, das heißt konkret auf 40% ihrer ursprünglichen Größe verkleinert. Dies wird bewerkstelligt durch:

```
model = mult(model, scalem(0.4,0.4,0.4));
```

Danach muss die Pyramide so rotiert werden, dass ihr Winkel mit dem der Fläche der oberen Pyramide übereinstimmt. Dazu muss der Schnittwinkel zweier Vektoren berechnet werden. Einmal der Vektor, der der die y-Achse entspricht und einmal der Vektor vom Mittelpunkt der kurzen Kante der Dreiecksfläche, wo die kleine Pyramide später anliegen soll, zur Spitze der umgedrehten Pyramide.

$\vec{u} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$ *Vektor , der der y − Achse entspricht*

$\vec{v} = \begin{pmatrix} 2 \\ 8 \\ 1 \end{pmatrix} - \begin{pmatrix} 2 \\ 4 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 4 \\ 1 \end{pmatrix}$ *Vektor auf der Dreiecksfläche*

Von diesen beiden Vektoren muss zuerst das Skalarprodukt ermittelt werden.

$0 \cdot 0 + 1 \cdot 4 + 0 \cdot 1 = 4$

Dann werden die Beträge der Vektoren ermittelt.

$|\vec{u}| = \sqrt{0^2 + 1^2 + 0^2} = 1$
$|\vec{v}| = \sqrt{0^2 + 4^2 + 1^2} = 4{,}1231$

Abschließend wird alles in die Formel für die Winkelberechnung eingefügt.

$\frac{4}{4{,}1231} = 0{,}97014$
$\cos^{-1}(0{,}97014) = 14{,}037\,°$
$90\,° + 104{,}037\,° \approx 104\,°$

Die Pyramide muss um 104° um die x-Achse gedreht werden.

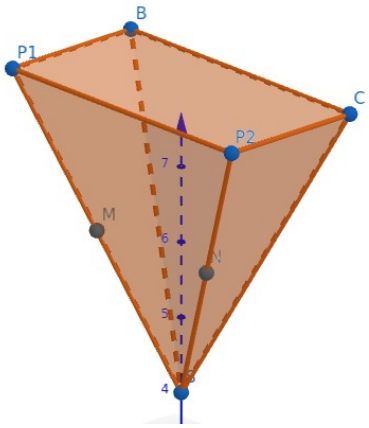
```
model = mult(model, rotate(104, [1, 0, 0] ));
```

Für die Translation muss der Mittelpunkt der Dreiecksfläche berechnet werden. Dazu werden die drei Punkte benötigt, die die Fläche eingrenzen.

$P1(-2,8,1)$ *und* $P2(2,8,1)$ *für die beiden Punkte an der kurzen Kante .*
$S(0,4,0)$ *für den Punkt an der Spitze .*

Um die jeweilige Mitte der beiden langen Kanten zu ermitteln, muss die Summe der betreffenden Punkte halbiert werden.

Mittelpunkt der Strecke $P1 - S: M = \frac{\begin{pmatrix} -2 \\ 8 \\ 1 \end{pmatrix} + \begin{pmatrix} 0 \\ 4 \\ 0 \end{pmatrix}}{2} = \begin{pmatrix} -1 \\ 6 \\ 0{,}5 \end{pmatrix}$

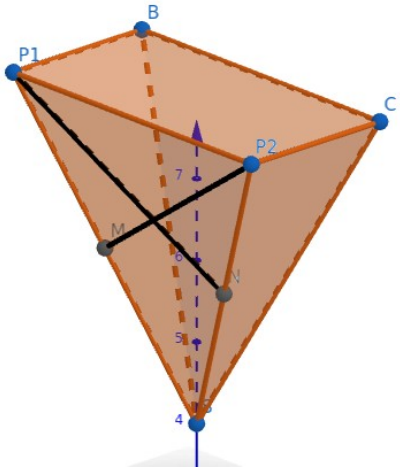


Mittelpunkt der Strecke $P2 - S: N = \frac{\begin{pmatrix} 2 \\ 8 \\ 1 \end{pmatrix} + \begin{pmatrix} 0 \\ 4 \\ 0 \end{pmatrix}}{2} = \begin{pmatrix} 1 \\ 6 \\ 0{,}5 \end{pmatrix}$

Um den Mittelpunkt der Fläche zu bestimmen, muss zuerst von jedem Kantenmittelpunkt der Vektor zu den jeweils gegenüberliegenden Eckpunkten gefunden werden. Zum Schluss ergibt sich der Zentralpunkt aus den Schnittpunkten dieser Vektoren.

$$\vec{t} = \begin{pmatrix} 2 \\ 8 \\ 1 \end{pmatrix} - \begin{pmatrix} -1 \\ 6 \\ 0{,}5 \end{pmatrix} = \begin{pmatrix} 3 \\ 2 \\ 0{,}5 \end{pmatrix}$$

$$\vec{c} = \begin{pmatrix} -2 \\ 8 \\ 1 \end{pmatrix} - \begin{pmatrix} 1 \\ 6 \\ 0{,}5 \end{pmatrix} = \begin{pmatrix} -3 \\ 2 \\ 0{,}5 \end{pmatrix}$$

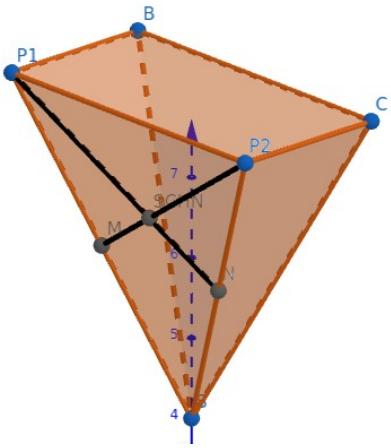


$$Gerade\, g1 = \begin{pmatrix} -1 \\ 6 \\ 0{,}5 \end{pmatrix} + s \cdot \begin{pmatrix} 3 \\ 2 \\ 0{,}5 \end{pmatrix}$$

$$Gerade\, g2 = \begin{pmatrix} 1 \\ 6 \\ 0{,}5 \end{pmatrix} + p \cdot \begin{pmatrix} -3 \\ 2 \\ 0{,}5 \end{pmatrix}$$

Mit diesen beiden Geraden kann man durch Gleichsetzen jetzt einen Schnittpunkt errechnen.

$$Schnittpunkt = \begin{pmatrix} 0 \\ 6{,}667 \\ 0{,}667 \end{pmatrix}$$



Damit lässt sich jetzt über

```
model = mult(model, translate(0, 6.667, 0.667));
```

die Pyramide an die korrekte Position verschieben.

f) Drehen aller Pyramide mittels Buttons

Da für jede Pyramide die Drehfunktionen aus dem ersten Würfel übernommen wurden, lassen diese sich auch simultan über die Buttons drehen.

GL.3

a) Kameraeinstellungen x-Achse

In `setCamera()` werden für `camIndex` 1 folgende Werte gesetzt:

Der Augpunkt wird auf der x-Achse an die Position 10 verschoben.
```
eye = vec3(10.0,0.0,0.0);
```

Der View Reference Point befindet sich in Weltkoordinaten bei 0 0 0, die Kamera blickt also vom Augpunkt aus entlang der negativen x-Achse.
```
vrp = vec3(0.0,0.0,0.0);
```

Der Upvektor zeigt in Richtung positiver y-Achse.
```
upv = vec3(0.0,1.0,0.0);
```

b) Kameraeinstellungen für y-Achse und z-Achse

Ähnlich wie bei a) werden für `camIndex` 2 und `camIndex` 3 die Werte für den Augpunkt, View Reference Point und den Upvektor so geändert, dass die Kamera auf den entsprechenden Achsen liegt und in die jeweilige negative Achsenrichtung auf den Nullpunkt schaut.

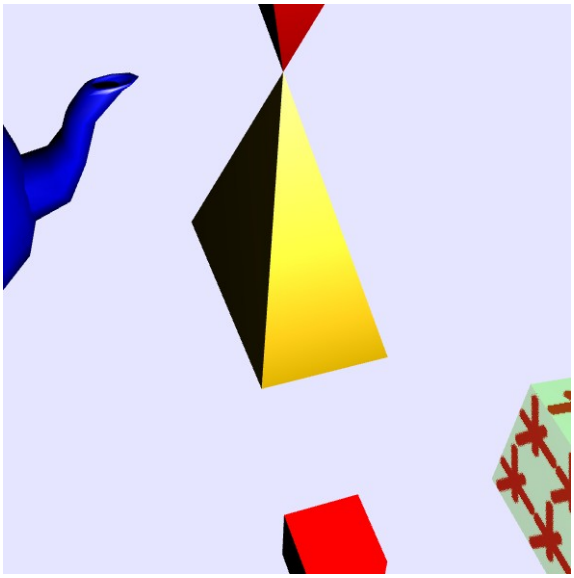
c) Kameraeinstellungen für Pyramidenspitze

Anders als bei den vorherigen Kameraeinstellungen, wurde hier auch der View Reference Point geändert. Dieser liegt jetzt bei 0 4 0, wurde also von der Standardeinstellung um 4 auf der y-Achse nach oben verschoben. Entsprechend muss auch der Augpunkt um 4 nach oben verschoben werden. Es ergeben sich dabei folgende Werte:

```
eye = vec3(10.0,4.0,0.0);
vrp = vec3(0.0,4.0,0.0);
upv = vec3(0.0,1.0,0.0);
```

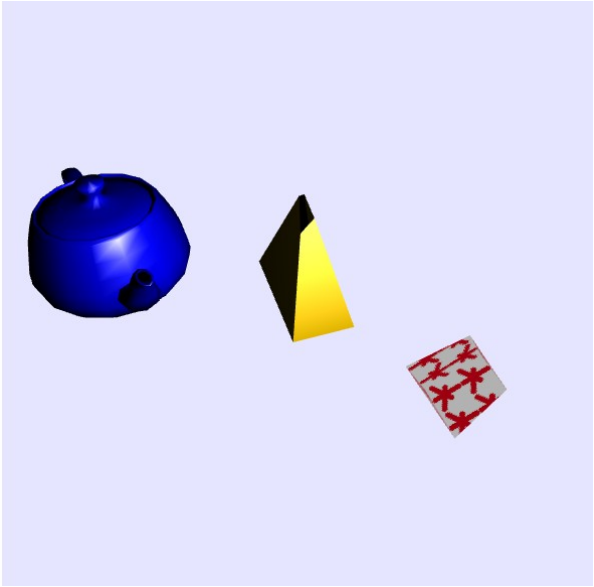
d) Öffnungswinkel der Kamera von 30°

Der Öffnungswinkel kann geändert werden, indem in Zeile 510 der erste Parameter von perspective() geändert wird.
Eine Veränderung auf 30° bewirkt eine vergrößerte Darstellung der Szene, wobei dabei einige Teile nicht mehr sichtbar sind.



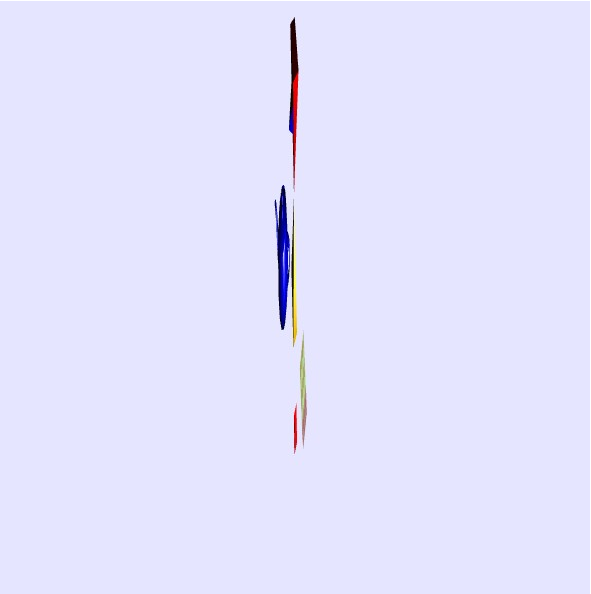
e) Near-Clipping-Plane auf 15

Durch das Verändern des Wertes der Near-Clipping-Plane wird das Bild erst ab dieser Entfernung von der Kamera dargestellt (Beginn des Sichtvolumens).



f) Aspect Ratio auf 16:9

Das Bild wird gestaucht und das Seitenverhältnis nicht geändert, da in der HTML-Datei das Seitenverhältnis für `canvas` mit 600x600 fest vorgegeben wird. Da nur diese Fläche zur Verfügung steht, wird beim Einfügen eines deutlich breiteren Bildes die Darstellung gestaucht.



GL.4

a) Beleuchtung für kleineren Würfel ausschalten und Umfärbung der Seiten

Die Beleuchtung für den Würfel wird durch das setzen der Beleuchtungs-Flag innhalb von displayScene() ausgeschaltet. Dort wird die Variable lighting auf false gesetzt.
Da für den Würfel jetzt keine Beleuchtungsrechnung mehr durchgeführt wird, werden die Farben der einzelnen Vertices herangezogen, um die Flächen anzuzeigen.

b) Würfel rot färben mit zwei gegenüberliegenden schwarzen Seiten

Um den Würfel komplett rot zu färben, mit zwei gegenüberliegenden schwarzen Flächen, wurden in drawCube() die Werte für colors wie folgt geändert:

```
colors = [
     vec4(1.0, 0.0, 0.0, 1.0), //rot
     vec4(0.0, 0.0, 0.0, 1.0), //schwarz
     vec4(1.0, 0.0, 0.0, 1.0), //rot
     vec4(1.0, 0.0, 0.0, 1.0), //rot
     vec4(0.0, 0.0, 0.0, 1.0), //schwarz
     vec4(1.0, 0.0, 0.0, 1.0), //rot
     vec4(1.0, 0.0, 0.0, 1.0), //rot
     vec4(1.0, 0.0, 0.0, 1.0) //rot
];
```

Die Übergänge zwischen den benachbarten Seiten sind schlecht zu erkennen, da keine Beleutungsrechnung für die einzelnen Seiten mehr durchgeführt wird. Das heißt, dass alle Seiten gleich gut sichtbar sind und es keine Kontrastunterschiede gibt, mit denen die Flächen zu differenzieren wären.

c und d) Einbau ambiente und spekulare Beleuchtung

Zur Realisierung von ambienten und spekularem Licht wird in der Javascript-Datei in calculateLights(), analog zur diffusen Beleuchtung, jeweils eine neue Variable angelegt.

```
var ambientDiffuse = vec4( amb, amb, amb, 1.0 );
var specularDiffuse = vec4( 1.0, 1.0, 1.0, 1.0 );
```

Für ambientDiffuse wurden die Werte schon mit einem Platzhalter ersetzt, der über einen Regler verändert werden kann.
Die Werte dieser Variablen werden mit der Lichtquelle verrechnet und in entsprechenden Variablen abgelegt.

```
var ambientProduct = mult(lightDiffuse, ambientDiffuse);
var specularProduct = mult(lightDiffuse, specularDiffuse);
```

Danach werden beide Werte an den Shader übergeben.

```
gl.uniform4fv(gl.getUniformLocation(program, "ambientProduct"),
flatten(ambientProduct) );

gl.uniform4fv(gl.getUniformLocation(program, "specularProduct"),
flatten(specularProduct) );
```

Im Shaderaufruf:

Auch hier wurden Variablen mit entsprechenden Namen angelegt.
Für die ambiente Beleuchtung wird die Variable in `fColor` direkt zu einem Wert für den Fragment-Shader verrechnet.

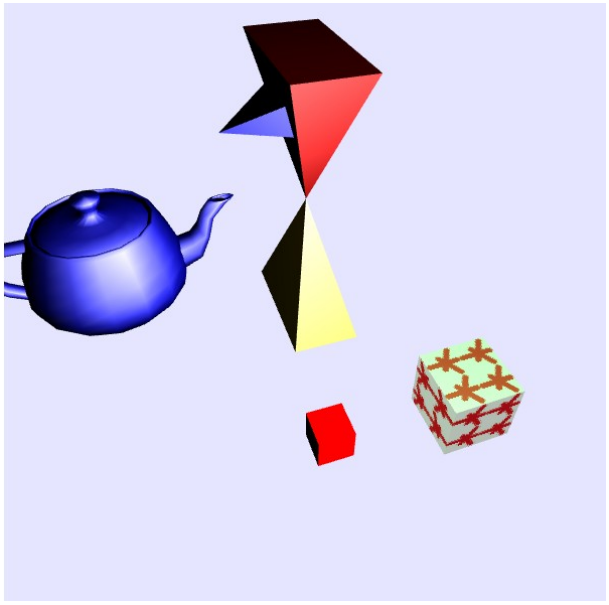
```
fColor = vec4(diffuse.xyz, 1.0) + vec4(spec.xyz, 1.0) +
vec4(ambientProduct.xyz, 1.0);
```

Für die spekulare Beleuchtung muss zuerst der Reflektionsvektor berechnet und normalisiert werden. Danach wird der Winkel zwischen diesem und der entsprechenden Punktnormale berechnet. Dieser Wert wird dann in der Variable `spec` zusammen mit dem Wert für `shininess` und dem `specularProduct` verrechnet. Auch dieser Wert findet sich dann in der Berechnung zu `fColor` wieder.

e) Wert der Shininess ändern

Shininess 5

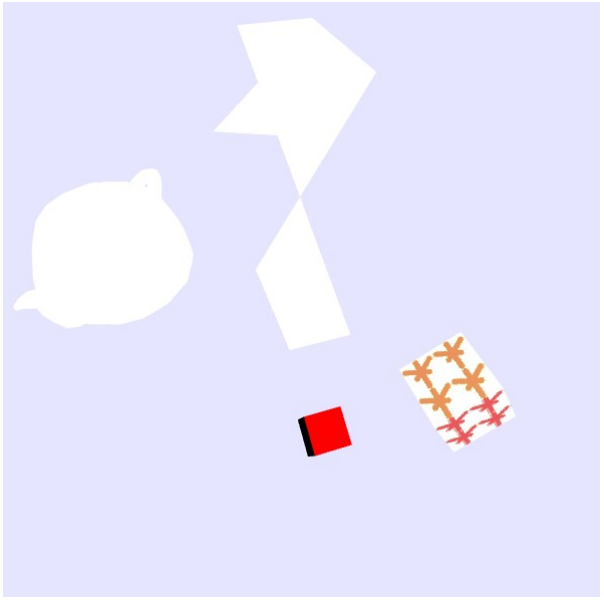
Je niedriger der Wert, desto glänzender die Objekte.



f) Wert ambientes Licht ändern

Ambient Intensity 1

Je näher der Wert an 1, desto stärker der Einfluss der ambienten Beleuchtung in der Berechnung der Farbwerte.



GL.5

a) Einfügen des Gifs in den Body der HTML-Datei

Eingefügt über

```
<img id="texImage" accept='image/*' src="hsrm.gif" hidden
alt="hsrm-logo / Würfeltextur"/>
```

b) Bild einlesen und Textur erstellen

Die Textur wird in der Javascript-Datei über die Hilfsmethode `handleTextureLoaded()` in Zeile 165 verarbeitet. Diese bekommt die Variable `image` übergeben, welche zuvor mit den Bilddaten aus dem HTML-Dokument initialisiert wurde.

```
var image = document.getElementById("texImage");
handleTextureLoaded( image );
```

Dort wird auch die MipMap erstellt.

```
gl.generateMipmap( gl.TEXTURE_2D );
```

Anschließend wird alles über die sampler2D-Variable `texture` an die GPU-Seite übergeben.

```
gl.uniform1i(gl.getUniformLocation(program, "texture"), 0);
```

c) Texturkoordinaten festlegen

Da nur ein Würfel texturiert werden soll, werden die Texturkoordinaten auch nur in drawCube() und quad() festgelegt. Zuerst wurden Variablen angelegt, die für die Texturierung benötigt werden.

Ein Array, in dem die Koordinaten abgelegt werden.
```
var texCoordsArray = [];
```

Eine Variable, in der die einzelnen Texturkoordinaten zuerst abgelegt werden.
```
var texCoord;
```

Eine Flag, die angibt, ob ein Objekt texturiert wird, oder nicht.
```
var isTextured = true;
```

In quad() werden die Texturkoordinaten analog zu den Daten der Vertices, Farben und Normalen, auf texCoordsArray gepusht. Dort wird auch vorher abgefragt, ob überhaupt texturiert werden soll. Bevor das passieren kann, müssen die Texturkoordinaten festgelegt werden. Die geschieht in drawCube(). Dort wird final auch ein Buffer festgelegt, mit den Daten aus texCoordsArray gefüttert und anschließend an die Variable vTexCoord im HTML-Dokument übergeben.

```
var tBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, tBuffer);
gl.bufferData(gl.ARRAY_BUFFER, flatten(texCoordsArray),
gl.STATIC_DRAW);

var vTexCoord = gl.getAttribLocation(program, "vTexCoord");
gl.vertexAttribPointer(vTexCoord, 2, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(vTexCoord);
```

d und e) Texturierung in den Shadern und Farbmischung

Auf Shaderseite übernimmt der Fragment-Shader die Texturierung. Nachdem alle Daten an die entsprechenden Variablen übergeben wurden, wird im Fragment-Shader zuerst die Abfrage gemacht, ob überhaupt texturiert wird. Die Flag, ob eine Texturierung durchgeführt werden soll, wird erst kurz vor dem Zeichnen des Objektes übergeben.

```
isTextured = true;
gl.uniform1i(gl.getUniformLocation(program, "isTextured"),
isTextured);
```

Weitere wichtige Variablen sind:

Im Javascript gefüttert durch die Funktion handleTextureLoaded()
`uniform sampler2D texture;`

Vertexkoordinaten für die Texturen
`vTexCoord`

Variable für die Texturkoordinaten vom Typ varying
`fTexCoord`

`fTexCoord` wird im Vertex-Shader mit `vTexCoord` initialisiert.
Falls eine Texturierung durchgeführt werden soll, wird im Fragment-Shader eine Variable `tColor` initialisiert mit den Daten aus den Variablen `texture` und `fTexCoord`. Anschließend wird `gl_FragColor` über die Funktion `mix()` mit den Textur- und Farbdaten gefüttert. Diese werden einem prozentualen Wert entsprechend gemischt.

GL.6

a) Einfügen der Teekanne

Für die Teekanne wurden die Inhalte aus teapot.js in das Javascript kopiert.
Ähnlich wie bei den anderen Objekten auch, wurde eine kapselnde Funktion erstellt, in welcher `drawTeapot()` aufgerufen wird. Dort werden auch die Transformationen durchgeführt.

Zuerst die Skalierung um den Faktor 0.3.
`model = mult(model, scalem(0.3, 0.3, 0.3));`

Dann die dauerhafte Rotation um die eigene y-Achse.

```
model = mult(model, rotate(zDegree, [0, 1, 0]));
```

Danach die Verschiebung um -5 auf der x-Achse und 6 auf der z-Achse.

```
model = mult(model, translate(-5,0,6));
```

Damit die Teekanne auch gezeichnet werden kann, ist es nötig, in der `init()` Funktion die erstellte Funktion `loadTeapot()` aufzurufen.

b) Cartoon-Shader

Der Cartoon-Effekt wird im Fragment-Shader realisert. Dort wird über eine if-Abfrage bestimmt, ob eine Farbveränderung für den Cartoon-Effekt durchgeführt werden soll. Wichtige Variablen sind:

Cartoon-Flag, die angibt, ob ein Cartoon-Shading zum Einsatz kommt

```
uniform bool cartoonOn;
```

Ein Zwischenspeicher für die jeweiligen Farbewerte.

```
vec4 fColor;
```

Die Cartoon-Intensität, zuvor abgeleitet durch die Variable Kd im Vertex-Shader

```
varying float cartoonIntensity;
```

Soll ein Cartoon-Shading durchgeführt werden, wird über eine Weitere Bedingungsabfrage die Farbe bestimmt. Ausschlaggebend sind hierbei die beiden Werte, die in `cartoonThresh_1` und `cartoonThresh_2` abgelegt werden. Durch Slider lassen sich diese dynamisch anpassen. Es werden jeweils zwei Farbwerte für das zuvor angelegte `fColor` gesetzt, je nachdem, ob der Wert der `cartoonIntensity` über oder oder unter einer der Schwellen liegt. Trifft nichts davon zu, wird auf einen default-Wert ausgewichen. Danach wird `fColor gl_FragColor` zugewiesen.

c) Button zum Ein- und Ausschalten des Cartoon-Shaders

Zuerst wurde in der HTML-Datei ein Button angelegt:

```
<div>
     <button id = "cartoonButton">Cartoon On/Off</button>
     <span id="cartoonBoolCheck"></span>
</div>
```

Dieser setzt in der Javascript-Datei die Variable `cartoonOn` auf true oder false, je nachdem, welchen Status sie vorher hatte.
Beim Zeichnen der Teekanne in `displayScene()` wird der Status von `cartoonOn` an die gleichnamige Variable im Fragment-Shader weitergereicht.

```
gl.uniform1i(gl.getUniformLocation(program, "cartoonOn"),
cartoonOn);
```

d) ZUSATZ Slider für Cartoon-Shading, ambientes Licht und Shininess

Die Slider für die unterschiedlichen Werte wurden alle sehr ähnlich angelegt und nach demselben Prinzip aufgebaut.
Zuerst wurden die Slider in der HTML-Datei angelegt. Hier beispielsweise der Slider für das ambiente Licht:

```
<div>
     <label for="ambientIntensityLabel">Ambient Intensity:</label>
     <input id="ambientSlider" type="range" min="0" max="1"
     step="0.1" value="0" />
     <span id="ambientInt"></span>
</div>
```

Danach wurden in der Javascript-Datei die entsprechend passenden Elemente gesetzt:

```
globalAmbient = document.getElementById("ambientSlider");

let updateAmbient = () =>
document.getElementById("ambientInt").innerHTML =
parseFloat(globalAmbient.value) ;

globalAmbient.addEventListener('input', updateAmbient);

updateAmbient();
```

Dabei wird mit jeder Veränderung auch gleichzeit die Anzeige für den aktuellen Wert im HTML-Dokument verändert.
Die Slider passen die Werte der entsprechenden Variablen an, die für Lichtintensität der ambienten Beleuchtung, der Shininess oder die Treshold-Werte des Cartoon-Shaders verantwortlich sind.

e) ZUSATZ Textur vervierfachen

Um die Textur auf dem Würfel vierfach erscheinen zu lassen, müssen in drawCube() die Werte der Texturkoordinaten, für die vorher eine 1 eingetragen war, auf den Wert 2 geändert werden.

```
texCoord = [
          vec2(0, 0),
          vec2(0, 2),
          vec2(2, 2),
          vec2(2, 0)
];
```

Damit wird angegeben, dass die Textur entlang der x- und y- Achse jeweils zwei mal angezeigt werden soll.

Quellen

https://www.lighthouse3d.com/tutorials/glsl-12-tutorial/toon-shader-version-ii/ Stand: 18.01.2022
https://webglfundamentals.org/webgl/lessons/webgl-3d-textures.html Stand: 18.01.2022
https://webglfundamentals.org/webgl/lessons/webgl-shaders-and-glsl.html Stand: 08.01.2022
https://developer.mozilla.org/de/docs/Web/API/WebGL_API/Tutorial/Getting_started_with_WebGL Stand: 28.12.2021