Visual Computing Praktikum - Shader

Basil Fierz, Matthias Büchi

March 6, 2024

Contents

Einleitung

In diesem Praktikum werden Sie verschiedene Shader implementieren. Für die verschiedenen Aufgaben erhalten Sie jeweils ein vordefiniertes Codegerüst. Wir arbeiten mit three.js. Das ist eine JavaScript Library, welche uns den Einsatz von WebGL vereinfacht. Somit können wir uns auf die Programmierung der eigentlichen Shader konzentrieren.

Hinweise

- Sie können die einzelnen Shader in beliebiger Reihenfolge implementieren. Diese bauen nicht aufeinander auf. Der Einfachheit halber sollten Sie mit dem Toonshader beginnen, da dieser sehr einfach zu implementieren ist und somit einen guten Einstieg bietet.
- Bevor Sie beginnen, sollten Sie den Anhang durchlesen. Dort sind die wichtigsten Funktionen zu three.js sowie einige theoretische Grundlagen. Mit diesen Informationen sollte es Ihnen leichter fallen die Aufgaben zu lösen.
- Um mit aktuelle Browsern zu arbeiten, muss ein Webserver verwendet werden. Das Augabenprojekt nutzt vite um die Abhängigkeiten zu verwalten und eine ausführbare Webanwendung zu präsentieren. Mit dem NPM-kommando 'npx vite' im Verzeichnis wird ein Webserver auf 'localhost' zur Verfügung gestellt.

1. Aufgabe - Toon Shader

In diesem Teil implementieren Sie einen einfachen Toon Shader. Dieser soll Ihnen einen einfachen Einstieg in die Shader-Programmierung geben.

Für diesen Teil arbeiten Sie in den folgenden Files:

- *toonshader.html*
- *toonshader.js*
- *shader/shadertoon.vert*
- *shader/toon.frag*

1.1. Vertex Shader

Transformieren Sie im Vertex Shader die Normale in den World Space und übergeben Sie den Wert einer `out` Variable. Die Normale wird im Fragment Shader für die Berechnung des Einfallwinkels des Lichts benötigt. Das ist alles was Sie für einen Toonshader benötigen. Natürlich müssen Sie noch den Output `gl_Position` setzen, da dieser in jedem Vertex Shader erwartet wird.

1.2. Fragment Shader

Als erstes definieren Sie die `varying` Variable für die Normale. Zusätzlich müssen Sie im Fragment Shader noch eine `uniform` Variable definieren, um aus der Szene die Richtung des Lichts zu übergeben. Anstatt dem Objekt einen weichen Farbverlauf zu verleihen, werden nun nur wenige Stufen/Farben verwendet. Welcher Pixel welche Farbe erhält wird über den Einfallswinkel des Lichtes bestimmt. Dazu müssen Sie das Skalarprodukt zwischen Normale und Lichtrichtung berechnen. Definieren Sie nun die verschiedenen Wertebereiche und weisen Sie jedem Wertebereich eine Farbe zu. Am Schluss setzen Sie den Output `gl_FragColor` auf den berechneten Farbwert.

1.3. Anpassung Szene

Nun fehlt nur noch die Übergabe der Lichtrichtung an den Shader. Sie haben bereits im Fragment Shader die `uniform` Variable für die Lichtrichtung deklariert. Nun müssen Sie die Material-Instanzierung so erweitern, damit der Richtungsvektor als `uniform` Variable übergeben wird.

1.4. Resultat

Das Resultat sollte in etwa wie folgt aussehen:

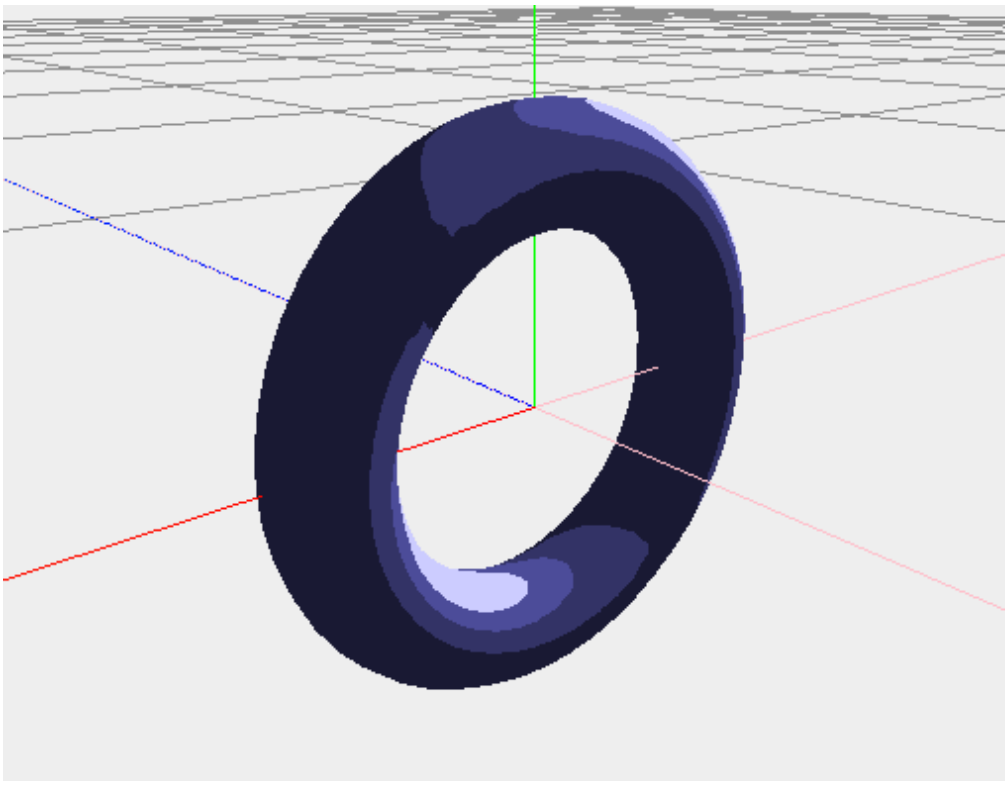

Figure 1.1.: Resultat Toon Shader

Dafür wurden folgende Werte verwendet:

- $Skalarprodukt > 0.8$: rgba(0.8,0.8,1.0,1.0)
- $Skalarprodukt > 0.6$: rgba(0.3,0.3,0.6,1.0)
- $Skalarprodukt > 0.3$: rgba(0.2,0.2,0.4,1.0)
- $Skalarprodukt <= 0.3$: rgba(0.1,0.1,0.2,1.0)

2. Aufgabe - Bump Mapping

Sie haben bereits einige Arten von Shading kennengelernt. Zum Beispiel Gouraud oder Phong Shading. Bump Mapping ist eine weitere Shading-Variante, welche dazu verwendet wird um die Oberfläche eines Objektes detailgetreuer darzustellen, ohne die eigentliche Geometrie zu ändern.

2.1. Texture Mapping

In einem ersten Schritt implementieren Sie einen Shader, welcher eine Textur auf das Objekt mappt.

Für diesen Teil arbeiten Sie in den folgenden Files:

- *bumpmapping.html*
- *bumpmapping.js*
- *shader/bump_texture.vert*
- *shader/bump_texture.frag*

2.1.1. Vertex Shader

Übergeben Sie im Vertex Shader die aktuellen Texturkoordinaten an den Fragment Shader. Diese erhalten Sie mit der vordefinierten attribute Variablen `uv`. Diese Koordinate gilt für den aktuellen Vertex. Im Fragment Shader erhalten Sie dann die interpolierten Koordination pro Fragment.

2.1.2. Fragment Shader

Deklarieren Sie im Fragment Shader eine `uniform` Variable vom Typ `sampler2D` mit welcher die Textur übergeben wird. Anhand der übergebenen Texturkoordinaten lesen Sie die Farbe aus dieser Textur aus und weisen Sie dem Output `gl_FragColor` zu. Um die Farbe auszulesen verwenden Sie die Funktion `texture2D()`.

2.1.3. Anpassung der Szene

Passen Sie in der Szene die Erstellung des Materials so an, damit die Textur *textures/mur_Ambiant.bmp* mit einer `uniform` Variablen übergeben wird.

2.2. Normal Mapping

Beim Normal Mapping werden die Normalen welche für die Lichtberechnung verwendet werden aus einer Normal Map gelesen. Die Normal Map ist eine 2D Textur, bei welcher die Normalen mit Farbwerten codiert sind.

Für diesen Teil arbeiten Sie in den folgenden Files:

- *bumpmapping.html*
- *bumpmapping.js*
- *shader/bump_normal.vert*
- *shader/bump_normal.frag*

2.2.1. Vertex Shader

Im Vertex Shader müssen folgende Werte berechnet werden:

- Richtung zum Licht
- Richtung zur Kamera

Für die Berechnungen benötigen Sie die Position des Lichts und die Tangente. Diese übergeben Sie als `uniform` Variable aus der Szene. Die Position der Kamera erhalten Sie mit der vordefinierten Variable `cameraPosition`.

Als erstes benötigen Sie die inverse TBN Matrix, um die Vektoren später in den Tangent Space zu transformieren. In unserem Fall können wir die transponierte Matrix als Inverse verwenden, da es keine Deformationen gibt. Somit sind die transponierte und die inverse Matrix identisch.

Berechnen Sie nun die Richtung zum Licht und zur Kamera und transformieren Sie die resultierenden Vektoren in den Tangent Space.

Übergeben Sie diese Vektoren, sowie die Texturkoordinaten an den Fragment Shader.

2.2.2. Fragment Shader

Erstellen Sie zuerst alle nötigen `uniform` Variablen:

- Textur
- Normal Map
- Ambiente Materialfarbe
- Diffuser Farbanteil

- Spekularer Farbanteil
- Shininess

Sowie die `varying` Variablen:

- Texturkoordinaten
- Lichtvektor
- Kameravektor

Lesen Sie nun die Normale aus der Normal Map mit den aktuellen Texturkoordinaten. Die Normalen liegen nun im Wertebereich [0,1]. Wir benötigen für die Berechnung allerdings den Wertebereich [-1,1]. Skalieren Sie also das Resultat mit dem Faktor 2 und subtrahieren Sie davon 1. Bevor Sie nun die Lichtberechnung durchführen normalisieren Sie alle Richtungsvektoren. Berechnen Sie nun die Farbe Des Fragments anhand der folgenden Formel.

$$
\begin{aligned}
halfVector &= normalize(light + eye) \\
lambert &= max(0.0, dot(norm, light)) \\
phong &= max(0.0, dot(norm, halfVector)) \\
specularPower &= phong^{shininess}
\end{aligned}
$$

$$
\begin{aligned}
diffuse &= I_d * lambert \\
specular &= I_{sp} * specularPower
\end{aligned}
$$

$$
I = p_a * baseColor + diffuse * baseColor + specular
$$

Legende:

- I resultierende Farbe
- p_a ambienter Materialanteil
- $baseColor$ Texturfarbe
- I_d diffuser Anteil der Lichtquelle
- I_{sp} spekularer Anteil der Lichtquelle

2.2.3. Anpassung der Szene

In der Szene müssen Sie nun alle `uniform` Variablen dem Shader übergeben. Verwenden Sie für die Normal Map die Textur *texture/mur_NormalMap.bmp*. Die restlichen Werte können Sie selber definieren, so dass es ein gutes Resultat ergibt.

2.2.4. Resultat

Das Resultat sollte in etwa wie folgt aussehen:

Figure 2.1.: Resultat Normal Mapping

2.3. Parallax Mapping

Beim Parallax Mapping berücksichtigt man beim mappen der Textur die aktuelle Position des Betrachters. Aufgrund einer Height Map erechnet man, welches Texel er wirklich sehen müsste.

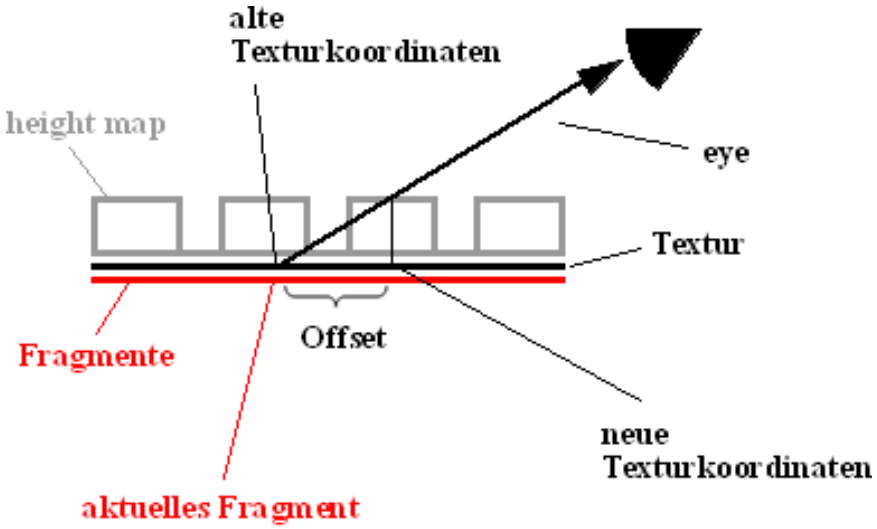

Figure 2.2.: Parallax Mapping

Für das Parallax Mapping kann der Vertex Shader vom Normal Mapping ohne Anpassungen übernommen werden.

Für diesen Teil arbeiten Sie in den folgenden Files:

- *bumpmapping.html*
- *shader/bump_parallax.vert*
- *shader/bump_parallax.frag*

2.3.1. Fragment Shader

Für das Parallax Mapping brauchen wir eine dritte Textur, die Height Map. Lesen Sie aus der Height Map die Höhe für das aktuelle Fragment aus. Verwenden Sie dafür einen der drei RGB Werte. Es spielt keine Rolle welchen, alle sind identisch.

Zusätzlich benötigen Sie zwei weitere `uniform` Variablen vom Typ `float`. Nennen Sie diese `scale` und `bias`. Berechnen Sie nun anhand folgender Formel die neuen Texturkoordinaten.

$$hsb = height * scale + bias$$
$$newCoord = oldCoord + hsb * eyeVec$$

Zum Schluss müssen Sie noch das Auslesen der Normale und der Farbe aus den Texturen durch die neuen Koordinaten anpassen.

2.3.2. Anpassung der Szene

Nun müssen Sie in der Szene noch die `uniform` Variablen definieren. Verwenden Sie dafür folgende Werte:

- heightMap *textures/mur_HeightMap.bmp*
- scale 0.04
- bias -0.02

2.3.3. Resultat

Das Resultat sollte in etwa wie folgt aussehen:

Figure 2.3.: Resultat Parallax Mapping

3. Aufgabe - Environment Mapping

Environment Mapping ist eine einfache Möglichkeit, Reflexionen an einem Objekt darzustellen. Dabei wird die Umgebung des Objektes in einer so genannten Environment Map abgespeichert. Mithilfe eines Shaders werden dann die Farbwerte der Environment Map auf das Objekt projiziert. Wir verwenden hier eine Cube Map, da diese mit einem Shader einfacher zu realisieren ist.

Für diesen Teil arbeiten Sie in den folgenden Files:

- *environmentmapping.html*
- *environmentmapping.js*
- *shader/env.vert*
- *shader/env.frag*

3.1. Vertex Shader

Die Aufgabe des Vertex Shaders ist hauptsächlich die Berechnung des reflektierten Strahls. Berechnen Sie die Richtung des reflektierten Sichtstrahls und übergeben Sie diese an den Fragment Shader. Arbeiten Sie im World Space damit Sie keine inverse Matrizen berechnen müssen. GLSL bietet bereits eine vordefinierte Funktion `reflect()` für die Berechnung des reflektierten Sichtstrahls.

Berechnen Sie nun noch die Lambert-Komponente. Diese berechnet sich aus dem Skalarprodukt von Lichtstrahl und Normale, wobei das Resultat im Minimum 0 sein darf. Übergeben Sie auch diesen Wert an den Fragment Shader.

3.2. Fragment Shader

Lesen Sie im Fragment Shader die Farbe aus der Environment Map anhand des reflektierten Sichtstrahls. GLSL bietet dazu die Funktion `textureCube()` an.

Um eine realistischere Farbe zu erhalten berechnen Sie den diffusen Farbanteil, welchen wir danach mit der Farbe aus der Environment Map mischen. Verwenden Sie dafür, die im Vertex Shader berechnete Lambert-Komponente.

Mischen Sie nun die beiden Farbwerte, anhand eines Mischverhältnisses, welches als `uniform` Variable übergeben wird. Dazu können Sie die Funktion `mix()` verwenden.

3.3. Anpassung der Szene

Nun müssen Sie sich noch darum kümmern, dass die Environment Map erstellt wird. three.js bietet dafür die `CubeCamera` an, was das ganze sehr vereinfacht. Erstellen Sie eine solche `CubeCamera`, welche Sie mit der Kugel mit bewegt. Als Environment Map können Sie nun einfach das Render Target der `CubeCamera` (`cubeCamera.renderTarget`) als `uniform` Variable übergeben. Achten Sie darauf, dass die Cube Map für jedes Frame aktualisiert wird (`textureCam.updateCubeMap()`).
Übergeben Sie zum Schluss noch die restlichen uniform Variablen:

- Position der Lichtquelle
- Mischverhältnis
- Grundfarbe des Materials

3.4. Resultat

Das Resultat sollte in etwa wie folgt aussehen:

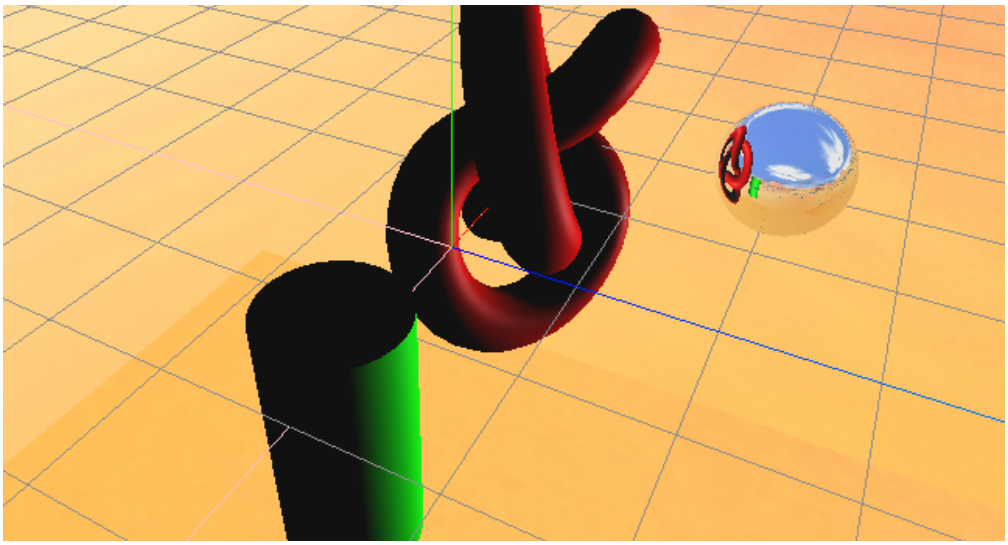

Figure 3.1.: Resultat Environment Mapping

Dafür wurden folgende Werte verwendet:

- Mischverhältnis: 0.1

- Position Lichtquelle: $xyz(1.0, -0.5, 1.0)$
- Grundfarbe: $rgb(0.5, 0.5, 0.5)$

Anhang

A. three.js

Three.js ist eine JavaScript Library für die Erstellung und Animation von 3D Objekten. Hier sind die wichtigsten Funktionen beschrieben, welche in diesem Praktikum benötigt werden.

A.1. Aufbau einer Applikation

```
<script>
  //Definition Variablen
  var camera, scene, renderer,
  geometry, material, mesh;

  init();
  animate();

  function init() {
      //Erstellung/Initialisierung der Szene
  }

  function animate() {
      //Animationsschleife
  }

  function render() {
   //Rendering der Szene
   //Wird fuer jedes Frame ausgefuehrt
  }
</script>
```

Listing A.1: Aufbau three.js Applikation

A.2. Shader Material

Um einen Shader auf ein Objekt anzuwenden wird das `ShaderMaterial` verwendet. Dabei werden `uniforms` sowie der Code der Shader übergeben.

```
var shaderMaterial = new THREE.ShaderMaterial({
  uniforms:       uniforms,
  vertexShader:   env_vs,
  fragmentShader: env_fs
});
```

Listing A.2: ShaderMaterial

A.2.1. Datenübergabe an Shader

Die `uniform` Variablen werden dem Material als Javascript Object übergeben. Jede Variable enthält einen Wert (`value`) und einem impliziten Typ.

```
var uniforms = {
  baseColor: {
    value:  new THREE.Vector3(0.5, 0.5, 0.5}
  }
};
```

Listing A.3: Deklaration uniform Variable

Eine Übersicht über unterstützte Datentypen finden sie hier:
https://threejs.org/docs/#api/en/core/Uniform

A.3. Shader

Der grundsätzliche Aufbau eines Shaders sieht wie folgt aus:

```
\\Deklaration uniforms
uniform samplerCube envMap;

\\Deklartion attributes
attribute float mixRatio;

\\Deklartion in/out
in/out vec3 reflectDir;

void main(void)
{
  \\Code
}
```

Listing A.4: Aufbau Shader

A.3.1. Vordefinierte uniforms / attributes

Three.js bietet bereits einige vordefiniert `uniform` und `attribute` Variablen an. Die wichtigsten sind hier aufgelistet.

- normal — Normale des aktuellen Vertex im Objectspace
- position — Position des aktuellen Vertex im Objectspace
- uv — Texturkoordinaten des aktuellen Vertex im Objectspace
- normalMatrix — Objectspace → Worldspace
- modelMatrix — Objectspace → Worldspace
- modelViewMatrix — Objectspace → Viewspace

- projectionMatrix
- cameraPosition Position der Kamera im Worldspace

A.3.2. Vordefinierte Funktionen

GLSL bietet bereits einige vordefiniert Funktionen an. Die wichtigsten sind hier aufgelistet.

- texture2D(sampler2D, vec2) Farbe aus Textur lesen
- textureCube(samplerCube, vec3) Farbe aus Cube Map lesen
- normalize(vec) Vektor normieren
- dot(vec, vec) Skalarprodukt
- power(float, float) Potenz
- reflect(vec, vec) Reflektierender Vektor bestimmen
- max(float, float)
- mix(vec3, vec3, float) Mischt zwei Vektoren mit bestimmtem Verhältnis

B. Theoretische Grundlagen

B.1. Tangent Space

Der Tangent Space ist der Raum, in welchem die Texturkoordinaten definiert sind. Der Tangent Space wird von drei Basisvektoren aufgespannt (**Tangent**, **Bitangente**, **Normale**), welche auch für die drei Grundfarben stehen, mit welchen die Normalen in der Normal Map codiert werden. Obwohl die Normal Map eine normale 2D-Textur ist, befinden sich die Normalen, welche in der Normal Map codiert werden, in einem dreidimensionalen Raum, welcher von den drei Vektoren gemäss Abbildung 2 aufgespannt wird. Damit wir mit diesen Vektoren rechnen können, müssen alle anderen Vektoren ebenfalls in den "Tangent Space" transformiert werden (oder man transformiert die Vektoren aus dem Tangent Space in den World Space). Dazu verwenden wir die so genannte inverse TBN Matrix (resp. die TBN Matrix).

$$M_{TBN} = \begin{pmatrix} t_x & b_x & n_x \\ t_y & b_y & n_y \\ t_z & b_z & n_z \end{pmatrix} \tag{B.1}$$

Die TBN Matrix setzt sich zusammen aus drei Vektoren: Tangente, Bitangente und Normale. Tangente und Bitangente können mit einem Gleichungssystem berechnet werden. Die Normale ist schliesslich einfach das Kreuzprodukt aus Tangente und Bitangente.

$$(M_{TBN})^T = \begin{pmatrix} t_x & t_y & t_z \\ b_x & b_y & b_z \\ n_x & n_y & n_z \end{pmatrix} \tag{B.2}$$

Um vom World Space in den Tangent Space zu transformieren, verwendet man die inverse TBN Matrix. Im Bereich der Computergrafik ist die Inverse der TBN Matrix häufig einfach die Transponierte (da TBN Matrix häufig orthogonal ist), was die Berechnung der Inversen natürlich stark vereinfacht.