Computergrafik 2

Lab 1

Aufgabe 1 Textur-Koordinaten laden und anzeigen

(a) In der `setup()` Methode von `Mesh3DApp` (Mesh3D.js) wird ein Modell wie folgt geladen:

```
const streamReader
= await loadBinaryDataStreamFromURL("./../../data/bunnyUV.smm");
const mesh = await SimpleMeshModelIO.load(streamReader);
```

Daraus wird ein `TriangleMeshGL` erzeugt:

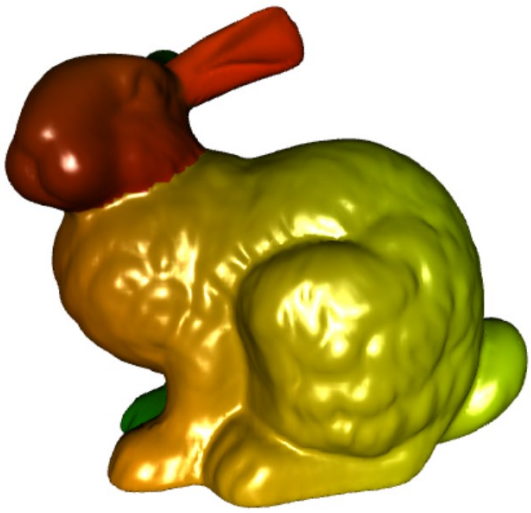
```
triangleMeshGL = new TriangleMeshGL(gl, mesh);
```

Erweitern Sie zunächst die Klasse `TriangleMeshGL` so, dass Texturkoordinaten für das Modell im Vertex-Shader verfügbar gemacht werden. Die Texturkoordinaten liegen bereits im Arbeitsspeicher der CPU und werden in `TriangleMeshGL.js` mittels

```
const texCoords     = simpleMeshIO.texCoords;
```

bereitgestellt. Diese müssen aber, ähnlich wie Normalen, Positionen und Farben, an die GPU in einem Array-Buffer hochgeladen werden und an das Vertex-Array gebunden werden. Laden Sie nun also die Texturkoordinaten auf die GPU hoch!

(b) Ersetzen Sie den roten und grünen Farbanteil der diffusen Farbe im Blinn-Phong-Beleuchtungsmodell durch die von der GPU perspektivisch korrekt interpolierten Texturkoordinaten. Den Blaukanal können Sie auf 0 setzen. Sie sollten folgendes Ergebnis erhalten. Passen Sie dazu die Shader mesh3d.vert.glsl und mesh3d.frag.glsl an!



Aufgabe 2 Textur Laden

Im Konstruktor der Klasse `TextureMap` wird bereits eine 1x1 Textur erzeugt. Dazu wir zunächst ein Handle im WebGL Treiber für eine Textur angelegt.

```
this.texture = gl.createTexture();
```

Anschließend wird diese Textur als 2D Textur gebunden. „Binden“ bedeutet, dass alle folgenden WebGL Kommandos, die Auswirkungen auf 2D Texturen haben, diese Textur betreffen werden.

```
gl.bindTexture(gl.TEXTURE_2D, this.texture);
```

Nun werden die Daten unsere 1x1 Textur wie folgt hochgeladen:

```
gl.texImage2D(gl.TEXTURE_2D, 0, this.gl.RGBA, this.width, this.height, 0, gl.RGBA,
gl.UNSIGNED_BYTE, new Uint8Array([255, 255, 255, 255]));
```

Machen Sie sich in der WebGL Dokumentation mit den Parametern und Varianten der Funktion `texImage2D` vertraut! Abschließend wird die Textur wie folgt entbunden:

```
gl.bindTexture(gl.TEXTURE_2D, null);
```

und wird bis auf Weiteres nicht verwendet.

(a) Wir müssen in der Lage sein, eine Textur von der Klasse MeshGL3D aus zu benutzen und somit zu binden und zu entbinden. Implementieren Sie deshalb die Methoden `bind()` und `unbind()` in TextureMap.js.

(b) Die Methode `loadTexture` (TextureMap.js) lädt asynchron ein Bild aus der Datei `filename` vom Dateisystem des Webservers. Sobald das Laden erfolgreich beendet ist, wird die Methode `loadTextureData` aufgerufen. Implementieren Sie diese, so dass die Pixeldaten aus `this.image` in der WebGL Textur bereitstehen. Erzeugen Sie zudem alle MIP-Map-Stufen.

Hinweis 1: Sie benötigen unter anderem die WebGL-Methoden `texImage2D` und `generateMipmap`.

Hinweis 2: Denken Sie ans Binden und Entbinden!

(c) Nutzen Sie die Methode `loadTexture` nun in Mesh3D.js und laden Sie die Textur aus der Datei `./../../data/bunnyUV.png`. Prefixen Sie den Aufruf mit `await` um zu Warten bis das Bild vollständig geladen ist.

(d) Damit die Textur beim Zeichnen des Netzes verwendet werden kann, muss diese vor dem Zeichnen gebunden werden. Nach dem Zeichen sollte sie wieder entbunden werden. Setzen Sie dieses Verhalten in der Methode `draw()` von `Mesh3DApp` (Mesh3D.js) um. Nutzen Sie dazu die Methoden `bind()` und `unbind()`, die Sie in Aufgabe (a) implementiert haben.

(e) Damit im Fragment-Shader mesh3d.frag.glsl von der Textur gelesen werden kann, muss einen uniforme Variable

```
uniform sampler2D u_textureA;
```

bereitgestellt werden. Mittels der GLSL Funktion `texture` kann nun eine Farbe von der Textur gesampelt werden. Recherchieren Sie die Funktionsweise der GLSL Funktion `texture` und nutzen Sie die Texturfarbe als diffuse Farbe für Ihre Beleuchtung. Sie sollten folgendes Ergebnis erhalten:

(f) Wie Ihnen sicher auffällt, stimmt irgendwas mit den Texturkoordinaten nicht, und zwar ist die v-Achse gespiegelt. Spiegeln Sie deshalb im Vertex-Shader die v-Achse geeignet um folgendes Ergebnis zu erhalten:

Aufgabe 3 MIP Mapping

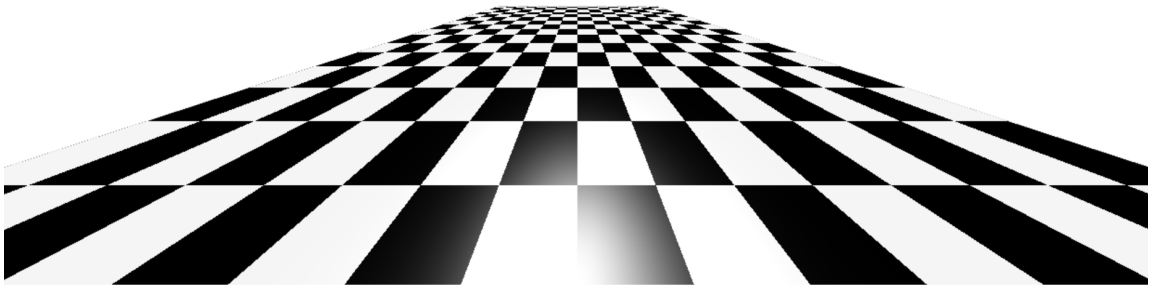
Laden Sie nun das Modell `"./../../data/plane.smm"` und die Textur `"./../../data/checkerboard.png"`. Stellen Sie in mesh3d.frag.glsl die Lichtquelle über das Modell, ungefähr so:

```
vec3 lightPosition              = vec3(0.0, 1000.0, 1.0);
```

Zudem empfiehlt es sich in Mesh3D.html die `value`-Einträge wie folgt zu setzen:

TranslateZ = -1.1, RotationX = 5, FieldOfView = 2, NearPlane = 0.01.

Sie sollten dann folgendes Bild erhalten:



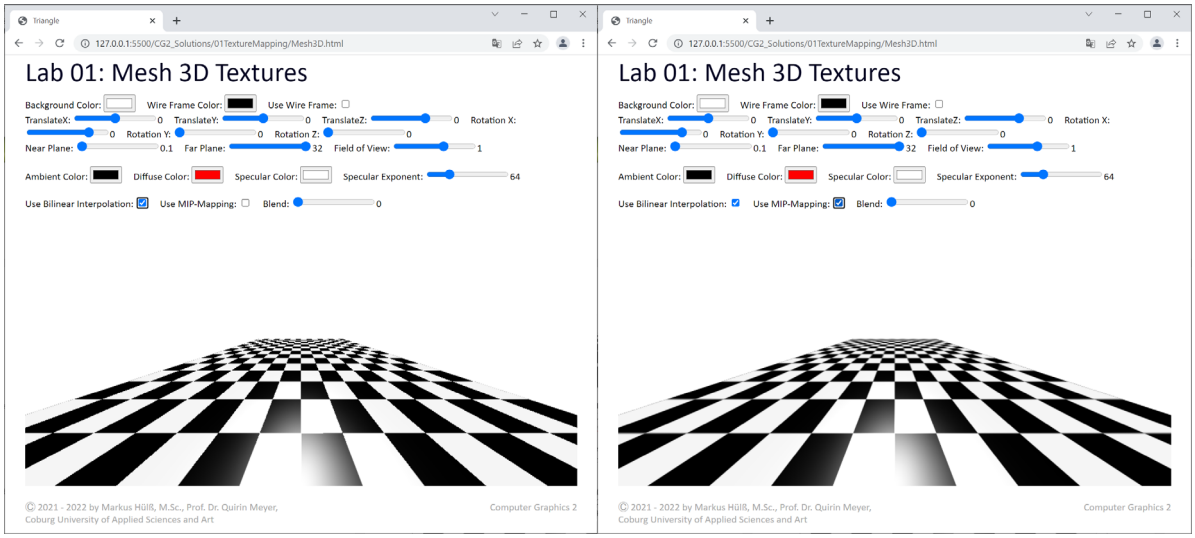
(a) Über die Checkbox `useBilinearInterpolation` sollen Nutzer:innen bilineare Interpolation entsprechend an- und ausschalten können. Übergeben Sie den Status der Checkbox an die Member-Variable `useBilinearInterpolation` Ihres Textur-Objektes.

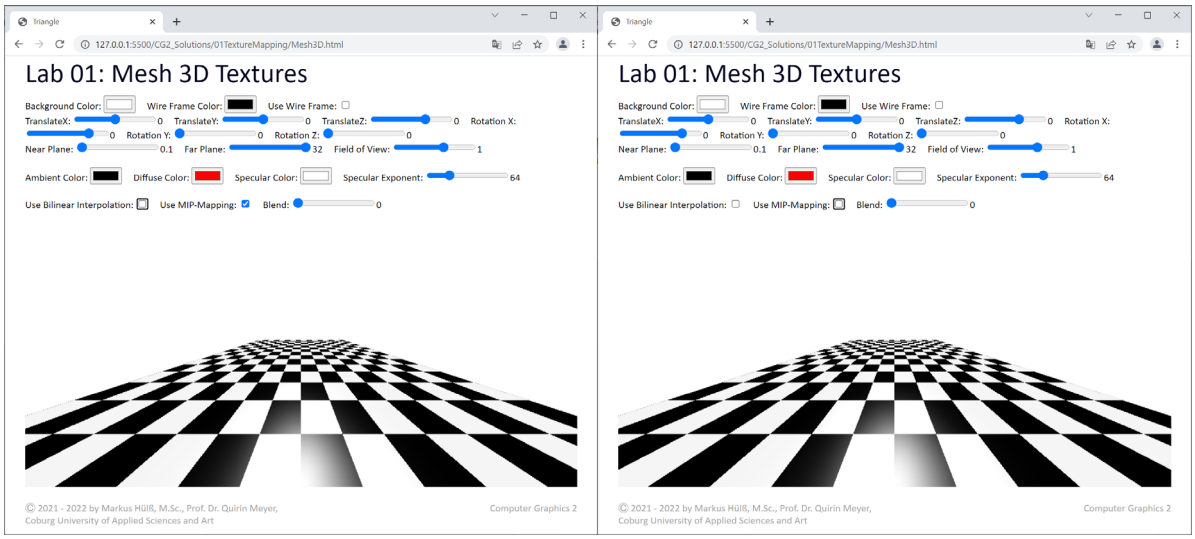
Konfigurieren Sie dann die in der Methode `bind()` (TexturMap.js) gebundene Textur so, dass bilineare Interpolation verwendet oder nicht verwendet wird. Dies können Sie WebGL mittels der Methode `texParameteri` mitteilen.

(b) Weiter sollen Nutzer:innen über die Checkbox `useMIPMapping` (Mesh3D.html) MIP Mapping aktivieren können. Teilen Sie Ihrem `TextureMap` Objekt über dessen Membervariable `useMIPMapping` den gewünschten Zustand mit. Schalten Sie mit dieser Information nun in der `bind()` MIP-Mapping an oder aus.

Hinweis: Sie müssen hier zusätzlich noch das Flag `useBilinearInterpolation` berücksichtigen.

Ein mögliches Ergebnis könnte so aussehen:





Aufgabe 4 Mehrere Texturen

(a) Erstellen Sie in Mesh3D.js eine weitere Textur. Dabei kommt der bereits einbaute zweite Parameter des Konstruktors von TextureMap zum Einsatz. Er legt die Textur-Unit fest. Weisen Sie der ersten Textur die Unit 0 und der zweiten die Unit 1 zu. Laden Sie die erste Textur mit dem checkerboard.png und die zweite Textur mit bunnyUV.png.

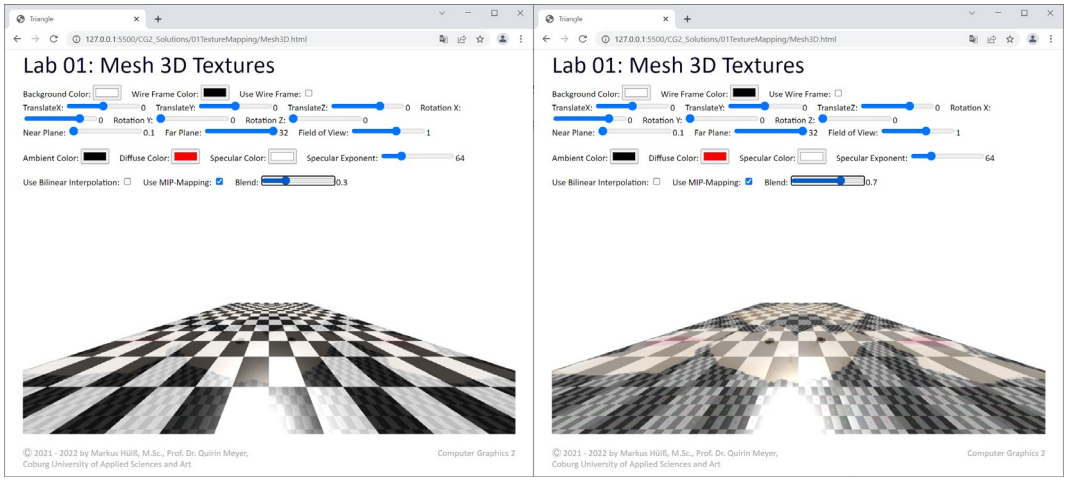
(b) Binden Sie die Textur in `bind()` an die richtige Textur-Unit. Wie das geht, finden Sie in der Dokumentation zu `gl.activeTexture`. Vergessen Sie nicht die Textur nach dem Draw-Call zu entbinden. Setzen Sie an der zweiten Textur die Werte `useBilinearInterpolation` und `useMIPMapping` genauso wie bei der ersten Textur.

(c) Legen Sie im Fragment Shader einen zweiten Sampler an. Setzen Sie mittels

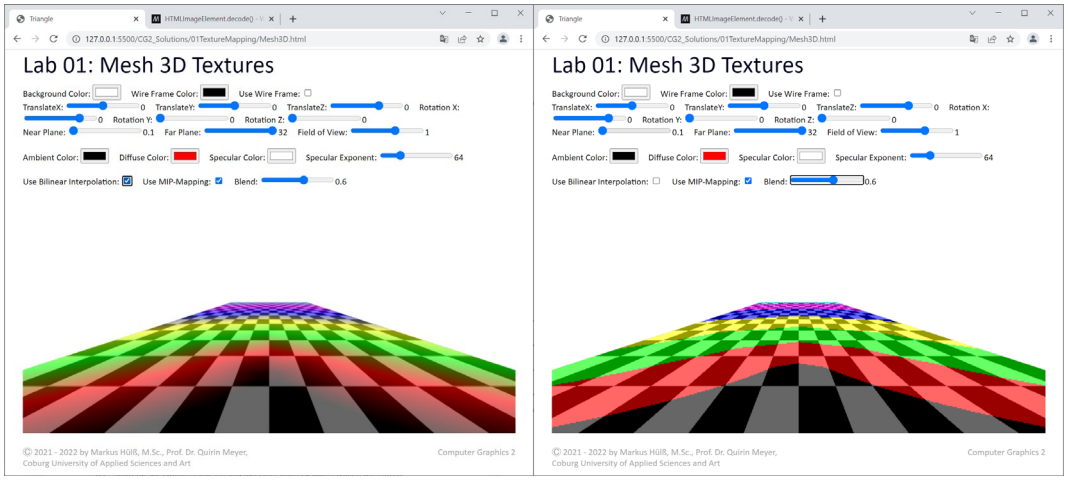
```
gl.uniform1i(mGlslProgram.getUniformLocation("u_sampler"),
             texture.getUnit());
```

die Textur-Unit jedes Samplers (also von Textur 1 und Textur 2) so, dass Sie von beiden Texturen im Shader samplen können. Samplen Sie im Shader auch von der zweiten Textur und setzen Sie zum Testen die Fragmentfarben auf den Wert der zweiten Textur.

(d) Übergeben Sie den Wert des Blend-Sliders an den Shadern und interpolieren Sie linear zwischen den beiden Texturen. Sie könnten folgendes Ergebnis erhalten.



(e) Statt für die zweite Textur ein Bild aus einer Datei zu laden, wollen wir mittels `createDebugTexture(maxLevel)` eine Textur erzeugen, die aus `maxLevel` MIP-Map Stufen besteht, wobei jede MIP-Map Stufe eine andere Farbe besitzt. Implementieren Sie diese Funktion! Anstelle eine Textur aus einer Datei zu alden, rufen Sie die Funktion auf, und setzen Sie `maxLevel` so, dass der Wert der Anzahl der MIP-Map Stufen der anderen Textur entspricht. Zusammen mit dem Blend-Slider können Sie so schön, die MIP-Map Level visualisieren:



Testen Sie das Ergebnis auch auf unterschiedlichen GPUs unterschiedlicher Hersteller. Was fällt Ihnen auf?