Praktikum 3: Texture Mapping

In diesem Praktikum werden Texturen auf ein 3D Modell gelegt.

1 Textur-Koordinaten laden und anzeigen

a) 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

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

bereitgestellt. Diese müssen aber, ähnlich wie Normalen und Positionen, an die GPU in einem Array-Buffer hochgeladen werden und an das Vertex-Array gebunden werden.

b) Reichen Sie die Texturkoordinaten vom Vertex Shader an den Fragment Shader! Geben Sie zum Testen, den Absolutbetrag der Texturkoordinaten als Farbe aus. Sie sollten folgendes Bild erhalten:

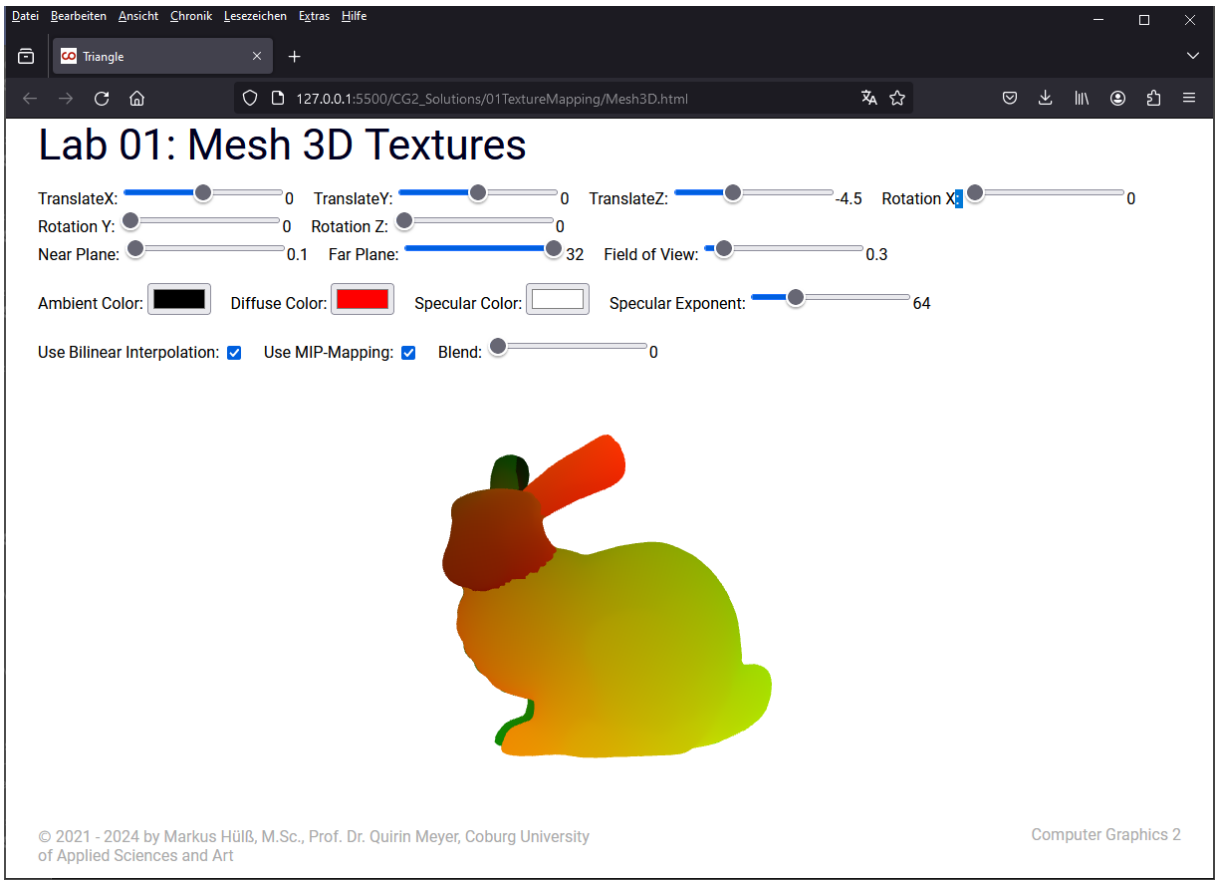


Abbildung 1: Der Hase mit Position als Vertex-Farben

2 Textur Laden

Im Konstruktor der Klasse `TextureMap` wird bereits eine 1x1 Textur erzeugt. Dazu wird zunächst WebGL Objekt für eine Textur angelegt.

```
1  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.

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

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

```
1  gl.texImage2D(gl.TEXTURE_2D, 0, this.gl.RGBA, this.width, this.height,
```

```
2                 0, gl.RGBA, gl.UNSIGNED_BYTE,
3                 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:

```
1 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 `Mesh3DApp` 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` (in `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!

   Hinweis 3: Höhe und Breite des Bildes bekommen Sie mittels `this.image.width` und `this.image.height`.

c) Nutzen Sie den `constructor` und die Methode `loadTexture` nun in `Mesh3DApp.js` um aus der Datei `./../../data/bunnyUV.png` eine Texture zu erzeugen. Prefixen Sie asynchrone Aufrufe direkt 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` (`Mesh3DApp.js`) um. Nutzen Sie dazu die Methoden `bind()` und `unbind()`.

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

   ```
   1 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:

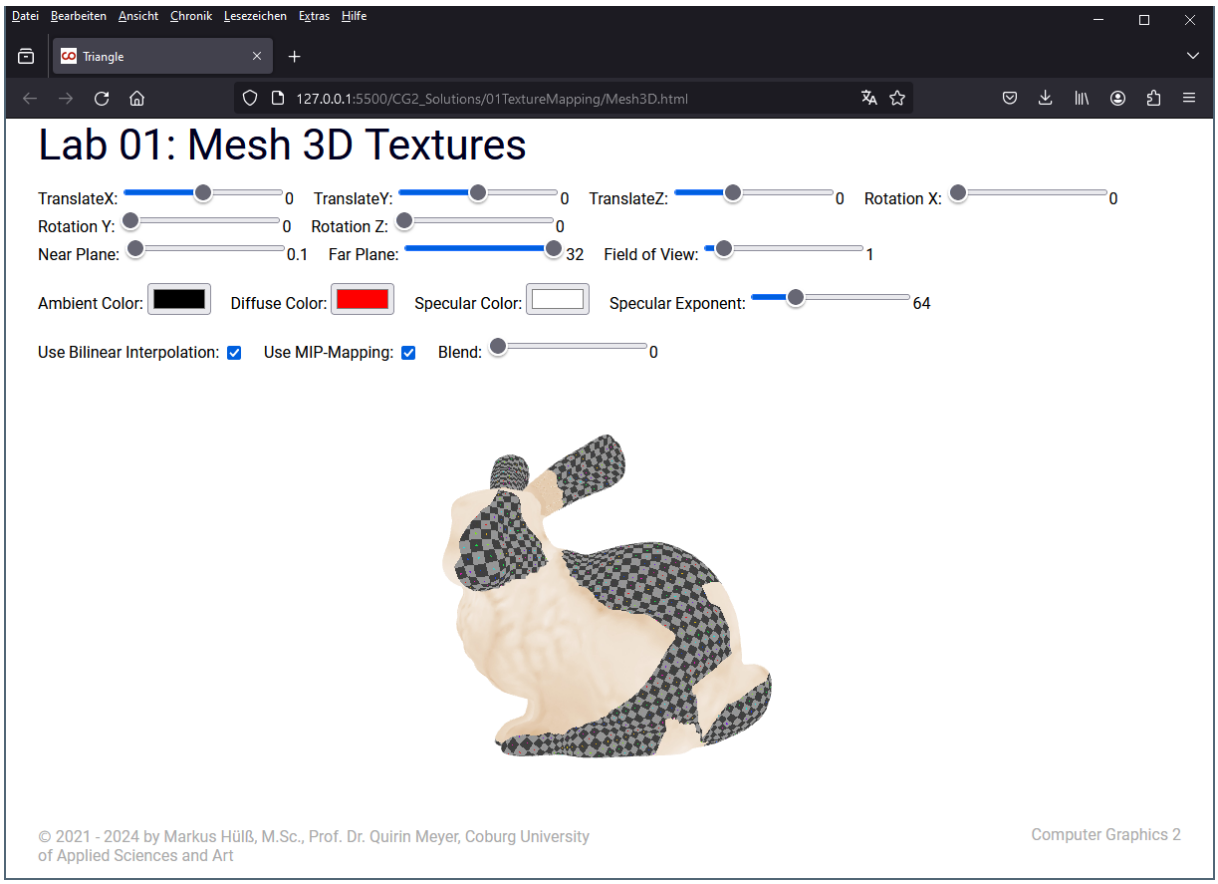


Abbildung 2: Der texturierte Hase, allerdings mit falschen Texturkoordinaten

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:

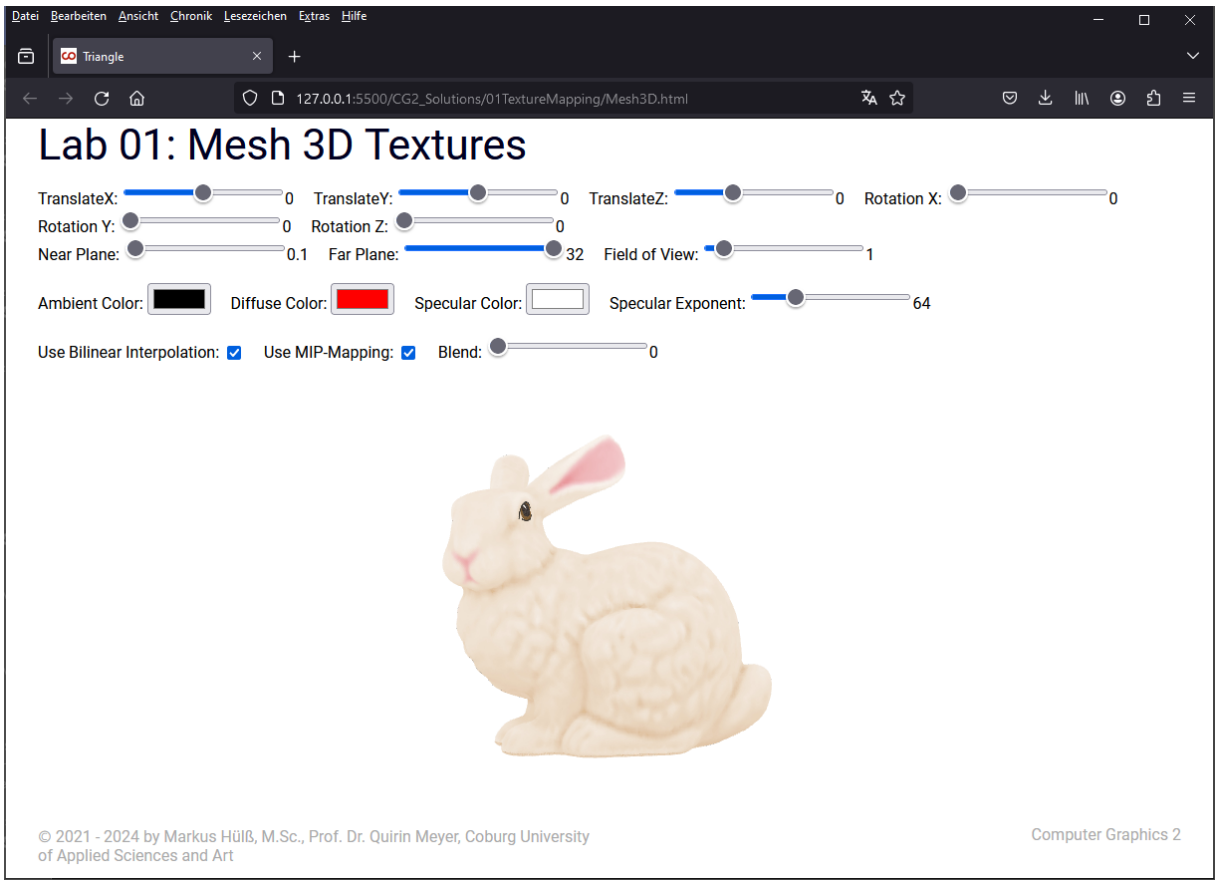


Abbildung 3: Nun ist der Hase richtig texturiert.

g) Ersetzen Sie im Blinn-Phong Beleuchtungsmodell die diffuse Farbe durch die Farbe aus der Textur!

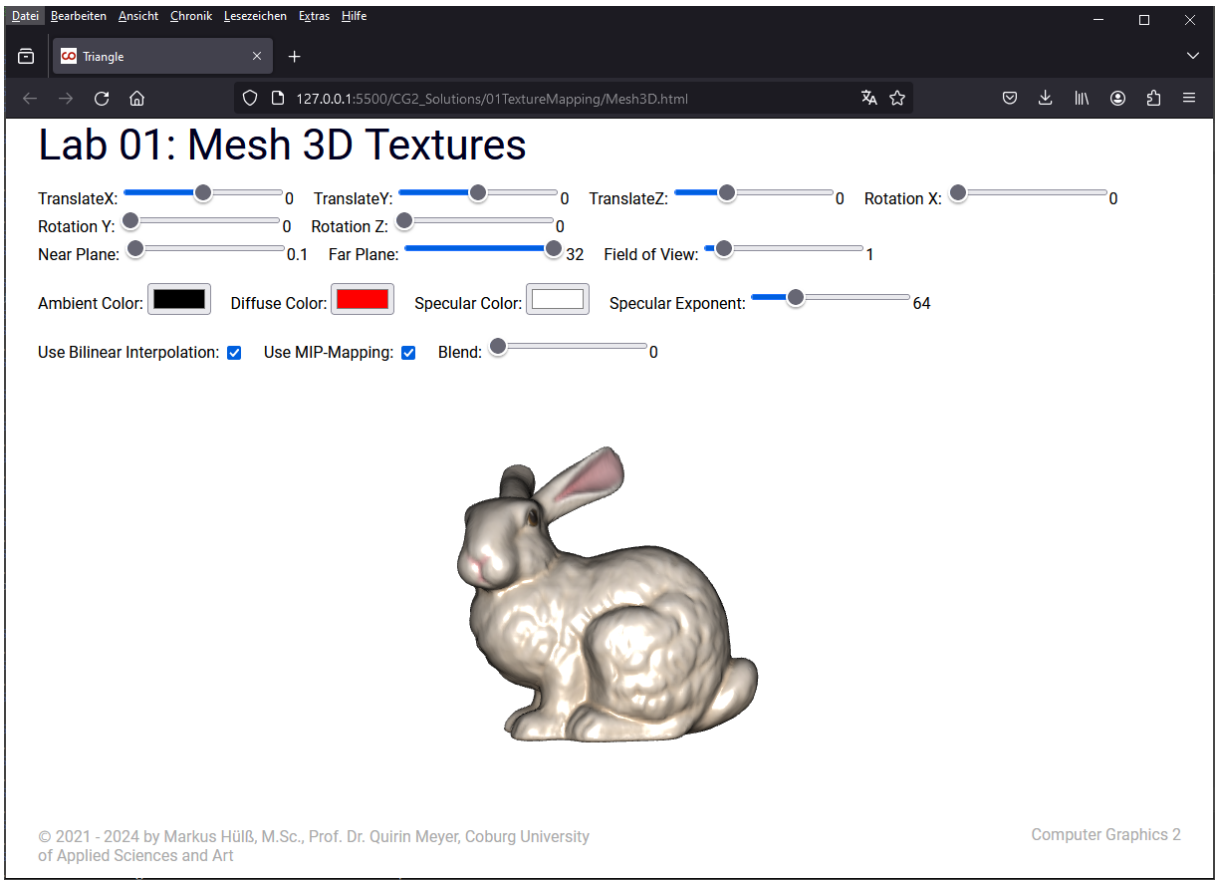


Abbildung 4: Der Hase mit Position als Vertex-Farben

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:

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

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

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

Sie sollten dann folgendes Bild erhalten:

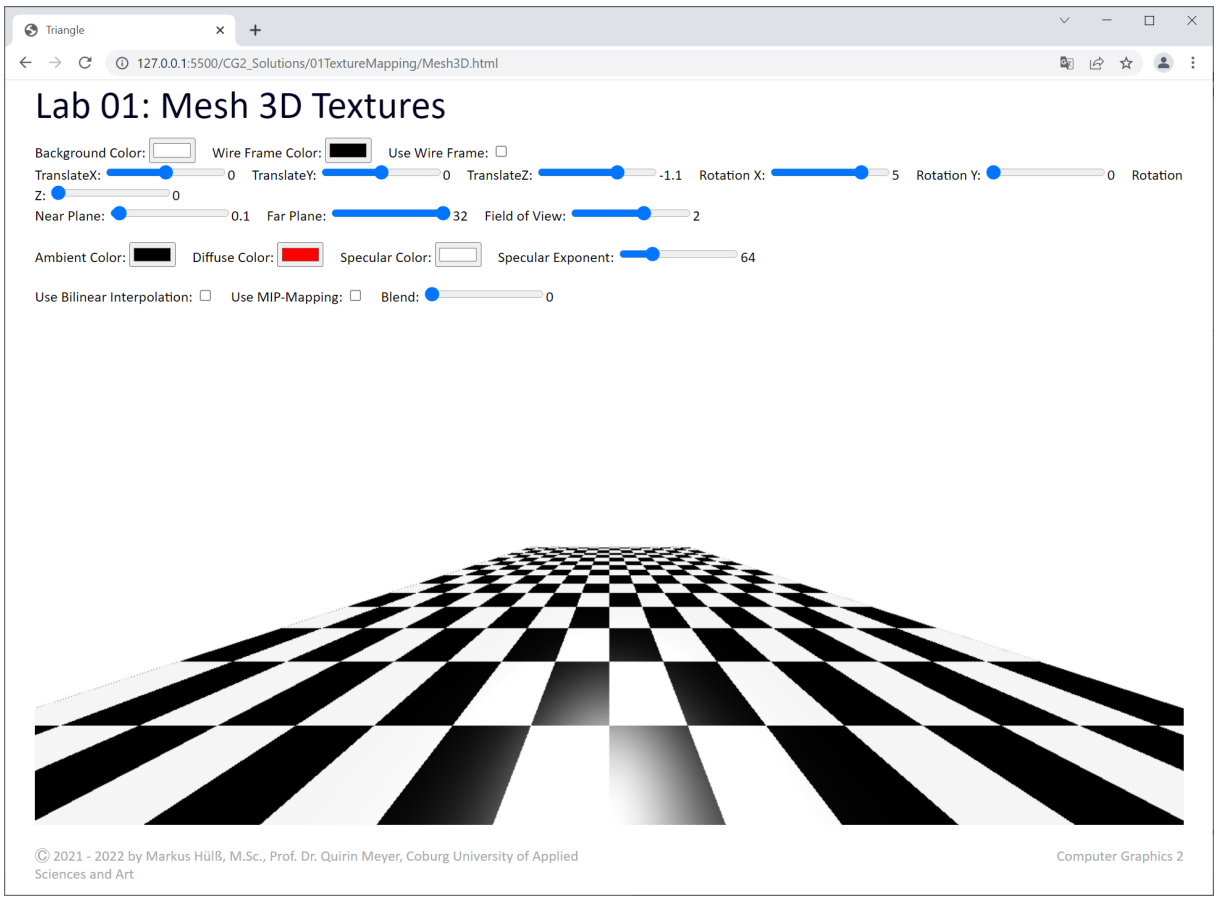


Abbildung 5: Eine Debug-Ansicht um MIP Mapping zun analysieren

Über die Checkbox `useBilinearInterpolation` sollen Nutzerinnen und Nutzer 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()` (`TextureMap.js`) gebundene Textur so, dass bilineare Interpolation verwendet oder nicht verwendet wird. Dies können Sie WebGL mittels der Methode `texParameteri` mitteilen.

Weiter sollen Nutzerinnen und Nutzer 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 `bind()` MIP-Mapping an oder aus.

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

Ein mögliches Ergebnis könnte so aussehen:

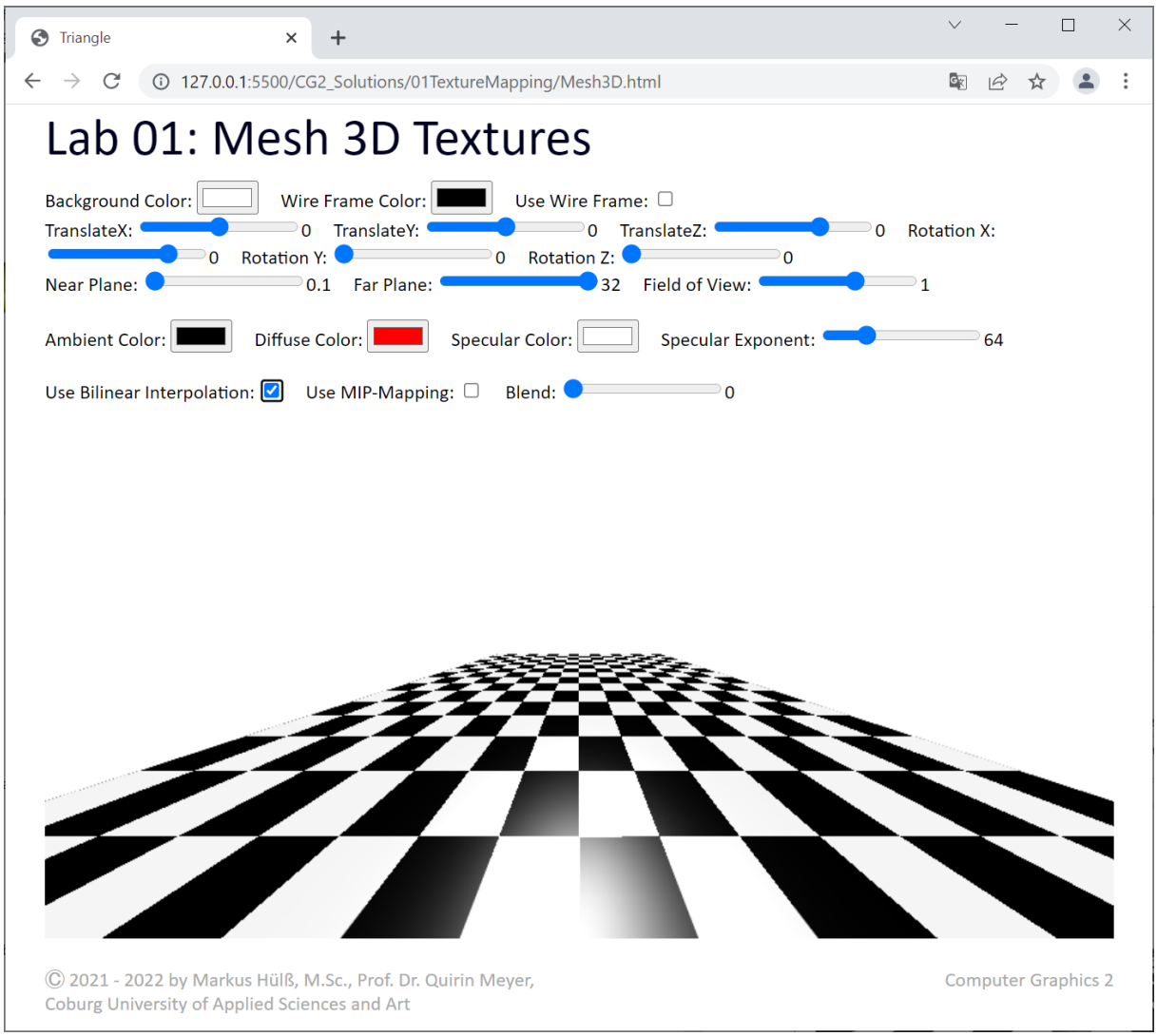


Abbildung 6: Bilineare Interpolation, aber kein MIP Mapping

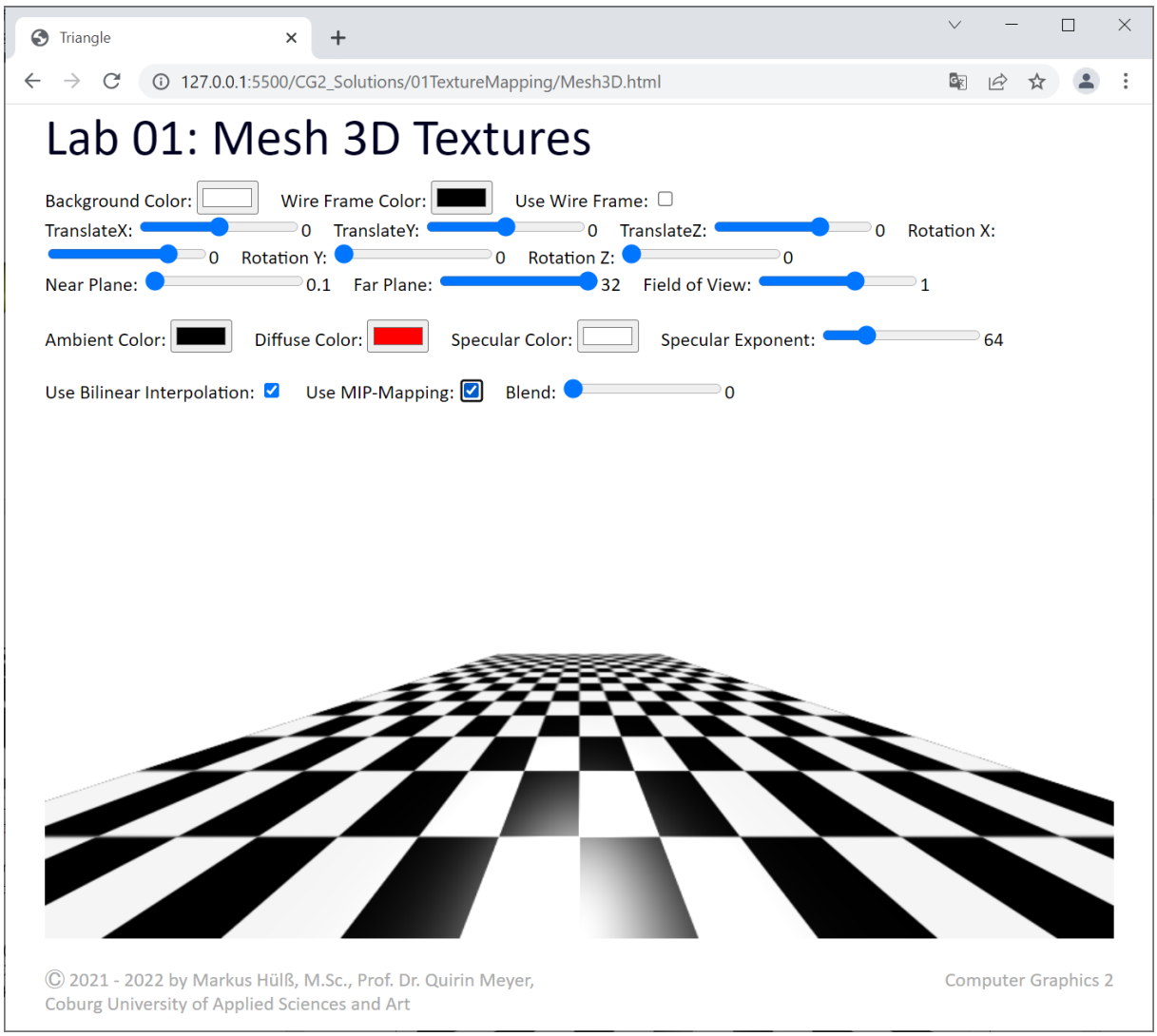


Abbildung 7: Trilinear Interpolation, und MIP Mapping

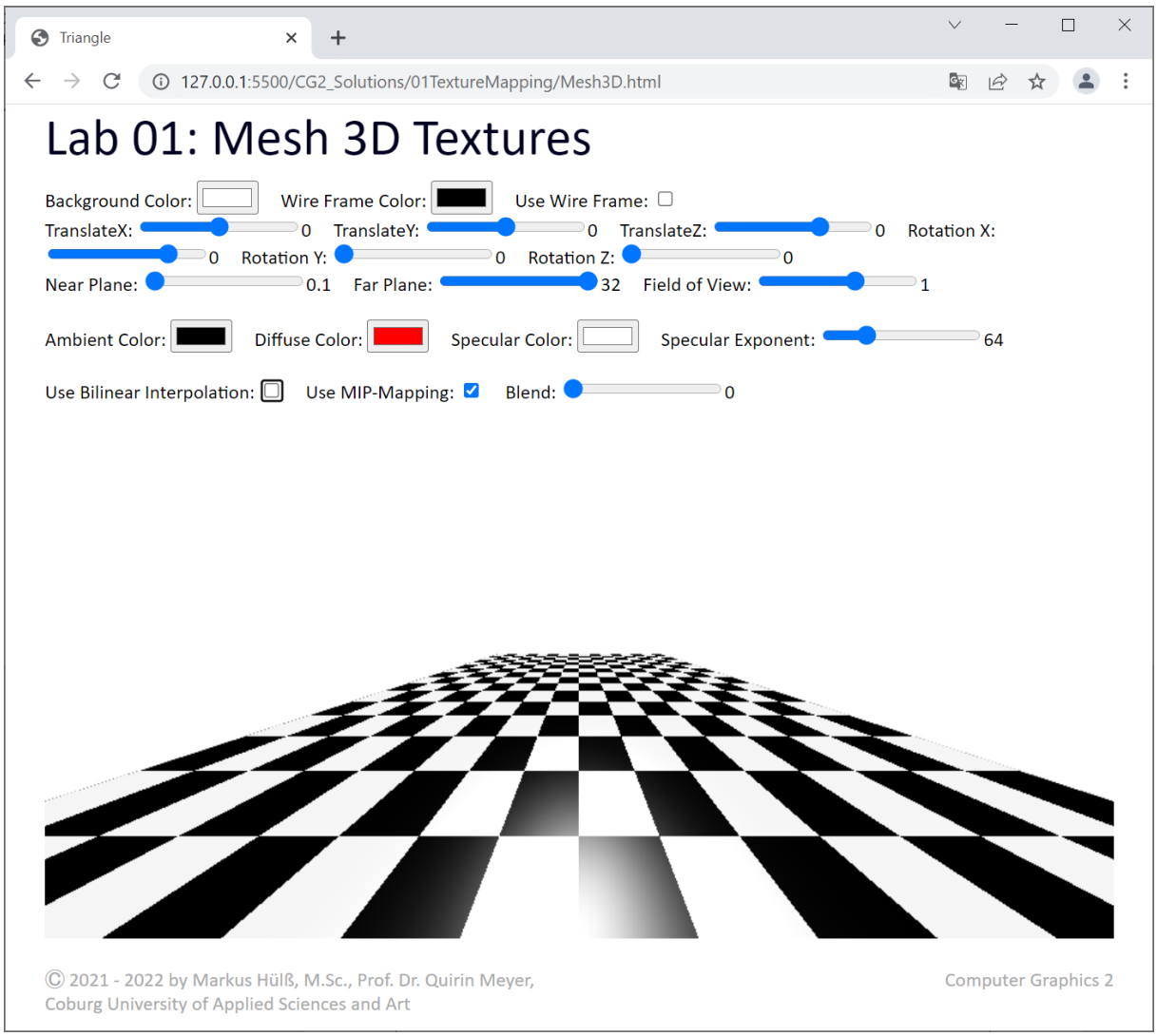


Abbildung 8: Nearest Neighbor interpolation, aber MIP Mapping

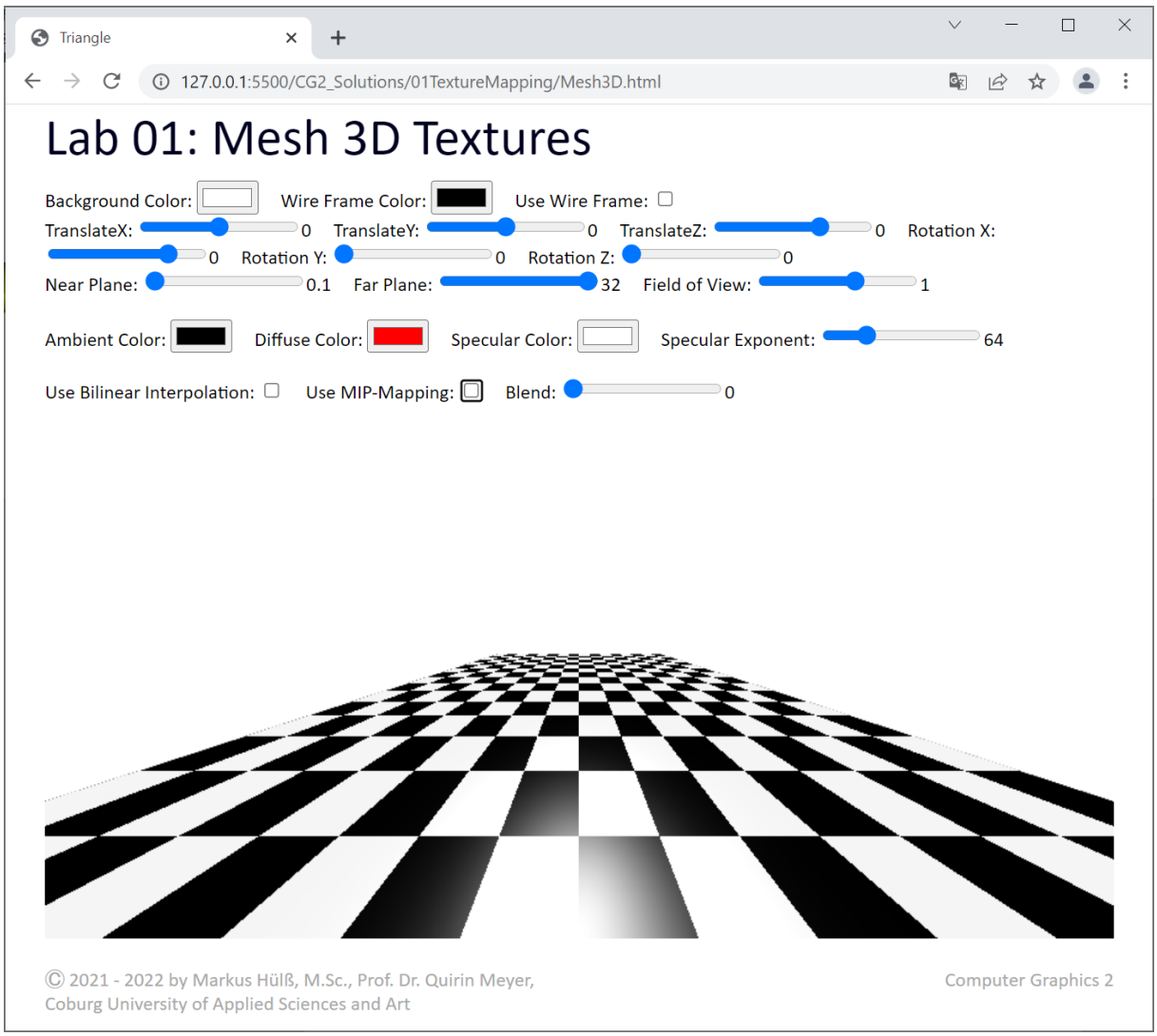


Abbildung 9: Nearest Neighbor interpolation, aber kein MIP Mapping

4 Mehrere Texturen

a) Erstellen Sie in `Mesh3DApp.js` eine weitere Textur. Dabei kommt der bereits eingebaute 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.

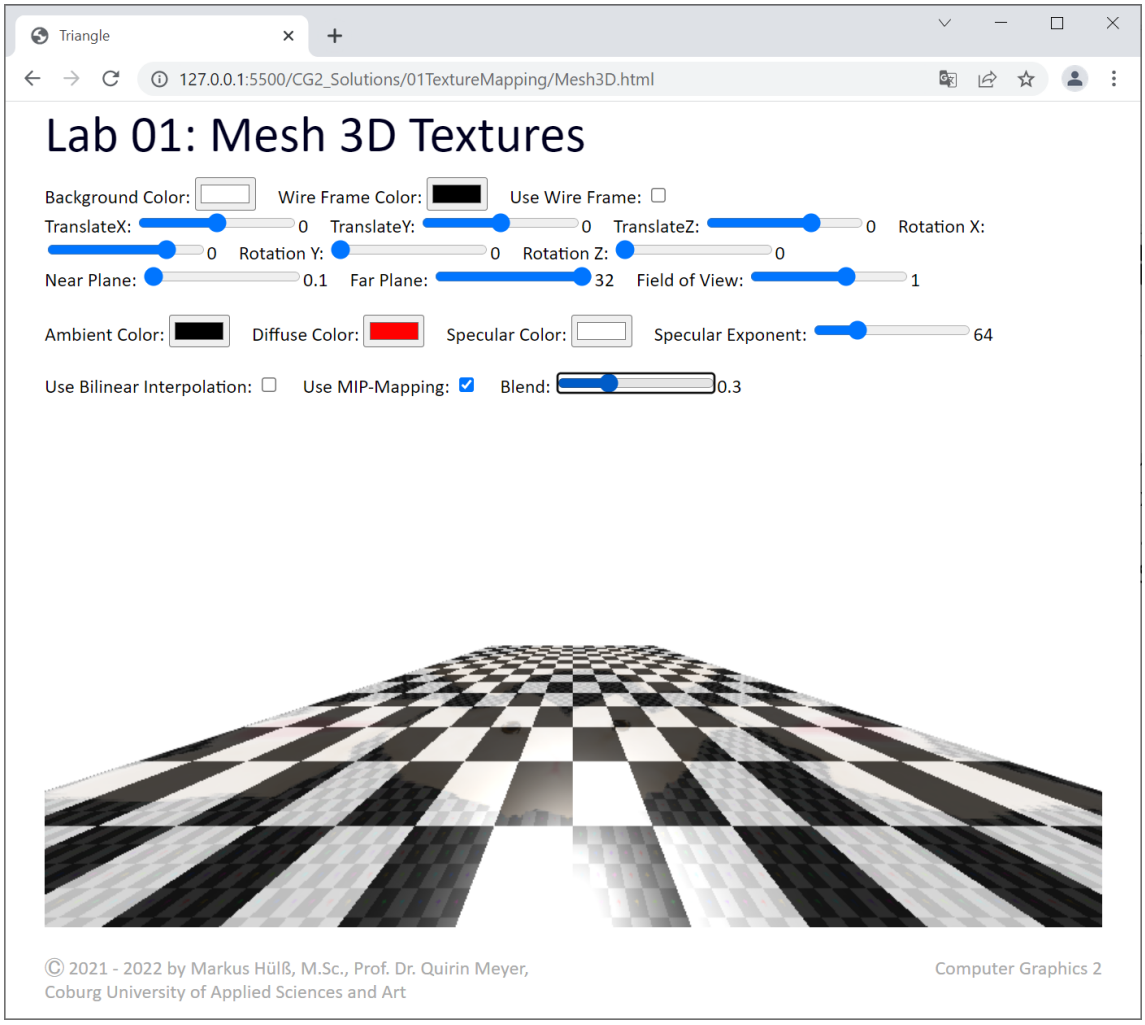
c) Legen Sie im Fragment Shader einen zweiten Sampler an. Setzen Sie mittels

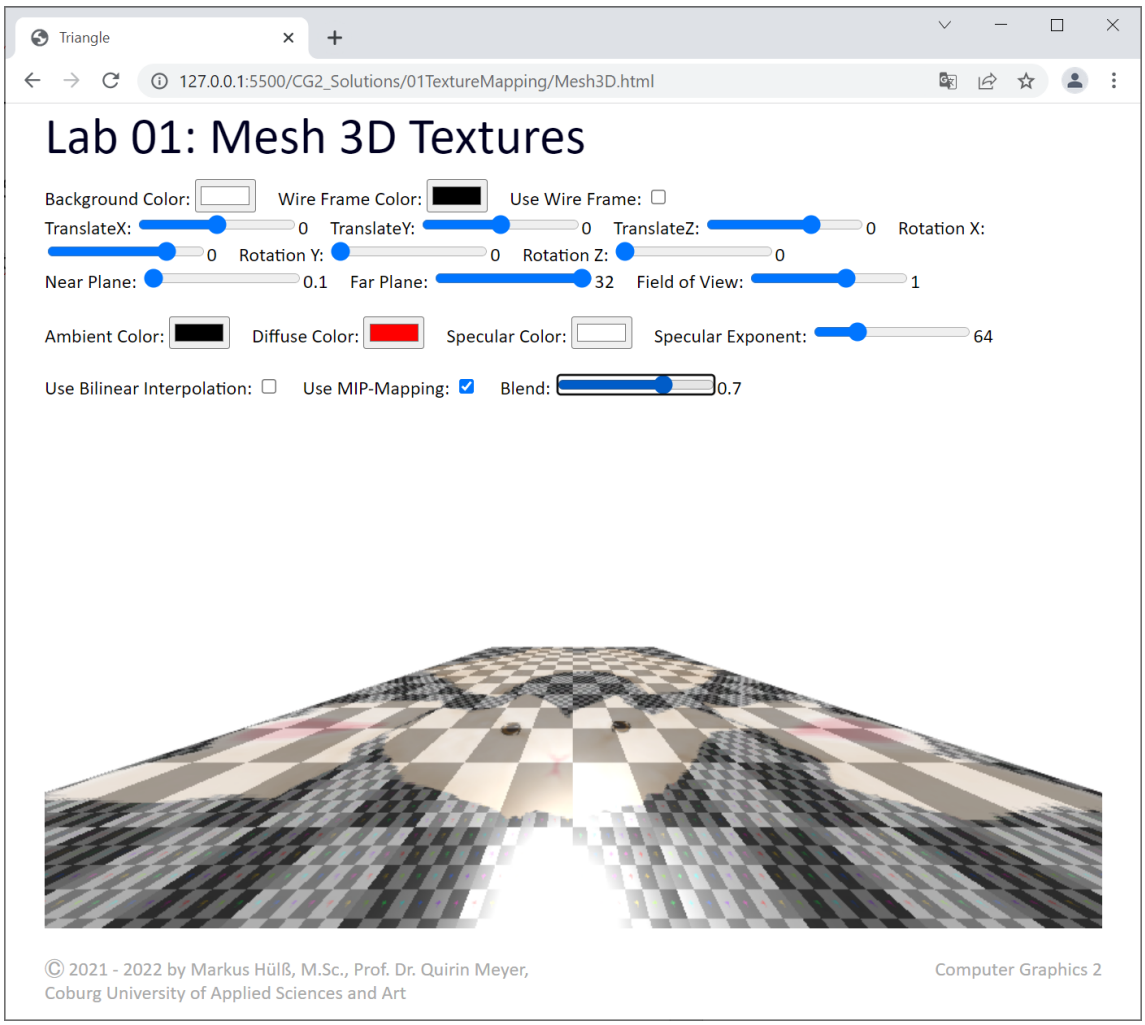
```
1  mGlslProgram.setUniform1i("u_sampler", texture.getUnit());
```

die Textur-Unit jedes Samplers (also von Textur A und Textur B) so, dass Sie von beiden Texturen im Shader samplen können. Sie müssen dazu die Klasse GLSLProgram um die Funktion `setUniform1i` erweitern.

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 laden, 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:

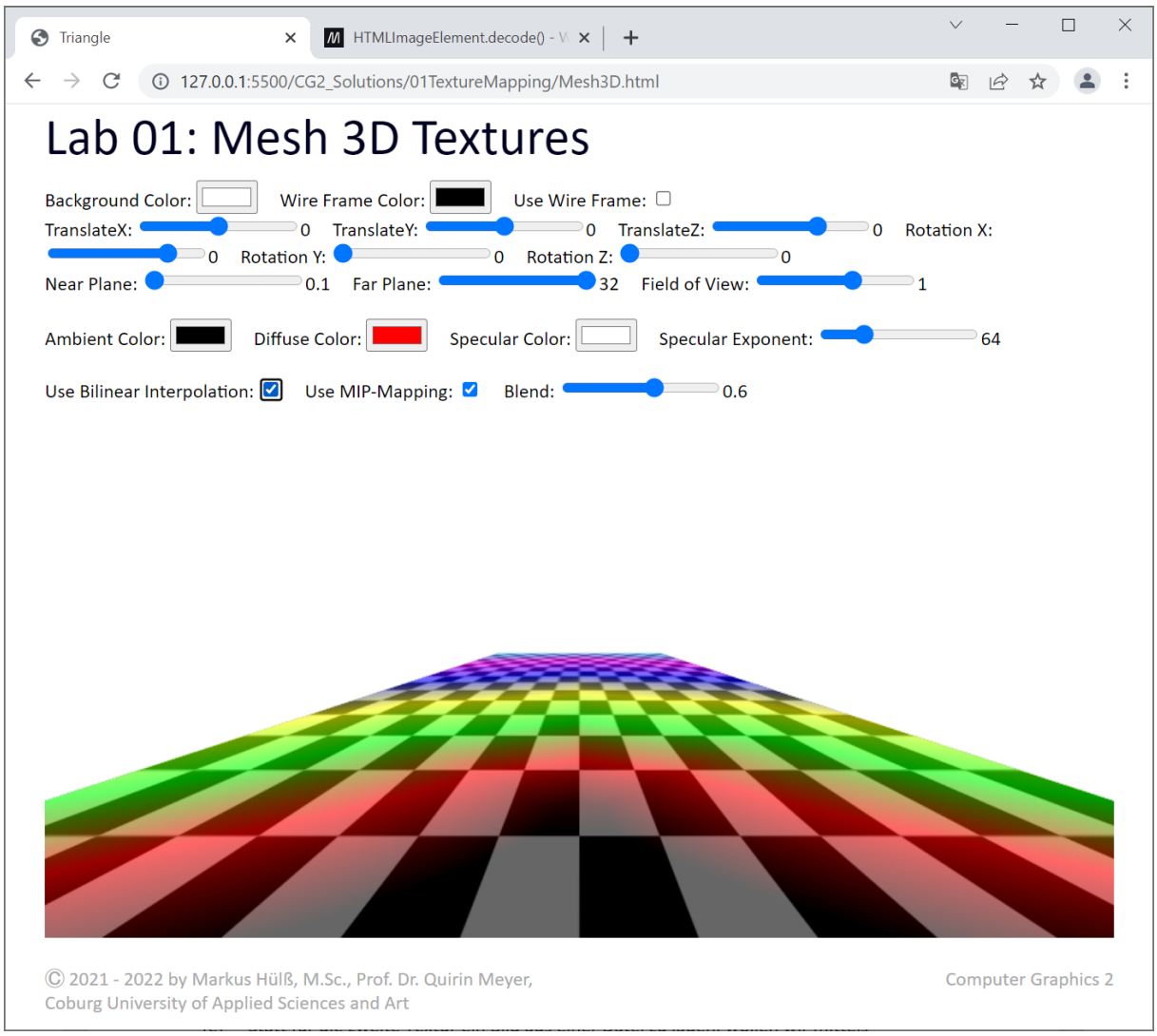


Abbildung 10: Multi-texturing um die MIP-Map Stufen zu visualisieren. Hier wird trilinear interpoliert …

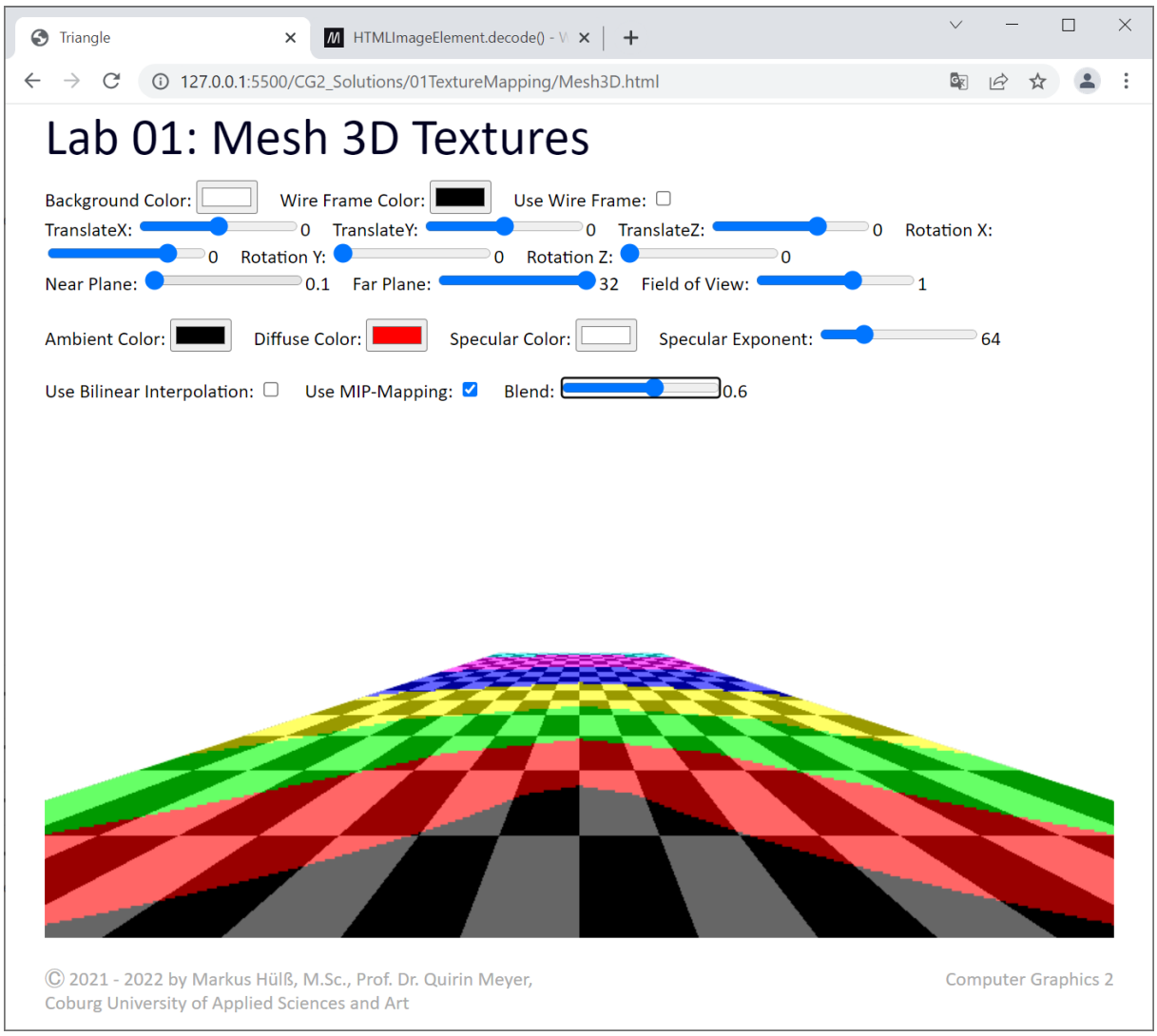


Abbildung 11: … und hier nur bilinear interpoliert.

Hinweise:

- Das maximale MIP-Map Level ist $\log_2 w$ wobei w die Breite (oder auch Höhe) der Texture mit der höchsten Auflösung ist.
- MIP-Map Level bezeichnet die Texture mit der höchsten Auflösung.
- Mittels gl.texImage2D kann man Pixeldaten für jedes MIP-Map Stufe individuell hochladen (siehe WebGL Dokumentation!)
- Die Farbe für MIP-Map Level l könnte z.B.

$$\vec{c}\,(l) = \begin{bmatrix} (l \wedge 1_2) \cdot 255 \\ ((l \wedge 10_2) \gg 1) \cdot 255 \\ ((l \wedge 100_2) \gg 2) \cdot 255 \end{bmatrix}$$

sein, wobei $\wedge$ die binäre UND-Operation und $\gg$ die binäre Rechtsschiebe-Operation darstellt.

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