Praktikum 2: Mesh3D

In diesem Praktikum werden wir ein 3D Netz mittels WebGL2, GLSL zeichnen und JavaScript zeichnen. Dazu müssen Sie ständig die Dokumentationen bereit halten und selber lesen. Ansonsten schaffen Sie es nicht! Diese finden Sie unter:

- WebGL API
- WebGL
- WebGL2
- GLSL

Wir setzen auch für die Prüfungen voraus, dass Sie die WebGL-Funktionen vertraut sind! Deshalb lohnt es sich, die Dokumentation von den hier verwendeten Funktion zu lesen, zu verstehen, eigenständige anzuwenden und zu erklären! Das Praktikum wird oder wird nicht Gegenstand der Prüfung sein!

Um zur passenden Code-Stelle zu kommen, suchen Sie einfach nach “Assignment” gefolgt von der Aufgabennummer (z.B. “Assignment 2e”).

1 Der Hase ohne Projektion

Der `constructor` der Klasse `TriangleMeshGL` bekommt als Parameter ein Objekt der Klasse `SimpleMeshModelIO`. `SimpleMeshModelIO` verwaltet die Daten des Meshes auf der CPU. Das Mesh besteht aus:

```
this.indices = indices;     // 3 consecutive integers make a triangle
this.positions = positions; // 3 consecutive floats make a 3d position
this.colors = colors;       // 3 consecutive floats make an RGB-color
this.normals = normals;     // 3 consecutive floats make a 3d normal
this.texCoords = texCoords; // 2 consecutive floats make a 2d tex coord
    .
```

Dies können Sie der Datei `lib/js/SimpleMeshModelIO.js` entnehmen.

Die Aufgabe von `TriangleMeshGL` ist es

1. diese Daten von der CPU auf die GPU hochzuladen und
2. das Mesh zu zeichnen.

Die Klasse soll jedoch **nicht** folgende Aufgaben erfüllen:

1. Hintergrundfarbe setzen,
2. Bildschirm löschen,

3. Shader binden oder
4. uniforme Variablen an den Shader weiterleiten.

a) Erzeugen und binden Sie zuerst ein Vertex-Array Objekt! Verwenden Sie dazu die Methoden `gl.createVertexArray` und `gl.bindVertexArray`. Das Vertex-Array Objekt soll in `this.vao` gespeichert werden.

b) Erzeugen Sie einen Buffer in WebGL für die Positionen, binden Sie diesen und laden die Positionen von der CPU auf die GPU hoch! Benutzen Sie dazu die Methoden `gl.createBuffer`, `gl.bindBuffer` und `gl.bufferData`. Verwenden Sie die JavaScript Klasse `Float32Array` um sicherzustellen, dass 32-Bit Floating-Point-Zahlen beim Hochladen an die GPU verwendet werden!

c) Um WebGL die Semantik der Daten mitzuteilen, benutzen Sie die Methoden `gl.vertexAttribPointer`. Aktivieren Sie mit `gl.enableVertexAttribArray` das Attribute. Die Attribute Location für die Positionen ist in der Konstanten `positionAttributeLocation` hinterlegt.

Achtung: Sie können nicht eins-zu-eins die Parameter aus der Vorlesung übernehmen! Sie müssen also vorher noch etwas nachdenken!

d) Erzeugen Sie einen Index Buffer auf der GPU und binden Sie diesen an Ihr Vertex-Array Objekt! Verwenden Sie dazu die Methoden: `gl.createBuffer`, `gl.bindBuffer` und `gl.bufferData`. Verwenden Sie die JavaScript Klasse `Uint32Array` um sicherzustellen, dass 32-Bit Unsigned-Integer-Zahlen beim Hochladen an die GPU verwendet werden!

e) Zum Zeichnen müssen Sie die Methode `TriangleMeshGL.draw` implementieren! Verwenden Sie dazu die Funktionen `gl.bindBuffer` und `gl.drawElements`.

f) Das Mesh wird in den Arbeitsspeicher der CPU asynchron in der Methode mittels

```
1  const meshCPU = await SimpleMeshModelIO.load(await
       loadBinaryDataStreamFromURL("./../../data/bunny.smm"));
```

geladen. Legen Sie nach dem Laden ein Objekt der Klasse `TriangleMeshGL` an und speichern Sie es in `this.triangleMeshGL`!

g) Definieren Sie in `mesh3d.vert.glsl` ein Attribute `a_position` vom Typ `vec3`, welches in der Lage ist, den GPU-Positions-Buffer zu lesen!

h) Schreiben Sie in `gl_Position` die `a_position`. Allerdings ist `gl_Position` vom Typ `vec4` und `a_position` vom Typ `vec3`. Nutzen Sie zur Konvertierung einen der GLSL Konstruktor von `vec4` und fügen Sie die vierte Komponente hinzu. Welchen Wert sollte diese vierte Komponente bei Punkten haben?

i) Zeichen Sie nun in `Mesh3DApp.draw` das WebGL Mesh! Sie sollten folgendes Ergebnis erhalten:

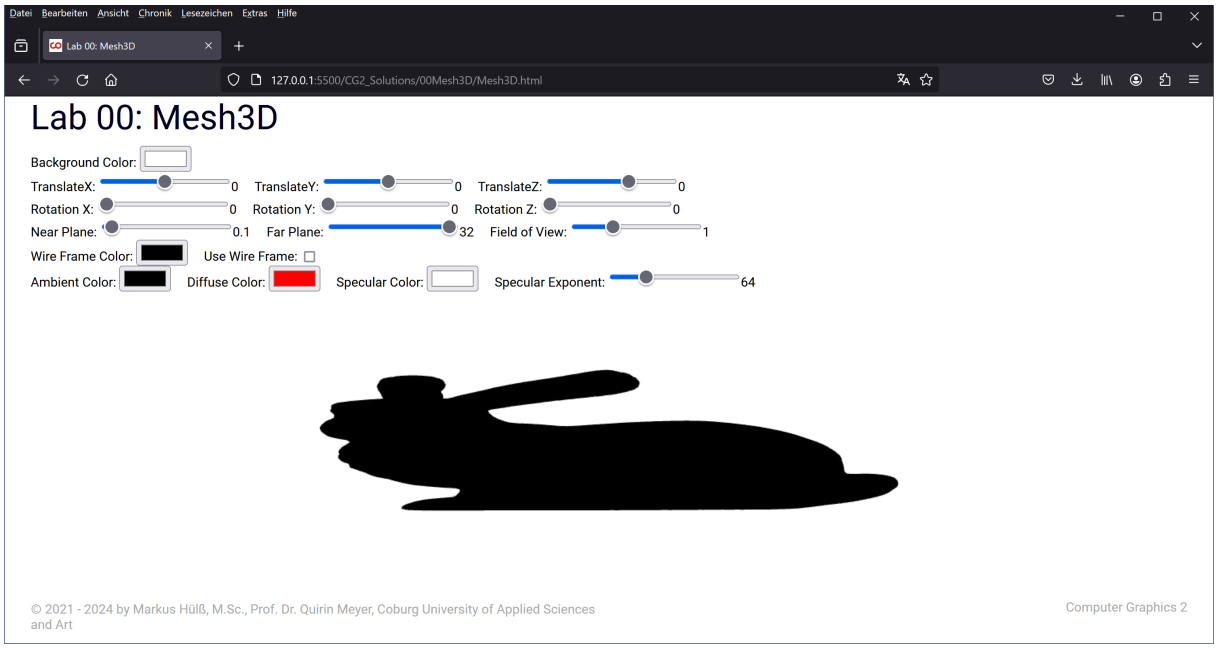

Abbildung 1: Der Hase ohne Farbe

2 Fragen

a) Erklären Sie den Unterschied zwischen Anzahl der Dreiecke und Anzahl der Indizes! Welchen Parameter müssen Sie `gl.drawElements` mitgeben?

b) Welche Funktionen machen von `gl.ARRAY_BUFFER` und `gl.ELEMENT_ARRAY_BUFFER` Gebrauch? Wann nimmt man welchen Parameter her?

c) In der `gl.drawElements` ist einer der Parameter `this.gl.UNSIGNED_INT`. Was bedeutet dieser Parameter? Welche anderen Parameter kann man noch nehmen und was bewirken diese? Welche Vor- und Nachteile ergeben sich dadurch?

d) Welche anderen Methoden außer `gl.drawElements` gibt es? Erklären Sie einen davon und erläutern Sie die Unterschiede sowie Vor- und Nachteile gegenüber `gl.drawElements`!

3 Der Hase mit Projektion

Nun wollen wir das Modell perspektivisch projizieren.

a) Machen Sie sich mit der Klasse `Matrix4` vertraut! Was tut diese Klasse? In welchem Speicher-Layout speichert die Klasse Matrizen?

b) Direkt in der `Mesh3DApp.draw`-Methode werden UI Parameter in passende lokale Variablen gelesen. Verwenden Sie nun `nearPlaneDistance`, `farPlaneDistance` und `fieldOfViewRadians` aus dem UI um den Aufruf `Matrix4.perspective` zu konfigurieren. Diese Methode erzeugt eine 4x4 Matrix, gespeichert als 16-elementiges Array, mit den entsprechende Einträgen, die eine Projektionsmatrix benötigt. Einzig der Parameter `aspectRatio`, also das Verhältnis aus Breite zu Höhe der WebGL Zeichenfläche, ist noch von Ihnen auszurechnen. Die restlichen Parameter erhalten Sie vom UI. Legen Sie die Matrix in der lokalen Variable `projectionMatrix` ab.

c) Deklarieren Sie in `mesh3d.vert.glsl` eine `uniform` Variable names `u_mvp` (mvp steht für **M**odel-**V**iew-**P**rojection.).

d) Transformieren Sie `a_position` mit der `u_mvp` Matrix mittels Matrix-Vektor Multiplikation und speichern Sie das Ergebnis in der Clip-Space Ausgabe `gl_Position`. In GLSL geht die Matrix-Vektor Multiplikation sehr elegant, denn die Operatoren sind bereits alle überladen! Das könne so Sprache wie JavaScript in Java halt nicht!

e) Befassen Sie sich mit der Java-Script Klasse `GLSLProgram`! Was macht diese Klasse? Wo wird Sie in `Mesh3DApp` instanziiert? Welche Shader-Files werden dort verwendet?

f) Implementieren Sie in `GLSLProgram.js` die Methode `setUniformMatrix4f`, welche 4x4 Matrizen, die im row-major Format abgelegt sind, an das Shader-Programm übergibt. Nutzen Sie dazu `getUniformLocation` und `this.gl.uniformMatrix4fv`. Welchen Wert muss der Parameter `transpose` haben damit unsere row-major Matrizen korrekt von der CPU and die GPU übertragen werden?

g) Übermitteln Sie nun die `projectionMatrix` von der CPU an die Shader-Variable `u_mvp`. Nutzen Sie dazu die eben implementierte Methode `setUniformMatrix4f`! Sie sollten dann folgendes Ergebnis erhalten:

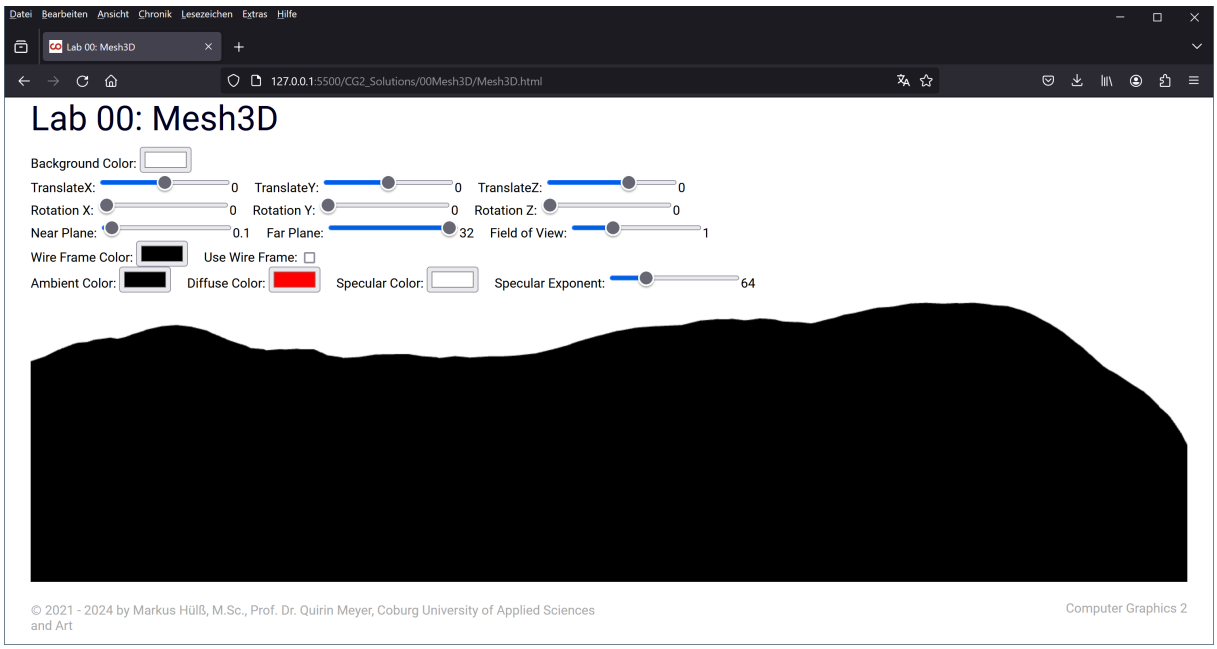

Abbildung 2: Der Hase ohne Farbe

4 Fragen

a) Was ist der Unterschied zwischen einer row-major und einer column-major Speicher-Layout für Matrizen!

b) Welche Matrix-Operatoren sind in GLSL implementiert?

5 Der Hase mit affinen Transformationen

a) Machen Sie sich mit der Klasse `Vec3` vertraut!

b) Nutzen Sie den `Vec3 translation`, welche die Werte der UI-Slider “TranslateX”, “TranslateY” und “TranslateZ” und bestimmen Sie eine 4x4 Translation0Matrix. Sie können dabei eine geeignete Methode aus der Klasse `Matrix4` verwenden!

c) Erzeugen Sie auch die Rotationmatrizen um die X-, Y- und Z-Achse aus dem `Vec3D rotation`.

d) Nutzen Sie `Matrix4.multiply` um die Matrizen zu einer Model-View-Matrix zusammen zu multiplizieren. Dabei soll erst um die Z-Achse, dann um die Y-Achse, dann um die X-

Achse rotiert werden. Am Schluss soll das Model auch noch verschoben! Speichern Sie das Ergebnis in der lokalen Variable `modelViewMatrix` ab!

e) Berechnen Sie nun aus der Model-View-Matrix und der Projection-Matrix die Model-View-Projection Matrix. Speichern Sie das Ergebnis der lokalen Variable `modelViewProjectionMatrix` ab!

f) Übergeben Sie dem Shader in `u_mvp` nun die Model-View-Projection Matrix!

Dann ist der Hase perspektivisch korrekt in Szenen gesetzt:

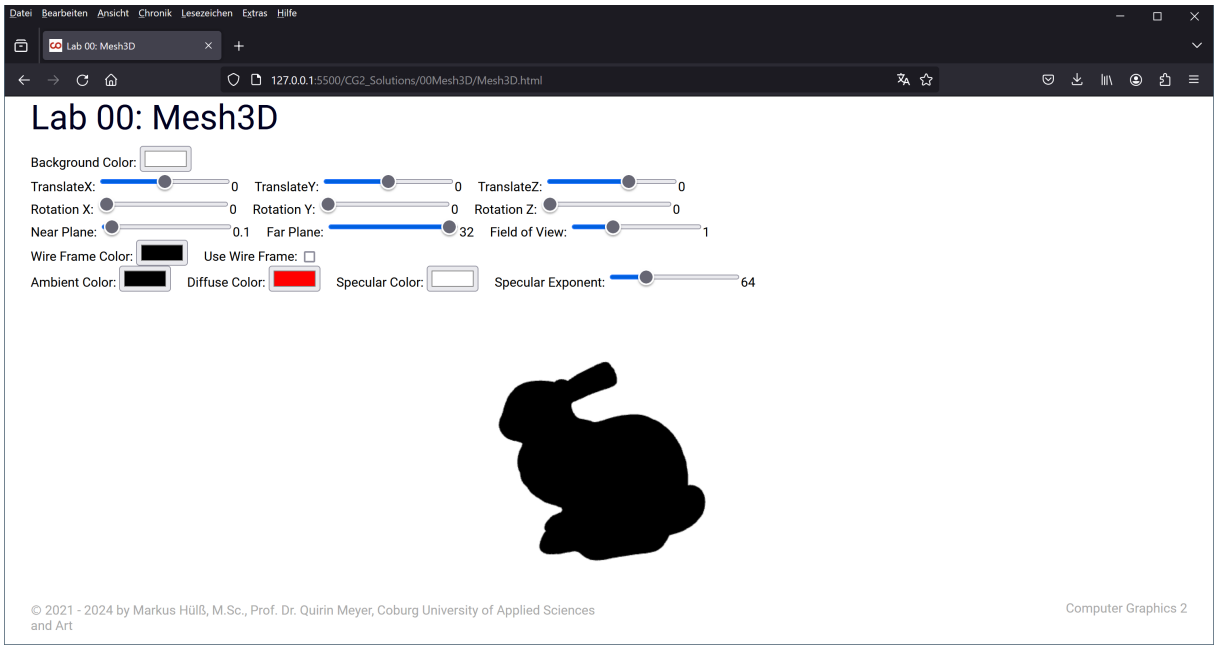

Abbildung 3: Der Hase mit Translation, Rotation und perspektische Projection

6 Normalen Vektoren

a) Fügen Sie im `constructor` von `TriangleMeshGL` nun die Normalen-Vektoren als WebGL Buffer an das Vertex-Array-Objekt hinzu. Laden Sie dazu das CPU Array `normals` auf die GPU hoch. Verwenden Sie die Attribute-Location `normalAttributeLocation`!

b) Fügen Sie im Vertex-Shader ein per-Vertex Attribute `a_normal` für die Normalen hinzu!

c) Leiten Sie das Vertex-Attribute vom Vertex-Shader an den Fragment-Shader weiter! Nennen Sie diese Variable `fs_normal`

d) Geben Sie den Absolutbetrag der Normalen als `fragColor` im Fragment Shader weiter! Sie sollten folgendes Ergebnis erhalten:

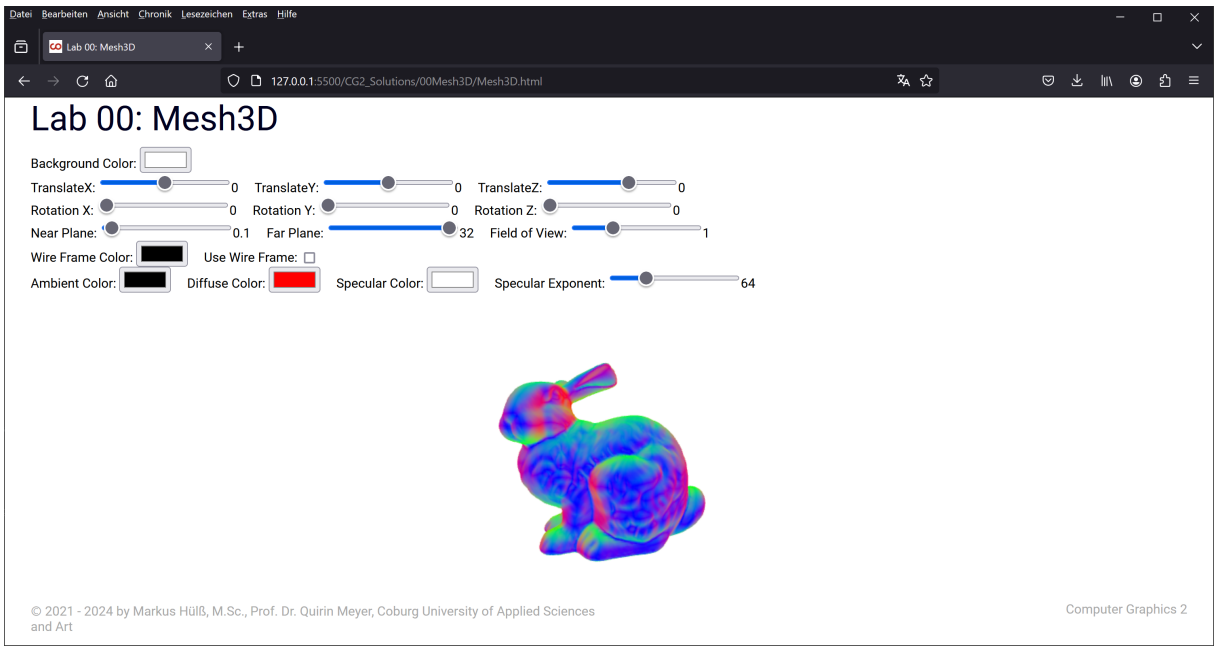

Abbildung 4: Die Normalen des Hasens als Absolutbeträge

7 Normalen und Position den Camera-Space transformieren

Um das Blinn-Phong Beleuchtungsmodell zu implementieren, müssen Positionen und Normalen mit der Model-View Matrix multipliziert werden. Diese Multiplikation findet im Vertex-Shader statt.

a) Reichen Sie die Model-View Matrix an der Vertex-Shader weiter!

b) Transformieren Sie Position und Normalen in den Camera-Space mit Hilfe der Model-View Matrix und leiten diese vom Vertex-Shader an den Fragment-Shader weiter. Nennen Sie die Variable für die Camera-Space Position `vec3 fs_position`!

c) Geben Sie zum Testen den Absolutbetrag der transformierte Position als Pixelfarbe aus:

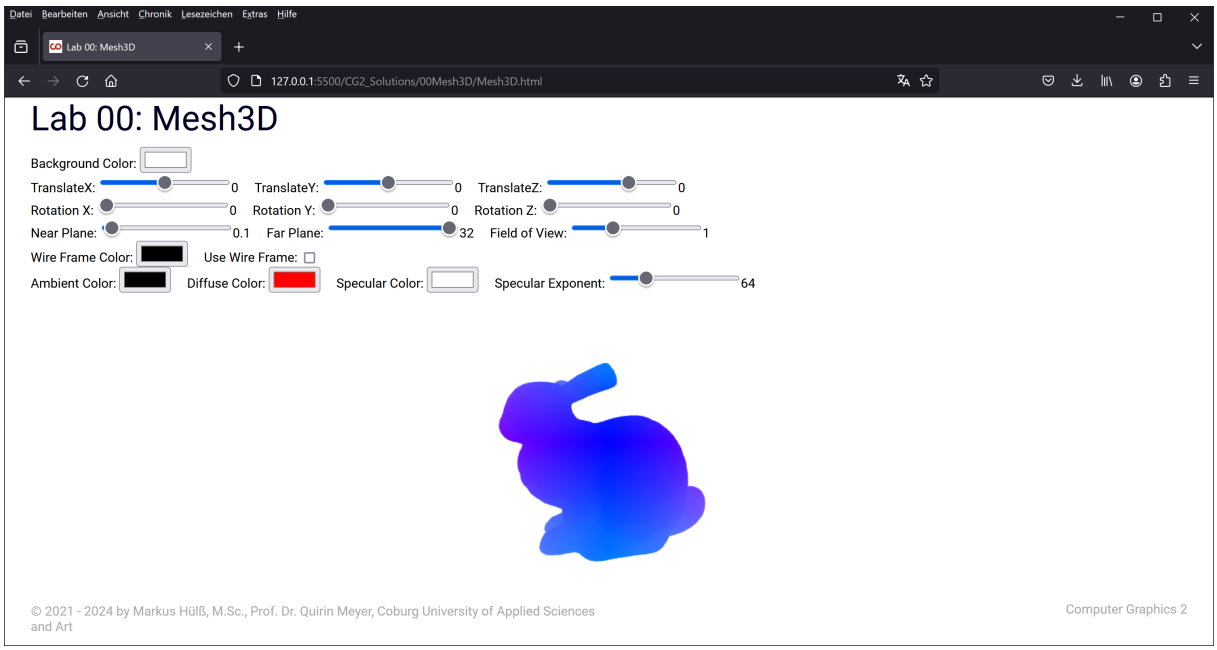

Abbildung 5: Die Position im Camera-Space des Hasens als Absolutbeträge

d) Geben Sie zum Testen den Absolutbetrag der transformierte Normale als Pixelfarbe aus:

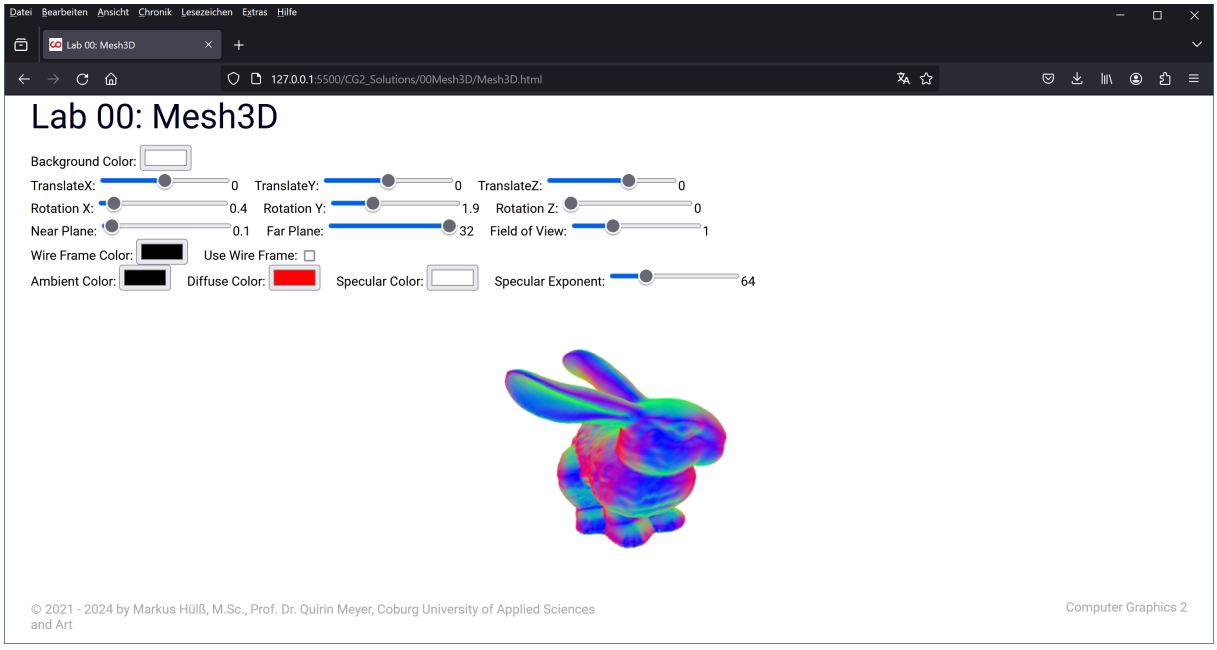

Abbildung 6: Die Normalen im Camera-Space des Hasens als Absolutbeträge

8 Blinn-Phong Beleuchtungsmodell berechnen

Das Blinn-Phong Beleuchtungsmodell benötigt ein paar Parameter. Diese werden in der `Mesh3DApp.draw()`-Methode bereits in lokale Variablen gelesen (`ambientColor`, `diffuseColor`, `specularColor` und `specularExponent`).

a) Implementieren Sie zunächst die Funktionen `setUniform3f` und `setUniform1f`!

b) Übergeben Sie UI-Variablen, die für das Blinn-Phong Beleuchtungsmodell benötigt werden an den Fragment-Shader mittels dieser beiden eben implementierten Funktionen. Definieren Sie dazu geeignete uniform Variablen im Fragment Shader!

c) Berechnen Sie die Farbe nach dem Blinn-Phong Belleuchtungsmodell. Sie sollten folgendes Ergebnis erhalten:

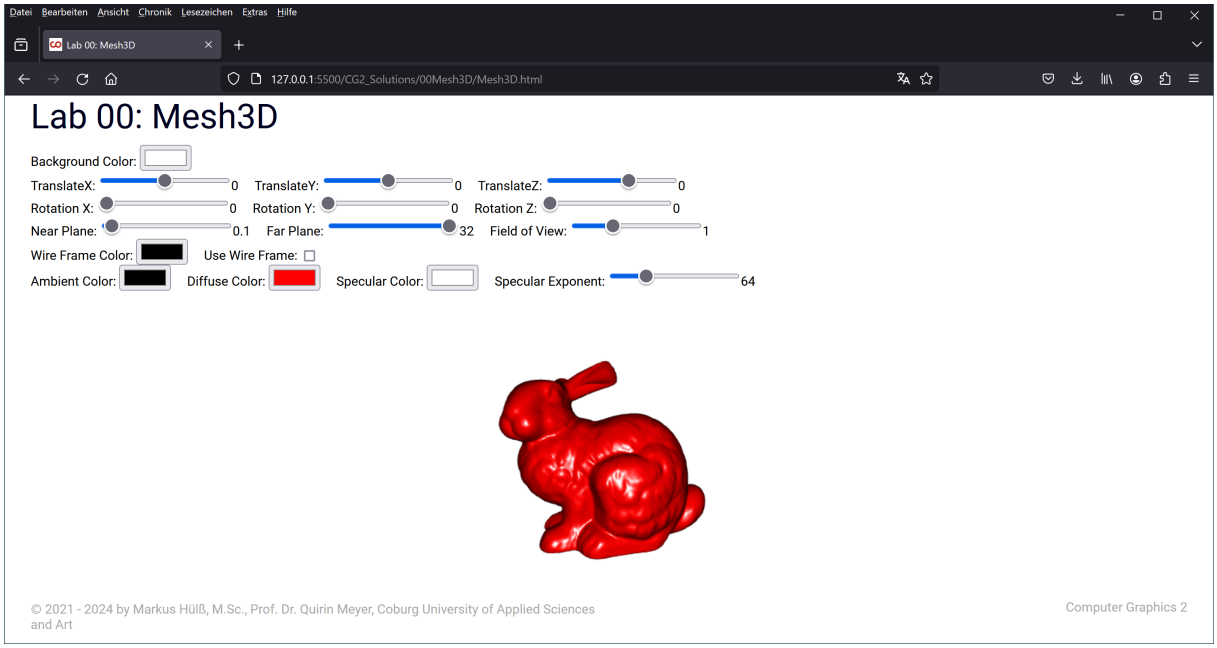

Abbildung 7: Der Hase mit Blinn-Phong Beleuchtung.