TH KÖLN
Prof. Dr. Florian Niebling
Advanced Media Institute

Technology
Arts Sciences
TH Köln

29. Mai 2024
Abgabe: spätestens am 21. Juni 2024

Computergrafik und Animation
Blatt 3 - Textures

In diesem Aufgabenblatt werden Sie sich mit Texturen beschäftigen. Hierfür werden Sie zum ersten Mal aktiv an den Shadern (Vertex- und Fragment-Shader) arbeiten, um Farben der Fragmente auf Farben aus einer geladenen Textur zu setzen. Schauen Sie sich hierfür die Vorlesungsinhalte zur Texturierung und den Aufgaben des Vertex-/Fragment-Shaders an. Sie werden allerdings bis zum Ende von Aufgabe 3.5 kein Zwischenergebnis erhalten.

3.1 Texture2D

Um Texturen zu verarbeiten bedarf es einiger Operationen, welche in der Klasse `Texture2D` des Packages `cga.exercise.components.texture` umgesetzt werden sollen. Diese Klasse implementiert das Interface `ITexture`. In dieser Klasse sind der Init-Block und das Companion-Objekt bereits gegeben. Ihre Aufgabe wird es nun sein, die Verarbeitung der Daten in der Methode `processTexture(...)` zu implementieren. Im Anschluss implementieren und verwenden Sie bitte die Methode `setTexParams(...)` um die Parameter der gerade aktiven Textur zu setzen. Hierzu verwenden Sie die OpenGL Funktion glTexParameteri[1]. Es soll definiert werden, was passiert, wenn das Ende der Textur in horizontaler und vertikaler Richtung erreicht ist (Texture Wrapping) und wie mit Texture Magnification bzw. Minification umgegangen werden soll (Texture Filtering - Inhalt der Vorlesung Textures). Nachdem dies erledigt ist, müssen Sie die Methoden `bind(...)` und `unbind()` vervollständigen. Nehmen Sie hierzu die Vorlesungsunterlagen zur Hand.

3.2 Material

Die Klasse `Material` im Package `cga.exercise.components.geometry` verwaltet die für die Darstellung notwendigen Texturen sowie die verwendeten TextureUnits. Dort sind insgesamt drei unterschiedliche Texturen als Eigenschaften hinterlegt — diffuse, spekulare und emissive Textur. Diese sind für die kommende Aufgabe zur Beleuchtung relevant. Die einzelnen Texturarten werden im Aufgabenblatt 4 näher beschrieben, ebenso die Eigenschaft `shininess`. Der `tcMultiplier` gibt an, wie häufig eine Textur auf einem Objekt wiederholt wird. Die Konstruktoren sind auch hier wieder gegeben.
Ihre Aufgabe nun besteht darin, dass in `bind(...)` die emissive Textur des Materials sowie der `tcMultiplier` an die Shader mittels Uniform-Variablen übermittelt werden. Beachten Sie die Datentypen. Die jeweilige TextureUnit wird im Shader als `sampler2D` erwartet, kann aber einfach als Integer hochgeladen werden (startend bei 0). Hierfür müssen Sie eine weitere Methode im ShaderProgram integrieren: `setUniform(String name, int value){...}`.

3.3 Shader-Erweiterung

Die im Vorfeld erzeugten Materialeigenschaften müssen noch in den Shadern entgegengenommen und verwendet werden. Im Vertex-Shader muss dafür die Uniform `tcMultiplier` angelegt und mit der an den Fragment-Shader übergebenen `textureCoordinate` multipliziert werden (falls noch nicht geschehen, integrieren Sie die `textureCoordinates` der Objekte jetzt und leiten Sie diese an den Fragment-Shader weiter). Machen Sie sich auch darüber Gedanken, warum diese Multiplikation den gewünschten Effekt hat.

[1] `https://javadoc.lwjgl.org/org/lwjgl/opengl/GL11.html#glTexParameteri(int,int,int)`

Im Fragment-Shader werden ferner `sampler2D` für die einzelnen Texturen als Uniforms benötigt, um einen Texture-Lookup zu gewährleisten. Dieser Texture-Lookup funktioniert mithilfe der Methode `texture(sampler2D tex, vec2 tc)`[2] und liefert den Farbwert der Textur `tex` an der Stelle der Texturkoordinate `tc` als 4-dimensionalen Vektor.
Verwenden Sie für den Anfang nur die emissive Textur als Farbwert.

3.4 Mesh

Da jetzt die Materialien verarbeitet werden können, sollten diese natürlich auch mit dem Mesh verbunden werden. Hierfür benötigen Sie eine private Eigenschaft des `Mesh`, welche das Material hält. Erweitern Sie den Konstruktor um ein optionales Material.

Um die Texturen beim Rendern der Meshes zu nutzen, muss das Material in der Klasse `Mesh` an das aktuelle `ShaderProgram` gebunden werden. Da dies bisher nicht möglich ist, weil keine Referenz auf das `ShaderProgram` existiert, müssen Sie eine weitere `render()`-Methode schreiben, welche als Parameter ein `ShaderProgram` entgegennimmt. Binden Sie das Material, falls vorhanden, und rendern im Anschluss wie gewohnt.

3.5 Scene

Abschließend müssen Sie das Material für den Boden erstellen und dem Renderable zuweisen. Im Ordner `assets/textures` finden Sie die benötigten Bilder, welche als Textur genutzt werden. Orientieren Sie sich an dem Dateinamen bzgl. der Zuordnung zu den Texture2D-Eigenschaften des Materials. Wählen Sie als Wert für die `shininess` $60.0f$ und für den `tcMultiplier` den `Vector2f(64.0f, 64.0f)`. Zur Definition der einzelnen Texturen gehört direkt im Anschluss das Setzen der Textur-Parameter mittels `setTexParams(...)`. Experimentieren Sie hier einmal mit den unterschiedlichen Werten für das Wrapping und das Filtering.
Löschen Sie die Sphere, da diese kein Material besitzt und folglich nicht erstellt werden kann. Wenn Sie alles richtig implementiert haben, sollten Sie ein ähnliches Ergebnis sehen, wie Abbildung 1 zeigt.

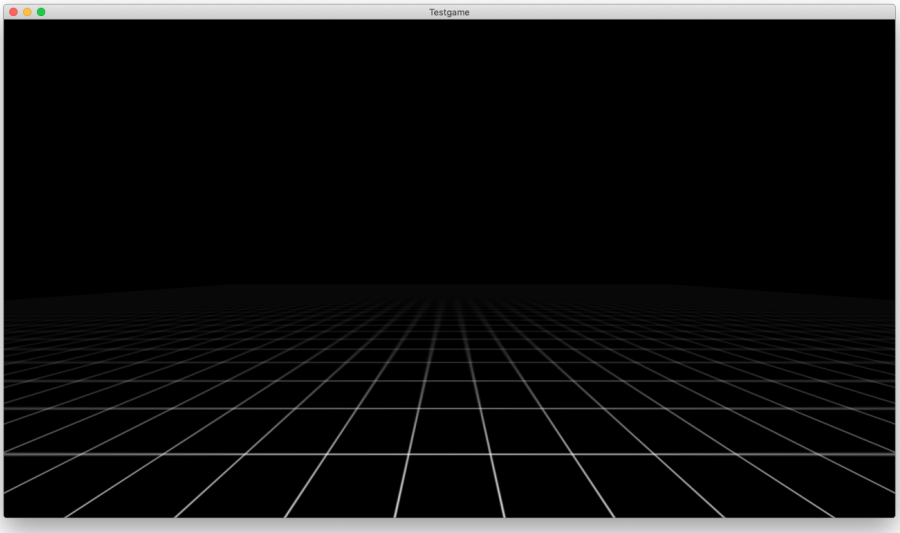

Abbildung 1: Ergebnis nach Aufgabe 3.5

Die Textur wird in Richtung des Horizontes sehr verwaschen. Dies liegt daran, dass eine Interpolation zwischen mehreren Texeln pro Pixel stattfindet (Stichworte: Mini- und Magnification). Um hierbei den Schärfeeindruck zu bewahren, können Sie in der Methode `Texture2D.setTexParams(...)` im Anschluss

[2] `https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/texture.xhtml`

an die Wrapping- und Filterdefinitionen folgende Zeilen hinzufügen:

```
GL11.glTexParameterf(GL11.GL_TEXTURE_2D,
  EXTTextureFilterAnisotropic.GL_TEXTURE_MAX_ANISOTROPY_EXT,
  16.0f)
```
[3]

3.6 Motorrad

Die zuvor gelöschte Kugel war eher unspektakulär, weshalb Sie nun ein komplexeres Modell in die Szene integrieren sollen. Hierfür reicht allerdings der bisher verwendete OBJLoader nicht aus, da das Modell des Motorrades bereits Informationen zu den Materialien der Meshes des Gesamtobjektes besitzt. Hierfür ist der mitgelieferte `ModelLoader` im Package `cga.framework` zu nutzen. Die statische Methode `loadModel(objpath: String, pitch: Float, yaw: Float, roll: Float)` sorgt dafür, dass alle notwendigen Objekte sowie Referenzen erstellt und entsprechend der Angaben rotiert werden. Die Methode liefert ein `Renderable`-Objekt zurück. Bitte laden Sie das Model `Light Cycle/HQ_Movie cycle.obj` ein und rotieren es um $-90°$ um die x-Achse und um $90°$ um die y-Achse. Skalieren Sie das Renderable im Anschluss um den Faktor $0.8f$. Integrieren Sie das Motorrad in den Render-Call. Setzen Sie außerdem das Motorrad als Parent der Kamera. Die Bewegungen, welche zuvor auf die Sphere gewirkt haben, sollen sich ab jetzt auf das Motorrad auswirken.
Wenn Sie die Szenerie jetzt rendern, sollten Sie das Motorrad von hinten sehen. Um das ganze etwas schöner darzustellen können Sie die Kamera ein wenig umpositionieren. Abbildung 2 nutzt für die Kamera hierbei eine Translation um `Vector3f(0.0f, 0.0f, 4.0f)` und anschließend eine Rotation um die x-Achse um einen Winkel von $-35°$.

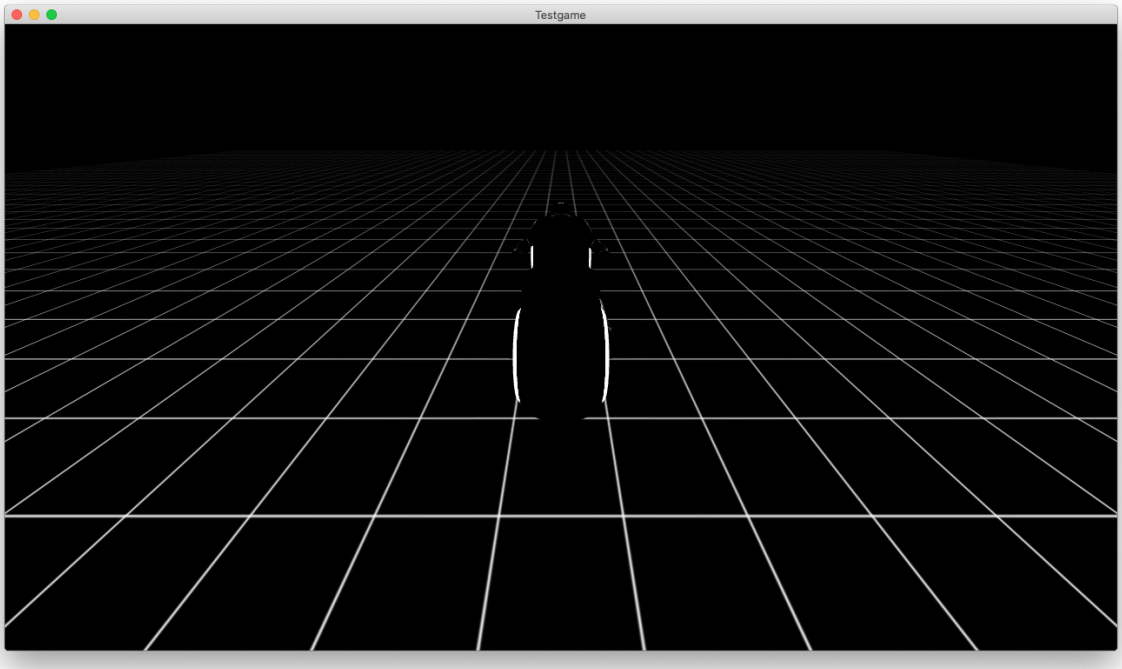

Abbildung 2: Motorrad von hinten

3.7 Motorrad-Bewegung: Kurvenfahrt

Um eine realistische Kurvenfahrt zu ermöglichen, passen Sie die `update()`-Methode der `Scene` entsprechend für die Tasten A und D an. Hierbei soll das Motorrad nicht auf der Stelle rotieren, sondern in der Vorwärtsbewegung.

[3] `https://javadoc.lwjgl.org/org/lwjgl/opengl/GL11.html#glTexParameterf(int,int,float)`