TH KÖLN
Prof. Dr. Florian Niebling
Advanced Media Institute

Technology
Arts Sciences
TH Köln

08. Mai 2024
Abgabe: spätestens am 07. Juni 2024

Computergrafik und Animation
Blatt 2 - Transformations

In diesem Aufgabenblatt werden Sie sich mit 3D-Transformationen beschäftigen. Unter Transformationen werden unter anderem Translationen, Rotationen und Skalierungen verstanden. Im Anschluss wird eine Kamera integriert, um die Szene flexibel betrachten zu können. Beiliegend zu dem Aufgabenblatt erhalten Sie drei Interfaces und zwei neue Shader. Integrieren Sie diese Dateien in Ihr Framework, beachten Sie dabei die vorliegende Ordnerstruktur. Die mitgelieferten Interfaces dürfen von Ihnen nicht verändert werden.

2.1 Transformationen

Im ersten Schritt der Implementierung des Tron-Spiels wird ein Spielfeld hinzugefügt. Bitte laden Sie hierfür das Objekt `ground.obj` auf die gleiche Weise wie die Sphere im vorherigen Aufgabenblatt. Starten Sie das Projekt, stellen Sie allerdings fest, dass Sie den Boden nicht sehen. Dies liegt an der Geometrie des Bodens. Dieser liegt vollständig in der x/z-Ebene und Sie schauen in dem Fall genau entlang dieser Ebene. Für den Anfang werden wir den Boden um die x-Achse drehen und skalieren, damit er sichtbar ist.

2.1.1 Object-to-World

Um vom Object-Space zum World-Space zu gelangen, wird die Modelmatrix (4×4-Matrix) benötigt. Diese vereinigt alle Transformationen, die auf das Objekt wirken, und platziert es mittels Multiplikation mit allen Vertices des Objektes im Weltkoordinatensystem.
Um diese Transformationen jetzt umsetzen zu können, erstellen Sie bitte für jedes Objekt (Boden und Kugel) eine eigene Variable vom Typ `Matrix4f`[1] der Bibliothek `org.joml` und wenden folgende Transformationen an:

- Boden
  - Rotation um 90° um die x-Achse
  - Skalierung um den Faktor 0.03
- Kugel
  - Skalierung um den Faktor 0.5

2.1.2 Uniforms

Die soeben erstellten Modelmatrizen müssen an die Grafikkarte übermittelt werden. Dies geschieht mit Hilfe von Uniform-Variablen, die von der CPU zur Laufzeit in den Shadern gesetzt werden können. Die aktuell verwendeten Shader sind allerdings noch nicht dafür ausgelegt. Verwenden Sie bitte ab jetzt die mitgelieferten Shader `tron_vert.glsl` und `tron_frag.glsl` aus dem `assets`-Ordner.
In dem Vertex-Shader finden Sie nun die `uniform`-Variable `mat4 model_matrix`, mit der in der `main()`-Methode die `gl_Position` berechnet wird, um die einzelnen Vertices im Weltkoordinatensystem zu platzieren.

[1] https://joml-ci.github.io/JOML/apidocs/org/joml/Matrix4f.html

Damit die Daten auch wirklich übermittelt werden, müssen die Matrizen in der `Scene.render()`-Methode mittels der Methoden `setUniform()` des genutzten `ShaderPrograms` gesetzt werden. Die hierfür notwendige Methode `setUniform(name: String, value: Matrix4f, transpose: Boolean)` fehlt jedoch noch und muss, analog zu anderen `setUniform()`-Methoden, von Ihnen erzeugt werden. Als kleiner Hinweis: die openGL-Methode `glUniformMatrix4fv(..)`[2] erwartet einen `FloatBuffer` [3] statt der JOML-Matrix. Diesen finden Sie bereits als Klassenattribut instanziert und müssen ihn nur noch vor der Übergabe an den Shader mittels `value.get(buffer)` mit Daten befüllen. Rendern Sie im Anschluss die Objekte. Wenn Sie nun alles richtig gemacht haben, dann sollten Sie nun ein ähnliches Ergebnis sehen wie in Abbildung 1.

2.1.3 Farbanpassung

Im nächsten Schritt sollen Sie den Absolutwert der Normalen jedes Vertex als seine Farbe ausgeben. Hierbei können die interpolierten Positionsdaten als Farbanteile (r, g, b) genutzt werden. Hierfür müssen Sie dafür sorgen, dass die Daten der Normalen jedes Objektes auch in den Shadern aus dem VAO entnommen und an der passenden Stelle verarbeitet werden. Vergegenwärtigen Sie sich hierbei die jeweiligen Aufgaben des Vertex- und Fragment-Shaders. Bedenken Sie zudem, dass durch die Interpolation der Vertex-Normalen für jedes Fragment deren Länge verändert werden kann. Normalisieren Sie daher die Normalen im Fragment-Shader vor der Ausgabe. Sollte sich der Boden bei Ihrer Ausgabe grün darstellen (und nicht wie in Abbildung 2 blau), vergegenwärtigen Sie sich, woran das liegen könnte. Welche Transformation ist nötig, um das Ergebnis zu erhalten?

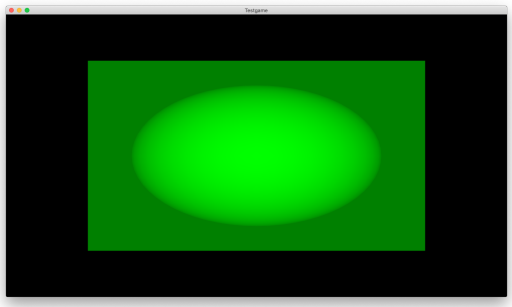

Abbildung 1: Ergebnis nach Aufgabe 2.1.2

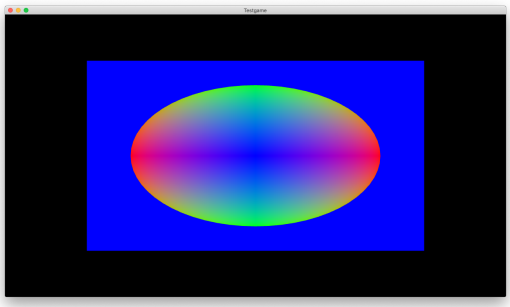

Abbildung 2: Ergebnis nach Aufgabe 2.1.3 mit den absoluten, normalisierten Werten der Normalen

2.2 Die Klasse Transformable

Die Model-Matrizen aller Elemente direkt in der Klasse `Scene` zu verwalten kann für komplexere Szenen schnell sehr unübersichtlich werden. Daher soll in dieser Aufgabe die Klasse `cga.exercise.components.geometry.Transformable` genutzt werden, in der zukünftig die Model-Matrizen der einzelnen Objekte gespeichert und transformiert werden können.

2.2.1 Transformationen in `Transformable`

Hierzu vervollständigen Sie bitte die Klasse `cga.exercise.components.geometry.Transformable` entsprechend der TODO Kommentare. Die Klasse dient im Folgenden als Elternklasse für alle Objekte die in den Szenegraph eingehängt werden sollen wie Renderable, Kamera usw.
Hinweis: Bei den Transformationen muss stets beachtet werden, in welcher Reihenfolge die Matrizen miteinander multipliziert werden müssen. Die "localXXX"-Transformationen bitte nicht nutzen. Die Dokumentation der Bibliothek liefert leider irreführende Informationen bzgl. dieser Methoden. Diese Metho-

[2] https://javadoc.lwjgl.org/org/lwjgl/opengl/GL20.html#glUniformMatrix4fv(int,boolean,java.nio.FloatBuffer)
[3] https://docs.oracle.com/javase/7/docs/api/java/nio/FloatBuffer.html

den sorgen für eine linksseitige Matrizenmultiplikation. Bauen Sie sich folglich besser die Matrizen selbst auf und multiplizieren sie eigenständig in der korrekten Reihenfolge für den jeweiligen Fall.

2.2.2 Szenegraph

Die Model-Matrizen in einer eigenen Klasse verwalten zu lassen ermöglicht eine elegante Implementierung des Szenegraphen. Hierfür muss jedes `Transformable` eine Referenz auf den eigenen Elternknoten (`parent`) des Szenegraphen speichern, und diese rekursiv zur Berechnung der finalen Model-Matrizen nutzen. Auch hierfür stellt die Klasse bereits Methoden bereit. Bedenken Sie, dass nicht jedes Objekt einen Elternknoten besitzen muss.

2.2.3 Ausführen der Testklasse

In dem mitgelieferten Sourcecode ist eine Testklasse `TransformableTest.kt` enthalten. Diese soll dabei helfen, die Transformationen korrekt zu implementieren.

2.3 Die Klasse `Renderable`

Erstellen Sie als nächstes eine neue Klasse `cga.exercise.components.geometry.Renderable` im selben Package, wo auch die Transformable-Klasse platziert ist. Übergeben Sie dem Renderable eine `MutableList` von Meshes im Constructor, damit diese schließlich über das Renderable gerendert werden können.
Lassen Sie die Klasse von `Transformable` erben. Ab diesem Zeitpunkt hat jedes Renderable seine eigene Model-Matrix, welche für die Transformationen, die auf das jeweilige Objekt wirken sollen, genutzt wird. Ändern Sie den Typ der bisherigen Objekte in der `Scene` in ein Renderable.
Nutzen Sie für die Transformationen, die Sie zuvor im `Init`-Block der Klasse `Scene` auf die jeweiligen Objekte angewendet haben, nun die Methoden der Klasse `Transformable`. Beachten Sie hierbei, dass Sie die Mathe-Bibliothek von `org.joml` verwenden.

2.3.1 `Renderable.render(...)`

Da nun jedes Renderable die eigene Model-Matrix speichert, kann diese auch direkt in der `render()`-Methode des `Renderable` an die Shader übermittelt werden. Hierfür ist es allerdings notwendig, dass der `render()`-Methode eine Referenz auf das aktuelle `ShaderProgram` übergeben wird. Nutzen Sie für diesen Fall das Interface `IRenderable` und implementieren Sie die nun hinzugefügte Methode, um die Model-Matrix zu übermitteln und die einzelnen Meshes zu rendern. Wenn Sie alles richtig implementiert haben und das Programm weiterhin ausführbar ist, sollte keine Veränderung zu Abbildung 2 zu erkennen sein.

2.4 Kamera

In dieser Aufgabe soll eine Kamera simuliert werden, um die projizierte Ansicht der Szene beliebig anpassen zu können. Als Kameratyp implementieren Sie eine perspektivische third-person Kamera. Durch die Anwendung der kameratypischen Matrizen (View- und Projection-Matrix) ist die Szene nicht mehr auf die Dimension [-1 ... 1] in x- und y-Richtung beschränkt. Machen Sie sich mit den Aufgaben der beiden Matrizen vertraut.

2.4.1 `TronCamera`-Klasse

Erstellen Sie im Package `cga.exercise.components.camera` die Klasse `TronCamera`, welche von der Klasse `Transformable` erbt und das Interface `ICamera` nutzt. Implementieren Sie die vom Interface geforderten Methoden in der Klasse `TronCamera` und erstellen einen Konstruktor, der die folgenden Klassenattribute initialisiert:

- *Field of View* — vertikaler Öffnungswinkel der Kamera [90 Grad in Radiant, Angabe in Float]
- *Seitenverhältnis (aspect ratio)* — (indirekt) horizontaler Öffnungswinkel [16.0f/9.0f]

- *Near Plane* — Entfernung der Near Plane zur Kamera [0.1f, quasi direkt vor der fiktiven Linse]
- *Far Plane* — Entfernung der Far Plane zur Kamera [100.0f]

Die Angabe in Klammern hinter den jeweiligen Attributen sind gängige Werte und können als Default-Wert gesetzt werden. Sie können aber auch ein wenig damit experimentieren, um ihre Auswirkungen besser verstehen zu können.

In der Methode `calculateViewMatrix()` soll die ViewMatrix berechnet werden, hierfür kann die JOML-Methode `lookAt(...)` verwendet werden. In der Methode `calculateProjectionMatrix()` soll die ProjectionMatrix berechnet werden, hierfür kann die JOML-Methode `perspective(...)` verwendet werden. Diese beiden Methoden werden in den Vorlesungsfolien näher beleuchtet und hergeleitet. Die `bind()`-Methode soll die beiden berechneten Matrizen an einen Shader übergeben.

2.4.2 Kameraintegration

Um nun die Kamera nutzen zu können, muss zuerst ein Objekt der Klasse `TronCamera` angelegt, im `Init`-Block der `Scene` initialisiert und positioniert werden(Rotation um die x-Achse mit -20 Grad, lokale Translation um $deltaPos = (0.0f, 0.0f, 4.0f)$).
Damit die Kamera auch Auswirkungen auf das Render-Ergebnis hat, müssen die Matrizen (View- und Projection-Matrix) jeden Frame aktualisiert und die Ergebnisse von Ihnen wieder an die Shader geschickt werden. Dort müssen diese zur Berechnung der `gl_Position` hinzugezogen werden. Des Weiteren müssen die Normalen ebenfalls angepasst werden. Hierfür ist es notwendig, die Inverse der transponierten Model-View-Matrix zu verwenden. Erläuterungen dazu finden Sie auf `https://paroj.github.io/gltut/Illumination/Tut09%20Normal%20Transformation.html`.
Ab dem Zeitpunkt, wo die Kamera integriert ist, muss der Boden nicht mehr transformiert werden. Nehmen Sie folglich alle Transformationen, die auf den Boden und die Sphere wirken, raus. Das Ergebnis sollte nun aussehen wie es Abbildung 3 darstellt.

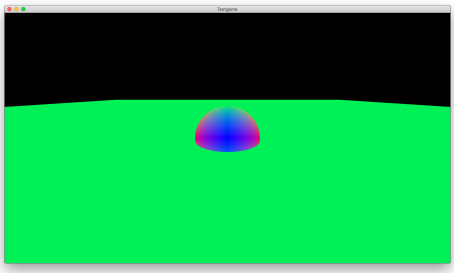

Abbildung 3: Ergebnis nach Integration der Kamera

2.5 Interaktion

Der nächste Schritt sieht die Integration einer Steuerung mittels der Tasten *WASD* vor. Hierfür soll in der Methode `Scene.update(...)` auf Tasteneingaben reagiert werden.
Das `GameWindow window` stellt für diesen Zweck die Methode `getKeyState(int key)` zur Verfügung. Die Methode liefert einen Boolean, ob die angefragte Taste (`key`) in diesem Moment gedrückt wird. Die Tasten können über die OpenGL-Konstanten (`GLFW_KEY_W`, `GLFW_KEY_A`, `GLFW_KEY_S`, `GLFW_KEY_D`) identifiziert werden.
Bei Eingabe der Tasten W oder S bewegt sich die Kugel nach vorne bzw. nach hinten. Bei Eingabe von A oder D soll sie lokal nach links (gegen den Uhrzeigersinn um die y-Achse) bzw. nach rechts (im Uhrzeigersinn um die y-Achse) rotieren. Skalieren Sie die jeweiligen Transformationen sinnvoll mit dem Parameter `dt`. Bewegen Sie mit diesen Tastatureingaben die Kugel entsprechend.
Abschließend nutzen Sie die Möglichkeit des Szenegraphen und definieren die Kamera als Child der Kugel. Haben Sie alles richtig gemacht, können Sie sich nun auch auf dem Spielfeld bewegen.