Hochschule Darmstadt
B. Frömmer, B. Meyer

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

03. November 2023

Visual Computing WiSe 2023/2024
Aufgabe 2

2.1 Auf in die dritte Dimension

Im Moodle finden Sie eine `Cube.h`-Datei, die Sie bitte in Ihr Projekt integrieren. Sie enthält einzig neue Vertex- und Index-Daten, um einen farbigen 3D-Würfel darzustellen.
Ersetzen Sie damit Ihre alte Geometrie (entweder das Haus aus dem 1. Praktikumsprojekt, oder Ihre selbstgebauten Initialen). Sie müssen Ihre Vertex Attribute Pointer entsprechend anpassen, damit die dreidimensionale Geometrie korrekt dargestellt wird. Zudem müssen Sie den Vertex-Shader aktualisieren, damit alle drei Koordinaten der Vertexposition verarbeitet werden. Sie finden die Shader im Ordner assets/shaders. Die angepassten Shader werden allerdings nur übernommen (und in das Build-Verzeichnis kopiert), wenn Sie das CMake-Projekt neu laden (unter CLion: Tools/CMake/Reload).
Wenn alles funktioniert, wird der Würfel in der Mitte des Fensters gerendert, allerdings frontal von einer Seite, also noch nicht wirklich 3D.

2.2 Transformationen

Um einen ersten dreidimensionalen Eindruck des Würfels zu erhalten, müssen wir ihn ein wenig drehen. Wie aus der Vorlesung bekannt sein sollte, werden alle Transformationen eines Objektes in einer akkumulierten Transformationsmatrix gespeichert. Unser Framework stellt hierfür die Klasse `Transform` bereit, die eine entsprechende Matrix verwaltet und bereits verschiedene Transformations-Methoden implementiert hat.
Erzeugen Sie für Ihren Würfel daher ein Transform-Objekt und rotieren es um zwei Achsen. Die `rotate`-Methode erwartet entweder einen Quaterionen zur Rotation (schwierig), oder einen dreidimensionalen Vektor (`glm::vec3(float x, float y, float z)`) mit drei Euler-Winkeln in Radiant, um die in X-, Y- und Z-Achse rotiert werden soll (welcher intern dann in einen Quaterionen transformiert wird).
Sie können Ihren Würfel entweder einmalig in der `Scene::init()` drehen, oder per Frame in der `Scene::render()`. Der Parameter `float dt` hilft dabei, die Drehgeschwindigkeit unabhängig von der jeweiligen Framerate zu definieren.

Die Definition der Transformation ist jedoch nur der erste Schritt. Sie muss noch in den Shader übertragen werden, um dort auf die einzelnen Vertices angewendet werden zu können.
Das Framework unterstützt bei diesem Datentransfer durch die Methode

```
Transform cubeTrans = new Transform;
...
m_shader->setUniform("nameOfVariable", cubeTrans->getMatrix() , false);
```

Die Variable "nameOfVariable" (bitte sinnvollen Namen einsetzen!) muss im Shader als `uniform` definiert werden und enthält danach die jeweilige Matrix des Transforms "cubeTrans". Nutzen Sie die Matrix, um die Vertex-Positionen entsprechend zu transformieren.

Anmerkung: In einer älteren Version des Frameworks war in der Transform.cpp ein Fehler unterlaufen, den es zu korrigieren gilt: die Rotationsmatrix muss in Zeile 16 mit `m_rotation(glm::vec3(0,0,0))` initialisiert werden, nicht mit `m_rotation()`.

2.3 Mein eigener Roboter

Wir wollen unseren Würfel nun nutzen, um einen einfachen Roboter zu bauen. Das Ergebnis kann wie in der rechten Abbildung aussehen. Hierbei soll der Roboter einen Rumpf, einen Kopf, zwei Beine, zwei Oberarme und zwei Unterarme besitzen. Jedes Körperteil muss eine eigene Transformationsmatrix besitzen, wir nutzen jedoch immer die gleiche Geometrie.
Insgesamt soll der Roboter in der Größe skalierbar sein, er soll bewegt werden können (um ihn später in einer beliebigen Szene zu platzieren) und er muss gedreht werden können. Zudem sollen die einzelnen Komponenten des Roboters in jeweils eine sinnvolle Richtung gedreht werden können (um einfache Bewegungsabläufe zu animieren).

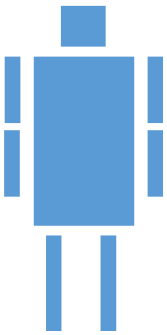

Abbildung 1: Beispielroboter

2.3.1 Szenegraph

Überlegen Sie sich zuerst, wie der zugehörige Szenegraph aussehen muss. Fertigen Sie eine kleine Zeichnung an, dadurch wird die folgende Implementierung deutlich vereinfacht. Bedenken Sie jedoch die Reihenfolge, in der Transformationen des Szenegraphen ausgeführt werden!

2.3.2 Hintergrund löschen

Sobald Sie beginnen, Ihre Objekte per Frame zu animieren, wird Ihnen auffallen, dass sie den Hintergrund "verschmieren". Genau genommen wird das alte Bild des letzten Frames nicht gelöscht und Sie zeichnen einfach "oben drauf". Dieses Problem lässt sich leicht beheben: Rufen Sie die OpenGL-Methode `glClear(GL_COLOR_BUFFER_BIT)` auf.
Sie benötigen zudem `glClearColor(0.0f, 0.0f, 0.0f, 1.0f)`. Was bewirken die beiden Methoden jeweils? Und als Folgerung: Wann müssen Sie welche Methode aufrufen?

2.3.3 Tiefentest

Bevor Sie nun mehrere Objekte in die Szene zeichnen, eine weitere Warnung: für die menschliche Wahrnehmung ist es selbstverständlich, dass Objekte, die dichter am Beobachter sind, andere Objekte in weiterer Entfernung verdecken. Dieses Prinzip ist in OpenGL nicht per se aktiviert, bis jetzt wird Verdeckung durch die Reihenfolge bestimmt, in der Objekte gezeichnet werden. Abhilfe schaffen diese Zeilen in der `Scene::init()`:

```
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_GREATER);
glClearDepth(0.0);
```

Um den Tiefentest jeden Frame erneut durchführen zu können (und kein "Verschmieren" der Tiefenwerte zu erhalten, ähnlich der Farbwerte), müssen Sie das `glClear()` erweitern zu `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);`.

Anmerkung: Dem geschulten Auge mag auffallen, dass wir hier in "falscher Richtung" testen. StandardmäSSig nutzt OpenGL ein rechtshändiges Koordinatensystem, die Kamera schaut entlang der negativen Z-Achse. Nur leider haben wir noch gar keine Kamera! Die von uns definierten Objekte liegen daher bereits in Normalized Device Coordinates (siehe spätere Vorlesung). Für Details sei auf `https://learnopengl.com/Advanced-OpenGL/Depth-testing` und `https://learnopengl.com/Getting-started/Coordinate-Systems` verwiesen.
Dieses Problem entfällt, sobald wir in Aufgabe **??** eine Kamera hinzufügen.

2.3.4 Szenegraph

Setzen Sie den aufgezeichneten Szenegraphen um. Erzeugen Sie sich für jede Transformationsgruppe im Graphen eine eigene Transform-Matrix. Laden Sie für jedes Körperteil die Matrix in den Shader (wie aus Aufgabe 2.2 bekannt) und rufen Sie den draw-Befehl für die Würfel-Geometrie auf.
Für die Implementierung des Szenegraphen müssen Sie lediglich die Matrizen aller Transformationsgruppen, die auf eine Geometrie wirken sollen, vor dem Hochladen in der richtigen Reihenfolge multiplizieren. Beschränken Sie sich der Einfachheit halber vorerst auf die richtige Positionierung und Skalierung der einzelnen Geometrien.

2.3.5 Rotationen

Nun sollen neben den Translationen und Skalierungen zur initialen Positionierung zusätzlich Rotationen eingebaut werden, entsprechend der Abhängigkeiten aus dem Szenegraphen. Die Beine des Roboters sollen dabei nach vorne und hinten "schwenken" können, um eine Gehanimation zu realisieren. Bedenken Sie, dass eine einfache Rotation um den Objektmittelpunkt des Beines nicht ausreicht, sondern sich das Rotationsgelenk an der Hüfte befindet. Hierfür stellt das Framework bereits die Methode `rotateAroundPoint(glm::vec3 point, glm::vec3 angles)` bereit, welche nach dem altbekannten Prinzip funktioniert, das Objekt erst in den Nullpunkt zu verschieben, dann zu rotieren und abschließend wieder in die ursprüngliche Position zurückzuschieben.
Neben den Beinen sollen auch die Oberarme schwenkbar realisiert werden. Die Besonderheit hier liegt wieder im Szenegraphen, der dafür sorgt, dass sich die Unterarme automatisch mit den Oberarmen bewegen, jedoch selber auch noch eine eigene Rotation aufweisen können.
Ihr Ergebnis könnte beispielsweise so aussehen:

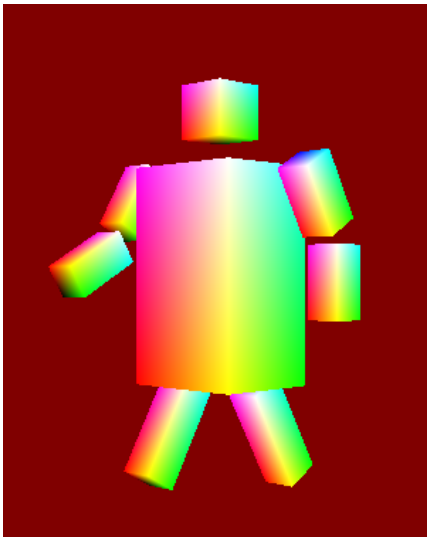

Abbildung 2: Fertiger Roboter in 3D

2.3.6 Animationen

Neben den Transformationen können in der `render()`-Methode weitere Animationen implementiert werden. Laden Sie hierfür eine float-Variable direkt in den Fragment-Shader und nutzen Sie diese, um die Farbdarstellung pro Frame zu verändern. Um keine plötzlichen Sprünge in den Farben zu erhalten, bieten sich Berechnungen mit `sin` oder `cos` an. Die genaue Art der Animation ist komplett Ihnen überlassen.