TH KÖLN
Prof. Dr. Florian Niebling
Advanced Media Institute

Technology
Arts Sciences
TH Köln

24.April 2024
Abgabe: spätestens am 10.Mai 2024

Computergrafik und Animation
Blatt 1 - Geometry

Das Praktikum besteht aus mehreren, zusammenhängenden Aufgaben in denen verschiedene Teile eines Frameworks implementiert werden. Das Ziel ist es, am Ende ein funktionierendes Spiel sowie das Framework mit den nötigen Tools für Ihr Abschlussprojekt fertiggestellt zu haben.
In diesem Aufgabenblatt beschäftigen Sie sich mit dem Zeichnen der Geometrie von Objekten. Konkret sind hier die Kenntnisse zu VertexArrayObjects, VertexBufferObjects und IndexBufferObjects gefragt. Mit der Aufgabenstellung haben Sie ein vollständiges Projekt bekommen, auf dem der Rest des Praktikums aufbauen wird. Verwenden Sie also ab diesem Praktikum das vorliegende Projekt.

1.1 3D-Geometrie

Sie haben ein Array mit Float-Werten, welche nach folgendem Schema aufgebaut sind:

$$VBO = [\underbrace{pos1_x, pos1_y, pos1_z, col1_r, col1_g, col1_b}_{\text{Vertex1}}, \underbrace{pos2_x, pos2_y, pos2_z, col2_r, col2_g, col2_b}_{\text{Vertex2}}, ...]$$

1.1.1 VBO

Zeichnen Sie die folgenden Punkte in das nachfolgend angefügte Koordinatensystem ein und versehen die Punkte mit der entsprechenden RGB-Farbe.

```
VBO=[ -0.5,  -0.5,  0.0,  0.0,  0.0,  1.0,
       0.5,  -0.5,  0.0,  0.0,  0.0,  1.0,
       0.5,   0.5,  0.0,  0.0,  1.0,  0.0,
       0.0,   1.0,  0.0,  1.0,  0.0,  0.0,
      -0.5,   0.5,  0.0,  0.0,  1.0,  0.0 ]
```

1.1.2 Stride und Offset

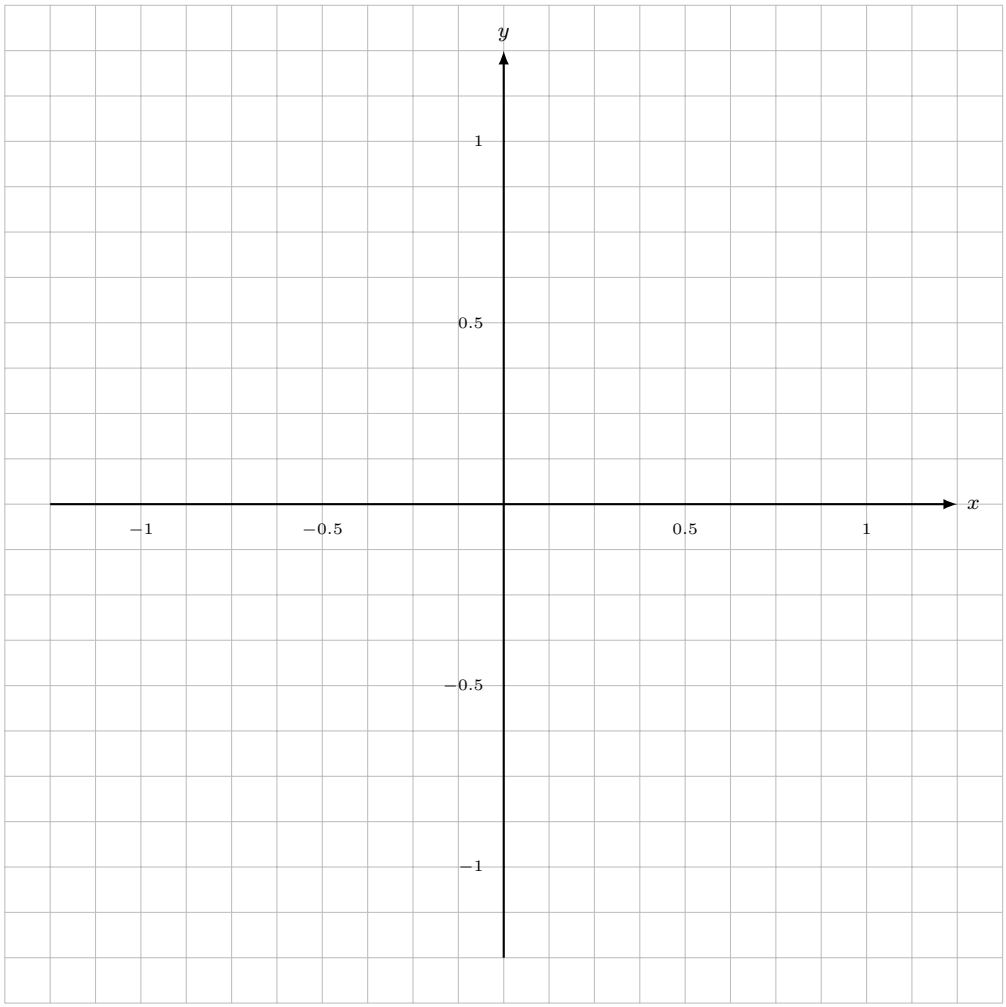
Wie würden Stride und Offset sowohl für die Positionsdaten als auch für die Farbwerte aus Aufgabe 1.1.1 aussehen?

1.1.3 IBO

Des Weiteren haben Sie ein Array, welches die Geometrie-Struktur beinhaltet. Jede Zeile des IBO ist in dem vorliegenden Fall als Dreieck zu interpretieren.

```
IBO=[ 0,  1,  2,
      0,  2,  4,
      4,  2,  3 ]
```

Zeichnen Sie die daraus resultierenden Dreiecke in dem Koordinatensystem auf der Folgeseite ein.


y
1
0.5
x
−1
−0.5
0.5
1
−0.5
−1

1.2 Basis-Geometrie

Diese Implementierung der Basis-Geometrie besteht aus drei Teilen und basiert auf der Geometrie von Aufgabe 1.1. Außerdem sind Sie gefordert selbst ein Mesh zu erzeugen und zu testen.

1.2.1 Scene-init

Im `Init`-Block der Klasse `cga.exercise.game.Scene` werden alle Objekte der anzuzeigenden Geometrie initialisiert. Zu Beginn sollen hier die aus Aufgabe 1.1.1 bekannten Vertices definiert werden. Zur Definition gehört ebenfalls die Beschreibung der einzelnen VertexAttribute, sodass der Shader weiß, wie die Daten zu interpretieren sind. Gegeben sei hierfür die Klasse `VertexAttribute` im Package `cga.exercise.components.geometry` zur Definition eines VertexAttributes:

VertexAttribute.kt

```
data class VertexAttribute(
    var n: Int,      // Number of components of this attribute
    var type: Int,   // Type of this attribute
    var stride: Int, // Size in bytes of a whole vertex
    var offset: Int  // Offset in bytes from the beginning of the vertex to the
        location of this attribute data
)
```

Übernehmen Sie im Nachgang die Indizes aus der vorherigen Aufgabe zur Definition der Geometrie und erzeugen Sie mit diesen drei Attributen ein Mesh.

1.2.2 Mesh-Klasse

Vervollständigen Sie den `Init`-Block der Klasse `cga.exercise.components.geometry.Mesh` zur Erzeugung und Verwaltung der notwendigen Buffer (VAO, VBO & IBO). Des Weiteren müssen Sie die `render()`-Methode implementieren, sodass die Geometrie des Meshes in jedem Frame neu gezeichnet wird.
Das nachfolgende Prozedere soll Ihnen dabei eine kleine Hilfe sein. Nutzen Sie zudem auch die Vorlesungsfolien.
Beim Initialisieren:

1) Erstellen und Binden eines VAO. Jedes VBO und IBO, die Sie in der Sequenz binden, wird dem aktuellen VAO zugeordnet.

2) Erstellen und Binden von Vertex- und Indexbuffern.

3) Füllen Sie die Vertex- und Indexbuffer mit den entsprechenden Daten.

4) Aktivieren und definieren Sie die jeweiligen VertexAttribute.

5) Danach sollte alles gelöst werden (unbind), um versehentliche Änderungen an den Buffern und VAO zu vermeiden. Denken Sie daran, das VAO zuerst zu lösen.

Beim Rendern:

1) Binden Sie das VAO.

2) Zeichnen Sie die Elemente.

3) Lösen der Bindung, um versehentliche Änderungen am VAO zu vermeiden.

1.2.3 Scene-`render()`

In jedem Frame wird von dem Framework die Methode `render()` der `Scene`-Klasse aufgerufen. Alle Objekte, die dargestellt werden sollen, müssen hier in die Rendering-Pipeline integriert werden. Außerdem muss jeweils zuvor definiert werden, welcher Shader das Rendern übernehmen soll. Um die Verantwortung an den jeweiligen Shader zu übergeben, nutzen Sie bitte die `use()`-Methode des `ShaderPrograms`.
Haben Sie alles richtig gemacht, sollte das Bild aus Abbildung 1 erscheinen:

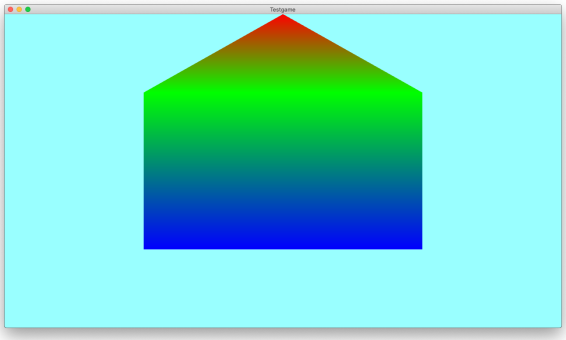
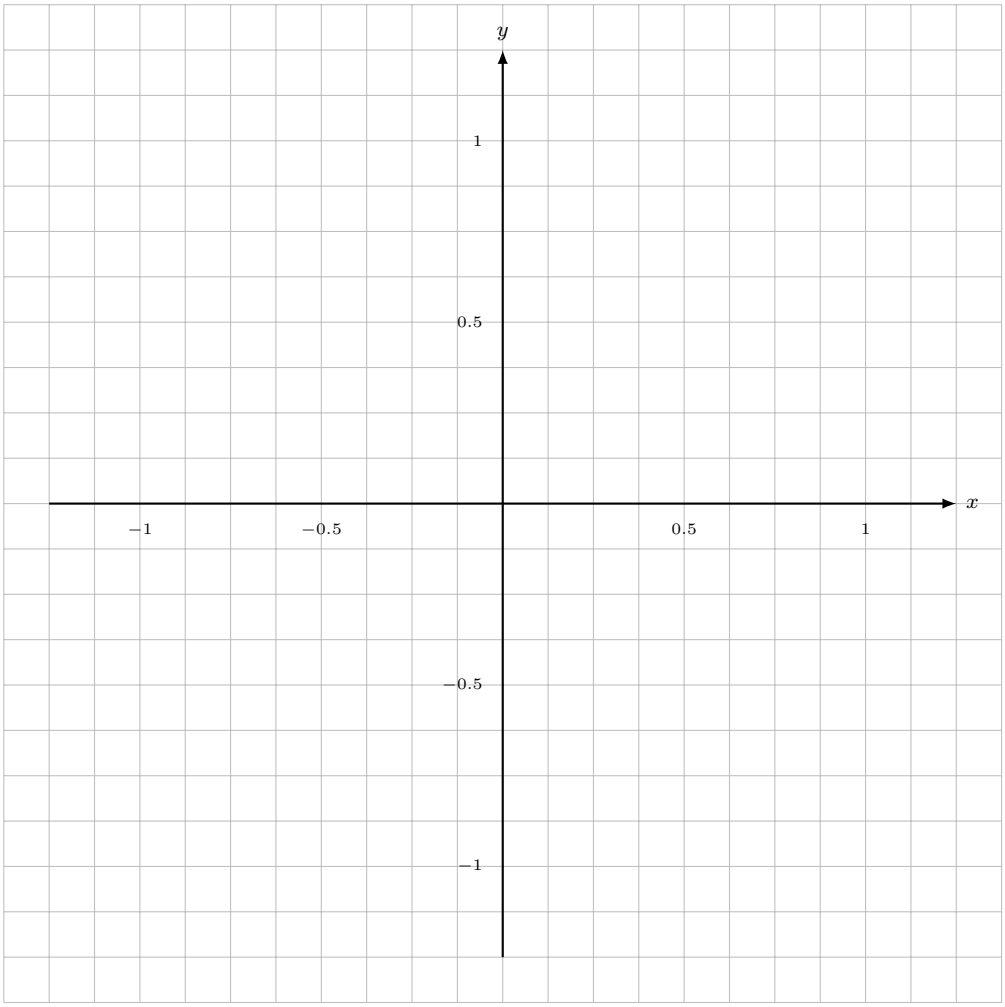


Abbildung 1: Gerendertes Haus

1.2.4 Initialen

Verändern Sie die Vertex- und Indexdaten, sodass die Initialen Ihres Namens dargestellt werden. Das nachfolgende Koordinatensystem soll Ihnen hierbei behilflich sein, um die entsprechenden Koordinaten zu ermitteln. **Halten Sie hierbei die Schrift einfach. Ein O muss nicht rund sein.**



1.2.5 Backface culling

Schalten Sie nun das Backface Culling ein, indem Sie die Methode in der Scene mit den passenden Parametern aufrufen. Hierzu kann es auch sinnvoll sein, sich die Dokumentation[1] durchzulesen. Die Dreiecke, welche durch die mitgelieferten Vertexdaten entstehen, sind gegen den Uhrzeigersinn definiert. Sollten Lücken in Ihren Initialen entstanden sein, so korrigieren Sie bitte die Orientierung.

1.3 Objekt-Handling

Die Grundlagen bzgl. des Geometrie-Handlings sollten nun verstanden sein, sodass wir einen Schritt weiter gehen und mit komplexeren Objekten arbeiten können. Komplexere Objekte können aus mehreren Meshes bestehen, (ein einzelnes Mesh modelliert in diesem Fall ein Feature des Objektes).

1.3.1 OBJLoader

Der mitgelieferte OBJLoader übernimmt das Einlesen und Interpretieren von obj-Dateien (Handbuch liegt bei). Bitte laden Sie das Objekt `sphere` aus dem `assets`-Ordner in die Scene.

[1] `https://www.khronos.org/opengl/wiki/Face_Culling`

Die Handlungsabfolge kann folgendermaßen aussehen:

a) `loadOBJ(...)` erstellt ein OBJResult

b) das OBJResult hält unter anderem eine Liste der enthaltenen Meshes. Diese können bspw. folgendermaßen erreicht werden:
val objRes: OBJResult
... //get OBJResult by OBJLoader
val objMeshList: MutableList<OBJMesh> = objRes.objects[0].meshes

c) Jedes OBJMesh besitzt die getter-Methoden für `vertexData` und `indexData`, mit denen Sie auf die Daten des Meshes zugreifen können.

d) Die Vertices speichern sowohl Position (pos), Texturkoordinaten (tex) als auch die Normale (norm) und besitzen folgende Struktur:

$$VBO = [\underbrace{pos1_x, pos1_y, pos1_z, tex1_u, tex1_v, norm1_x, norm1_y, norm1_z}_{\text{Vertex1}}, \underbrace{pos2_x, pos2_y, pos2_z, ..., ...}_{\text{Vertex2}}]$$

Definieren Sie die `VertexAttributes` und erzeugen Sie die `Mesh`-Objekte.

e) Rendern Sie das/die Objekt(e) in der Methode `Scene.render()`.

Starten Sie das Projekt, um sicherzustellen, dass Sie alles korrekt implementiert haben. Vergleichen Sie bitte Ihr Render-Ergebnis mit Abbildung 2 und interpretieren es.
Schalten Sie anschließend mit Hilfe der mitgelieferten Funktion in der Klasse Scene den Tiefentest[2] an. Sie sollten nun das Ergebnis aus Abbildung 2 sehen. Das Fenster besitzt die Dimensionen -1 bis 1 in x- als auch in y-Richtung und 0 in z-Richtung. Die Kugel ist als Einheitskugel definiert und nimmt somit das gesamte Fenster ein. Die Farbe kommt in diesem Fall aus dem VertexAttribut der Texturkoordinaten (vec2), welcher im Shader als `vec3` interpretiert und bei dem die b-Komponente auf 0 gesetzt wird.

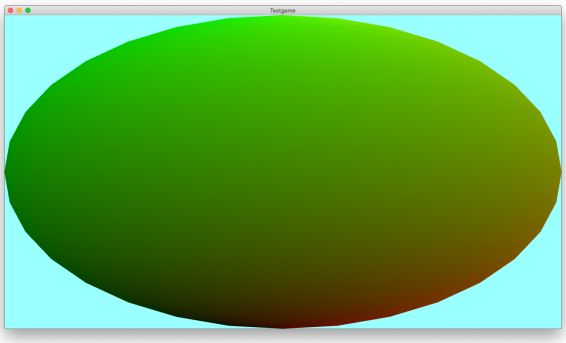


Abbildung 2: Integration der Sphere

Ändern Sie abschließend die Hintergrundfarbe auf schwarz, indem Sie folgende Zeile im Init-Block der Scene in die gegebenen Werte ändern.

Scene

```
init {
...
        glClearColor(0.0f, 0.0f, 0.0f, 1.0f); GLError.checkThrow()
...
}
```

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