# Rendern mit OpenGL

## Aufgabe 1

Im Tarball `meshrenderer-2021.tar.bz2` finden Sie ein minimalistisches Programm zum rendern des selben Dreiecksnetzes wie in der letzten Aufgabe mittels OpenGL.

Die nötigen Abhängigkeiten zum Übersetzen auf Debian-basierten Systemen können via

```
sudo apt install build-essential libglew-dev libglfw3-dev libglm-dev
```

installiert werden. Einige davon haben Sie sicher für das letzte Blatt schon installiert.

In den Dateien `src/context.{h,cpp}` findet das Aufsetzen des Fensters, des OpenGL 'Kontextes' und der Verbindung beider statt. Sie können die Datei gerne durchsehen, für die erfolgreiche Bearbeitung der Übungsaufgabe ist das aber nicht nötig.

In der Datei `src/main.cpp` sehen Sie den Ablauf des Programms. Zu Beginn wird OpenGL (und das Fenstersystem) initialisiert, danach wird, wie in der vorigen Übungsaufgabe, das Dreiecksnetz ('mesh') geladen. Bevor nun das tatsächliche Rendern beginnt werden zusätzlich die Shader geladen, übersetzt und gelinkt, wonach der sog. 'Main'-Loop betreten wird. Hier läuft das Programm so lange bis das Fenster geschlossen wird (z.B. via ESC).

Im Main-Loop wird zuerst das im vorigen Frame gezeichnete Bild gelöscht (`glClear`), hier können Sie mittels `glClearColor` verschiedene Farbwerte ausprobieren. Welcher Wertebereich muss hier verwendet werden? (Gilt im weiteren auch im Shadercode weiter.)

Dann wird der zuvor geladene Shader gebunden, Uniform-Daten auf die GPU übertragen, das Mesh gezeichnet (der Draw-Call) und schließlich die Shaderbindung gelöst. Im letzten Schritt werden 'die Buffer getauscht'.

### Aufgabe 1.1

Der letzte Schritt im Main-Loop wird gemeinhin 'Buffer-Swap' genannt. Was passiert hier, warum ist das sinnvoll? Recherchieren Sie.

### Aufgabe 1.2

Beim Laden der Shader wurde kein Code entfernt, vergleichen Sie ihn trotzdem mit dem Inhalt der Vorlesungsfolien. Beachten Sie auch die 'Lebenszeiten' der OpenGL Objekte. Werden diese in wirklich jedem Fall 'aufgeräumt'?

### Aufgabe 1.3

In der Datei `mesh.cpp` sind einige Stellen auskommentiert. Implementieren Sie in `load_mesh` ein Indexed Face Set bei dem die Daten (Dreiecke und Indizes) in OpenGL Buffer Objekten liegen, deren Verbindung wiederum in einem OpenGL Vertex Array Object gespeichert ist.

Sie können als Zwischenschritt (falls Ihnen das einfacher erscheint) zuerst die Variante ohne VAO implementieren.

Implementieren Sie dann die Funktion `draw_mesh` so dass diese den/die passenden Buffer bindet und den entsprechenden Draw-Call absetzt.

Zum Test verwenden Sie das genannte Clip-Space Dreieck (`#define CS_TRI`). Wie Sie dem Shadercode aus `shader.vert` entnehmen können werden die Eckpunkte des übertragenen Dreiecks als Punkte im Clip-Space verstanden und direkt an den Rasterisierer weitergegeben.

### Aufgabe 1.4

Verwenden Sie in `mesh.cpp` nun `#define ES_TRI`, ergänzen Sie in `main.cpp` die Projektionsmatrix und verändern Sie den Shader so, dass diese Matrix verwendet wird.

### Aufgabe 1.5

Verwenden Sie in `mesh.cpp` nun `#define BUNNY`, ergänzen Sie in `main.cpp` die Viewingmatrix und verändern Sie den Shader so, dass die Transformation von Weltkoordinaten nach Clippingkoordinaten ausgeführt wird.

### Aufgabe 1.6

Analog zum letzten Übungsblatt sollen die Dreiecke des Modells nun unterschiedliche Farben erhalten. Das soll über das Hinzufügen eines neuen Vertex-Attributes erfolgen.

Ergänzen Sie dazu `load_mesh`, so dass für je drei aufeinanderfolgende Eckpunkte die gleiche, zufällige Farbe verwendet wird. Verwenden Sie dazu `glm::linearRand(...)` mit dem weiter oben ermittelten Wertebereich.

Fügen Sie das so erzeugte VBO dem VAO hinzu und nehmen Sie die Daten im Vertex-Shader in Empfang. Sie können die Farbe dann direkt aus dem Vertex-Shader an den Fragment-Shader weitergeben (Hinweis: `in`/`out` Variablen).

Nun sollten Sie uniform eingefärbte Dreiecke sehen. Warum eigentlich? Um das Ziel zu erreichen müssten die Farben korrekterweise an den Dreiecken selbst hängen, nicht and (beliebig indizierten!) Eckpunkten. Können Sie erklären warum es hier trotzdem funktioniert?

Zur Veranschaulichung der Vertex-Attribut-Interpolation wie in der zweiten Vorlesung besprochen geben Sie für die Eckpunkte eines Dreiecks leicht unterschiedliche Farben an, Sie sollten dann einen Gradienten auf den gerenderten Oberflächen sehen.

### Aufgabe 1.7

Nun können wir, ebenfalls wie im vorigen Übungsblatt, noch um das gezeichnete Objekt rotieren. Ändern Sie pro Frame die Kameraposition wie im Python-Skript der letzten Aufgabe. Wahrscheinlich ist die Ausgabe relativ hektisch.

Wenn Sie möchten können Sie sich aus dem C++ `chrono` Modul bedienen um vergangene Zeit zu messen. Sie können auch einfach nur alle $N$ Frames die Kamreaposition ändern, aber: was ist das Problem mit diesem Ansatz?

### Aufgabe 1.8

Im letzten Blatt wurde der Tiefentest implementiert, hier ist er automatisch aktiv. Schalten Sie ihn nun auch hier testweise mit `glDisable` ab und beobachten Sie das Ergebnis.

**Happy Hacking :)**