Setup

Der gesamte Quellcode befindet sich in *cgintro-3-tutorial.zip*, genaue Buildanweisungen findet Ihr in `README.de.md`.

Suzanne

Geometrie von Hand einzugeben, wie wir es bisher gemacht haben, ist mühselig und fehleranfällig. Ab dieser Übung werden wir stattdessen *Wavefront OBJ* Dateien benutzen, um komplexere Modelle zu importieren. Eine typische *OBJ* Datei sieht ungefähr so aus:

```
o Suzanne
v 0.490583 0.186287 0.719791
v 0.475731 0.192902 0.737549
v 0.476952 0.160851 0.722916
v 0.507355 0.178955 0.699112
...
f 5 6 2
```

Die erste Zeile definiert ein neues Objekt mit dem Namen “Suzanne” (“o”-Zeile) – die folgenden Zeilen definieren 4 vertices (“v”-Zeilen) und die letzte Zeile definiert ein Dreieck aus dem fünften, sechsten und zweiten vertex (“f”-Zeilen). Das ganze können wir dann als Datei einlesen, parsen und uns daraus Vertex-/Indexbuffer bauen.

Code dafür befindet sich in `src/framework/objparser.cpp` und `src/framework/mesh.cpp`. Mit folgendem Befehl können wir ein Modell laden:

```
mesh.load("meshes/suzanne.obj");
```

`mesh.load()` verschmilzt alle Objekte aus der übergebenen Datei zu einem Mesh, wobei jeder Vertex aus 3 Attributen besteht:

```
struct VertexPCN {
    /* Attribut 0 ist die Position des Vertex */
    glm::vec3 position;
    /* Attribut 1: Die Texturkoordinate des Vertex (wird später wichtig)
       */
    glm::vec2 texCoord;
    /* Attribut 2: Die Oberflächennormale des Vertex (für
       Beleuchtungsberechnungen) */
    glm::vec3 normal;
};
```

Das Mesh besitzt Bufferobjekte für Indices und Vertices und ein Vertex-Array, dass Informationen über das Layout enthält:

```
class Mesh {
   public:
    unsigned int numIndices;
    VertexArray vao;
    Buffer vbo;
    Buffer ebo;
};
```

Die Anzahl der Indices wird außerdem benötigt, um das Mesh später zu zeichnen:

```
void Mesh::draw() {
    vao.bind();
    glDrawElements(GL_TRIANGLES, numIndices, GL_UNSIGNED_INT, 0);
    vao.unbind();
}
```

Die Transformationskette von OpenGL

In dieser Übung wollen wir das erste Mal etwas 3-dimensional abbilden. Dafür müssen wir unser Mesh, dessen Vertices bisher noch im lokalen Modellraum liegen, auf den Bildschirm projizieren.

Üblicherweise passiert das durch 6 aufeinanderfolgende Transformationen:

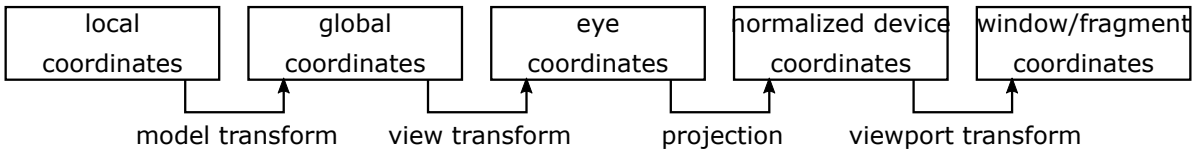

Abbildung 1: Die Transformationskette

1. Die *Model*-Matrix transformiert die Vertices aus dem lokalen Modellraum in eine globale Lage innerhalb unserer Szene. Diese Matrix spezifiziert also die Transformation eines individuellen Objekts.
2. Die *View*-Matrix transformiert das Objekt aus der globalen Lage in den betrachterspezifischen Eye-Space. Wir können uns das so vorstellen, als würden wir statt die Kamera zu bewegen, die ganze Szene so bewegen, dass die Kamera im Ursprung liegt. Die *View*-Matrix ist also genau *invers* zur Kameramatrix.
3. Die *Projection*-Matrix ist für die perspektivische Projektion zuständig, und bildet aus dem 3-dimensionalen Raum auf den Clip-Space ab. Da wir die Welt vorher genau so transformiert haben, dass die Kamera im Ursprung liegt, ist diese Matrix im wesentlichen konstant und hängt nur von der Breite des Sichtfeldes (Field of View) ab.

4. Im Clip-Space beschneidet OpenGL jetzt die Dreiecke, so dass diese alle innerhalb des Sichtfeldes liegen, das durch die Bounding-Box zwischen $(-w, -w, -w)$ und (w, w, w) definiert ist.
5. OpenGL dividiert jetzt x, y, z jeweils durch w um die Koordinaten in Normalized-Device-Coordinates zwischen $(-1, -1, -1)$ und $(1, 1, 1)$ zu transformieren.
6. Die *Viewport*-Transformation überführt diese Normalized-Device-Coordinates nun in den Screen-Space, dessen Ausmaße wir mit `glViewport` spezifizieren.

Wir müssen uns in unserem Programm nur um die ersten 3 Transformationen kümmern, den Rest erledigt die OpenGl Pipeline.

Auf den ersten Blick klingt das ganze immer noch unnötig aufwändig; tatsächlich ist aber das Gegenteil der Fall, da wir die Assoziativität der Matrixmultiplikation zur Optimierung nutzen können. Nennen wir die Model-Matrix M, die Kameramatrix C und die Projektionsmatrix P, dann müssen wir jedes Vertex $\mathbf{v}$ wie folgt aus lokalen Modellkoordinaten in den Clip-Space transformieren:

$$\mathbf{v}' = P(C^{-1}(M\mathbf{v})) = (P \cdot C^{-1} \cdot M)\mathbf{v} = (P \cdot V \cdot M)\mathbf{v}, \quad \text{mit } V := C^{-1}.$$

Die Matrix V ist die oben genannte *View*-Matrix. Damit reduziert sich die Berechnung auf eine Matrixmultiplikation pro Vertex.

Im Falle unseres Beispiels ist die *model*-Matrix eine Rotation um die y-Achse:

```
mat4 model = rotate(mat4(1.0f), App::time, vec3(0.0f, 1.0f, 0.0f));
```

Die Projektionsmatrix habt ihr bereits in der Vorlesung kennengelernt. Diese berechnen wir mit der Funktion `glm::perspective`, welche gegeben eines Sichtfeldwinkels (ein *field of view*, oder meistens kurz *FOV*) eines Seitenverhältnisses (meistens Breite des Fensters durch Höhe des Fensters) und der sogenannten *near* und *far* Werte eine Matrix für eine perspektivische Transformation erstellt. *near* und *far* definieren zwischen welchen z-Werten OpenGL Dreiecke clippt.

Um die Kamerabewegung kümmert sich die `Camera`-Klasse. Die *View*-Matrix und die *Projection*-Matrix werden beide von der `Camera`-Klasse bereitgestellt:

```
// Berechne die View- und Projection-Matrix bei Änderung neu
camera.updateIfChanged();
mat4 view = camera.viewMatrix;
mat4 projection = camera.projectionMatrix;
```

Die `Camera`-Klasse enthält die Funktionen `Camera::zoom`, `Camera::rotate` und `Camera::resize`, um die Kamera um den Ursprung herum zu bewegen, diese kontrollieren wir mit der Maus, indem wir die dazugehörigen Callbacks überschreiben:

```
void MainApp::scrollCallback(float amount) {
    camera.zoom(amount);
```

```
}
void MainApp::moveCallback(const vec2& movement, bool leftButton, bool
    rightButton, bool middleButton) {
    if (rightButton) camera.rotate(movement * 0.01f);
}
void MainApp::resizeCallback(const vec2& resolution) {
    camera.resize(resolution.x / resolution.y);
}
```

Uniform Shader Variables

Wir haben an dieser Stelle unsere Matrizen, um diese effizient anzuwenden, multiplizieren wir sie **im Shader** mit unseren Vertices. Da die Matrizen ja für jeden Vertex (zumindest eines Objektes) die gleichen sind, benutzen wir Uniform Variablen um die Matrizen an den Shader zu übergeben.

Unser Vertex-Shader sieht dann wie folgt aus:

```
#version 330 core

layout (location = 0) in vec3 position;
layout (location = 2) in vec3 normal;

uniform mat4 uLocalToClip; // = projection * view * model
uniform mat4 uLocalToWorld; // = model

out vec3 interpNormal;

void main() {
    gl_Position = uLocalToClip * vec4(position, 1.0);
    interpNormal = (uLocalToWorld * vec4(normal, 0.0)).xyz;
}
```

Wir spezifiziern die Vertex-Attribute, die wir benutzen, und die Matrizen, die wir als Uniform Variablen von der CPU benötigen.

Für die Berechnungen erweitern wir die Vertexposition und -normale zu homogenen Koordinaten, die Vertexposition muss aus dem lokalen Modellraum in den Clip-Space transformiert werden, die Vertexnormale für Beleuchtungsberechnungen nur in den World-Space. Beachtet hierbei, dass für Positionen $w = 1$ und für Richtungen $w = 0$ gilt.

Uniform Variablen können sowohl im Vertex- als auch im Fragment-Shader verwendet werden – in unserem Fragment-Shader nutzen wir eine uniform Variable um die Richtung zu definieren, aus der das Licht kommt:

```
#version 330 core
```

```
in vec3 interpNormal;

out vec3 fragColor;

uniform vec3 uLightDir;

void main() {
    vec3 normal = normalize(interpNormal); // Renormalisieren nach
       Hardware-Interpolation
    fragColor = max(dot(normal, uLightDir), 0.0) * vec3(1.0);
}
```

Jetzt müssen wir die Matrizen nur noch an die GPU übergeben. OpenGL teilt jeder Uniform Variable eine eigene Adresse zu, welche wir uns mittels `glGetUniformLocation` auslesen können. Nun können wir unsere 4×4 Matrizen mittels `glUniformMatrix4fv` hochladen, für den 3D Vektor der Lichtrichtung wiederum benutzen wir `glUniform3fv`. Beides kapselt die `Program`-Klasse für uns hinter Funktionen:

```
mat4 localToClip = projection * view * model;
mat4 localToWorld = model;

program.set(program.uniform("uLocalToClip"), localToClip);
program.set(program.uniform("uLocalToWorld"), localToWorld);
program.set(program.uniform("uLightDir"), normalize(vec3(1.0f)));
```

Going 3D

Wenn wir das Programm an diesem Punkt ausführen, sollte die Ausgabe so aussehen:

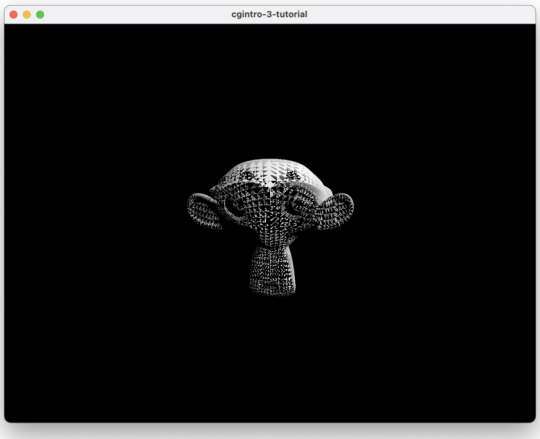

Abbildung 2: Ausgabe ohne Culling und Depth-Test

Die Ausgabe ist zwar 3D, sieht aber noch chaotisch aus, da OpenGL mit durcheinanderkommt welche Faces vorne und welche hinten sind. Um das zu verhindern, können wir OpenGL anweisen nur Faces zu zeichnen, die nach vorne zeigen:

```
void MainApp::init() {
    glEnable(GL_CULL_FACE);
}
```

Das Ergebnis sollte dann so aussehen:

Abbildung 3: Ausgabe mit Culling und ohne Depth-Test

Das macht die Ausgabe deutlich besser jedoch besteht immer noch das Problem, das Oberflächen, die hintereinander liegen, nicht richtig sortiert werden. Das liegt daran, dass wir OpenGL noch mitteilen müssen, dass wir auch tiefensortieren wollen:

```
void MainApp::init() {
    glEnable(GL_CULL_FACE);
    glEnable(GL_DEPTH_TEST);
}
```

Außerdem müssen wir den Tiefenpuffer noch zwischen Frames leeren, indem wir `glClear(GL_COLOR_BUFFER_BIT)` zu `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)` erweitern. Damit sieht Suzanne jetzt endlich richtig aus:

Abbildung 4: Ausgabe mit Culling und Depth-Test

GUI

Das Framework bindet die Bibliothek Dear ImGui ein, die es uns ermöglicht unser Programm etwas interaktiver zu gestalten.

ImGui ist eine sogenannte *immediate mode* GUI-Schnittstelle. Solche APIs bieten sich vor allem in Echtzeit-Anwendungen an, da alle GUI-Elemente live und ohne Duplikation von Daten mit minimalem Eingriff in den bestehenden Quellcode eingebaut werden können.

Zuerst müssen wir noch ein paar Variablen einbauen, die wir verändern können. Dafür erweitern wir den Fragment-Shader:

```
#version 330 core

in vec3 interpNormal;
out vec3 fragColor;

uniform vec3 uLightDir;
uniform vec3 uLightColor;
uniform bool uShowNormals;

void main() {
    vec3 normal = normalize(interpNormal);
    vec3 lighting = uLightColor * max(dot(normal, uLightDir), 0.0);
    fragColor = uShowNormals ? normal * 0.5 + 0.5 : lighting;
}
```

Wir können jetzt mit `uShowNormals` umschalten, ob wir die Normale visualiseren wollen oder das

Objekt beleuchten, und mit `uLightColor` die Farbe des Lichts einstellen. Jedes dieser Felder müssen wir natürlich noch im Code setzen:

```
class MainApp : public App {
   private:
    ...
    vec3 lightDir = normalize(vec3(1.0f, 1.0f, 1.0f));
    vec3 lightColor = vec3(1.0f);
    bool showNormals = false;
}

void MainApp::render() {
    ...
    program.set(program.uniform("uLightDir"), lightDir);
    program.set(program.uniform("uLightColor"), lightColor);
    program.set(program.uniform("uShowNormals"), showNormals);
}
```

In unserem Framework können wir nun GUI Elemente innerhalb der `buildImGui`-Methode auflisten:

```
void MainApp::buildImGui() {
    ImGui::StatisticsWindow(App::delta, App::resolution);

    ImGui::Begin("Settings", nullptr, ImGuiWindowFlags_AlwaysAutoResize);
    ImGui::SphericalSlider("Light Direction", lightDir);
    ImGui::ColorEdit3("Light Color", value_ptr(lightColor),
       ImGuiColorEditFlags_Float);
    ImGui::Checkbox("Show Normals", &showNormals);
    if (ImGui::SliderAngle("Field Of View", &camera.fov, 0.0f, 180.0f))
       camera.invalidate();
    if (ImGui::Button("Load Bunny")) mesh.load("meshes/bunny.obj");
    if (ImGui::Button("Load Suzanne")) mesh.load("meshes/suzanne.obj");
    if (ImGui::Button("Load Sphere")) mesh.load("meshes/highpolysphere.obj
       ");
    ImGui::End();
}
```

`ImGui::StatisticsWindow` ist aus `framework/imguiutil.cpp` und zeichnet die Framerate in die obere linke Ecke des Fensters.

Zwischen `ImGui::Begin` und `ImGui::End` definieren wir ein neues GUI-Fenster mit dem Namen `"Settings"`, dessen Größe sich automatisch an den Inhalt anpasst.

`ImGui::SphericalSlider` ist ebenfalls aus `framework/imguiutil.cpp` und ermöglicht uns einen Einheitsvektor anhand von zwei Winkel einzustellen.

Mit `ImGui::ColorEdit3`, `ImGui::Checkbox` und `ImGui::SliderAngle` stellen wir die Variablen ein, die per Pointer übergeben werden, `glm::value_ptr` ist eine Funktion, die einen Pointer auf die Felder eines Vektors zurückgibt.

Jedes GUI-Element gibt einen `bool`-Wert zurück, der **`true`** ist, wenn eine Änderung aufgetreten ist. Damit implementieren wir die Buttons und teilen der Kamera über `Camera::invalidate` mit, dass die Projektions- und Viewmatrix im nächsten `update` Schritt neuberechnet werden müssen.

Über das Feld `App::imguiEnabled` können wir die Sichtbarkeit der GUI-Elemente steuern:

```
void MainApp::keyCallback(Key key, Action action) {
    // Close the application when pressing ESC
    if (key == Key::ESC && action == Action::PRESS) App::close();
    // Toggle GUI with COMMA
    if (key == Key::COMMA && action == Action::PRESS) App::imguiEnabled =
        !App::imguiEnabled;
}
```