Diese Lektion bezieht sich auf den Code in `cgintro-4a.zip` und `cgintro-4b.zip`.

Raycasting

In dieser Lektion wollen wir Geometrie rendern, indem wir entlang von Strahlen nach Schnittpunkten mit der Szene suchen.

Dazu brauchen wir eine Möglichkeit Shadercode pro Pixel auszuführen, das machen wir indem wir ein großes Dreieck zeichnen, dass den gesamten Bildschirm abdeckt.

Wir legen dafür zuerst ein Shaderprogramm, ein Mesh und eine Kamera, die später benötigt wird, als Felder in unserer App an:

```
class MainApp : public App {
   public:
    MainApp();
    ...
   private:
    Program program;
    Mesh mesh;
    Camera camera;
}
```

Im Konstruktor laden wir die vordefinierten Vertices und Indices des Bildschirmdreiecks und kompilieren unsere Shaderprogramm:

```
MainApp::MainApp() : App(800, 600) {
    App::setTitle(Config::PROJECT_NAME);
    mesh.load(FULLSCREEN_VERTICES, FULLSCREEN_INDICES);
    program.load("raygen.vert", "raymarch.frag"); // bzw "raytrace.frag"
}
```

Die `FULLSCREEN_VERTICES` und `FULLSCREEN_INDICES` sind in `framework/mesh.hpp` definiert:

```
const std::vector<float> FULLSCREEN_VERTICES = {
    -1.0f, -1.0f, 0.0f,
     3.0f, -1.0f, 0.0f,
    -1.0f,  3.0f, 0.0f,
};
const std::vector<unsigned int> FULLSCREEN_INDICES = {
    0, 1, 2,
};
```

Wie schon in den vorherigen Übungen überschreiben wir die Callback-Funktionen, um Nutzerinteraktion zuzulassen und das Bewegen der Kamera zu ermöglichen.

Auch unsere Renderloop bleibt unspektakulär, da wir lediglich die Kameraparameter und Zeit an unseren Shader weitergeben:

```
void MainApp::render() {
    program.set(program.uniform("uTime"), App::time);

    if (camera.updateIfChanged()) {
        program.set("uCameraMatrix", camera.cameraMatrix);
        program.set("uAspectRatio", camera.aspectRatio);
        program.set("uFocalLength", camera.focalLength);
        program.set("uCameraPosition", camera.cartesianPosition);
    }

    mesh.draw();
}
```

Den Aufruf von `glClear` können wir uns sparen, da wir in jedem Frame den gesamten Bildschirm übermalen. Alles wichtige passiert dieses mal im Shader. Den Strahlursprung haben wir bereits gegeben mit der Position der Camera `uCameraPosition`, was wir noch brauchen um Strahlen erzeugen zu können ist die Richtung der Strahlen. Diese variiert bei einer perspektivischen Projektion für jeden Pixel. Um uns jedoch Arbeit pro Pixel zu sparen, benutzen wir die Hardware-Interpolation von Vertexattributen zwischen Fragment- und Vertex-Shader, und berechnen die Strahlrichtung nur explizit an den 3 Eckpunkten unseres Bildschirm-Dreiecks:

```
#version 330 core

layout (location = 0) in vec3 _position;

out vec3 viewDir;

uniform mat4 uCameraMatrix;
uniform float uAspectRatio;
uniform float uFocalLength;

void main() {
    gl_Position = vec4(_position, 1.0);
    viewDir = mat3(uCameraMatrix) * vec3(_position.x * uAspectRatio,
       _position.y, -uFocalLength);
}
```

Dafür brauchen wir diesmal die Kamera-Matrix, die, wie wir uns erinnern, das inverse Gegenstück zur View-Matrix ist. Die Kamera-Matrix bildet von dem Eye-Space in den World-Space ab, d.h. sie bildet die Tranformation der Kamera ab. Durch das Beschneiden der Kamera-Matrix mit `mat3`, setzen wir implizit $w = 0$. Wenn wir also einen Richtungsvektor mit der Kamera-Matrix multiplizieren, wenden wir die Rotation der Kamera auf diesen Vektor an. Wir brauchen also nur die Kamera-Matrix mit unsere Strahlrichtung im Eye-Space zu multiplizieren, um die Strahlrichtung im World-Space zu erhalten.

Unsere Strahlrichtung im Eye-Space ergibt sich einfach als der Vektor vom Fokuspunkt zum Pixel auf dem Bildschirm. Der Fokuspunkt sitzt bei `vec3(0, 0, uFocalLength)`, wobei sich `uFocalLength` aus dem Field-Of-View-Winkel ergibt (zur Berechnung siehe `camera.cpp`), der Pixel bei `vec3(x * uAspectRatio, y, 0)`. Daraus ergibt sich dann als Richtungsvektor `vec3(x * uAspectRatio, y, -uFocalLength)`.

Damit haben wir alles beisammen, um Kamerastrahlen zu erzeugen, die Normalisierung der Richtungsvektoren nehmen wir erst im Fragment-Shader vor, da das Interpolieren die Vektorlänge verändert. Im Folgenden werden wir diese Strahlen in zwei unterschiedlichen Techniken benutzen, um Bilder zu erzeugen.

Raymarching

Beim Raymarching stellen wir unsere Objekte implizit durch sogenannte Signed-Distance-Funktionen dar. Eine Signed-Distance-Funktion ist in unserem Fall eine Abbildung aus dem $\mathbb{R}^3$ in $\mathbb{R}$, die jedem Punkt p einen vorzeichenbehafteten Abstand zur nächstgelegenen Oberfläche zuordnet. Dabei ist das Vorzeichen negativ, wenn sich p innerhalb eines Objektes befindet, und positiv außerhalb.

Für eine Kugel mit dem Radius r ist die SDF somit einfach der Abstand vom Zentrum minus den Radius:

```
float sdSphere(vec3 pos, float r){
    return length(pos) - r;
}
```

In `raymarch.frag` ist eine kleine Beispielszene mit zwei Objekten beschrieben:

```
vec2 sdScene(vec3 pos) {
    vec2 result = vec2(Inf, intBitsToFloat(SKY_ID));
    result = combine(result, sdBox(pos, vec3(2.0, 0.2, 2.0)), BOX_ID);
    result = combine(result, sdBouncyBall(pos, uTime), BALL_ID);
    return result;
}
```

`result` ist dabei ein `vec2`, der die vorzeichenbehaftete Distanz zur Szene in der x-Komponente enthält und eine Objekt-ID in der y-Komponente, um uns später das Texturieren der Szene zu ermöglichen. Wurde noch nichts getroffen, ist die Distanz positiv unendlich. Die Funktion `combine` kombiniert dabei die Szene (`result`) mit einem weiteren Objekt, indem sie das Minimum bildet:

```
vec2 combine(vec2 result, float dist, int ID) {
    return dist < result.x ? vec2(dist, intBitsToFloat(ID)) : result;
}
```

Die Objekt-ID ist als Integer codiert, um also einen `vec2` zu bilden packen wir diese mit der GLSL-Funktion `intBitsToFloat` zuerst in einen **`float`**.

Um jetzt einen Schnittpunkt mit der Szene zu ermitteln beginnen wir im Strahlursprung (plus ggf. einen `near`-Wert) und ermitteln den geringsten Abstand zur Szene. Diesen Abstand können wir jetzt entlang des Strahls laufen und uns sicher sein, dass wir auf dem Weg keinen Schnittpunkt verpassen. Das wiederholen wir solange, bis der Abstand zur Szene 0 oder ausreichend klein ist (`< uEpsilon`), oder wir unsere maximale Schrittanzahl `uSteps` erreicht haben. Danach geben wir unsere Approximation für den Schnittpunkt zurück. Wenn wir jedoch vorher die Far-Plane (`far`) überschreiten, schneidet der Strahl die Szene nicht und wir geben als Tiefenwert Unendlich zurück (`Inf`).

Haben wir nun ein Objekt getroffen, können wir die Normale am Schnittpunkt bestimmen, indem wir den Gradienten der SDF normalisieren, also die Richtung des größten Distanzanstiegs ermitteln. Statt den Gradienten analytisch zu berechnen, können wir diesen auch numerisch approximieren.

```
vec3 pos = rayOrigin + rayDir * depth;
vec3 normal = normalScene(pos);
```

Die Normale können wir dann wie bei der Rasterisierung für unsere Beleuchtungsberechnungen nutzen:

```
vec3 lighting = max(dot(normal, uLightDir), 0.0) * uLightColor;
```

Da wir mit Strahlen arbeiten, ist es außerdem leicht Schatten in unsere Szene einzubauen, indem wir für jeden Schnittpunkt einen zusätzlichen Schattenstrahl generieren und diesen bis zur Lichtquelle ablaufen:

```
float shadow = (isinf(raymarchScene(pos, uLightDir, uNear, uFar).x)) ? 1.0
    : 0.0;
```

Um die Szene zu färben, definieren wir uns eine Funktion `colorScene` die abhängig von der Position und der Objekt-ID eine Farbe zurückgibt:

```
vec3 colorScene(vec3 pos, float ID) {
    switch (floatBitsToInt(ID)) {
        case BOX_ID:
            return vec3(0.6, 0.4, 0.2) * (checkerPattern(pos * 5.00000001)
                * 0.5 + 0.5);
        case BALL_ID:
            return vec3(1.0, 0.2, 0.2);
        default:
            return vec3(1.0, 0.0, 1.0);
    }
}
```

Die Box bekommt hier ein prozeduelles Karomuster als Textur zugewiesen, der Ball eine uniforme

Farbe.

Raytracing

Raymarching funktioniert gut für einfache geometrische Objekte, die wir als Formeln definieren können. Für komplexere Objekte ist es jedoch schwierig eine einfache implizite Form zu finden und es ist deutlich effizienter Schnittpunktberechnungen explizit aubszuführen. Wir limitieren uns in dieser Übung auf Schnittpunktberechnungen zwischen Strahl und Dreieck und benutzen diese im Folgenden um eine eingelesene OBJ-File auf der Grafikkarte zu raytracen.

Dazu müssen wir zuerst eine OBJ-File auslesen. Das Framework stellt dafür die Funktion `ObjParser::parse` bereit:

```
void loadObj(Program& program, const std::string& filename) {
    std::vector<Mesh::VertexPCN> vertices;
    std::vector<unsigned int> indices;
    ObjParser::parse(filename, vertices, indices);

    GLuint triangleCount = indices.size() / 3;

    // Vertices an GPU übergeben
    glProgramUniform4fv(program.handle, program.uniform("uVertices"),
        vertices.size() * 2, value_ptr(vertices[0].position));

    // Indices an GPU übergeben
    glProgramUniform3uiv(program.handle, program.uniform("uIndices"),
        triangleCount, indices.data());

    // Anazahl Dreiecke übergeben
    program.set("uTriangleCount", triangleCount);
}
```

`ObjParser::parse` liest dabei die Vertices als `Mesh::VertexPCN` ein:

```
struct VertexPCN {
    glm::vec3 position;
    glm::vec2 texCoord;
    glm::vec3 normal;
};
```

Wir fassen jetzt jeweils `(position.xyz, texCoord.x)` und `(texCoord.y, normal.xyz)` zu einem `vec4` zusammen und übergeben das ganze Konstrukt dann als Array an unseren Shader, was wir mit einem Aufruf von `glProgramUniform4fv` bewerkstelligen können, da `std::vector` seine Inhalte linear im Speicher anordnet.

Analog fassen wir je drei aufeinanderfolgende Indices zu einem `uvec3` zusammen und übergeben diese an den Shader.

Das Codegegenstück im Shader sieht wie folgt aus:

```
const uint MAX_TRIANGLES = 100u;
uniform vec4 uVertices[MAX_TRIANGLES * 3u * 2u];
uniform uvec3 uIndices[MAX_TRIANGLES];
uniform uint uTriangleCount;
```

Beachtet, dass Arrays in GLSL eine zur Compile-Zeit vordefinierte Größe haben müssen und die Anzahl an Uniform-Komponenten durch `GL_MAX_FRAGMENT_UNIFORM_COMPONENTS` begrenzt ist. Um mehr Daten an die Grafikkarte zu übertragen, braucht man sogenannte Uniform-Buffer-Objekte (UBO) oder Shader-Storage-Buffer-Objekte (SSBO), für unser Beispiel reichen jedoch die Uniform-Slots der GPU, da unser unoptimierter Raytracer sowieso nicht schnell genug sein wird um größere Modelle zu rendern.

Die Strahlerzeugung funktioniert hier wieder genauso wie beim Raymarching, in unserer Renderloop laufen wir jetzt jedoch einmal über jedes Dreieck:

```
float depth = uFar;
for (uint i = 0u; i < uTriangleCount; i++) {

    // Vertexpositionen auslesen
    vec3 v0 = uVertices[uIndices[i].x * 2u].xyz;
    vec3 v1 = uVertices[uIndices[i].y * 2u].xyz;
    vec3 v2 = uVertices[uIndices[i].z * 2u].xyz;

    // Schnittpunktberechnung -> vec3(u, v, t)
    vec3 result = intersectTriangle(rayOrigin, rayDir, v0, v1, v2);

    // Übermalen, wenn das Dreieck näher ist
    if (result.z < depth) {
        depth = result.z;
        vec3 barycentrics = vec3(1.0 - result.x - result.y, result.xy);

        // Interpolieren der Normalen
        vec3 n0 = uVertices[uIndices[i].x * 2u + 1u].yzw;
        vec3 n1 = uVertices[uIndices[i].y * 2u + 1u].yzw;
        vec3 n2 = uVertices[uIndices[i].z * 2u + 1u].yzw;
        vec3 normal = normalize(mat3(n0, n1, n2) * barycentrics);
        // = normalize(n0 * barycentrics.x + n1 * barycentrics.y + n2 *
           barycentrics.z);

        // Beleuchtungsberechnung
        fragColor = max(dot(normal, uLightDir), 0.0) * uLightColor + vec3
           (0.005);
    }
}
```

Wenn wir einen Schnittpunkt gefunden haben und dieser näher am Betrachter liegt als der vorherige Schnittpunkt, benutzen wir die baryzentrischen Koordinaten, die wir während der Schnittpunktberechnung ermitteln, um die Normalen der Eckpunkte zu interpolieren und die Normale unseres Fragments

zu ermitteln.