Farben

Der gesamte Code befindet sich in `cgintro-2-tutorial.zip`

In dieser Übung wollen wir uns ein bisschen mit Farben befassen und dabei direkt lernen wie das Zusammenspiel von *Vertex*- und *Fragment*-Shadern in OpenGL funktioniert.

Dabei müssen wir unsere Daten erst einmal um Farbdaten erweitern – genauso wie wir in der letzten Übung für jeden *Vertex* eine Position im Raum definiert haben können wir zusätzlich jedem Vertex eine Farbe zuweisen. Ein Vertex ist aber zunächst einmal nur ein Punkt im Raum ohne Fläche oder Volumen. Wenn wir von Farbe reden meinen wir aber die Farbe von Objektoberflächen.

Da letztere in OpenGL aus *Primitiven* (i.d.R. Dreiecken) zusammengesetzt werden, müssen wir also unsere für die Vertices definierten Farben irgendwie auf die daraus zusammengebauten Dreiecke erweitern.

Praktischerweise wird diese Interpolation von Attributen (wie eben der Farbe) von OpenGL für uns erledigt (siehe Beschreibung der Rendering Pipeline in der letzten Übung). Wir müssen lediglich in Vertex- und Fragment-Shader angeben, welche Attribute zu interpolieren sind und das Ergebnis dann im Fragment-Shader verwenden. Aber fangen wir zunächst mit den Vertex-Attributen an.

Verschachtelte Vertex-Attribute

In dieser Lektion benutzen wir das Framework, um uns von manueller Speicher- und Pointerverwaltung zu befreien, und die Allokation und Freigabe von OpenGL Ressourcen in C++ Konstruktoren/Destruktoren auszulagern. Dafür benutzen wir die Klassen `Program`, `Buffer` und `VertexArray`, diese Klassen sind simple Wrapper um einen Pointer, der auf die zugehörige OpenGL Ressource verweist. Die Designphilosophie dahinter nennt man Ressourcenbelegung ist Initialisierung, für OpenGL Ressourcen müssen wir noch ein paar Ausnahmen beachten.

Was uns dieses Designkonzept im Endeffekt ermöglicht ist, uns von `glGen`- und `glDelete`-Aufrufen zu befreien.

Im Header unserer Anwendung legen wir dafür einfach Felder für alle Ressourcen an, die unser Programm benötigt:

```
class MainApp : public App {
    // ...
   private:
    Program shaderProgram; // Shader Programm
    VertexArray vao; // Vertex Array
    Buffer vbo; // Vertex Buffer
    Buffer ebo; // Index/Element Buffer
```

```
};
```

Das Framework lagert außerdem die Initialisierung von GLFW und OpenGL in eine Überklasse `App` aus. Wir müssen uns nur noch darum kümmern unsere Ressourcen im Konstruktor zu laden und die Renderschleife `render()` zu überschreiben.

Der Code zum Laden eines Dreiecks aus der letzten Übung wird zu:

```
const std::vector<float> VERTICES = {
    -0.6f, -0.5f, 0.0f,
     0.6f, -0.5f, 0.0f,
     0.0f,  0.5f, 0.0f,
};

const std::vector<unsigned int> INDICES = {
    0, 1, 2,
};

// VertexBufferObject erstellen
vbo.load(Buffer::Type::ARRAY_BUFFER, VERTICES);
// Statt: unsigned int vbo = makeBuffer(GL_ARRAY_BUFFER, GL_STATIC_DRAW,
   sizeof(vertices), vertices);

// ElementBufferObject erstellen
ebo.load(Buffer::Type::INDEX_BUFFER, INDICES);
// Statt: unsigned int ebo = makeBuffer(GL_INDEX_BUFFER, GL_STATIC_DRAW,
   sizeof(indices), indices);

// Binde Buffer an das VertexArrayObject
vao.bind();
vbo.bind(Buffer::Type::ARRAY_BUFFER);
ebo.bind(Buffer::Type::INDEX_BUFFER);

// Definition der Attribute auf Basis der Daten
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)
    0);
glEnableVertexAttribArray(0);
```

Farben hinzuzufügen ist hier eigentlich sehr einfach. Zunächst einmal brauchen wir Farbdaten:

```
const std::vector<float> VERTICES = {
    -0.6f, -0.5f, 0.0f, 1.f, 0.f, 0.f,
     0.6f, -0.5f, 0.0f, 0.f, 1.f, 0.f,
     0.0f,  0.5f, 0.0f, 0.f, 0.f, 1.f,
};
```

In obigen Codebeispiel wurden einfach hinter jede Position noch 3 weitere Werte (für rot, grün, blau) geschrieben, sodass wir jetzt pro vertex 6 `float` Werte haben. Beim Mapping der Attribute auf die Daten sind dann analog zwei Attribute zu definieren:

```
size_t stride = 6 * sizeof(float);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, stride, (void*) (0 *
    sizeof(float)));
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, stride, (void*) (3 *
    sizeof(float)));
glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);
```

Das erste Attribut welches wir mit `glVertexAttribPointer` definieren, ist immer noch die Position, nur dass wir beim *stride* Parameter jetzt angeben müssen, dass ein Punkt 6 Werte hat. Die Parameter der ersten Zeile bedeuten also:

1. Wir definieren das erste Attribut (mit Index 0).
2. Es hat 3 Werte…
3. … vom Typ **`float`**.
4. Der Wert wird nicht normalisiert, sondern so wie er ist an den Shader übergeben.
5. Der Abstand vom ersten zum zweiten Punkt ist 6-mal die Größe eines **`float`**.
6. Das Attribut beginnt beim Offset 0 (pro Vertex kommt das Attribut als erstes).

Analog gibt die zweite Zeile dann die entsprechenden Werte für die Farbe an. In diesem Fall 3 **`float`** und der Offset pro Vertex ist 3-mal die Größe eines **`float`**. An sich recht einfach – beim Schreiben dieser Zeilen kann man sich dennoch leicht mal vertun. Unpraktischerweise wird man dann bei einem Fehler selten mit einer Fehlermeldung belohnt…

sRGB

In der ersten Vorlesung habt Ihr schon etwas über Farbräume gehört. Das Framework aktiviert standardmäßig sRGB-Rendering (mit `glfwWindowHint(GLFW_SRGB_CAPABLE, GLFW_TRUE);` und `glEnable(GL_FRAMEBUFFER_SRGB);`), d.h. im Shader sind alle Farbvariablen physikalisch richtig linear und erst im letzten Schritt passt OpenGL die Farbausgabe an die nicht-lineare menschliche Wahrnehmung an, sendet die nicht-linearen Werte an den Monitor und der Monitor konvertiert die Farben dann wieder in lineare Helligkeitswerte. Das ermöglicht effiziente Kodierung von wahrnehmbaren Farben, aber hält Berechnungen mit Farben im Shader deutlich einfacher. Ihr müsst jedoch darauf achten, dass Farben die Ihr z.B. als Vertex-Attribute übergebt vielleicht anders aussehen als geplant. Wir werden nochmal auf sRGB-Farben eingehen, wenn es um Texturen geht.

Shader und Interpolation

Damit haben wir jetzt pro Vertex zwei Attribute und müssen daher entsprechend auch in den Shadern damit umgehen. Im Vertex-Shader sieht das so aus:

```
#version 330 core

layout (location = 0) in vec3 position;
layout (location = 1) in vec3 color;

out vec3 interpColor;

uniform float uTime;

vec3 animateColor(vec3 color, float time) {
    float lerp = 0.5 * (sin(time) + 1.0);
    return mix(color.rgb, color.gbr, lerp);
}

void main() {
    gl_Position = vec4(position.x, position.y, position.z, 1.0);
    interpColor = animateColor(color, uTime);
}
```

Oben haben wir jetzt analog zur `position` einen 3-dimensionalen Vektor für die Farbe. Der fundamentale Unterschied ist aber, dass wir jetzt auch einen expliziten Output (`interpColor`) haben welchen wir mittels `out` als solchen deklarieren. Um die Variable `uTime` und die Funktion `animateColor()` kümmern wir uns gleich. Sofern nicht explizit annotiert, werden Output-Variablen des Vertex-Shaders pro Primitiv interpoliert und an den Fragment-Shader übergeben:

```
#version 330 core

in vec3 interpColor;

out vec3 fragColor;

void main() {
    fragColor = interpColor;
}
```

Dort muss eine entsprechende input Variable (mit `in` annotiert) mit *gleichem* Namen existieren, welche dann die interpolierten Werte beinhaltet. In diesem Fall geben wir die interpolierte Farbe zurück und erhalten so pro Pixel die interpolierte Farbe:

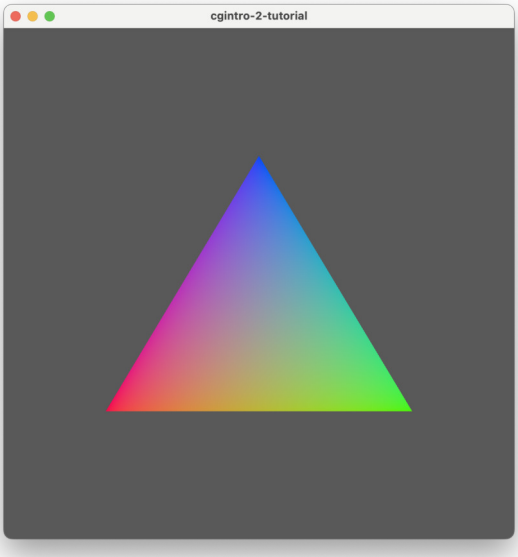

Abbildung 1: Interpolierte Farben.

Uniform Shader Variables

Beim Blick in den Vertex Shader haben wir schon eine Variable mit dem Präfix `uniform` gesehen und mit solchen wollen wir uns nun näher befassen. Denn im Gegensatz zur Position und Farbe, die ja natürlich variieren, sind andere Variablen zumindest in derselben Iteration gleich. Dazu zählen unter anderem die Zeit bzw. der Index des gerade zu rendernden Frames. Bei interaktiven Medien kann es auch vorteilhaft sein, den Shadern die Mausposition oder Keyboardevents mitzuteilen. Es können also zahlreiche Informationen an den Shader weitergereicht werden. Hierzu wird zuerst ein handle deklariert, mit dem wir die Variable identifizieren können:

```
GLuint uTime = glGetUniformLocation(shaderProgram.handle, "uTime");
```

Diese *location* kann man analog zu den handles die wir für VBOs/IBOs/etc. haben nutzen, um die Variable zu verändern. Zum hochladen von Daten auf die GPU nutzen wir dafür je nach Art der Variablen verschiedene Varianten der `glUniform*` Funktionen. Unsere Fließkommazahl können wir mittels `glUniform1f` hochladen (wobei `glfwGetTime()` die Zeit in Sekunden seit Start zurückgibt):

```
glUniform1f(uTime, static_cast<float>(glfwGetTime()));
```

Diese Codezeile muss natürlich bei jeder Iteration im Event Loop ausgeführt werden, da die Zeit für unsere Animation sonst “still steht”. Damit sind wir dann aber auch fertig.

Werden wir nun noch einen letzten Blick in die Funktion `animateColor()`:

```
vec3 animateColor(vec3 color, float time) {
    float lerp = 0.5 * (sin(time) + 1.0);
    return mix(color.rgb, color.gbr, lerp);
}
```

Die Funktion wählt einen Wert zwischen 0 und 1 abhängig von `uTime` und “mischt” anschließend die Farben von `color` neu. Auffällig ist hier die Art, wie wir die Elemente von `c` adressieren: Anstatt `xyzw` lässt sich nämlich auch `rgba` (z.B. für Farben), sowie `stpq` (für Koordinaten) verwenden, was den Code deutlich lesbarer gestalten kann. Eine weitere nützliche Eigenschaft von GLSL Vektoren ist das sogenannte Swizzling.