Das Setup

Voraussetzungen

Die Übungen werden in C++ geschrieben (eigentlich werden wir eher in C schreiben, aber an vielen Stellen auf C++ zurück greifen), daher wird zunächst ein C++ Compiler benötigt. Unter Windows empfiehlt sich da der mit Visual Studio gelieferte Compiler (getestet wurden die Übungen mit der kostenlos erhältlichen Community Edition von Visual Studio 2017).

Alternativ bietet sich Visual Studio Code mit den C++ und CMake Extensions als kostenlose IDE an.

Unter Linux wären gcc oder clang zu empfehlende Compiler; unter macOS/BSD ist clang am weitesten verbreitet (clang kommt zusammen mit den Xcode Command Line Tools, die sich mit `xcode-select --install` installieren lassen). Getestet wurden die Übungen hier in einem Linux (Arch Linux) mit gcc und macOS mit clang.

Des weiteren wird eine Installation von cmake angenommen um platform-spezifische Projektdateien zu generieren. Zur Benutzung kommen wir gleich weiter unten.

Sobald die Abhängigkeiten installiert sind muss von eCampus die *framework.zip* heruntergeladen und entpackt werden. Das Framework bildet die Basis, auf der alle Übungen aufbauen. Für die erste Übung (wie auch alle Folgeübungen) muss zusätzlich die *uebung1_sources.zip* heruntergeladen und in den Framework-Ordner entpackt werden (die Archive dieser und aller folgenden Übungen haben die selbe Verzeichnisstruktur wie das Framework und beinhalten nur die Dateien die neu sind oder für die Übung geändert wurden). Danach geht es ab jetzt platform-spezifisch weiter.

Linux

Öffnet ein Terminal und wechselt in den build Unterordner des entpackten Verzeichnisses:

```
> cd /path/to/framework/build
```

(das > heißt hier immer, dass es sich um eine Terminal-Eingabe handelt und ist nicht mit zu tippen)

Dann können mit cmake die benötigten Dateien zum bauen generiert werden und anschließend das Projekt gebaut werden. Beim ersten mal werden dabei auch eben erwähnte Abhängigkeiten (libraries für lineare Algebra, model loading etc.) gebaut - das kann gerne auch mal länger dauern.

```
> cmake ..
> make
```

Das Bauen der Abhängigkeiten wird möglicherweise auch eine Menge Warnungen ausgeben - die könnt Ihr ignorieren. Wenn alles gebaut hat müsstet Ihr mit

```
> ./uebung_1a
```

das erste Programm starten können und hoffentlich ein leeres graues Fenster angezeigt bekommen. Wenn nicht bitte beim Tutor melden und am besten etwaige Ausgaben mitliefern.

Falls das geklappt hat geht es weiter im nächsten Kapitel (*Ein leeres Fenster*).

Windows

Wenn Ihr den Code entpackt habt sollte er so aussehen:

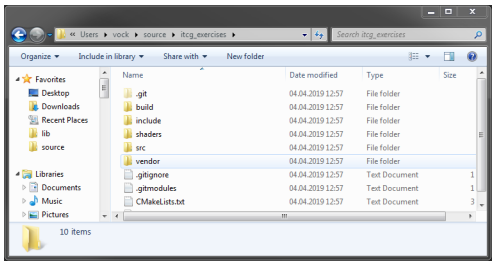

Abbildung 1: Framework Ordnerstruktur.

Startet nun die CMake GUI, gibt oben als source folder den gerade erstellten *framework* Ordner an und als build folder den Unterordner *framework/build*.

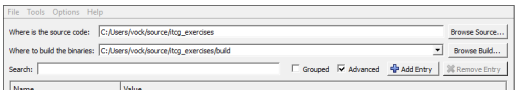

Abbildung 2: Pfadkonfiguration (im Screenshot ist *"itcg_exercises"* der Framework-Ordner).

Klickt dann auf *Configure*. möglicherweise kommt ein Fenster wo ihr den Compiler auswählen müsst. Im Falle von Visual Studio 2017 sähe das dann so aus:

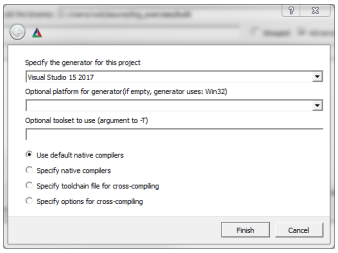

Abbildung 3: Compiler-Auswahl.

Danach wird das Projekt initial konfiguriert – ignoriert dass alles rot ist und klickt nochmal auf *Configure* (ja, das ist etwas seltsam). Zuletzt müsst ihr noch auf *Generate* klicken um die Visual Studio Projektdateien zu erstellen:

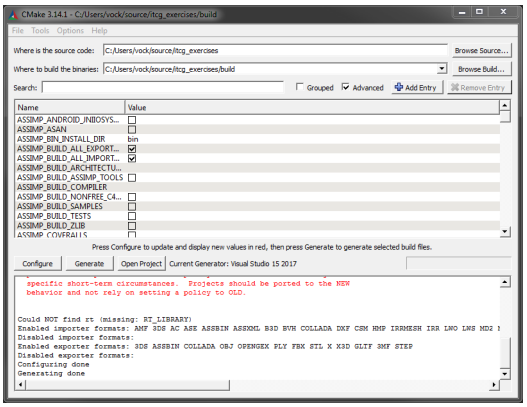

Abbildung 4: CMake nach *"Generate"*.

Jene Projektdateien wurden im *build* Unterordner angelegt, wo Ihr auch die Solution Datei mit Endung *.sln* öffnen könnt:

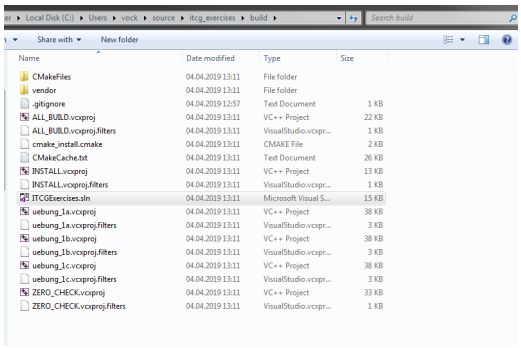

Abbildung 5: Generierte Projektdateien

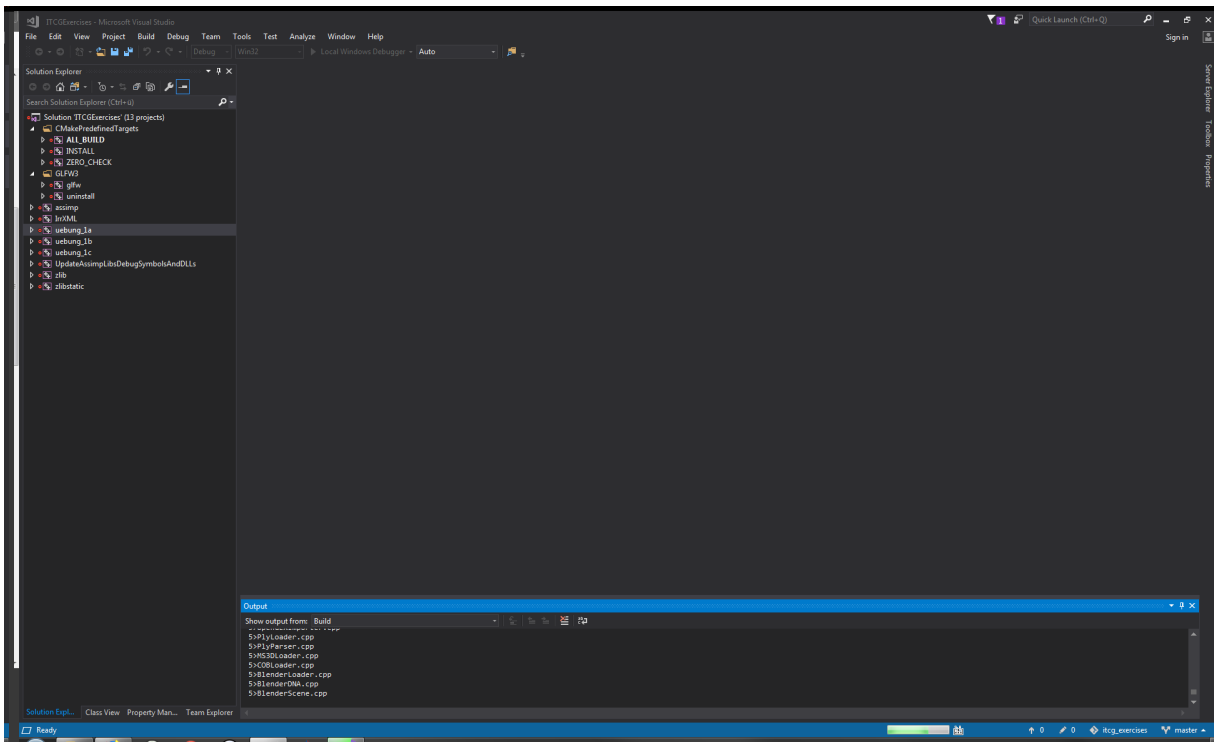

Abbildung 6: Solution in VS

Wenn Ihr links im *Solution Explorer* die erste Übung (*uebung_1a*) rechtsklickt könnt Ihr mit *Build* die Übung bauen. Beim ersten mal werden dabei auch alle Abhängigkeiten gebaut, was eine Weile dauern kann aber irgendwann hoffentlich erfolgreich ist:

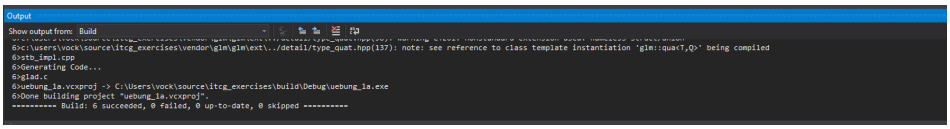

Abbildung 7: Kompilation war erfolgreich.

Um das erste Programm (ein leeres Fenster) zu starten, könnt Ihr wieder rechtsklick auf die Übung machen und das Projekt per *Set as StartUp Project* zum aktuellen Projekt machen. Danach reicht es einmal *F5* zu drücken und das Projekt sollte im Debugger gestartet werden.

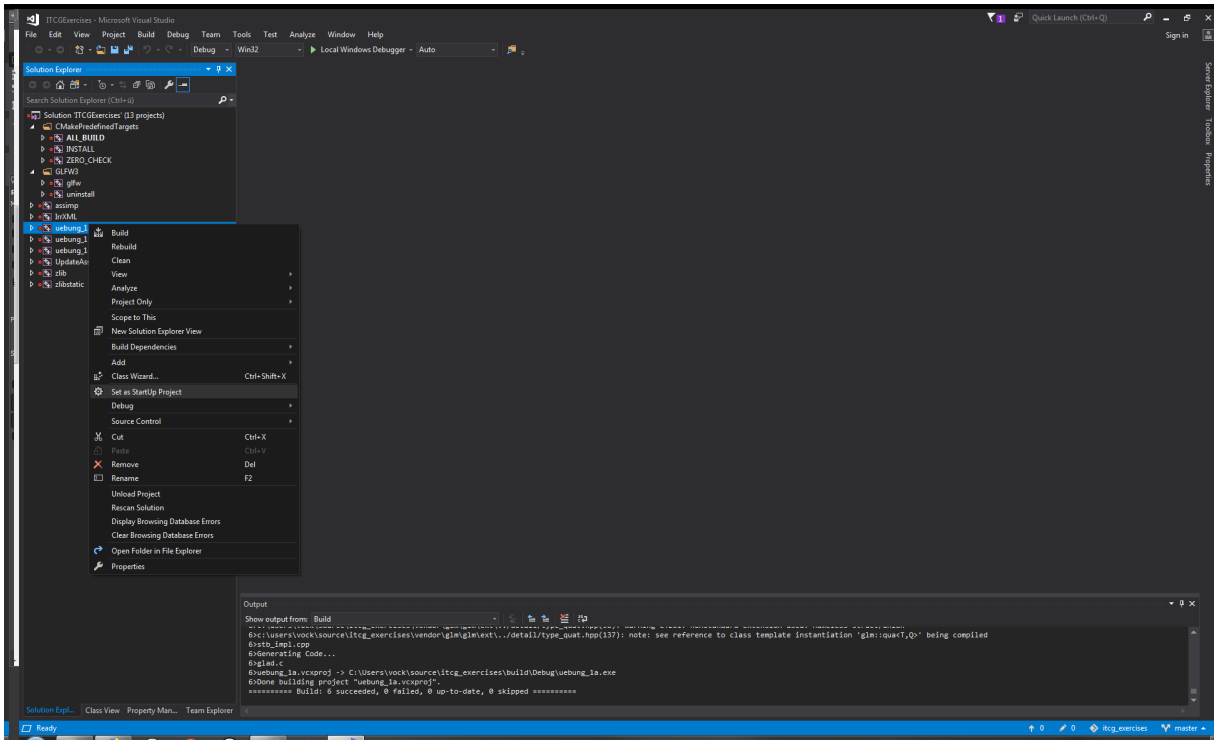

Abbildung 8: Set as StartUp Project.

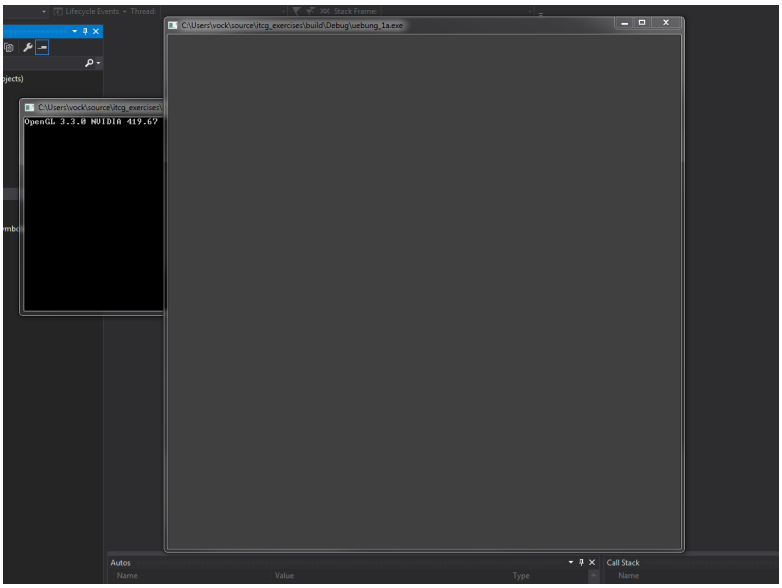

Abbildung 9: Ein leeres Fenster.

Falls das geklappt hat geht es weiter im nächsten Kapitel (*Projekt-/Übungsstruktur*).

Projekt-/Übungsstruktur

Die Ordnerstruktur aus dem Github Repository wird sich im Laufe der Übungen nicht mehr wesentlich ändern - daher lohnt es sich jetzt ein paar Worte darüber zu verlieren.

Alles was sich im *vendor* Ordner befindet gehört zu den mitgelieferten Bibliotheken und kann für die Übungen ignoriert werden. Dennoch werden wir die entsprechenden Bibliotheken bei der ersten Verwendung kurz beschreiben, verlinken und eventuell Dokumentation ihrer Funktionen nachliefern. Für jetzt sind aber das *include* und das *src* Verzeichnis interessanter:

Ersteres beinhaltet alle *Header* Dateien, also vorwiegend die *Deklarationen* von Funktionen welche jetzt noch überschaubar sind, in Zukunft jedoch wesentlich mehr werden. Zweiteres beinhaltet die *Source* Dateien mit den *Definitionen*, also den Implementationen erwähnter Funktionen. Die *.cpp* Dateien direkt im *src* Ordner sind die kleinen Programme die wir im Laufe der Übungen schreiben werden und sind daher die wichtigsten.

Grundsätzlich werden die Übungen immer so aussehen, dass mit jedem Aufgabenblatt dort neue Quelldateien hinzukommen, welche neue Funktionalitäten von OpenGL erklären und von Euch zu erweitern sind. Damit der Inhalt dieser Dateien aber nicht immer größer wird werden wir häufig verwendete Funktionalität in Funktionen/Klassen auslagern und uns so eine eigene **library** bauen (deren Quelldateien in *src/library* liegen). Mehr dazu weiter unten bei der ersten Übung.

Der Unterordner *src/vendor* ist wieder für externe Abhängigkeiten und kann ignoriert werden.

Für die erste Übung sind zunächst nur die *src/uebung_XY.cpp* Dateien interessant. In der *uebung_1a.cpp* werden wir uns mit der Erstellung eines Fensters befassen was den Inhalt des nächsten Kapitels darstellt.

Ein leeres Fenster

(auch wenn wir in diesem Kapitel relevante Code-Stellen zitieren werden ist es ratsam die uebung_1a.cpp Datei zu öffnen um parallel einen Überblick über den Code zu haben)

Um mit OpenGL irgendetwas machen zu können, bedarf es eines sogenannten **Kontexts** in dem die unzähligen Variablen (der sog. *state*) als auch Informationen über das Ausgabemedium (die sog. *surface*, i.d.R. der Fensterinhalt) gespeichert werden. Letzteres ist so eng mit dem Kontext verbunden, dass es teilweise erstaunlich schwer ist ohne Fenster (z.B. in ein Bild) zu rendern – üblicherweise wird zu diesem Zweck ein unsichtbares Fenster erstellt.

Diese Kontext- und Fenstererstellung ist je nach Plattform völlig unterschiedlich, weshalb sich extra für diesen libraries etabliert haben. Die bekanntesten sind

- glfw (was wir verwenden werden)
- freeglut
- SDL (was noch wesentlich mehr kann)

Mit glfw sehen die ersten Zeilen eines OpenGL Programms üblicherweise dann so aus:

```
glfwInit();
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 0);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
GLFWwindow* window = glfwCreateWindow(WINDOW_WIDTH, WINDOW_HEIGHT, argv
    [0], nullptr, nullptr);

if (window == nullptr) {
    std::cerr << "Failed to Create OpenGL Context" << std::endl;
    std::terminate();
}

// create context and load OpenGL functions
glfwMakeContextCurrent(window);
glfwSetFramebufferSizeCallback(window, resize_callback);

gladLoadGL();
```

(im Code werdet ihr sehen, dass wir noch eine extra Zeile für macOS mit drin haben - das ist der traditionell miserablen OpenGL Implementation von Apple geschuldet)

Die erste Zeile (das `glewInit`) ist relativ selbsterklärend eine Funktion die intern den State von OpenGL (und glfw) initialisiert. Die folgenden 3 Zeilen sind allerdings schon interessanter, denn sie definieren welche "Art" von OpenGL wir gerne programmieren würden. OpenGL ist – trotz den Namens – keine library, sondern ein Standard der von einem Konsortium von vorwiegend Firmen (die sog. Khronos Group) stetig weiterentwickelt wird. Die Implementation selber wird dann meistens von den Treiberentwicklern (nVidia, AMD, Intel, etc.) als library (die sogenannte *libgl*) geliefert. Dabei gibt es einen festgelegten Funktionsumfang der von allen implementiert werden muss (das sogenannte *Core Profile*), sowie optionale *extensions* die allerdings einzeln geladen werden müssen und eben nicht notwendigerweise vom Treiberhersteller implementiert sind.

Und dann gibt es da noch die ganzen Altlasten an Funktionalität die sich als Fehler herausgestellt haben und unter Umständen noch in altem Code benutzt werden (das sogenannte *Compat Profile*). Spätestens seit OpenGL mit Version 3 erhebliche Teile der OpenGL Funktionalität programmierbar gemacht hat (über sogenannte *shader*) ist aber dringend von diesem alten Profil abzuraten. Wir wählen hier entsprechend Version 4.0, das Core Profile und erstellen dann unser Fenster mit `glfwCreateWindow`, was uns einen Pointer auf dar erstellte Fenster gibt (was auch schief gehen kann, dann hat der Pointer den Wert `nullptr`).

Ein Blick in die Dokumentation der Funktion zeigt uns dass sie folgende Parameter bekommt:

```
glfwCreateWindow (int width, int height, const char *title, GLFWmonitor *
    monitor, GLFWwindow *share)
```

Das wären die Breite des Fensters, die Höhe des Fensters, der Titel (wir verwenden hier das 0te Argument des Programmes, was der Name des Programms bei Aufruf ist) und zwei weiteren Parametern die für fullscreen mode bzw. context sharing wichtig wären (in unserem Fall aber nicht gebraucht werden). Den resultierenden Pointer testen wir auf Erfolg (`!= nullptr`) und sagen glfw per *glfwMakeContextCurrent(window)* dass wir das Fenster gerne zum Rendern benutzen wollen (wenn man mehrere Fenster hat müsste man das Fenster jedes mal vor dem rendern hiermit wechseln).

Die darauf folgende Zeile

```
glfwSetFramebufferSizeCallback(window, resize_callback);
```

ist wichtig, da wir bei der Erstellung des Fensters zwar eine Größe (per `width` und `height`) angegeben haben, diese sich aber natürlich ändern kann (sofern wir das nicht explizit verbieten darf der Benutzer ja die Fenstergröße ändern). Üblicherweise regelt man solche Fälle mit Hilfe einer *callback Funktion* die bei einem *resize event* mit der neuen Fenstergröße aufgerufen wird. Diese wird bei uns vor der `main` Funktion deklariert und dahinter definiert:

```
void resize_callback(GLFWwindow*, int width, int height) {
    glViewport(0, 0, width, height);
}
```

Diese Funktion wird bei jedem resize aufgerufen und teilt OpenGL mittels `glViewport` mit wie groß das Fenster ist. Genau genommen sagen wir, dass das Fenster bei dem Pixel mit *(x,y)* Koordinaten *(0,0)* anfängt und die Breite und Höhe *(width, height)* hat.

Die letzte Zeile `gladLoadGL()` ist dann nochmal ein besonderes Thema, insbesondere da *glad* eine eigene, weitere Bibliothek ist deren Funktionalität wir hier nutzen. Weiter oben war bereits beschrieben, dass es neben der Standardfunktionalität auch noch Erweiterung gibt die gesondert geladen werden müssen. Dieser Prozess ist äußerst aufwändig, weshalb er wiederum üblicherweise mit Hilfe von Bibliotheken erledigt wird. Die bekanntesten sind

- glad
- glew
- gl3w

und welche man davon wählt ist eher Geschmackssache. In allen 3 Fällen ist eine Header Datei einzubinden und eine Zeile (bei uns, da wir glad nutzen, eben `gladLoadGL()`) auszuführen um den ansonsten recht hohen Aufwand zu vermeiden (wer mal einen Blick in die 18959 Zeilen große *vendor/glad/include/glad/glad.h* Datei wirft wird ein Gefühl für den Aufwand bekommen der da potentiell vermieden wird).

Tatsächlich reicht das schon um ein Fenster zu erstellen, da unser Programm aber hier endet würde das Fenster auch direkt wieder geschlossen werden, weshalb man in OpenGL Programmen etwas hat, was man üblicherweise um jeden Preis vermeiden will: Eine Endlosschleife. In der Rendering-Welt (bzw. eigentlich UI-Welt) nennt sich das *Rendering Loop* oder auch *Event Loop* und macht den Rest des Beispielprogrammes aus:

```
while (glfwWindowShouldClose(window) == false) {
    glClearColor(0.25f, 0.25f, 0.25f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    // render something... (here: nothing)

    glfwSwapBuffers(window);
    glfwPollEvents();
}

glfwTerminate();
```

Die Schleife wird solange ausgeführt bis das Fenster geschlossen wird (in diesem Fall wäre das Ergebnis von `glfwWindowShouldClose` gleich true), und macht in jedem Durchlauf das selbe:

1. Füllen des Fensters mit der Hintergrundfarbe
2. Rendern der Szene (wir machen hier erst mal gar nichts)
3. Der Buffer-Swap
4. Abarbeiten etwaiger *events* (hier checkt glfw ob z.B. Keyboard- oder Maus-Interaktion stattfand oder eben die Fenstergröße geändert wurde).

Die Idee ist, dass man in jedem Durchlauf in ein Bild im GPU Speicher (den sogenannten *framebuffer*) malt (das eigentlich *rendering*) und das dann anzeigt. Nun gibt es aber Timing-Probleme, vor allem, weil ein Monitor in einem festen Takt (der Bildwiederholfrequenz) neuen Input will. Wenn wir in der Zeit bis das nächste Bild gefordert wird aber z.B. nur ein halbes Bild fertig-gerendert haben würden wir auch nur dieses halbe Bild über den vorigen Inhalt drüber malen können. Das Ergebnis davon nennt sich Screen Tearing und sieht etwas unschön aus. Daher wird stattdessen ein Vorgehen gewählt was sich *Double Buffering* nennt, bei dem 2 Bilder auf der GPU gehalten werden:

- Ein Bild in das wir rendern (der sogenannte *backbuffer*) und
- ein Bild was wir zum Monitor schicken (der *frontbuffer*).

Solange wir jetzt rendern wird (potentiell mehrfach) der *frontbuffer* gezeigt und sobald wir fertig sind vertauschen wir einfach die beiden Bilder. Dieses Vertauschen der Buffer wird durch die `glfwSwapBuffers` Funktion erledigt - weshalb sie immer nach dem Zeichnen des Inhaltes ausgeführt wird.

Sobald wir das gemacht haben steht im Inhalt des *backbuffers* der Inhalt der vorher im *frontbuffer* war,

deswegen wird oft (aber nicht notwendigerweise) zu Beginn der render loop der alte Inhalt mit der Hintergrundfarbe überschrieben. Diese Hintergrundfarbe setzen wir mit `glClearColor`, wobei die Parameter der Rot-, Grün-, Blau- und Alphaanteil (in der Reihenfolge) der Farbe im Intervall [0,1] sind. Der Alphawert gibt an wie *opak*, also deckend ist. Ein Wert von 1.0 heißt völlig opak, 0.5 würde die Farbe halb transparent machen und 0.0 völlig durchsichtig. Transparenz wird aber wesentlich später behandelt, daher werden wir ein Weile immer einen Wert von 1.0 annehmen.

Im Anschluss wird dann das tatsächliche Überschreiben mit glClear erledigt. Genau genommen müssen wir hier angeben, dass wir die Farbe überschreiben wollen, weil im back- und frontbuffer nicht nur Farben, sondern auch andere Werte stehen (mehr dazu in späteren Übungen).

Dieser ganze Code zum Initialisieren, Fenster erstellen und Rendern ist zwar leicht müßig, wird sich aber im Laufe der Übungen nicht mehr wirklich ändern, weshalb der initiale Teil schon im nächsten Kapitel in eine Funktion ausgelagert wird die wir dann am Anfang nur noch aufrufen um alles in einer Zeile zu erledigen:

```
GLFWwindow* window = initOpenGL(WINDOW_WIDTH, WINDOW_HEIGHT, argv[0]);
```

Das macht dann entsprechend auch die main Funktion wieder übersichtlicher, sodass wir sie mit zusätzlichen Code (zum Rendern eines Dreiecks) wieder unübersichtlich machen können.

Ein einzelnes Dreieck

Bevor wir in den Code zum zweiten Teil einsteigen muss ein paar (ent-)warnende Worte zu diesem Kapitel:

Üblicherweise ist das Zeichnen eines Dreiecks das Äquivalent des “Hello World” Programmes in OpenGL was vor allem daran liegt, dass der Zweck von OpenGL (stark vereinfacht) ist, sehr viele Dreiecke möglichst schnell durch die gleiche *pipeline* von Funktionen durchzupressen. Das ganze wird dann dank hoher Parallelisierung von der GPU sehr effizient erledigt. Die gute Nachricht ist dabei, dass der Unterschied zwischen einem und einer Million Dreiecke aus Programmierersicht trivial ist; die schlechte Nachricht ist, dass man eigentlich schon die komplette pipeline braucht um nur dieses eine Dreieck zu rendern.

Mit anderen Worten: Die Lernkurve geht in diesem Kapitel steil nach oben, flacht aber in den folgenden Übungen dafür auch sehr schnell wieder ab. Das ist durch das Design selber bedingt, weshalb wir nicht vermeiden können hier recht viele neue Konzepte anzugucken - auch wenn ihre Verwendung noch sehr simpel gehalten ist. Genau genommen lernen wir:

- Wie OpenGL ohne Objektorientierung Objekt-orientiert arbeitet,
- wie wir Daten in rauhen Mengen auf die GPU schicken können,

- wie Teile der OpenGL pipeline durch eigene Programme verändert werden können,
- dadurch auch eine neue Programmiersprache (yep, sorry, aber die ist dafür auch nur ein stark vereinfachtes C),
- wie wir diese Sprache compilen und linken und
- auf unsere Daten GPU-seitig zugreifen können.

Wir können das ganze aber dadurch vereinfachen, dass wir nur 2-dimensional arbeiten - so bleibt uns zumindest vorerst die ganze lineare Algebra erspart (aber nicht lange).

Pipeline

Bevor wir uns dann endlich den Code angucken lohnt es sich einen Blick auf die *OpenGL Rendering Pipeline* zu werfen um einen Überblick über die zu erledigenden Aufgaben zu bekommen.

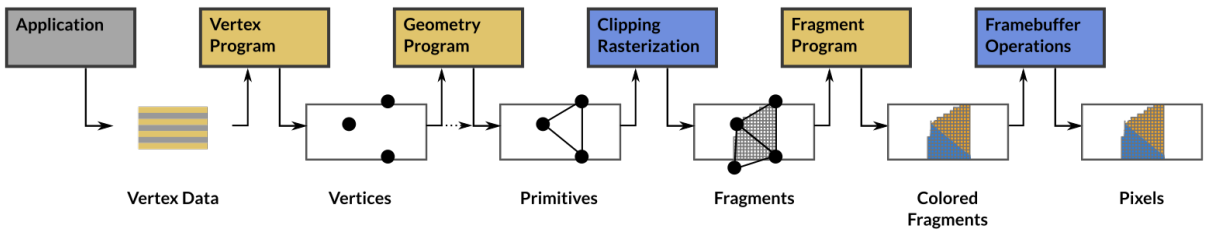

Abbildung 10: Die OpenGL Rendering Pipeline (vereinfacht)

Um Daten rendern zu können brauchen wir zunächst erst mal Daten. OpenGL ist dabei tatsächlich vollkommen egal wie die aussehen – es verlangt am Ende nur einen Pointer auf irgendwas (ein `void` Pointer um genau zu sein). Üblicherweise laden wir Bilder als Texturen hoch (das kommt in späteren Übungen) und Geometriedaten als ein oder mehrere große Arrays (in OpenGL heißen diese *vertex buffer*).

Für die Geometriedaten ist wichtig, dass am Ende daraus *vertices* gemacht werden. Ein *vertex* ist ein Punkt im Raum der mindestens ein Attribut, nämlich die Position (egal welcher Dimension) hat. Üblicherweise kommen aber noch andere Attribute dazu, wie z.B.

- Farbdaten (als RGB),
- Normalen (die Richtung orthogonal zur Tangentialebene der approximierten Oberfläche),
- Textur-Koordinaten.

Oder eben irgendwelche anderen Attribute die für die Visualisierung wichtig sind. Welche Daten zu welchen Attributen sagen wir OpenGL mittels mehrerer Funktionen dir wir in sogenannten *vertex array objects* kapseln können.

Ziel ist im ersten von uns programmierbaren *shader*, dem *vertex program*, die Daten als Eingabe-Variablen zu bekommen und ihre tatsächliche Position auszurechnen. Später, wenn wir mit (Kamera-) Transformationen arbeiten, werden wir in diesen Schritt mehr Denkarbeit stecken müssen.

Die Ausgabe davon wird von OpenGL zu vorher spezifizierten *Primitiven* (z.B. Punkte, Liniensegmente oder Dreiecke) zusammengebaut. Falls gewünscht können wir einem eigenen shader, dem *geometry program* diese Primitive noch manuell modifizieren.

Danach kommen mit dem *clipping* und der *rasterization* zwei Schritte die von uns nur mittels globaler Variablen beeinflussbar sind und in der Regel in Hardware implementiert sind. Der Zweck der *rasterization* ist es die von uns gelieferten *vertex Attribute zu interpolieren* und in *fragments* zu verwandeln. In den meisten Fällen kann man sich *fragments* als Pixel vorstellen, die wesentlichen Unterschiede sind:

- *fragments* haben noch eine Tiefe, also eine 3. Koordinate.
- für jedes Pixel im Ausgabebild am Ende können mehrere *fragments* zuständig sein, was wichtig für *Antialiasing* ist.

Bei dieser Rasterisierung werden auch direkt vertices und fragments welche am Ende nicht sichtbar sind verworfen (das sogenannte *clipping*, was später in der Vorlesung ausführlichen besprochen wird).

Ergebnis sind mehrere *fragments* welchen im nächsten – von uns wieder programmierbaren – Schritt, dem *fragment program* eine Farbe zugewiesen wird. Üblicherweise wird diese Farbe abhängig von simulierten Beleuchtungsmodellen bestimmt (das sogenannte *shading*).

An der Stelle sind wir dann an sich fertig – OpenGL allerdings abhängig davon was wir noch für Optionen vorher gesetzt haben nicht. So müssen z.B. mehrere übereinander liegende *fragments* (wie gesagt, diese haben noch eine Tiefe) in einen Pixel verwandelt werden, was je nachdem ob sie (semi-) transparent sind oder nicht erstaunlich kompliziert wird. Außerdem können noch Teile des Bildes oder sogar einzelne Pixel doch auf Basis von Maskierungsoperationen verworfen und möglicherweise in ein anderes Farbformat konvertiert werden.

Ist das geschehen werden die resultierenden *pixel* in den gebundenen *framebuffer* (das kann das Fenster sein, oder auch ein Bild) geschrieben und die pipeline ist einmal fertig durchgelaufen. Eben jenes Durchlaufen aller Schritte der Pipeline wird als *render pass* bezeichnet - später werden wir nicht nur einen, sondern mehrere *render passes* haben, jetzt reicht uns aber erst mal einer.

Und das ganze als Code

(an dieser Stelle bietet es sich wieder an die uebung_1b.cpp Datei zu öffnen um parallel einen Überblick über den Code zu haben)

Zu Beginn initialisieren wir zunächst wie im vorigen Beispiel den OpenGL Kontext und erstellen ein Fenster. Das ganze ist diesmal aber in eine Funktion `initOpenGL` gekapselt (die man sich in *src/library/common.cpp* angucken kann:

```
GLFWwindow*
initOpenGL(int width, int height, const char* title) {
    // Implementation wie in uebung_1a.cpp
}
```

Damit können wir uns in dann direkt ein Fenster handle geben lassen und wieder den callback für den *window resize* Fall setzen:

```
GLFWwindow* window = initOpenGL(WINDOW_WIDTH, WINDOW_HEIGHT, argv[0]);
glfwSetFramebufferSizeCallback(window, resizeCallback);
```

Vertex Shader

Danach kommt dann auch direkt für uns neuer Code. Da wir diesmal tatsächlich Geometrie rendern wollen kommen wir nicht an *shadern* vorbei, um genau zu sein brauchen wir einen *vertex shader* und einen *fragment shader*, die wir laden, compilen und zusammen in ein *shader program* linken. Der erste *shader* in der Pipeline ist wie oben in der Pipeline-Abbildung zu sehen der vertex shader, den wir in der Datei *shader/uebung_1b.vert* definieren:

```
#version 330 core
layout (location = 0) in vec3 position;

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

Shader werden für OpenGL in einer eigenen Programmiersprache, der GL Shading Language (kurz GLSL) geschrieben. An sich ist das ein stark vereinfachtes C mit ein paar besonderen Sprachkonstrukten. Ganz oben wird i.d.R. zunächst einmal die shader Version (die korrespondiert zu bestimmten OpenGL Version) spezifiziert.

Danach folgt die Deklaration von input/output Variablen und genau einmal die Definition der `main()` Funktion. In diesem Fall wollen wir ein Dreieck rendern, wofür wir mindestens die *Position* der vertices kennen müssen. Das heißt wir definieren eine **input** Variable vom Type `vec3` (ein dreidimensionaler Vektor). Die konkrete Zeile sieht so aus:

```
layout (location = 0) in vec3 position;
```

Der Teil vorne (das `layout (location = 0)`) ist spezieller GLSL Syntax mit dem wir diesem Attribut

die Zahl 0 sozusagen als Adresse der Variable zuweisen. Später können wir dann diese Adresse nutzen um unsere tatsächlichen Daten damit zu verknüpfen. Über `position.x` können wir so dann z.B. auf die x-Koordinate jedes Vertex zugreifen für den der shader durchlaufen wird. Angenommen wir wollten noch ein zweites Attribut – z.B. die Farbe des vertex – hinzufügen, dann könnte dieser Teil so aussehen:

```
layout (location = 0) in vec3 position;
layout (location = 1) in vec4 color;
```

In diesem Fall würden wir eine andere Adresse (=1), z.B. einen vierdimensionalen Vektor (R, G, B und Alpha), und natürlich einen anderen Namen wählen. Für jetzt reicht uns aber die Position.

In der `main()` Funktion müssen wir dann die eigentliche Aufgabe des shaders erledigen, die es primär ist der speziellen, eingebauten Variable `gl_Position` den korrekten Wert zuzuweisen. Aus Gründen die wir später sehr ausführlich behandeln werden sind Positionen in OpenGL i.d.R. vierdimensionale Vektoren, wo die 4. Koordinate (die *w-Koordinate*) mit 1 initialisiert wird. Da die Ausgabe des shaders die Position des vertex sein soll schreiben wir in diesem Fall genau das da rein: Die x-, y-, und z-Koordinaten des Punktes. Einen vierdimensionalen Punkt können wir mit

```
vec4(x, y, z, w)
```

erstellen. `gl_Position` ist übrigens nicht die einzige eingebaute Variable, es gibt mehrere vordefinierte Ein- und Ausgabevariablen, die hier genauer dokumentiert sind. Allerdings wird dort möglicherweise noch nicht alles verständlich sein.

Fragment Shader

Wo wir uns gerade eh GLSL angucken ist es vielleicht sinnvoll direkt einen Blick auf den anderen shader zu werfen (*shaders/uebung_1b.frag*). Zur Erinnerung: Der *fragment shader* ist dafür zuständig Fragmenten (hier ist es wie erwähnt legitim an "Pixel" zu denken, auch wenn das nur die halbe Wahrheit ist) eine Farbe zuzuweisen. Da wir möglichst ein Minimalbeispiel besprechen wollen reicht es uns hier jedem vertex die selbe Farbe zuzuweisen:

```
#version 330 core
out vec4 frag_color;

void main()
{
    frag_color = vec4(1.0f, 0.5f, 0.2f, 1.0f);
}
```

Auch hier definieren wir zunächst die (gleiche) Version des shaders um dann die input und output Variablen zu definieren. Input Variablen wären die output Variablen des vertex shader (allerdings

interpoliert und rasterisiert in fragments) – da die Farbe aber nicht von der Position abhängt brauchen wir das hier nicht. Stattdessen haben wir nur eine output Variable: Ein vierdimensionaler Vektor für die Ausgabefarbe (wieder rot, grün, blau und alpha).

In der `main()` Funktion wird dann in diese Variable einfach eine fixe Farbe (genauer ein Pastell-Orange mit Alphawert 1) geschrieben.

Das war es dann auch schon was die shader angeht, außer natürlich, dass wir die shader auch laden, compilen und linken müssen. Für das laden und compilen haben wir bereits eine fertige Funktion (`compileShaderHelper(filename, type)`):

```
unsigned int vertexShader = compileShaderHelper("uebung_1b.vert",
    GL_VERTEX_SHADER);
unsigned int fragmentShader = compileShaderHelper("uebung_1b.frag",
    GL_FRAGMENT_SHADER);
```

Die Funktion bekommt als Parameter den Dateinamen (der Ordner ist immer *shader/*) und den Typ des shaders. Die Implementation ist unter der `main` Funktion und sieht so aus:

```
unsigned int
compileShaderHelper(const char* filename, unsigned int type) {
    const char* shaderSource = loadShaderFile(filename);

    // create shader object
    unsigned int shader = glCreateShader(type);
    glShaderSource(shader, 1, &shaderSource, NULL);
    // try to compile
    glCompileShader(shader);
    // source code is no longer needed
    delete [] shaderSource;

    // check if compilation succeeded
    int  success;
    char infoLog[512];
    glGetShaderiv(shader, GL_COMPILE_STATUS, &success);
    if(!success) {
        glGetShaderInfoLog(shader, 512, NULL, infoLog);
        std::cerr << "Shader compilation failed\n" << infoLog << std::endl
            ;
        return 0;
    }

    return shader;
}
```

Zunächst muss der Quellcode des shaders aus der Datei in einen String geladen werden – wofür wir direkt eine Funktion mitgeliefert haben die wir hier jetzt nicht besprechen werden. Der nächste Schritt ist ein shader Objekt anzulegen und ihm diesen Code zuzuweisen. Da das Handling von “Objekten”

(OpenGL ist in C geschrieben und kennt an sich keine Klassen o.ä.) in OpenGL immer gleich aussieht lohnt es sich das kurz genauer anzugucken.

Üblicherweise gibt es für jede Objektart eine Funktion die eine oder mehrere Objekte generiert und ein *handle* (immer einfach eine für dieses Objekt einzigartige Nummer) zurückgibt. In diesem Fall sieht das so aus:

```
unsigned int shader = glCreateShader(type);
```

Braucht man später dieses Objekt nicht mehr kann man es mit einer korrespondierenden Funktion wieder zerstören – bei shadern sieht das so aus:

```
glDeleteShader(shader);
```

Dazwischen können wir eine Menge von Funktion aufrufen die etwas mit diesem Objekt machen. Dafür gibt es 2 Methoden, die klassische und die moderne. Die moderne Variante sähe so aus:

```
doSomethingWithShader(shader, possibly, more, parameters);
```

Da wir der klassischen Methode auch begegnen werden müssen wir die allerdings auch kurz besprechen. Diese, oft leicht nervige Methode basiert auf der Tatsache, dass OpenGL wie eine state machine designed wurde. Die Idee ist, dass es immer ein Objekt gibt was gerade “aktuell” ist und auf das alle aufgerufenen Funktionen arbeiten. Das setzen des gerade aktuellen Objektes nennt sich *bind* (das Gegenteil wäre *release*) und ein Pseudobeispiel sähe dann so aus:

```
glBindObject(handle);
// Funktionen arbeiten von hier an mit dem shader "shader"
doSomethingWithObject(some, relevant, parameters);
glBindObject(0); // release, setzt aktuellen shader auf "keinen"
```

Im Falle unseres shaders setzen wir mit `glShaderSource` den code, und kompilieren ihn mit `glCompileShader`:

```
unsigned int shader = glCreateShader(type);
glShaderSource(shader, 1, &shaderSource, NULL);
glCompileShader(shader);
delete [] shaderSource;
```

Nach dem Kompilieren wird der string mit dem source code nicht mehr benötigt und daher gelöscht.

An sich würde das jetzt reichen, allerdings gibt es in der Funktion noch ein paar Zeilen um zu testen ob das Kompilieren erfolgreich war und gegebenenfalls die Ausgabe von Fehlern sollte dem nicht so sein.

Am Ende wird dann das handle zurückgegeben da wird damit außerhalb noch arbeiten wollen.

Shader Program

Konkret müssen wir die beiden shader noch mittels `glAttachShader` an ein *shader program* binden welches wir dann mittels `glLinkProgram` linken. Das ganze sieht dann so aus:

```
unsigned int shaderProgram;
shaderProgram = glCreateProgram();
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
glLinkProgram(shaderProgram);
if (!checkProgram(shaderProgram)) {
    std::terminate();
}
glDeleteShader(fragmentShader);
glDeleteShader(vertexShader);
```

Die letzten Zeilen testen wieder auf Linkerfehler und zerstören am Ende die nicht mehr benötigten Shader Objekte. Das *shader program* ist damit fertig und muss nur noch vor dem Rendern mittels

```
glUseProgram(shaderProgram);
```

aktiviert werden (alle Operationen die Geometrie zeichnen verwenden dann diesem Programm, bis ein anderes gesetzt wird).

Vertex Buffers

Mit dem einbinden der shader haben wir die Hälfte schon geschafft, was wir jetzt noch machen müssen ist die Geometrie zu definieren und auf die GPU hochzuladen. Dazu erstellen wir ein sogenanntes *vertex buffer object* (kurz VBO) in welches wir die vorher definierten vertex Daten stecken:

```
float vertices[] = {
    //  X,     Y,    Z,
    -0.5f, -0.5f, 0.0f,
     0.5f, -0.5f, 0.0f,
     0.0f,  0.5f, 0.0f
};

unsigned int VBO;
glGenBuffers(1, &VBO);
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
```

In vertices schreiben wir die X-, Y-, und Z-Koordinaten unserer Punkte (3, da Dreieck), dann erstellen wir mit `glGenBuffers` einen vertex buffer, und definieren ihn mittels `glBindBuffer` als aktuell bearbeiteten Buffer (die klassische Methode für Objekte also). In diesem Fall müssen wir auch direkt noch

den Typ des Buffers angeben - in diesem Fall `GL_ARRAY_BUFFER` für VBOs. Im nächsten Kapitel lernen wir auch noch eine andere Art von Buffer, den `GL_ELEMENT_ARRAY_BUFFER` *index buffer* kennen.

Die folgenden Funktionen arbeiten dann auf diesem Buffer. Die erste (`glBufferData`) lädt tatsächlich unsere Daten (das *vertices* Array) in den vertex buffer – und damit auf die GPU – und benötigt daher die Daten selber und ihre Größe. Der letzte Parameter gibt noch einen Hinweis wie die Daten verwendet werden, damit die GPU des Speicherort besser wählen kann. In diesem Fall sagen wir mit `GL_STATIC_DRAW`, dass wir vorhaben die Daten nicht mehr zu ändern ("static"), sondern nur damit zu rendern ("draw"); daher wird der Treiber versuchen den Speicherort nur für das Lesen der Daten zu optimieren.

Die nächsten beiden Funktionen sind dafür da die hochgeladen Daten dann noch mit der (`position`) Variablen im shader zu verknüpfen. Schauen wir in die Dokumentation von `glVertexAttribPointer`, sehen wir folgende Definition:

```
void
glVertexAttribPointer(GLuint index,
                      GLint size,
                      GLenum type,
                      GLboolean normalized,
                      GLsizei stride,
                      const GLvoid * pointer);
```

`index` ist hier gerade der Index den wir im shader mittels

```
layout (location = 0) in vec3 position;
```

angegeben haben (also in diesem Fall 0). Danach kommen die Dimension des Attributs und der Skalartyp (hier ein 3-dimensionaler Vektor mit floating point Werten). Der nächste Parameter gibt an ob fixed-point Werte normalisiert werden sollen – was wir hier nicht wollen, da wir nicht mit normalisierten fixed-point Werten arbeiten. Danach kommt der sogenannte `stride`, also wie viele Bytes muss man überspringen um vom ersten zum zweiten Attribut zu gelangen. In diesem Falle ist das die Größe von 3 floating-point Werten. Angenommen unsere Daten würden noch Farben mit enthalten und so aussehen:

```
float vertices[] = {
    //  X,     Y,    Z,   R,   G,   B,   A,
    -0.5f, -0.5f, 0.0f, 1.f, 0.f, 0.f, 1.f,
     0.5f, -0.5f, 0.0f, 1.f, 0.f, 0.f, 1.f,
     0.0f,  0.5f, 0.0f, 1.f, 0.f, 0.f, 1.f
};
```

Dann müssten wir als stride `7 * sizeof(float)` angeben, da wir ja 7 floats überspringen müssten bis die nächsten Positionsdaten kommen. Die (nach void-Pointer gecastete) 0 am Ende gibt die Quelle der Daten an. Dabei heißt 0, dass die Daten im aktuell gebundenen *vertex buffer*, also hier "VBO" liegen.

Zuletzt aktivieren wir mit

```
glEnableVertexAttribArray(0);
```

noch das Attribut mit Index 0 selber, damit der shader das auch nutzt.

Vertex Array Objects

An sich könnten wir das Setzen und Aktivieren des Attributs jetzt jedes mal wenn wir diese Daten rendern wollen wiederholen und so die Daten rendern. Da das aber sehr repetitiv wäre gibt es eine Möglichkeit das nur einmal zu machen, sich mit Hilfe eines speziellen Objektes zu merken und mit einem Einzeiler zu wiederholen. Dazu erstellen wir ein sogenanntes *vertex array object* (VAO) und *binden* es als aktuelles VAO

```
unsigned int VAO;
glGenVertexArrays(1, &VAO);
glBindVertexArray(VAO);

// glBindBufffer, glVertexAttribPointer glEnableVertexAttribArray
```

Das müssen wir nur einmal machen und können dann später vor dem rendern per `glBindVertexArray` alles nötige zum rendern auf einmal erledigen.

Draw Calls

Was uns direkt zur letzten noch zu erwähnenden Codestelle bringt – das rendern selber:

```
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
glDrawArrays(GL_TRIANGLES, 0, 3);
```

Die ersten 2 Zeilen haben wir schon, erklärt, bleibt die Zeile in der tatsächlich die rendering pipeline gestartet wird. `glDrawArrays` bekommt als erstes Argument die Art von Primitiven die gerendert werden sollen (eine komplette Übersicht ist hier). Das zweite Argument gibt den Index des ersten zu zeichnenden vertex an und das dritte wieviele vertices insgesamt zu zeichnen sind.

Indizierte Geometrie

Das war jetzt erstaunlich viel Hintergrund und Code für einen erstaunlich unterwältigenden Effekt:

Abbildung 11: Ein Dreieck.

Die Flexibilität dieser Vorgehensweise bedeutet aber auch, dass wenn wir den shader code ein wenig ändern und die vertices (statt selber einzutippen) aus einer Datei laden, wir mit wenig Aufwand wesentlich spannendere Ergebnisse erzielen können.

Eine Sache die aber noch etwas unflexibel ist, ist der Aufruf von `glDrawArrays`. Der erwartet nämlich unsere vertices in der richtigen Reihenfolge dicht hintereinander gepackt. Was aber wenn die Reihenfolge nicht stimmt, wir nur das erste und das fünfte Dreieck rendern wollen oder wir beim rendern mehrerer Dreiecke einen vertex für mehrere Dreiecke wiederverwenden wollen?

OpenGL sieht dafür mit *index buffers* eine relativ simple aber effektive Methode vor. Die Idee ist, einfach in einen zweiten buffer die Indizes der vertices zu schreiben die gerendert werden sollen. Wie in *src/uebung_1c.cpp* zu sehen bauen wir uns dafür völlig analog zu den vertices ein Array von `unsigned int`:

```
unsigned int indices[] = {
    0, 1, 2
};
```

Wiederum analog zu den *vertex buffers* können wir das dann auf die GPU laden. Das Ergebnis nennen wir dann lediglich *index buffer* oder *element buffer* und haben den Vorteil, dass wir das ganze nicht an unseren shader koppeln müssen (wir sparen uns also `glVertexAttribPointer`).

Wie ein Blick in den Beispielcode zeigt wurde der Code zum Erstellen bereits in eine Funktion ausgelagert:

```
unsigned int
makeBuffer(unsigned int bufferType, GLenum usageHint, unsigned int
    bufferSize, void* data) {
    unsigned int handle;
    glGenBuffers(1, &handle);
    if (data) {
        glBindBuffer(bufferType, handle);
        glBufferData(bufferType, bufferSize, data, usageHint);
    }

    return handle;
}
```

Somit müssen wir unserer `main` Funktion nur noch diese Funktion aufrufen:

```
unsigned int VBO = makeBuffer(GL_ARRAY_BUFFER, GL_STATIC_DRAW, sizeof(
    vertices), vertices);
```

Analog können wir nun unseren index buffer erstellen – nur dass wir hier als Type `GL_ELEMENT_ARRAY_BUFFER` angeben und eben die indices statt vertices übergeben müssen:

```
unsigned int IBO = makeBuffer(GL_ELEMENT_ARRAY_BUFFER, GL_STATIC_DRAW,
    sizeof(indices), indices);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, IBO);
```

Analog zum VBO muss auch der IBO noch per *bind* an das *vertex array object* gebunden werden, daher das `glBindBuffer` (wieder mit `GL_ELEMENT_ARRAY_BUFFER`).

Beim Zeichnen verwenden wir dann statt `glDrawArrays` die Funktion `glDrawElements`,

```
glDrawElements(GL_TRIANGLES, 3, GL_UNSIGNED_INT, (void*) 0);
```

die wieder den Typ der Primitive bekommt, dann die Anzahl der *indices* und den Typ der indices. Mit dem Pointer am Ende gibt man ziemlich kompliziert an bei welchem Index man anfängt – wir wollen beim ersten anfangen, daher die 0.

Und das war es schon – das Ergebnis beim Ausführen sollte exakt das gleiche sein, nur dass wir hier wieder einen Freiheitsgrad gewinnen der uns bereits in den Übungsaufgaben hilft vertices zu sparen (wenn auch für's erste nur 2).