Tutorial: Geometrie-Daten auf die GPU kopieren

Einleitung

Damit wir mit der Grafikkarte 3D-Objekte rendern können, müssen alle Daten, die wir dazu brauchen, im Grafikspeicher (Video-RAM) vorliegen. Mittels C++ können wir allerdings gar nicht direkt auf den Grafikspeicher zugreifen, geschweige denn Variablen vom Typ `std::vector<Triangle>` dort anlegen. Damit wir Geometrie-Daten wie unsere `std::vector<Triangle>` also in den Grafikspeicher kopieren können, müssen wir die OpenGL-API verwenden und uns nach den dort verfügbaren Funktionen richten.

Vorweg ein paar Erklärungen von Begriffen, um Missverständnisse möglichst zu verhindern.

Buffer

Ein Buffer ist ein recht allgemeiner Begriff, der einfach nur einen beliebigen Block von Daten bezeichnet. Beispielsweise kann man die Variable vom Typ `std::vector<Triangle>` auch als Buffer auffassen, welcher unsere Geometrie-Daten im Arbeitsspeicher (RAM) speichert. Aber auch bereits eine einfache `int`-Variable kann als Buffer aufgefasst werden, selbst wenn ein solcher Buffer nur einen Wert speichern kann.

Vertex Buffer Object (VBO)

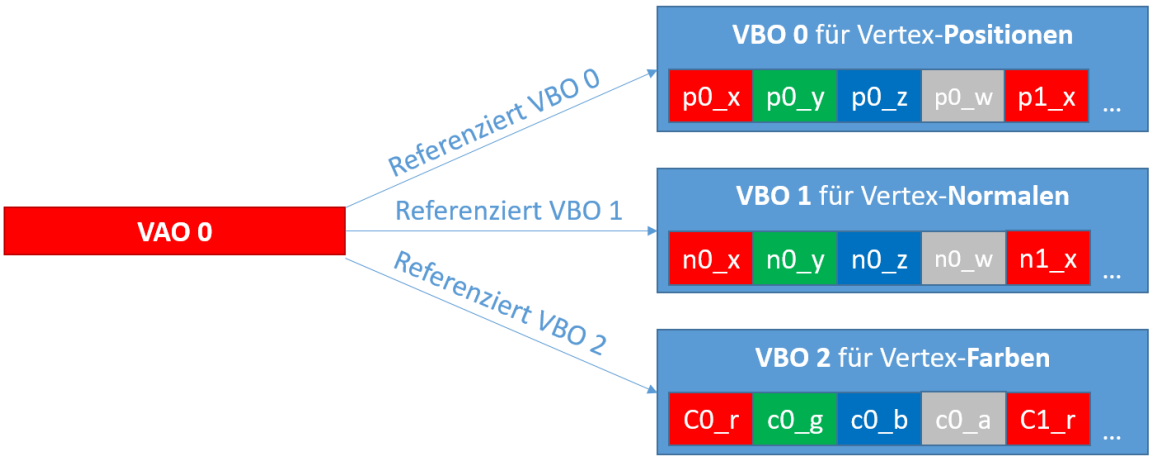
In OpenGL gibt es sogenannte *Buffer Objects*, die Daten auf dem Grafikspeicher (Video-RAM) abspeichern. Nutzt man diese, um Vertex-Daten wie Position, Normalen oder Farben zu speichern, werden diese auch *Vertex Buffer Objects* genannt.

Hinweis: Ein Vertex Buffer Object (VBO) weiß selbst nicht, was für Daten in ihm gespeichert sind – auch nicht, ob dort einzelne float, integers o.ä. drin gespeichert sind. Ein VBO kennt neben den binären Daten aber seine Größe in Bytes.

Vertex Array Object (VAO)

Ein *Vertex Array Object* speichert Referenzen auf einen oder mehreren *Vertex Buffer Objects* und speichert zudem noch, wie die Daten, die dort liegen, zu verstehen sind.

Ein ganz grobes Konzeptbild ist nachfolgend abgebildet, welches zeigt, was in Aufgabe 1 des dritten Übungsblattes erstellt werden soll.



VBO erstellen und Daten hochladen

VBO erstellen

Um ein Buffer Object zu 'deklarieren', verwendet man die Funktion
`void glGenBuffers(unsigned int n, unsigned int* name)`, dabei:

- gibt der erste Parameter `n` an, wie viele Buffer wir erstellen wollen.
- gibt der zweite Parameter eine Speicheradresse an, wo eine Nummer hineingeschrieben wird, die den Buffer eindeutig identifiziert.

Wenn man folgenden Befehl ausführt:

```
unsigned int myBufferReference;
glGenBuffers(1, &myBufferReference);
```

so wird ein Buffer erstellt, und ein `int`-Wert in die Variable `myBufferReference` gespeichert. Bei dem ersten Buffer, den wir bei der Ausführung des Programms anlegen, wird oft eine `1` in die Variable abgespeichert. Rufen wir die `glGenBuffers`-Funktion erneut auf, so wird ein weiterer Buffer erstellt und eine `2` in die Variable des übergebenen Pointers geschrieben, usw.

Hinweis: In der OpenGL-Doku wird auch oft von einem "Namen" eines Buffers oder einer Textur gesprochen. Damit ist i.d.R. diese Zahl gespeichert, die den Buffer referenziert.

VBO binden

Die OpenGL-API ist an einigen Stellen ein wenig umständlich/schwerfällig. Denn bevor man die Daten eines Buffers ändern kann, muss man diesen "binden" (= als aktiven Buffer setzen). Dies geht über die Funktion:
`void glBindBuffer(GLenum target, unsigned int buffer_name)`

Eine Erklärung der Parameter der Reihenfolge nach:

- OpenGL hat mehrere Typen von Buffer. Von jedem Buffer-Typ kann nur einer gleichzeitig aktiv sein. Um Vertex-Daten hochzuladen, verwenden wir immer den Buffer-Typ `GL_ARRAY_BUFFER` - d.h. bei uns steht immer dies im Aufruf.
- Dies ist die Referenz auf den Buffer und kann z.B. unsere Variable `myBufferReference` sein.

Den vorher erstellten Buffer könenen wir also mittels folgendem Funktionsaufruf als aktiven Buffer für Vertex-Daten setzen:

```
glBindBuffer(GL_ARRAY_BUFFER, myBufferReference);
```

Daten in VBO hochladen

Um schließlich die Daten von dem Arbeitsspeicher (RAM) in den VBO hochzuladen, der sich im Grafikspeicher (Video-RAM) befindet, verwendet man die Funktion:
`glBufferData(GLenum target, GLsizeiptr size, void* data, GLenum usage)`

Eine Erklärung der Parameter der Reihenfolge nach:

- Hier verwenden wir `GL_ARRAY_BUFFER`, was bedeutet, dass der aktuell aktive Buffer für Vertex-Daten beschrieben wird, den wir im vorherigen Abschnitt gebunden haben.

- `size` gibt an, wie viele Bytes hochgeladen werden sollen. Hinweis: Ein `float`-Wert ist in C++ i.d.R. vier Byte groß (auf Nummer sicher geht man, wenn man `sizeof(float)` verwendet). Ein Vec4f enthält vier `float`-Werte, usw.
- `data` gibt ein Pointer auf den ersten Byte der Daten im Arbeitsspeicher an, die hochgeladen werden sollen.
- `usage` gibt an, wie der Buffer verwendet wird. Wir verwenden hier in dieser Aufgabenstellung `GL_STATIC_DRAW`, was bedeutet, dass der Buffer dafür optimiert ist, nur ein einziges Mal beschrieben zu werden, aber oft von der Grafikkarte ausgelesen zu werden – das passiert ja in jedem Frame, wo das Objekt ge

Um die Vertex-Positionen eines Dreiecks zu erstellen und in den zuvor erzeugten und als aktiv gesetzten Buffer hochzuladen, können wir schreiben:

```
// Erstelle Positionsdaten fuer ein einzelnes Dreieck:
std::vector<Vec4f> positions;
positions.push_back(Vec4f(0.0f, 0.0f, 0.0f));
positions.push_back(Vec4f(0.5f, 1.0f, 0.0f));
positions.push_back(Vec4f(1.0f, 0.0f, 0.0f));

// Berechne Groesse der Daten in Bytes:
unsigned int byteSize = positions.size() * 4 * sizeof(float);

// Sage OpenGL, dass es die Daten aus der positions-Liste in den VBO kopieren soll:
glBufferData(GL_ARRAY_BUFFER, byteSize, &positions[0], GL_STATIC_DRAW);
```

VAO erstellen und mit VBO verlinken

VAO erstellen

Bezüglich des Erstellen eines VAOs verhält es sich genauso wie mit einem VBO. Dieses Vertex-Array können wir einfach über glBindVertexArray als aktives Array setzen, was durch nachfolgende OpenGL-Funktionsaufrufe verändert wird.

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

VAO und VBO verlinken

Um das aktive *Vertex Array Object (VAO)* mit dem aktiven *Vertex Buffer Object (VBO)* zu verlinken, verwendet man die Funktion:
`glVertexAttribPointer(index, size, type, normalized, stride, offset)`

- `index` gibt einen Index für das Vertex-Attribut an. *Achtung: Für eine Position verwenden wir in diesem Übungsblatt 3 den Index 0, für eine Normale den Index 1 und für eine Farbe den Index 2 – dies ist nämlich im vorgegebenen Shader so festgelegt.*
- `size` gibt an, wie viele Komponenten eine Variable besitzt (bzw. die Dimension des Vektors) - da wir einen Vec4f hochladen, geben wir hier immer **4** an.
- `type` gibt an, welchen Typ die einzelnen Komponenten besitzen. Da einzelne Komponenten eines `Vec4f` den Typ `float` haben, geben wir hier immer `GL_FLOAT` an.
- `normalized` gibt an, ob die Daten normalisiert werden sollen - hier verweden wir vorerst einfach `GL_FALSE`.
- `stride` gibt einen Byte-Offset zwischen einzelnen Variablen an. Wir verwenden i.d.R. kein Interleaved Vertex Layout, daher folgenden die Vektoren direkt aufeinander und wir können/müssen 0 angeben.

- `offset` gibt einen Byte-Offset vom Anfang der Daten an - da unsere VBOs immer direkt mit den ersten Vertex starten, geben wir hier auch 0 an.

Um die Verlinkung zwischen dem zuvor erstellten aktiven VAO und dem aktiven VBO herzustellen können wir also angeben:

```
glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, 0);
```

Zudem müssen wir noch sicherstellen, dass das jeweilige Vertex-Attribut aktiviert ist. Das machen wir mit dem nachfolgenden Befehl, wobei der übergebene Parameter der Index des Vertex-Attribut ist *(in diesem Übungsblatt verwenden wir für Positionen auch hier die 0, für Normalen eine 1 und für Farben eine 2)*:

```
glEnableVertexAttribArray(0);
```

Nun haben wir die Positionsdaten erfolgreich auf die GPU hochgeladen.

Zusammenfassung des Beispiels

```
// VAO erstellen und als aktiv setzen:
unsigned int myVAOReference;
glGenVertexArrays(1, &myVAOReference);
glBindVertexArray(myVAOReference);

// VBO fuer Positionen erstellen und als aktiv setzen:
unsigned int myBufferReference;
glGenBuffers(1, &myBufferReference);
glBindBuffer(GL_ARRAY_BUFFER, myBufferReference);

// Erstelle Positionsdaten fuer ein einzelnes Dreieck:
std::vector<Vec4f> positions;
positions.push_back(Vec4f(0.0, 0.0, 0.0));
positions.push_back(Vec4f(0.5, 1.0, 0.0));
positions.push_back(Vec4f(1.0, 0.0, 0.0));

// Berechne Groesse der Daten in Bytes:
unsigned int byteSize = positions.size() * sizeof(Vec4f);

// Sage OpenGL, dass es die Daten aus der positions-Liste in den VBO kopieren soll:
glBufferData(GL_ARRAY_BUFFER, byteSize, &positions[0], GL_STATIC_DRAW);

// Verlinke das aktive VAO mit dem VBO und sage, wie die Daten im VBO zu verstehen sind:
glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, 0);

// Aktiviere schliesslich das Vertex-Attribut mit dem gesetzten Index:
glEnableVertexAttribArray(0);

// Sicherheitshalber setzen wir das aktive VBO und VAO zurueck,
// um spaetere OpenGL-Funktionsaufrufe nicht zu beeinflussen:
glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindVertexArray(0);
```