Schritt 4: Shader und Buffer Objects verknüpfen bzw. Rendern

Wir gehen das Grundprinzip erst mal ohne Vertex Array Objects durch. Wenn wir schließlich und endlich unsere Geometrie darstellen wollen, tun wir dies mit folgendem Befehl:

```
glDrawElements( PrimitiveType, IndexCount, IndexDatatype, nullptr );
```

PrimitiveType hängt vom Layout der Indices ab. *IndexCount* ist die Anzahl der Indices, die bearbeitet werden sollen. *IndexDatatype* ist der Datentyp der Indices – praktisch immer GL_UNSIGNED_INT. Das letzte Argument, der nullptr, sagt OpenGL wo die Indexdaten herkommen: Nämlich aus dem im Moment an das GL_ELEMENT_ARRAY_BUFFER Target gebundenen Buffer Object. (Es wäre theoretisch möglich, hier direkt Indices zu übergeben, aber das werden wir nie tun, daher wird's immer der nullptr sein...)

Der Befehl bewirkt im Wesentlichen folgendes: *IndexCount* Indices werden aus dem aktuell ans GL_ELEMENT_ARRAY_BUFFER Target gebundenen Buffer Object geladen. *IndexDatatype* hilft dabei, die Bytes im Buffer Object korrekt als Ganzzahlen zu interpretieren. Jeder Index wird benutzt, um die entsprechenden Vertex Attribute zu *fetchen*, und zwar aus einem zuvor spezifizierten Buffer Object. Für jeden *gefetchten* Vertex wird eine Instanz des Vertex Shaders mit den entsprechenden Vertex Attributen als Input gestartet. In der Primitive Assembly der GPU Pipeline werden die bearbeiteten Vertices zu *PrimitiveType* zusammengesetzt.

Die GPU hat schon einen Großteil der Infos, die sie braucht, um rendern zu können. Es fehlen aber noch die Vertex Attribute. Die Index Daten sind leicht zu interpretieren, denn jeder Index ist genau eine Ganzzahl – wir müssen nur angeben, wie vieles Byte diese Ganzzahl hat, was wir im `glDrawElements` Befehl getan haben. Wo die Index Daten zu finden sind, ist auch klar: im Buffer Object, welches an GL_ELEMENT_ARRAY_BUFFER gebunden ist. Die Sache mit den Vertex Attributen ist dagegen viel komplizierter, weil es ja mehr als ein Buffer Object geben kann und die Vertex Daten *interleaved* in den Buffer Objects gespeichert sein könnten. Folgende Infos benötigt die GPU daher noch zusätzlich:

- Woher – also aus welchem Buffer Object – kommen die Vertex Attribute?
- Was ist Grunddatentyp der Attribute – Float oder Int? (Etwas anderes geht nicht.)
- Wie viele Komponenten hat das Attribut – 1, 2, 3 oder 4? (Mehr geht nicht.)
- Wie viele Bytes liegen zwischen zwei aufeinanderfolgenden Attributen im Buffer Object? (→ wichtig, weil sie ja mit anderen Attributen *interleaved* sein könnten)
- Wie groß ist der Offset – wieder mal in Bytes – zum ersten Attribut im Buffer Object?
- Wo soll das Attribut hin geladen werden? (In welches *Attributs Register* der Shader Einheit?)

Auf unser konkretes Beispiel bezogen

Hier müssen wir uns daran erinnern, wie die Vertex Attribute ganz genau im Buffer Object abgelegt wurden. Gleichzeitig ist wichtig, welche Inputs (→ deklariert mit dem Schlüsselwort `in`) der Vertex Shader in welchem Register (→ `location` qualifier) erwartet.

Inputs des Vertex Shaders werden nämlich aus speziellen Registern ausgelesen, den sogenannten *Attributs Registern*. Jeder Shader Core hat eine Anzahl solcher Register - die genaue Zahl ist GPU spezifisch, aber wir werden in der LVA nie in die Situation kommen, dass wir die Zahl ausschöpfen - und identifiziert werden diese über einen Index, bei 0 beginnend. Zum Beispiel hab ich einfach so mal beschlossen, dass der Vertex Shader in diesem Tutorial Farben aus dem Register #8 auslesen soll und Positionen aus dem Register #0. Bei Ihren eigenen Anwendungen können Sie das im Grunde selber entscheiden, wobei *Position auf #0* und *Normale auf #1* eine gute Grundkonvention ist. Es gibt übrigens mehrere Methoden, dieses *Attribute Binding* festzulegen; Wir verwenden in dieser LVA die stärkste Variante: explizites Binding im Vertex Shader. In den CG1 Samples wurde das Binding teils zur Laufzeit spezifiziert, nämlich über `glBindAttribLocation` – ein vorhandenes Vertex Shader Binding hätte auch dann immer noch Vorrang. Die dritte Variante wäre, es der GPU zu überlassen, und die Bindings zur Laufzeit abzufragen: mit `glGetAttribLocation`.

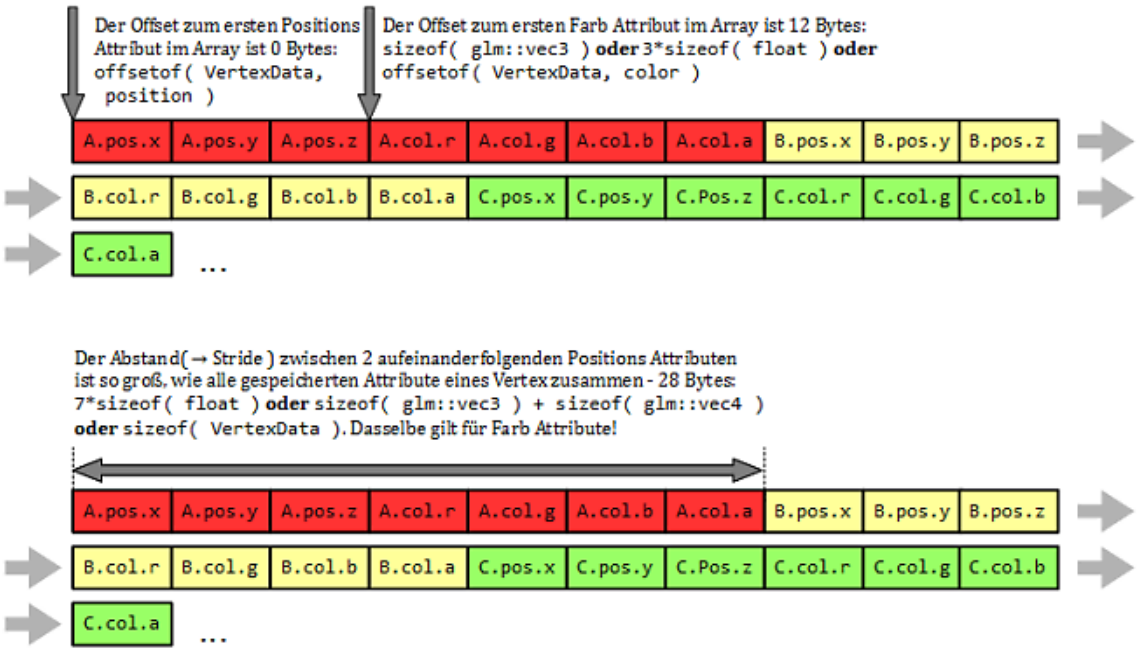
Um auf unsere Attribute zurückzukommen: Die Vertex Position hat 3 Komponenten XYZ und ist ein GL_FLOAT Attribut. Die Anordnung der Attribute im Buffer Object ist: zuerst Position, dann Farbe. Offset für die Position ist somit 0 Bytes, das erste Attribut ist ja gleich eine Position. Zwischen zwei aufeinanderfolgenden Positionen liegen 28 Bytes. (→ Man beachte auch die Grafik auf der nächsten Seite) Die Position soll ins *Attributs Register mit der Id 0* geschickt werden, dies legt der Vertex Shader fest:

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

Die Vertex Farbe hat 4 Komponenten RGBA und ist ein GL_FLOAT Attribut. Die Farbe kommt im Daten Array als zweites, nach der Position, der Offset entspricht somit der Größe eines Positions Attributs in Bytes, also 12 Bytes. Der Abstand zwischen zwei aufeinanderfolgenden Farben im Daten Array entspricht wiederum 28 Bytes. Die Farbe soll ins *Attributs Register mit der Id 8* gespeichert werden:

```
layout ( location = 8 ) in vec4 vertexColor;
```

Dass der Vertex Shader als Input einen `vec3` und einen `vec4` erwartet, ist übrigens zwar schön – weil es damit übereinstimmt, wie wir unser Attribute spezifiziert haben – hat aber eigentlich null Auswirkung darauf, was die GPU in welches Register schreibt. Es ist vielmehr so: was auch immer in dem Register steht, wird als `vec3` bzw. `vec4` interpretiert werden, wenn der Vertex Shader letztlich ausgeführt wird; Es ist also unsere Aufgabe, dafür zu sorgen, dass die richtigen Daten - mit dem richtigen Datentyp und der richtigen Anzahl an Komponenten - im richtigen Register ankommen.



Der Code fürs verknüpfen von Buffer Object und Vertex Shader – und fürs Rendering – sieht dann so aus:

```
glBindBuffer( GL_ARRAY_BUFFER, handleToAttribBuffer );

glVertexAttribPointer( 0, 3, GL_FLOAT, false, sizeof( VertexData ), reinterpret_cast<
GLvoid const * >( offsetof( VertexData, position ) ) );
glVertexAttribPointer( 8, 4, GL_FLOAT, false, sizeof( VertexData ), reinterpret_cast<
GLvoid const * >( offsetof( VertexData, color ) ) );

glEnableVertexAttribArray( 0 );
glEnableVertexAttribArray( 8 );

glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, handleToIndexBuffer );

glDrawElements( GL_TRIANGLES, indexData.size(), GL_UNSIGNED_INT, nullptr );

glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, 0 );

glBindBuffer( GL_ARRAY_BUFFER, 0 );

glDisableVertexAttribArray( 0 );
glDisableVertexAttribArray( 8 );
```

Die `glVertexAttribPointer` Aufrufe verändern den OpenGL Render State und beziehen sich immer auf den aktuell ans GL_ARRAY_BUFFER Target gebundene Buffer Object. Bei einem *non-interleaved* Setup wäre als der einzige Unterschied zum Code oben, dass Sie zwischen den beiden Aufrufen von `glVertexAttribPointer` das Buffer Object austauschen, also das Zweite binden.

Der Befehl `glVertexAttribPointer` kommuniziert der GPU im Wesentlichen die zuvor aufgelisteten Punkte – also wie viele Komponenten das Attribut hat, in welches *Attributs Register* es geladen werden soll, wie groß der Offset ist etc., und setzt voraus, dass noch ein `glEnableVertexAttribArray` für das Register aufgerufen wird. Solange kein `glDisableVertexAttribArray` kommt, wird die GPU danach jedes Mal beim Rendern

Daten in das Register laden. (Unabhängig davon, was für ein Vertex Shader gerade in Verwendung ist, also auch wenn der Vertex Shader das Attribut gar nicht benötigt.)

Es gibt einen Sonderbefehl, der einen einzelnen Wert in ein *Attributs Register* speichert - und diesen Wert für alle Vertices unverändert lässt: `glVertexAttribX`, dies ist hilfreich wenn Sie ein Model mit einem Shader verwenden möchten, der zu viele Attribute fordert: So haben die meisten 3D Modelle keine Vertex Farbe gespeichert, man könnte aber einem Vertex Shader, der ein Farbe als Input erwartet, auf diese Weise einen konstanten Wert als Attribut vortäuschen.

Bezüglich des Samplecodes: Die Definition des `struct VertexData` für die Vertex Attribute, und die Verwendung eines `std::vector< VertexData >` machen Ihnen die Verwendung der OpenGL Funktionen etwas einfacher, hauptsächlich wegen `sizeof` und `offsetof` – Sie müssen dann nicht irgendwelche Bytes zählen. Aber: Wenn Sie möchten, können Sie für ihre Vertex Daten durchaus ein simples C-Style Array (für floats dann, oder, wenn Sie völlig wahnsinnig sind, für Bytes) verwenden, und da die einzelnen Werte XYZ der Position bzw. RGBA der Farbe in der richtigen Reihenfolge hinein schreiben. Am Ende macht dies für die GPU nämlich überhaupt keinen Unterschied…

Nur eins müssen Sie immer tun: eine Datenstruktur verwenden, der ein Array zugrunde liegt, also eine Datenstruktur, bei der die Speicherzellen direkt aufeinander folgen. In der C++ Sprache wäre dies idealerweise ein `std::vector` oder `std::array`, notfalls eben auch ein C-Style Array (ob dynamisch oder statisch allokiert, ist egal). Was nie, nie, nie gut geht sind `std::list`, `std::set` oder `std::map`. Gerade die `std::list` ist eventuell für Sie gefährlich, weil Sie aus CG1 ja gewöhnt sind, dass eine C# `List` verwendet wird.