Schritt 3: Die Geometrie auf der CPU, und der Datentransfer auf die GPU

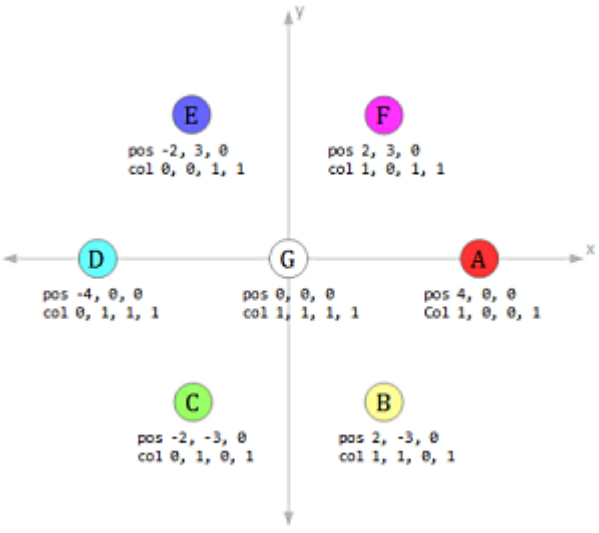
Um etwas rendern zu können, brauchen wir Geometrie: Vertices & Primitive. Diese müssen in einer Form gespeichert werden, die mit der GPU kompatibel ist. Dazu ist zu sagen, dass die Möglichkeiten auf der Grafikkarte extrem begrenzt sind, es gibt im Grunde nur zwei unterstützte Formen der Objektrepräsentation:

- als sogenannte Vertex Liste
- als Vertex Liste mit Indices

Für diverse CG Algorithmen werden komplexere Strukturen benötigt: Beispielsweise gibt es die *Winged Edge* und *Half Edge* Strukturen, um Dreiecks Meshes zu speichern. Diese werden im 3D Modeling Anwendungen gerne eingesetzt – für Berechnungen auf der CPU - lassen sich aber nicht wirklich auf das Datenmodell der Grafikkarte übertragen. Wenn wir so eine Datenstruktur vorliegend haben, müssen wir extra für den Transfer auf die GPU die Geometrie in eine passende Datenstruktur kopieren. Auch die Library, die wir später für das Laden von 3D Modellen verwenden werden, liefert die Daten nicht von sich aus in einer kompatiblen Struktur.

Aus Gründen der Performance ist in fast allen Anwendungsfällen die Verwendung einer Vertex Liste mit Indices zu bevorzugen: Die erlaubt der Grafikkarte Optimierungen, z.B. muss ein Vertex, welcher von mehreren Primitiven geteilt wird, evtl. nur einmal durch den Vertex Shader. (Stichwort: Post Transform Cache…) In der CG1 haben Sie eine andere Form der Optimierung kennengelernt, nämlich die Verwendung von GL_TRIANGLE_STRIP und GL_TRIANGLE_FAN anstelle individueller GL_TRIANGLES. Diese bewirken ähnliches, sind aber nur bedingt für den Einsatz in echten Anwendungen geeignet: Man kann gewisse 3D Modelle nicht einfach so in Strips umwandeln, individuelle Primitive geben dem 3D Artists mehr Freiheit. Eine Kombination von Indices und Strips ist aber durchaus möglich, wenn es für das Objekt passt.

Dies sei unsere Beispielgeometrie: 7 Vertices, Attribute sind Position und Farbwert.



Für diese Geometrie brauchen wir nun mindestens zwei Arrays:

- mindestens eines für die Vertex Attribute, je nach Speicherart: *interleaved* vs. *non–interleaved*.
- und zusätzlich eines für die Indices

Nehmen wir beispielshalber an, die Vertices lägen in dieser Reihenfolge im Datenarray:

V = { A, B, C, D, E, F, G }

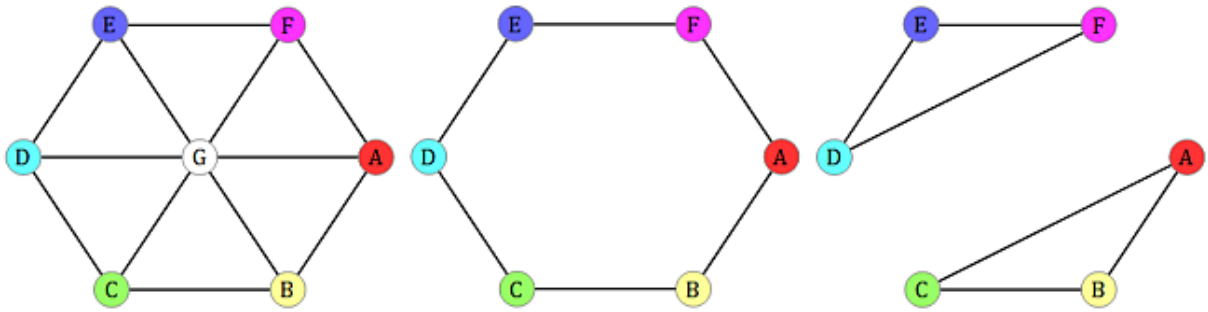
Dann könnte unser Index Array so aussehen:

Fall 1: I = { 0, 6, 1, 1, 6, 2, 2, 6, 3, 3, 6, 4, 4, 6, 5, 5, 6, 0 }

Fall 2: I = { 6, 0, 1, 2, 3, 4, 5 }

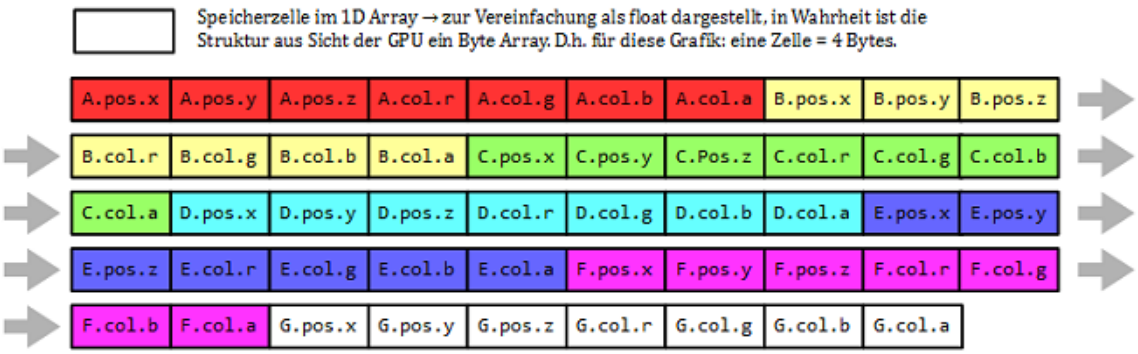
Fall 3: I = { 0, 1, 2, 3, 4, 5 }

Fall 1 und Fall 2 sind auf Dreiecksprimitive ausgelegt. Der einzige Unterschied ist, dass in Fall 1 die GL_TRIANGLES explizit ausspezifiziert werden, während Fall 2 dieselben Polygone über einen GL_TRIANGLE_FAN definiert (→ siehe die Abbildung unten links). Fall 3 könnte man verwenden, um einen GL_LINE_LOOP (→ unten Mitte) um die äußeren Vertices zu erzeugen – oder, in Kombination mit GL_TRIANGLES, um 2 nicht verbundene Dreiecke zu zeichnen (→ unten rechts).



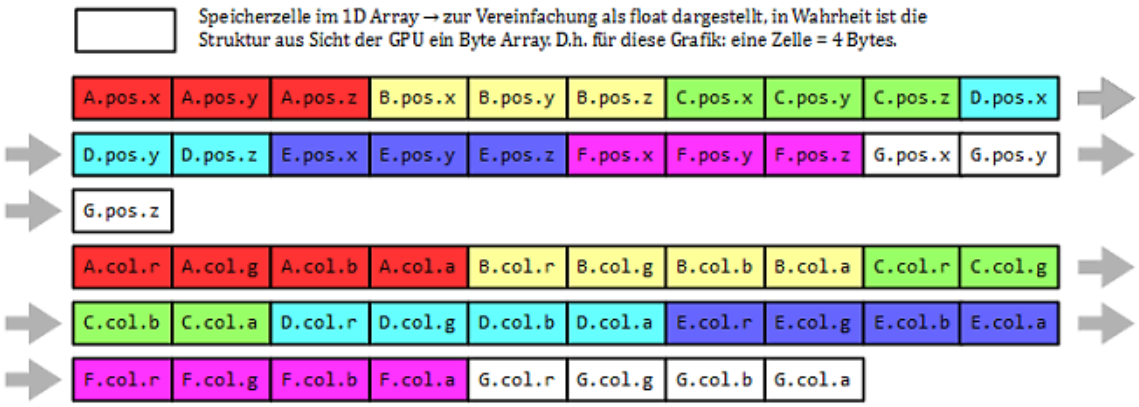
Sehen wir uns jetzt genauer an, wie die Vertex Attribute gespeichert werden. Wir gehen hier davon aus, dass unsere Geometrie Vertex Positionen und Vertex Farben definiert. Das Attribut Vertex Position werden wir in der LVA bei jeder Geometrie zu 100% haben – was irgendwie klar ist, denn was soll man mit einem Vertex ohne Position im Shader anfangen? Alle anderen Attribute richten sich nach Anwendungsfall: Normalen fürs Shading, Tangenten & Bitangenten fürs Normal Mapping, UV Koordinaten… Vertex Farben sind in der Realität übrigens selten, dafür verwendet man lieber über Texturen. Wir können uns nun aussuchen, ob wir die Attribute *interleaved* oder *non-interleaved* speichern möchten, wobei das Code Sample *interleaved* sein wird.

Interleaved bedeutet: alle Attribute eines Vertex werden gemeinsam gespeichert, man hat am Ende *ein großes 1D Array* mit Vertex Attributen:



Wobei die Reihenfolge der Attribute (also ob wir zuerst die Farbe oder die Position eines Vertex speichern) von uns frei festgelegt werden kann, es muss nur für alle Vertices im Array konsistent sein. Und es muss immer für jeden Vertex jedes Attribut gespeichert sein, Sie können sich keinen Speicherplatz sparen, wenn z.B. mehrere Vertices dieselbe Farbe haben, oder andersherum, wenn ein Vertex mehrfach mit unterschiedlichen Farben auftritt. (Bsp.: Ein Würfel mit unterschiedlich gefärbten Seiten, jeder der acht Vertices müsste mehrfach ins Array)

Im Gegensatz dazu hätten wir bei *non-interleaved* Speicherung *zwei kleinere 1D Arrays* mit folgendem Layout:



Auch hier muss übrigens wieder sichergestellt sein, dass für jeden Vertex jedes Attribut gespeichert ist, es kann also nicht ein Array weniger Attribute speichern als das andere – auch dann nicht, wenn alle Vertices zufällig dieselbe Farbe haben.

Die Kombination von *interleaved* und *non-interleaved* Speicherung ist möglich, und wäre ab 3 Attributen denkbar: Zwei Attribute *interleaved* in einem Array, das dritte Attribut in einem eigenen Array.

Datentransfer

Auf der GPU speichern wir Geometrie in (Vertex) Buffer Objects – auch die Indices! Dabei wird für jedes Array auf der CPU ein Buffer Object auf der GPU angelegt. Die Schritte sind dabei immer dieselben:

- Wir erzeugen ein Buffer Object, auf das wir aber nicht direkt zugreifen können, sondern nur über ein Handle. Bei OpenGL sind die Handles immer vom Typ `GLuint`. (→ entspricht einer 32 Bit Ganzzahl ohne Vorzeichen)
- Wir binden das Buffer Object ans entsprechenden Buffer Target: GL_ARRAY_BUFFER für Vertex Daten, GL_ELEMENT_ARRAY_BUFFER für Index Daten.
- Wir starten den Datentransfer, wobei wir gleichzeitig die Größe des Buffer Objects festlegen. Dies ist eine der wenigen blockierenden Operationen in OpenGL. (→ Logisch, sonst könnten wir nämlich die Daten überschreiben, während der Transfer läuft)
- Wir unbinden das Buffer Object wieder.

Der gesamte Code fürs Definieren der Geometrie bis hin zum Transfer spielt sich im INIT ab und sieht wie folgt aus:

```
std::vector< unsigned > indexData = { 0, 6, 1, 1, 6, 2, 2, 6, 3, 3, 6, 4, 4, 6, 5, 5,
6, 0 };

struct VertexData
{
	glm::vec3 position;
	glm::vec4 color;
};

std::vector< VertexData > attribData =
		{{ glm::vec3(  4.f,  0.f, 0.f ), glm::vec4( 1.f, 0.f, 0.f, 1.f ) },
		 { glm::vec3(  2.f, -3.f, 0.f ), glm::vec4( 1.f, 1.f, 0.f, 1.f ) },
		 { glm::vec3( -2.f, -3.f, 0.f ), glm::vec4( 0.f, 1.f, 0.f, 1.f ) },
		 { glm::vec3( -4.f,  0.f, 0.f ), glm::vec4( 0.f, 1.f, 1.f, 1.f ) },
		 { glm::vec3( -2.f,  3.f, 0.f ), glm::vec4( 0.f, 0.f, 1.f, 1.f ) },
		 { glm::vec3(  2.f,  3.f, 0.f ), glm::vec4( 1.f, 0.f, 0.f, 1.f ) },
		 { glm::vec3(  0.f,  0.f, 0.f ), glm::vec4( 1.f, 1.f, 1.f, 1.f ) }};

GLuint handleToIndexBuffer = 0;
glGenBuffers( 1, std::addressof( handleToIndexBuffer ) );
glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, handleToIndexBuffer );
glBufferData( GL_ELEMENT_ARRAY_BUFFER, sizeof( unsigned ) * indexData.size(),
reinterpret_cast< GLvoid const* >( indexData.data() ), GL_STATIC_DRAW );
glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, 0 );

GLuint handleToAttribBuffer = 0;
glGenBuffers( 1, std::addressof( handleToAttribBuffer ) );
glBindBuffer( GL_ARRAY_BUFFER, handleToAttribBuffer );
glBufferData( GL_ARRAY_BUFFER, sizeof( VertexData ) * attribData.size(),
reinterpret_cast< GLvoid const* >( attribData.data() ), GL_STATIC_DRAW );
glBindBuffer( GL_ARRAY_BUFFER, 0 );
```

Im UN-INIT kommt das Aufräumen der Buffer Objects hinzu. (`glGenX` → `glDeleteX`)

```
glDeleteBuffers( 1, std::addressof( handleToIndexBuffer ) );
glDeleteBuffers( 1, std::addressof( handleToAttribBuffer ) );
```

Das Ganze hat nur ein Problem: Sobald die Daten auf der GPU sind, sind es nur mehr Bytes. Dies erkennen Sie auch daran, dass bei `glBufferData` auf den Typ `GLvoid` gecastet wird, also das ganz explizit der ursprüngliche Datentyp verworfen wird. Alles, was die Grafikkarte weiß, ist, dass da ein Speicherbereich mit einer bestimmten Größe ist. Ob die Werte darin jetzt Indices, oder Vertex Farben sind – ob es sich um floats oder ints handelt – ist völlig unbekannt.

Zum Buffer Binding:

OpenGL ist in der Hinsicht etwas eigen. Wir können leider nicht sagen: ‚*Ich möchte jetzt in das Buffer Object mit diesem Handle Daten übertragen*'. Stattdessen sagen wir: ‚*Ich möchte jetzt in das Buffer Object, das genau in diesem Moment an das Buffer Target mit der Bezeichnung GL_ARRAY_BUFFER (bzw. GL_ELEMENT_ARRAY_BUFFER) gebunden ist, Daten übertragen*.' Das entsprechende Buffer Object muss also korrekt gebunden werden. Dieses Binding Konzept wird uns noch öfter unterkommen, zum Beispiel beim Rendern, und auch bei der Verwendung von Texturen.