Actor (Engine: Rendering)

Abstrahiert die Batches. Hier kommen zum Beispiel Actors wie *TileMap, Image,* oder so.

Batch (Engine: Rendering)

Abstrahiert die OpenGL Renderschicht (VAO, VBO). Es werden Vertices und Material übergeben und daraus werden *DrawCalls* erzeugt.

StreamingBatch

- Beliebig viele, verschiedene Vertices
- Ein Material für alle Vertices

InstancingBatch

- Vertices die exakt ein Objekt darstellen
- Ein Material
- *Transform* 's die eine Instanz des Objekts repräsentieren

Buffer (Java, OpenGL)

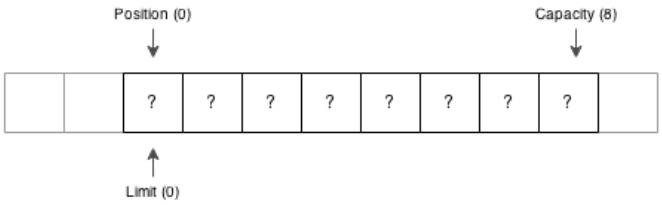
Speicheradresse die nicht zum Buffer gehört

Speicheradresse die zum Buffer gehört

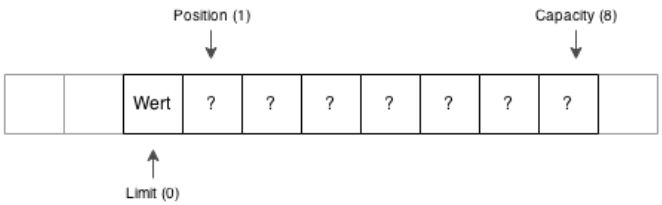
Pointer

Position	Auf welche Stelle im Buffer die nächste Aktion ausgeführt wird.
Limit	Bis wohin der Buffer Daten enthält, damit verhindert wird dass undefinierter (uninitialized) Speicher gelesen wird.
Capacity	Wie groß die Capacität des Buffers ist.

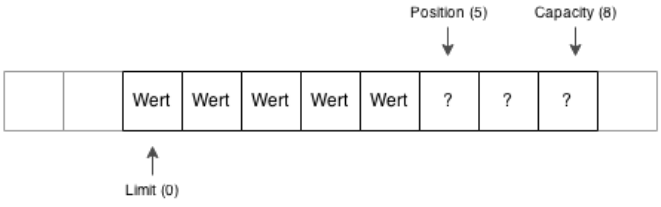
Buffer nach Initialisierung mit Kapazität 8



Buffer nach Schreiben von einem Wert in den Buffer (put)



Buffer nach Schreiben von 4 Werten in den Buffer (put)

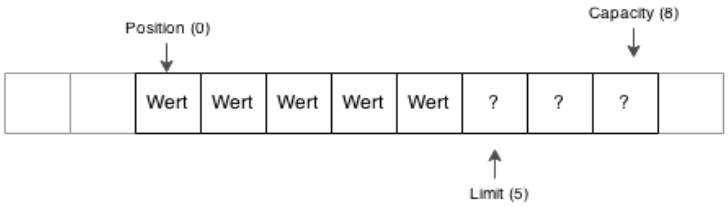


Damit wir den Buffer nun lesen können müssen wir `limit` und `position` anpassen. Und zwar müssen wir das Limit auf unsere momentane Position setzen und die Position auf 0 zurücksetzen. Dies kann man entweder manuell machen über die jeweiligen Methoden `position(int)` und `limit(int)` oder man benutzt die Methode `flip()` die einen Shortcut für

```
limit(position());
position(0);
```

darstellt.

Buffer nach flip()



Buffer sind in Java, neben Arrays und Listen, eine weitere Möglichkeit primitive Typen zu sammeln. Buffer sind sehr Hardware nah und werden daher oft für die Kommunikation zwischen Java und anderen Programmiersprachen verwendet.

(Buffer sind quasi wie *C-Arrays* mit Pointern)

Auch OpenGL verwendet Buffer um Daten intern zuspeichern, diese werden *BufferObject* genannt.

BufferObject (OpenGL)

Ein OpenGL Buffer.

Color4f (Fachwert)

Ein Vektor mit 4 Komponenten, `r`, `g`, `b`, & `a`. Wird dafür verwendet um eine Farbe zu beschreiben. `r` entspricht dem Rot-Anteil, `g` dem Grün-Anteil, `b` dem Blau-Anteil, und `a` der Sichtbarkeit. Der Wertebereich für `r`, `g`, `b` & `a` liegen zwischen `0.0f` und `1.0f`.

`a = 1.0f` entspricht voller Sichtbarkeit, `0.0f` kompletter Transparenz, und `0.5f` halber Sichtbarkeit/Transparenz.

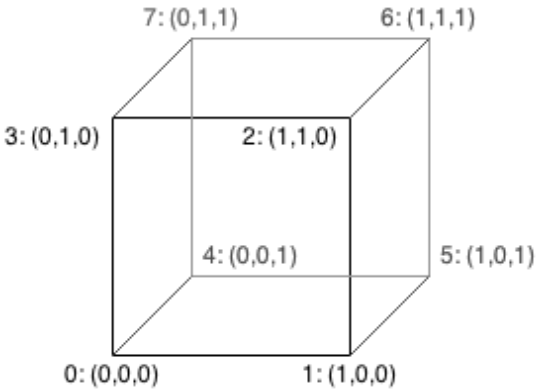
DrawCall (OpenGL)

Sobald Vertices an OpenGL übergeben wurden, Texturen an TexturUnits gebunden wurden, ein Shader als aktiver Shader gesetzt wurde und dann eine Funktion mit der Signatur `glDraw…` aufgerufen wird, dann werden die Vertices durch den Shader *gejagt* und in den BackBuffer gemalt.

GameObject (Engine: Logic)

Technik: IndexedRendering

Es kommt des Öfteren vor das einzelne Vertices mehrfach verwendet werden um ein Objekt zurendern, z.B. eine Ecke eines Würfels wird mind. 4 mal benutzt, wenn der Würfel aus Dreiecken gerendert wird sogar mind. 8 Mal. Der naive Ansatz wäre einfach die Vertices wie folgt an die GPU zusenden:




```
// Vorderes Quadrat
(0,0,0) , (1,0,0), (0,1,0), // 1. Dreieck
(1,0,0) , (1,1,0), (0,1,0), // 2. Dreieck

// Rechtes Quadrat
(1,0,0) , (1,0,1), (1,1,0), // 1. Dreieck
(1,0,1) , (1,1,1), (1,1,0), // 2. Dreieck

...
```

Es ist einfach zusehen dass wir sehr viel redundante Informationen an die GPU senden. Daher führen wir die Technik des *Indexed Rendering* ein. Wir übergeben als erstes wie gewohnt Vertices an den VBO, allerdings diesmal nur jeden Vertex genau einmal. Anschließend erzeugen wir einen Integer-Buffer in dem wir die Reihenfolge angeben in welcher Reihenfolge die Vertices gerendert werden sollen. Das gute hierbei ist, dass Vertices mehrfachverwendet werden können.

Angenommen wir haben die Vertices der Reighenfolge nach, wie in der Abbildung durchnummeriert, in den VBO geschrieben, würde der Index-Buffer wie folgt aussehen:

`0, 1, 3, 1, 2, 3, 1, 5, 2, 5, 6, 2, …`

Nun kann man sich leicht ausrechnen was wir an Datenvolumen gespart haben:

Naiver Ansatz:

> 6 * 6 Vertices (als Dreiecke rendern) der Größe 36 Bytes : 1296 bytes

Indexed Rendering:

> 8 Vertices der Größe 36 Bytes, 36 Indizes (Integer der Größe 4 bytes) : 432 bytes

Wir haben also bereits bei diesem „kleinen“ Beispiel das übertragene Datenvolumen auf ca. 1/3 reduziert.

Wenn wir nun den Würfel zeichnen wollen verwenden wir nicht mehr wie gewohnt `glDrawArrays` sondern `glDrawElements` und übergeben hier den Index-Buffer.

Technik: **Instanced Rendering** (OpenGL)

Es kommt hin und wieder vor dass man ein und dasselbe Objekt, identisch an vielen Plätzen darstellen will (Achtung, die Objekte müssen exakt identisch sein, nicht ein Attribut darf abweichen, ansonsten ist InstancedRendering nicht möglich, zumindest nicht mit unserem Code). Dies erreicht man indem man die Vertexdaten für ein Objekt in einen VBO speichert und dann separat für jede Instanz eine Transformationsmatrix übergibt. Der Renderer rendert dann das übergebene Objekt das im VBO liegt einmal kombiniert mit jeder Transformationsmatrix die im DrawCall übergeben wurden.

Material (Engine: Rendering)

Besteht genau 1 Texture und 1 Shader. Gegebenenfalls auch einer Color4f.

Matrix4f (Fachwert)

Eine 4 x 4 Matrix. Wird im Allgemeinen dazu verwendet um eine Transformation (*siehe Transform*) im 3-dimensionalen Raum zubeschreiben. Matrizen werden insofern gerne verwendet da man konkrete Punkte im 3D-Raum die als Vector3f dargestellt sind einfach mit einer Matrix multiplizieren kann und damit auch schon die komplette beschriebene Transformation auf den Punkt angewendet wurde.

(Achtung: Matrix ist 4x4, die Vektoren jedoch nur 1x3, das heißt wir müssen vor jeder Matrix-mit-Vektor Multiplikation den Vektor in einen Vektor 1x4 umwandeln. Dies geht ganz einfach, man übernimmt die 3 Komponenten wie sie sind und „hängt" eine vierte Komponente mit dem Wert `1.0f` an. Dies beeinflusst nicht das Resultat.

Pseudo-Code Bsp:

```
matrix * vec4( vector.x, vector.y, vector.z, 1.0f))
```

Modelspace (OpenGL)

Module (Engine)

Module: LogicModule (Engine: Logic)

Module: RenderModule (Engine: Rendering)

Module: PhysicModule (Engine: Physics)

Module: InputModule (Engine: Input)

Projectionspace (OpenGL)

Quaternion4f (Fachwert)

Wird dazu benutzt um Rotationen zu beschreiben, ohne die Rotation als Verkettung von Rotation über die (Euler-)Winkel (Pitch, Roll und Yaw) zu betrachten. Grund hierfür ist, dass die Betrachtung von Rotationen als Verkettung von Rotationen um Eulerwinkel zu einen *Gimbal-Lock* führen kann.

Quaternionen können miteinander multipliziert werden um Rotationen zu kombinieren (Quaternionen-Multiplikationen sind wie Matrix-Multiplikationen NICHT kommutativ)

(Wie genau eine Rotation in Quaternionen gespeichert/repräsentiert wird weiß ich nicht mehr, ich werde ggf. Quellen zu dem Thema noch ergänzen.)

Shader (OpenGL)

Auch Shader-Program genannt, ist eine Kompilation von mehreren Shadermodulen. Diese legen fest wie Vertices durch die Grafikkarte visualisiert werden. Ein Shadermodul wird in einer C ähnlichen Sprache geschrieben „GLSL" (GL Shading Language).

Shader: **Fragmentshader** (OpenGL)

Wird als zweites Shadermodul ausgeführt. Es legt die Farbe fest die der Vertex hat.

```
#version 330 // Welche GLSL Version benutzt wird

uniform sampler2D uni_texture0; // Texture Slot 0, an den eine Texture gebunden werden kann

in vec4 fs_color;
in vec2 fs_texcoord;

out vec4 out_color;

void main() {
    out_color =  texture2D(uni_texture0,fs_texcoord) * (fs_color + vec4(0.5,0.5,0.5,0));
}
```

Shader: **Vertexshader** (OpenGL)

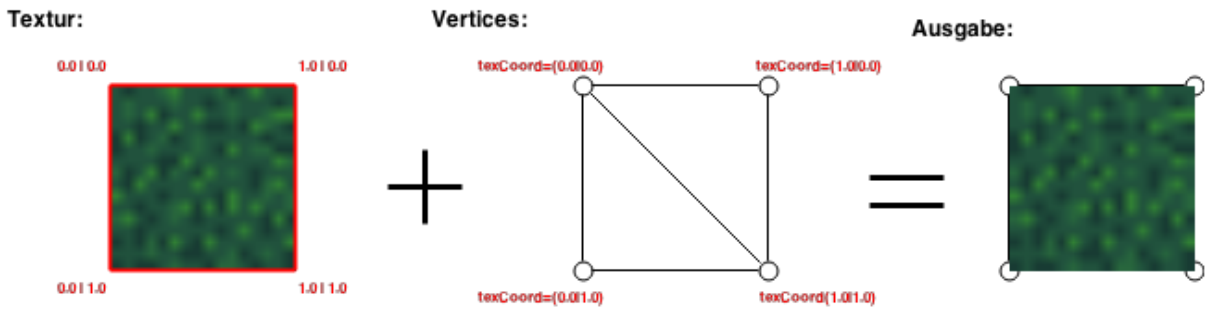
Wird als erstes Shadermodul ausgeführt. Es legt die Position fest an dem der Vertex gerendert wird.

Die Position an der der Vertex gerendert werden soll muss der von OpenGL vordefinierten Variable `gl_Position` zugewiesen werden.

```
#version 330                // Welche GLSL Version benutzt wird

uniform mat4 uni_wvp;       // Die Model/View/Projection-Matrix

// layout: Position des Attributpointers; in: Eingehende Variable mit Wert; vec2/vec3/vec4/mat4… GLSL-Typen
layout(location = 0) in vec3 vs_position;// Die Attributposition in dem VBO mit der Position
layout(location = 1) in vec4 vs_color;    // Die Attributposition in dem VBO mit der Farbe
layout(location = 2) in vec2 vs_texcoord;// Die Attributposition in dem VBO mit der Tex-Koordianten

out vec4 fs_color;          // Hier stellen wir 2 out-Variablen bereit deren Werte an den Fragment-
out vec2 fs_texcoord;       // Shader weiter gegeben werden


void main() {
    gl_Position = vec4(vs_position, 1.0) * uni_wvp; // Position des Vertex * MVP
    fs_color = vs_color;          // Farbe & Tex-Koordinaten an den Fragmentshader weiter
    fs_texcoord = vs_texcoord;    // reichen
}
```

Texture (OpenGL)

Ein Bild das mittels eines Shaders „auf Vertices gelegt“ wird. Die `texCoord` Werte eines Vertex geben an welche Texturkoordinate an den Vertex gemapped werden.



TileMap (Engine: Rendering)

Eine 2D-Spielwelt wird oft in rechteckige Bereiche unterteilt. Am besten stellt man sich das wie eine Mauer vor (daher auch der Name *tile, englisch* für *Ziegel*). Eine TileMap ist eine Ansammlung von Tiles die dann zusammen eine zusammenhängende Welt suggerieren soll.

Eine TileMap hat ein 2D-Array indem an jedem Platz eine Zahl zwischen 0 und *n* steht, die einem Unterbild im dazugehörigen Tileset entspricht. Die TileMap zeichnet nun für jede Zahl im Array an dem entsprechenden Platz das dazugehörige Unterbild des Tilesets, so entsteht ein zusammenhängendes, allerdings sehr flexibel modifizierbares Gesamtbild.

TileSet (Engine: Rendering)

Ein Tileset ist ein Bild dass mehrere Unterbilder besitzt. Diese Unterbilder werden von eine TileMap als Tiles dargestellt. Als Beispiel das Tileset von Minecraft (http://s44.photobucket.com/user/Dante80/media/Minecraft/terrain-6.png.html).

Die Unterbilder eines Tileset's werden typischerweise durchnummeriert (beginnend von oben -links nach oben rechts, und von oben nach unten 0,1,2,...,*n*.

Transform (Engine: Rendering)

Beschreibt eine Transformation im 3-dimensionalen Raum. Teil einer Transformation ist die Beschreibung der Position (Translation), der Skalierung (Scaling) und der Rotation (Rotation).

Die Translation wird repräsentiert durch einen *Vector3f.*

Das Scaling wird repräsentiert durch einen *Vector3f.*

Die Rotation wird repräsentiert durch einen *Quaternion4f.*

Vertex (Engine: Rendering)

Beschreibt einen Punkt im grafischen 3D-Raum. Ein Vertex besteht meistens aus der Position im 3D-Raum (*Vector3f*), der Farbe (*Color4f*), und einer Texturkoordinate (damit OpenGL weiß wie eine Texture auf ein Objekt gelegt werden soll)(*Vector2f*)

Texturkoordinaten = UV = TexCoord = TexKoordinaten

VertexArrayObject / VAO (OpenGL)

Wird in OpenGL benutzt um weitere Render-Objekte zubündeln. Enthält sonst keine weiteren Daten. Enthält immer ein VBO. Kann enthalten IndexBuffer oder InstanceBuffer.

VertexBufferObject / VBO (OpenGL)

Ist ein OpenGL Buffer der Vertices speichert

Ist ein Puffer im Grafikkartenspeicher der dafür benutzt wird um Vertices zuspeichern.

Vector3f (Fachwert)

Ein Vektor mit 3 Komponenten, x, y & z. Wird u.a. dafür verwendet Positionen oder Skalierungen im 3-dimensionalen Raum zubeschreiben.

Hier entspricht dann z.B. die x Komponente die Distanz auf der X-Achse zum Ursprung, bzw. die Skalierung auf der x-Achse.

Vector2f (Fachwert)

Ein Vektor mit 2 Komponenten, x & y. Wird u.a. dafür verwendet Positionen oder Skalierungen im 2-dimensionalen Raum zubeschreiben.

Viewspace (OpenGL)

Manager