Ostfalia - Hochschule für angewandte Wissenschaften - Braunschweig/Wolfenbüttel -

WPF - Computergrafik

B.Sc. Fabian Sonczeck

April 20, 2016

Part I.
Einführung in Computergrafik

1. Grundlegender Aufbau der grafischen Datenverarbeitung (GDV)

Figure 1: Aufbau Grafiksystem

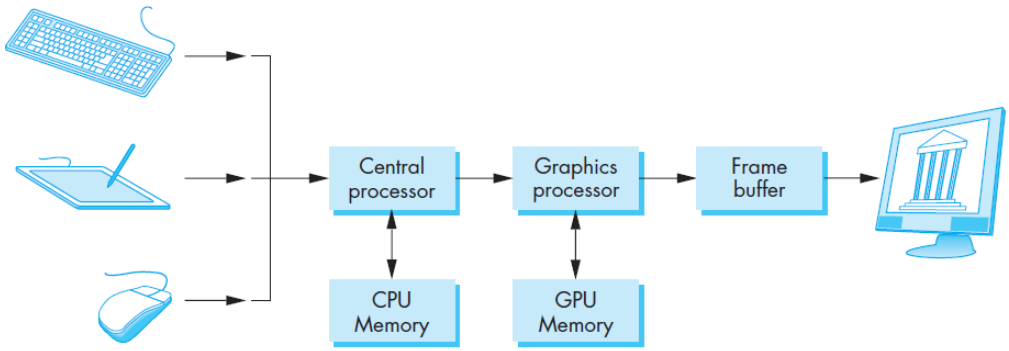


FIGURE 1.1 A graphics system.

Anhand der Grafik erkennt man, dass folgende Komponenten in einem Grafik-System eine bestimmte Rolle einnehmen:

- Eingabegeräte
- CPU
- GPU
- Framebuffer
- Ausgabegeräte

1.1. Framebuffer

In der Regel sind heutzutage alle modernen Grafikkarten “rasterbasiert”. D.h. was wir auf dem Monitor sehen ist ein “Array” von Bildelementen (Pixel), welche innerhalb eines Grafikprozessors berechnet worden sind. Alle berechneten Bildelemente werden im sog. “Framebuffer” abgelegt bzw. gespeichert. Die Anzahl der Pixel innerhalb des Framebuffers bestimmt den Detailreichtum eines Bildes für den menschlichen Betrachter. Ein Bild erscheint uns realistischer, je kleiner man die einzelnen Bildbereiche setzt und somit ein Bild mit mehr Pixeln darstellen kann. Man spricht in diesem Sinne auch von “Auflösung”. Eine Auflösung von 600*400 bedeutet also, dass der Framebuffer 240.000 Pixel enthält, welche jeweils Farbinformationen an einer bestimmten Position enthalten.

Weiterhin bestimmt die “Tiefe” bzw. “Präzision” die Anzahl der Bits die pro Pixel für Farbinformationen genutzt werden kann. Beispielsweise bestimmt eine Tiefe von 1-bit, dass ein Pixel nur zwei Farbwerte annehmen kann 0 und 1 (aka Schwarz und weiß).

see reference (Wiki Color-Depth)

Part II.
Einführung in OpenGL

Figure 2: Grafik API Einbindung

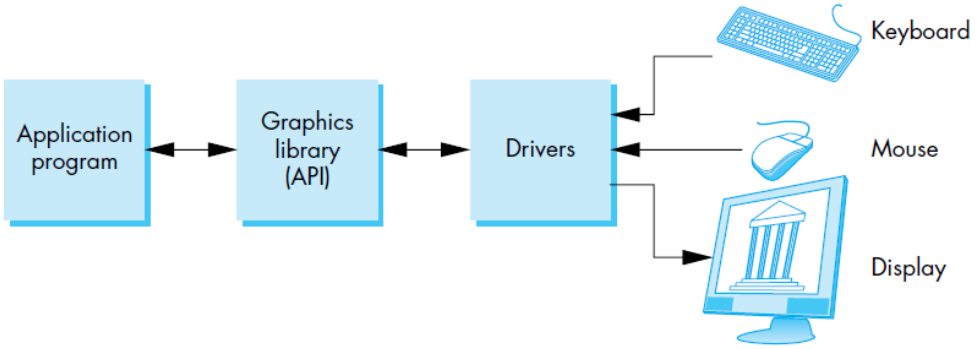


FIGURE 1.28 Application programmer's model of graphics system.

2. Einführung OpenGL

2.1. Was ist OpenGL?

OpenGL ist eine sogenannte “API” - (Application Programming Interface) oder zu deutsch eine Programmierschnittstelle.
Sie liefert eine Bibliothek mit deren Hilfe man virtuelle Szenen auf dem Bildschirm erzeugen kann. Sie ist Hardware und Platformunabhängig. Allerdings liefert OpenGL keine Funktionen um Fenster oder Benutzereingaben zu verarbeiten. Diese Funktionen müssen für die Applikation separrat (z.B. über FreeGlut oder GLFW) eingebunden werden.

Figure 3: Aufbau der Bibliotheken

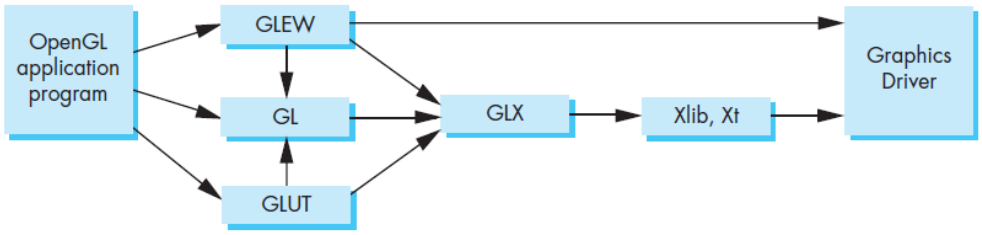


FIGURE 2.4 Library organization.

2.2. Zusätzliche Bibliotheken

see reference (Toolkits)

2.2.1. Kontext bzw. Fensterhandler

Folgende Bibliotheken sind am Meisten verbreitet um einen OpenGL Kontext bzw. ein Fenster zu erzeugen um in diesem zu “Rendern”:

- GLFW
- FreeGLUT
- SDL

2.2.2. Hardware wrapper

Diese Bibliothek befasst sich mit der Kompatibilität zwischen OpenGL und der Zielplatform. Sie lädt für uns die OpenGL Funktionen, welche unter der aktuellen Platform und der gewünschten OpenGL Version möglich sind. Mithilfe von GLEW ist es zum Beispiel nicht möglich OpenGL Befehle auszuführen, wenn diese nicht von der Platformumgebung unterstützt wird. Sodass man schon zur Compilezeit Fehlermeldungen erhält.

- GLEW

2.2.3. Texturen

Bibliotheken zum einbinden von Bild-Formaten:

- SOIL
- FreeImage

3. Malen nach Daten

3.1. Drawing Techniques

3.1.1. Immediate Mode bis OpenGL 2.x

In älteren Versionen war es in OpenGL etwas einfacher und vielleicht auch lesbarer um geometrische Objekte bzw. “Primitives” auf den Bildschirm zu zaubern.

Beispiel Immediate Mode:

```
1  void Render()
2  {
3    PreRenderSetup();
4
5    glBegin(GL_TRIANGLES);
6      glVertex3f(-1.0f,0.0f,0.0f);   //lower-left corner
7      glVertex3f(0.0f,1.0f,0.0f); //lower-right corner
8      glVertex3f(0.5f,1.0f,0.0f); //upper-mid
9    glEnd();
10 }
```

Immediate Mode bedeutet im Prinzip: “Sende folgende Daten an die GPU und stelle sie direkt auf dem Bildschirm dar”. Alle Primitiven Objekte die so erzeugt werden, müssen jedesmal von neuem generiert und an die GPU gesendet werden (Stichwort Performance). In unserem Fall wird das Dreieck jeden Frametick neu berechnet und dargestellt.

3.1.2. Retained Mode

Im “Immediate Mode” wurden Primitve Objekte erzeugt und direkt dargestellt. Eine etwas erweiterte Methode ist der Retained Mode, in dem die Primitiven Punkte/Objekte zunächst in einem Array der Applikation zwischengespeichert werden und zu gegebener Zeit komplett an die GPU gesendet und auf dem Bildschirm dargestellt werden. Dadurch, dass die Objekte gespeichert werden kann man nachträglich noch etwaigen Einfluss auf deren Daten nehmen, jedoch ist auch hier die GPU stark beansprucht. Ein simples Beispiel ist die Bewegung bzw. Animation eines Objektes zu einer neuen Position. Wenn sich nur der Ort ändert so muss doch jeder einzelne Punkt (Vertex) des Objektes verändert werden und anschließend als Komplettpaket an die GPU gesendet werden.

Beispiel Retained Mode:

```
int main()
{
  Triangles[] triangles = MakeSomeTriangles();
}

void Render()
{
  for(each_triangles)
  {
    triangle->Display();
  }
}

Triangle::Display()
{
  glBegin(GL_TRIANGLES);
    glVertex3f(point1);
    glVertex3f(point2);
    glVertex3f(point3);
  glEnd();
}
```

3.1.3. Buffered Mode

Während die ersten beiden Modi ihre Daten in der Applikation selbst erzeugen (und damit auf dem RAM bzw. CPU Memory ablegen) und speichern, befasst sich diese Methode mit der Variante alle Primitiven Objekte im Speicher der GPU abzulegen und diese Daten zu nutzen um sie auf dem Bildschirm darzustellen. Wir erzeugen uns einen sogenannten “Buffer” dafür. Ein Buffer selbst ist nichts weiter als ein Array. Dieser Buffer speichert Vertexinformationen und wird, wie gesagt, in den GPU-Memory abgelegt. Der Zugriff auf diesen Speicher ist um einiges schneller und somit effizienter und es stellt somit kein Problem mehr dar, große Mengen an Daten auf einmal darzustellen.

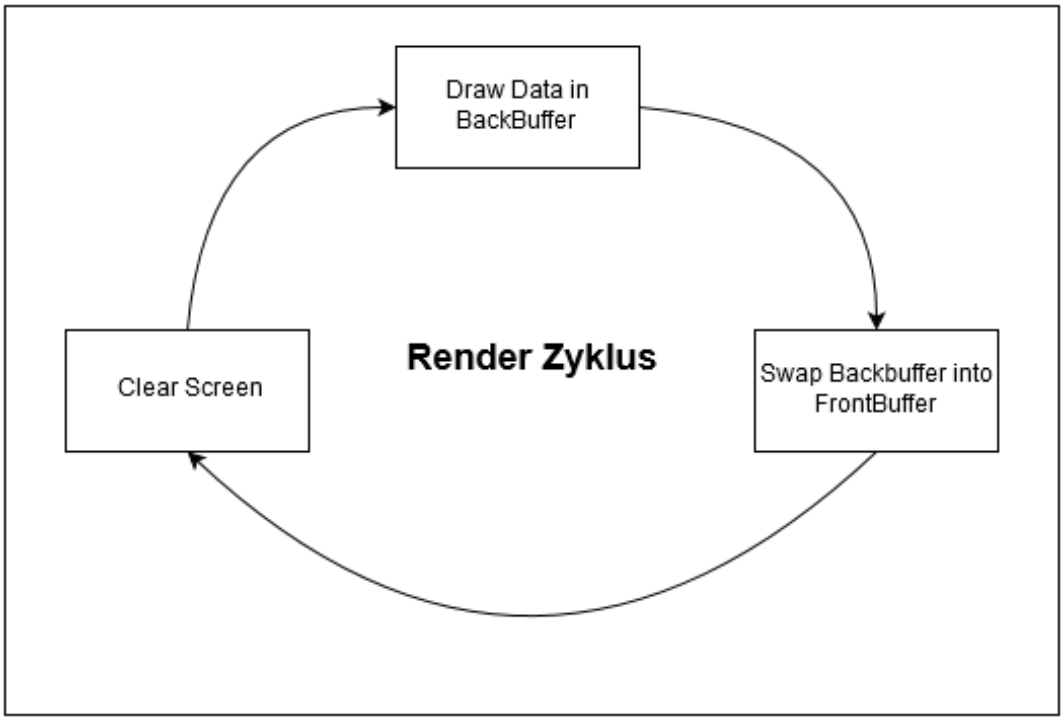
Beispiel Buffered Mode:

```
1  int main()
2  {
3     Triangles[] triangles = MakeSomeTriangles();
4     StoreDateIntoGPUMemory(triangles);
5  }
6
7  void Render()
8  {
9     GPU->Flush();
10 }
```

3.1.4. Render Zyklus

Hier einmal ein Diagram, wann “was” in OpenGL gemacht wird um etwas auf dem Bildschirm zu präsentieren:

Figure 4: Render Zyklus



Part III.
Umgang mit OpenGL

4. Initialisierung

4.1. Initialisieren des Fensters und erzeugen des OpenGL Kontexts

Um einen OpenGL Context und ein Fenster zu erzeugen, verwenden wir in diesem Skript die GLFW Bibliothek. Um überhaupt eine bemalbare Oberfläche zu haben benötigen wir ein Fenst ("Window"):

Welche Aufgaben hat ein "Window"?

- Um ein "echtes" Window zu erzeugen, möchte GLFW (und die anderen Bibliotheken gehen nahezu identisch vor) Informationen über die Größe des Fensters, sowie den Fenster Titel haben.
- Sie beinhaltet die "Update" bzw. "Render" Methode, die dafür verantwortlich ist die Daten von der GPU auf dem "Fenster" anzuzeigen.
- Ein Window hat in der Regel einige Rückruf-Funktionen (Callbacks), welche bei bestimmten Ereignissen ausgeführt werden sollen. Bekannte Callbacks sind u.a.:
  - Resize -> Wenn sich die Größe des Fenster verändert
  - Input -> Sofern das Window aktiv im Fokus steht, werden Input Events, wie Tastendruck, Mausbewegung etc. aufgerufen
- Sie löscht den aktuellen Bildschirm("Framebuffer") bevor etwas neues darauf gezeichnet wird.

4.1.1. Init und Terminierung von GLFW

Um GLFW zu initialisieren um danach einen Kontext zu erstellen müssen wir den folgenden Befehl ausführen:

```
1 glfwInit()
```

Wenn die Anwendung geschlossen werden soll muss GLFW terminiert werden.

```
1 glfwTerminate();
```

4.1.2. Erstellen eines GLFW Windows

Mit der Einbindung von “glfw3.h” haben wir Zugriff auf die Funktionen von GLFW.

Um ein “Window” Objekt zu erzeugen nutzen wir den Befehl:

```
GLFWwindow* window = glfwCreateWindow(Breite, Höhe, Titel, NULL,
    NULL);
```

Wir erhalten einen Pointer von GLFW auf eine Struct GLFWwindow.

Der dritte Parameter steht für den Monitor auf welchem das Fenster im Fullscreen angezeigt wird. Ist dieser nicht definiert, wird ein Fenster erzeugt.

Mit dem letzten Parameter besteht die Möglichkeit zwischen Mehreren Fenster information zu teilen. Diese Funktion wird aber innerhalb der Vorlesung nicht behandelt.

Und Fullscreen?

Um das Fenster im “Vollbild” Modus zu erzeugen, muss definiert werden auf welchem Monitor, dieses passieren soll. Und dieser Monitor wird dann an den 3. Parameter übergeben.

Über den Befehl:

```
GLFWmonitor* primaryMonitor = glfwGetPrimaryMonitor()
```

erhalten wir einen Handle auf den ersten aktiven Bildschirm.

Alternativ kann man auch ein Array aller Monitor erhalten über:

```
int anzahlMonitore;
GLFWmonitor** monitors = glfwGetMonitors(&anzahlMonitore);
```

Setzen des aktuellen Window Kontextes

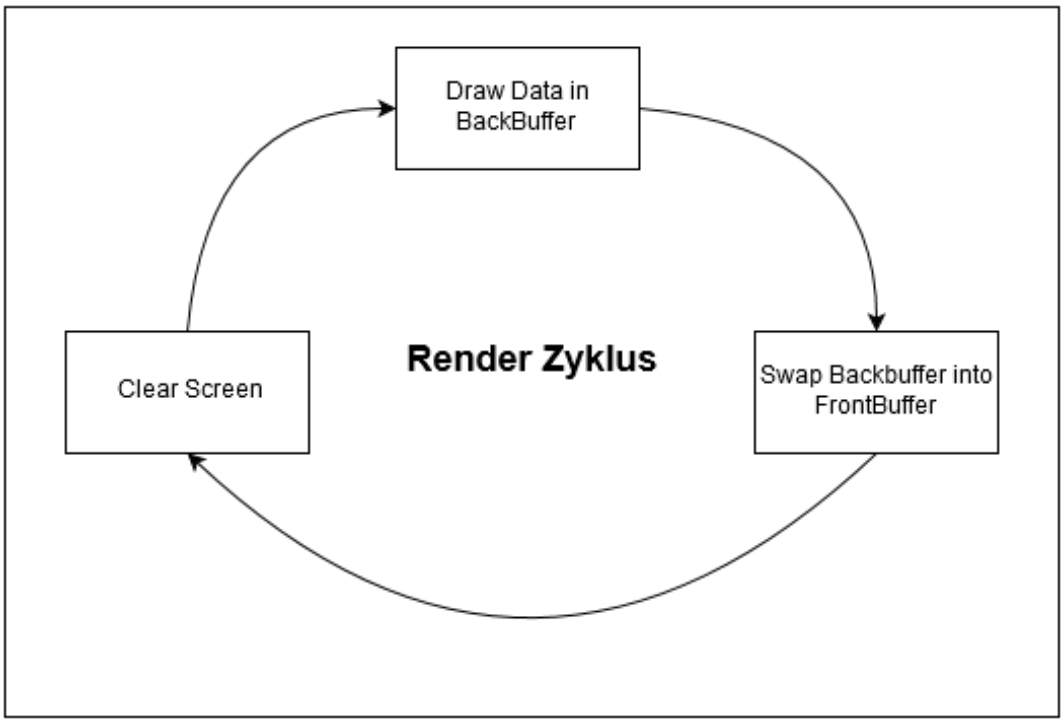
Nachdem man einen Handle auf ein GLFWwindow hat muss OpenGL mitgeteilt werden, dieses zu verwenden, sodass beim ausführen des Programms auch ein Fenster angezeigt wird.

```
glfwMakeCurrentContext(window);
```

4.1.3. Die Update-Schleife

Zunächst einmal der Zyklus des “Renderings”:

Figure 5: Render Zyklus



Dieser Zyklus sieht immer gleich aus

1. Das aktuelle Fenster (Vordergrund-Buffer) wird mit einer Hintergrundfarbe “gelöscht”
2. Man zeichnet die “Renderables” auf den “Hintergrund” Buffer (nicht für den Client sichtbar)
3. Nachdem das Zeichnen beendet ist, wird der Hintergrund-Buffer durch den Vordergrund-Buffer getauscht (das Gezeichnete ist nun für den Client sichtbar)
4. Fange von vorne an

Clear

Man kann in OpenGL eine Hintergrundfarbe definieren mit der gelöscht wird (Sofern nicht definiert wird Scharz (RGBA: 0,0,0,1) gesetzt):

```
glClearColor(R,G,B,A); // also für z.b rot (1,0,0,1);
```

Innerhalb einer Update Schleife wird als erstes der Bildschrim gelöscht:

```
glClear(GL_COLOR_BUFFER_BIT);
```

Malen

Jetzt soll alles das gezeichnet werden, was im nächsten Frame sichtbar sein soll (oder zumindest theoretisch da ist). Ob ein Objekt sichtbar ist hängt natürlich davon ab,

ob es innerhalb des “sichtbaren” Bereichs des Bildschirms ist.

Swappen der Framebuffers

In GLFW passieren hier 2 Dinge einmal wird über den Aufruf von:

```
glfwPollEvents();
```

abgefragt ob Ereignisse wie Input oder WindowResize vorhanden sind und mit diesem Befehl abgearbeitet werden. (Die gesetzten Callbacks werden dann aufgerufen).

Und zuletzt werden die beiden Framebuffer Back und Front getauscht (Die Anzahl ist standardmäßig 2, in älteren OpenGL Versionen 1 und es können auch mehr als 2 Framebuffer möglich sein):

```
glfwSwapBuffers(window);
```

4.1.4. Further Information

Unter: GLFW Window Doku

erhalten Sie detailierte Information um zum Beispiel Informationen aus dem Fenster zu Laufzeit zu bekommen oder wie die Callback Signaturen aussehen müssen um eigene Funktionen binden zu können.

4.2. GLEW Init

Wie bereits erwähnt beinhaltet GLEW Sicherheitsmechanismen bezüglich der Kompatibilität zwischen OpenGL und der Platform. Auch sind in dieser Bibliothek die typischen OpenGL Datentypen definiert, wie “GLint, GLuint, GLsizei, GLchar, usw....). Immer wenn wir also OpenGL Datentypen nutzen wollen können wir dies mit Hilfe des Includes (glew.h) tun.

Um GLEW zu initialisieren MÜSSEN wir dieses NACH der Erzeugung eines OpenGL Kontextes (siehe Kapitel: 4.1) machen. Der Befehl dazu sieht folgendermaßen aus:

```
1 glewInit();
```

Achtung!

Wer in die Versuchung gerät und prüfen möchte ob die Initialisierung erfolgreich war, der wird über folgendes Beispiel enttäuscht werden:

```
1 if(glewInit())
2 {
3   // Erfolgreich?
4 }
```

Diese Methode liefert bei Erfolg eine Definierte Konstante zurück namens (GL_NO_ERROR), welches das Symbol bzw. den Wert “0” ersetzt und dementsprechend “false” liefert.

Richtig wäre z.B. folgende lesbare Kontrolle:

```
1 if(glewInit() == GL_NO_ERROR)
2 {
3 }
```

5. Modernes “Malen” - The Cool New Fancy OpenGL Way!

5.1. Primitive Drawmodes

Im folgenden werden die verschiedenen OpenGL Drawmodes vorgestellt, wie eine Reihe von zu malenden Punkten von OpenGL interpretiert werden können.

5.1.1. Einzelne Punkte

Ein Reihe(Array) von Punkten, welche wir auf unserem Bildschrim darstellen werden, können von OpenGL unterschiedlich interpretiert und behandelt werden.

Man könnte innerhalb des Arrays davon ausgehen, dass jeder Punkt keine Relation zu einem anderen hat. Sodass auf unserem Bildschirm nur ein Bild mit einzelnen Punkten entsteht. Wie man sich vielleicht vorstellen kann sind “Punkte” die einfachste Form unter den Primitives.

```
1 // PseudoCode
2
3 Punkte = A,B,C;
4
5 Male Punkte und interpretier sie als GL_POINTS
6
7 Draw(A,B,C); // Ein Punkt bei A, ein Punkt bei B, ein Punkt bei C
```

Listing 1: Drawmode: GL_POINTS

5.1.2. Linien

Die nächste höhere Form von Punkten sind je 2 Punkte die von OpenGL mit einer Linie verbunden werden soll. Tatsächlich ist selbst die Linie nur eine Reihe von Punkten die dicht bei einander ein Linie für das Auge suggerieren. Das trifft für all primitiven Formen zu.

GL_LINES

Über GL_LINES erwartet OpenGL immer 2 Punkte für eine Linie. Als Beispiel haben wir 3 Punkte A,B,C. Zwischen A und B, sowie B und C sollen Linien entstehen. Wir müssen OpenGL 4 Punkte als Daten übergeben, damit dies funktioniert.

```
// PseudoCode

Punkte = A,B,C;

Male Punkte und interpretier sie als GL_LINES

Draw(A,B,B,C); // Linie zwischen A und B, Linie zwischen B und C
```

Listing 2: Drawmode: GL_LINES

GL_LINE_STRIP

Die Redundanz bei zusammenhängenden Linien kann durch die Nutzung von GL_LINE_STRIP vermieden werden. Dieser Befehl geht davon aus, dass die ersten beiden Punkte die erste Linie ergibt. Jeder weitere Punkt wird mit dem vorherigen verbunden:

```
// PseudoCode

Punkte = A,B,C, ..., n;

Male Punkte und interpretier sie als GL_LINE_STRIP

Draw(A,B,C, ...,n); // Linie von A nach B nach C nach ... nach n
```

Listing 3: Drawmode: GL_LINE_STRIP

Zusatzform GL_LINE_LOOP

Diese Konstante agiert genauso wie GL_LINE_STRIP allerdings wird zusätzlich der erste und letzte Punkt mit einer Linie verbunden

5.1.3. Dreiecke

Die wohl gängigste Form des “Zeichnens” sind Dreiecke, sie bilden die primitivste Form an um Flächen darstellen zu können. Die Reihe an Punkten, können auf 3 unterschiedliche Arten verstanden werden.

GL_TRIANGLES

Die einfachste Dreieck Methode ist diese, welche für jedes zu malende Dreieck auch 3 Punkte erhält. D.h. auf der anderen Seite, die Anzahl Punkte muss, theoretisch, durch 3 teilbar sein. Es kommt zwar zu keinem Fehler, wenn das nicht der Fall ist, aber so versteht man das Prinzip deutlicher.

```
1 // PseudoCode
2
3 Punkte = A,B,C,D,E,F;
4
5 Male Punkte und interpretier sie als GL_TRIANGLES
6
7 Draw(A,B,C,D,E,F); // Ein Dreieck mit den Punkten A,B,C; Ein Dreieck
      mit den Punkten D,E,F
```

Listing 4: Drawmode: GL_TRIANGLES

GL_TRIANGLE_STRIP

Wenn wir eine Reihe von zusammenhängenden Dreiecken haben, so können wir diese auch als Strip zeichnen. Die ersten 3 Punkte bilden dabei das erste Dreieck, und jeder zusätzliche Punkt definiert ein weiteres Dreieck mit den letzten und vorletztem Punkt des vorangegangenen Dreiecks.

```
1 // PseudoCode
2
3 Punkte = A,B,C,D;
4
5 Male Punkte und interpretier sie als GL_TRIANGLE_STRIP
6
7 Draw(A,B,C,D); // Ein Dreieck mit den Punkten A,B,C; Ein Dreieck mit
      den Punkten B,C,D
```

Listing 5: Drawmode: GL_TRIANGLE_STRIP

GL_TRIANGLE_FAN

Dieser Drawmode bietet sich insbesondere dann an, wenn das zu Zeichnende einen Ausgangspunkt hat. Ein Kreis zum Beispiel hat einen Mittelpunkt und augehend von diesem und dem Radius des Kreises werden viele Dreiecke gezeichnet um den “Kreis” zu formen. Dabei gilt je mehr Dreiecke desto “optisch” runder wird der Kreis.

```
1 // PseudoCode
2
3 Punkte = A,B,C,D;
4
5 Male Punkte und interpretier sie als GL_TRIANGLE_FAN
6
7 Draw(A,B,C,D); // Ein Dreieck mit den Punkten A,B,C; Ein Dreieck mit
      den Punkten A,C,D
```

5.1.4. Weitere Modi

Zusätzlich zu Dreiecken gibt es noch 2 weitere Formen.

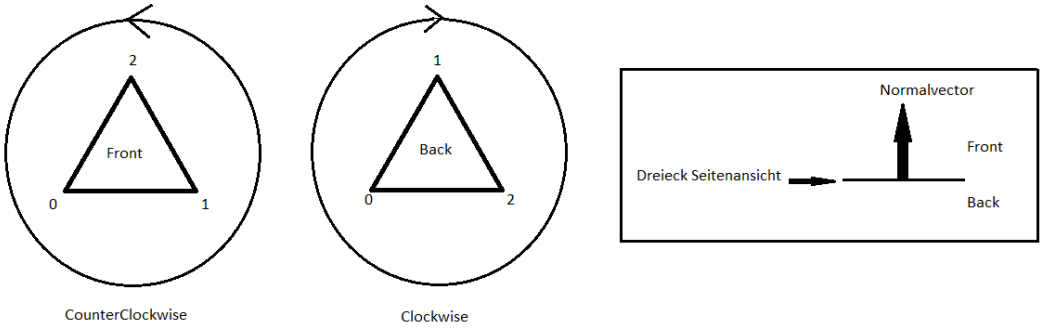
- GL_QUADS -> Erwartet 4 Punkte mit denen ein Viereck gezeichnet wird. Auch hier gilt Anzahl der Punkte durch 4 teilbar.
- GL_QUAD_STRIP -> Verbundene Vierecke. Jedes weitere Viereck wird über 2 neue Punkte erzeugt und definiert sich aus diesen und den 2 vorherigen Punkten.
- GL_POLYGON -> Polygon = Vieleck, Alle Punkte definieren jeweils die Ecken des zu erzeugenden Vielecks.

5.1.5. Eigenschaften von Polygonen

Sobald Flächen gezeichnet werden (also Polygone, zu denen natürlich auch Dreiecke und Vierecke gehören). Hat eine Fläche eine Vor- und eine Rückseite. Diese werden auch “Front Face” und “Back Face”

genannt. Über die Reihenfolge der Punkte eines Polygons wird bestimmt wo vorne und wo hinten ist. Die Konvention besagt, dass Flächen, dessen Punkte-Reihenfolge entgegen des Uhrzeiger Sinns erzeugt werden als Front bezeichnet wird. Auf dieser Fläche steht der Normal Vektor. Diese Eigenschaft wird entsprechend interessant, wenn man mit Culling (also das “Nicht-Zeichnen”) von Flächen arbeitet. Sowie auch für Lichtberechnung.

Figure 6: Beispiel Front/Back Face Normal



6. Shader - Einführung in GLSL

6.1. Intro

Im modernen OpenGL geht ohne Shader nicht wirklich viel. Von daher müssen wir uns wohl oder übel mit diesem Thema auseinander setzen. Wir stellen uns zunächst die Frage was Shader überhaupt sind.

6.2. Was sind Shader?

Shader sind kleine Programme, welche innerhalb der GPU kompiliert und auf jeden einzelnen Vertex angewendet werden.

Zur Erinnerung:

Figure 7: Grafik Pipeline Reminder

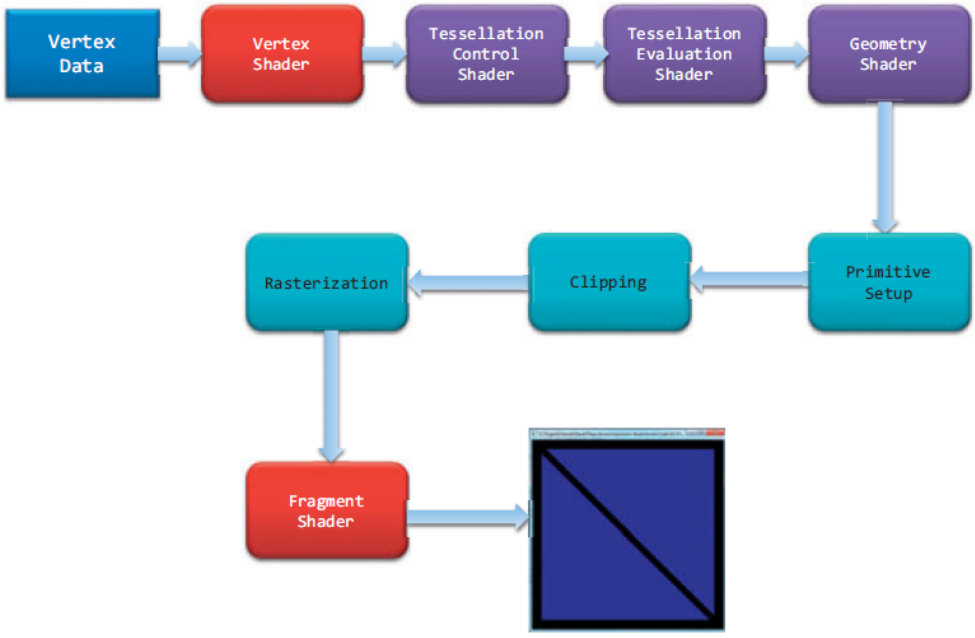


Figure 1.2 The OpenGL pipeline

Wie wir wissen besteht jedes Model/Mesh/Sprite, kurz Renderable, mindestens aus Punkten die wir auf unserem Bildschirm zeichnen wollen. Wenn wir noch etwas professioneller werden wollen, hat jeder dieser Punkte auch noch eine Farbe, Textur Koordinaten, Normals (für Lichtberechnung) und und und...

Die Grundbasis der Shader umfasst den Vertex und den Fragment Shader. Ohne diese Beiden funktioniert das System nicht.

6.3. Standard Shadertypen

6.3.1. Der Vertex Shader

Dieser Shader ist der Eintrittspunkt der Pipeline nachdem wir unsere Vertex Daten definiert und "gebuffert" haben.

Die grundlegenste Aufgabe dieses Shaders ist, die Endposition auf dem Bildschirm zu bestimmen. Das heißt ein Vertex kommt als "Input" an und entsprechend unserer "Optik" (Dazu später mehr) wird die Position auf unserem Bildschirm berechnet. Anders ausgedrückt Bestimmt der Vertexshader die 2D Projektion eines 3D-Punktes auf unserem Bildschrim.

Hier die einfachste Form eines Vertexshaders:

```
1 #version 330 core
2
3 layout(location = 0) in vec3 vertexPosition;
4
5 void main()
6 {
7   gl_Position = vec4(vertexPosition,1);
8 }
```

Listing 6: Simple Vertex Shader

Jedes Shaderprogramm beinhaltet zumindest...:

- die Direktive der GLSL version (hier 330 core)
- Ein Shader Attribut mit Index (hier vertexPosition am Index "0")
- und eine main Funktion

GLSL hat eigene Datentypen und Konstanten, die wir beschreiben können. Im obigen Program passiert nicht viel, außer das wir sagen, dass der eingehende Vertex (vertexPosition) = gl_Position ist. Der im Vertexbuffer definierte Vertex wird nicht verändert und landet so wie er ist auf der entsprechenden Position auf dem Bildschirm. Sobald dieser Shader durchgelaufen ist, wird für diesen Vertex der nächste Shader ausgeführt.

6.3.2. Der Fragment Shader

Dieser Shader ist der Letzte, welcher den finalen Vertex berechent. Während wir durch den Vertex Shader die Position bestimmt haben, befasst sich dieser Shader mit der Koloriering des Vertex. Wir geben unserem Vertex eine Farbe.

Der einfachste Fragment Shader sieht z.B. so aus:

```
1 #version 330 core
2
3 layout (location = 0) out vec4 outColor;
4
5 void main()
6 {
7   outColor = vec4(1,1,1,1);
8 }
```

Listing 7: Simple Fragment Shader

Auch hier haben wir einen Index, welcher allerdings kein “Input” sondern einen “Output” in Form einer Farbe darstellt. Die Farbe definieren wir hier zunächst noch statisch, indem wir dem “Out” Attribut einfach “weiß” zuweisen.

Shader müssen selbstverständlich selbst geschrieben werden. In der Regel werden dafür extra Dateien angelegt, welche die jeweiligen Shader implementieren.

6.3.3. Auslagern der Shaderprogramme in sepparate Dateien

```
/* in BasicShader.vertexshader */

#version 330 core

layout(location = 0) in vec3 vertexPosition;

void main()
{
  gl_Position = vec4(vertexPosition,1);
}
```

Die Dateiendung kann freigewählt werden (also z.B. wäre auch .vs möglich). Der Name der Datei sollte die Art des Shaders wiedergeben (also welche spezielle Aufgabe er übernimmt). In diesem Fall heißt er BasicShader und definiert damit den als Standard genutzten. Die Dateiendung sollte den Shader Typen beschreiben.

```
/* in BasicShader.fragmentshader */
#version 330 core

layout(location = 0) out vec4 color;

void main()
{
    color = vec4(1, 0, 1, 1);
}
```

6.4. Kompilieren eines Shaderprogramms

Ein Shaderprogramm besteht aus mehreren (mind. 2 Shadern). Die beiden wichtigsten und mindestens vorhandenen Shader sind, wie gesagt, der Vertex und der Fragment Shader.

Shader sind, wie bereits gesagt, kleine Programme die auf der GPU ausgeführt werden.

Wenn wir unsere einzelnen Shader geschrieben haben. Kann man danach damit beginnen diese Shader in ein Shaderprogramm zu laden.

Der Ablauf dafür ist immer derselbe und sieht i.d.R. folgender Maßen aus:

- Definiere eine neue ID, welche ein Shader Programm repräsentiert

```
GLuint shaderProgramID = glCreateProgram();
```

- Jeder Shader wird mit einer ID versehen und einzeln kompiliert

```
GLuint myShaderID = glCreateShader(GL_<myShaderType>_SHADER); //
    GL_VERTEX_SHADER | GL_FRAGMENT_SHADER ...
```

- Jeder ShaderID werden i.d.R. eine odere mehrere Shaderprogramme zugewiesen, der dritte und vierte Parameter nimmt jeweils ein char array und ein int array an. Im Falle mehrerer Programme muss für jedes Programm die Länge mit gegeben werden, sodass das Programm richtig gelesen werden kann. NULL teilt in diesem Fall mit das OpenGL die Länge des Quelltextes automatisch ermitteln soll. Was möglich ist, da die Größe des statischen Arrays (in unserem Triangle Konstruktor) definiert ist.

```
glShaderSource(myShaderID, 1, &myShaderSource,NULL);
```

- Mit der Verbindung von Quellcode und der ShaderID sind wir nun in der Lage das Programm zu kompilieren.

```
glCompileShader(myShaderID);
```

- Wenn wir kompiliert haben, fügen wir diesen Shader unserem Shader Programm hinzu.

```
glAttachShader(shaderProgramID, myShaderID);
```

- Das Shader Programm muss nun mit den Shaderprozessoren auf der GPU verbunden werden. Jeder Shader Typ hat einen Prozessor, welcher das Executable Shader Programm ausführt. Wenn wir Linken werden die Shader im Shaderprogramm mit den entsprechenden Prozessoren verbunden. (Ein Vertex Shader wird in einem Vertex Prozessor verarbeitet)

```
glLinkProgram(shaderProgramID);
```

- Der folgende Befehl ist im Prinzip optional, aber da wir uns wohl oder übel später mit Fehlerbehandlung ,innerhalb der Shader Kompilierung, beschäftigen müssen bereiten wir uns schon einmal darauf vor. Wenn die vorherigen Schritte irgendwo fehlgeschlagen sind bekommen wir keine Fehlermeldung. Jeder Shader und jedes Programm hat jedoch ein InfoLog welches wir auf Fehler untersuchen können. Der folgende Befehl füllt dieses Log mit Informationen über unser Shaderprogramm.

```
glValidateProgram(shaderProgramID);
```

- Wenn wir ein Shaderprogram haben, benötigen wir die einzelnen Shaderprogramme nicht mehr, diese sind persistent in unserem Programm gespeichert. Somit können wir den Speicher der einzelnen Shader wieder freigeben.

```
glDeleteShader(myShaderID);
```

- Zuletzt teilen wir der OpenGL State Machine mit welches Shaderprogram aktuell genutzt werden soll. Es kann immer nur ein Programm zur Zeit genutzt werden und das gilt auch im Prinzip für alle Elemente in OpenGL. Wenn wir ein anderes Programm nutzen wollen müssen wir das aktuelle ausschalten.

```
/* Enable the program */
glUseProgram(shaderProgramID);

/* Disable the current Program */
glUseProgram(0);
```

6.5. Fehlerbehandlung von Shadern

Leider erhalten wir keinerlei Info darüber ob während der Kompilierung oder Validierung von Shadern und Programmen etwas schief gelaufen ist. Einziger Indikator dafür ist, dass man vermutlich nicht das auf dem Bildschirm sieht, was man erwartet hat. Meistens bleibt dieser bei einem Fehler schwarz.

Um allerdings die Shaderkompilierung zumindest als syntaktischen Fehler auszuschließen, wenden wir uns nun dazu wie wir die Fehler Informationen erhalten um diese zum Beispiel auf der Console ausgeben zu können.

6.5.1. Beispielmethode

Der folgende Code beinhaltet die allgemeine Vorgehensweise um Fehlerinformationen aus Shadern zu bekommen. Im Code prüfen wir den "Compile-Status" des Shaders. Dabei ist zu beachten, dass diese Prüfung NACH dem Befehl von glCompileShader(shaderID) durchgeführt werden muss.

Man kann sich verschiedene Informationen aus Shadern und Programmen holen. Wenn man prüfen will ob während der Compilierung des Shaders etwaige Fehler auftauchten, so müssen wir folgende Schritte unternehmen um an das OpenGL Log zu kommen.

1.
```
int success = 0;
glGetShaderiv(shaderID, GL_COMPILE_STATUS, &success);
```

   a) Prüft ob das Kompilieren erfolgreich (=1) war oder nicht.

```
// Im Falle von Fehlschlag
GLint logSize = 0;
glGetShaderiv(shaderID, GL_INFO_LOG_LENGTH, &logSize);
```

2. a) Ermittle die Größe des Logs um ein entsprechendes Array zu erstellen, welches wir im nächsten Schritt mit dem Logfile füllen wollen.

3.
```
std::vector<GLchar> errorLog(logSize);
glGetShaderInfoLog(shaderID, logSize, &logSize, &errorLog[0]);
```

   a) In diesem Fall wurde ein Vector mit "char"s erstellt, in welches wir die Log Informationen schreiben.
   b) der 2. und 4. Parameter ist wichtig für die typische Vorgehensweise, wie man statische Arrays in C übergibt. (Arraygröße und Speicheradresse)
   c) der 3. Parameter erhält innerhalb der Funktion die Größe des Strings ohne Null-Terminator

4. Nun stehen im Vector die Logfile informationen, welche man auslesen kann.

Der Befehl

```
glGetShaderiv(shaderID, Attribut, outParameter)
```

leitet ein, dass wir einen Wert aus dem Shader auslesen wollen. Der erste Parameter ist der Shader von dem wir etwas wissen wollen, der zweite Parameter ist “Was” wir wissen wollen, diese Information wird in den 3. Parameter geschrieben. Möglich sind folgende Abfragen:

- GL_SHADER_TYPE -> gibt die Art des Shaders (also z.B. GL_VERTEX_SHADER) zurück
- GL_DELETE_STATUS -> gibt zurück ob dieser Shader sich im “Löschen” Status befindet.
- GL_COMPILE_STATUS -> Liefert GL_TRUE wenn die Kompilierung des Shaders erfolgreich war. (GL_FALSE wenn eben nicht)
- GL_INFO_LOG_LENGTH -> Liefert die Anzahl Zeichen des Logs (ein char array mit “\0” Terminator)
- GL_SHADER_SOURCE_LENGTH -> glaube ich selbst erklärend :)

Die gleiche Vorgehensweise passiert auch für Shaderprogramme, mit dem Unterschied, dass wir anstatt glGetShaderiv -> glGetProgramiv nutzen. Programme haben allerdings noch andere und weit mehr Abfragewerte, hier der Link: glGetProgramiv(...)

6.6. Die OpenGL State Machine

Der Begriff “OpenGL State Machine” ist nun schon an der einen oder anderen Stelle gefallen. Gemeint ist damit, dass das Rendern eines Bildes immer auf Basis des aktuellen Status von OpenGL ausgeführt wird.

Im “Intermediate Mode” sieht man das sehr gut wie diese State Machine bearbeitet wird. Alle Elemente ob Shader, Matrizen, Buffertypen, Farb- und Lichtinformationen usw. sind innerhalb eines OpenGL States nur einmal vertreten bzw. Aktiv. Das bedeutet innerhalb eines States ist nur 1 Shaderprogramm, eine Projektionsmatrix, ein Buffer vom Typ X. Demnach gilt für alle Elemente, dass man sie bei Bedarf ein- und ausschaltet.

Ein Renderbarer Zustand funktioniert so:

- Definieren der Vertexdaten
- Shaderprogramm erzeugen
- Shaderprogramm aktivieren
- Vertexdaten mit den Shaderattributen verbinden
- Vertexdaten Aktivieren
- Zeichnen
- Vertexdaten deaktivieren
- Shaderprogramm deaktivieren

7. Vertexdaten

7.1. Buffer

Alle Informationen, welche wir brauchen um etwas auf unseren Bildschirm zu zaubern, wird in OpenGL in sogennanten Buffern beschrieben. Ein Buffer ist im Endeffekt nichts weiter als ein Array. Vertexattribute eines Objektes, wie zum Beispiel der Position der einzelnen Eckpunkte eines Objektes, werden in so einen Buffer geschrieben und in den Shadern später verarbeitet.

In diesem Kapitel werden wir uns damit befassen, wie Buffer erzeugt, verwendet und wieder frei gegeben werden. Zudem werden wir auch auf die “Best Practices” eingehen, welche man mit Buffern realisiert.

7.1.1. Erzeugen und Füllen von Buffer Objekten

Wie auch schon bei Shadern kennengelernt, verwenden wir auch hier keine direkten Objekte einer Klasse, sondern arbeiten mit Handles oder Identifiern, welche quasi ein Objekt suggerieren.

So einen Handle erzeugen wir folgendermaßen:

```
1 GLuint vbo; // vbo = VertexBufferObject
2 glGenBuffers(1, &vbo);
```

Damit reservieren wir uns sozusagen ein Objekt, welches wir mit Daten füllen und auf der GPU ablegen können. Mit dieser Funktion können auch direkt mehrere Buffer reserviert werden. Über den 1. Parameter wird die Anzahl der gewünschten Buffer angegeben und der 2. Parameter enthält ein Array von GLuints, welches ebenso viele Elemente enthält wie die Anzahl Buffer angibt. Der eigentliche GPU Speicher ist damit allerdings noch nicht belegt. Diese ID definiert zunächst nur einen “Platzhalter”, was besagt, dass OpenGLs Memory Management mitteilt: “Da kommt demnächst noch Speicherbedarf”.

Um tatsächlich das Buffer Objekt und den damit verbundenen Speicher zu erzeugen, muss der Handle gebunden werden und sobald das das erste Mal geschiet wird der Speicher addressiert.

```
1 glBindBuffers(BufferTyp, BufferHandle);
```

Der 2. Parameter entspricht dem Handle oder der ID, welches wir mit glGenBuffers reserviert haben (also die Variable ’vbo’).

Der 1. Parameter bestimmt den Buffertypen, es gibt eine Reihe von Buffertypen, die erzeugt werden können. (siehe Referenz: Buffer Typen) Am Anfang werden wir uns aber fast nur mit dem GL_ARRAY_BUFFER beschäftigen, welchen wir mit Vertex Informationen füllen.

Zuletzt müssen eben nun die gewünschten Daten “gebuffert” werden, dieser Schritt ist dafür verantwortlich, die Daten auf der GPU abzulegen. Was uns befähigt, den CPU Speicher zu bereinigen.

```
glBufferData(BufferTyp, ArrayGröße, Array, SpeicherModus);
```

Auch hier definieren wir zuerst die Art des Buffers, welches wir mit Daten beschreiben wollen.

Danach übergeben wir die Größe des BufferArray in BYTES! Und anschließend den Pointer auf das Array selbst.

Zuletzt teilen wir OpenGL noch mit wie "Variabel" diese Daten sind. Also wie oft verändern sich diese Daten.

7.1.2. Buffer Typen

Der Typ eines Buffers bestimmt wie OpenGL intern damit umgeht. Das erste Fragment definiert dabei die Behandlung des Speicherbereichs.

- Static - Wenn sich die Daten des Buffers nie oder nur selten ändern und oft genutzt werden, befähigt man OpenGL damit die Daten besser im Speicher anzulegen um optimalere Speicherbereichs Auslastung zu erreichen. Diese Optimierung ist sehr aufwendig und ist von daher eben nur für Statische Daten geeignet.
- Dynamic - Bezeichnet einen Buffer, wenn sich dessen Daten öfters ändern und sehr oft genutzt werden(Beispiel: Modellierung).
- Das Stream Fragment ist auch für Daten gedacht, die sich selten ändern aber auch nur selten genutzt werden.

Die Wahl der Methode gestalltet sich manchmal schwierig, jedoch führt es nicht zu einem Fehler, wenn man eine ungünstige Methode wählt. Die Wahl der Methode wird am Besten über Performance Checks geprüft.

Das Zweite Fragment bestimmt welches Art von Operation für diesen Buffer gedacht sind.

- Draw - Bufferdaten werden auf Anwendungsebene modifiziert und für allgemeine “Drawing” Operationen verwendet.
- Read - Bufferdaten werden von OpenGL modifiert und sind dazu gedacht von der Anwendungsebene abgefragt zu werden.
- Copy - Bufferdaten werden von OpenGL modifiert und für allgemeine “Drawing” Operationen verwendet.

Figure 8: Buffer Behandlung

Table 3.3 Buffer Usage Tokens

Token Fragment	Meaning
STATIC	The data store contents will be modified once and used many times.
DYNAMIC	The data store contents will be modified repeatedly and used many times.
STREAM	The data store contents will be modified once and used at most a few times.
_DRAW	The data store contents are modified by the application and used as the source for OpenGL drawing and image specification commands.
_READ	The data store contents are modified by reading data from OpenGL and used to return that data when queried by the application.
_COPY	The data store contents are modified by reading data from OpenGL and used as the source for OpenGL drawing and image specification commands.

7.1.3. Erweitertes Buffering

SubBuffer

Was für Daten kann man in einem Buffer ablegen? Grundsätzlich kann man in einem einzigen Buffer Objekt alle VertexDaten als Bytestream ablegen und nutzen. Zusätzlich kann auch der 3. Parameter während des Bufferings offen gehalten werden:

```
glBufferData(BufferTyp, GrößeVertexDaten, NULL, Speichermodus);
```

Was wir wissen müssen ist die Gesamtgröße der Daten in Bytes, jedoch die einzelnen Werte (z.B. die Position der einzelnen Punkte müssen noch nicht bekannt sein). Man hat nachträglich die Möglichkeit die Daten einzeln hinzu zufügen.

Um dann die Daten tatsächlich dem Buffer hinzu zufügen kann man dies einzeln über SubBuffer machen:

```
glBufferSubData(BufferTyp, Offset, Größe, Daten);
```

Siehe dazu ein Beispiel:

```
// Vertex Positionen
const GLfloat positions[] =
{
-1.0f, -1.0f, 0.0f, 1.0f,
1.0f, -1.0f, 0.0f, 1.0f,
1.0f, 1.0f, 0.0f, 1.0f,
-1.0f, 1.0f, 0.0f, 1.0f
};

// Vertex Farben
const GLfloat colors[] =
{
1.0f, 0.0f, 0.0f, 0.0f,
1.0f, 0.0f, 0.0f, 0.0f,
1.0f, 1.0f, 1.0f, 1.0f
};

// Der Hauptbuffer
GLuint vbo;
glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, buffer);

// Alloziere den Grund Speicherbedarf des Buffers
glBufferData(
      GL_ARRAY_BUFFER, // Buffertyp
       sizeof(positions) + sizeof(colors), // Seicherbedarf
```

```
27        NULL, // keine Daten übergeben
28        GL_STATIC_DRAW); // Speicherverwaltungs Hinweis
29
30 // Füge das "positions" Array dem Hauptbuffer hinzu,
31 // Offset = 0 heißt am Anfang des Buffer Arrays
32 glBufferSubData(
33          GL_ARRAY_BUFFER, // Buffertyp
34          0, // Offset
35          sizeof(positions), // Bytegröße
36          positions); // Daten
37
38 // Füge das "colors" Array hinter "positions" dem Hauptbuffer hinzu
39 glBufferSubData(
40        GL_ARRAY_BUFFER,
41        sizeof(positions), // Offset
42        sizeof(colors), // Größe
43        colors);
```

Löschen von Bufferdaten

Im Falle des Bedarf, dass die gebufferten Daten auf der GPU gelöscht werden sollen, kann dies über folgende Methode gemacht werden.

```
1 glClearBufferData();
2 glClearBufferSubData();
```

Der OpenGL Addresspointer

Alle Funktionen die mit dem Buffern von Daten zu tun haben, wie z.B. glBufferData, machen im Endeffekt nichts anderes als das sie die übergebenen Daten kopieren und dann innerhalb von OpenGL als verwalteter Speicher vorhanden sind. Der Speicher, welcher innerhalb unserer Applikation entsteht, u.a. durch "const GLfloat positions[] = {...} ist zunächst temporärer Speicher auf unserer CPU. Mit glBufferData wird dieser Speicher in OpenGL kopiert, welcher dann auf der GPU gelagert wird. Es stellt sich die Frage wie man performant die Buffer Daten verändern kann. Wir wollen möglichst wenig Speicherzugriffe und auch das "Swappen" von CPU zu GPU klingt nicht gerade optimal. Es wäre daher gut einen Pointer auf den OpenGL verwalteten Addressbereich (aka GPU Memory) zu bekommen. Dieses ermöglicht uns folgende Funktion:

```
1 void* glMapBuffer(GLenum target, GLenum access);
```

WICHTIG: Man übersieht leicht, dass hier ein Rückgabetyp angegeben ist, welcher einen void-Pointer darstellt. Die Eigenschaft des void Pointers ist im Prinzip, dass er alles und nichts darstellt. Da Pointer wie Arrays behandelt werden können, haben wir durch ihn Zugriff auf den gesamten, zur Zeit aktiven, Buffer-Speicher.

Neben dem Buffertyp (target), wird mit OpenGL eine Art Vertrag geschlossen, wie eine Anwendung mit dem Pointer umgehen will.

Figure 9: Map Buffer Acces Modes

Table 3.4 Access Modes for glMapBuffer()

Token	Meaning
GL_READ_ONLY	The application will only read from the memory mapped by OpenGL.
GL_WRITE_ONLY	The application will only write to the memory mapped by OpenGL.
GL_READ_WRITE	The application may read from or write to the memory mapped by OpenGL.

Die Verwendung eines MapBuffer Pointers wird uns wieder begegenen, wenn wir uns mit dem Rendering, mit Hilfe eines Renderers, beschäftigen.

7.2. Vertexarrays

7.2.1. Einführung

Mit Hilfe von Buffern sind wir also in der Lage unsere Vertexdaten abzulegen und so zum Zeichnen verfügbar zu machen. Diese Buffer haben allerdings noch keinerlei Bindung zum Shader gehabt, deshalb haben wir innerhalb der Vorlesung folgende Zeilen genutzt um unsere Vertexdaten an den Shader zu binden:

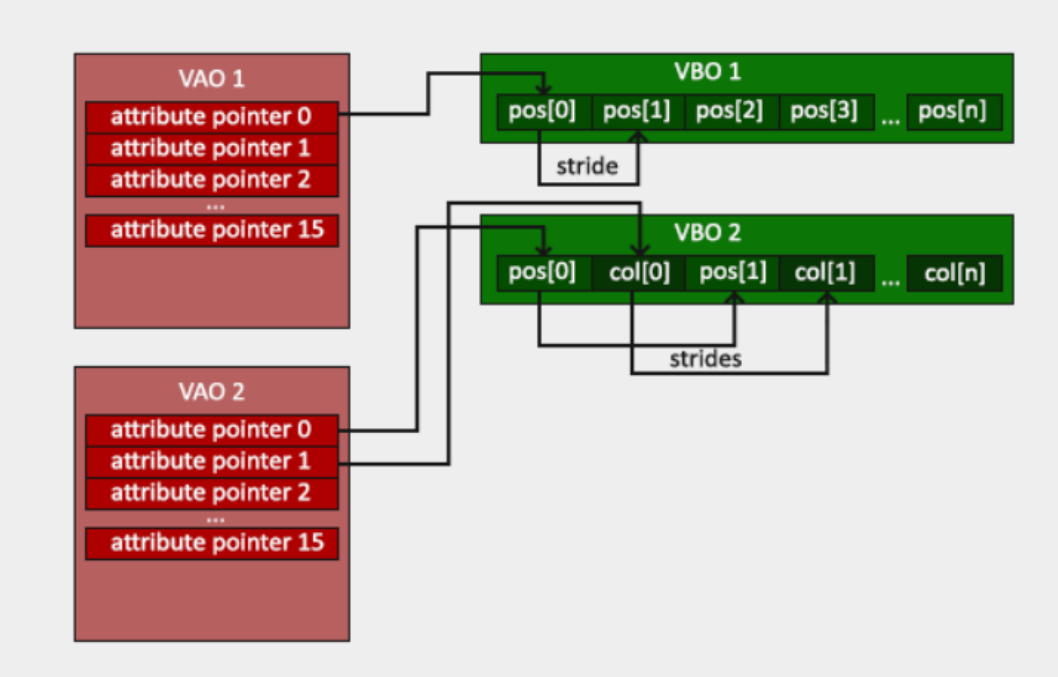
```
1  glEnableVertexAttribArray(0);
2
3  glVertexAttribPointer(
4      0,
5      3,
6      GL_FLOAT,
7      GL_FALSE,
8      0,
9      0
10     );
```

Die beiden obigen Befehle, verknüpfen Shader Attribute mit etwas, das sich Vertex Array (Objekt) nennt. Diese Art Objekt haben wir bisher noch nicht kennen gelernt und nur implizit genutzt. Der Grund, warum wir bisher überhaupt etwas zeichnen konnten ist, das wir, ohne es zu wissen, ein Vertexarray bekommen haben, welches Aktiv ist und unsere Buffer mit den Vertexshader verbunden hat. Testen kann man dieses, wenn man einfach einmal mehrere Objekte erzeugt und versucht alle zu malen. Der Effekt wird höchst wahrscheinlich der sein, dass nur das zuletzt erzeugte Objekt auf dem Bildschirm gemalt wird.

Wir benötigen eine Art Map, welche alle Buffer eines Objektes mit einem Shader verbindet. Dieses Objekt heißt VertexArrayObjekt (kurz VAO) und wird, gemäß der OpenGL State Machine, Aktiv bevor gezeichnet wird.

7.2.2. Übersicht der VAO Map

Figure 10: VAOs



7.2.3. Erzeugen von VAOs

Um für ein oder mehrere Objekte ein VAO zu erzeugen, bedarf es nur weniger Befehle.
Genau wie für einen Buffer nutzen wir auch hier die OpenGL Einleitung glGen...

```
GLuint vao;
glGenVertexArrays(1, &vao);
```

Damit erhalten wir wieder einen Identifier auf ein Vertex-Array-Objekt. Und mit dem folgenden Befehl aktivieren bzw. deaktivieren wir so ein Objekt:

```
glBindVertexArray(vao); // Aktivieren
glBindVertexArray(0); // Deaktivieren
```

Nun haben wir explizit ein VAO erzeugt und gebunden. Alle Shader Attribute und Buffer, welche erzeugt und gebunden werden, werden automatisch in dem derzeit aktiven VAO gelinkt.
Alle Shader Attribute sind standardmäßig “disabled”. Deshalb müssen wir sie aktivieren bevor wir Objekte mit Hilfe eines Shaders “zeichnen”.

```
glEnableVertexAttribArray(attribID);
```

Für unser VAO wird das Shader Attribut (attribID) aktiv. Und somit können wir zu diesem Attribut einen Buffer binden.

```
glVertexAttribPointer(attribID, componentSize, compDataType,
    Normalized?, offsetStride, bufferIndexOffset);
```

Die ersten 4 Parameter sind relativ einfach:

- attribID -> = Shader Location Attribut
- componentSize -> Anzahl Element pro Komponente (Vector3 = 3 * sizeof(Datatype))
- compDataType -> Datentyp der Komponenten (bsp.: GL_FLOAT)
- Normalized? -> fragt ob die Werte auf einen Bereich zwischen 0-1 gemappt werden sollten, was in der Regel immer “false” (GL_FALSE) ist.

Die nächsten beiden Parameter sind etwas komplexer. Da ein Objekt und dessen Vertizes aus mehr als nur Positionen bestehen, sondern zumindest i.d.R. auch noch eine Farbe haben, müssen wir OpenGL auch das Layout dieser Daten in einem Buffer darstellen.

- offsetStride -> definiert die Größe in Bytes, welche eine Komponente definiert. D.h. Die Bytegröße des Arrays / Offset = Anzahl Komponenten
- bufferIndexOffset -> sofern innerhalb einer Komponente mehrere Attribute existieren, müssen wir für jedes Attribut definieren, an welchem Bit im Bytestream es beginnt.

Best Practices?

Buffer und VAOs sowie das Verbinden mit Shader Attribute ist so vielfältig und variabel, dass man schnell nicht weiß, wie man sein Layout definiert.

So hat man Beispielsweise die Möglichkeit, anstatt sich mit Offsets etc. befassen zu müssen, für jedes Attribut einen Buffer zu erzeugen und diesen entsprechend im VAO zu binden. Und das für jedes “Drawable” Objekt.

```
GLuint vao;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);

// Bufferclass - implizit glGen,glBind,glBufferData
Buffer* positionBuffer;

glEnableVertexAttribArray(0);
glVertexAttribPointer(
  0,
  positionBuffer->GetComponentSize(), // Vector3 liefert 3*sizeof(
    GLfloat)
  GL_FLOAT,
  GL_FALSE,
  0,
  0);

Buffer* colorBuffer;

glEnableVertexAttribArray(1);
glVertexAttribPointer(
  1,
  colorBuffer>GetComponentSize(), // Vector4 liefert 4*sizeof(
    GLfloat)
  GL_FLOAT,
  GL_FALSE,
  0,
  0);
```

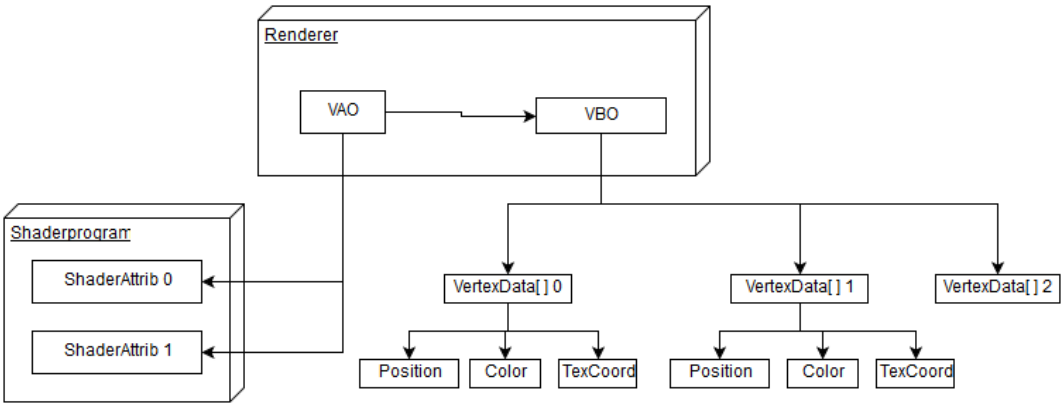
Das funktioniert einwandfrei, jedoch sollten wir immer in Betracht ziehen, dass das Binden und Verbinden Zeit kostet und Performanceeinbußen mit sich bringt.

Anstatt auf die Idee zu kommen, jedem Objekt sein eigenes VAO mit eigenen Buffern zu geben und es während der “Draw” Methode zu binden, zu zeichnen, wieder zu entbinden, was zu besagten Einbußen führt, wenn man sich vorstellt, dass man vielleicht mehrere tausend Objekte zugleich zeichnen will. So könnte man auf den Gedanken kommen die Anzahl der VAOs und VBOs zu reduzieren um weniger binden zu müssen.

Was wäre also, wenn man sagt, es gibt nur ein VAO und ein VBO für Objekte, bzw. pro Objektgruppe? Wir müssten nur einmal binden und malen alle Objekte des Buffer quasi “instantly”.

Folgendes Diagramm soll das etwas verdeutlichen:

Figure 11: Single VAO



7.3. Index Buffer

7.3.1. Noch ein Buffer?

Eine Sonderform eines Buffertypen sind die sogenannten IndexBufferObjects (IBOs). Die Idee dahinter ist rein performanter und Speicher schonender Natur.

So wie wir bisher "gezeichnet" haben hat vorrausgesetzt, dass wir jeden einzelnen Vertex eines Objekts in einen Buffer gespeichert haben. Wenn wir also mit Hilfe von Dreiecken (GL_TRIANGLES) ein Rechteck zeichnen wollten brauchten wir dafür 2 Dreiecke mit je 3 Punkten.

Siehe Beispiel:

```
const GLfloat positions[] =
{
// 1. Dreieck
-1.0f, -1.0f, 0.0f, // 1. Punkt
1.0f, -1.0f, 0.0f,  // 2. Punkt
1.0f, 1.0f, 0.0f, // 3. Punkt
// 2. Dreieck
1.0f, 1.0f, 0.0f, // 4. Punkt
-1.0f, 1.0f, 0.0f,  // 5. Punkt
-1.0f, -1.0f, 0.0f  // 6. Punkt
};
```

Mit Hilfe von (GL_QUADS) könnten wir der sichtbaren Redundanz von Punkten Herr werden. Jedoch wollen wir eine Alternative Methode verwenden, die Überall Anwendbar ist und zudem noch Effektiver als die Veränderung des Drawmodes ist. Denn wo uns GL_QUADS hier hilft die Redundanz der Punkte 1 und 6 sowie 3 und 4 zu eliminieren, so entsteht das Problem erneut wenn wir statt eines 2D Dreiecks einen 3D Würfel zeichnen wollen. Ein Würfel ist durch 8 Punkte definiert und auch mit GL_QUADS für alle 6 Flächen des Würfels hätten wir immer redundante Punkte.

7.3.2. Erzeugen und nutzen eines Indexbuffers

Die Erstellung ist auch hier wieder wie üblich. Zunächst erzeugt man sich einen neuen Buffer mit dem Befehl:

```
GLuint ibo;
glGenBuffers(1, &ibo);
```

Allerdings Buffern wir nicht die Vertexdaten an sich sondern die Reihenfolge wie die einzelnen Vertices gemalt werden (Indexierung).

Diese Reihenfolge bestimmen wir ebenfalls in einem Array:

```
GLuint indices[] =
{
  0,1,2,0,2,3
};
```

Unter Berücksichtigung das wir "Counterclockwise" zeichnen bilden wir die 6 Punkt-Indices in entsprechender Reihenfolge ab. Die Reihenfolge kann natürlich auch anders geschrieben werden, beispielsweise würde 0,1,2,2,3,0 auch beide Dreiecke entgegen des Uhrzeigersinns definieren.

Mit diesen Indices können wir unser "vertices" Array verkleinern:

```
const GLfloat positions[] =
{
-1.0f, -1.0f, 0.0f, // 1. Punkt
1.0f, -1.0f, 0.0f,  // 2. Punkt
1.0f, 1.0f, 0.0f, // 3. Punkt
-1.0f, 1.0f, 0.0f,  // 4. Punkt
};
```

Nun müssen wir unseren Buffer mit den Indices füllen. Für diesen Zweck nutzen wir einen neuen Buffertypen:

```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo);
```

Die Idee ist, dass wenn gezeichnet wird, 2 Arten von Buffer gebunden werden.

Im VAO binden wir die Vertexdaten selber vom Typ GL_ARRAY_BUFFER. Und binden anschließend die Indexierung dieser Daten in einem Element-Array. Sodass OpenGL anhand der Indices die richtigen Punkte aus dem VertexArray herausnimmt und zeichnet.

Nach dem Binden erfolgt noch das Buffern des Element Buffers:

```
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices,
    GL_STATIC_DRAW);
```

Zuletzt wird die Draw Methode verändert, sodass das Element Buffer zum Zeichnen genutzt werden soll:

```
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0); // 6 = Anzahl
    Indizes
```

7.3.3. Zusammenfassung Beispielcode

```
class Rectangle
{
  VAO* vao;
  IBO* ibo;
  void Draw();
}

Rectangle::Rectangle()
{
  const GLfloat positions[] ={
    -1.0f, -1.0f, 0.0f, // 1. Punkt
    1.0f, -1.0f, 0.0f,  // 2. Punkt
    1.0f, 1.0f, 0.0f, // 3. Punkt
    -1.0f, 1.0f, 0.0f,  // 4. Punkt
  };

  GLuint indices[] ={
    0,1,2,0,2,3
  };

  vao = new VAO();       // Erzeugt vao
  vao->AddBuffer(positions); // Bindet vao und ruft bufferData(
   positions) auf

  // Generiert IndexBuffer und buffert das indices Array
  ibo = new IBO(indices);
}

Rectangle::Draw()
{
  vao->Enable();
  ibo->Enable();
  glDrawElements(
    GL_TRIANGLES,
    ibo->GetNumberOfIndices(),
    GL_UNSIGNED_INT,
    0
    );
  ibo->Disable();
  vao->Disable();
}
```

Part IV.
Das Auge des Betrachters

8. Einleitung

In diesem Kapitel befassen wir uns mit Thema eine virtuelle 3D-Szene auf unserem Bildschirm darzustellen. Im Einzelnen werden wir uns mit den verschiedenen Matrizen beschäftigen, die uns dieses ermöglichen. Dazu gehören die World-Matrix, die Modelview-Matrix, die Projektions-Matrix und eine Matrix die man als "Kamera" beschreiben könnte.

Weiterhin werden wir uns auch mit den Themen der Transformation von Vertex-Daten beschäftigen. Sprich wie bewegt, rotiert und skaliert man Objekte im Raum.

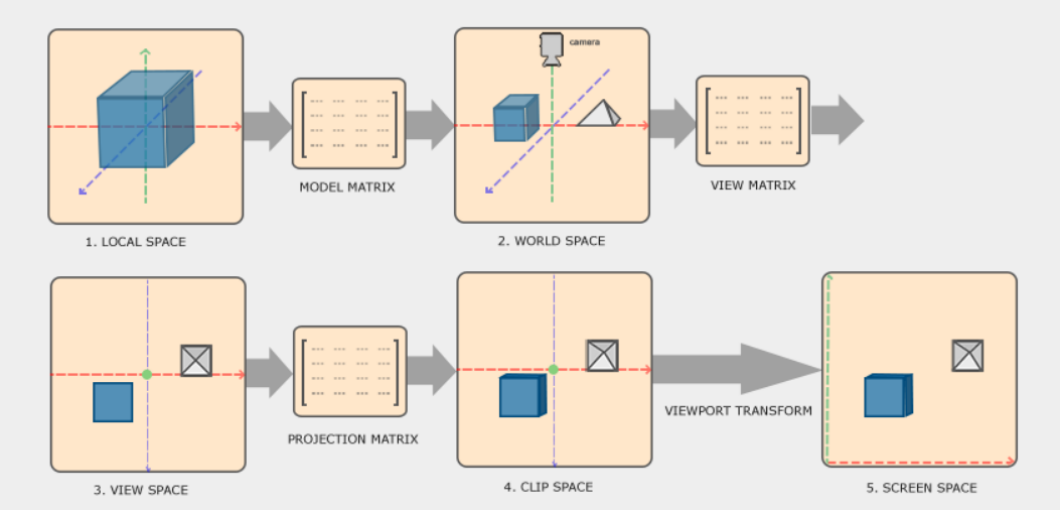
9. Das "Viewing"

9.1. Einführung

Gemäß dem Fall, das wir ein Objekt auf unserem Bildschirm darstellen wollen, bedarf es einiges an Vorbereitung. Wir können nicht einfach ein Objekt mit z.B. metrischen Daten in das Programm senden und hoffen das es so wie es beschrieben ist dargestellt wird. Wir müssen zum Einen die metrischen Daten in Pixel umrechnen und zusätzlich wollen wir natürlich auch in der Lage sein das Objekt aus verschieden Blickwinkeln und an verschiedenen Positionen zu betrachten. Und im Endeffekt müssen wir uns um die Projektion von 3D auf unseren flachen 2D Monitor kümmern.

9.2. Übersicht der Projektions-Pipeline

Figure 12: Projektions-Pipeline



Man beginnt zunächst das Objekt im “Local Space” zu beschreiben. Jedes Objekt ist durch seine eigene “Modelmatrix” definiert bzw. kann man auch sagen, dass es ein eigenes Koordinatensystem (KOS) hat. Nebenbei gemerkt ist in diesem KOS der Punkt (0,0,0) auch der Pivot-Point eines Objektes um den rotiert wird, wenn man eben rotieren möchte.

Die Koordinaten aller Models sind relativ zum “World Space”. Das heißt alle Koordinaten aller Models werden von dessen Ursprung (0,0,0) transformiert. Die sogenannte “Viewmatrix” definiert die Sicht auf den “World Space” mit seinen Models bzw. die Koordinaten der Models im “World Space” werden in den “View Space” transformiert.

Aus dem “View Space” heraus wird nun der “Clip Space” in Form der “Projektionmatrix” erstellt. Diese Matrix rechnet die Koordinaten in Clip Koordinaten im Bereich -1, 1 um. Dadurch kann bestimmt werden was im Bereich des “Sichtbaren” ist und was “geclipt” werden muss.

Im letzten Schritt werden die Clip Koordinaten in Screen Coordinaten des “Screen Space” gemappt. Hintergrund ist eine Viewport Transformation, welche die -1,1 Koordinaten auf die Viewport Größe (z.b. 800x600) umrechnet.

9.3. Projektionen

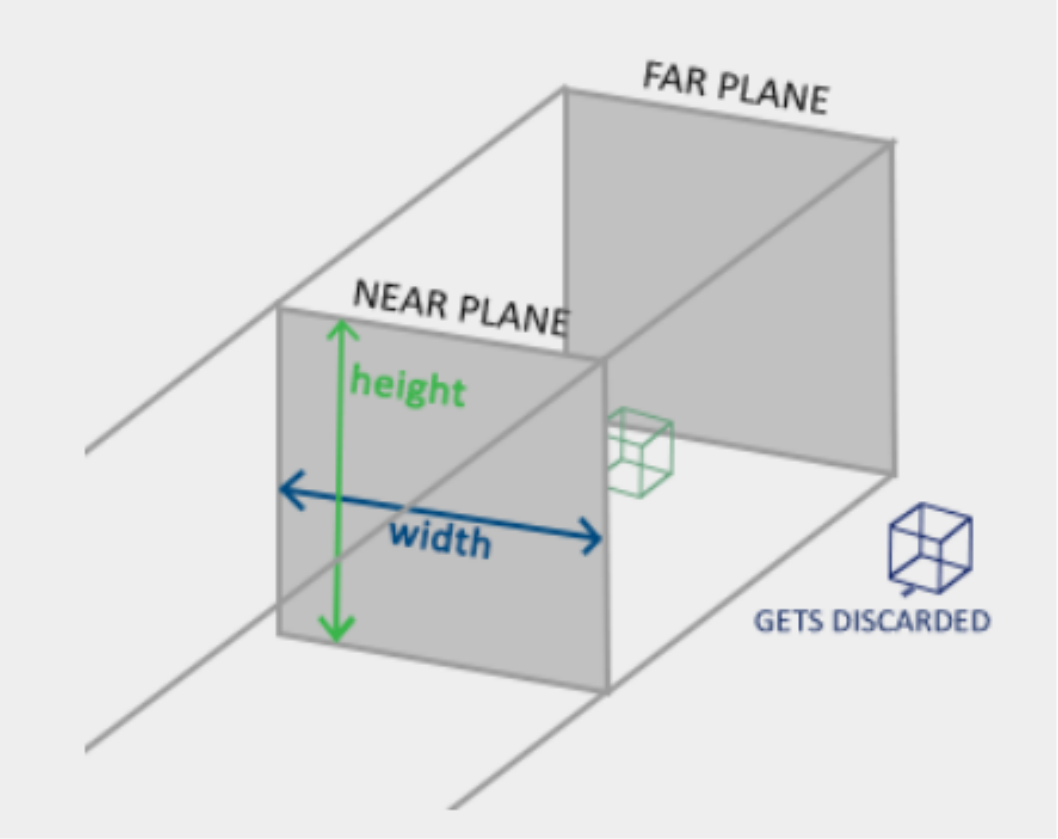
9.3.1. Einleitung

Anhand des vorherigen Kapitels haben wir einen Eindruck bekommen, welche Schritte die Koordinaten eines Objektes durchlaufen bis Sie auf dem Bildschirm landen. Dieser Abschnitt befasst sich mit den möglichen Projektion, wie die “Sichtweise” auf ein Objekt definiert werden kann.

9.3.2. Orthografische Sichtweise

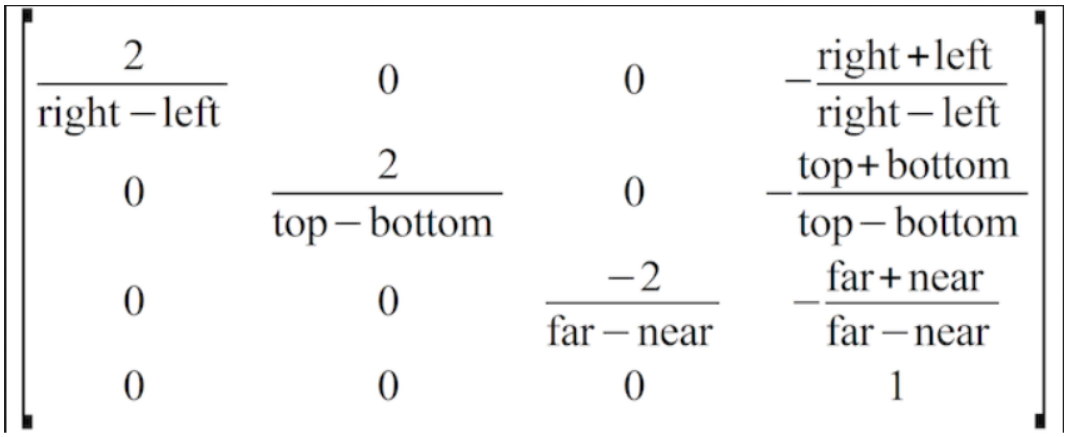
Die Form der Sichtweise mapt die Koordinaten eines Objektes direkt auf den 2D-Bildschirm ohne dabei Dinge wie Perspektive bzw. Tiefe zu simulieren. Stellt man sich einen Würfel, bei dem wir orthogonal auf eine Seite schauen, so würden wir, egal wo sich der Würfel auf dem Bildschirm befindet, immer nur diese Seite sehen. Alle Objekte werden immer in ihrer Originalgröße gezeichnet werden.

Figure 13: Orthografische Projekion



Das Clipping Volume (Frustum) sieht wie ein Würfel aus. Die Definition einer orthografischen Projektion wird über die Seiten dieses Würfels beschrieben (Left, Right, Bottom, Top, Far, Near) beschrieben. Die zugehörige Matrixbeschreibung sieht dann so aus:

Figure 14: Orthografische Matrix



Die Anwendung von orthografischen Projektionen ist für den Betrachter eher unrealistisch, beispielsweise würd keine 3D-Tiefe simuliert, sodass man immer quasi “platt” gedrückte Objekte sieht. Dieser Effekt kann in einigen Fällen aber auch praktisch sein. Üblicherweise nutzt man diese tatsächlich für Draufsichten (Grundrisse etc.). Auch kann man “Displacements” von Vertices leichter erkennen als in einer perspektivischen Sicht.

Was bedeuten diese Werte? Betrachten wir zunächst die Diagonale. Man könnte erkennen, dass die Berechnung irgendetwas mit den Achsen des Koordinaten Systems zu tun haben und es anscheinend eine 4. Dimension (=1) gibt (dazu gleich mehr). Tatsächlich definiert die Matrix den Mittelpunkt des Viewports auf -1,1 gemappt. Hätten wir also ein Frustum mit den Grenzen:

- Links = 0
- Rechts = 10
- Bottom = 0
- Top = 10
- Near = 0.1
- Far = 10

Hinweis: Die Near Komponente darf nicht 0 sein, eine typische Belegung ist 0.1.

Programmatische Anwendung z.B. mit Hilfe der GLM Bilbiothek:

```
glm::mat4 orthographicMatrix = glm::ortho(0,10,0,10,0.1,10);
```

Würde das bedeuten das jedes Objekt mit Koordinaten zwischen 0-10 auf allen Achsen(X,Y,Z) sichtbar sein wird. (Wenn man nur Modelmatrix und Projektionmatrix betrachtet). Da wir den Clipping Space im Bereich -1,1 mappen, ergibt sich durch die Berechnung oben, Faktoren die mit allen Vertices multipliziert werden und eben dann im Bereich -1,1 liegen. Beispiel der X Achse. Nehmen wir an wir wollen einen Punkt mit dem X-Wert = 5 zeichnen. D.h. er wird genau in der Mitte des Bildschirms auf der X-Achse erwartet (bei 0).

Wir rechnen 2 / 10 - 0 = 0.2 multipliziert mit dem Vertex.X = 5 * 0.2 = 1... naja fast 0 oder? Hier kommt die Translations Komponente (4. Spalte) ins Spiel, welche die Schrittbewegung zum Mittelpunkt beschreibt. Also für X erhält man bei 10 + 0 / 10 - 0 = 1, mit -1 multipliziert = -1. Addiert man nun unseren gemappten Vertex mit -1 erhalten wir 0... Tadaa den Mittelpunkt der X-Achse. Warum addieren? Weil, eine Translation nichts weiter als simple Vektorrechnung ist. Wenn wir etwas bewegen wollen, wenden wir einen Vektor auf einen anderen Vektor an und erhalten eine neue Position.

Was ist nun aber mit dieser ominösen 4. Komponente? Die Anwendung eines Vertex auf eine Matrix passiert über die Multiplikation von Matrix und Vektor. Bei einer 4x4 Matrix bedarf es einen Vector4 um die Multiplikation ausführen zu können.

```
vec4 output = Projection * InputVector;
```

Wir wissen das die ersten 3 Komponenten des Vectors die X,Y,Z-Position im kartesischen Koordinatensystem beschreibt. Wenn man die 4. Zeile mit dem Vektor addiert erhält man die w-Komponente des Vektors. Diese Komponente wird von OpenGL dazu verwendet eine perspektivische Position (engl. Begriff = perspective Division) des Vertex zu berechnen, also die Position die den Vertex in seiner Tiefe simuliert. Nachdem die ersten 3 auf den Bereich -1,1 gemappt worden sind, führt OpenGL folgende Berechnung aus:

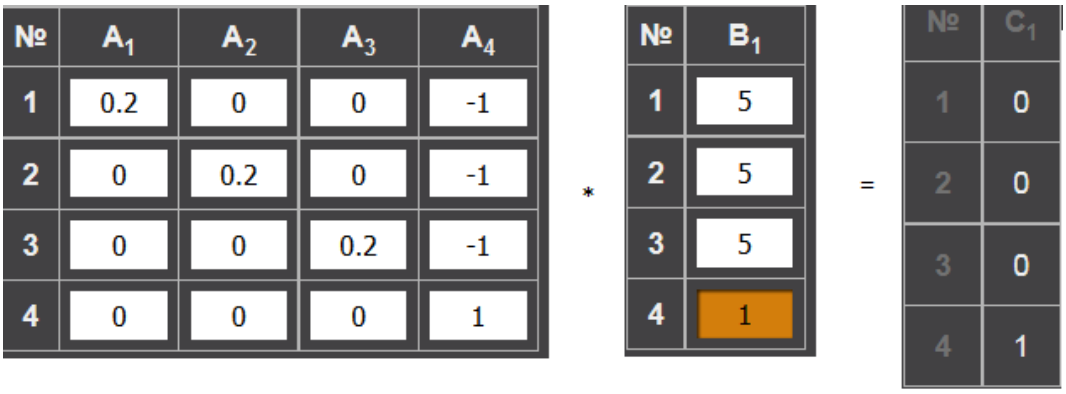
Figure 15: Anwendung W-Komponente

$$out = \begin{pmatrix} x/w \\ y/w \\ z/w \end{pmatrix}$$

Dieser Wert wird ohne unser Wissen von OpenGL dazu verwendet eine perspektivische Skalierung von Vertexdaten zu generieren, diese existiert jedoch nicht innerhalb einer orthografischen Projektion und die Matrix ist so definiert, dass für die 4. Komponente des Output Vectors immer 1 herauskommt und so die anschließende Teilung keinen Einfluss auf das Vertex-Resultat hat.

Beispielrechnung:

Figure 16: Orthografische Berechnung Beispiel



№	A_1	A_2	A_3	A_4
1	0.2	0	0	-1
2	0	0.2	0	-1
3	0	0	0.2	-1
4	0	0	0	1

*

№	B_1
1	5
2	5
3	5
4	1

=

№	C_1
1	0
2	0
3	0
4	1

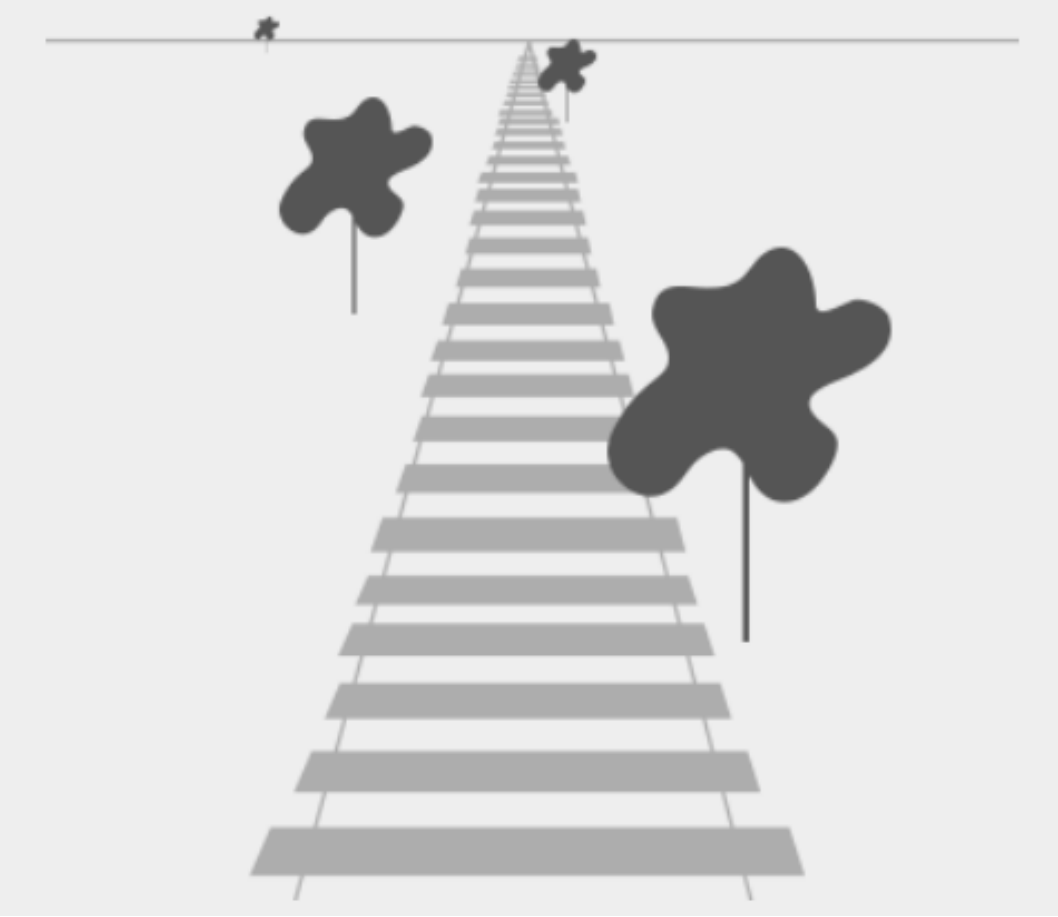
```
glm::ortho(0,10,0,10,0.1, 10)
```

9.3.3. Perspektivische Sichtweise

Neben der eher unrealistischen orthografische Sicht, wollen wir nun realistisch werden und Objekte die weit entfernt sind entsprechend kleiner darstellen als Objekte selber Größe, welche jedoch näher an unserer “Linse” sind.

Um dies zu berechnen kommt hier die w-Komponente zum tragen, welche für jeden Vertex dessen Tiefe mit einkalkuliert. Nachdem der Vertex seine Clipping-Koordinaten (-1,1) erhalten hat, wird die “perspective division” mit Hilfe der w-Komponente vollzogen, dabei gilt je höher w ist, desto weiter entfernt erscheint der Punkt bzw. desto näher wird der Punkt zur Mitte gezeichnet (vgl. Bild unten).

Figure 17: Perspektivische Projektion



Und die dazugehörige Matrix ist folgendermaßen definiert:

Figure 18: Perspektivische Matrizen

$$\begin{bmatrix} \frac{1}{aspect * \tan(\frac{fov}{2})} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\frac{fov}{2})} & 0 & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & -\frac{2*far*near}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Wir haben 2 Möglichkeiten unser Frustum zu beschreiben. Die erste Matrix nimmt folgende Werte entgegen:

Perspektive mit Angabe des FOV (Field of View)

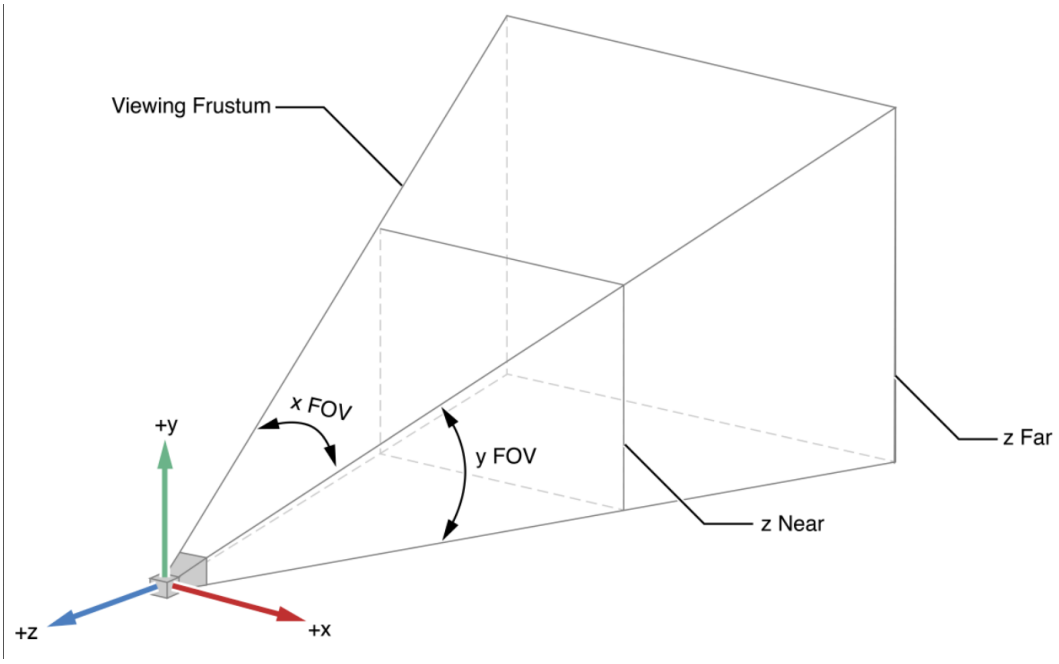
Diese Form bildet immer ein synchrones Frustum um die Z-Achse herum, d.h die Bereiche Mitte-Links, Mitte-Rechts sind gleich Groß, bzw. jeweils eine Hälfte des FOVs.

- Field of View -> Angabe in Grad, welche die linken und rechten Grenzen definiert.
- Seitenverhältnis des Bildschirms/Fensters
- Die Near und Far Clipping Planes

```
glm::mat4 Projection = glm::perspective(35.0f, 1.0f, 0.1f, 100.0f);
```

Das entstehende Frustum (Clipping Volume) sieht so aus:

Figure 19: Synchrones Frustum

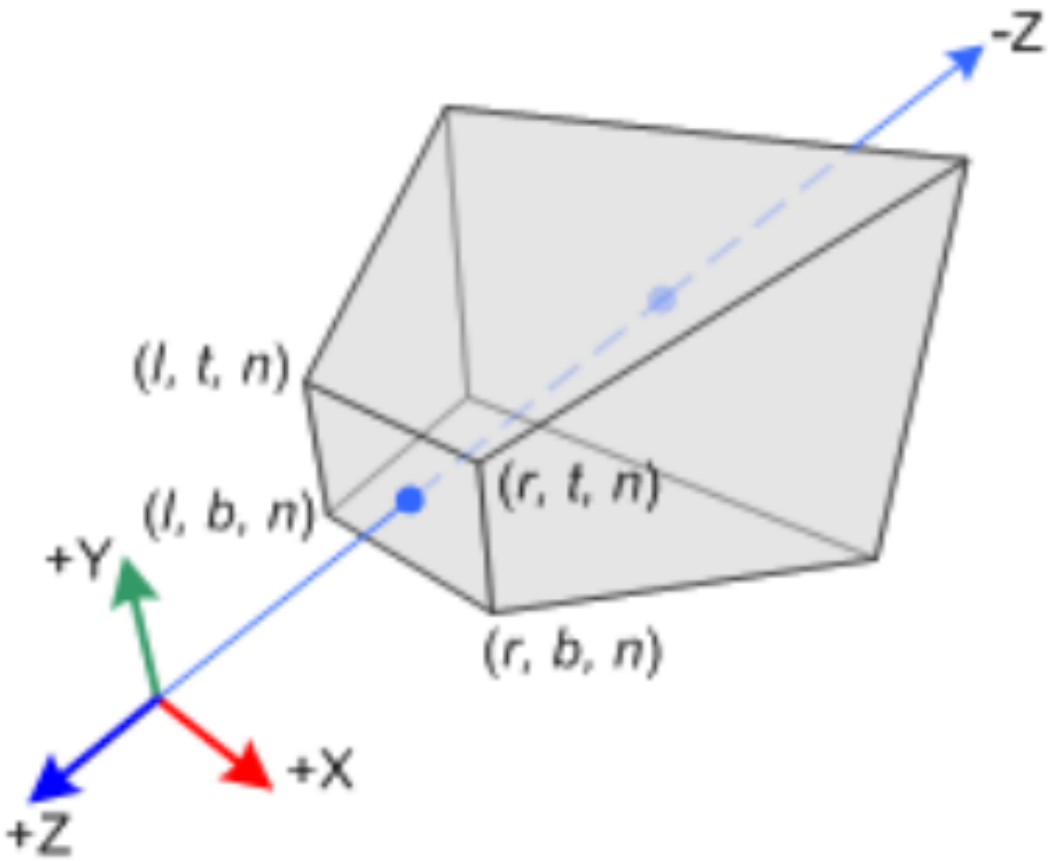


Perspektive mit Hilfe definierter Clipping Planes (left, right, top, bottom, near, far)

Mit dieser Form kann man asynchrone Frustums erstellen, da die Clipping Grenzen Rechts-Links, Oben-Unten, Nah-Fern mit unterschiedlichen Grenzwerten belegt werden können. Ein ungleichmäßiges Sichtfeld entsteht. Man kann sich das so vorstellen, dass man z.B. mit dem Linken Auge einen größeres Blickfeld hat als mit dem Rechten.

```
glm::mat4 Projection = glm::frustum(-10,10,-10,10,1,10);
```

Figure 20: Asynchrones Frustum



Alle Z-Ebenen des Frustums (von Near bis Far) sind auch hier im “Clipping Space” auf -1,1 gemappt. Jedoch erhält man den Effekt, dass zwei Punkte, welche z.B. dieselben X,Y Werte, jedoch unterschiedlich Tiefe (Z) haben, nicht übereinander dargestellt werden, sondern je weiter ein Punkt weg ist, desto weiter würde er auf dem Bildschirm nach innen verschoben werden.

Innerhalb der perspektivischen Matrizen, fällt im Vergleich zur orthogonalen Matrix auf, dass das Layout etwas anders aussieht. Die Mutliplikation der 3. Zeile der Matrix mit einem Vektor, bestimmt dessen Z-Komponente und die 4. Zeile bestimmt die W-Komponente. Aus diesen beiden Komponenten wird der “Offset” errechnet um den Vertex in seiner Tiefe zu simulieren. Dieser Offset hat dan logischerweise Einfluss auf die X,Y Komponenten, da wir bekannterweise auf eine 2D Ebene projezieren müssen.

Beispielrechnung:

Figure 21: Perspektive Kalkulation Beispiel

№	A_1	A_2	A_3	A_4
1	0.1	0	0	0
2	0	0.1	0	0
3	0	0	1.2222	2.2222
4	0	0	-1	0

*

№	B_1
1	-10
2	-10
3	−1
4	1

=

№	C_1
1	-1
2	-1
3	1
4	1

glm::frustum(-10,10,-10,10,1,10)

Während in der orthografischen Berechnung die W-Komponente immer 1 war, und somit keine perspektivische Vertiefung angewendet wurde, kann sich innerhalb dieser Kalkulation ein anderer Wert für W ergeben. Im Rechenbeispiel ist W = 1 herausgekommen, d.h. der Vertex muss nicht mehr weiter skaliert werden und sitzt somit exakt am unterem linken Bildschirmpunkt. Sofern eben ein Wert > 1 entsteht, wird die perspektivische Division angewendet. Dabei wird der gesamte Vertex wie in Abbildung:15 gezeigt mit dem Kehrwert der Z-Komponente multipliziert und ist danach auch komplett normiert. Ein Wert < 1 bedeutet, dass der Vertex außerhalb des Frustums liegt und somit geclipt werden kann.

Beispielrechnung Perspektivische Normierung

Figure 22: Perspektive Kalkulation Beispiel mit W