Ostfalia - Hochschule für angewandte Wissenschaften - Braunschweig/Wolfenbüttel -

WPF - Computergrafik

B.Sc. Fabian Sonczeck

April 7, 2016

Part I.
Einführung in Computergrafik

1. Grundlegender Aufbau der grafischen Datenverarbeitung (GDV)

Figure 1: Aufbau Grafiksystem

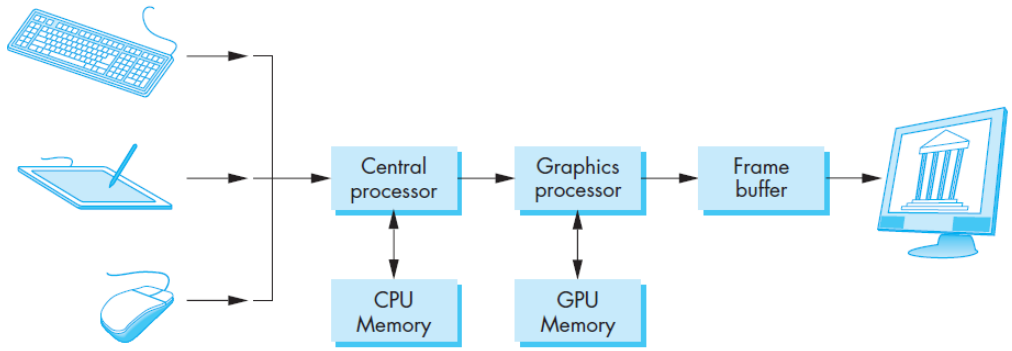


FIGURE 1.1 A graphics system.

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

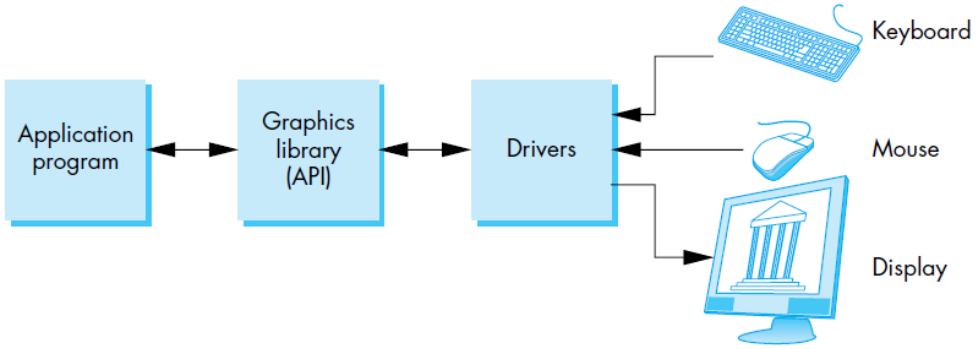


FIGURE 1.28 Application programmer's model of graphics system.

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

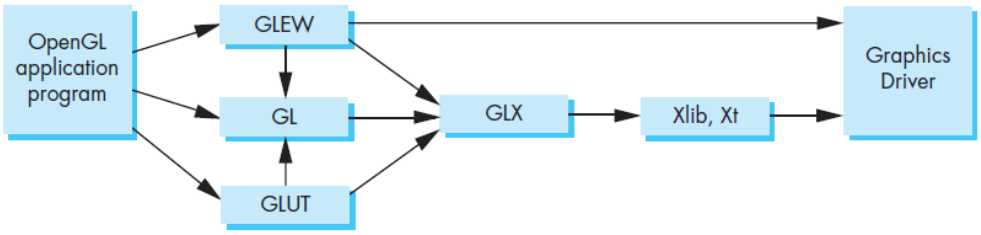


FIGURE 2.4 Library organization.

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 Modes

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.

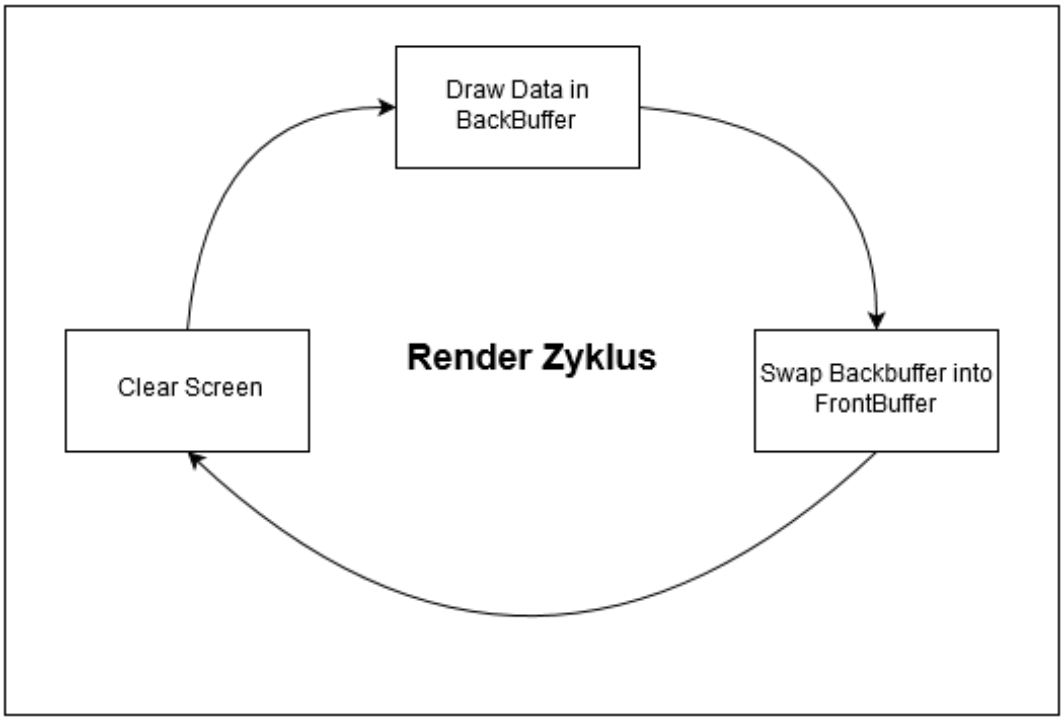
Beispiel Buffered Mode:

```
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.
Let‘s program 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. Wir beginnen zunächst damit unser OpenGL Projekt um die Klasse “Window” zu erweitern.

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.
- Unsere Klasse Window fungiert als “Wrapper”, welche einen OpenGL Context/Window als Member speichert.
- Sie enthält die Update Methode um Events abzufragen und die Framebuffer zu “switchen”
- Sie kann abgefragt werden ob der OpenGL Kontext geschlossen wurde (z.B. der User hat auf den “Schließen(X)”-Button geklickt)
- Sie löscht den Bildschirm bevor etwas neues darauf gezeichnet wird.

Unsere Klasse Window erhält zunächst folgende Member und einen Konstruktor:

```
/* in Window.h */

#include "glew.h" // Für die GL Datentypen (GLint, GLsizei usw.)

#include "glfw3.h" // Für die GLFWwindow struct und Initialisierung

class Window
{
  // Member
private:
  const char* windowTitle;
  GLsizei windowWidth, windowHeight;

  GLFWwindow* windowInstance;

  // Ctor
public:
  Window(const char* inTitle, GLsizei inWidth, GLsizei inHeight);
  ~Window();

private:
  bool InitWindowAndContext();

public:
  void Update();
  bool WasWindowClosed() const;

}
```

Listing 1: Klasse Window Header

```cpp
/* in Window.cpp */
#include <iostream>
#include "Window.h"

Window::Window(const char* inTitle, GLsizei inWidth, GLsizei inHeight)
  :windowTitle(inTitle),windowWidth(inWidth),windowHeight(inHeight)
{
  windowInstance = nullptr;

  if(!InitWindowAndContext())
  {
    std::cout << "Failed to initialize!" << std::endl;
  }
}

Window::~Window()
{
  // Shutdown Methode der GLFW Lib, zerstört auch unseren windowInstance Pointer
  glfwTerminate();
}

bool Window::InitWindowAndContext()
{
  bool wasSuccessful = false;
  if (glfwInit() == GL_TRUE)
  {
    windowInstance = glfwCreateWindow(windowWidth, windowHeight, windowTitle, NULL, NULL);

    if (!windowInstance)
    {
      glfwTerminate();
      return wasSuccessful;
    }

    glfwMakeContextCurrent(windowInstance);
    wasSuccessful = true;
  }

  return wasSuccessful;
}
```

Listing 2: Klasse Window Quelldatei (Ctor und Init)

```cpp
/* in Window.cpp */

/*
   ... ctor ...
   ... init ...
*/

void Window::Update()
{
   glfwPollEvents();
   glfwSwapBuffers(windowInstance);
}

bool Window::WasWindowClosed() const
{
   return glfwWindowShouldClose(windowInstance);
}
```

4.2. The Main and The Window

Die Main dient als Einstiegspunkt für unsere Applikation und sollte so klein wie möglich gehalten werden. In der Regel startet man von hier aus eine Art Motor/Engine Klasse oder System und alles weitere passiert dann innerhalb dieser Klassen.

Wir nutzen die Main zunächst dazu unser Fenster zu erzeugen und unsere “Update Schleife” zu starten:

```
1  /* in Main.cpp */
2
3  #include "Window.h"
4
5  int main()
6  {
7    Window window("Einführung in Computergrafik", 800, 600);
8
9    // Solange das Fenster NICHT geschlossen wurde führe den Update
       durch
10   while(!window.WasWindowClosed())
11   {
12     window.Update();
13   }
14 }
```

4.3. Löschen des Fensters

Mit “Löschen” ist nicht die Zerstörung gemeint, sondern wie schon gesagt, ist das Window dafür verantwortlich den “bemalbaren” Bereich auf das “Malen” vorzubereiten. Dazu wird jeden Frame der Bildschrim “gelöscht”. In der Regel wird dazu eine Farbe genommen. Diese Farbe wird dann auf die gesamte bemalbare Fläche angewendet.

Wir erweitern die Klasse Window um die Methode “Clear”:

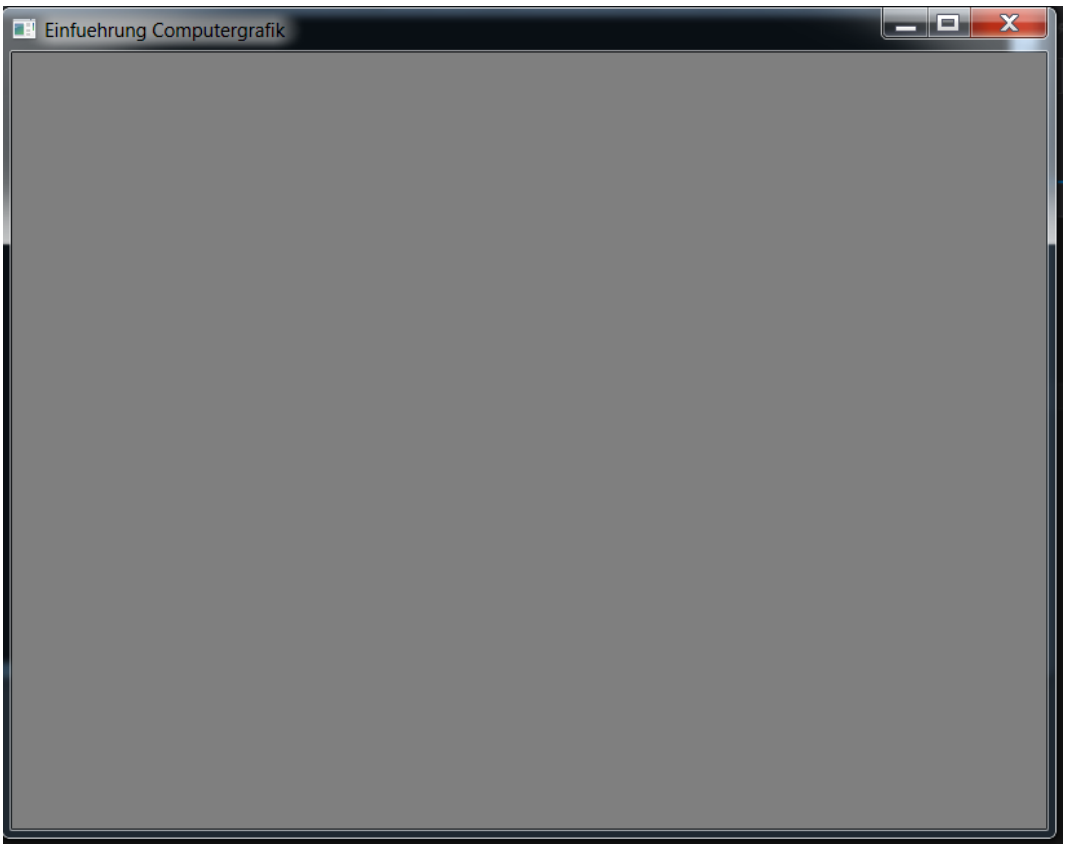
```
/* in Window.h */

class Window
{
  /*[...]*/
public:
  void Clear() const;
}

/* in Window.cpp */

// im Window Ctor oder Init
{
  // Definiere die Farbe mit der der Hintergrund gelöscht wird
  glClearColor(0.5f,0.5f,0.5f,1.0f); // -> Löschen mit grauer Farbe
}

void Window::Clear() const
{
  // Lösche den Hintergrund
  glClear(GL_COLOR_BUFFER_BIT); // GL_COLOR_BUFFER_BIT Konstante
   bezieht sich auf die glClearColor
}
```

Und nun können wir unseren “Render Zyklus” innerhalb unserer while Schleife vervollständigen:

```
/* in main.cpp */
while(!window.WasWindowClosed())
{
  window.Clear(); // Löschen des Bildschirms

  // Malen

  window.Update(); // Eventhandling und Framebuffer switch
}
```

Ergebnis:

Figure 5: Fenster mit ClearColor



5. Das erste Dreieck - Old School OpenGL

Im folgenden werden wir anhand weniger Zeilen Code ein einfaches Dreieck auf unseren Bildschrim zaubern, so wie man es “früher” gemacht hat.

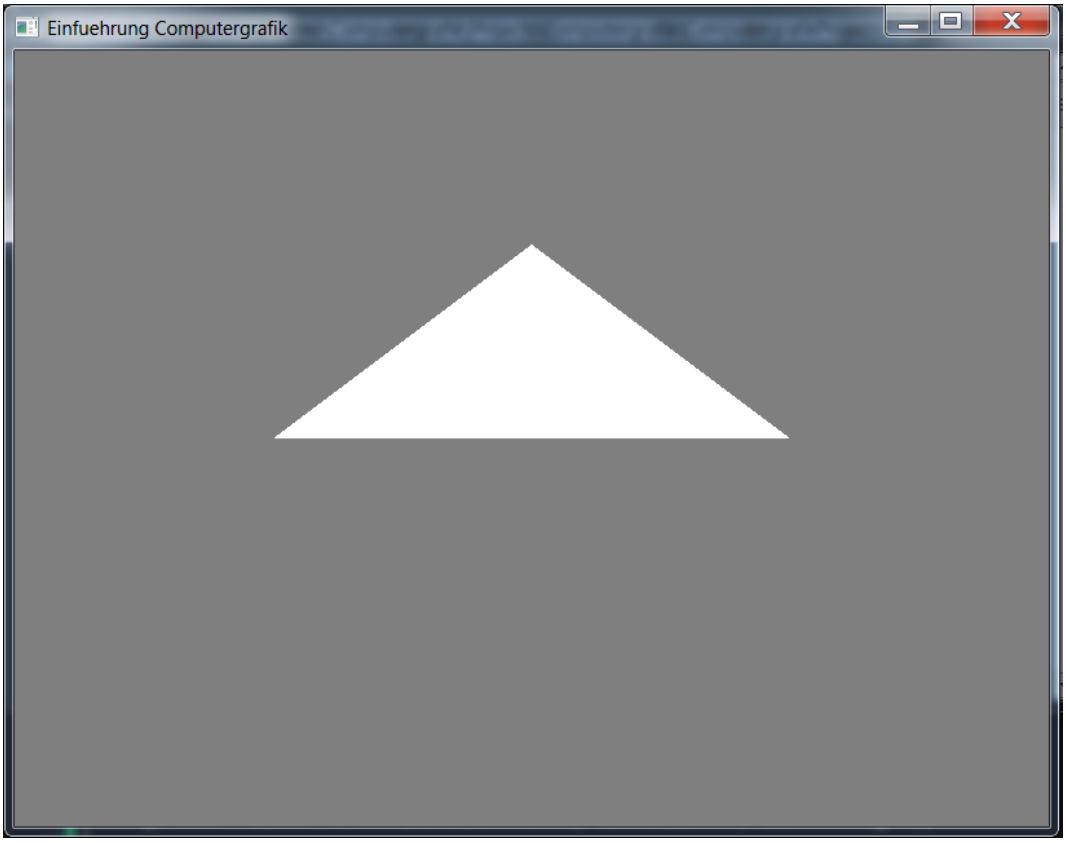
Wir fügen folgenden Code in die Update Methode des “Windows”:

```
void Window::Update()
{
    glfwPollEvents();

    glBegin(GL_TRIANGLES);
        glVertex2f(-0.5f, 0.0f);
        glVertex2f(0.5f, 0.0f);
        glVertex2f(0.0f, 0.5f);
    glEnd();

    glfwSwapBuffers(windowInstance);
}
```

Listing 3: Oldschool OpenGL Triangle

Das sollte folgendes Ergebnis liefern:

Figure 6: Ein Dreieck mit Oldschool OpenGL



5.1. Das “Window-Resize”-Event

Im letzten Part haben wir das erste Dreieck in der Mitte des Bildschirms gerendert. Wenn man nun manuell die Fenstergröße verändert sollte man feststellen, dass sich das Dreieck nicht mehr in der Mitte befindet. Das ist ein Fehler und diesen müssen wir beheben.[1]

Wenn die Fenstergröße verändert wird müssen wir das Fenster bzw. den Viewport in OpenGL über die neuen Dimensionen Width und Height informieren und diese neu setzen. Das Verändern der Größe erzeugt einen sogenannten “Callback” bzw. löst es ein Event aus, für welches wir uns quasi registrieren müssen um darauf zu reagieren.

Wir möchten also eine Funktion haben die Aufgerufen wird sobald dieses Event eintritt.

Diese Callbacks werden außerhalb eines Klassenscopes definiert, sie haben nicht direkt etwas mit der Klasse wie zum Beispiel Window selbst zu tun. Wir könnten die Datei Window nutzen oder wir lagern unsere Callbacks in eine oder mehrere sepparate Dateien. Letzteres werden wir tun und so fügen wir unserem Projekt einen neue Datei “WindowCallbacks.h” und “WindowCallbacks.cpp” hinzu.

Danach definieren wir unseren Resize Callback wie folgt:

```
1  /* WindowCallbacks.h */
2
3  // Der Name der Function kann frei gewählt werden, nur die Parameter
       Reihenfolge und Typen sind wichtig
4  void OnWindowResize(GLFWwindow* resizedWindow, GLsizei newVertices?
       Width, GLsizei newHeight);
5
6  /* ---------------------------------- */
7  /* in WindowCallbacks.cpp */
8
9  void OnWindowResize(GLFWwindow* resizedWindow, GLsizei newWidth,
       GLsizei newHeight)
10 {
11   // Setze den Viewport auf folgenden Bereich
12   glViewport(0,0,newWidth, newHeight);
13 }
```

Listing 4: Deklaration und Definition des Resize Callbacks

[1] Abhängig des verwendeten Compilers, in manchen Fällen wird dieser Fehler schon automatisch behoben

Nun haben wir den Callback definiert und müssen diesen nur noch registrieren. Dazu nutzen wir die Init unserer Window Klasse in der ich eine Funktion namens SetWindowCallbacks() angelegt habe:

```
1 // Aufruf während der Initialisierung
2 void Window::SetWindowCallbacks()
3 {
4   // Sobald das Fenster unserer Window-Instanz verändert wird, wird
     OnWindowResize aufgerufen
5   glfwSetWindowSizeCallback(windowInstance, OnWindowResize);
6 }
```

Listing 5: Registrieren des Resize Events

Wenn wir unser Programm nun wieder starten sollte sich beim Verändern der Fenstergröße das Dreieck stets in der Mitte befinden.

Denkaufgabe Welches Problem ist entstanden bezüglich unserer Wrapper Klasse Window und dem Resize-Event? (Tipp: Auflösung)

5.2. Refactoring der Window Klasse

Durch unser Resize Event entsteht eine Inkonsistenz zwischen den Member der Klass Window: windowWidth und windowHeight. Diese werden nicht aktualisiert, wenn unser Fenster sich verändert hat.

Es stellt sich jedoch die Frage ob wir diese Member überhaupt brauchen. Wenn man zur Laufzeit die aktuelle Fenstergröße in Erfahrung bringen möchte kann man dies über den Aufruf von

glGetFrameBufferSize(window,&int, &int);

herausbekommen. Auch die Speicherung des Titels ist nicht wirklich von nöten. Deshalb können wir unsere Window Klasse etwas verfeinern.

```
/* in Window.h */
class Window
{
private:
  // Title, Width und Height gelöscht...
  GLFWwindow* windowInstance;

public:
  Window(const char* inTitle, GLsizei inWidth, GLsizei inVertices?
   Height);
  ~Window();

private:
  // Da Member fehlen werden die Ctor Parameter an diese Funktion
   weitergegeben
  // Ursprünglich war das die Funktion "InitWindowAndContext"
  bool CreateWindowInstance(const char* inTitle, GLsizei inWidth,
   GLsizei inHeight);

  void SetWindowCallbacks();

public:
  // Updates the renderscene of the windowinstance
  void Update();

  // Clears the window with the specified clear color from
   glClearColor(color)
  void Clear() const;

  // Checks if the window was closed
  bool WasWindowClosed() const;
  };
```

Listing 6: Refactoring Window.h

Und in der Cpp:

```
/* in Window.cpp */
Window::Window(const char* inTitle, GLsizei inWidth, GLsizei
    inHeight)
{
  if (!CreateWindowInstance(inTitle, inWidth, inHeight))
  {
    std::cout << "Failed to create window!" << std::endl;
  }

  glClearColor(0.5f, 0.5f, 0.5f, 1.0f);
  SetWindowCallbacks();
}

bool Window::CreateWindowInstance(const char* inTitle, GLsizei
    inWidth, GLsizei inHeight)
{
  bool wasSuccessful = false;

  // Der Return von glfwInit ist ein Integer von 0 oder 1. Implizite
    Konvertierung ist auf den meisten Compilern möglich, wird aber
  // meist auch vom Compiler als Warnung angezeigt. Von Daher
   Vergleichen wir den Return mit GL_TRUE (aka 1) oder GL_FALSE (aka
    0)
  if (glfwInit() == GL_TRUE)
  {
    windowInstance = glfwCreateWindow(inWidth, inHeight, inTitle,
   NULL, NULL);

    if (!windowInstance)
    {
      glfwTerminate();
      return wasSuccessful;
    }

    glfwMakeContextCurrent(windowInstance);
    wasSuccessful = true;
  }

  return wasSuccessful;
}
```

Listing 7: Refactoring Window.cpp

5.3. 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 (aka unsere windowInstance) machen. D.h. nachdem die Init von GLFW erfolgreich war erweitern wir unsere Inits der Klasse Window wie folgt:

```
1 /* in Window.h */
2 class Window
3 {
4 [...]
5 private:
6   bool InitGLEW();
7 [...]
8 };
```

Listing 8: GLEW Init Header

```cpp
/* in Window.cpp */
Window::Window(const char* inTitle, GLsizei inWidth, GLsizei
    inHeight)
{
  if (!CreateWindowInstance(inTitle, inWidth, inHeight))
  {
    std::cout << "Failed to create window!" << std::endl;
  }

  if(!InitGLEW()
  {
    std::cout << "Failed to initialize GLEW" << std::endl;
  }

  glClearColor(0.5f, 0.5f, 0.5f, 1.0f);
  SetWindowCallbacks();
}

bool Window::InitGLEW()
{
  bool wasSuccessful = false;

  // SEHR WICHTIG: glewInit liefert sogennante ErrorCodes zurück.
    Alle Werte != 0 stellen einen bestimmten
  // Typ von Fehler dar. Wenn 0 zurück kommt gab es keine Fehler.
  if (glewInit() == GL_NO_ERROR)
  {
    // Kleiner Glew Test, aktuelle OpenGL Kompatibilität wird auf
    der Konsole ausgegeben
    std::cout << glGetString(GL_VERSION) << std::endl;
    wasSuccessful = true;
  }

  return wasSuccessful;
}
```

6. Das erste Dreieck - The Cool New Fancy OpenGL Way!

Anstatt, wie noch im “Intermediate Mode” üblich, das Dreieck über Begin und End zu zeichnen möchte ich die modernere Methode vorstellen - Bufferered Mode. Alle Vertex Informationen eines Objektes werden in Form von Buffers auf der GPU gespeichert und von dort ausgelesen und ge”malt”.

Ein “Renderable” beinhaltet zumindest ein Array von Punkten “Vertices”. Jedoch speichern wir nicht die vertices sondern nur den Identifier des Buffers, den wir für dieses Objekt generieren. Folgender Code entsteht:

```
1  /* SimpleTriangleShape2D.h */
2
3  class SimpleTriangleShape2D
4  {
5  private:
6    // Vertex Buffer Object
7    GLuint vbo;
8
9  public:
10   SimpleTriangleShape2D();
11   ~SimpleTriangleShape2D();
12
13   void Draw();
14 }
```

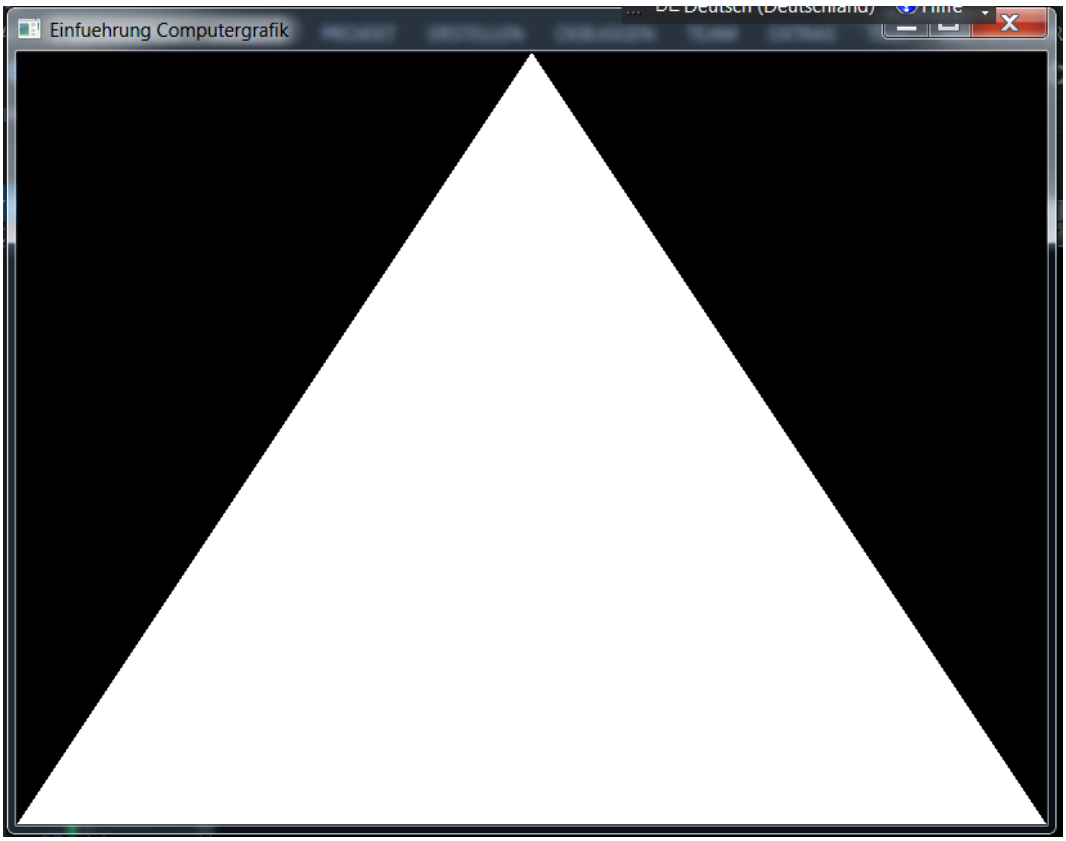
Listing 9: Simple Triangle - Buffered

```
/* SimpleTriangleShape2D.cpp*/

SimpleTriangleShape2D::SimpleTriangleShape2D()
{
  // Ein Array bestehend aus 3 Vertices
  GLfloat vertices[] =
  {
    -1,-1,0,
    1,-1,0,
    0,1,0
  }

  // Erzeuge Buffer Identifier (OpenGL nutzt diese ID um auf GPU
   Memory Bereiche zuzugreifen)
  glGenBuffers(1,&vbo);

  // Aktiviere den GPU Memory um lesenden und schreibenden Zugriff
   zu haben
  glBindBuffer(GL_ARRAY_BUFFER, vbo);

  // Speichere das Array vertices auf der GPU
  glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
   GL_STATIC_DRAW);

  // Fake Shader
  glEnableVertexAttribArray(0);
  glVertexAttribPointer(
  0,        // Shader Attribute Location
  3,        // Wie viele Elemente bilden 1 Vertex
  GL_FLOAT,    // Datentyp des Vertex Arrays
  GL_FALSE,    // Normalized -> so gut wie immer false
  0,         // stride
  0           // Array Buffer offset

  // Deaktiviere den Buffer (Hint: Opengl State Machine)
  glBindBuffer(GL_ARRAY_BUFFER, 0);
}

SimpleTriangleShape2D::~SimpleTriangleShape2D()fragmentColor
{
  glDeleteBuffer(&vbo);
}

SimpleTriangleShape2D::Draw()
{
  // Aktiviere Objekt Buffer
```

```
44      glBindBuffer(GL_ARRAY_BUFFER, vbo);
45
46      // Male Buffer Daten
47      glDrawArrays(GL_TRIANGLES, 0, 3);
48
49      // Deaktiviere Buffer wieder...
50      glBindBuffer(GL_ARRAY_BUFFER, 0);
51  }
```

Das Resultat (Nachdem ein Objekt erzeugt und Draw innerhalb der Schleife gecallt wird):

Figure 7: Fancy Triangle



7. Shader - Einführung in GLSL

7.1. 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.

7.2. Was sind Shader?

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

Zur Erinnerung:

Figure 8: Grafik Pipeline Reminder

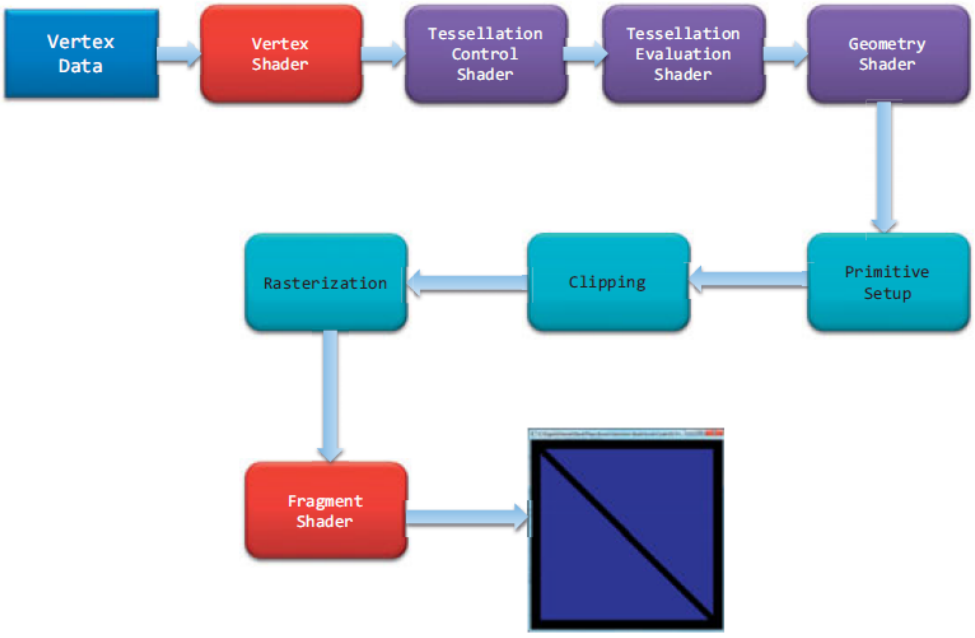


Figure 1.2 The OpenGL pipeline

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.

7.3. 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:

```
#version 330 core

layout(location = 0) in vec3 vertexPosition;

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

Listing 10: 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.

7.4. 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:

```
#version 330 core

layout (location = 0) out vec4 outColor;

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

Listing 11: 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.

7.5. Kompilieren eines Shaderprogramms

Ein Shaderprogramm besteht aus mehreren (mind. 2 Shadern). In unserem Fall ist das der Vertex und der Fragment Shader.

Da wir bis dato noch keine Dateien einlesen können, müssen wir den Shader zu Fuß in einem char array schreiben. Dazu nutzen wir zunächst unsere main Datei.

Die Definition der beiden Shader sieht wie folgt aus:

```
/* in main.cpp */

int main()
{
  Window window("Einfuehrung Computergrafik", 800, 600);

  const char* vertexShaderSource =
  "#version 330 core\n\
  \
  layout(location = 0) in vec3 vertexPosition;\
  \
  void main()\
  {\
    gl_Position = vec4(vertexPosition, 1);\
  }";

  const char* fragmentShaderSource =
    "#version 330 core\n\
    \
    layout(location = 0) out vec4 color;\
    \
    void main()\
    {\
      color = vec4(1, 0, 1, 1);\
    }";

  [...loop...]
}
```

Nun haben wir den Quellcode von 2 Shadern. Diese beiden Shader müssen nun kompiliert und in ein Shaderprogramm geladen werden.

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);
```

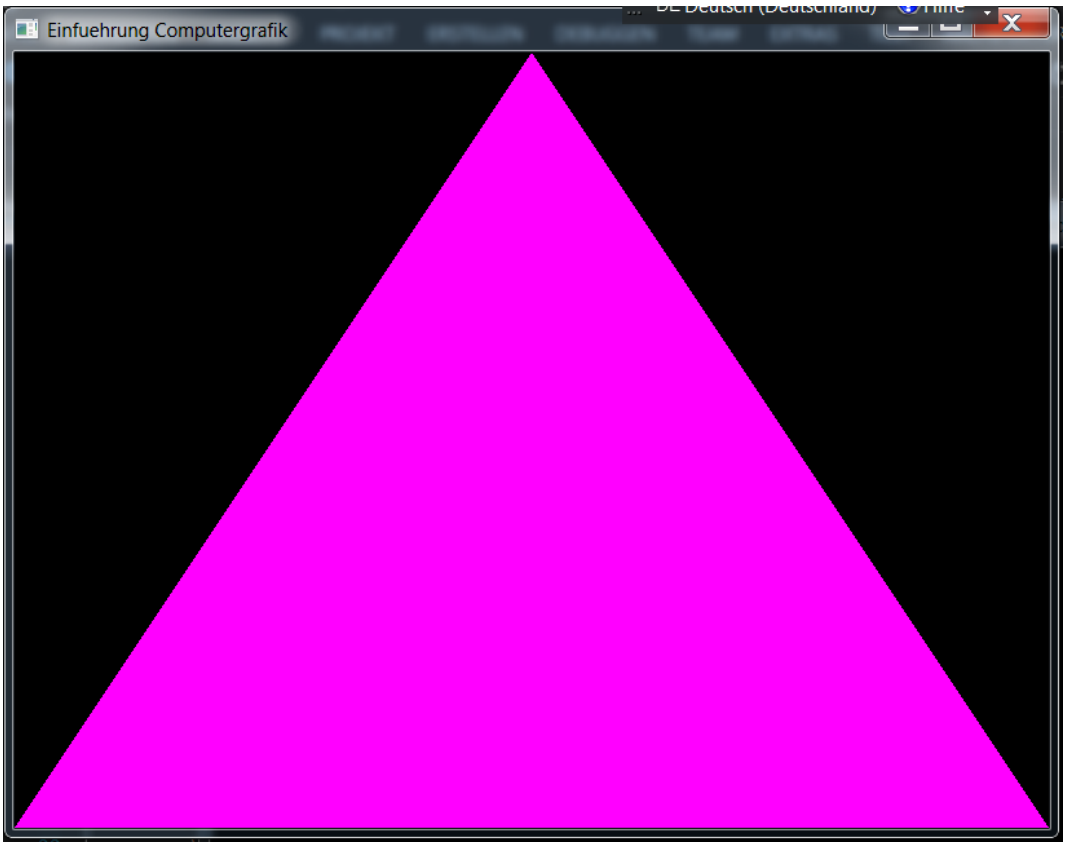
Unsere Main sieht für unsere beiden Shader so aus:

```
/* main.cpp */
[... window ...]
[... vertex und fragment shader source ...]

  // Erzeuge Programm
  GLuint shaderProgram = 0;
  shaderProgram = glCreateProgram();

  // VertexShader
  GLuint vertexShaderID = glCreateShader(GL_VERTEX_SHADER);
  glShaderSource(vertexShaderID, 1, &vertexShaderSource, NULL);
  glCompileShader(vertexShaderID);

  // FragmentShader
  GLuint fragmentShaderID = glCreateShader(GL_FRAGMENT_SHADER);
  glShaderSource(fragmentShaderID, 1, &fragmentShaderSource, NULL);
  glCompileShader(fragmentShaderID);

  // Verknüpfung Shader und Programm
  glAttachShader(shaderProgram, vertexShaderID);
  glAttachShader(shaderProgram, fragmentShaderID);

  // Verknüpfung Programm und GPU Prozessoren
  glLinkProgram(shaderProgram);

  // Programminfo in Logfile
  glValidateProgram(shaderProgram);

  // Speicherbereinigung einzelner Shader
  glDeleteShader(vertexShaderID);
  glDeleteShader(fragmentShaderID);

  // Aktivieren des Programm, sodass es das aktuell genutzte ist
  glUseProgram(shaderProgram);

  // Ein Dreieck Objekt
  SimpleTriangle triangle = SimpleTriangle();
  while (!window.WasWindowClosed())
  {
    window.Clear();
    triangle.Draw();
    window.Update();
  }
```

Listing 12: Shaderprogramm Kompilierung

Das Ergebnis auf unserem Bildschirm sollte folgendes sein:

Figure 9: Ein Dreieck mit Shadern



7.6. Verbindung zwischen Buffer und Shader

Nun haben wir ein hübscheres Dreieck gezeichnet, jedoch stellt sich die Frage warum sehen wir überhaupt das Objekt Triangle? Wir haben nun zwar 2 Shader geschrieben, die sogar funktionieren, doch wo verbinden wir die Shader mit den Daten die wir zeichnen wollen?

Ein Rückblick in SimpleTriangle.cpp:

```
/* im ctor von SimpleTriangle.cpp */

// Fake Shader
  glEnableVertexAttribArray(0);
  glVertexAttribPointer(
  0,        // Shader Attribute Location
  3,        // Wie viele Elemente bilden 1 Vertex
  GL_FLOAT,    // Datentyp des Vertex Arrays
  GL_FALSE,    // Normalized -> so gut wie immer false
  0,         // stride
  0           // Array Buffer offset
```

Innerhalb dieses Konstruktors hatten wir zunächst ein VertexBufferObject (vbo) erzeugt und in diesem die Daten des Dreiecks auf der GPU gespeichert. Mit den beiden obigen Befehlen entsteht zudem die Verbindung zum VertexShader. Denken Sie an die Pipeline, in der nach dem Block “VertexData” der VertexShader kommt.

Der erste Befehl:

```
glEnableVertexAttribArray(0);
```

Teilt der OpenGL State Machine mit, dass das Attribut vertexPosition (am Index “0”) für den aktuell gebundenen Buffer zu binden. Jeder eingehende Vertex kommt somit an folgender Stelle als “Input” an:

```
/* main.cpp VertexShaderSource */
layout(location = 0) in vec3 vertexPosition;
```

Damit verknüpfen wir also schon einmal unseren Buffer mit einem Attribut im Shader. Nun müssen wir OpenGL noch das Format unserer vertices mitteilen, folgende Fragen müssen für OpenGL beantwortet werden:

- Welche Location im Shader sprechen wir an?
- Wieviele Elemente im Array bilden 1 Vertex? (Also ein Vector3 besteht aus 3 Elementen im Array)
- Aus welchem Datentype (um die Bytelänge pro Vertex zu bestimmen) besteht 1 Vertex
- Soll ein Vertex auf den Range [0..1] gemappt werden? (Normalisieren) -> Ist in der Regel immer False
- Sind die Vertices zusammenhängend? Also kommt nach einem Vertex direkt der nächste?
- Von welcher Position im Array soll gestartet werden?

Damit haben wir OpenGL alles mitgeteilt, was nötig ist um unser Dreieck so zu zeichnen, wie wir es definiert haben.