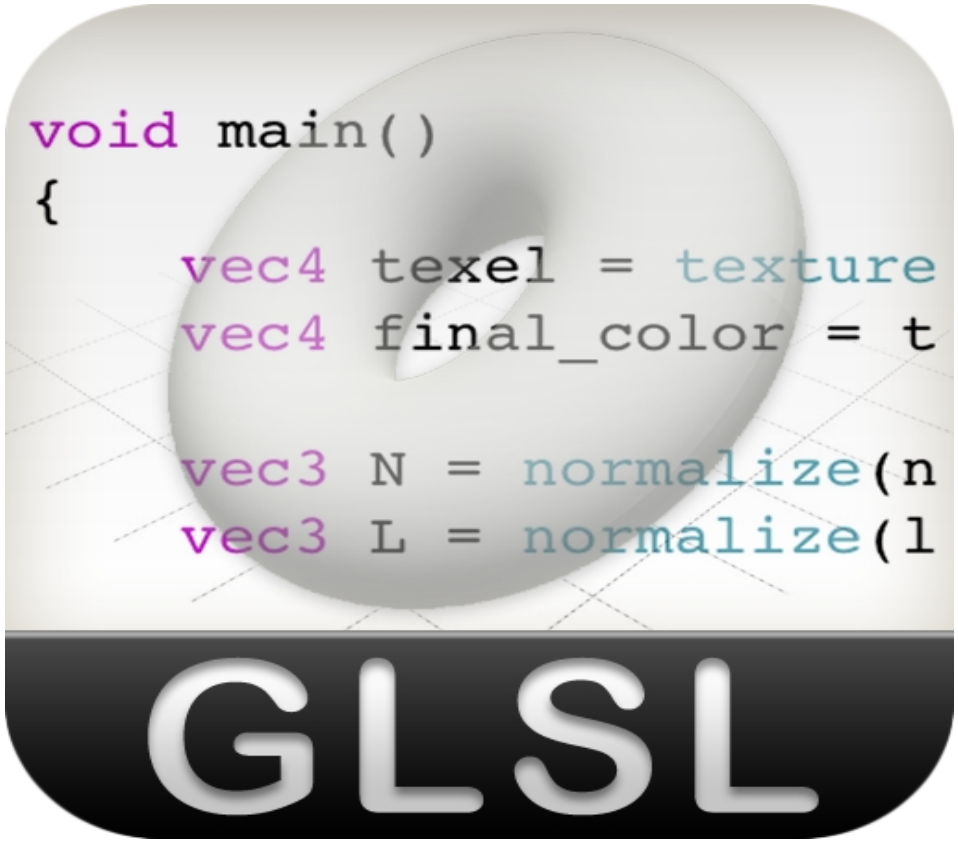
Praktikum Computergrafik

Abteilung für Bild- und Signalverarbeitung

Betreuer: Baldwin Nsonga


OpenGL


void main()
{
vec4 texel = texture
vec4 final_color = t
vec3 N = normalize(n
vec3 L = normalize(l
GLSL

OpenGL

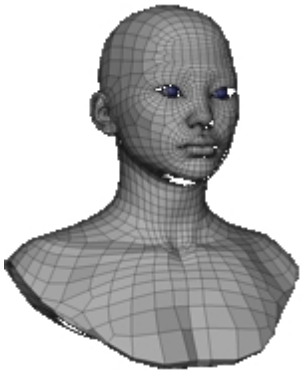
- **OpenGL** (**Open G**raphics **L**ibrary)
  - plattform- und programmiersprachenunabhängige Programmierschnittstelle zur Entwicklung von 2D- und 3D-Computergrafik
  - ermöglicht die Darstellung komplexer 3D-Szenen in Echtzeit
  - Implementierung ist normalerweise durch Grafikkartentreiber gewährleistet (hardwarebeschleunigt), ansonsten auf der CPU
  - Windows-Pendant: Direct3D

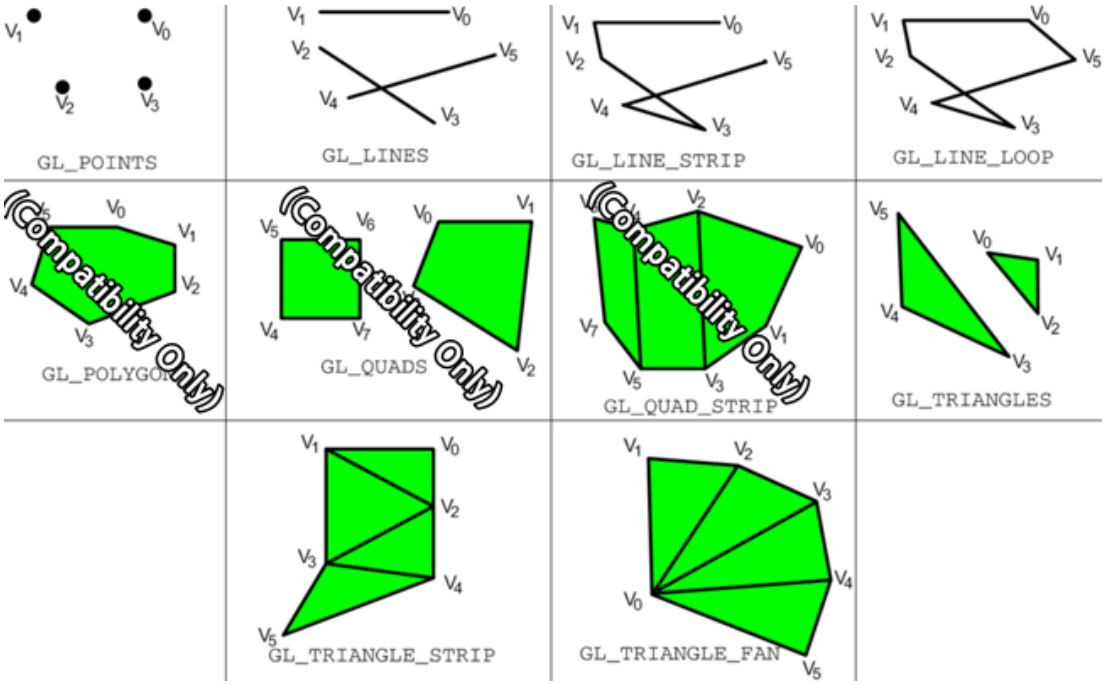
OpenGL

- Bekannte Engines:
  - GrimE-Engine (Escape From Monkey Island)
  - Id Tech 4/5 Engine (Doom 3, Brink, Rage)
  - Aurora Engine (Neverwinter Nights)
  - Source Engine (Half Life 2)
  - Unreal Engine (Goat Simulator)

OpenGL Einführung - Geometrie

- Komplexe 3D-Modelle bestehen immer aus geometrischen Primitiven



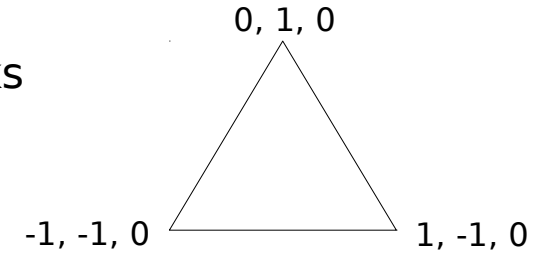


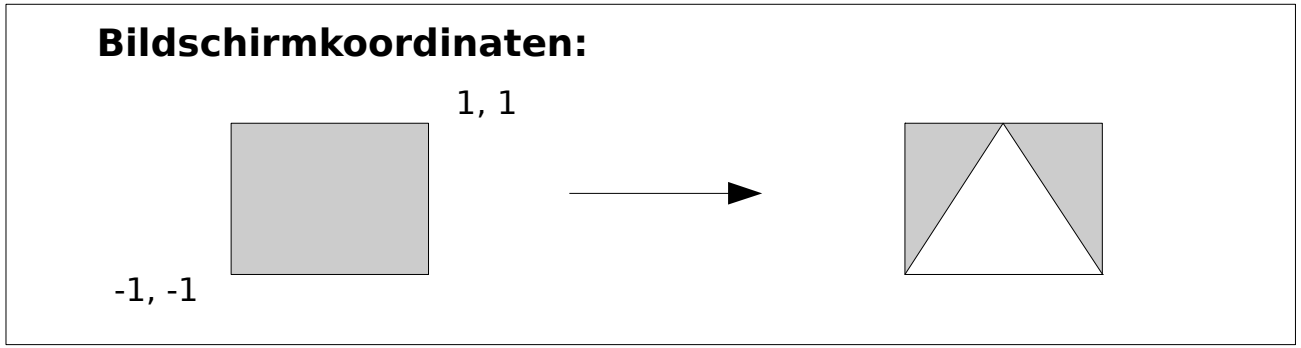
Im Praktikum: Beschränkung auf Dreiecke (GL_TRIANGLES)

- Primitive bestehen immer aus Vertices (Eckpunkte). Jeder Vertex kann mehrere Attribute haben (Position, Farbe, Normale...).

OpenGL Einführung - Geometrie

- Bsp: Zeichnen des Dreiecks





- Positionen der Vertices in Array speichern:

```
GLfloat pos_data[] = { -1.0f, -1.0f, 0.0f,
                        1.0f, -1.0f, 0.0f,
                        0.0f,  1.0f, 0.0f, };
```

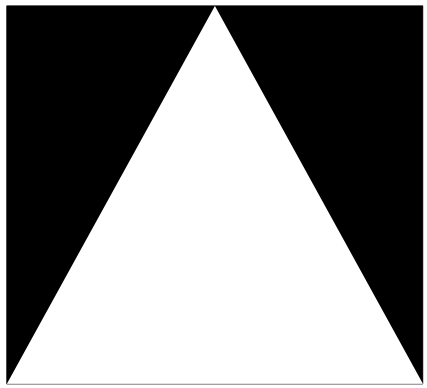
- Diese Daten werden als sogenannte Buffer auf der Grafikkarte gespeichert:

```
GLuint positionBuffer;
//Buffer erstellen
glGenBuffers(1, &positionBuffer);
//Buffer als aktiv setzen (OpenGL ist eine State Machine)
glBindBuffer(GL_ARRAY_BUFFER, positionBuffer);
//den Buffer mit den Positionsdaten befüllen
glBufferData(GL_ARRAY_BUFFER, sizeof(pos_data), pos_data, GL_STATIC_DRAW);
```

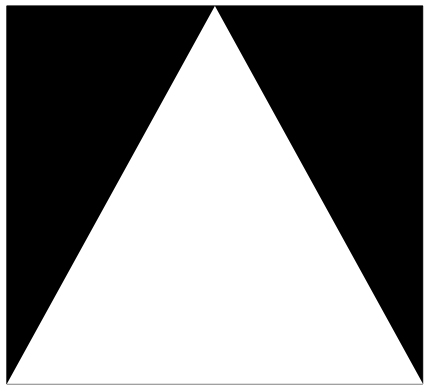
- Zeichnen der Daten mittels glDrawArrays:

```
glBindBuffer(GL_ARRAY_BUFFER, positionBuffer);
//VertexAttribute festlegen
glVertexAttribPointer(0,          //wichtig für shader (später)
                      3,          //Größe eines Vertexattributes
                      GL_FLOAT,   //Datentyp
                      GL_FALSE,   //normalisiert
                      0,          //stride
                      (void*) 0   //Offset im Buffer
                      );
//Zeichnen
glDrawArrays(GL_TRIANGLES, 0, 3);
```

- Ergebnis:



- Ergebnis:



Langweilig!

OpenGL Einführung - Geometrie

- Frage: Wie macht man so etwas ?

OpenGL Einführung - Geometrie

- Frage: Wie macht man so etwas ?

oder so etwas?

OpenGL Einführung - Geometrie

- Frage: Wie macht man so etwas ?

oder so etwas?

oder auch das?

OpenGL Einführung - Shader

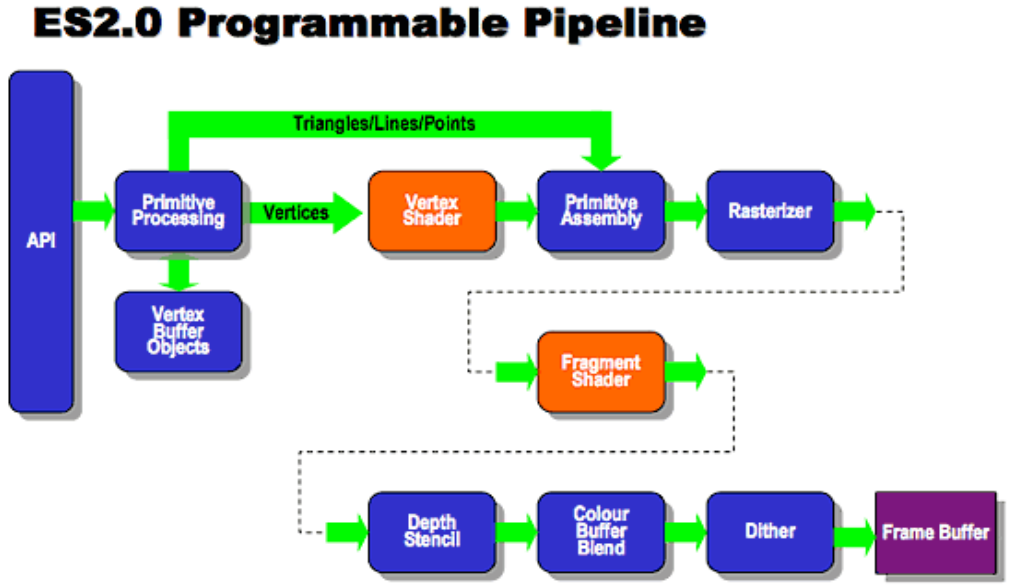
- Antwort: mit **Shadern**
  - Shader sind Programme, die direkt auf der Grafikkarte ausgeführt werden

    → die Grafikkarte ist eine frei programmierbare Multiprozessorplatform
- Mehrere Arten von Shadern

  → hier Beschränkung auf die 2 wichtigsten Shader: Vertex- und Fragmentshader (es gibt noch Geometry-, Tesselation- und Computeshader)

OpenGL Einführung - Shader

- **Shader Pipeline**:
  - OpenGL übergibt Vertex mit verschiedenen Eigenschaften (Position, Farbe, Texturkoordinaten usw.)
  - **Vertexshader** “bearbeitet” den Vertex und evtl. die übergebenen Eigenschaften
  - **Pixelshader** bekommt die *interpolierten* Eigenschaften (z.B. Vertexfarbe) und färbt das Pixel im Framebuffer



OpenGL Einführung - Shader

- **GLSL:**
  - Programmiersprache für Shader
  - DirectX-Pendant: HLSL
  - Syntax entspricht im Wesentlichen ANSI-C
  - wurde um spezielle Datentypen erweitert, wie z.B. Vektoren, Matrizen und Sampler (für Texturzugriffe)
  - Tutorial z.B. unter http://www.opengl-tutorial.org/

OpenGL Einführung - Shader

- **Bsp:** Vertex Shader

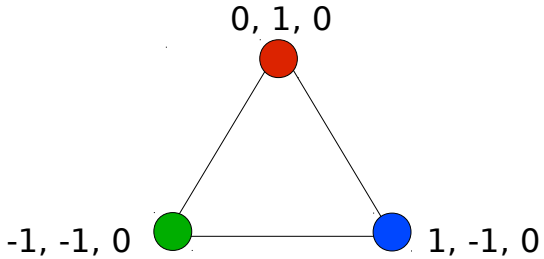
```
#version 330 core

//vertex attributes
in vec3 position;
in vec3 color;

//uniform variables (constant for the primitive)
uniform mat4 someMatrix;

//Varyings, are passed to the fragment shader
out vec3 col;

void main(void)
{
    //Vertex position in screen space
    gl_Position = someMatrix * vec4(position, 1.0);
    col = color;
}
```

- **Bsp:** Fragment Shader

```
#version 330 core

//some other uniform variable (constant for the primitive)
uniform float scale;

//Varying coming from the vertex shader
in vec3 col;

//the output of the fragmentshader, i.e. the color
out vec4 finalColor;

void main(void)
{
    finalColor = vec4( col*scale, 1.0 );
};
```

OpenGL Einführung - Shader

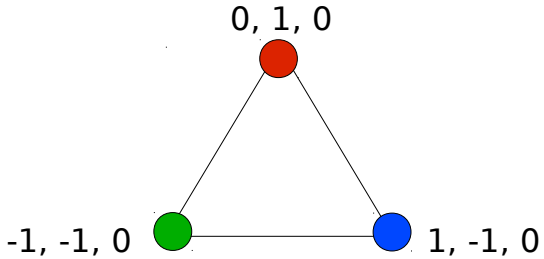
- **Ergebnis:**
  - Eingabe
    - Vertexattribute




```
GLfloat pos_data[] = { -1.0f, -1.0f, 0.0f,
                        1.0f, -1.0f, 0.0f,
                        0.0f,  1.0f, 0.0f, };
GLfloat col_data[] = {   0.0f, 1.0f, 0.0f,
                         0.0f, 0.0f, 1.0f,
                         1.0f, 0.0f, 0.0f, };
```

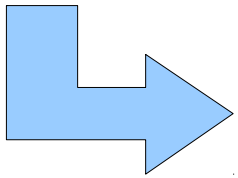
    - Uniforms:
      - `someMatrix` = Einheitsmatrix;
      - `scale` = 1.0f;

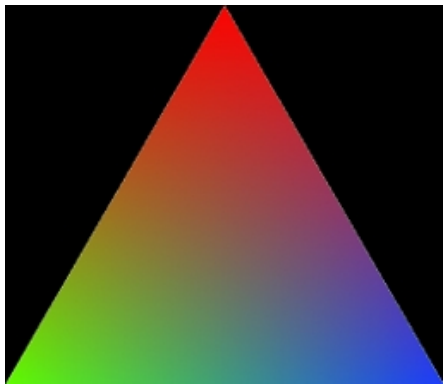
- **Ergebnis:**
  - Eingabe
    - Vertexattribute




```
GLfloat pos_data[] = { -1.0f, -1.0f, 0.0f,
                        1.0f, -1.0f, 0.0f,
                        0.0f,  1.0f, 0.0f, };
GLfloat col_data[] = {   0.0f, 1.0f, 0.0f,
                         0.0f, 0.0f, 1.0f,
                         1.0f, 0.0f, 0.0f, };
```

    - Uniforms:
      - `someMatrix` = Einheitsmatrix;
      - `scale` = 1.0f;





OpenGL Einführung - QGL

- Problem:
  - OpenGL API z.T. recht umständlich
  - Keine native Unterstützung für spezielle GLSL Datentypen (Vectoren, Matrizen)

    → es existieren viele Wrapper für OpenGL, die versuchen diese Nachteile aufzuwiegen

OpenGL Einführung - QGL

- Problem:
  - OpenGL API z.T. recht umständlich
  - Keine native Unterstützung für spezielle GLSL Datentypen (Vectoren, Matrizen)

    → es existieren viele Wrapper für OpenGL, die versuchen diese Nachteile aufzuwiegen

Wir benutzen Qt

OpenGL Einführung - QGL

- Qt:
  - http://qt-project.org/
  - platformunabhängiges Anwendungs und UI Framework

    → hauptsächlich benutzt zum Erstellen von GUI-Anwendungen mit C++
  - *CGViewer* ist mit Qt geschrieben
  - Unterstützt auch die die Darstellung GPU-beschleunigter Inhalte mittels OpenGL
    - besitzt auch OpenGL-Wrapperklassen und eigene Vector/Matrix Klassen, die entsprechend mit den OpenGL-Wrappern zusammenarbeiten

OpenGL Einführung - QGL

- Qt Klassen:
  - Vectoren: z.B. `QVector2D, QVector3D, QVector4D`
  - Matrizen: z.B. `QMatrix3x3, QMatrix4x4`
    - ```
      //translation matrix
      QMatrix4x4 translation;
      translation.translate(dx, dy, dz);
      ```
  - Buffer: `QGLBuffer`
    - ```
      QGLBuffer positionBuffer = QGLBuffer( QGLBuffer::VertexBuffer );
      positionBuffer.setUsagePattern( QGLBuffer::StaticDraw );
      positionBuffer.create();
      positionBuffer.allocate( &positions[0], positions.size()*sizeof(QVector3D) );
      //bind the buffer
      positionBuffer.bind();
      ```
  - Shader program: `QGLShaderProgram (creation)`
    - ```
      QGLShaderProgram pr;
      pr.addShaderFromSourceFile(QGLShader::Vertex, vertexShaderSource);
      pr.addShaderFromSourceFile(QGLShader::Fragment, fragmentShaderSource);
      pr.link();
      ```
      ```

- Qt Klassen:
  - Shader program (Frts.): `QGLShaderProgram (drawing)`
    - ```
      //set the program active
      pr.bind();
      //set the uniforms in the shader
      QMatrix4x4 matrix;
      pr.setUniformValue( pr.uniformLocation("someMatrix"), matrix );
      pr.setUniformValue( pr.uniformLocation("scale"), 1.0f );
      //enable the vertex attributes (one for positions, one for colors)
      pr.enableAttributeArray( pr.attributeLocation("position") );
      pr.enableAttributeArray( pr.attributeLocation("color") );
      //tell the shader, which data the vertex attributes should use
      positionBuffer.bind();
      pr.setAttributeBuffer( pr.attributeLocation("position"), GL_FLOAT, 0, 3 );
      colorBuffer.bind();
      pr.setAttributeBuffer( pr.attributeLocation("color"), GL_FLOAT, 0, 3 );
      //draw
      glDrawArrays(GL_TRIANGLES, 0, 3);
      //deactivate program
      pr.release();
      ```
      ```

CGViewer

- Framework, welches im Laufe des Praktikums erweitert werden soll

CGViewer

- Framework, welches im Laufe des Praktikums erweitert werden soll
  - Features
    - triangulierte Modelle im Wavefront OBJ-Format laden (http://de.wikipedia.org/wiki/Wavefront_OBJ)
    - Modelle können frei in der Szene bewegt, rotiert und skaliert werden
    - Hinzufügen von mehreren Lichtquellen zur Szene
    - Laden und Speichern von Szenen
    - Anzeigen der Szene erfolgt mittels OpenGL und der im Praktikum erstellten Shader

CGViewer

- Wichtige Dateien/Klassen
  - Klasse **Model**
    - Dateien Model.h, Model.cpp
    - besitzt statische Funktion zum laden von Modellen
    - verwaltet ein geladenes Modell mitsamt aller nötiger Buffer
    - verwaltet die Modellbewegung in der Szene (Koordinaten)
    - Besitzt eine `render`-Funktion, um sich zu zeichnen
      - der `render`-Funktion wird ein Shaderprogram übergeben, mit dem sich das Modell rendern soll

CGViewer

- Wichtige Dateien/Klassen
  - Klasse **Light** (erbt von Model)
    - Dateien Light.h, Light.cpp
    - wird immer als Kugel dargestellt
    - besitzt zudem Funktionen zum setzen und auslesen von Farb/Licht-Informationen

CGViewer

- Wichtige Dateien/Klassen
  - Klasse **Scene** (erbt von QGLWidget)
    - Dateien Scene.h, Scene.cpp
    - reagiert auf Mauseingaben des Nutzers (z.B. für Kamerabewegungen)
    - verwaltet das Shaderprogram (`QGLShaderProgram *m_program`)
    - verwaltet die geladenen Modelle (`std::vector< std::shared_ptr<Model> > models`)
    - zeichnet die Szene alle 33ms neu
      - Funktion `void paintGL()`, ruft die `render`-Funktion aller geladenen Modelle auf

CGViewer

- Wichtige Dateien/Klassen
  - Datei **CGTypes.h**
    - beinhaltet ein paar spezielle Datentypen
    - wichtig ist vor allem die struct `Material`
      - wird in den entsprechenden Aufgabenstellungen näher erläutert

CGViewer

- Wichtige Dateien/Klassen
  - **Shader**
    - sind im Unterverzeichnis Shader abgelegt:
      - VertexShader: vertex.glsl
      - FragmentShader: fragment.glsl
      - werden von der Scene automatisch geladen: `void reloadShader()`
    - während der Aufgaben werden neue Shaderdateien hinzugefügt