|  |
| --- |
| C:\Users\holzer\Downloads\teapot.obj.png |
| OpenGL 3D Model Viewer  Bericht  **Modul: BZG1310 "Objektorientierte Geometrie" Dozent: Marx Stampfli Autoren: Michael Koch, Joel Holzer**  **Version: 1.0, 18.01.2016** |
| **Berner Fachhochschule**  Technik und Informatik  Informatik |

# Einleitung

Im Rahmen des Moduls „Objektorientierte Geometrie“ an der Berner Fachhochschule erstellen wir eine Java Applikation, welche die Möglichkeit bietet, eine 3D Figur darzustellen und die 3D Figur von unterschiedlichen Seiten zu betrachten. Diese Applikation soll ein GUI beinhalten, über welches .OBJ Dateien eingelesen und angezeigt werden können. OBJ ist ein offenes Dateiformat zum Speichern von dreidimensionalen geometrischen Formen. Die angezeigte Figur kann via Maus-Interaktion von allen Seiten betrachtet werden. Zudem ist es möglich, mit dem Mausrad rein und raus zu zoomen und so die 3D Figur grösser oder kleiner zu betrachten.

Zur Modellierung der 3D Figuren verwenden wir OpenGL und die Shadersprache GLSL. Wir benötigen eine Projektionsmatrix und eine ViewMatrix. Die Projektionsmatrix lässt weit entfernte Objekte (in Richtung Z-Achse) kleiner darstellen und nahe bei der Kamera liegende Objekte grösser. Für die Bewegung der Kamera benötigen wir eine Viewmatrix, welche die aktuelle Position der Kamera beinhaltet.

Wir generieren eine Lichtquelle und berechnen mittels Phong Shading das daraus resultierende Ambient, Diffuse und Spekular Light der Szene.

Die aktuelle Version ist auf unserem GitHub Repository zu finden:

<https://github.com/joe1990/OpenGl3DModelViewer>

# Vorgehen

## Soll-Tasks

Nachfolgende Auflistung gibt die während der Umsetzung des Projekts anfallenden Tasks wieder, begonnen beim ersten Task. Beide Studierenden sind an der Arbeit aller Tasks beteiligt.

Vorgehen:

1. Einrichtung Entwicklungsumgebung (IntelliJ IDEA) mit GitHub
2. GUI Generierung mit GUI-Designer
3. Realisierung des Uploads für .Obj Dateien
4. Realisierung der Logik für Vertex Array Object (VAO) und Vertex Buffer Object (VBO)
5. Realisierung von Vertex und Fragment Shader
6. Realisierung der Projektion und Viewmatrix
7. Realisierung der Objekt oder Maus Rotiation
8. Finalisierung der Dokumentation
9. Präsentation und Abgabe

## Technologien & Tools

Für die Realisierung des Projektes haben wir folgende Technologien eingesetzt:

|  |  |
| --- | --- |
| **Anwendungsbereich** | **Technologie** |
| Umsetzung des User Interfaces (GUI) | Swing, AWT |
| Umsetzung des .Obj File-Uploader | Java |
| 3D-Modelierung | OpenGL mit GLSL (Programmable Pipeline) |
| OpenGL Toolkit | LWJGL |
| Build Management Tool | Maven |
| Entwicklungsumgebung | IntelliJ IDEA |
| Versionsverwaltung Programmcode | GitHub |
| Betriebssystem Entwicklungscomputer | Windows 7 |

Tabelle 1 Zum Einsatz kommende Technologien

## Methoden

Nachfolgend sind die wichtigsten Methoden, welche bei der Umsetzung der Anwendung eingesetzt wurden, kurz erläutert:

**Vertex Array Object (VAO) und Vertex Buffer Object (VBO):**  
Das VBO enthält die eigentlichen Vertexdaten beispielsweise die Positionen, die Normalenvektoren, die Farben oder die Textur-Koordinatenen. Das VAO hingegen enthält die Informationen, in welchem VBO sich die benötigten Daten befinden und in welchem Format sie vorliegen.

Das im Programm verwendete VAO besitzt folgende Form:

|  |  |  |
| --- | --- | --- |
| Position | Länge | Name |
| 1 | 3 floats | Positions |
| 2 | 3 floats | Normals |
| 3 | 3 floats | Colors |

Der Vorteil vom Einsatz von VAO und VBO ist, dass die Vertexdaten im VRAM der Grafikkarte gespeichert werden, statt wie ohne Einsatz von VAO und VBO im Hauptspeicher der CPU. Das Problem bei einer Speicherung der Vertexdaten im Hauptspeicher ist, dass die Grafikkarte deutlich langsamer auf den Hauptspeicher also auf das eigene VRAM zugreifen kann. Dies wirkt sich negativ auf die Performance der Anwendung aus, weil ein Flaschenhals zwischen CPU und GPU existiert. Dieser Flaschenhals kann mit dem Einsatz mit VAO und VBO umgangen werden.

**Projektion-, Transformation- und View Matrix:**Die Projektionsmatrix lässt weit entfernte Objekte (in Richtung Z-Achse) kleiner darstellen und nahe bei der Kamera liegende Objekte grösser. Für die Bewegung der Kamera benötigen wir eine Viewmatrix. Die Transformationsmatrix enthält die Translation, Rotation um alle Achsen und die Streckung mit der Objekte manipuliert werden können.   
Die genannten Matrizen kommen im Vertex Shader zur Anwendung um die endgültige Position eines Knotens zu finden:

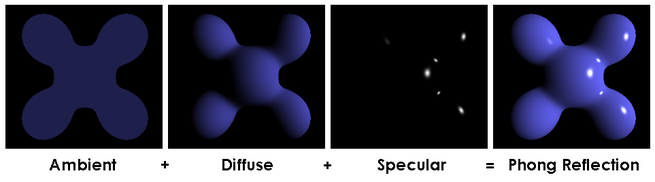
Position = projectionMatrix \* viewMatrix \* transformationMatrix

**Shader:**GLSL ist eine C-ähnliche Programmiersprache, mit welcher es möglich, selbst Shaders zu schreiben und auf der GPU auszuführen. In der zu realisierenden Anwendung werden 2 verschiedene Shaders benötigt, den Fragment Shader und den Vertex Shader. Der Vertex Shader wird pro Vertex (Knoten) aufgerufen und er dient dazu die Geometrie einer Szene zu manipulieren (View, Projektion, Transformation). Der Fragment Shader bestimmt den Farbwert pro Pixel.

Für die Realisierung des Pong Shadings verwendeten wir GLSL

**Phong Reflection**

Das Phong-Beleuchtungsmodell setzt sich aus drei Einzelkomponenten zusammen:



Im Vertexshader bestimmen wir die Position des aktuellen Knotens und geben Ihn zusammen mit dem Normalenvektor, dem Lichtvektor, dem Kameravektor und der Farbe an den FragmentShader weiter.

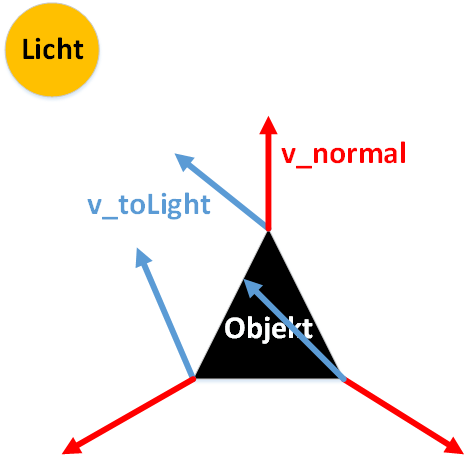
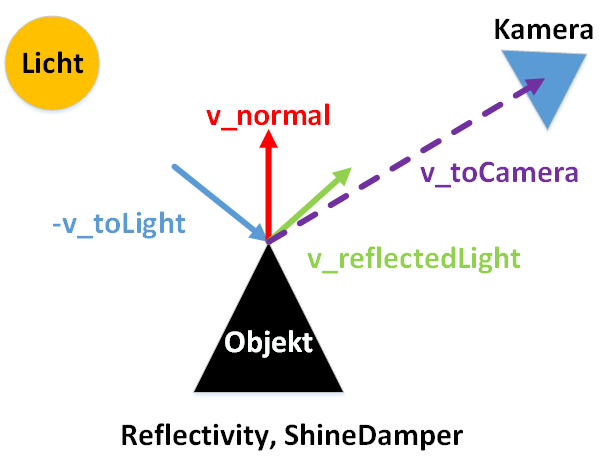
Nachfoldende Codeauszüg zeigen unsere implementationen des VertexShaders und des FragmentShaders.

//VERTEX SHADER

#version 400 core  
  
//in variables  
in vec3 position;  
in vec3 normal;  
in vec3 colour;  
  
//out variables  
out vec3 v\_toLight;  
out vec3 v\_toCamera;  
out vec3 v\_normal;  
out vec3 v\_color;  
  
//uniforms  
uniform vec3 light;  
uniform mat4 projection;  
uniform mat4 transformation;  
uniform mat4 view;  
  
void main()  
{  
 //defines the world-position of each vertex  
 vec4 transpos = transformation \* vec4(position, 1.0f);  
 gl\_Position = projection \* view \* transpos;  
  
 //übergebe die Farbe des Objektes  
 v\_color = colour;  
  
 //übergebe den Normalenvektor  
 v\_normal = (transformation \* vec4(normal, 1.0)).xyz; // 0.0  
  
 //übergebe den Vektor model->light  
 v\_toLight = light - position.xyz;  
  
 //übergebe den Vektor model->camera  
 v\_toCamera = (inverse(view) \* vec4(0.0,0.0,0.0,1.0)).xyz - transpos.xyz;  
}

//FRAGMENT SHADER

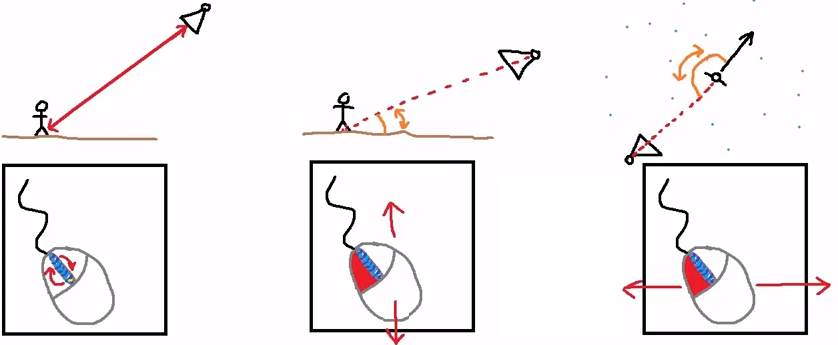
#version 400 core  
  
in vec3 v\_toLight;  
in vec3 v\_toCamera;  
in vec3 v\_normal;  
in vec3 v\_color;  
  
uniform vec3 lightColor;  
  
void main()  
{  
  
 //Normalize Vectors  
 vec3 normNormal = normalize(v\_normal); //normalize: vec länge auf 1 setzen  
 vec3 normToLight = normalize(v\_toLight);  
 vec3 normToCamera = normalize(v\_toCamera);  
  
 //diffuse  
 float diffFactor = dot(normNormal, normToLight);  
 float brightness = max(diffFactor, 0.4); //Ambient  
 vec3 diffuse = brightness \* lightColor;  
  
 //specular  
 vec3 lightDirection = -normToLight;  
 vec3 reflectedLight = reflect(lightDirection, normNormal);  
 float specFactor = dot(reflectedLight, normToCamera);  
 specFactor = max(specFactor, 0.0);  
  
 //Abweichung des Winkels zwischen toCamera und reflectedLight = 10  
 float aberrationFactor = pow(specFactor, 10);  
  
 //Reflektionsfähigkeit des Materials = 1  
 vec3 specular = aberrationFactor \* 1 \* lightColor;  
  
 //Ambient, diffuse und specular  
 gl\_FragColor = (vec4(diffuse, 1.0) \* vec4(v\_color, 1.0)) + vec4(specular, 1.0);  
  
}

**Objekt- und Mausrotation:**Die geladenen Objekte sollen auf einem Boden (Gitternetz) platziert werden. Sobald mit der Maus die Kamera rotiert wird, bewegt sich die ganze Szene, die Kamera hingegen bleibt statisch. Da sich auch der Boden mitbewegt, erhält der Benutzer der Anwendung den Eindruck, als bewege er die Kamera.

Der Benutzer hat 3 Möglichkeiten die Kameraperspektive zu verändern:

1. Mausrad: Distanz zwischen Kamera und Objekt
2. Links klick: Y-Rotation der Kamera (pitch)
3. Links klick: Winkel um das Objekt



Folgende Schnitte sind nun notwendig um die Position der Kamera und deren Rotation zu bestimmen.

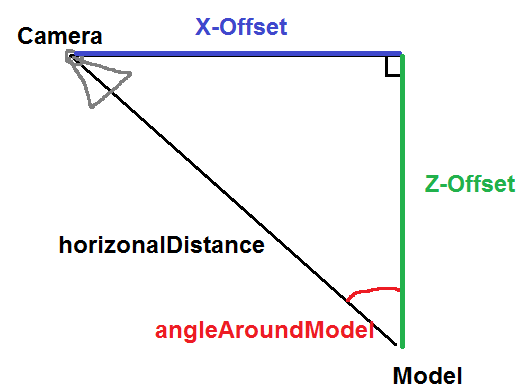
1. Berechnung der horizontalen und vertikalen Distanz zwischen Kamera und Objekt mittels den Winkelfunktionen sin und cos.

distance\_horizontal = **distanceToModel** \* Math.*cos*(**pitch**)  
distance\_vertical = **distanceToModel** \* Math.*sin*(**pitch**)

2. Berechnung des X-Offsets und Z-Offsets mittels den Winkelfunktionen sin cos

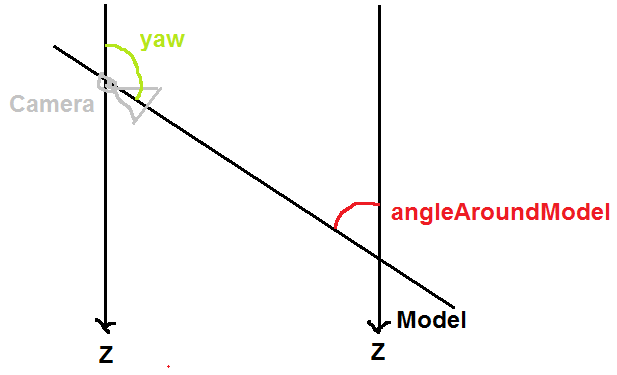
xPos = distance\_horizontal \* Math.*sin*(**angleAroundModel**)  
zPos = distance\_horizontal \* Math.*cos*(**angleAroundModel**)

position.**x** = - xPos  
position.**z** = - zPos  
position.**y** = distance\_vertical



3. Berechung der Kamerarotation (yaw)

**yaw** = 180 - **angleAroundModel**;

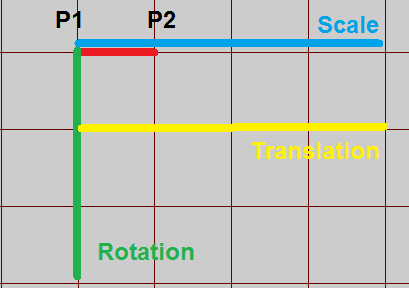


**Gitternetz**

Für das Gitternetz werden nur 2 Knoten verwendet:

* P1(0,0,0)
* P2(1,0,0)

Jeder Geradenabschnitt referenziert P1 und P2. Die genaue Bestimmung der Anfangs- und End-Position erfolgt via einer Transformationsmatrix: Translation, Rotation, Streckung/Stauchung.



# UserInterface

Nachfolgende Abbildung zeigt das User Interfaces. Im oberen Bereich kann mit Hilfe eines FileOpenDialogs ein .Obj-File ausgewählt und geladen werden. Der Inhalt dieses .Obj-File wird dann von der Anwendung im unteren Bereich als 3D-Objekt angezeigt.

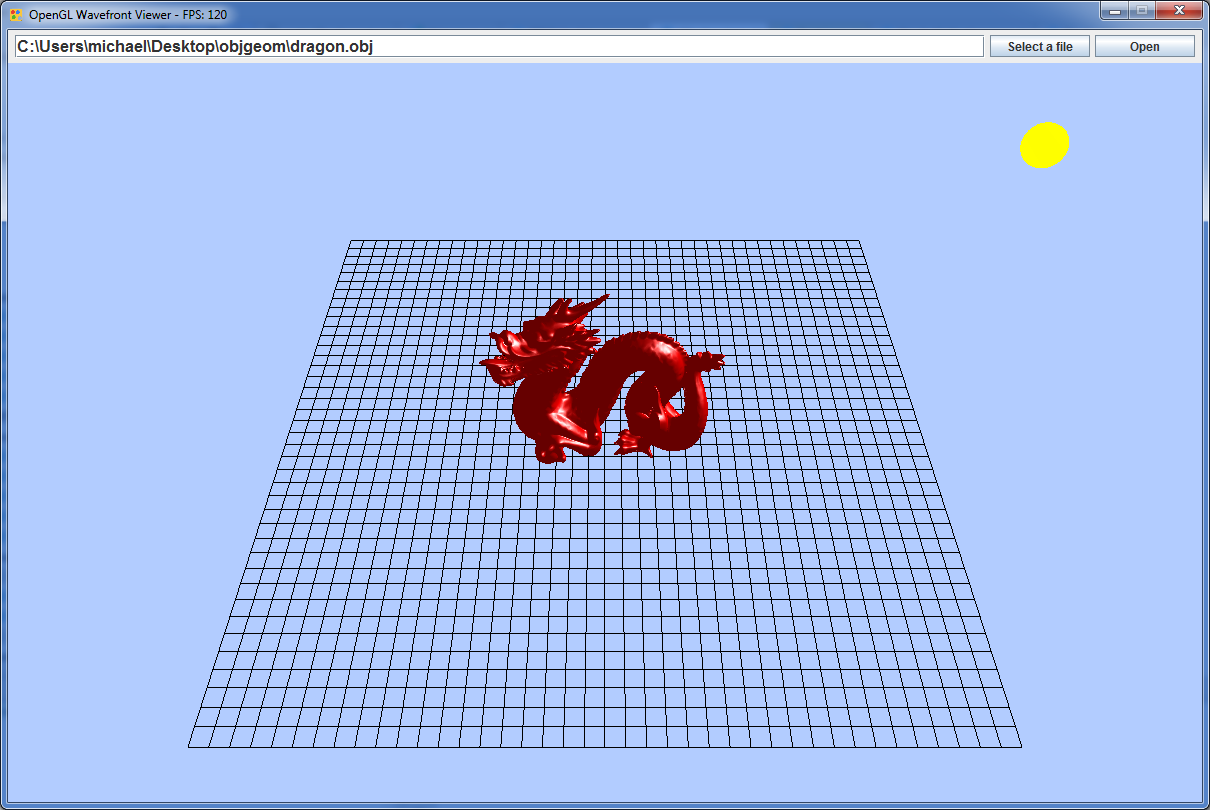


Abbildung 2 Skizze des User Interfaces

# Ist-Zeitplan

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Tätigkeit** | **Prio.** | **KW 49** | **KW 50** | **KW 51** | **KW 52** | **KW 53** | **KW 01** | **KW 02** | **KW 03** |
| **30.11** | **7.12** | **14.12** | **21.12** | **28.12** | **4.1** | **11.1** | **18.1** |
| Konzept erstellen | Muss |  |  |  |  |  |  |  |  |
| Entwicklungsumgebung einrichten | Muss |  |  |  |  |  |  |  |  |
| Realisierung: DisplayManager und GUI erstellen | Muss |  |  |  |  |  |  |  |  |
| Realisierung: .obj File loader implementieren | Muss |  |  |  |  |  |  |  |  |
| Realisierung: Logik für VAO und VBO implementieren | Muss |  |  |  |  |  |  |  |  |
| Realisierung: GLSL Shaders initialisieren | Muss |  |  |  |  |  |  |  |  |
| Realisierung: Pointlight Ambient, Diffuse, Spekular. (Phong Shading) umsetzen | Kann |  |  |  |  |  |  |  |  |
| Realisierung: Projektion und View-Matrix generieren | Muss |  |  |  |  |  |  |  |  |
| Realisierung: Maus-Rotation implementieren | Muss |  |  |  |  |  |  |  |  |
| Finalisierung der Dokumentation | Muss |  |  |  |  |  |  |  |  |
| Präsentation und Abgabe | Muss |  |  |  |  |  |  |  |  |

Tabelle 2 Zeitplan

# Referenzen

**Wavefront OBJ**  
*Wikipedia, Stand 18.11.2015*[*https://de.wikipedia.org/wiki/Wavefront\_OBJ*](https://de.wikipedia.org/wiki/Wavefront_OBJ) 3

**GLSL**  
*OpenGL Webseite, Stand 21.11.2015*[*https://www.opengl.org/documentation/glsl/*](https://www.opengl.org/documentation/glsl/) 3

**Lightweight Java Game Library 3 (LWJGL)***LWJGL Webseite, Stand 21.11.2015*[*https://www.lwjgl.org*](https://www.lwjgl.org) 3

**OpenGL***OpenGL Webseite, Stand 21.11.2015*[*https://www.opengl.org*](https://www.opengl.org) 3

**LWJGL Grundstruktur***Shader/Renderer*[*https://www.opengl.org*](https://www.opengl.org) 3

**OBJReader***Wavefront Reader*[*https://www.opengl.org*](https://www.opengl.org) 3

**Matrizen***Projection/Transformation/View*[*https://www.opengl.org*](https://www.opengl.org) 3

# Versionskontrolle

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Datum** | **Beschreibung** | **Autor** |
| 0.1 | 09.11.2015 | Dokument erstellt | Michael Koch |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |