|  |
| --- |
|  |
| 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 |

Inhaltsverzeichnis

[1 Einleitung 3](#_Toc440719135)

[2 Vorgehen 3](#_Toc440719136)

[2.1 Durchgeführte Tasks 3](#_Toc440719137)

[2.2 Zeitplan (Soll-Ist-Vergleich) 4](#_Toc440719138)

[2.3 Eingesetzte Tools & Technologien 4](#_Toc440719139)

[3 Angewendete Methoden & Konzepte 5](#_Toc440719140)

[3.1 Übersicht 5](#_Toc440719141)

[3.2 Vertex Array Object (VAO) und Vertex Buffer Object (VBO) 5](#_Toc440719142)

[3.2.1 Erläuterung des Konzepts 5](#_Toc440719143)

[3.2.2 Einsatz des Konzepts in der realisierten Anwendung 5](#_Toc440719144)

[3.3 Projektion-, Transformation- und View Matrix: 6](#_Toc440719145)

[3.3.1 Erläuterung des Konzepts 6](#_Toc440719146)

[3.3.2 Einsatz des Konzepts in der realisierten Anwendung 6](#_Toc440719147)

[3.4 Shader 6](#_Toc440719148)

[3.4.1 Erläuterung des Konzepts 6](#_Toc440719149)

[3.4.2 Einsatz des Konzepts in der realisierten Anwendung 6](#_Toc440719150)

[3.5 Phong Reflection Model 6](#_Toc440719151)

[3.5.1 Erläuterung des Konzepts 6](#_Toc440719152)

[3.5.2 Einsatz des Konzepts in der realisierten Anwendung 7](#_Toc440719153)

[3.6 Objekt- und Mausrotation 9](#_Toc440719154)

[3.7 Gitternetz 11](#_Toc440719155)

[4 Ergebnis 12](#_Toc440719156)

[4.1 User-Interface 12](#_Toc440719157)

[4.2 Anforderungsabdeckung der realisierten Anwendung 12](#_Toc440719158)

[5 Persönliches Fazit der Autoren 13](#_Toc440719159)

[5.1 Persönliches Fazit von Joel Holzer 13](#_Toc440719160)

[5.2 Persönliches Fazit von Michael Koch 13](#_Toc440719161)

[6 Tabellenverzeichnis 14](#_Toc440719162)

[7 Abbildungsverzeichnis 14](#_Toc440719163)

[8 Literaturverzeichnis 14](#_Toc440719164)

[9 Versionskontrolle 14](#_Toc440719165)

# Einleitung

Im Rahmen des Moduls „Objektorientierte Geometrie“ an der Berner Fachhochschule erstellten die Autoren eine Java Applikation, welche die Darstellung einer 3D-Figur ermöglicht. Der Benutzer der Applikation kann diese 3D-Figur von unterschiedlichen Seiten betrachten.

Die erstellte Applikation beinhaltet ein User-Interface, über welches der Benutzer eine Wavefront Obj-Datei angeben kann, welche von der Applikation dargestellt werden soll. Wavefront OBJ ist ein offenes Dateiformat zum Speichern von dreidimensionalen geometrischen Formen [1][[1]](#footnote-1). Neben der geladenen 3D-Figur beinhaltet die Szenerie der Applikation zudem eine Lichtquelle (Sonne), welche die geladene 3D-Figur beleuchtet.

Der Benutzer kann die ganze Szenerie (3D-Figur inkl. Lichtquelle), via Maus-Interaktion, von allen Seiten betrachten. Zudem kann er mit dem Mausrad rein und raus zoomen und so die 3D-Figur grösser oder kleiner darstellen lassen.

Zur Modellierung der 3D-Figuren verwendeten die Autoren OpenGL [3] und die Shadersprache GLSL [2]. Die konzeptionelle Umsetzung gelang durch den Einsatz einer Projektions- und Viewmatrix. Die Projektionsmatrix lässt weit entfernte Objekte (in Richtung Z-Achse) kleiner darstellen und nahe bei der Kamera liegende Objekte grösser. Die Viewmatrix ihrerseits beinhaltet die aktuelle Position der Kamera und ermöglicht so die Bewegung der Kamera. Zur Berechnung der Beleuchtung (ambientes, diffuses und spekulares Licht) wurde auf das Phong Reflection Model zurückgegriffen.

Dieser Bericht soll dem Leser einen Einblick in die Umsetzung der genannten Applikation ermöglichen. Zuerst folgt einer Erläuterung des Vorgehens (Tasks) und der eingesetzten Tools & Technologien. Im Hauptteil gehen die Autoren detailliert auf die angewendeten Methoden und Konzepte ein und erläutern deren Zweck und die Art der Umsetzung in der realisierten Applikation. Danach folgt eine Betrachtung des Ergebnisses und zu guter Letzt wird der Bericht mit einem persönlichen Fazit der Autoren abgeschlossen.

# Vorgehen

## Durchgeführte Tasks

Nachfolgende Auflistung gibt die während der Umsetzung des Projekts anfallenden Tasks wieder, begonnen beim ersten Task. Beide Autoren (Studierende) waren 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

## Zeitplan (Soll-Ist-Vergleich)

Nachfolgender Zeitplan zeigt den Vergleich zwischen den geplanten Umsetzungsterminen (Soll) und den effektiven Umsetzungsterminen (Ist) der durchgeführten Tasks. Die Soll-Felder sind Blau markiert, während die Ist-Felder Grün markiert sind.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **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** |
| 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 1 Zeitplan (Soll-Ist-Vergleich)

## Eingesetzte Tools & Technologien

Nachfolgende Tabelle zeigt die bei der Umsetzung der Applikation zum Einsatz kommenden Tools und Technologien:

|  |  |
| --- | --- |
| Anwendungsbereich | Tool oder Technologie |
| Umsetzung des User Interfaces (GUI) | Java mit Swing und AWT |
| Umsetzung des Wavefront .Obj File-Loader | Java |
| 3D-Modelierung | OpenGL mit GLSL (Programmable Pipeline) |
| OpenGL Toolkit | LWJGL [4] |
| Build Management Tool | Maven |
| Entwicklungsumgebung | IntelliJ IDEA |
| Versionsverwaltung Programmcode | GitHub |
| Betriebssystem Entwicklungscomputer | Windows 7 |

Tabelle 2 Eingesetzte Tools & Technologien

# Angewendete Methoden & Konzepte

## Übersicht

Für die Realisierung des „Open GL 3D Model Viewer“ kamen verschiedenen Methoden und Konzepte aus der Grafikprogrammierung und der Mathematik zum Einsatz. Dieses Kapitel soll einerseits diese Methoden und Konzepte benennen, andererseits den Leser in die jeweiligen Thematiken einführen und auch aufzeigen, wie die Methode oder das Konzept in der Applikation umgesetzt wurde.

## Vertex Array Object (VAO) und Vertex Buffer Object (VBO)

### Erläuterung des Konzepts

Um mit OpenGL 3D-Objekte darzustellen, sind Vertices (Deutsch: Knoten), aber auch Normalenvektoren, nötig. Vertices verfügen über eine Position (x-, y- und z-Koordinate) und eine Farbe. In OpenGL gibt es nun 2 Varianten, wie die Daten der Vertices zur Darstellung auf die Grafikkarte gelangen. Die einfachere Variante, wie sie auch in den meisten Open-GL-Tutorials für Einsteiger gezeigt wird, ist es, die Vertexdaten im Hauptspeicher der CPU abzulegen. Das Problem dabei ist jedoch, dass die Grafikkarte deutlich langsamer auf den Hauptspeicher zugreifen kann als auf anderen Komponenten, wie beispielsweise das eigene VRAM. Durch die langsame Zugriffszeit kann es bei grösseren OpenGL Anwendungen zu Performance-Einbussen kommen, weil zwischen der CPU und der GPU ein Flaschenhals existiert.

Die zweite Variante ist der Einsatz von Vertex Array Object (VAO) und Vertex Buffer Object (VBO). Diese beiden Konzepte ermöglichen, dass die Vertexdaten im VRAM der Grafikkarte gespeichert werden, statt wie ohne Einsatz von VAO und VBO im Hauptspeicher der CPU. Da die Grafikkarte deutlich schneller auf das VRAM zugreift als auf den Hauptspeicher, kann der genannte Flaschenhals zwischen CPU und der GPU eliminiert und so die Performance der Anwendung verbessert werden.

### Einsatz des Konzepts in der realisierten Anwendung

Die Konzepte von VAO und VBO kamen in der umgesetzten Anwendung zum Einsatz und führen dazu, dass die Applikation sehr performant arbeitet (120 Frames per Seconds). Ohne den Einsatz von VAO und VBO wären wohl nur 80-90 Frames per Seconds möglich gewesen.

Das VBO enthält die eigentlichen Daten: die Positionen, die Normalenvektoren und die Farben. Das VAO hingegen enthält die Informationen, in welchem VBO sich die benötigten Daten befinden und in welchem Format sie vorliegen.

Das in der Applikation verwendete VAO besitzt folgende Form:

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

Tabelle 3 Form des in der Applikation zum Einsatz kommenden VAO

## Projektion-, Transformation- und View Matrix:

### Erläuterung des Konzepts

Die Projektionsmatrix lässt weit entfernte Objekte (in Richtung Z-Achse) kleiner darstellen und nahe bei der Kamera liegende Objekte grösser [5]. Für die Bewegung der Kamera ist der Einsatz einer Viewmatrix nötig. Die Transformationsmatrix enthält die Translation, Rotation um alle Achsen und die Streckung mit der Objekte manipuliert werden können.

### Einsatz des Konzepts in der realisierten Anwendung

Die genannten Matrizen kommen im Vertex Shader zur Anwendung um die endgültige Position eines Knotens zu finden. Weitere Infos zu den umgesetzten Shaders (u.a. Vertex Shader) sind im Kapitel 3.4 zu finden. Die endgültige Position eines Knotens berechnet sich wie folgt:

Position des Knotens = projectionMatrix \* viewMatrix \* transformationMatrix

## Shader

### Erläuterung des Konzepts

GLSL ist eine C-ähnliche Programmiersprache, welche das Schreiben und Ausführen von eigenen Shaders auf der GPU ermöglicht [6]. In der Grafikprogrammierung wird unter dem Begriff „Shading“ die Veränderung von Knoten und Fragmenten innerhalb der Grafikpipeline verstanden. Ein Shader hat also die Aufgabe, das Aussehen eines Objektes (Textur/Muster) zu berechnen oder Spezialeffekte, wie beispielsweise eine Beleuchtung oder einen Schatten, zu erstellen.

### Einsatz des Konzepts in der realisierten Anwendung

In der realisierten Anwendung wurden 2 verschiedene Shaders erstellt, der Fragment-Shader und der Vertex-Shader. Der Vertex-Shader wird für jeden Vertex (Knoten) aufgerufen und dient dazu, die Geometrie einer Szene zu bestimmen (z.B. Veränderung der Form von Objekten).

Der Fragment-Shader bestimmt den Farbwert pro Fragment/Pixel (in der umgesetzten Anwendung wird damit der Farbwert der geladenen 3D-Figur oder der Farbwert des Gitternetzes festgelegt).

GLSL kam auch bei der Umsetzung des Phong Reflection Models (Beleuchtung der 3D-Figur) zum Einsatz. Nähere Erläuterungen dazu sind im Kapitel 3.5 zu finden.

## Phong Reflection Model

### Erläuterung des Konzepts

Das Phong-Beleuchtungsmodell (Englisch „Phong Reflection Model“) dient zur Berechnung der Beleuchtung von Objekten [7]. In der umgesetzten Anwendung kommt das Phong Reflection Model zum Einsatz um die Beleuchtung der mit dem Obj-Reader geladenen 3D-Figur zu berechnen. Die geladene 3D-Figur wird dabei von einem Punktlicht (Sonne) beleuchtet. Das Phong-Beleuchtungsmodell setzt sich aus den drei Einzelkomponenten Ambient-, Diffuse- und Spekular-Licht zusammen:

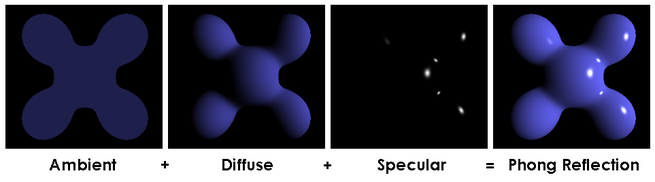


Abbildung 1: Phong Reflection Model (Ambient-, Diffuse- und Spekular-Licht)

Die Ambient-Komponente des reflektierten Lichts ist abhängig von der Intensität des Umgebungslichts, jedoch unabhängig vom Einfallswinkel des Lichtstrahls und vom Blickwinkel des Beobachters, d.h. in unserem Fall vom Blickwinkel der Kamera.

Die Diffuse-Komponente ist eine Reflektion in alle Richtungen, welche auch unabhängig vom Standpunkt des Beobachters (Kamera) ist. Im Gegensatz zur Ambient-Komponente hängt die Diffuse-Komponente jedoch vom Einfallswinkel der Punktlichtquelle (Sonne) ab.

Bei der Spekular-Komponente handelt es sich um die spiegelnde Komponente. Diese ist vom Einfallswinkel des Lichtstrahls (Punktlichtquelle), vom Reflektionsfaktor (Materialkonstante), sowie vom Blickwinkel der Kamera abhängig.

### Einsatz des Konzepts in der realisierten Anwendung

Im Vertex-Shader wird die Position des aktuellen Knotens bestimmt und dieser zusammen mit dem Normalenvektor, dem Lichtvektor, dem Kameravektor und der Farbe an den Fragment-Shader weitergegeben.

Nachfolgende Codeausschnitte zeigen die Implementation des Vertex-Shaders und des Fragment-Shaders.

**//VERTEX SHADER**

#version 400 core  
  
//Input Variablen  
in vec3 position;  
in vec3 normal;  
in vec3 colour;  
  
//Output Variablen  
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()  
{  
 //Berechnet die Koordinaten der Welt  
 vec4 transpos = transformation \* vec4(position, 1.0f);  
 gl\_Position = projection \* view \* transpos;  
  
 //Weitergabe der Daten an den Fragment-Shader  
 v\_color = colour;  
 v\_normal = (transformation \* vec4(normal, 1.0)).xyz; // 0.0  
  
 //Vektor-Model für Licht  
 v\_toLight = light - position.xyz;  
  
 //Vektor-Model für die Kamera  
 v\_toCamera = (inverse(view) \* vec4(0.0,0.0,0.0,1.0)).xyz - transpos.xyz;  
}

**//FRAGMENT SHADER**

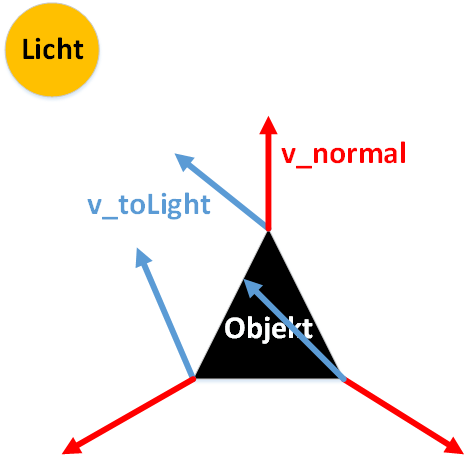
#version 400 core

//Input Variablen  
in vec3 v\_toLight;  
in vec3 v\_toCamera;  
in vec3 v\_normal;  
in vec3 v\_color;

//Uniforms  
uniform vec3 lightColor;  
  
void main()  
{  
 //Normalisiert alle Vektoren  
 vec3 normNormal = normalize(v\_normal); //normalize: vec länge auf 1 setzen  
 vec3 normToLight = normalize(v\_toLight);  
 vec3 normToCamera = normalize(v\_toCamera);  
  
 //Berechnung des diffusen Anteils  
 float diffFactor = dot(normNormal, normToLight);  
 float brightness = max(diffFactor, 0.4); //Ambient Anteil = 0.4  
 vec3 diffuse = brightness \* lightColor;  
  
 //Berechnung des spekularen Anteils  
 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;  
  
 //Phong Reflection: Ambient, diffuse und specular  
 gl\_FragColor = (vec4(diffuse, 1.0) \* vec4(v\_color, 1.0)) + vec4(specular, 1.0);  
}

**Berechnung von Ambient- und Diffuse-Licht**

Die Grösse des Winkels zwischen dem toLight-Vektor und dem Normalen-Vektor bestimmt, wie hoch der spekulare Anteil an diesem Punkt ist. Mithilfe der Max-Funktion von GLSL werden negative Zahlen verhindert und die Mindest-Helligkeit, der Ambient-Anteil, festgelegt.



***Skalarprodukt:***

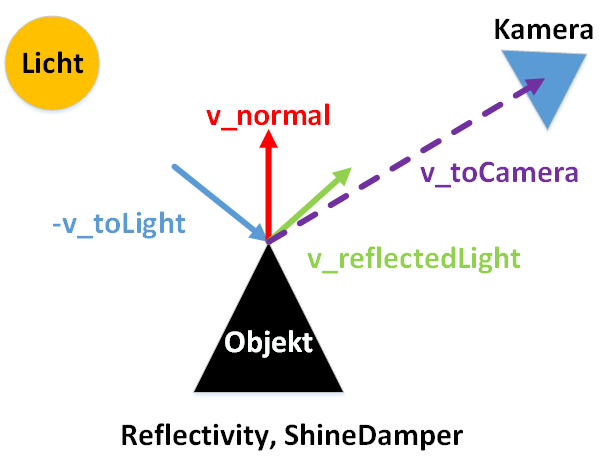
diffuseFactor = dot(v\_normal, v\_toLight);

***Ambient (Max-Funktion):***

brightness = max(diffuseFactor , ambient);  
diffuse = brightness \* lightColor;

Abbildung 2: Berechnung von Ambient-   
und Diffuse-Licht

**Berechnung des spekularen Lichts**

Zunächst wird der negative toLight-Vektor am Normalenvektor gespiegelt um den reflectedLight-Vektor zu erhalten. Der spekulare Anteil ergibt sich nun aus dem Skalarprodukt des reflectedLight-Vektors mit dem toCamera-Vektor.

***Reflektion:***

v\_reflectedLight = reflect(-v\_toLight, v\_normal);

***Skalarprodukt:***

specularFactor = dot(v\_reflectedLight , v\_toCamera);

***ShineDamper und Reflectivity:***

dampedFactor = pow(specularFactor, shineDamper);

finalSpecular = dampedFactor \* reflectivity \* lightColor;

Abbildung 3: Berechnung des spekularen Lichts

## Objekt- und Mausrotation

Die geladenen 3D-Objekte werden auf einem Boden (Gitternetz) platziert. 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 mit Rechts- und Linksverschiebung der Maus: Y-Rotation der Kamera (pitch)
3. Links Klick mit Hoch- und Runterverschiebung der Maus: Winkel um das Objekt

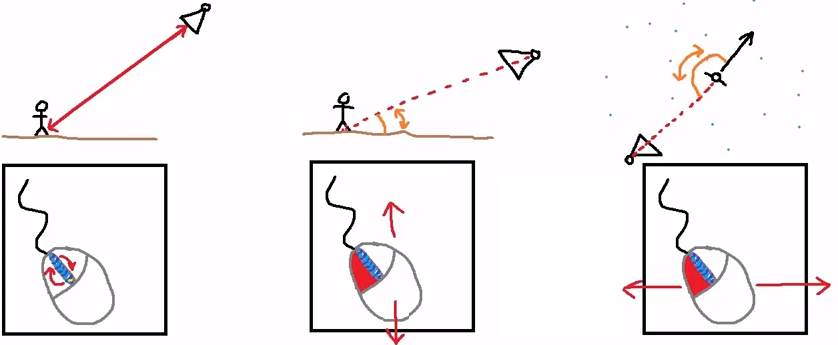


Abbildung 4: Objekt- und Mausrotation

Folgende Schnitte waren notwendig, um die Position der Kamera und deren Rotation zu bestimmen:

**1. horizontale und vertikale Distanz**

Als erstes wird die horizontale- und vertikale Distanz zwischen Kamera und Objekt mit Hilfe der Winkelfunktionen Sinus und Cosinus berechnet:

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

**2. X-Position und Z-Position der Kamera**

Im nächsten Schritt wird die x-Position und Z-Position ebenfalls mit Hilfe der Winkelfunktionen Sinus und Cosinus berechnet.

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

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

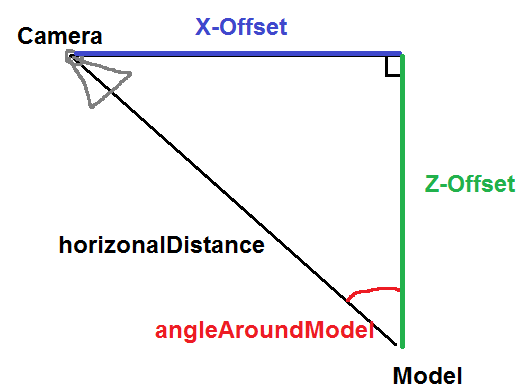


Abbildung 5: Berechnung der X- und Z-Position der Kamera

**3. Y-Rotation der Kamera (yaw)**

Abschliessend wird noch die Kamerarotation (Neigung der Kamera) berechnet.

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

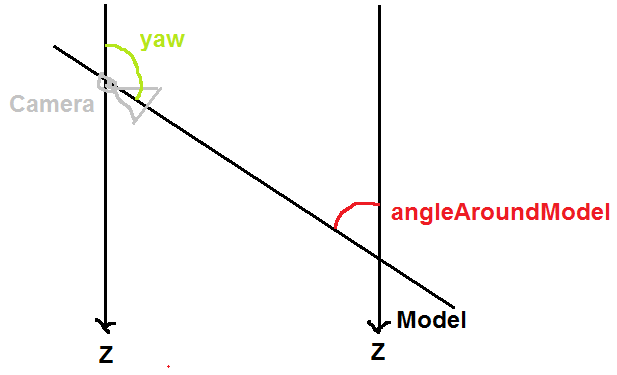


Abbildung 6: Berechnung der Y-Rotation der Kamera

## Gitternetz

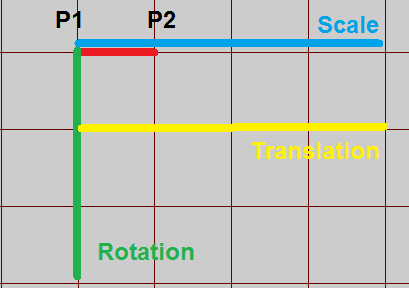
Bereits im vorhergehenden Kapitel wurde erwähnt, dass sich in der Szenerie ein Gitternetz befindet, auf welchem die geladene 3D-Figur platziert wird. Dieses Gitternetz dreht sich bei der Mausrotation mit dem Objekt mit, d.h. die ganze Szenerie dreht sich.

Um eine möglichst gute Performance der Anwendung zu erreichen, wurde die Erstellung eines Gitternetzes mit möglichst wenig Knoten angestrebt. Das Gitternetz wurde schlussendlich mit 2 Knoten, welche sich im Speicher befinden, realisiert. Diese 2 Knoten sind:

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

Jeder Geradenabschnitt referenziert P1 und P2 (Anfang und Ende einer Linie). Um die gewünschte Linie zu erhalten, transformiert die Anwendung die beiden Knoten (Translation, Rotation, Streckung/Stauchung). Für jede gewünschte Linie des Gitternetzes führt die Anwendung die gewünschte Transformation aus, was schlussendlich zum Erhalt des Gitternetzes führt.

Die nachfolgende Abbildung zeigt die Skalierung, Translation und Rotation.



**Skalierung**: Streckung/Stauchung der Linie   
auf die gewünschte Länge.

**Rotation**: Drehung der Linie. Gitternetz   
besteht aus vertikalen und horizontalen   
Linien.

**Translation**: parallele Verschiebung einer   
Linie.

Abbildung 7: Skalierung, Rotation und Translation bei  
der Umsetzung des Gitternetzes

# Ergebnis

## User-Interface

Nachfolgende Abbildung zeigt das User Interface der umgesetzten Anwendung. Im oberen Bereich kann der Benutzer mit Hilfe eines „Java-FileOpenDialogs“ ein Wavefront OBJ-File auswählen und dieses in die Anwendung laden. Der Inhalt des ausgewählten ObJ-Files wird dann von der Anwendung im unteren Bereich als 3D-Objekt angezeigt. Das geladene Objekt wird auf einem Gitternetz platziert. Das Gitternetz ist bereits vorhanden, bevor ein OBJ-File geladen wurde.

Oben rechts ist zudem die Punktlichtquelle, realisiert als gelber Punkt, zu sehen. Diese leuchtet das geladene 3D-Objekt an (im vorhergehenden Kapitel erläuterte Phong Reflection).

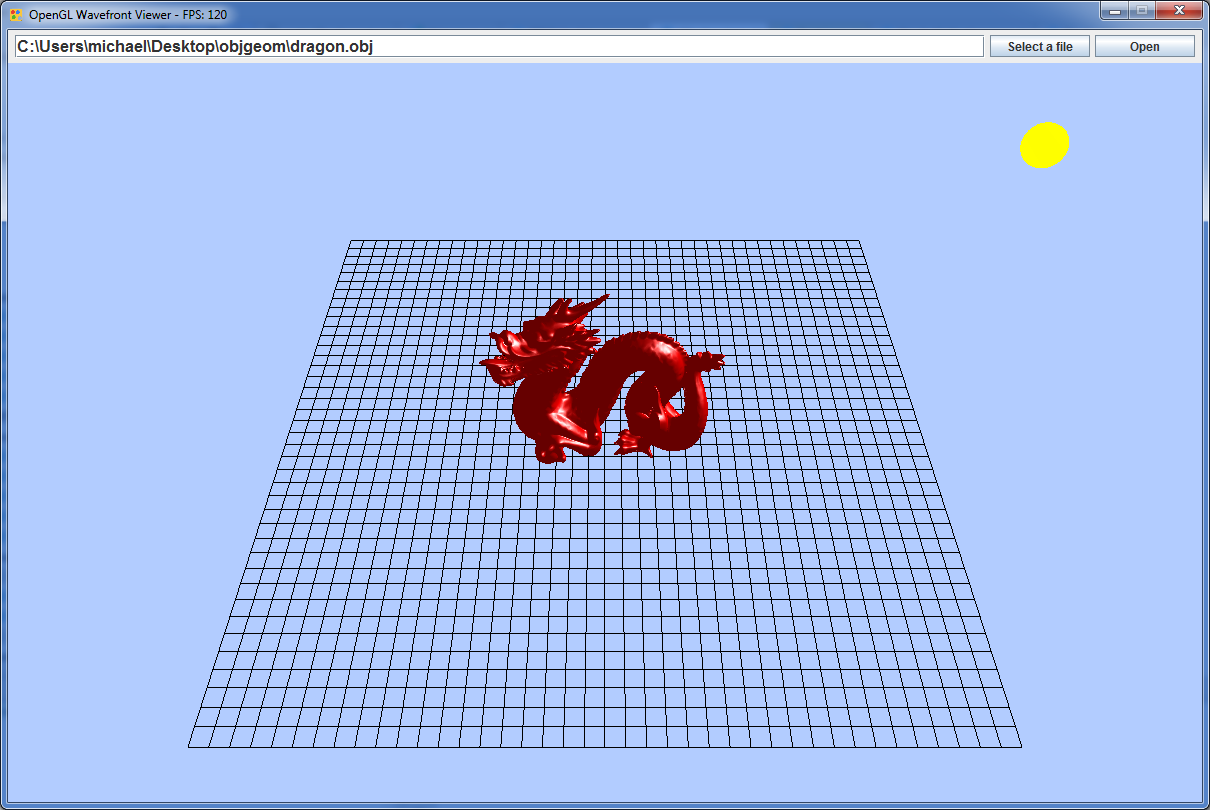


Abbildung 8: Skizze des User Interfaces

## Anforderungsabdeckung der realisierten Anwendung

Die realisierte Anwendung deckt alle im Konzept definierten Anforderungen ab. Auch, das im Konzept als erweiterte Anforderung (Kann-Ziel) definierte Punktlicht mit Phong Shading, wurde umgesetzt.

Die Anwendung wurde mit diversen, mit Hilfe von Blender erstellten, Wavefront OBJ-Files getestet und hat alle Tests mit Bravour bestanden.

# Persönliches Fazit der Autoren

## Persönliches Fazit von Joel Holzer

Für mich war die Entwicklung einer OpenGL Anwendung in dieser Grössenordnung eine neue, sehr lehrreiche Erfahrung, welche ich auf keinen Fall missen möchte.

Vor der Realisierung dieser Anwendung hatte ich nur wenig Erfahrung mit OpenGL, welche ich mir bei der Umsetzung einiger Tutorials angeeignet hatte. Durch die Zusammenarbeit mit Michael Koch, welcher dank der CPVR-Vertiefung bereits einen grossen OpenGL-Erfahrungsschatz mitbrachte, konnte ich jedoch viele neue Konzepte und Methoden bei der OpenGL-Programmierung kennen und anwenden lernen. So lernte ich beispielsweise die beiden Konzepte „VAO“ und „VBO“ zur performanteren Programmierung kennen. Auch das Konzept von Phong Shading und deren Anwendung in der eigenen Applikation ist ein wichtiger Punkt, welcher ich aus diesem Projekt mitnehmen werde.

Ich hoffe, dass ich auch in Zukunft (nach meinem Studium) ab und zu dazu kommen werde, die gewonnenen Erkenntnisse in eigenen OpenGL-Projekten anzuwenden. Vielleicht wird es sich ja auch mal ergeben, dass ich OpenGL mit Arbeiten meiner Vertiefung (Mobile Computing), beispielsweise bei der Entwicklung eines Games für Android, kombinieren kann.

## Persönliches Fazit von Michael Koch

# Tabellenverzeichnis

[Tabelle 1 Zeitplan (Soll-Ist-Vergleich) 4](#_Toc440719166)

[Tabelle 2 Eingesetzte Tools & Technologien 4](#_Toc440719167)

[Tabelle 3 Form des in der Applikation zum Einsatz kommenden VAO 5](#_Toc440719168)

# Abbildungsverzeichnis

Abbildung 1: Phong Reflection Model (Ambient-, Diffuse- und Spekular-Licht) 6

Abbildung 2: Berechnung von Ambient- und Diffuse-Licht 8

Abbildung 3: Berechnung des spekularen Lichts 9

Abbildung 4: Objekt- und Mausrotation 9

Abbildung 5: Berechnung der X- und Z-Position der Kamera 10

Abbildung 6: Berechnung der Y-Rotation der Kamera 10

Abbildung 7: Skalierung, Rotation und Translation bei der Umsetzung des Gitternetzes 11

Abbildung 8: Skizze des User Interfaces 12

# Literaturverzeichnis

1. **Wavefront OBJ**  
   *Wikipedia, Stand 18.11.2015*[*https://de.wikipedia.org/wiki/Wavefront\_OBJ*](https://de.wikipedia.org/wiki/Wavefront_OBJ) 3
2. **GLSL**  
   *OpenGL Webseite, Stand 21.11.2015*[*https://www.opengl.org/documentation/glsl/*](https://www.opengl.org/documentation/glsl/) 3
3. **OpenGL***OpenGL Webseite, Stand 21.11.2015*[*https://www.opengl.org*](https://www.opengl.org) 3
4. **Lightweight Java Game Library 3 (LWJGL)***LWJGL Webseite, Stand 21.11.2015*[*https://www.lwjgl.org*](https://www.lwjgl.org) 4
5. **Matrizen***LWJGL-Wiki, Stand 16.01.2016*[*http://wiki.lwjgl.org/wiki*](http://wiki.lwjgl.org/wiki) 6
6. **Shader / Renderer***LWJGL-Wiki, Stand 16.01.2016*[*http://wiki.lwjgl.org/wiki*](http://wiki.lwjgl.org/wiki) 6
7. **Phong Reflection Model**Wikipedia, Stand 16.01.2016  
   [*https://de.wikipedia.org/wiki/Phong-Beleuchtungsmodell*](https://de.wikipedia.org/wiki/Phong-Beleuchtungsmodell) 6

# Versionskontrolle

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Datum** | **Beschreibung** | **Autor** |
| 1.0 | 18.01.2016 | Dokument finalisiert und Version 1.0 freigegeben. | Michael Koch, Joel Holzer |

1. Der Quellenverweis [1] und alle weiteren sind im Literaturverzeichnis nachzuschlagen [↑](#footnote-ref-1)