Projektbericht

Modeler 2018

Fachhochschule Bielefeld
Campus Minden
Studiengang Informatik

Beteiligte Personen:

Name	Matrikelnummer
Johannes Ens	1047037
Daniel Sprick	1078398
Martin Ziel	1078228
Eduard Ljaschenko	1027656

---

22. Juli 2018

Inhaltsverzeichnis

1 EINLEITUNG

Die in diesem Dokument beschriebene Software, ist eine Modellier-Software, die dazu imstande ist Körper und Flächen zu erstellen. Außerdem lassen sich die erstellten Modelle als OBJ-Datei speichern. Gespeicherte Modelle können geladen und auch bearbeitet werden.
Zur Erstellung der Software wurde die Programmiersprache C/C++ genutzt unter Verwendung von OpenGL und ist mit Linux als auch mit Windows kompatibel.

2 BEDIENUNG

OBJ-Datei laden:	**File** → **Open** oder **Strg + O**
Neues Objekt:	**File** → **New** oder **Strg + N** hh
Objekt als OBJ speichern:	**File** → **Save** oder **Strg + S**
Kamera Position verändern:	'**w**', '**a**', '**s**', '**d**', '**q**', '**e**' - Tasten
Kameraausrichtung ändern:	**linke Maustaste** + bewegen der Maus
Raster de/aktivieren:	'**g**' - Taste
Punkte de/aktivieren:	'**p**' - Taste
Kanten de/aktivieren:	'**k**' - Taste
Punkt/e auswählen:	**Strg** + Klick auf Punkt
Neuer Punkt:	**rechte Maustaste** auf Raster oder Objekt
Ausgewählte Punkte löschen:	'**r**' - Taste
Ausgewählte Punkte verschieben:	Pfeiltasten, '**+**' und '**-**'
Erstellung eines Face:	Punkte gegen Uhrzeigersinn auswählen + '**f**' - Taste
Catmull-Clark Unterteilung:	'**c**' - Taste je Schritt

Unterteilung zurücksetzen: 'Strg + Z

Gewichtung der ausgewählte Punkte: Vertex → VertexWeight

Smoothing: 'l' - Taste je Schritt

3 SHADER

Das Programm verwendet einen **Vertex Shader** und einen **Fragment Shader** für die Umsetzung des **Phong-shadings**. In die Beleuchtungs-Berechnung fließen die Position der primären Lichtquelle, die Farbe dieser Lichtquelle sowie weitere Faktoren wie die Farbe und Stärke des Umgebungslichtes und die Eigenschaften des Materials hinsichtlich der Spiegelungen. In Abhängigkeit des Betrachtungswinkels wird die Beleuchtung berechnet, indem die Normalen an den Vertices interpoliert werden. Außerdem werden mithilfe der Shader die Vertices als Kreise dargestellt.

Vorhergegangene Versuche zeigten deutlich, dass die Darstellung der Vertex-Punkte nicht effizient gelöst werden kann, wenn jeder Punkt durch ein eigenes VAO repräsentiert wird. Im Zuge dieser Erkenntnis entschloss sich das Team die Punkte darzustellen, indem alle Vertices mithilfe des Parameters GL_POINTS als Point-Cloud gerendert werden. Diese Lösung ist effizient, jedoch ergab sich das Problem, dass die Punkte mit GL_POINTS als Vierecke gerendert wurden. Die Lösung für dieses Problem besteht darin einen Radius zu definieren und alle Fragmente, die sich Außerhalb dieses Radius befinden, nicht zu rendern. Da die Größe dieser Kreise nicht in Raumkoordinatengrößen definiert werden, sondern in Pixelgröße, wurden die Punkte alle gleich groß, unabhängig von der Entfernung, gerendert. Um eine dynamische Größe zu gewährleisten, wird die Entfernung der Kamera zu dem Punkt berechnet und der Radius des Punktes wird proportional zur Entfernung kleiner.

4 KAMERA

Um das Betrachten des Objektes von allen Seiten zu ermöglichen entschied sich das Team eine Lösung zu implementieren die **Quaternionen** nutzt. So wurde eine Computerspiel-artige Steuerung der Kamera entwickelt, welche es ermöglicht die Ausrichtung der Kamera mithilfe der Maus zu beeinflussen und die Position der Kamera mithilfe der Tasten w, a, s, d, q und e zu verändern. Beim Drücken der Linken Maustaste und ziehen in eine Richtung wird eine Achse definiert, welche senkrecht zu der gezogenen Bahn steht. Um diese Achse wird die Ausrichtung der Kamera während der Bewegung rotiert.

5 CATMULL-CLARK UNTERTEILUNG

Beim Catmull-Clark-Algorithmus werden beliebige Polygon-Gitter in Quad-Gitter unterteilt. Jedes n-Gon wird in n-Quads unterteilt. Zunächst wird für jedes Face ein Face-Punkt gesetzt, dieser wird aus dem Durchschnitt der Ausgangspunkte des jeweiligen Faces berechnet. Danach wird für jede Kante ein Kanten-Punkt hinzugefügt, der anhand der jeweiligen Endpunkte und der Nachbar Face Punkte und ebenfalls aus dem Durchschnitt dieser Punkte berechnet wird. Jeder neu berechnete Vertex wird mit den neu berechneten Kanten verbunden und weiter mit den Ausgangsvertices verbunden. Wenn das Ausgangsquad bereits zu einem Quad-Gitter unterteilt wurde, wird es erneut in vier weitere Quads unterteilt. Je öfter man den Algorithmus anwendet, desto feiner wird das Mesh.

5.1 Limitpunkte und Limitnormalen

Bei unendlich ofter Anwendung der Unterteilungsregeln auf ein Mesh wird die Limitfläche erzeugt. Durch Anwendung der Limitpunktregeln lässt sich für jeden Kontrollpunkt eines Kontrollgitters der entsprechende Punkt auf der Limitfläche bestimmen. Die Normale eines Limitpunktes erhält man durch das Kreuzprodukt der Oberflächentangenten an ebendieser Stelle des Limitpunktes.
In der Implementierung wird mit der Berechnung der Limitpunkte begonnen, nachdem man die Kontrollpunkte des neuen Meshes durch einen Unterteilungsschritt erhalten hat. Nun wird für jeden einzelnen Kontrollpunkt die Limitpunktregel angewandt. Dazu wird zunächst überprüft, ob und wie viele scharfe Kanten/Ränder mit dem Kontrollpunkt verbunden sind und entsprechend die Regel für glatte oder scharfe Kanten verwendet. Die Informationen über Valenz k und 1-Nachbarschaft erhält man durch geschicktes Manövrieren innerhalb der Halfedge-Datenstruktur.

6 ARCHITEKTUR

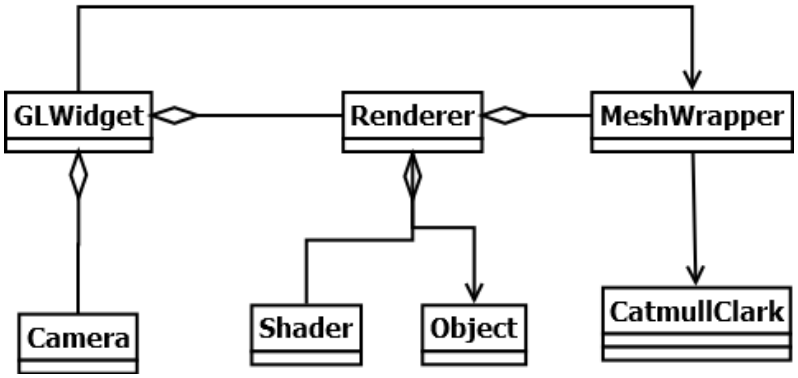

Abbildung 1: Übersicht

6.1 GLWidget

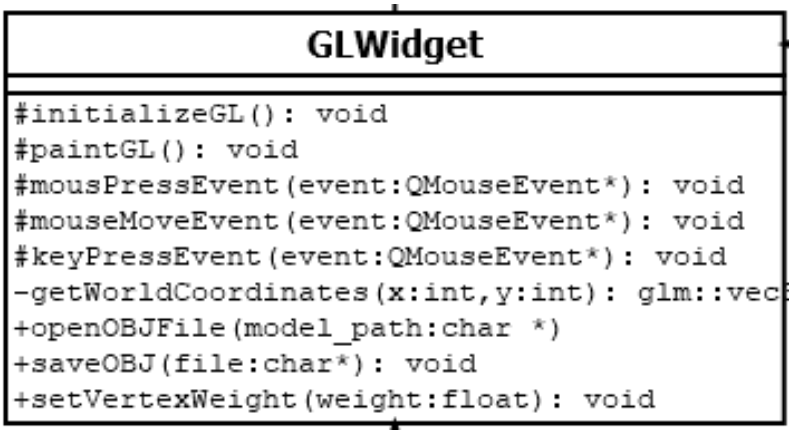

Abbildung 2: GLWidget

Die Klasse **GLWidget** erweitert die QT-Klasee **QGLWidget** und gibt Funktionen wie das Laden einer OBJ-Datei oder das Speichern einer OBJ-Datei, sowie das Setzen der Vertex-Gewichtung an den MeshWrapper weiter.

initializeGL() OpenGL Funktionalitäten werden initialisiert, Kamera wird konfiguriert und Renderer wird mit seinem Shader-Programm geladen.

paintGL() Diese Funktion wird von QT in einer Rendering-Schleife aufgerufen, aktualisiert die Attribute der Kamera und bindet die aktuellen Matrizen, die von dem Shader benötigt werden, an den Shader.

mousePressEvent() Durch das Drücken einer Maustaste ausgelöst, führt die Aktionen **Vertex-Auswahl** und **Vertex-Erzeugung** aus. Beide Aktionen bedienen sich der Funktion **getWorldCoordinates()**.

getWorldCoodinates() implementiert Ray-Picking verfahren mithilfe der Funktionen **gluUnProject()** und **glReadPixels()**. In dem Ray-Picking Verfahren wird ein Strahl durch die 3D-Welt zwischen Near- und Far-Pane projiziert. Die Koordinaten im 3D-Raum an denen zuerst ein gerendertes Objekt gekreuzt wird, werden als Rückgabewert der Funktion **getWorldCoordinates()** zurückgeliefert. Aus diesem Grund bestand die Notwendigkeit eine Fläche in der 3D-Welt zu rendern, die von einem Raster überlagert wird. Diese Fläche ermöglicht es Vertices per Mausklick mittels des Ray-Picking-Verfahren hinzuzufügen. Die Fläche und das Raster können ausgeblendet werden, wenn sie beispielsweise bei der Betrachtung oder der Bearbeitung eines Objektes stören.

keyPressEvent() behandelt weitere Benutzerinteraktionen wie die Bewegung durch den Raum oder die Manipulation des Mesh und gibt diese an die jeweiligen Klassen weiter.

6.2 Camera

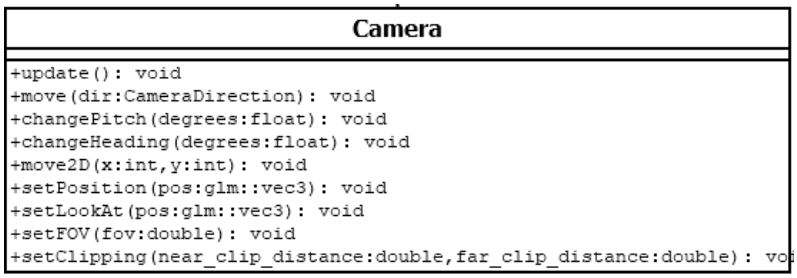

Abbildung 3: Camera

move() verändert die Position der Kamera und definiert die Richtung in die, die Kamera sich bewegt bei den jeweiligen Tasteneingaben.

changePitch() sorgt für die Drehung der Kamera und überprüft den Grad, um die sich die Kamera wenden soll mit einem Maximalwert um so die Geschwindigkeit einzuschränken.

changeHeading() bestimmt die Richtung der Fortbewegung.

move2D() berechnet anhand der vorherigen Mausposition die Drehung und Fortbewegungsrichtung der Kamera durch die Maus.

update() Diese Funktion aktualisiert und adaptiert die Kamera und dessen Attribute wie z. B. die Richtung, Drehung und Bewegung der Kamera.

6.3 Catmull-Clark

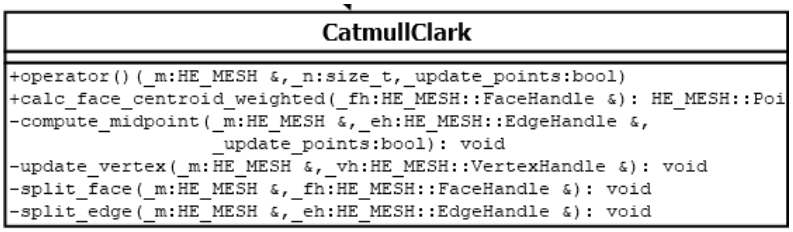

Abbildung 4: Catmull-Clark

operator() Zunächst werden die Gewichte auf 0 gesetzt, wodurch die Kanten geglättet werden. Danach werden n-Unterteilungen durchgeführt und für jedes Face wird die Mitte berechnet. Dann werden die Positionen der neuen Kantenpunkte berechnet. Darauf folgt die mittige Teilung jeder Kante.

calc_face_centroid_weighted() berechnet das Zentrum eines Faces anhand seiner Vertices.

compute_midpoint() berechnet den Mittelpunkt einer Kante anhand der Endpunkte dieser Kante und den Nachbarkanten.

update_vertex()

split_face() Teilt ein n-Gon in n-Quads durch die Verbindung der neu erzeugten Vertices.

split_edge()	erstellt für die Mitte einer Kante einen neuen Vertex, dieser wird mit den Endpunkten neu verbunden.

6.4 Object

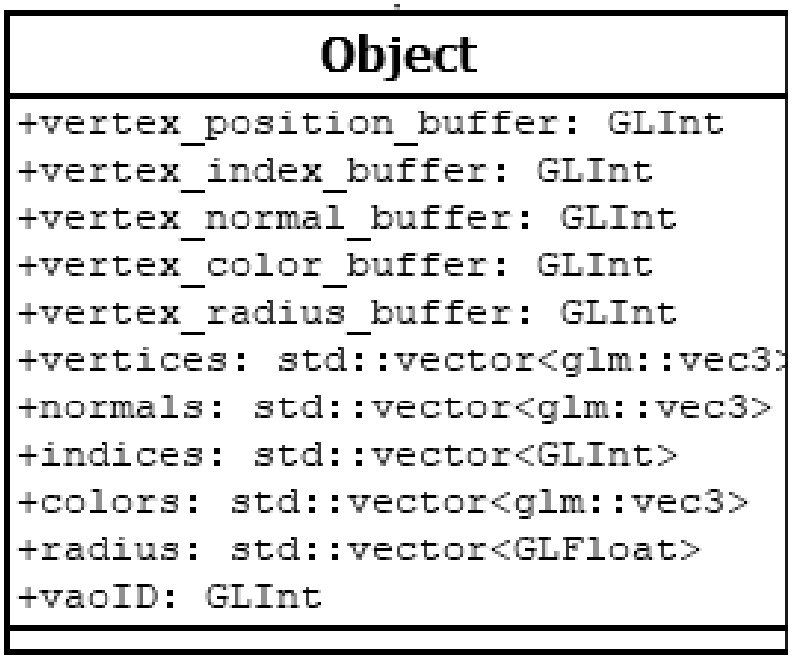

Abbildung 5: Object

Die Klasse **Object** wird im Renderer benötigt. Hier werden alle Daten zu einem Objekt, welches gerendert werden soll, gespeichert. Die Daten bestehen aus den ID´s der unterschiedlichen Buffer-Objects, der ID des Vertex-Array-Objektes, sowie den Daten die dieses Objekt definieren. Der Renderer ist so Implementiert das nicht zwangsläufig alle Attribute des Objekts definiert sein müssen. Beispielsweise wird davon ausgegangen, dass die Vertices in Triangulierter-Form und Reihenfolge im vertex_position_buffer vorliegen, wenn der vertex_index_buffer nicht definiert ist.

6.5 Renderer

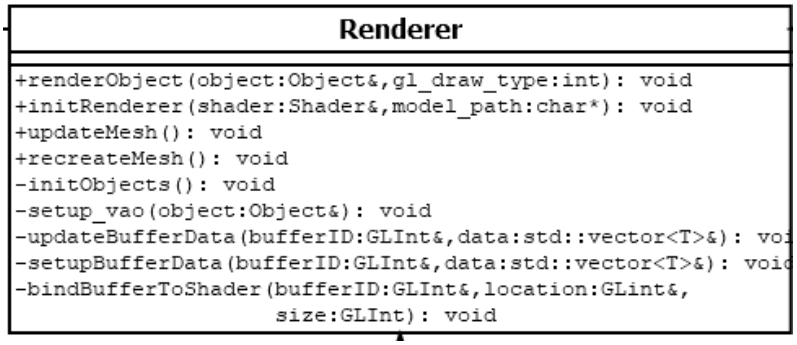

Abbildung 6: Renderer

In dieser Klasse werden alle Operationen ausgeführt die den Rendering-Prozess betreffen. Die Klasse hält verschieden Instanzen der Object Struktur. Das Mesh wird in dreifacher Variation als Object gehalten:

1. für die Darstellung der Flächen
2. für die Darstellung der Punkte
3. für die Darstellung der Linien, die die Kanten der Faces visualisieren.

Außerdem werden Objects für das Raster sowie die Fläche unterhalb des Rasters initialisiert.
Bei der Initialisierung der einzelnen Objekte werden die Daten-Attribute der Object Instanz belegt und für jedes Objekt wird ein VAO erstellt sowie die buffer_objects der Daten-Attribute, die belegt sind, werden erzeugt. Hierfür werden die Template-Funktion **setup_buffer_data()** und **bindBufferToShader()** verwendet. **updateBufferData()** ist eine Template-Funktion, welche **glBufferSubData** nutzt, um die gespeicherten Daten zu aktualisieren. Dies geschieht zum Beispiel, wenn ein Vertex-Punkt verschoben wird. Wird ein Punkt hinzugefügt, entfernt oder ein Face erstellt, wird die Größe des Buffer-Objects verändert. Für diesen Fall eignet sich **glBufferSubData** nicht mehr und die entsprechende Buffer-Objects werden mittels **recreateMesh()** neu erstellt.
Bei der Initialisierung der Renderer Instanz werden die Dimensionen des Mesh-Objektes berücksichtigt. Es werden die minimalen und maximalen Vertex-Positionen ermittelt und das Raster-Objekt wird unterhalb der niedrigsten Position auf der Y-Achse positioniert. Zusätzlich wird das Raster auf eine Größe, die 50% größer als die Differenz aus der Maximalen X und Minimalen X Position oder, wenn diese größer ist, die Differenz auf der Z-Achse, beschränkt.

6.6 Shader

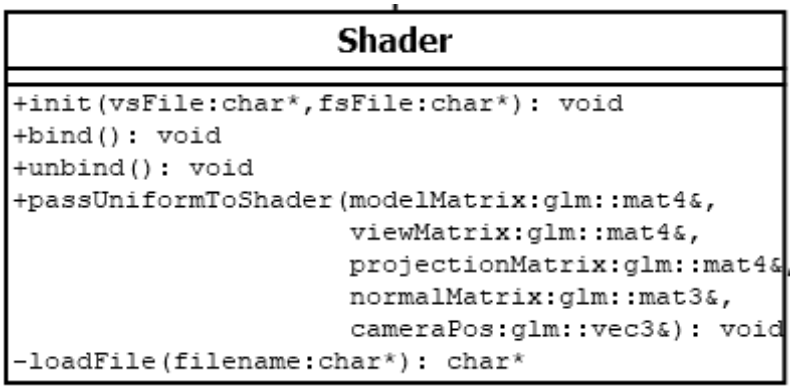

Abbildung 7: Shader

Die Shader-Klasse bindet die übergebenen zwei GLSL-Shader an die Grafikkarte. Hierfür werden die übergebenen Shader ausgelesen und die benötigten Parameter an die Shader weitergeleitet. In unserem Programm beschränken wir uns auf die Verwendung eines **Vertex-Shaders** und eines **Fragment-Shaders**, welche das **Phong-Shading** umsetzen und für die korrekte Darstellung der Punkte zuständig sind.

6.7 MeshWrapper

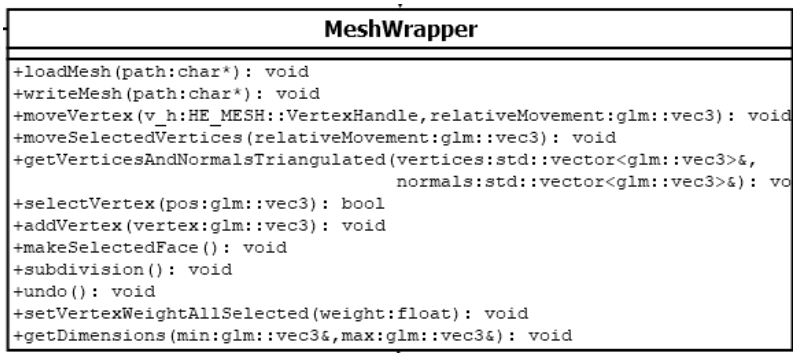

Abbildung 8: Meshwrapper

Der **MeshWrapper** schachtelt die OpenMesh-Funktionalitäten und schafft eine Schnittstelle um die Interaktion mit dem Mesh zu ermöglichen. Für die Konfiguration des Meshes wird ein struct verwendet, welches festlegt, dass die Punkte und Normalen als float gespeichert werden. Somit ist die Integration in die Umgebung, welche vorwiegend mit dem Datentyp float arbeitet gewährleistet. Da für die Gewichtung der Vertices bei der Berechnung der Unterteilungsflächen eine vierte Komponente

benötigt wird, wird die Klasse **HE_MESH** genutzt, welche von der Klasse PolyMesh, aus der OpenMesh-Bibliothek, abgeleitet wird.
Der MeshWrapper verfügt über ein Objekt der Klasse **HE_MESH**. Dieses Objekt repräsentiert das Mesh. Außerdem wird ein std::vector verwendet der die aktuell markierten Vertices speichert. Zudem wird ein Backstack angelegt, der bei jeder Ausführung des Catmull-Clark Unterteilungsflächen-Algorithmus, das Mesh speichert. Dadurch kann jeder Unterteilungsschritt rückgängig gemacht werden.
Für die Verwendung im Renderer liefert die Funktion **getVerticesAndNormalsTriangulated()** eine Triangulierte Version des Meshes. Hierfür werden alle Faces durchlaufen und jeder dem aktuellen Face zugehöriger Vertex wird in einem std::vector gespeichert.
Für die Darstellung der Kanten werden alle Kanten des Meshes durchlaufen und die Anfangs-, sowie End-Vertices in einem std::vector gespeichert.
Um einen Vertex auszuwählen, werden die Welt-Koordinaten die, in der Klasse GLWidget bestimmt wurden an die Funktion **selectVertex()** übergeben. Hier werden nun alle Vertices überprüft, ob diese auf den Koordinaten liegen. In diesem Prozess wird eine kleine Abweichung toleriert. Ist der angeklickte Vertex schon in der Liste, wird er entfernt.

7 ALGORITHMEN

7.1 Kamera

```
1
2 projection = glm::perspective(glm::radians(field_of_view), aspect,
      near_clip, far_clip);
3
4 /*
5 * Achse für Pitch-Rotation festlegen
6 */
7 glm::vec3 axis = glm::cross(camera_direction,glm::vec3(0.0f, 1.0f,
      0.0f););
8
9 /*
10 * Quaternion für Pitch auf Basis des Kamera-Pitchwinkels berechnen
11 * Pitch wird durch Mausbewegung in horizontaler Richtung beeinflusst
12 */
13 glm::quat pitch_quat = glm::angleAxis(camera_pitch, axis);
14
15 /*
16 * Bestimmung des Heading-Quaternions aus dem Kamera-Aufwärtsvektor
      und dem Heading-Winkel
17 * heading wird durch Mausbewegung in vertikaler Richtung beeinflusst
18 */
19 glm::quat heading_quat = glm::angleAxis(camera_heading, glm::vec3
      (0.0f, 1.0f, 0.0f););
20
21 /*
22 * Kreuzprodukt der beiden Quaternionen, normalisieren
23 */
24 glm::quat temp = glm::cross(pitch_quat, heading_quat);
25 temp = glm::normalize(temp);
26
27 /*
28 * die Richtung der Kamera mit der Quaternion aktualisieren
29 */
30 camera_direction = glm::rotate(temp, camera_direction);
31
32 /*
33 * Kamera position aktualisieren
34 */
35 camera_position += camera_position_delta;
```

```
36
37 /*
38 * lookAt aktualisieren
39 */
40 camera_look_at = camera_position + camera_direction * 1.0f;
```

7.2 Smoothing

Die Glättung wird erreicht, indem jeder Vertex an die Position verschoben wird, die dem Durchschnittswert seiner benachbarten Vertices entspricht. Um dies zu erreichen werden die neuen Positionen der Vertices zwischengespeichert. Nachdem jeder Vertex neu errechnet wurde, werden die alten Werte durch die zwischengespeicherten Werte ersetzt.

$$\overline{x}_i = \frac{1}{N} \sum_{j=1}^{N} \overline{x}_j$$

wobei $\overline{x}_i$ die neue Position des Vertex und N die Anzahl der benachbarten Vertices ist.

7.3 Shader

Vertex Shader

```
1  vec4 vertex = vec4(vertex_position, 1.0);
2
3  /*
4  * Transformation der Normale und der Position für den Fragmentshader
5  */
6  normal = normalize(normalMatrix * vertex_normal);
7  position = vec3(viewMatrix * modelMatrix * vertex);
8
9  /*
10 * Grundfarben ändern sich nicht pro Pixel - können jetzt berechnet
       werden
11 */
12 ambient = materialAmbient * lightAmbient;
13 diffuse = materialDiffuse * lightDiffuse;
14 ambientGlobal = materialAmbient * lightGlobal;
15
16 /*
```

```
17 * Dynamische Anpassung der Punktgröße abhängig von der Entfernung
       der Kamera
18 */
19 float distance = length(vertex_position - cameraPos);
20 gl_PointSize = radius_attr/distance;
21
22 /*
23 * Vertex-Position in OpenGL setzen
24 */
25 gl_Position = projectionMatrix * viewMatrix * modelMatrix * vertex;
```

Fragment Shader

```
1  /*
2  * Lichtberechnung
3  */
4  vec3 N = normalize(normal);
5  vec3 L = normalize(lightPosition - position);
6  vec3 R = 2 * dot(L, N) * N - L;
7
8  float cosTheta = max(dot(L, N), 0.0);
9  float cosAlpha = max(dot(N, R), 0.0);
10
11 float attenuation = 1.0 / (constantAttenuation + length(L) *
       linearAttenuation);
12
13 vec3 tempColor = VertexIn.vertex_color + ambientGlobal;
14
15 if (cosTheta &gt; 0.0) {
16 tempColor += attenuation * (diffuse * cosTheta + ambient);
17 tempColor += attenuation
18 * materialSpecular
19 * lightSpecular
20 * pow(cosAlpha, materialShininess);
21 }
22
23 /*
24 * ist das Radius-Attribut gesetzt, handelt es sich um einen Punkt
25 */
26 if(VertexIn.radius &gt; 0){
27 vec3 N;
28 N.xy = gl_PointCoord * 2.0 - vec2(1.0);
```

```
29 float mag = dot(N.xy, N.xy);
30
31 /*
32 * Pixel außerhalb des Kreises werden nicht dargestellt
33 */
34 if (mag &gt; 1.0) discard;
35
36 N.z = sqrt( 1.0 - mag);
37 }
38
39 /*
40 * Pixelfarbe in OpenGL setzen
41 */
42 color = vec4(tempColor, 1.0);
```

8 PROBLEME UND LÖSUNGEN

Um zu Beginn möglichst schnell einen Prototyp zu entwickeln, der das Testen und weiterarbeiten vereinfacht, setzte unser Team anfänglich auf die Verwendung des alten OpenGL Kontextes, der es ermöglicht direkt zu rendern ohne die Verwendung von Shadern und VAO, VBO, etc. Dieser Prototyp verwendete eine überarbeitete Version der Half-Edge Datenstruktur aus dem 4. Semester. Mit dem Prototyp konnten alle Anforderungen der ersten Milestones erfüllt werden.
Da die eigene Half-Edge Datenstruktur nicht über die Möglichkeit verfügte Vertices oder Faces zu löschen, entschieden wir uns für die Verwendung der OpenMesh Half-Edge-Datenstruktur.
Um unseren Ansprüchen Genüge zu tun, endschieden wir uns auf einen aktuellen OpenGL Kontext umzusteigen. Dieser Umstieg erforderte es ein völlig neues Projekt zu erzeugen, da wir viele Dinge selber implementieren mussten, die vorher nicht notwendig waren. Beispielsweise die Darstellung der Punkte, welche sich im Prototyp auf die Nutzung der GLUT-Funktion glutSolidSphere() beschränkte, wurde zu einem umfangreicheren Unterfangen. Trotz einiger Schwierigkeiten und der Erzeugung von Mehraufwand bereuen wir diesen Schritt jedoch nicht, da wir unser Wissen aus dem ersten Computergrafik-Modul auffrischen konnten, was nötig und hilfreich für das Verständnis einiger Themen aus dem aktuellen Kurs war.
Als Entwicklungsumgebung setzten wir auf die IDE „CLion“ und achteten darauf das wir unabhängig vom Betriebssystem sind, um mit Windows sowie Linux an dem Projekt arbeiten zu können. Hierbei wurden wir vor einige Herausforderungen, mit welchen wir so nicht gerechnet hatten, gestellt. Diese Probleme beruhten zum großen Teil auf der Kompatibilität der Toolchain der einzelnen Module. So mussten wir die meisten Bibliotheken, die wir verwenden, selber kompilieren und hatten dennoch

Schwierigkeiten ein fehlerfreies Zusammenspiel der Komponenten zu gewährleisten. Für zukünftige Projekte werden wir wahrscheinlich auf Visual-Studio und die Visual-Studio Toolchain setzen, da die meisten Bibliotheken im Computergrafik Bereich hierfür konfiguriert sind und so einiges an Problemen und Arbeit eingespart wird.

9 BENUTZEROBERFLÄCHE

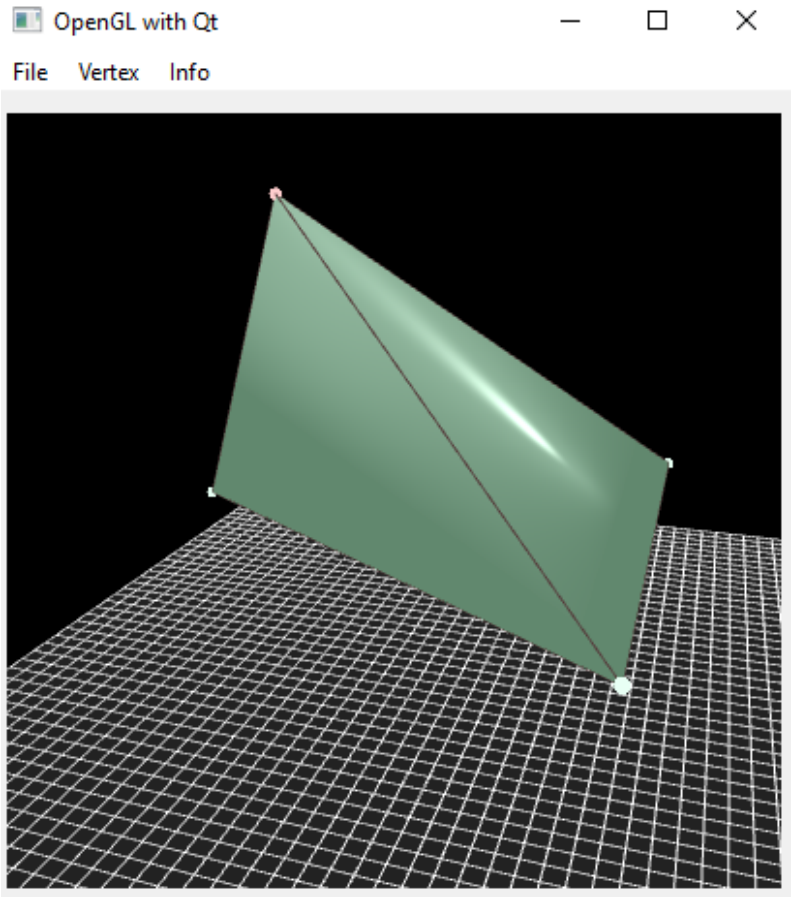

Abbildung 9: Benutzeroberfläche