**Bedienung**

**OBJ-Datei laden:** File->Open oder Strg+O

**Neues Objekt:** File->New oder Str+N

**Objekt als OBJ**

**speichern:** File->Save oder Strg+S

**Kamera Position**

**verändern:** ‘w‘,‘a‘,‘s‘,‘d‘,‘q‘,‘e‘

**Kamera Ausrichtung**

**Ändern:** Linke Maustaste + bewegen der Maus

**Raster de/aktivieren:** ‘g‘

**Punkte de/aktivieren:** ‘p‘

**Punkt/e auswählen:** Strg + Klick auf Punkt

**Neuer Punkt:** Rechte Maustaste auf Raster oder Objekt

**Ausgewählte Punkte**

**löschen:** ‘r‘

**Ausgewählte Punkte  
verschieben:**  Pfeiltasten, ‘+‘, ‘-‘

**Erstellung eines Face:** Punkte gegen den Uhrzeigersinn auswählen und ‘f‘ drücken

**CatmullClark  
Unterteilung:**  ‘c‘ für einen Schritt  
  
**Unterteilung   
rückgängig machen:** Strg+Z

**Gewichtung der   
ausgewählten Punkte:** Vertex->VertexWeight

**Shader**

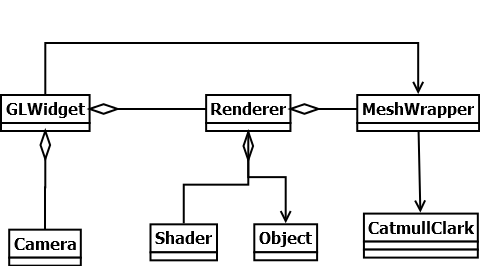
Das Programm nutzt einen *Vertex-Shader* und einen *Fragment-Shader*, die dafür genutzt werden das *Goroud-Shading* umzusetzen. In die Beleuchtungs-Berechnung fließen die Position der Primären Lichtquelle, die Farbe dieser Lichtquelle sowie weitere Faktoren wie die Farbe und Stärke des Umgebungslichtes 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.

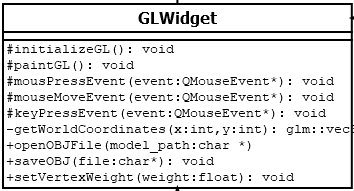
**Kamera**

Um das Betrachten des Objektes von allen Seiten zu ermöglichen endschied sich das Team eine Lösung zu implementieren die *Quaternions* 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.

**Architektur**



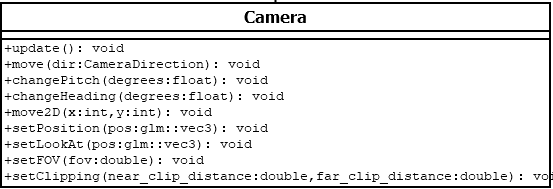
Die Abbildung zeigt eine vereinfachte Architektur des Programmes.



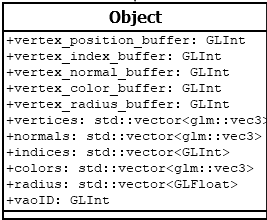
Die Klasse *GLWidget* erweitert die QT-Klasse *QGLWidget* und gibt Funktionen wie das Laden einer OBJ-Datei oder das Speichern einer OBJ-Datei, sowie das setzen der Vertex-Gewichtung an den *MeshWrapper* weiter. In *initializeGL()* werden die OpenGL Funktionalitäten initialisiert sowie die Kamera konfiguriert und der Renderer wird mit seinem Shader-Programm geladen. *paintGL()* wird von QT in einer Rendering-Schleife aufgerufen und aktualisiert die Attribute der Kamera und bindet die aktuellen Matrizen, die von dem Shader benötigt werden, an den Shader.

Wird eine Maustaste gedrückt, wird die Funktion *mousePressEvent()* aufgerufen. Diese Funktion führt die Aktionen Vertex-Auswahl und Vertex-Erzeugung aus. Beide Aktionen bedienen sich der Funktion *getWorldCoordinates(),* welche das *Ray-Picking* verfahren mithilfe der Funktionen *gluUnProject()* und *glReadPixels()* implementiert. 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()* zurück geliefert. 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 inklusive dem Raster können ausgeblendet werden, wenn sie beispielsweise beim Betrachten oder bearbeiten eines Objektes stören.

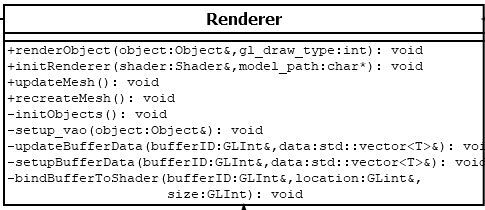
*keyPressEvent()* behandelt weitere Benutzerinteraktionen wie die Bewegung durch den Raum oder die Manipulation des *Mesh* und gibt diese an die jeweiligen Klassen weiter.



KAMERA TEXT BLA BLA



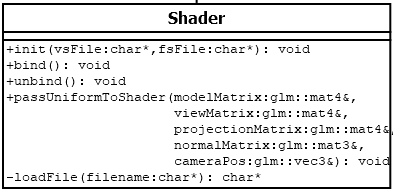
Die Klasse *Object* wird im Renderer benötigt. Hier werden alles Daten zu einem Objekt, welches gerendert werden soll, gespeichert. Die Daten bestehen aus den ID´s der unterschiedlichen *Buffer-Objects*, der ID des *Vertex-Array-Objektes*, sowie den Daten die dieses Objekt definieren. Der Renderer ist 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.



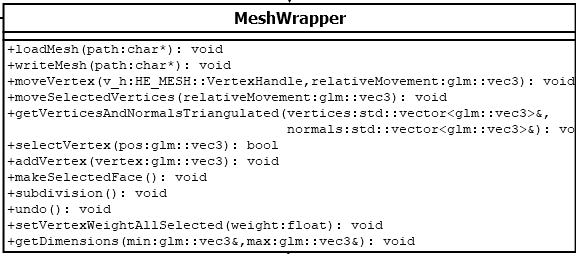
In dieser Klasse werden alle Operationen ausgeführt die den Rendering-Prozess betreffen. Die Klasse hält verschieden Instanzen der *Object* Struktur. Das *Mesh* wird in dreifacher Variation als *Object* gehalten: 1. für die Darstellung der Flächen, 2. für die Darstellung der Punkte und 3. für die Darstellung 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 berücksichtig. 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.



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 *Gouroud-Shading* umsetzen und für die korrekte Darstellung der Punkte zuständig sind.



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* durchlafen 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.

**Probleme und Lösungen während der Entwicklung**

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.