Bachelorarbeit

Rüdiger Bock

Berechnung von Volumenmodellen aus
3D-Punktwolken und deren schnelle Darstellung
mit DirectX

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Rüdiger Bock

Berechnung von Volumenmodellen aus 3D-Punktwolken und deren schnelle Darstellung mit DirectX

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Ing. Andreas Meisel
Zweitgutachter : Prof. Dr. rer. nat. Stephan Pareigis

Abgegeben am 20. August 2007

Rüdiger Bock

Thema der Bachelorarbeit

Berechnung von Volumenmodellen aus 3D-Punktwolken und deren schnelle Darstellung mit DirectX

Stichworte

Delaunaytriangulation, Alphashapes, Voronoidiagramme, DirectX9, OpenMP, Meshoptimierung

Kurzzusammenfassung

Diese Arbeit zeigt eine zukunftsorientierte Lösung zur Berechnung von 3D-Objekten aus 3D-Punktwolken. Es wird eine 3D-Punktmenge trianguliert, die Daten für die Darstellung mit DirectX9 aufbereitet, und das berechnete Objekt dargestellt. Weiterhin werden Lösungen für Speichermanagement, Datenreduktion, Optimierung der triangulierten Objekte und Verteilung von Rechenlasten auf Multiprozessorsystemen sowie auf Grafikhardware vorgestellt.

Rüdiger Bock

Title of the paper

Computation of Volumemodels from 3D-Pointclouds and fast Visualization with DirectX9

Keywords

Delaunaytriangulation, Alphashapes, Voronoidiagrams, DirectX9, OpenMP, Meshoptimizing

Abstract

This paper shows a future-oriented solution to the computation of 3D-Objects from 3D-Pointclouds. A 3D-Pointcloud is triangulated, the data is prepared for the representation with DirectX9 and the computed object is presented. Further solutions for the memory management, data reduction, optimization of the triangulated objects and distribution of computing loads on multiprocessor systems as well as on graficshardware are presented.

Danksagung

An dieser Stelle möchte ich meinem Betreuer Herrn Professor Meisel für die Hilfe bei der Auswahl dieses Themas und für die weitere Unterstützung danken. Auch Herrn Professor Pareigis, der sich als Zweitgutachter zur Verfügung stellte, sei hier gedankt.

Aus meinem familiären Kreis möchte ich ganz herzlich meinen Eltern danken, ohne deren Unterstützung und Motivation das gesamte Studium nicht möglich gewesen wäre - und ganz besonders meiner Mutter, vor der ich keinen Rechtschreibfehler verstecken konnte.

Ohne diese Unterstützung von allen wäre diese Arbeit in diesem Rahmen sicher nicht möglich gewesen.

Vielen Dank!

Inhaltsverzeichnis

1 Einführung

3D Grafiken sind mittlerweile eine ganz normale Sache. Wenn es darum geht, ein Objekt realistisch darzustellen, sind den Möglichkeiten und der Kreativität heutzutage keine Grenzen gesetzt. Vorausgesetzt, man hat teilweise das nötige Kleingeld für den Erwerb von Lizenzen für 3DSMax oder vergleichbaren Werkzeugen, die eine Fülle von Design- und Darstellungsmöglichkeiten bieten. Man braucht nur einige Grundkörper, wie Würfel oder Kugeln, geschickt in Stellung zu bringen, ein bisschen am Mesh herumzuzerren und schon hat man ein Objekt, das auch gleich in die integrierte Renderengine für die Darstellung geworfen werden kann. Selbst mit freien Werkzeugen, wie Blender, ist es relativ einfach möglich, komplexe Objekte, für zum Beispiel ein entstehendes Spiel, zu entwerfen und diese auch gleich mit den versehenen Texturen darzustellen.

Was ist aber, wenn nur eine Wolke aus 3D-Punkten vorhanden ist. Wo soll jetzt die Textur oder wie soll nun die Oberfläche festgelegt werden? Wenn nur ein Würfel mit den 8 bekannten Punkten vorhanden ist, stellt es kein Problem dar, diese 8 Punkte im 3D Raum geschickt zu verbinden, so dass dieser Körper auch als Würfel dargestellt werden kann. Wenn aber nun die Punktanzahl weitaus höher liegt, bekommt man ein nicht so einfach zu ignorierendes Problem. Die vorhandenen Werkzeuge bieten nicht die Möglichkeit, dieses Problem zu lösen. Das Ergebnis ist, dass man hier einfach aufgeben muss, da es nicht möglich ist, diese große Menge von Punkten zu einem Objekt zu verbinden.

Das Ziel dieser Arbeit ist, genau dieses Problem zu lösen und als Ergebnis einen darstellbaren Körper dieser Punktmenge zu erzeugen. Dieser Körper wird selbst dargestellt werden und es wird die Möglichkeit bestehen, diesen Körper so abzuspeichern, dass er in bekannten Werkzeugen zur Weiterverarbeitung importiert werden kann. Ein Beispiel für die Anwendung dieser Arbeit ist, ein mit einem Laserscanner eingescanntes Objekt darzustellen. Der Laserscanner würde die Punktwolke liefern, die aber so nicht ohne weiteres Aufbereiten der Vertexdaten anschaulich darstellbar wäre.

Diese Arbeit ist in die Hauptteile Allgemeiner Teil und Technischer Teil aufgeteilt. Der Allgemeine Teil enthält das theoretische Hintergrundwissen für die Verarbeitung und anschließende Darstellung der Punktmenge und der Technische Teil umfasst die Erklärung des Designs und der entwickelten Algorithmen. Zudem enthält er auch noch Tipps und Tricks, die für eine erfolgreiche Weiterverwendung des Projektes hilfreich sind.

2 Allgemeiner Teil

2.1 CGAL

2.1.1 Einleitung

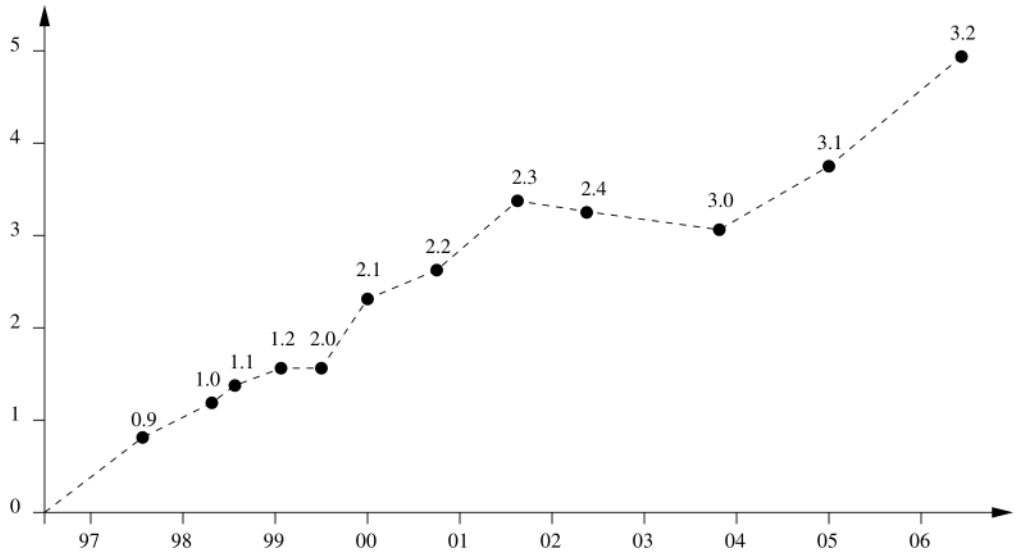

Abbildung 2.1: Umfang des CGAL Projektes

Für die Triangulation wurde die OpenSource Bibliothek CGAL verwendet. Die Abbildung 2.1 zeigt die Entwicklung und den Umfang der Bibliothek. Die vertikale Achse trägt die Einheit Codezeilen*100000, die horizontale Achse trägt die Jahreszahlen. Die Abkürzung CGAL steht für Computational Geometry-Algorithms Library. Für nichtkommerzielle Projekte ist sie kostenlos nutzbar. Andernfalls muss eine Lizens erworben werden. CGAL stellt Datenstrukturen und Algorithmen für diverse Anwendungsfälle zur Verfügung. Neben denen für Triangulationen stehen auch Algorithmen für Voronoidiagramme, Polygonoperationen, Mesher, Alphashapes, konvexe Hüllen und vieles mehr bereit. In diesem Abschnitt wird nun beschrieben, wie eine Triangulation, speziell eine delaunaykonforme Triangulation, durchgeführt wird.

2.1.2 Die Triangulation

Das Problem der Vernetzung

Eine Triangulation bedeutet, eine geschickte Vernetzung von vorhandenen (in diesem Fall im 3D-Raum) Punkten vorzunehmen. Abbildung 2.2 soll das Problem des geschickten Verbindens näher bringen.

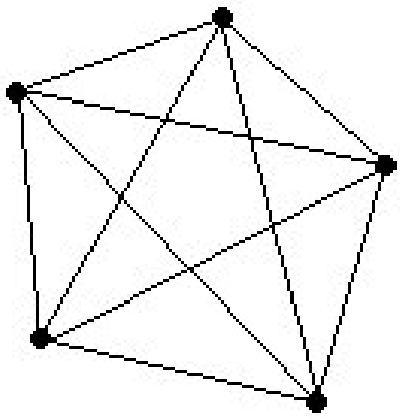

Abbildung 2.2: mögliche Vernetzung

Es ist hier die einfachste Vorgehensweise dargestellt, bei der man jeden Punkt mit allen anderen verbunden hat. Wie zu erkennen ist, überschneiden sich diverse Verbindungslinien, so dass keine übersichtliche Vernetzung vorhanden ist. Um dieses „Chaos“ zu beseitigen, liegt die Bedingung nahe, dass sich keine Verbindungslinien überschneiden dürfen. Des Weiteren sollen durch die Verbindungen Dreiecke entstehen, die die Grundlage jeder Fläche darstellen.

Unter diesen Bedingungen sind dann folgende Verbindungsvarianten möglich. Der Einfachheit halber wurde von einer Punktmenge mit 4 Punkten ausgegangen.

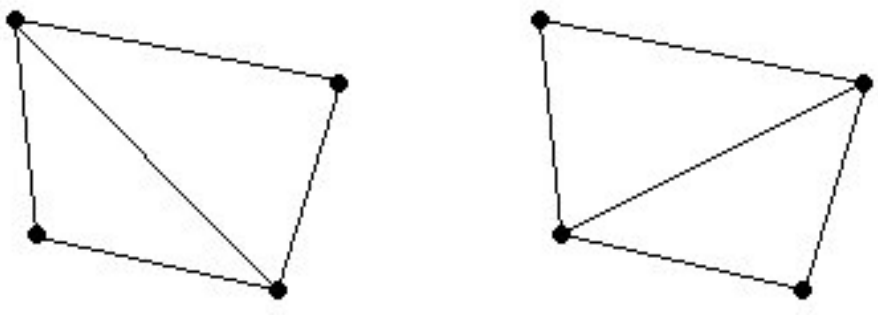

Abbildung 2.3: mögliche Vernetzung

Die in der Abbildung 2.3 dargestellten Vernetzungen erfüllen alle Bedingungen. Es ist aber zu sehen, dass es keine eindeutige Lösung gibt.

Das Kreiskriterium

Für eine eindeutige Lösung wird das Kreiskriterium definiert. Mit Hilfe von drei Punkten ist es möglich, einen Kreis zu berechnen, auf dessen Oberfläche alle drei Punkte liegen. Es dürfen nun innerhalb dieses berechneten Kreises keine weiteren Punkte, die in der Punktmenge enthalten sind, liegen. Wenn das Kreiskriterium somit erfüllt ist, wird aus den beteiligten Punkten des gebildeten Kreises ein Dreieck erstellt. Als wichtiger Hinweis sei hier angeführt, dass weitere Punkte genau auf der Oberfläche liegen dürfen. Diese Tatsache verletzt nicht das Kreiskriterium für die Erstellung der Dreiecke innerhalb dieses Kreises. In diesem Fall gibt es [Anzahl-Der-Punkte - 2] Lösungen für die korrekte Vernetzung.

Die folgenden Abbildungen sollen diesen Sachverhalt verdeutlichen.

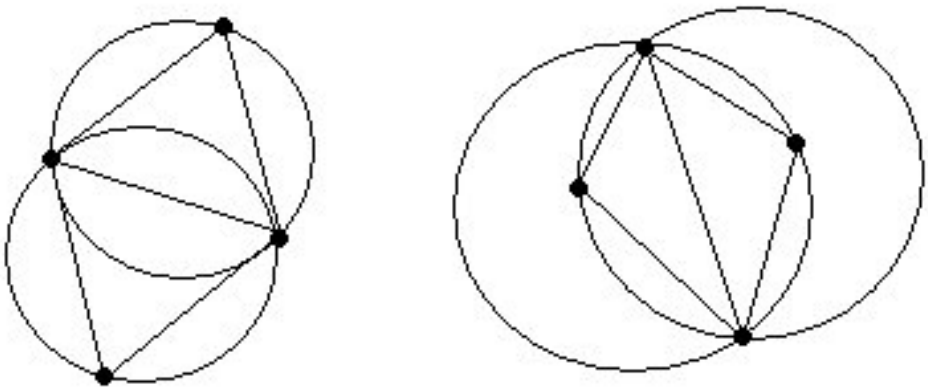

Abbildung 2.4: links eine korrekte und rechts eine falsche Triangulation

Die linke Abbildung zeigt, wie alle berechneten Kreise das Kreiskriterium erfüllen. Hier liegen keine weiteren Punkte der gesamten Punktmenge innerhalb der Kreise als die beteiligten Punkte für die Berechnung des Kreises. Die rechte Abbildung verdeutlicht die Verletzung des Kreiskriteriums sogar in beiden Fällen der Kreise. Es ist daher keine delaunaykonforme Triangulation. Wenn in diesem Fall auf diese Konformität verzichtet werden kann, ist diese Triangulation akzeptabel, da sich keine Polygone schneiden.

Berechnung eines Kreises aus drei Punkten

Abbildung 2.5 soll einmal zeigen, wie relativ einfach es ist, mit einem kleinen Trick aus drei Punkten (2D) einen Kreis zu berechnen. Ziel ist es natürlich, den Mittelpunkt und den Radius zu ermitteln, da diese Größen bekannterweise die Kenngrößen eines Kreises darstellen.

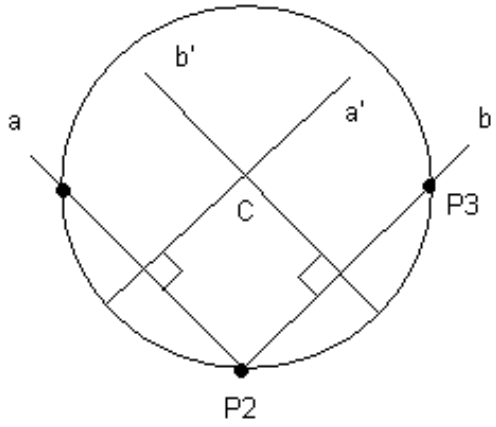

Abbildung 2.5: Vorgehensweise für die Ermittlung eines Kreismittelpunktes

Um das Beispiel anschaulich zu gestalten, werden die Punkte mit folgenden Koordinaten definiert:

P1(1,3), P2(3,1), P3(5,3)

Die Zuordnungsvorschriften der Funktionen a und b lauten somit:

$$y\,a = -x + 4 \tag{2.1}$$

$$y\,b = x - 2 \tag{2.2}$$

Die um 90 Grad gedrehten Funktionen lauten:

$$y\,a' = x \tag{2.3}$$

$$y\,b' = -x + 6 \tag{2.4}$$

Für die Berechnung der Merkmale dieser Funktionen gelten folgende Regeln:

Steigung: entspricht dem negativen reziproken Wert der nicht gedrehten Funktionen

Y-Achsenabschnitt: Da die Steigung nun bekannt ist, wird nur noch ein Punkt benötigt, durch den die gedrehte Funktion läuft. Dieser Punkt liegt auf der Hälfte der Strecke zwischen den beiden Punkten, mit denen die nicht gedrehte Funktionen berechnet wurde. In diesem Punkt schneiden sich die Ausgangs- und die gedrehte Funktion in einem Winkel von 90°.

Nun braucht nur noch der Schnittpunkt der beiden Funktionen ya' und yb' berechnet werden. Diese Koordinaten entsprechen dem Mittelpunkt des Kreises.

$$x = -x + 6 \tag{2.5}$$

$$x = 3 \tag{2.6}$$

Umlaufsinn von Polygonen und Zellen

Bei der Bearbeitung von geometrischen Problemen sind teilweise Informationen hilfreich, die einen Algorithmus beschleunigen können oder die für zum Beispiel die Darstellung unerlässlich sind. Die Information über die Orientierung von Zellen ist einerseits für die Berechnung der Triangulation notwendig, und andererseits ist die Orientierung einzelner Dreiecke für die Darstellung mit DirectX wichtig. Eine Zelle in CGAL wird aus genau vier Vertices aufgebaut und besteht somit aus vier Polygonen. Die Orientierung einer Zelle soll mit Hilfe der Abbildung 2.6 veranschaulicht werden. Sie muss bei der Dreiecksextraktion, die im technischen Teil vorgestellt wird, berücksichtigt werden.

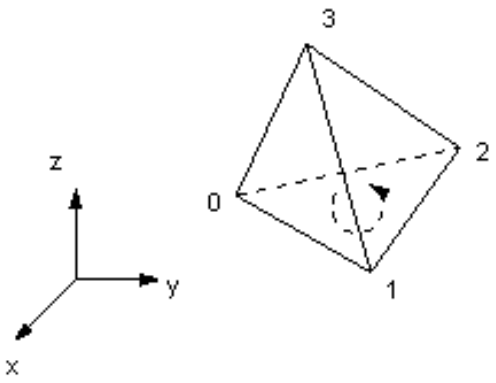

Abbildung 2.6: Orientierung der CGAL-Zelle

Die Orientierung (Indizierung der Vertices) der Zelle erfolgt in positiver Richtung - also entgegen dem Uhrzeigersinn. Man kann sich nun leicht vorstellen, dass jede Zelle genau vier Nachbarn hat. Diese Nachbarn der Zelle werden auch mit den Indices 0 bis 3 versehen, so dass es auf diese Weise durchaus möglich ist, die von CGAL aufgebaute Datenstruktur zu traversieren. Das besondere an der Indizierung der Nachbarn ist, dass wenn man zum Beispiel von Vertex null ausgeht, die Nachbarzelle mit dem Index null genau dem Vertex null gegenüber liegt. Diese Nachbarzelle teilt sich also die Dreiecksfläche mit den Vertexindices 1, 2 und 3.

Es soll nun gezeigt werden, wie der Umlaufsinn eines Polygons bestimmt werden kann. Der Umlaufsinn von Dreiecken ist in diesem Rahmen ein wichtiger Bestandteil für Darstellung mit DirectX. Für nähere Informationen über die Verarbeitung des Umlaufsinns für Polygone (speziell dem Cullmode - beschreiben auf Seite 29) sei der Leser auf den DirectX Abschnitt in diesem Teil der Arbeit hingewiesen. Teilweise spricht man auch von einem Umlaufsinn mit dem Uhrzeigersinn oder gegen den Uhrzeigersinn. Beide Ausdrucksweisen beinhalten natürlich den selben Hintergrund.

Bei Dreiecken gibt es nur zwei Umlaufrichtungen:

- die positive Umlaufrichtung
- die negative Umlaufrichtung

Abbildung 2.7 stellt eine positive Umlaufrichtung eines Dreiecks dar, und Abbildung 2.8 stellt eine negative Umlaufrichtung dar.

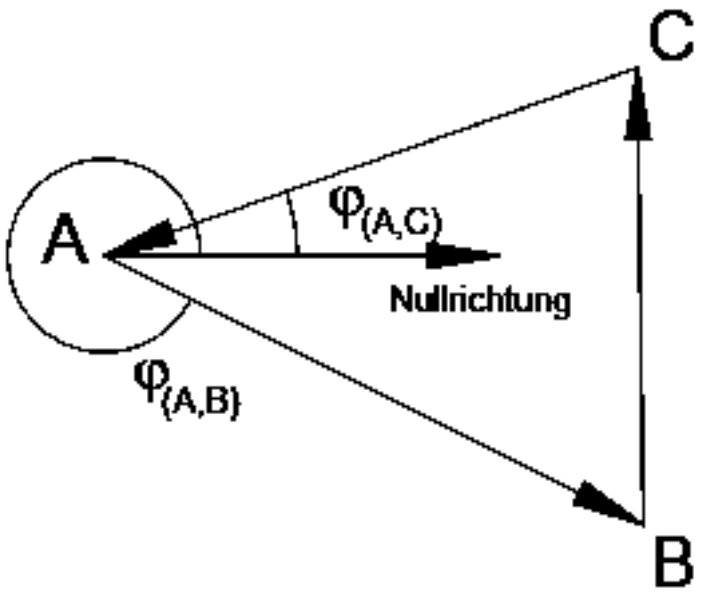

Abbildung 2.7: positiver Umlaufsinn

Mit folgenden Regeln kann der Umlaufsinn immer bestimmt werden:

für den positiven Umlaufsinn gelten folgende Bedingungen:

$$\varphi_{A,C} > \varphi_{A,B} \wedge (\varphi_{A,C} - \varphi_{A,B} < \pi)$$

$$\varphi_{A,C} < \varphi_{A,B} \wedge (\varphi_{A,B} - \varphi_{A,C} < \pi)$$

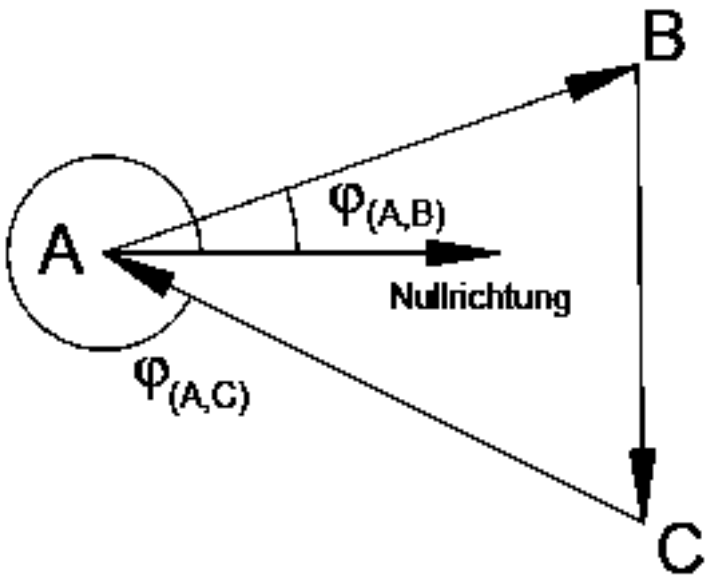

Abbildung 2.8: negativer Umlaufsinn

für den negativen Umlaufsinn gelten folgende Bedingungen:

$$\varphi_{A,C} > \varphi_{A,B} \wedge (\varphi_{A,C} - \varphi_{A,B} > \pi)$$

$$\varphi_{A,C} < \varphi_{A,B} \wedge (\varphi_{A,B} - \varphi_{A,C} > \pi)$$

Der Umlaufsinn kann auch mit Hilfe des Wertes der Determinante der in einer Matrix enthaltenen Basisvektoren bestimmt werden. Die Basisvektoren werden so in die Matrix eingetragen, wie sie dem betrachteten Umlaufsinn entsprechen sollen. Der positive Umlaufsinn (mit dem Uhrzeigersinn) entspricht einem negativen Wert der Determinante. Für den negativen Umlaufsinn spricht ein positiver Wert.

2.1.3 Voronoi Diagramme

Es soll nun kurz die Beziehung zwischen der Delaunaytriangulation und den Voronoidiagrammen veranschaulicht werden. Die folgende Abbildung stellt eine Triangulation einer Punktmenge und das Voronoidiagramm dieser Punktmenge zusammen dar.

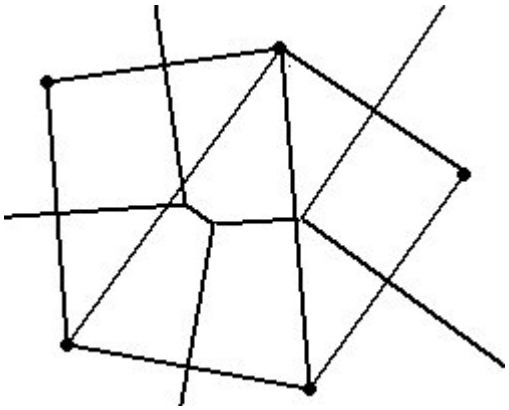

Abbildung 2.9: Triangulation und Voronoidiagramm

Aus der Abbildung ist die eigentliche Triangulation ersichtlich. Sie umfasst, wie schon beschrieben, die Vernetzung der einzelnen Vertices, so dass die beschriebenen Kriterien der Delaunaytriangulation erfüllt sind. Das andere vorhandene Netz ist das Voronoidiagramm. Es ist ersichtlich, dass das Voronoidiagramm die einzelnen Vertices in eigene Regionen verteilt. Es befindet sich immer nur ein Vertex in einer Region des Voronoidiagramms - niemals zwei! Bei genauer Betrachtung ist zu erkennen, dass jede Kante der Triangulation jeweils eine Regionsgrenze des Voronoidiagramms exakt im Winkel von 90 Grad schneidet. In einigen Fällen, wie auch diese Darstellung zeigt, muss die Regionsgrenze mit etwas Vorstellungskraft verlängert werden. Diese Tatsache kann auch für die Berechnung einer Delaunaytriangulation ausgenutzt werden, die auf einem Voronoidiagramm basiert. Ein weiterer Aspekt der Voronoiregionen ist, dass wenn man nun einen neuen Punkt einbringt, sofort ersichtlich ist, zu welchem vorhandenen Punkt er die kürzeste Distanz hat. Der neue Punkt liegt in diesem Fall in der Region des vorhandenen Punktes. Ein anschauliches Beispiel sei das Suchen des nächsten Postamtes in seiner eigenen Nähe. Es ist allerdings auch möglich, aus einer anderen Sichtweise auf das Voronoidiagramm die Delaunaytriangulation zu berechnen. Die Abbildung 2.10 soll diese Sichtweise einmal anschaulich darstellen.

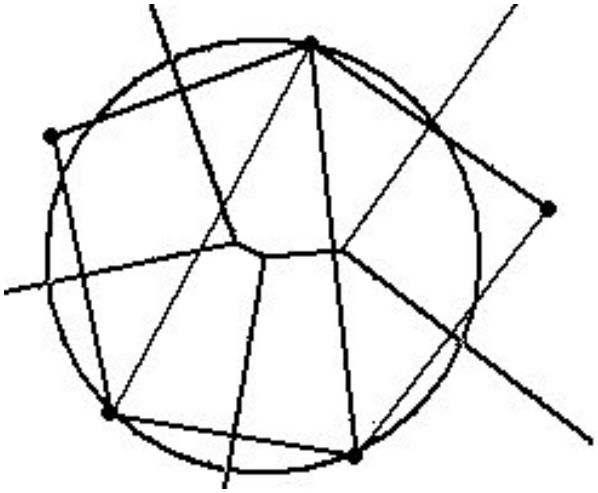

Abbildung 2.10: Triangulation und Voronoidiagramm mit einem Kreis

Zu sehen ist wieder die Triangulation mit dem dazugehörigen Voronoidiagramm. Außerdem ist genau ein Kreis vorhanden, der das Kreiskriterium erfüllt und genau 3 Vertices verbindet. Diese drei Vertices sind aufgrund der Erfüllung der Bedingung miteinander verbunden. Bei genauerer Betrachtung ist nun ersichtlich, dass der Mittelpunkt des dargestellten Kreises genau einem Knoten (dem mittleren) entspricht.

2.1.4 Alpha Shapes

Zur Einführung in Alphashapes soll die folgende schmackhafte Vorstellung dienen:

Man stelle sich eine riesige Portion Schokoladeneiscreme mit verteilten Schokoladenstückchen vor. Um dieser Eiscremeportion Herr zu werden, verwendet man einen kugelartigen Löffel und versucht nun, entgegen einer normalen Vorgehensweise, die Eiscreme zwischen den Schokoladenstückchen zu essen. Falls nun die Schokoladenstückchen so dicht beieinanderliegen, dass es unmöglich ist, mit dem Löffel in darunterliegende Eiscremeregionen zu gelangen, bleiben eben diese Bereiche weiterhin bestehen. Wie nun deutlich wird, hängt die Menge der vernichteten Eiscreme von dem Durchmesser des verwendeten Löffels ab. Sollte der Durchmesser des Löffels unendlich groß sein, hat man ganz schlechte Karten und die Figur freut sich.

Die Alphashapes stellen eine Untermenge von Polygonen im 3D-Raum und Kanten (Verbindung zwischen zwei Punkten) im 2D-Raum dar. Für die Berechnung eines Alphakomplexes ist ein sogenannter Alphawert und eine berechnete Delaunaytriangulation notwendig. Der Alphawert ist der quadrierte Radius der Testkreise. Sollte es möglich sein, zwei Vertices in irgendeiner Weise auf der Oberfläche eines Kreises zu platzieren, ohne dass ein weiterer

Vertex aus der Quellmenge innerhalb dieses Kreises liegt, gehört die Kante (Verbindung dieser beiden Vertices) dem Alphakomplex an.

Eine Verfeinerung eines Objektes mit Hilfe von Alphashapes wäre zum Beispiel die Darstellung einer Dachkante. Voraussetzung hierfür ist, dass die Mauer sowie das Dach selbst aus vielen Vertices besteht. Man sollte dann einen Alphawert in der Größenordnung der längsten Kante, die entweder auf der Mauerfläche oder der Dachfläche liegt, wählen. Somit werden die weitaus längeren Kanten von der eigentlichen Dachkante bis zur unteren Mauerkante gelöscht. Die folgenden Abbildungen sollen die Berechnung von Alphakomplexen anhand der Dachkantenverfeinerung veranschaulichen.

Abbildung 2.11: Punktmenge für nachfolgende Abbildungen

Abbildung 2.11 zeigt die Punktmenge, von der in den folgenden Abbildungen ausgegangen wird. Sie stellt die Grundlage für die in der nächsten Abbildung dargestellte Triangulation. Diese Darstellung der Punktmenge bedeutet ein Alphashape mit einem Alphawert von genau 0. Der Alphawert der folgenden Abbildungen wird kontinuierlich erhöht, bis für die letzte Abbildung ein unendlicher Wert angenommen wird.

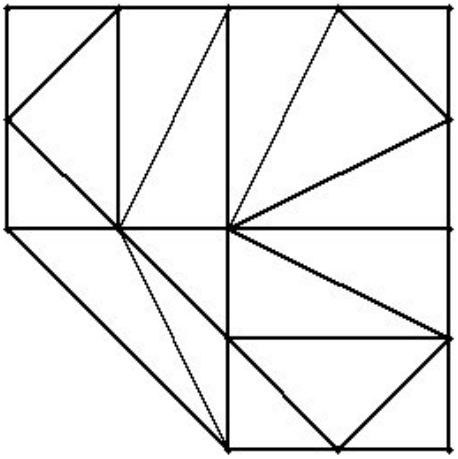

Abbildung 2.12: Triangulation der Punktmenge

Abbildung 2.12 stellt die Triangulation der Punktmenge dar. Für die Berechnung des Alphakomplexes wird sie benötigt.

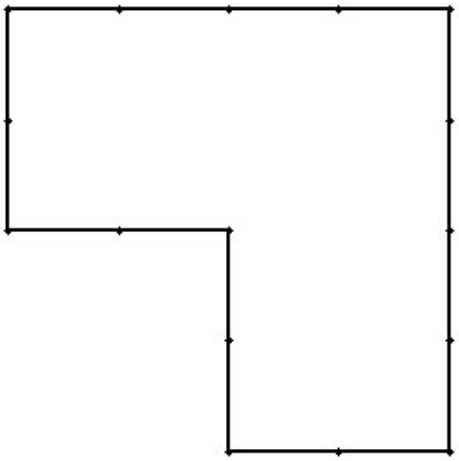

Abbildung 2.13: Alphashape mit optimalem Alphawert für das Objekt

Abbildung 2.13 zeigt schon das Alphashape, das die perfekte Annäherung an das gewünschte Objekt darstellt. Es sei darauf hingewiesen, dass nicht immer der kleinste Alphawert gleich die perfekte Annäherung bedeutet. Dies liegt in diesem Beispiel nur an der regelmäßigen Punkteverteilung.

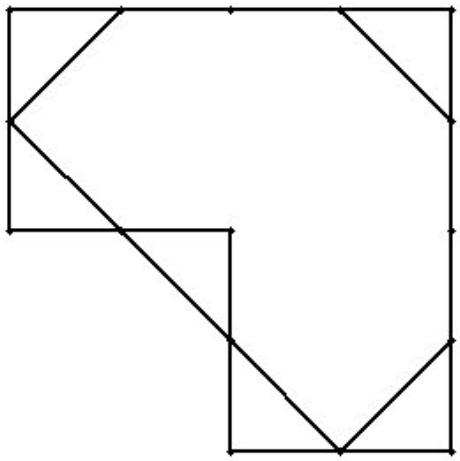

Abbildung 2.14: erhöhter Alphawert

Hier kann man anhand der diagonalen Verbindungen erkennen, dass die Durchmesser der leeren Kreise etwas größer sind als die Abstände zwischen der am nächsten beieinander liegenden Vertices.

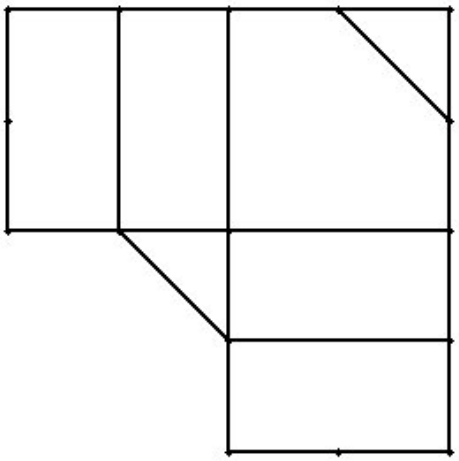

Abbildung 2.15: weiter erhöhter Alphawert

Die Durchmesser sind mittlerweile so groß, dass richtige Querverbindungen zwischen weiter auseinander liegenden Vertices möglich sind. Dies ist auch nur möglich, da innerhalb des Objektes keine Vertices vorhanden sind.

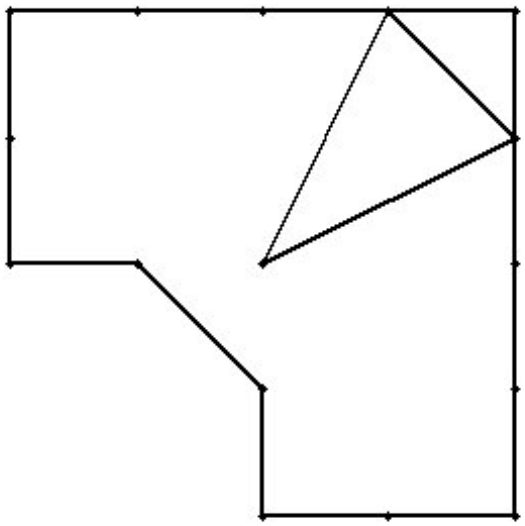

Abbildung 2.16: letztmöglicher Alphawert bevor man ihn auf unendlich setzten kann

Zu erkennen ist hier schon die Annäherung der Außenhülle an die konvexe Hülle des Objektes.

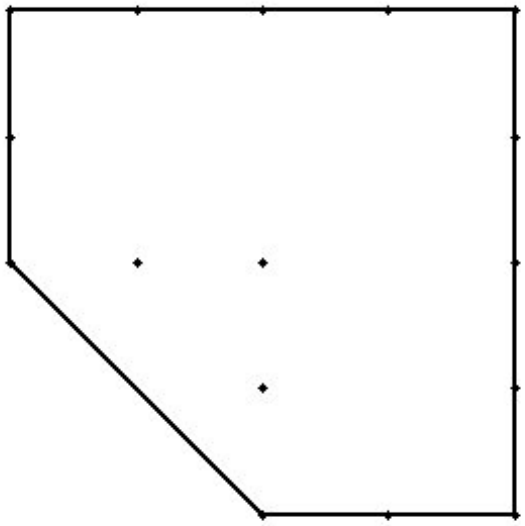

Abbildung 2.17: unendlicher Alphawert

Dieses Alphashape entspricht der konvexen Hülle, die man auch aus der Triangulation hätte gewinnen können.

Die bisher dargestellten Abbildungen zeigen in allen Fällen immer eine geschlossene Oberfläche, die einer Darstellung genügen würde. Es gibt aber natürlich auch Situationen, in denen bei einigen Alphawerten das eigentliche Objekt oder dessen Oberfläche nicht annähernd zu erkennen sind. Die folgende Abbildung soll dies näher bringen.

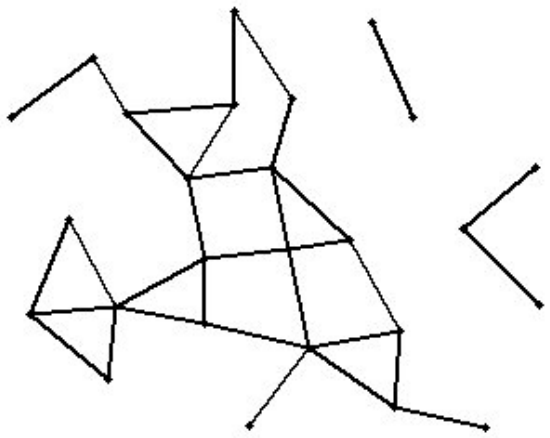

Abbildung 2.18: bei komplexeren Punktmengen und bestimmten Alphawerten ist keine geschlossene Oberfläche sichtbar

Als abschließendes Beispiel soll die Darstellung zweier Kugeln, deren Vertices in der selben Punktmenge enthalten sind, angeführt werden. Hier wird nun deutlich, welche großen Vorteile Alphashapes gegenüber der Delaunaytriangulation haben. Mit der Delaunaytriangulation ist es nur möglich, die konvexe Hülle der gesamten Punktmenge zu berechnen. Die Extraktion der Kugeloberflächen aus der Triangulation alleine wäre ein nicht zu unterschätzender Aufwand. Zwischen den Kugeln befindet sich zusätzlich eine Wand, die die konvexe Hülle stark beeinflußt, wie in der Triangulation zu sehen ist.

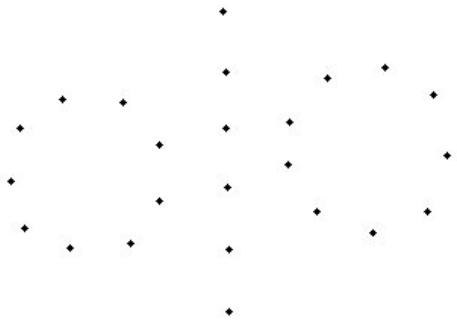

Abbildung 2.19: Die Punktmenge der beiden Kugeln

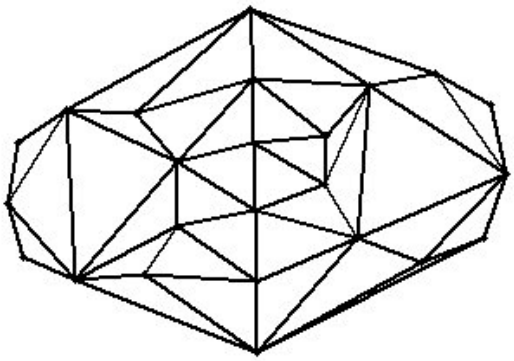

Abbildung 2.20: Die Triangulation der beiden Kugeln

Hier ist sehr gut zu erkennen, wie die Wand die konvexe Hülle vergrößert.

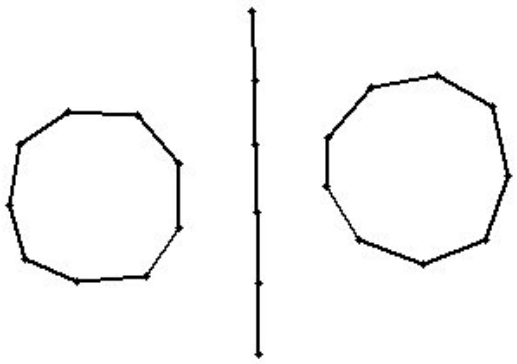

Abbildung 2.21: Das Alphashape der beiden Kugeln

Zu beachten sind die Abstände zwischen den Vertices der Kugeln. Sie sind kleiner als die Abstände der Vertices zwischen den Kugeln und der Wand. Daher kann ein Alphawert gefunden werden, der einen maximalen Wert von

$$\alpha = \left(\frac{Abstand\_Kugel\_Wand}{2} \right)^2$$

und einen minimalen Wert von

$$\alpha = \left(\frac{Abstand\_Kugelvertex\_Kugelvertex}{2} \right)^2$$

hat.

2.1.5 Abschluss

Es wurde dargestellt, welche Kriterien eine delaunaykonforme Triangulation erfüllt. Des Weiteren ist nun ersichtlich, wie auf unterschiedlichen Wegen eine Triangulation berechnet werden kann. Da für diese Arbeit eine geordnete Struktur für die Darstellung benötigt wird, stellt die Delaunaytriangulation eine gute Grundlage für dieses Problem dar. Es ist auch ersichtlich, dass die Alphashapes eine eventuelle weitere Verfeinerung des darzustellenden Objektes liefern. Es wurden wegen der einfacheren Darstellung nur zweidimensionale Fälle dargestellt. Die Regeln, die für den zweidimenisonalen Fall vorgestellt wurden, gelten auch für den dreidimensionalen Raum. Wie die Datenstruktur der Triangulation verwendet wird, um das Objekt darzustellen, inklusive der Vor- und Nachteile der Alphashapes, wird im technischen Teil dieser Arbeit erklärt.

2.2 DirectX

2.2.1 Einleitung

Für die Darstellung der triangulierten Objekte wurde die DirectX - API ausgewählt. Alternativ hätte natürlich auch die OpenGL - API genutzt werden können. DirectX bietet allerdings die beste Unterstützung für Programme unter Windows, da diese Schnittstelle tief in das Windowsbetriebssystem integriert ist. Bei OpenGL hingegen müssen zum Beipiel teilweise die DLLs mit dem Programm ausgeliefert werden. Zudem setzt Microsoft mit DirectX die neuen Standards für die Grafikprogrammierung, und Opensourceprojekte wie die OpenGL - Library müssen anschließend die neue Technologie implementieren. Für Windows Vista hat Microsoft zu diesem Zeitpunkt geplant, für die OpenGL Schnittstelle einen Wrapper zu erstellen. Mit dieser Implementation hat OpenGL nicht mehr die Möglichkeit, direkt die Grafiktreiber anzusprechen. Für die Nutzung der DirectX - API stellt Microsoft das bekannte DirectX - SDK zum Download bereit. Diese Arbeit wurde gegen das April 2007 SDK gelinkt, da dieses voraussichtlich das letzte SDK sein wird, mit dem es möglich ist, zur Laufzeit entweder für WindowsXP die DirectX9 Schnittstelle oder für Vista die DirectX10 Schnittstelle zu nutzen.

Nachfolgend wird auf die wichtigen Themenbereiche eingegangen, die für die Erstellung dieser Arbeit als Grundwissen dienten. Für spezielle Nachforschungen sei der Leser auf die DirectX - SDK Dokumentation hingewiesen, da es über den Rahmen dieser Arbeit hinausgehen würde.

2.2.2 Das Grafiksystem

Abbildung 2.22 zeigt die Direct3D Pipeline. Es sind die einzelnen Stationen sichtbar, die die Quelldaten durchlaufen, damit am Ende ein Bild entsteht. Die Quelldaten bestehen aus den Vertexkoordinaten und Texturen. Die Vertices bilden die Verbindungspunkte der Dreiecke des Meshes. Die Texturen werden anschließend an die Vertices „gehängt“.

Heutige Grafikhardware bietet die direkte Programmierung einzelner Stufen der Pipeline, was ein deutlich flexibleren Umgang mit den Ausgangsdaten und somit beeindruckende Effekte ermöglicht. Diese Programme werden als Shader bezeichnet. Zu diesem Zeitpunkt lässt es aktuelle Hardware zu, die folgenden Stufen der Pipeline mit eigenen Shadern zu programmieren:

- Die Vertexverarbeitung
- Die Geometrieverarbeitung
- Die Pixelverarbeitung

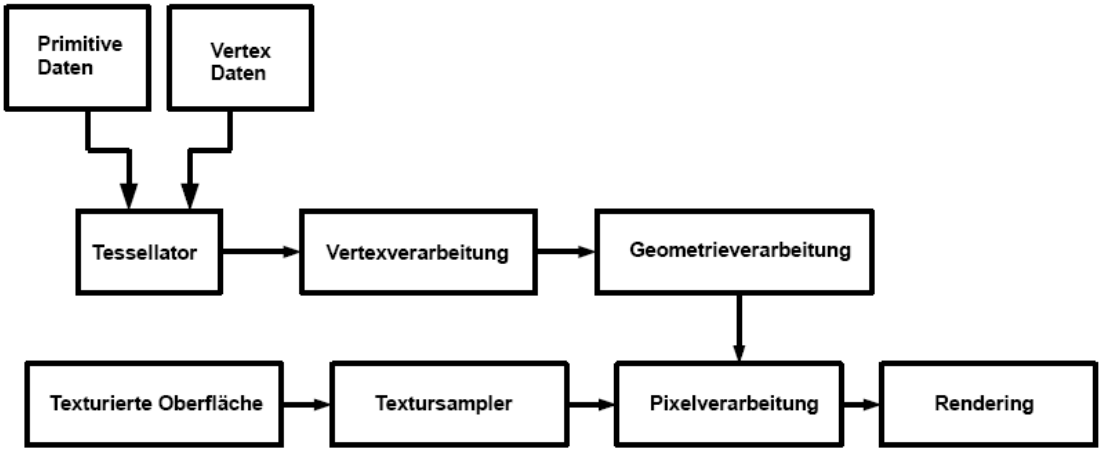

Abbildung 2.22: Die Grafikpipeline

- Am Anfang der Pipeline werden die untransformierten Vertexkoordinaten und die primitiven Daten an den Tesselator weitergereicht.
- Die Aufgabe eines Tesselators ist die Zerlegung eines Objektes in Dreiecke. Die Koordinaten der Eckpunkte dieser Dreicke sind die Ergebnisse der Tesselation. Somit ist zum Beispiel unter anderem die Aufgabe des Tesselators, eine Displacementmap zu verarbeiten. Als primitive Daten sind Punkte, Linien, Dreiecke und Polygone zu verstehen. Die Anordnung der Vertexkoordinaten im Ausgangsvertexpuffer und die spätere Angabe der Art der primitiven Daten sind für die Grafikhardware notwendig, um die Vertexdaten korrekt interpretieren zu können.
- Nach dem Tesselationsvorgang werden die Vertexdaten in der Vertexverarbeitung in den Sichtbereich des Betrachters, die Kamera, transformiert.
- Anschließend werden die Vertexdaten in der Geometrieverarbeitung auf objektorientierter Ebene interpretiert und zum Beispiel mit der Rückseite zum Betrachter liegende Dreiecke vernachlässigt (Backfaceculling). Es werden auch Tiefenprüfungen durchgeführt, damit bei der Texturierung festgestellt werden kann, ob die Textur überhaupt sichtbar ist.
- Nach diesen Schritten sind die Objekte im Raum platziert und können nun texturiert werden. Hierzu werden die Texturen (texturierte Oberflächen) mit dem Textursampler vorbereitet, der sich zum Beispiel der benötigten Detailtiefe der Textur annimmt. Die Geometriedaten werden dann in der Pixelverarbeitung mit den Texturdaten kombiniert, so dass nach dem Rendering das fertige Bild entsteht.

Diese Arbeit zeigt im technischen Teil, wie die Transformationen der Vertices auf die Grafikhardware verlagert werden können, um von ihr abhängige Berechnungen nicht auf dem Hauptprozessor durchführen zu müssen. Heutige Grafikhardware ist um einige Faktoren schneller im Umgang mit Floatingpoint - Operationen als sogar aktuelle Dualcoreprozessoren.

Diese Arbeit nutzt einerseits die 3D Funktionen der Grafikhardware (sofern vorhanden) mit Hilfe von Direct3D und andererseits wird die normale 2D Funktionalität für die GUI Oberfläche genutzt. Abbildung 2.23 soll einmal veranschaulichen, wie beide Grafiksysteme für Anwendungen bereitgestellt werden.

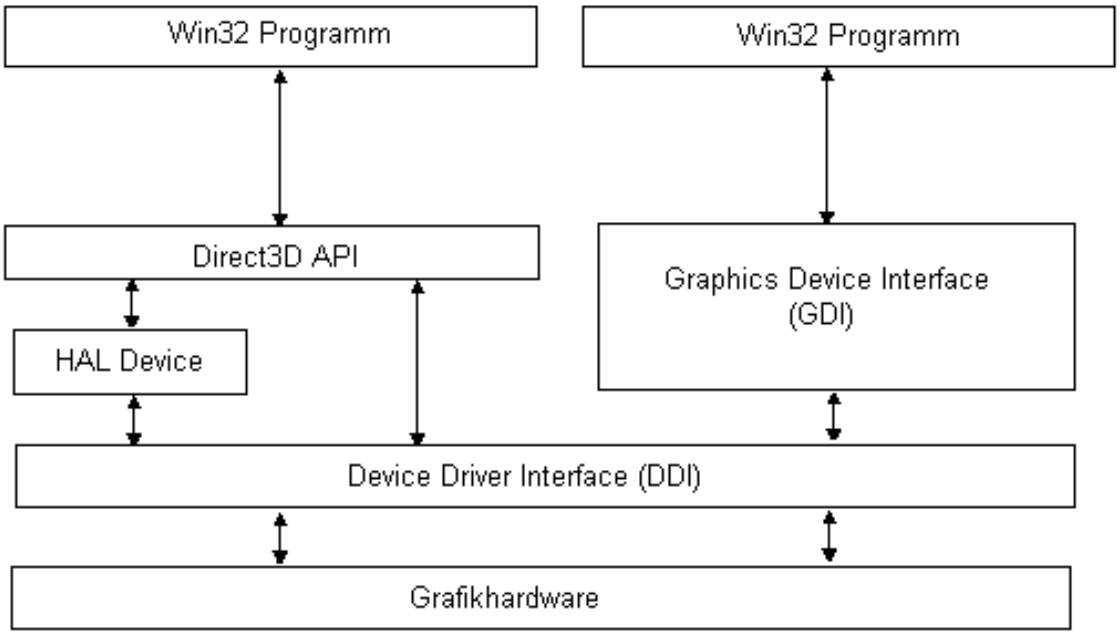

Abbildung 2.23: Das 2D und 3D Grafiksystem im Gesamtsystem

Für 2D Anwendungen werden die Funktionen der GDI Schnittstelle genutzt. Auf der anderen Seite verwenden 3D Anwendungen die Direct3D API, um aufwändigere Funktionen für 3D Berechnungen nicht eigens implementieren zu müssen. Diese beiden Layer besitzen natürlich die typischen Merkmale eines Interfaces, da kein Wissen über das Interface des Grafiktreibers vorhanden sein muss. Das HAL Device (Hardwareabstractionlayer) dient dazu, Funktionen, die die Grafikhardware besitzt, aufzulisten.

2.2.3 Das Direct3D Device

Das Direct3D Device ist die eigentliche Renderingkomponente. Sie kapselt und verwaltet die einzelnen Zustände und führt die Transformationen, Beleuchtung und die Rasterisierung aus.

Ein Device besitzt somit - abstrakt gesehen - die folgenden Module:

- Transformationsmodul
- Beleuchtungsmodul
- Rasterizer

Diese Module sind in zwei Arten von Devices integriert. Einerseits existiert das HAL–Device und andererseits existiert ein Reference–Device. Normalerweise wird versucht, das HAL–Device zu nutzen, da es direkt die Hardware anspricht.

Das HAL Device

Eine Instanz des HAL–Devices ermöglicht die direkte Nutzung der Grafikhardware. Es werden die Transformationen, Beleuchtung und Rasterisierung direkt auf der Grafikhardware durchgeführt. Ein HAL–Device stellt Hardwarevertexverarbeitung sowie Softwarevertexverarbeitung zur Verfügung. Sollte die Hardware eine Funktionalität nicht unterstützen, so muss auf das nachfolgend erklärte Reference Device zurückgegriffen werden.

Das Reference Device

Das Reference–Device ist eine Softwareimplementation der Hardware. Es werden fast alle Funktionen, die eine Hardware unterstützen sollte, in Software emuliert. Das Reference–Device ist vor allen Dingen für Debuggingzwecke konzipiert und ist nur bei installiertem DirectX–SDK verfügbar. Ein wichtiger Vorteil des Reference–Devices ist, dass Entwickler in der Lage sind, in einer virtualisierten Umgebung die Entwicklung der 3D Applikation vorzunehmen und somit auf eine Softwareimplementation der Grafikhardware angewiesen sind. Aufgrund der Tatsache, dass das Reference–Device die Funktionalitäten, die die Hardware unterstützen soll, korrekt implementiert, stellt sich das Reference–Device somit als ein wichtiges Vergleichsobjekt dar, um für Hardwarehersteller als Leitfaden zu dienen.

Da es sich um eine vollständige Softwareumgebung handelt, wird der Nachteil des Geschwindigkeitsverlustes deutlich. Es werden alle Vorgänge, die eigentlich die GPU übernehmen sollte, auf die CPU verlagert. Zwar wird bei modernen Prozessoren versucht, durch die

bevorzugte Nutzung von MMX und SSE eine Beschleunigung der Berechnungen zu erreichen, aber diese Optimierungen reichen bei weitem nicht, um mit der Leistung einer Grafikhardware zu konkurrieren.

Die Zustände eines Devices

Damit das Rendering in gewünschter Form abgeschlossen werden kann, ist es teilweise notwendig, die Standardwerte der Zustände einzelner Pipelineabschnitte zu überschreiben. Die Angabe solcher Werte hat natürlich ein abweichendes Verhalten der Hardware zur Folge, damit man bestimmte Effekte realisieren kann. Die folgenden schon von der Namensgebung her eingängigen Zustände sollen kurz erläutert werden:

- Beleuchtung:

  Dieser Zustand wird entweder an- oder ausgeschaltet. Wenn die Beleuchtung gewünscht wird, kann man auf eigene Berechnungen von Vertexfarben mit Vertex- oder Pixelshadern verzichten. Es muss allerdings eine Normale für jeden Vertex angegeben werden, damit die Beleuchtung berechnet werden kann. Weiterhin müssen auch Lichtquellen verwaltet werden, die sich dann zusätzlich selbst unsichtbar im 3D Raum befinden. Diese Arbeit verwendet keine Lichtquellen, sondern vergibt eine Farbe für jeden Vertex, die im Vertexshader verarbeitet wird.

- Antialiasing: Auch bekannt als Kantenglättung. Standardmäßig ist diese Berechnung ausgeschaltet - also auf einen Sample pro Pixel gesetzt.

- Culling:

  Wie schon im Abschnitt 2.1.2 auf Seite 13 beschrieben wurde, wird einem Dreieck eine Vorder- und eine Rückseite zugeordnet. Die Vorderseite entspricht dem Umlaufen im Uhrzeigersinn. Es besteht die Möglichkeit durch entsprechendes Setzen dieses Renderzustandes, die Rückseiten oder Vorderseiten eines Dreiecks nicht darzustellen, was eine Beschleunigung der Berechnung der Szene zur Folge hat. Das Nichtberechnen der Rückseiten eines Dreiecks ist der Standardzustand. Im technischen Teil werden die Gründe (Abschnitt 3.4.3 auf Seite 57) beschrieben, die eine Darstellung der Vorder- und Rückseiten eines Dreiecks erfordern.

- Nebel:

  Die Berechnung von Nebel und Unschärfe verleiht der Szene mehr Realitätstreue. Die Berechnung von Nebel und Unschärfe ist standardmäßig ausgeschaltet.

- Shadingmodus:

  Es kann zwischen dem Flat- und Gouraudshading gewählt werden. Eine detailliertere Beschreibung ist auf Seite 36 zu finden. Die Shadingmodes beeinflussen die Sichtbarkeit von Kanten eines Objektes.

- Tiefenpuffer:

  Der Tiefenpuffer enthält Informationen über die Lage von Oberflächen im Raum. Sollte eine neue Oberflächen eine schon vorhandene Oberfläche überdecken wollen, wird getestet, ob die neue Oberfläche vor der alten liegt und somit die alte verdeckt. Sollte die neue Oberfläche hinter der schon gerenderten liegen, wird sie verworfen.

Lost Devices

Der Normalzustand von Devices ist, wenn alle Daten vollständig und korrekt gerendert werden können. Unter Umständen kann allerdings das Device auch verloren gehen, wenn zum Beispiel der Focus der Tastatureingabe nicht mehr beim Device liegt. Somit gehen eventuelle wichtige Eingaben, die für das korrekte Rendering notwendig sind, verloren und das Renderingverhalten entspricht nicht mehr dem gewollten Verhalten. Eine bekannte Aktion ist die Eingabe der Tastenkombination ALT+TAB, die, wenn eine 3D Applikation im Vollbildmodus läuft, ein anderes vorhandenes Fenster einer anderen Applikation in den Vordergrund holt und ihr den Eingabefocus der Tastatur gibt. Dieser Sachverhalt führt auf jeden Fall zu einem sogenannten verlorenen Device. Es gibt viele verschiedene Sachverhalte, die zu einem verlorenen Device führen. Ein weiteres Beispiel ist das Auftreten von Energiesparmodi. Sollte eine Vollbildmodus–3D–Applikation es versäumen, den Bildschirmschoner vorübergehend zu deaktivieren, ist es auch sehr wahrscheinlich, dass nach dem Laden des Bildschirmschoners das Renderingdevice verloren ist. Es ist also durchaus möglich, für einige Ereignisse Vorkehrungen zu treffen, um eine Restauration des Renderingdevices zu vermeiden.

Um zu erkennen, dass der Fall des verlorenen Devices vorliegt, sollte der Rückgabewert der Funktion IDirect3DDevice9::Present überwacht werden. Nur an diesem Rückgabewert ist zu erkennen, ob das Device verloren gegangen ist. Andere Setupfunktionen lassen diesen Zustand nicht erkennen. Wenn das Device nun verloren ist, müssen unter Umständen alle Ressourcen wiederhergestellt werden. Das heißt, dass zum Beispiel die Vertexpuffer, die im Standardpool allociert sind, neu initialisiert werden müssen, da die Daten im Grafikspeicher überschrieben worden sind. Die einzigste Methode, die den Zustand von verloren auf normal setzen kann, ist IDirect3DDevice9::Reset. Ihr werden die Presentparameter übergeben, mit denen auch das vorherige Device instantiiert worden ist. Es kann allerdings der Aufwand, der bei einem verlorenen Device entsteht, umgangen werden. Alle Ressourcen

sollten entweder als vom System verwaltete Ressourcen oder gleich im normalen Systemhauptspeicher angelegt werden. Die vom System verwalteten Ressourcen werden in den Grafikspeicher kopiert und vom System wird eine Kopie gehalten. Sollten sich Änderungen ergeben, müssen beide Speicher auf den neusten Stand gebracht werden. Dieser Vorgang kostet natürlich Rechenleistung und die Darstellung ist somit langsamer, als wenn die Ressourcen nur im Grafikspeicher liegen. Ressourcen, die gleich im Systemspeicher angelegt werden, können direkt von der CPU verarbeitet werden. Diese sollten auch nur dann so angelegt werden, wenn dies zum Beispiel bei dem Update von Oberflächen nötig ist.

Verarbeitung von Vertexdaten

Die Grunddaten von 3D Objekten sind bekannterweise die Eckpunkte der Dreiecke, aus denen das Objekt besteht. Die Koordinaten dieser Eckpunkte sowie eine optional zugeordnete Farbe und/oder eine Normale können in der Beschreibung des Vertex enthalten sein. Diese Daten werden in sogenannten Streams, die deklariert werden müssen, der Hardware bereitgestellt. Die Ziele der Verarbeitung der Vertexdaten ist die Berechnung von Beleuchtungen und Farbinterpolationen zwischen verbundenen Vertices sowie der Transformation der Welt-, Sicht- und Projektionskoordinaten der beteiligten Dreiecke. Es besteht die Möglichkeit, diese Daten einerseits vollständig mit der Hardware oder andererseits in Software oder aber auch im Mixedmode zu verarbeiten. Für die softwareseitige Verarbeitung bestehen festgelegte Grenzen der Möglichkeiten, wie zum Beispiel die maximale Anzahl von möglichen Lichtquellen in der Szene. Die Grenzen der hardwareseitigen Verarbeitung können nur zur Laufzeit abgefragt werden. Es besteht allerdings die Möglichkeit, während der Laufzeit zwischen hardwareseitiger und softwareseitiger Verarbeitung zu wechseln.

2.2.4 primitive Daten

Mit den folgenden Datenrepräsentationen kann DirectX arbeiten:

- Punktlisten:

  Hier stehen nur in irgendeiner Ordnung die Vertices im Puffer.

- Linienlisten:

  Eine Linie besteht immer aus einer Verbindung von zwei Vertices. Der erste und der zweite Vertex einer Linie müssen im Puffer hintereinander stehen. Sonst wird die Linie nicht erkannt und es entsteht eine fehlerhafte Darstellung.

- Linestrips:

  Wenn zusammenhängende Linien modelliert werden sollen, empfiehlt es sich, sie als Linestrips zu speichern. Abbildung 2.24 soll die Linestrips einmal veranschaulichen.

Abbildung 2.24: Linestrips

Wie zu erkennen ist, entspricht der zweite Vertex der ersten Linie dem ersten Vertex der zweiten Linie. Hierdurch kann Speicherplatz eingespart werden, da der Verbindungsvertex nur einmal im Puffer vorkommt.

- Dreieckslisten: Diese Datenanordnung der Vertices im Puffer entspricht vom Prinzip her der Anordnung der Linienlisten, nur dass hier eben drei hintereinander stehende Vertices ein Dreieck definieren. Nur dieses Prinzip wird in dieser Arbeit verwendet.
- Dreieckstrips:

  Vom Prinzip her bestehen an der Art der Datenrepräsentation zwischen Dreieckstrips und Linestrips keine Unterschiede. Das Besondere an dieser Darstellung ist allerdings, dass das Folgedreieck, dessen ersten beiden Vertices den letzten beiden Vertices des vorhergehenden Dreiecks entsprechen, einen Umlaufsinn entgegen dem Uhrzeigersinn besitzt. Für Informationen betreffend des Umlaufsinns von Objekten sei der Leser auf den Abschnitt 2.1.2 auf Seite 13 hingewiesen. Abbildung 2.25 soll die Dreieckstrips veranschaulichen.
- Dreiecksfächer:

  Eine Besonderheit sind die Fächer. Man kann sie sich als eine Art Kreis mit einem Vertex als Mittelpunkt vorstellen. Der Kreisumfang muss allerdings nicht, wie man es bei einem klassischen Kreis gewohnt ist, geschlossen sein - daher auch die Bezeichnung „Fächer“. In diesem Fall ist der Mittelpunktvertex in allen Dreiecken vorhanden. Abbildung 2.26 dient zur Veranschaulichung.

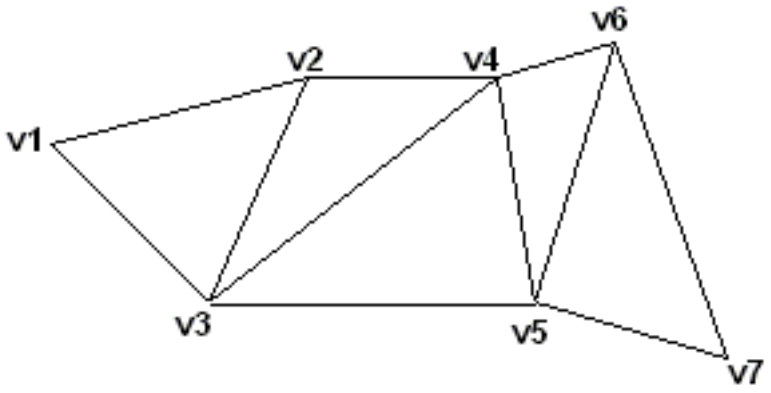

Abbildung 2.25: Dreieckstrips

In diesem Fall kann der Speicherplatz von zwei Vertices für die Darstellung von zwei Dreiecken eingespart werden.

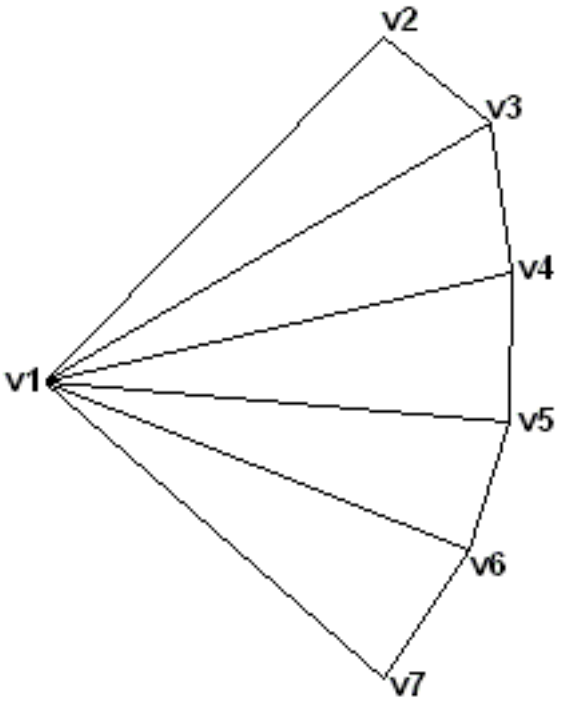

Abbildung 2.26: Dreiecksfächer

Hier kann der Speicher von Anzahl_Dreiecke - 1 Vertices eingespart werden.

2.2.5 Das Koordinatensystem

Bekannt sind das Linkehand- und das Rechtehandkoordinatensystem. Das Linkehandkoordinatensystem ist vorstellbar, wenn der Daumen der linken Hand die positive Richtung der X–Achse, der Zeigefinger die positive Richtung der Y–Achse und der Mittelfinger die positive Richtung der Z–Achse repräsentiert. Direct3D verwendet nur das Linkehand–System. Sollten die Vertexdaten auf ein Rechtehandsystem ausgelegt sein, müssen diese Daten bei der Portierung für Direct3D aufbereitet werden. Außerdem muss die Welttransformationsmatrix angepasst werden.

2.2.6 Transformationen

Die folgenden Transformationen sind notwendig, um ein Objekt korrekt in einem virtuellen 3D–Raum darzustellen. Sie werden explizit über Direct3D Funktionsaufrufe in der Fixed–Function–Pipeline gesetzt, da die Szene sonst nicht wie geplant gerendert werden kann.

- Welttransformation:

  Bei der Modellierung eines Objektes werden dessen Vertices im Koordinatensystem dieses Objektes gesetzt. Alle Vertices des Objektes stehen somit relational zueinander und definieren hiermit die Ausmaße des Objektes. Wird nun das Objekt in einer Szene eingesetzt werden, muss das Objekt zu der Ausgangsposition in dem neuen Weltkoordinatensystem verschoben (transformiert) werden. Für diese Operation wird eine Weltmatrix definiert, die diese Aufgabe erledigt. Sollte das Objekt in der Szene weiter verschoben oder auch skaliert werden müssen, wird diese Matrix neu berechnet.

- Sichttransformation:

  Für ein Bild oder einen Film benötigt man gewöhnlich eine Kamera, die aus ihrem Standpunkt heraus das Bild oder die Szene festhält. In einer 3D Szene ist diese Kamera nicht als reales Objekt vorhanden - man kann sie sich aber als eine virtuelle Kamera vorstellen. Die Sichtmatrix beschreibt nun den Aufenthaltsort und den Sichtwinkel der Kamera. Für die Berechnung dieser Matrix stellt DirectX eine Initialisierungsfunktion zur Verfügung, die folgende Parameter benötigt:

  - einen 3D Vektor, der den Standort im 3D Raum beschreibt
  - einen 3D Vektor, der den Punkt im 3D Raum beschreibt, der direkt von der Kamera betrachtet wird
  - einen 3D Vektor, der definiert, in welcher Richtung für die Kamera „oben“ ist

- Projektionstransformation:

  Die Projektionsmatrix definiert einen dreidimensionalen Sichtbarkeitsbereich. Sollten sich Objekte oder Teile von Objekten in diesem Bereich befinden, sind sie somit für die Kamera sichtbar und werden gerendert. Dieser Bereich wird in der Tiefe durch zwei Ebenen begrenzt. Alles was sich vor der Frontebene oder hinter der Hinterebene befindet, ist außerhalb des Sichbarkeitsbereiches. Weiterhin wird ein Öffnungswinkel der Kamera benötigt, der normalerweise bei 45 Grad liegt. Da die Frontebene kleinere Ausmaße hat als die Hinterebene, werden Objekte innerhalb des Sichtfeldes, je nachdem an welcher Tiefenposition sie liegen, skaliert, um einen perspektivischen Eindruck zu vermitteln. Diese Skalierung ist nicht mit der Skalierung in der Welttransformation zu verwechseln. Die folgenden Informationen reichen für die Berechnung einer Projektionsmatrix aus:

  - Öffnungswinkel in Radians
  - Seitenverhältnis der Frontebene
  - Abstand der Kamera zur Frontebene
  - Abstand der Kamera zur Hinterebene

2.2.7 Vertexpuffer

Ein Vertexpuffer ist als Container für die Speicherung von mindestens den Vertexkoordinaten zu verstehen. Vertexdaten können einerseits als schon transformierte oder untransformierte Daten vorliegen. Im Fall, dass die Koordinaten transformiert worden sind, brauchen die Welt-, Sicht- und Projektionsmatrix nicht gesetzt zu werden. Um einen Vertexpuffer zu erstellen, wird für die Fixed–Function–Pipeline sowie für einen Vertexshader eine Beschreibung über den Aufbau der Vertexdaten benötigt. Diese Beschreibung wird aus Flexible–Vertex–Format (FVF) Codes zusammengesetzt. Für einen Vertexshader wird hingegen eine Vertexdeklaration benötigt. Im technischen Teil wird ersichtlich, wie auf die Fixed–Function–Pipeline zurückgegriffen wird, falls ein Vertexshader nicht gesetzt werden konnte. Mit den FVF–Codes wird unter anderem beschrieben, ob es sich zum Beispiel um transformierte oder untransformierte Vertices oder Farbdaten handelt. Weiterhin muss für den Vertexpuffer angegeben werden, wo er reserviert werden soll. Es bestehen folgende Möglichkeiten:

- Standardpool:

  Der Puffer wird je nach Häufigkeit des Gebrauchs im AGP- oder Grafikkartenspeicher erstellt. Dies beinhaltet den Nachteil, der im Abschnitt 2.2.3 (Lost Devices) auf Seite 30 beschrieben wurde.

- verwalteter Pool:

  Bei Bedarf werden die Daten aus dem vom System gesicherten Puffer in den Grafikkartenspeicher transferiert.

- Systempool:

  Die Daten werden nur in einem Systempuffer gehalten, der für die Grafikkarte nicht unbedingt zugreifbar ist. Mit diesen Puffern können Puffer, die sich im Standardpool befinden (Surfaces und Texturen), aktualisiert werden.

Um die Daten aus einem Vertexpuffer rendern zu können, müssen diese als Eingangsdaten von Streams gesetzt werden. Je nach Hardware können Puffer auf mehrere Streams aufgeteilt werden. Der Vollständigkeit halber seien nun noch die indizierten Vertexpuffer angeführt. Hier existieren zwei Puffer nebeneinander:

- Puffer mit Indizes, die einen Offset im Vertexpuffer darstellen
- Vertexkoordinaten, wie sie in nichtindizierten Puffern vorhanden sind

Der Vorteil von Indexpuffern liegt darin, dass auf duplizierte Vertices für die Beschreibung von nebeneinander liegenden Polygonen verzichtet werden kann. Dies ist auch das in Abschnitt 2.2.4 auf Seite 31 beschriebene Prinzip mit dem Unterschied, dass hier ein extra Indexpuffer benötigt wird.

2.2.8 Shading

Unter dem Shading versteht man die Berechnung von Lichtintensität und Farben für jeden Punkt auf dem Polygon. Die folgenden zwei Shadingmodes sollen einmal betrachtet werden:

- Flat–Shading:

  Das gesamte Polygon erhält die Farbe, die seinem ersten Vertex zugeordnet worden ist. Dieser Shadingmodus lässt scharfe Kanten eines Objektes besonders erkennen, wenn die verschiedenen Polygone unterschiedliche Farben besitzen. Der Vorteil dieses Shadingmodus liegt in seiner Geschwindigkeit der Darstellung, da keine aufwändigen Berechnungen durchgeführt werden müssen.

- Gouraud–Shading:

  In diesem Shadingmodus wird zwischen den Farben der einzelnen Vertices des Polygons interpoliert. Hier werden also alle angegebenen Farben der Vertices berücksichtigt und verarbeitet. Der Farbwechsel zwischen zwei verschiedenfarbigen Vertices sieht hier fließender aus.

2.2.9 Direct3D Surfaces

DirectX versteht unter einem Surface die Fläche eines Fensters, die jede Applikation nutzt, um zum Beispiel Buttons oder Bilder zu zeichnen. Eine 2D Applikation nutzt für das Erneuern der Fläche die GDI Funktionen. Unter DirectX werden die Funktionen des Interfaces IDirect3DSurface9 verwendet. Die Bereitstellung solcher Funktionen impliziert, dass eine Anwendung niemals direkt auf den Speicherbereich der Flächen zugreifen kann. Eine herkömmliche 2D Anwendung erstellt für gewöhnlich eine Fläche, die für die Darstellung genutzt wird. Direct3D erstellt hingegen einen Frontbuffer, der das aktuelle Bild enthält und dargestellt wird, und optional mehreren Backbuffer (aber mindestens einen!), die als Rendertargets verwendet werden. Die Sammlung dieser Puffer wird als Swapchain bezeichnet. Alle benötigten Angaben, wie das Format oder die Anzahl der Backbuffer werden bei der Instanziierung des Devices benötigt.

Im Zusammenhang mit der Darstellung von den Daten der Surfaces tritt ständig das Problem, das als Tearing bezeichnet wird, auf. Der Tearing Effekt zeigt sich dadurch, dass ein neueres Bild teilweise in ein altes Bild auf dem Bildschirm dargestellt wird. Es entstehen sichtbare Trennlinien. Eine Möglichkeit, den Tearingeffekt zu unterdrücken, besteht darin, auf die vertikale Synchronisation (V–Sync) zu warten und während dessen den Frontbuffer zu rendern. Bei aufwändigen Szenen kann teilweise die Zeit allerdings nicht ausreichen. Die andere Möglichkeit ist das Backbuffering. Für das Backbuffering werden ein oder mehrere Backbuffer benötigt und als Rendertarget gesetzt. Wenn die vertikale Synchronisation stattfindet, wird nur ein Pointer auf den nächsten Backbuffer gesetzt und der aktuelle Frontbuffer wird in die Swapchain als ein Backbuffer eingefügt. Das Umsetzen des Pointers geschieht so schnell, dass ein Tearing effizient unterbunden wird.

2.2.10 Pixel- und Vertexshader

Moderne Grafikhardware bietet die Möglichkeit, sie selber zu programmieren, um unterschiedliche Aufgaben, die in diesem Fall nicht in Hardware implementiert werden müssen, zu erledigen. Diese Programme werden als Shader bezeichnet. Einerseits existieren die Vertexshader, die die Vertices der Objekte transformieren und sich der Vertexfarben annehmen müssen, und andererseits existieren die Pixelshader, die besondere Effekte auf Basis der Texturen berechnen. Ab der DirectX–Version 10 existieren zusätzlich noch die Geometrieshader. Es besteht die Möglichkeit, diese Shaderprogramme in Assembler oder in der HLSL (High Level Shader Language) zu schreiben. Für die Assemblierung der Sourcecodes existieren besondere Shadercompiler, die zur Laufzeit der 3D Anwendung aufgerufen werden. Unter DirectX sind in der D3DX–Api Funktionen zu finden, die unter anderem den Code compilieren.

2.2.11 Die D3DX–API

Die D3DX–Api stellt Funktionen zur Verfügung, die nicht direkt mit der Grafikhardware in Verbindung treten. Hier sind Helferfunktionen zu finden, mit denen zum Beispiel die Transformationsmatrizen erstellt werden oder die für die Verwaltung ganzer Meshes nützlich sein können.

2.2.12 DXUT - Das DirectX Utility

Microsoft stellt mit dem DirectX–SDK das DirectX–Utility zur Verfügung, das auch diese Arbeit nutzt. Dieses Framework nimmt unter anderem dem Programmierer den Setupcode zur Instanziierung eines Devices und auch die Windowsmessageverwaltung ab. Es stehen diverse Callbackfunktionen zur Verfügung, mit denen der Programmierer die Funktionalität des Programmes überwachen kann und in besonderen Ausnahmefällen eingreifen kann. Als Beispiel für eine Reaktion auf einen Ausnahmefall kann das Auftreten eines Lostdevices angeführt werden. Das Utility liegt mit Sourcecode und Projektdateien für Visual Studio dem SDK bei und kann somit statisch oder auch dynamisch zugelinkt werden.

2.2.13 Abschluss

Der DirectX Abschnitt sollte einen allgemeinen Überblick über den Aufbau und die Besonderheiten von DirectX geben. Für ausführlichere Informationen sei der Leser auf die Dokumentation des DirectX–SDKs hingewiesen.

2.3 OpenMP

2.3.1 Einleitung

OpenMP ist ein offener Standard für Multiprozessorsysteme. Heutzutage finden sich schon vermehrt Multicoreprozessoren, die mehrere Cores auf einem Die besitzen, in Standardcomputern. Während Software, die auf Singlecoreprozessoren ausgelegt ist, immer noch weit verbreitet ist, liegt auf zum Beispiel einem Dualcoreprozessor 50% Rechenleistung einfach brach. Natürlich müssen die Algorithmen einer Anwendung auch die Voraussetzung erfüllen, parallelisierbar zu sein. Sind diese Voraussetzungen nicht erfüllt, bleibt nichts anderes übrig, als nur einen Core mit der Arbeit auszulasten. Falls die Voraussetzung für die Parallelisierung gegeben sind, besteht die Möglichkeit, die anderen vorhandenen Cores mit weiteren Threads eines Prozesses auszulasten. In diesem Fall steigt die Rechenleistung der Anwendung beachtlich für den parallelisierten Algorithmus. Damit der Programmierer sich der Threadverwaltung entledigen kann und über Compilerdirektiven die Threaderzeugung steuern kann, wurde OpenMP entwickelt. Da auch diese Arbeit den OpenMP Standard nutzt, sollen im Weiteren die wichtigsten Funktionen von OpenMP vorgestellt werden, um auch für andere Probleme, eine schnelle Lösung zu finden.

2.3.2 Globale Systemvariablen und OpenMP–API Funktionen

Die folgenden globalen Systemvariablen sollten dem Programmierer bekannt sein:

- OMP_NUM_THREADS:

  Es besteht nicht unbedingt die Notwendigkeit, diese Variable im System zu definieren. Sollte sie nicht definiert sein, ermittelt OpenMP die Anzahl vorhandener Systemprozessoren und erzeugt für parallele Bereiche eben diese Anzahl von Threads. Somit würde jeder Systemprozessor einen Thread zugeteilt bekommen. Durch Definition der Variablen und Zuweisung eines positiven Wertes erzeugt OpenMP immer diese angegebene Anzahl von Threads für einen parallelen Bereich. Wenn die Anzahl von Systemprozessoren genau bekannt ist, kann der Programmierer somit selber entscheiden, ob es durchaus sinnvoll sein kann, durchschnittlich auch zwei Threads für einen Systemprozessor zu erzeugen.

- OMP_NESTED

  Diese Variable legt fest, ob bei Betreten eines parallelen Bereiches mit mindestens einem eingebetteten parallelen Bereich für den eingebetteten Bereich wiederum die ermittelte Anzahl von Threads erzeugt wird. Wenn zum Beispiel zwei Systemprozessoren vorhanden sind, werden für den äußeren parallelen Bereich zwei

Threads erzeugt. Beide Threads erreichen nun den eingebetteten parallelen Bereich und es werden nun zwei weitere Threads erzeugt, die mit den eingebetteten parallelen Bereich durchlaufen. Im eingebetteten Bereich sind somit insgesamt vier Threads aktiv. Im Falle von zwei Systemprozessoren bleibt die Gesamtanzahl von Threads überschaubar. Es sollte jedoch sehr genau darauf geachtet werden, ob eingebettete Threads wirklich erwünscht sind, da bei mehreren Systemprozessoren und eventuell mehr eingebetteten Bereichen die Gesamtanzahl von Threads ein Ausmaß erreicht, die das Betriebssystem beeinträchtigt.

Allgemein betrachtet verhält sich die Erzeugung von Threads so, dass der Thread, der einen parallelen Bereich erreicht, zum sogenannten Masterthread wird und so viele Threads erzeugt, dass die Gesamtanzahl (inklusive dem Masterthread) dem Wert der Variablen OMP_NUM_THREADS oder der Anzahl vorhandener Systemprozessoren entspricht.

Nachfolgend sollen die wichtigsten Funktionen, die zur Laufzeit aufgerufen werden können, kurz erklärt werden.

- omp_set_num_threads

  Diese Funktion setzt den Wert von OMP_NUM_THREADS oder die Anzahl von Systemprozessoren außer Kraft. Somit werden nach dem Aufruf dieser Funktion mit einem positiven Wert die Anzahl von Threads erzeugt, wie der übergebene Wert angibt. Es ist hiermit möglich, dynamisch zur Laufzeit die Threaderzeugung mit einem Wert von 1 auszuschalten.

- omp_get_num_threads

  Diese Funktion gibt die Anzahl von Threads im aktuellen Team zurück. Außerhalb eines parallelen Bereiches würde dann der Wert 1 zurückgegeben werden.

- omp_get_max_threads

  Der Rückgabewert entspricht der maximalen Anzahl von Threads, die für einen parallelen Bereich erstellt würden.

- omp_get_thread_num

  Jeder Thread bekommt eine von OpenMP vergebene ID. Der Masterthread hat immer die ID 0. Diese Funktion gibt die ID des aufrufenden Threads zurück.

- omp_get_num_procs

  Durch Aufruf dieser Funktion kann die Anzahl vorhandener Systemprozessoren ermittelt werden.

- omp_in_parallel

  Mit dieser Funktion lässt sich feststellen, ob der aufrufende Thread gerade einen parallelen Bereich abarbeitet.

Die vorgestellten globalen Variablen und Funktionen stellen eine Grundlage dar, mit der es durchaus möglich ist, Algorithmen zu parallelisieren.

2.3.3 parallele Konstrukte

Es sollen nun Konstrukte vorgestellt werden, die dem Compiler signalisieren, dass für die deklarierten Sektionen mehrere Threads erzeugt werden sollen.

Der einfachste Konstrukt bildet der parallele Abschnitt, der wie folgt deklariert wird.

```
#pragma omp parallel
{
}
```

Nach dieser Compilerdirektive kann ein Codeblock aus mehreren Zeilen in geschweiften Klammern stehen. Dieser Bock wird dann von der entsprechenden Anzahl von Threads ausgeführt.

Es ist auch möglich, mehrere sogenannte Sektionen mit dem folgenden Konstrukt zu deklarieren.

```
#pragma omp sections
{
    #pragma omp section
    {
    }
    #pragma omp section
    {
    }
}
```

Im Fall, dass mehr Sektionen wie Threads vorhandenen sind, müssen mehrere Sektionen von einem Thread bearbeitet werden. Im umgekehrten Fall sind einige Threads „arbeitslos“. Es kann dann nicht vorhergesagt werden, welcher Thread, welche Sektion bearbeitet.´

Das nachfolgende Konstrukt ermöglicht die Abarbeitung einer Schleife, für die mehrere Threads erzeugt werden.

```
#pragma omp parallel for
{
    for (int i = 0; i < 3; i++)
    {
        \\Code
    }
}
```

Die Werte des Intervalls [0..2] der Laufvariablen i werden auf die einzelnen Threads aufgeteilt. Es ist sichergestellt, dass niemals mehrere Threads den selben Wert erhalten. Falls die Indizes für einen Zugriff auf ein Array benutzt werden, kann unter Umständen somit auf eine Synchronisierung verzichtet werden. Da in diesem Konstrukt „nur“ die Indexwerte auf die Threads aufgeteilt werden, können keine komplexeren Abbruchbedingungen durch boolsche Ausdrücke programmiert werden.

2.3.4 Synchronisation

Wenn mehrere Threads auf den selben Daten arbeiten, ist eine Synchronisation notwendig. Das folgende Konstrukt ermöglicht die Deklaration eines Codeblockes innerhalb einer parallelen Sektion, die immer nur von einem Thread durchlaufen wird.

```
 #pragma omp critical
{
}
```

2.3.5 Abschluss

In diesem Teil sollte veranschaulicht werden, welche Möglichkeiten OpenMP bietet, um Algorithmen zu parallelisieren. Es wird deutlich, dass durch die einfache Verwendung von Compilerdirektiven durchaus eine Menge Code eingespart werden kann, der die Threadverwaltung übernehmen müsste. Durch OpenMP bleibt der Code weiterhin übersichtlich und wird praktisch „nur“ durch Compilerdirektiven ergänzt. Der einzigste kleine Nachteil könnte sein, dass natürlich ältere Compiler keine OpenMP–Unterstützung besitzen, da es eine Voraussetzung ist, dass von Seiten des Compilers her OpenMP unterstützt werden muss.

3 Technischer Teil

3.1 Einleitung

In diesem Teil der Arbeit soll die softwaretechnische Lösung des in der Einleitung dieser Arbeit beschriebenen Themas dargestellt werden. Dieser Teil wird in folgende Hauptbereiche unterteilt:

- Das Design der Anwendung

  Hier soll dem Entwickler ein Überblick vermittelt werden, aus welchen Komponenten die Software besteht und wie diese im Zusammenhang zueinander stehen.

- CGAL

  Dieser Abschnitt beinhaltet eine Darstellung des Softwaredesigns von CGAL und wie CGAL in diese Arbeit integriert worden ist.

- DirectX

  DirectX wurde zur Darstellung der triangulierten Punktwolken ausgewählt. In diesem Abschnitt sind Informationen über die Einbindung und Nutzung der DirectX–API und die Verwaltung der triangulierten Daten für die Darstellung zu finden.

- Das GUI Interface

  Das GUI Interface bildet die Schnittstelle zu dem Anwender, der eine Punktwolke triangulieren und gleichzeitig darstellen möchte. Hier werden die einzelnen Bedienelemente und deren Funktionalität beschrieben.

- Anhang

  Im Anhang sind wertvolle Informationen über das Projekt selbst enthalten. Sollte ein Entwickler diese Softwareumgebung weiter verwenden oder anderswo integrieren wollen, findet er hier wichtige Informationen, wie Einstellungen für einzelne Teile der Software vorzunehmen sind.

3.2 Das Design der Anwendung

Abbildung 3.1 soll eine Übersicht geben, wie die einzelnen Softwaremodule untereinander in Beziehung stehen.

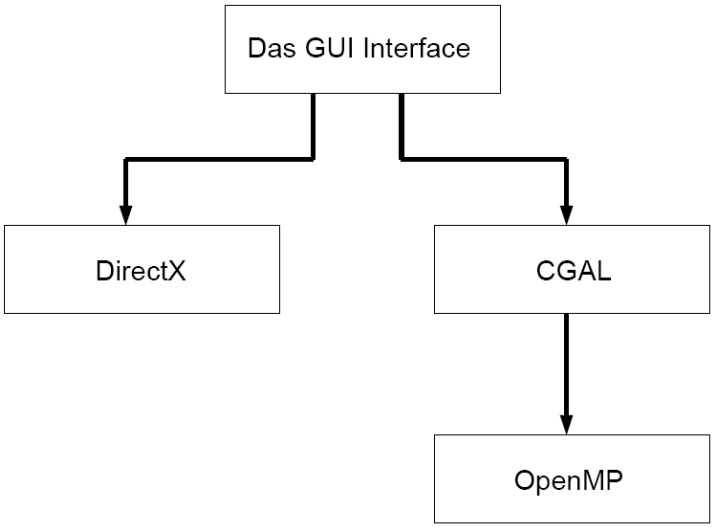

Abbildung 3.1: Das Softwaredesign

Aufgrund von Unverträglichkeiten zwischen DirectX und CGAL wurden die entsprechenden Algorithmen in die jeweiligen Module ausgelagert. Diese Module sind DLLs und können somit theoretisch zur Laufzeit geladen und entladen werden. Die Hauptaufgabe des Programms ist es, aus einer 3D Punktwolke ein Volumenmodell zu berechnen. Dies geschieht in der CGAL–DLL. Sie hält alle benötigten Puffer zur Laufzeit und gibt diese beim Beenden des gesamten Programms wieder frei. Eine nähere Beschreibung dieses Konzeptes ist im Abschnitt 3.4.2 (TriangulatedData) auf Seite 54 gegeben. Das DirectX–Modul dient nur als Visualisierungskomponente, die das berechnete Volumenmodell darstellt. Das Interface des DirectX–Moduls ist so konzipiert, dass zur Compilezeit festgelegt werden kann, ob das Modul zur Laufzeit entladen und somit austauschbar sein soll. Dies ermöglicht die Implementation einer Visualisierung zum Beispiel mit Hilfe von OpenGL. Das CGAL–Modul wird hingegen immer beim Programmstart geladen und beim Beenden entladen, da dies die Hauptkomponente des Programms ist und die Datenpuffer hält. Um bei komplexeren zu berechnenden Volumenmodellen die Rechenleistung eines Multiprozessorsystems auszunutzen, wurde in dem CGAL–Modul OpenMP eingebunden. In dem GUI–Interface ist es auch möglich, mehrere Triangulationen nebeneinander zu berechnen. Diese Berechnungen werden von hierfür erzeugten Threads übernommen. Hiermit ist der Grundstein für die Auslastung von Multiprozessorsystemen gelegt.

3.3 Das GUI–Interface

Das GUI–Interface besteht aus den folgenden Hauptteilen:

- TriangulationDlg

  Hier ist das eigentliche Userinterface programmiert.

- Sphere

  Dieses Modul ermöglicht die Erzeugung einer Kugel mit dem gewünschten Detailgrad. Es wurde zu Testzwecken entworfen.

3.3.1 TriangulationDlg.h

Diese Headerdatei deklariert hauptsächlich die Klasse CTriangulationDlg, welche das GUI–Interface implementiert. Weiter ist ersichtlich, dass die Module CGAL und DirectX zusammen eingebunden werden, wie es in der Designsbeschreibung ersichtlich ist.

3.3.2 TriangulationDlg.cpp

Die folgenden Variablen sind global deklariert:

- g_DXUnloadCriticalSection

  Dieses Mutex synchronisiert das Beenden zwischen der Anwendung und DirectX. Während der Initialisierung des Dialogs wird der Thread für das DirectX–Modul erzeugt. Dieser Thread übernimmt das Rendern der Objekte. Es wird der Pointer des Mutex dem DirectX–Modul übergeben und dort sofort versucht, das Mutex zu reservieren. Falls das DirectX Fenster zuerst geschlossen werden sollte, wird das Mutex freigegeben. Falls das Dialogfenster vor dem DirectXfenster geschlossen werden sollte, wird eine Message zu dem DirectXfenster gesendet (in OnClose), dass dies geschlossen wird. Ist es geschlossen, wurde das Mutex vom DirectX–Thread freigegeben und das Dialogfenster kann auch geschlossen werden. Somit ist es möglich, auf das Rendern zu verzichten und nur noch Triangulationen zu berechnen.

- g_CGAL_DX_Handles_CriticalSection

  Da diese Anwendung das nebenläufige Berechnen von Triangulationen ermöglicht, wird über dieses Mutex der Zugriff auf den STL–Vektor g_CGAL_DX_Handles synchronisiert.

- g_ProgressBarThreadHandle

  Dies ist ein Threadhandle, das verwendet wird, um den Thread, der den Progressbar aktualisiert, zu beenden.

- g_CGAL_DX_Handles

  Hier werden die CGAL und DirectX Handles, die von den jeweiligen Modulen angefordert werden müssen, als Paare in einem STL–Vektor gespeichert. Es ist somit jedes CGAL Handle einem DirectX Handle somit zuordenbar und umgekehrt.

LOAD_DIRECTX_EXPLICIT Definition

Falls dieser Ausdruck im Präprozessor definiert wird, ist es möglich, zur Laufzeit das DirectX–Modul zu entladen und auf ein anderes Darstellungsmodul umzusteigen. Nähere Informationen sind im Abschnitt 3.5.1 auf Seite 64 zu finden.

CTriangulationDlg::OnInitDialog

Hier werden die Mutexe initialisiert und die Limits für die Editfelder festgelegt. Weiterhin wird CoInitialize aufgerufen, um einen Fehler bei installiertem Acrobatreader 7 zu umgehen, der im Dateiöffnendialog auftrat. Falls gewünscht, werden die Funktionspointer für das zur Laufzeit entladbare DirectX–Modul initialisiert. Zuletzt werden der Progressbarthread und der Renderthread erzeugt.

CTriangulationDlg::GetFileName (TCHAR* filename, DWORD strlen, BOOL File_must_exist)

Diese Funktion erzeugt den bekannten Dateiöffnendialog. Diese Funktion wird verwendet, um den Dateinamen zu ermitteln. Wenn in eine Datei geschrieben werden soll, sollte File_must_exist den Wert FALSE, oder wenn gelesen werden soll, den Wert TRUE besitzen. So kann sichergestellt werden, dass die Datei, aus der gelesen werden soll, existiert und spart somit weiteren Code.

CTriangulationDlg::ProgressbarThread (LPVOID thisPtr)

Diese Funktion wird von einem extra erzeugten Thread ausgeführt, der beim Beenden der Anwendung terminiert wird. Es wird über das CGAL–Modul die Anzahl der in die Triangulation einzufügenden Vertices und die Anzahl der aktuell eingefügten Vertices ermittelt und somit der Progressbar gesteuert.

CTriangulationDlg::DirectXThread

Dieser Thread ist der eigentliche Renderthread, der nur im DirectX–Modul arbeitet und über das Mutex g_DXUnloadCriticalSection (Erklärung im Abschnitt 3.3.2 auf Seite 45) mit dem Beenden der Anwendung synchronisiert wird.

CTriangulationDlg::TriangulationThread

Für diese Funktion wird ein Thread erzeugt, wenn eine Triangulation zu berechnen ist. Der Thread führt im CGAL–Modul die Triangulation durch und übergibt die triangulierten Daten an das DirectX–Modul. Dort werden die Daten vom DirectX–Thread gerendert. Das CGAL–Handle und das DirectX–Handle wird in einem STL–Pair Pointer übergeben.

CTriangulationDlg::AlphaShapeThread

Für diese Funktion wird ein Thread erzeugt, der ein Alphashape berechnet und die Daten des Objektes an das DirectX–Modul übergibt. Dem Thread wird als Parameter ein Array mit dem CGAL–Handle, dem DirectX–Handle und dem Alphawert übergeben.

CTriangulationDlg::OnClose

Hier wird der Progressbarthread terminiert. Es wird eine Message an das DirectX–Modul geschickt, damit das DirectX–Fenster geschlossen wird und dann über das Mutex g_DXUnloadCriticalSection (Erklärung im Abschnitt 3.3.2 auf Seite 45) gewartet, bis der DirectX–Thread terminiert.

CTriangulationDlg::OnBnClickedBtnLoadtriangulateddata

Es kann ausgewählt werden, ob eine Datei mit einem wavefrontkonformen Format oder die reinen Rohdaten geladen werden sollen. Die Rohdaten beinhalten schon die Triangulation - also ein darstellbares Objekt. Nähere Informationen zum Wavefrontformat sind im Abschnitt 3.4.3 auf Seite 60 vorhanden.

CTriangulationDlg::OnBnClickedBtnLoaduntriangulateddata

Hier wird eine Berechnung einer Triangulation initiiert und der Triangulationsthread erzeugt. In der auszuwählenden Datei brauchen nur die Vertexdaten zu stehen, die anschließend ausgelesen werden.

CTriangulationDlg::OnBnClickedBtndeltriangulation

Hier kann eine abgeschlossende Triangulation gelöscht werden. Es werden die DirectX- und CGAL–Handles freigegeben und die Triangulation aus der Combobox entfernt.

CTriangulationDlg::OnBnClickedBtnsavetirangulateddata

Hier kann eine Triangulation im wavefrontkonformen Format oder als Rohdaten in einer Datei gespeichert werden. Nähere Informationen zum Wavefrontformat sind im Abschnitt 3.4.3 auf Seite 60 vorhanden.

CTriangulationDlg::OnBnClickedBtnGenerateSphere

Über die Editfelder für die Anzahl der Ringe und Segmente der zu erzeugenden Kugel wird eine Instanz dieser erzeugt und gefragt, ob der Puffer directxkonform sein soll. Die Vertices der Kugel werden immer gleich in eine Datei geschrieben, da sie entweder gleich dargestellt oder erst trianguliert werden müssen. Die Datei ist über die entsprechenden Buttons zu laden.

CTriangulationDlg::OnCbnSelchangeCombotriangulatedObjects

Über diesen Handler werden die triangulierten Objekte ausgewählt und die Checkboxen „enable/disable Rendering of Object“ und „rotate Object“ initialisiert.

CTriangulationDlg::OnBnClickedChkSetrenderstate

Legt fest, ob das Objekt gerendert werden soll. Diese Option wird dann hilfreich, wenn mehrere Objekte renderbar sind und nicht zusammen wegen der Übersichtlichkeit dargestellt werden sollen.

CTriangulationDlg::OnBnClickedChkrotateobject

Standardmäßig drehen sich die Objekte nach dem Laden oder der Triangulation. Dies kann hiermit abgeschaltet werden, so dass das Objekt komfortabler mit der Maus gedreht werden kann.

CTriangulationDlg::OnBnClickedBtnAlphashape

Bevor dieser Button geklickt wird, sollte ein Alphawert im entsprechenden Editfeld eingetragen sein, der dann für die Alphashapeberechnung verwendet wird. Es wird der Alphashapethread erzeugt, der dann die nebenläufige Berechnung übernimmt.

CTriangulationDlg::OnBnClickedBtnoptimizemesh

Es sollte in der Combobox ein Objekt ausgewält werden, dessen Mesh optimiert werden soll. Die Meshoptimiertung hat unter bestimmten Umständen eine Verringerung der Vertices zur Folge. Für die Optimierung wird ein Winkel in Grad benötigt, der in Rad umgerechnet und als Cosinus des Wertes an die OptimizeMesh–Funktion des CGAL–Moduls übergeben wird. Der Meshoptimizer wird in Abschnitt 3.4.3 auf Seite 61 beschrieben.

3.3.3 Sphere.h

Diese Klasse dient zum Testen der Triangulation. Es wird eine Kugel erzeugt, die über die Parameter Anzahl_Ringe und Anzahl_Segmente definiert wird. Somit ist es möglich, die Anzahl der Vertices flexibel zu halten. Außerdem besteht die Möglichkeit, den Vertexpuffer als eine Dreiecksliste (Abschnitt 2.2.4 primitive Daten auf Seite 31) aufzubauen und somit sofort mit DirectX darzustellen.

3.3.4 Sphere.cpp

CSphere::generateSphere(BOOL D3DX_conform)

Diese private Memberfunktion der Klasse CSphere beinhaltet den eigenlichen Algorithmus zur Berechnung der Koordinaten der einzelnen Vertices. Die zu erzeugende Kugel wird in Ringe und Segmente unterteilt. Die Werte dieser beiden Größen werden in den privaten Membervariablen m_numSphereRings und m_numSphereSegments vom Konstruktor gespeichert. Über den Parameter D3DX_conform kann festgelegt werden, ob der Vertexpuffer gleich mit DirectX verwendet werden kann, da hierfür einzelne Vertices dupliziert werden müssen, um eine Dreiecksliste aufzubauen. Wenn der Puffer nicht directxkonfom ist, sind keine Duplikate enthalten und bietet sich somit zur Triangulation an. Der Ringwinkel läuft von 0 bis PI/2 und der Segmentwinkel läuft von 0 bis 2*PI. Es wird außerhalb über den Ringindex iteriert und innerhalb über den Segmentindex, wobei die Vertexkoordinaten von zwei benachbarten Ringen berechnet werden. Somit ist der Aufbau der Dreiecke problemlos möglich.

CSphere::CSphere
(UINT numSphereRings, UINT numSphereSegments, BOOL D3DX_conform)

Die Mindestanzahl der Ringe und Segmente beträgt 3. Mit diesem Ausmaßen ist es erst möglich, einen 3D–Körper aufzubauen. Werte unter dieser Mindestanzahl haben einen leeren Vertexpuffer zur Folge. Die Anzahl der Vertices in Abhängigkeit zur DirectX–Konformität ergibt sich aus den Formeln:

- directxkonform: 2*m_numSphereRings*(m_numSphereSegments+1)
- nicht directxkonform: (numSphereRings-2)*numSphereSegments+2

CSphere::SaveCustomSphereVertexBuffer(TCHAR *filename)

Diese Funktion ermöglicht das Speichern des Vertexpuffers im Wavefrontformat. Dieses Format kann unter anderem von Blender importiert werden, so dass es möglich ist, die Kugel weiter zu bearbeiten. Nähere Informationen zum Wavefrontformat sind im Abschnitt 3.4.3 auf Seite 60 vorhanden.

Die weiteren hier nicht aufgeführten Funktionen sind Helferfunktionen und sind selbsterklärend.

3.4 CGAL

Für die Berechnung der Triangulation und der Alphashapes wurde CGAL als Bibliothek ausgewählt. In diesem Abschnitt soll das Softwaredesign von CGAL und die Einbindung der für die Triangulation und Alphashapes verwendeten Klassen, die CGAL zur Verfügung stellt, vorgestellt werden. Weiterhin werden Techniken für die Speicherverwaltung der triangulierten Daten, das Abspeichern der Daten und eine Idee der Datenreduktion (Meshoptimizer) vorgestellt. Wie auch im DirectX–Modul wird hier das Prinzip des Handles genutzt.

3.4.1 Das Softwaredesign von CGAL

Das Design von CGAL ist in drei große Bereiche aufgeteilt, die bestimmte Funktionalitäten zur Verfügung stellen und somit zur Übersichtlichkeit der Bibliothek beitragen. Diese Bereiche sollen kurz vorgestellt werden:

- Der Kernel

  Der Kernel enthält die geometrischen Daten, wie zum Beipiel Punkte. Für den 3D Raum würden dann drei Punkte den Ort eines Vertex beschreiben. Weiterhin stellt der Kernel Operationen zur Verfügung, die auf diesen Daten arbeiten. Als Beispiel sei die euklidische Distanz oder auch einfach die Berechnung einer Wurzel zu erwähnen. Als Prädikate des Kernels sind dessen Funktionen zu sehen, die auf dessen Grunddaten Operationen ausführen. Diese Operationen können präzise oder auch unpräzise Ergebnisse erzeugen. Die Berechnung dieser Ergebnisse wird als Constuction bezeichnet. Bei unpräziser Berechnung sind Rundungsfehler zu erwarten, die sich natürlich auch in weiteren Berechnungen fortpflanzen können.

- Die Datenstruktur

  Der zweite Bereich besteht aus dem Aufbau der Datenstruktur. Auch für die Triangulationsklasse muss eine Datenstruktur definiert werden, um letztendlich die Triangulation repräsentieren zu können. Diese Datenstruktur muss aus einer sogenannten Vertexbasis und einer Zellenbasis aufgebaut werden. Die Vertexbasis wird wieder mit einem Kernel parametrisiert und definiert den Datentyp „Vertex“. Für den 3D Raum würde dann ein Vertex aus drei Punkten bestehen und die Operationen für diese Punkte stellt der Kernel zur Verfügung. Die Zellenbasis definiert den Typ „Zelle“. Sie besteht im 3D Raum aus vier Vertices, auf die sie ein Handle speichert. Zusätzlich werden auch die Nachbarn eines Vertex gespeichert. Weiterhin sind Kanten und Faces (der Ausdruck für Dreiecke) und ihre Nachbarn definiert. Somit ist es zum Beispiel möglich, anhand der Handles über das konstruierte Objekt zu traversieren.

- nichtgeometrische Funktionalitäten

  Hier werden IO–Funktionen, Cirkulatoren und Randomizer zur Verfügung gestellt, die aus in dieser Arbeit nicht weiter verwendet wurden.

Das gesamte Klassendesign von CGAL ist parametrisiert aufgebaut. Das heißt, es muss zum Beispiel die Triangulationsklasse über ausgewählte Templateklassen als Parameter instanziiert werden. Hierzu sind Templateklassen aus den Bereichen Kernel und Datenstruktur notwendig. Für die meisten Berechnungen ist der Exact_predicates_inexact_constructions_kernel ausreichend. Somit wird er auch in dieser Arbeit vorrangig eingesetzt. Falls exakte Berechnungen gewünscht werden, wurde auch der Exact_predicates_exact_constructions_kernel deklariert. Mit dessen Verwendung müssen dann allerdings die Koordinaten der Verices über Rundungsfunktionen (sind im Sourcecode auskommentiert) ausgelesen werden.

3.4.2 TriangulationCore.h

Hier werden die notwendigen Headerdateien betreffend CGAL und Windows inkludiert. Es ist darauf zu achten, dass die Windowsheaderdateien NACH CGAL inkludiert werden, da sonst Compilerfehler auftreten. Die Arbeit verwendet erstens die normale Triangulation, die mit der Klasse Triangulation_hierarchy_3 berechnet wird und zweitens die Alphashapes, die mit der Klasse Alpha_shape_3 berechnet werden. Dies sind die parametrisierten Toplevelklassen, für die die Templateparameter aufgelöst werden müssen. Nachfolgend soll dies als „Top–Down–Typauflösung“ beschrieben werden:

- Alpha_shape_3<Triangulation> Alpha_shape_3

  Aufgrund der Abhängigkeit der Berechnung eines Alphashapekomplexes von einer Delaunaytriangulation, muss der Templateparameter eine solche angeben.

- Triangulation_hierarchy_3<ADelaunay> Triangulation

  Seitens CGAL wird empfohlen, die Hirarchie zu verwenden, um eine größere Punktmenge verarbeiten zu können. Da innerhalb dieser Hierarchie mehrere Delaunaytriangulationsinstanzen verwendet werden, muss der Parameter die Delaunaytriangulation beinhalten.

- Delaunay_triangulation_3<Ki,ATds> ADelaunay

  Dies ist die Definition des Datentypen für eine Delaunaytriangulation. Wird dieser instanziiert, können Vertices über Memberfunktionen eingefügt werden und somit die Triangulation berechnet werden. Dieser Datentyp hat zwei Templateparameter, dessen

erster den beschriebenen Bereich „Kernel“ und dessen zweiter die „Datenstruktur“ umfasst. Der Kernel ist als grundlegender Typ in diesem Fall nicht weiter parametrisiert. Die Datenstruktur muss aufgrund ihrer zwei Templateparameter weiter aufgelöst werden.

- Triangulation_data_structure_3<AVbh,ACb> ATds

  Dies ist der Typ für die Datenstruktur einer Triangulation. Sie wird erstens aus der Vertexbasis und zweitens aus der Zellbasis zusammengesetzt, die nachfolgend weiter definiert werden.

- Alpha_shape_cell_base_3<Ki> ACb

  Für eine Triangulation wird die Triangulation_cell_base_3 benötigt, die der Alpha_shape_cell_base_3 als Grundlage dient. Daher kann gleich die Zellenbasis für die Alphashapes definiert werden. Die Zellenbasis benötigt nur noch einen Kernel als Templateparameter.

- Triangulation_hierarchy_vertex_base_3<AVb> AVbh

  Aufgrund der Verwendung der Triangulationshierarchie muss zusätzlich die Vertexbasis für die Hierarchie definiert werden. Dies wäre für eine normale Triangulation nicht notwendig.

- Alpha_shape_vertex_base_3<Ki> AVb

  Dieser Typ stellt die Vertexbasis zur Verfügung und hat als Parameter einen Kerneltyp. Die Alphashapevertexbasis stellt wie die Alphashapezellenbasis eine Erweiterung der Triangulationsvertexbasis dar. Daher wird diese gleich hiermit definiert.

Die Bedeutung des Enumtypen VERTEXDATACONSTRUCTIONTYPE wird in der Beschreibung der CTriangulationCore Memberfunktion GetTriangulation erklärt.

Die Struktur CUSTOMTRIANGULATIONVERTEX beschreibt einen Vertex im Puffer der Struktur TriangulatedData. Die Beschreibung dieser Struktur TriangulatedData wird in einem eigenen Abschnitt auf Seite 54 beschrieben. Ähnlich wie in der DirectX.h wird hier das Interface des CGAL–Moduls mit den exportierten und importierbaren Funktionen deklariert (eine Erklärung hierzu ist im Abschnitt 3.5.1 auf Seite 65 zu finden).

globale Variablen

- g_HandleCache

  Dies ist ein STL–Vektor, der Instanzen des Containers HandleCacheElementStructure speichert. Dieser wiederum speichert eine Instanz der Klasse CTriangulationCore mit

ihrem zugehörigen Mutex. Falls kein der Triangulationsinstanz dediziertes Mutex vorhanden wäre, müsste das globale Mutex g_criticalSection verwendet werden und es wären somit keine parallel berechenbaren Triangulationen möglich.

- g_criticalSection

  Dieses Mutex dient nur der Synchronisation des Zugriffs auf g_HandleCache in der exportierten Funktion GetCGALHandle. Beim Freigeben des Handles in der exportierten Funktion ReleaseCGALHandle wird das Mutex für die Triangulationsinstanz genutzt, da nicht mehr auf g_HandleCache zugegriffen werden muss.

- g_TriangulatedDataDummy

  Wird nur gebraucht, wenn ein ungültiges Handle einer exportierten Funktion übergeben wurde.

CTriangulationCore

Die Klasse CTriangulationCore stellt die eigentliche Instanz einer Triangulation oder eines Alphashapes dar. Es müssen vor der Anforderung dieser Berechnungen alle Punkte eingefügt worden sein. Diese Punkte werden in dem STL–Vektor m_PointCache zwischengespeichert und dann der privaten Triangulationsklasseninstanz übergeben. Eine berechnete Triangulation oder ein Alphashape wird aus der CGAL Datenstruktur extrahiert und anschließend in der Struktur TriangulatedData für DirectX aufbereitet gespeichert. Pro Instanz der Klasse CTriangulationCore wird entweder eine Triangulation oder ein Alphashape gespeichert, da die Struktur TriangulatedData innerhalb dieser Klasse nur einmal instanziiert wird. Beide Berechnungsergebnisse nebeneinander in einer Instanz zu halten, würde weiteren unnötigen Aufwand nach sich ziehen, da die dem Alphashape zugrunde liegende Triangulationsinstanz zerstört wird und somit die eigentliche Triangulation für sich berechnet werden muss. Hier kann der Benutzer einfacher die selbe Punktmenge einmal als Alphashape und einmal als Triangulation berechnen lassen.

Die wichtige Rolle der Klasse TriangulatedData

Nach der Berechnung einer Triangulation oder eines Alphashapes werden die Dreiecke (in CGAL „Faces“ genannt) aus der CGAL–Datenstruktur extrahiert (siehe hierzu die Beschreibung GetTriangulation/GetAlphaShape in Abschnitt 3.4.3 auf Seite 56) und für die Darstellung mit DirectX aufbereitet, so dass im DirectX–Modul diese Daten direkt auf die Vertexpuffer für die Grafikhardware aufgeteilt werden können (siehe hierzu Abschnitt 3.5.4 auf Seite 71). Aus dieser unausweichlichen Vorgehensweise wird ersichtlich, dass diese Vertexdaten

für die Darstellung kopiert werden müssen. Um ein weiteres Kopieren bei der Übergabe zwischen dem CGAL– und DirectX–Modul zu vermeiden, wurde die Struktur TriangulatedData entwickelt. Sie soll dafür sorgen, dass die extrahierten Vertexdaten nur einmal als Ergebnis einer Triangulation oder eines Alphashapes im Systemspeicher existieren. Um dies zu realisieren, wird es notwendig, die Referenzen auf diesen Puffer zu zählen, und da der Einsatz in einer Multithreadingumgebung stattfindet, muss für eine entsprechende Synchronisation kritischer Abschnitte gesorgt werden. Nachfolgend werden die Memberfunktionen näher beschrieben und die Implementation dieser Aspekte dort erklärt:

- Der Standardkonstruktor

  Hier werden alle Member initialisiert. Bevor dies allerdings geschieht, wird das Mutex, das pro Instanz vorhanden ist, initialisiert und sofort reserviert, um die Member geschützt vor anderen Threadzugriffen zu initialisieren. Das Mutex ist vorrangig für die Synchronisation der Zugriffe auf den Referenzzähler vorhanden. Da es denkbar ist, dass nach der Speicheranforderung und vor der Initialisierung des Mutex ein anderer Thread im Copykonstruktor oder Zuweisungsoperator aus der Testschleife, ob der Mutexpointer gültig ist, herausläuft, und somit versuchen würde, das uninitialisierte Mutex zu reservieren (es würde eine Exception ausgelöst werden), wird die boolsche Variable „initialised“ vor der Speicheranforderung für das Mutex auf explizit 0 gesetzt und nach der gesamten Initialisierung auf 1 gesetzt. Dies hat den Vorteil, dass die mögliche Exception aufgrund einer Reservierung eines nichtinitialisierten Mutex vermieden wird und dass einer der anderen Threads im Copykonstruktor oder Zuweisungsoperator das Mutex sofort reservieren können.

- Der Copykonstruktor

  Aufgrund der Möglichkeit der sofortigen Initialisierung nach der Deklaration einer Variablen muss auch der Copykonstruktor überschrieben werden. Auch hier muss daher wieder die Synchronisation implementiert werden. Der Aufruf eines Copykonstruktors setzt eine erstmalige Initialisierung (mindestens des Mutex) voraus. Somit ist das Mutex in einem initialisierten Zustand. Der Pointer auf das Mutex wird zuerst kopiert. Sollte der Pointer ungültig sein, wird solange versucht, eine gültige Adresse zu bekommen, bis der Standardkonstruktor gültigen Speicher für das Mutex hat. Ist dies geschehen, wird solange gewartet bis der Standardkonstruktor die vollständige Initialisierung durchgeführt hat. Die Initialisierung wurde vollständig durchgeführt, wenn die boolsche Variable „initialised“ genau den Wert 1 besitzt. Danach kann auf jeden Fall versucht werden, synchronisiert auf den Referenzzähler zuzugreifen.

- Der Destruktor

  Da der Destruktor auf vollständig initialisierten Objekten arbeitet, kann das Mutex sofort reserviert werden. Es wird die boolsche Variable „initialised“ auf 0 gesetzt und

danach das gesamte Objekt zerstört, falls der Referenzzähler auf Null steht, und das Objekt somit insgesamt nicht mehr referenziert wird. Die Freigabe angeforderter Puffer geschieht ausschließlich im CGAL–Modul. Dies entspricht einer sauberen Vorgehensweise, da getestet wird, ob der Speicher vor allem im Debugbuild vom anfordernden Modul auch wieder freigegeben wird. Für den Test wird das Modulhandle (GetModuleHandle) benötigt. Für den Fall, dass eventuell der Zuweisungsoperator nach dem Destruktor für das zerstörte Objekt aufgerufen wird, muss der Threadprogrammierer von außen dafür Sorge tragen.

- Der Zuweisungsoperator

  Dieser Operator muss sicherstellen, dass das aktuelle Objekt und das zu kopierende Objekt vollständig initialisiert sind. Es wäre denkbar, dass zwei Threads zwei verschiedene Objekte initialisieren und ein dritter versucht, gleichzeitig das eine in das andere zu kopieren. Eine weitere Besonderheit ist, dass der Referenzzähler des aktuellen Objektes erniedrigt werden muss bevor kopiert wird. Dies wird durch den Destruktoraufruf des eigenen Objektes implementiert.

3.4.3 TriangulationCore.cpp

In diesem Abschnitt werden die Funktionen der Klasse CTriangulationCore und die exportierten Funktionen des CGAL–Moduls beschrieben.

Die Funktionen der Klasse CTriangulationCore ohne die Funktionen des Meshoptimizers (der Meshoptimizer wird auf Seite 61 beschrieben):

CTriangulationCore::GetTriangulation

Dies ist die Funktion, die letztendlich die Triangulation berechnet und die Daten aus den Datenstrukturen von CGAL in den Puffer von TriangulatedData extrahiert. Falls für die aktuelle Instanz schon eine Triangulation der Vertices im Pointcache berechnet worden ist, wird gleich der fertige Puffer m_finalTriangulationPointSet zurückgegeben. Falls vorher ein Alphashape berechnet worden ist, wird dieses zerstört und anschließend die Triangulation der Punktmenge berechnet. Während die Punkte aus dem Pointcache in die Triangulationsklasse Triangulation_hierarchy_3 eingefügt werden, wird die Variable m_num_Points_in_Triangulation_inserted inkrementiert. Ist die Triangulation fertig berechnet, wird damit begonnen, die CGAL Datenstruktur zu traversieren. Für die Extraktion der Hülle des triangulierten Objektes ist die Erkenntnis notwendig, dass auch Zellen existieren, die unbegrenzt sind. Diese unbegrenzten Zellen besitzen mindestens einen „unendlichen“ Vertex. Daraus kann gefolgert werden, dass die Faces, die die Oberfläche des Objektes

darstellen, aus unbegrenzten Zellen bestehen. Dies ergibt sich aus der Tatsache, dass jede Zelle vier Nachbarn besitzt. Im Falle des Oberflächenfaces, ist dieses einerseits Teil einer inneren begrenzten Zelle und andererseits Teil einer unbegrenzten Zelle, die einen „unendlichen“ Vertex besitzt. CGAL stellt Iteratoren bereit, die einerseits alle begrenzten und andererseits alle Zellen traversieren. Somit muss über alle Zellen iteriert werden und immer getestet werden, ob die aktuelle Zelle unbegrenzt ist. Ist eine unbegrenzte Zelle gefunden worden, muss ermittelt werden, welcher Vertex der Zelle der „Unendliche“ ist. Das dem „unendlichen“ Vertex gegenüberliegende Face ist das zu extrahierende Dreieck, dessen Vertices im Uhrzeigersinn mit Hilfe der von CGAL definierten Orientierung ausgelesen werden. Informationen zur von CGAL definierten Orientierung von Zellen sind im Abschnitt 2.1.2 auf Seite 13 zu finden. Diese Extraktionstechnik wird angewendet, wenn D3DXCONFORM_INFINITECELLTRAVERSE_TRIANGLELIST angegeben wird. Der Vollständigkeit halber wurde mit D3DXCONFORM_FINITECELLTRAVERSE_TRIANGLELIST eine Extraktion des „Innenlebens“ des Objektes implementiert, bei der der Test auf den „unendlichen“ Vertex entfällt und alle Faces der begrenzten Zellen extrahiert werden. Hierbei fallen allerdings größere Datenmengen an, deren inneren Dreiecke von der Hülle des Objektes während der Darstellung verdeckt werden.

CTriangulationCore::GetAlphaShape

Die Extraktion der Alphafaces geschieht analog zu der Extraktion der Hüllenfaces der Triangulation. Hier wird allerdings ein Backinserter initialisiert, der dann alle Faces des Alphakomplexes enthält. Über diesen wird iteriert und das Face des gegenüberliegenden (im Iterator enthaltenen) Vertex extrahiert. Der Iterator enthält ein Paar, das als erstes Element das Zellenhandle und als zweites Element den Index des Vertex in der Zelle enthält. Somit sind wieder vier Fallunterscheidungen aufgrund des Vertexindexes zu programmieren. Da hier die Lage relativ zum ganzen Objekt nicht ermittelt werden kann, muss für die Darstellung das Backfaceculling (Abschnitt 2.2.3 auf Seite 29) ausgeschaltet werden. Für jedes Face kann nicht bestimmt werden, welche Seite die Oberfläche ist.

CTriangulationCore::InsertPoint

Hier wird der Vertex in den Pointcache eingefügt und der globale Vertexzähler erhöht.

CTriangulationCore::GetGlobalNumPoints

Diese Funktion gibt die Anzahl der eingefügten Punkte in den Pointcache zurück.

CTriangulationCore::GetNumPointsInTriangulationInserted

Diese Funktion gibt die Anzahl der eingefügten Punkte in die Triangulation zurück. Die private Variable m_num_Points_in_Triangulation_inserted wird während des Einfügens in die CGAL–Klasse erhöht. Somit ist es möglich, zusammen mit der Funktion GetGlobalNumPoints den aktuellen Status der Berechnung der Triangulation von außen darzustellen. Diese Darstellung wird von der ProgressbarThread–Funktion der GUI genutzt.

CTriangulationCore::DumpTriangulationPointSet

Diese Funktion wird von der exportierten Funktion SaveCGALBuffer aufgerufen. Dies soll eine schnellere Alternative des Abspeicherns der triangulierten Daten ermöglichen, da bei einer hohen Anzahl von Vertices das Abspeichern im Wavefrontformat sehr zeitaufwändig ist.

CTriangulationCore::LoadTriangulationPointSet

Diese Funktion wird von der exportierten Funktion LoadCGALBuffer aufgerufen, die das zugehörige Handle zurückgibt. Die ausgelesene Datei sollte den Puffer von TriangulatedData enthalten, da diese Daten so direkt geladen werden.

Die exportierten Interfacefunktionen des CGAL–Moduls:

GetCGALHandle

Diese Funktion erzeugt eine neue Instanz für eine Triangulation inklusive des Mutex für den synchronisierten Zugriff auf die Triangulation. Außerhalb des CGAL–Moduls ist nur das Handle sichtbar.

ReleaseCGALHandle

Diese Funktion löscht die Instanz einer Triangulation. Das Mutex wird beim Entladen der DLL freigegeben. Das Mutex wird beim Entladen der DLL zerstört.

LoadUnTriangualtedData

Es wird davon ausgegangen, dass die angegebene Datei nur Vertexdaten enthält, die ausgelesen werden. Die Funktion fügt die Vertices mit der Funktion InsertPoint in eine neue Instanz im CGAL–Modul ein. Das Handle für die Instanz wird zurückgegeben. Anschließend kann der User über das Handle die Triangulation berechnen lassen.

LoadTriangulatedData

Hier wird die angegebene Datei mit den deklarierten Vertices und Faces ausgelesen. Die Vertices werden in einem Vektor gespeichert. Die Facedaten, die die Indizes in diesen Vektor enthalten, ermöglichen den letztendlichen Aufbau des Puffers von TriangulatedData.

containsVertex

Dies ist die einzigste Helferfunktion für SaveTriangulatedData, die nicht exportiert wird. Sie wird in der nachfolgend erklärten Funktion SaveTriangulatedData benutzt und testet, ob ein übergebener Vertex in der übergebenen Liste vorhanden ist. Diese Funktion ist mit OpenMP für eine Multithreadingsuche optimiert. Generell wird zuerst ab der Mitte der Liste angefangen zu suchen, da es wahrscheinlicher ist, dass der Vertex später eingefügt wurde. Dies haben Tests bestätigt. Falls der Vertex im ersten Durchlauf nicht gefunden wurde, wird vom Anfang der Liste bis zur Mitte gesucht. In den OpenMP Blöcken wird ein global für jeden Thread sichbarer Iterator synchronisiert weitergesetzt. Jeder Thread kopiert sich den aktuellen Stand des Iterators in seinen privaten Iterator und vergleicht mit ihm den adressierten Vertex. Sollte ein Thread den Vertex in der Liste gefunden haben, bricht eine boolsche Variable die globale Suchschleife für alle Threads ab und der fündig gewordene Thread speichert seinen privaten Iterator in einer globalen Variable ab. Zuletzt muss dann der Offset des gefundenen Vertex in der Liste ermittelt werden, da dieser zurückgegeben wird. Zusätzliche Informationen in Bezug auf OpenMP sind im gleichnamigen Kapitel auf Seite 39 verfügbar.

SaveTriangulatedData

Diese Funktion speichert die Triangulation im Wavefrontformat (Seite 60) ab. Hierfür muss eine Vertexliste und eine Faceliste (Dreiecksliste) erstellt werden. Da ein Vertex mehreren Dreiecken angehören kann, muss getestet werden, ob der Vertex in der zu erstellenden Liste schon enthalten ist. Dies wird von der Funktion containsVertex festgestellt. Sie gibt den Index des vorhandenen Vertex in der Liste oder „nicht gefunden“ zurück. Mit Hilfe der ermittelten Indizes können die einzelnen Faces beschrieben werden. Tests haben gezeigt,

dass die Suche bei einer sehr hohen Anzahl von Vertices lange dauern kann. Daher wurde die Funktion containsVertex für Multiprozessorsysteme mit OpenMP optimiert.

ReleaseTriangulationdata

Diese Funktion wird nur für den Destruktor von TriangulatedData exportiert. Für nähere Informationen sei der Leser auf die Beschreibung des Destruktors auf Seite 54 hingewiesen.

Die folgenden Funktionen sind nur synchronisierte Wrapper für Memberfunktionen der Klasse CTriangulationCore und bedürfen keiner ausführlicheren Beschreibung:

- GetTriangulationShape
- OptimizeMesh
- GetAlphaShape
- InsertPoint
- GetGlobalNumPoints
- GetNumPointsInTriangulationInserted
- SaveCGALBuffer
- LoadCGALBuffer

Export der triangulierten Daten in das Wavefront–Format

Um im Wavefrontformat zu speichern, müssen die einzelnen Vertices jeweils in einer Zeile stehen und mit dem Buchstaben „v“ am Anfang der Zeile gekennzeichnet werden. Die drei Komponenten eines Vertex werden durch jeweils ein Leerzeichen getrennt. Nach der Auflistung der Vertices können die Dreiecke definiert werden. Die Reihenfolge der Vertices in der Liste gibt die Indizierung dieser vor. Um ein Dreieck (Face) zu definieren, ist am Anfang der Zeile ein „f“ notwendig, gefolgt von drei Indizes, die die Ecken des Dreiecks bedeuten. Es sei hier angemerkt, dass der Export von Objekten mit einer sehr hohen Anzahl von Vertices im Wavefrontformat sehr rechenintensiv ist. Daher wurde die Möglichkeit gegeben, das Objekt als Rohdaten abzuspeichern. Die Rohdaten lassen sich natürlich dann nicht mit externen Werkzeugen laden und dort weiterverarbeiten.

Der Meshoptimizer

Der Meshoptimizer wurde entwickelt, um die Datenmenge eines vorhandenen Meshes so zu reduzieren, dass der resulierende niedrigere Detailgrad nicht auffällt. Als Optimierungskriterium wurden kleine Winkeldifferenzen zwischen den einzelnen Normalen der Dreiecke des Meshes gewählt. Betrachtet wird immer ein Vertex und somit alle Dreiecke, denen dieser Vertex angehört. Der Benutzer kann einen Winkel angeben, der größere Differenzen zulässt. In diesem Fall würde natürlich der sehr viel niedrigere Detailgrad sichtbar werden. Der Meshopimizer kann theoretisch auf jede Datenmenge angewendet werden, da die Implementation so gewählt wurde, dass nicht eine zugrundeliegende Triangulation vorhanden sein muss. Es sei hier am Rande erwähnt, dass es nicht möglich gewesen wäre, den Meshoptimizer auf ein Alphashape anzuwenden, da die zugrundeliegende Triangulation des Alphashapes seitens CGAL zerstört wurde. Der Meshoptimizer arbeitet auf dem Puffer von TriangulatedData, der mit einer auch nicht wavefrontkonformen Datei initialisiert werden kann. Somit ist er praktisch universell einsetzbar. Der nachfolgende Pseudocode soll grob den algorithmischen Ablauf den Meshoptimizers veranschaulichen:

Algorithm 3.4.1: OPTIMIZEMESH($FLOAT\ maxAngle\_in\_rad$)

```
BOOL optimiert = FALSE
while (!optimiert)
   do { optimiert = TRUE;
        for index ← 0 to Groesse_des_Vertexpuffers − 1
            do { Vertex = VertexPuffer[index];
                 if (!ist_Vertex_ein_Nachbar_eines_loeschbaren_Hauptpunktes
                  and !ist_Vertex_in_Testpunktliste_vorhanden)
                    then { erstelle_neuen_Testpunkt_mit_seinen_Randpunkten;
                           if (ist_Testpunkt_loeschbar)
                           { in_Liste_mit_loeschbarem_Hauptpunkten_eintragen;
                             optimiert = FALSE; }
        Triangulation_der_Randpunktmenge_der_loeschbaren_Testpunkte;
        triangulierte_Randpunkte_und_Testpunkte_
        durch_Neuanordnung_des_Originalpuffers_loeschen;
        triangulierte_Loecher_fuellen;
```

Der Testpunkt ist der aktuelle Vertex, dessen Löschbarkeit festgestellt werden muss. Der Vertex ist so oft im Originalpuffer vorhanden, wie die Anzahl der Dreiecke, deren Mitglied er ist. Somit muss auch getestet werden, ob der Vertex nicht schon als zu löschender Punkt in einer Liste vorhanden ist.

Ist der aktuell betrachtete Vertex ein Randpunkt eines löschbaren Vertex, wird er vorerst nicht in die Liste der löschbaren Punkte aufgenommen. Falls man dies dennoch machen würde, müsste man die Randpunktmengen des aktuellen Vertex und die des anderen löschbaren Vertex vereinigen. Weiterhin müssten von allen Normalen der Dreiecke, die sich innerhalb der vereinigten Randpunktmenge befinden, die Winkeldifferenzen gegeneinander getestet werden, um festzustellen, ob die inneren Punkte gelöscht werden können. Im Falle einer Kugel entstehen somit große Flächen, deren inneren Punkte gelöscht werden. Wenn dann ein Differenzwinkel den maximalen Winkelwert übersteigt, könnte die gesamte Fläche vorerst nicht optimiert werden. Es müsste dann entweder geschickt gefiltert werden, welche inneren Punkte dann nicht gelöscht werden oder die letzten Vertices, die die Randpunktmenge erweitert haben, nach und nach wieder aus der Menge der löschbaren Punkte herausgenommen werden. Insgesamt würde mit dieser Vorgehensweise, die Kugel nicht gleichmäßig optimiert werden. Der implementierte Algorithmus hingegen erzeugt einzelne „Waben“, die sich gleichmäßig über die Kugel verteilen würden.

Wenn der Vertexpuffer vollständig traversiert wurde, ist eine Liste mit löschbaren Punkten und deren Randpunkte erstellt worden. Nun werden die Randpunkte der löschbaren Hauptpunkte neu trianguliert. Somit sind die Hauptpunkte indirekt schon gelöscht worden. Die Koordinaten der Hauptpunkte und der Randpunkte werden beim Einfügen in die temporäre Triangulation im Vertexpuffer auf den Maximalwert eines Floats gesetzt und somit als gelöscht markiert. Für diese Markierung wurden die Indizes der zu löschenden Hauptpunkte und der Randpunkte im Vertexpuffer in einem Array gespeichert. Zu diesem Zweck wurde die Struktur Point_to_delete entwickelt, die in der Klasse CTriangulationCore privat deklariert ist.

Jede temporäre Triangulation wird mit GetTriangulation in eine DirectX–konforme Dreiecksliste umgewandelt, die später in den aufbereiteten Vertexpuffer eingefügt wird. Es ist einleuchtend, dass ein Punkt, der auf einer ebenen Fläche liegt, somit gelöscht wird. Wird allerdings eine ebene Fläche mit CGAL trianguliert, können keine Dreiecke extrahiert werden, da CGAL keine begrenzten Zellen erzeugt. Diese Randbedingung wird abgefangen und es wird zusätzlich der Nullpunkt in die Triangulation eingefügt. Da es unwahrscheinlich ist, dass auch der Nullpunkt auf dieser Fläche liegt, können nun die Dreiecke extrahiert werden. Die Dreiecke, die den Nullpunkt als Mitglied besitzen werden vernachlässigt.

Es ist nicht auszuschließen, dass die triangulierte Punktmenge des Loches neben der natürlich neuen gewünschten Oberfläche des gesamten Objektes auch Dreiecke enthält, die von der Gesamtoberfläche bei der Darstellung verdeckt werden. Dieser als minimaler Overhead eingestufte Aufwand wird akzeptiert, da der Rechenaufwand so schon groß genug ist. Bei dem notwendigen Einfügen des Nullpunktes entstehen indes keine überflüssigen Dreiecke, da diese, wie schon erwähnt, vernachlässigt werden können.

Nachdem die gesamte Dreiecksliste der temporären Triangulationen erstellt wurde, wird der

Ausgangsvertexpuffer neu geordnet, das heißt, es werden die auf den maximalen Floatwert gesetzten Koordinaten beim Kopieren vernachlässigt und die neue Dreiecksliste hinten angehängt.

Dieser Vorgang wird so oft wiederholt, bis kein löschbarer Punkt mehr gefunden worden ist.

Anzumerken sei noch, dass einzelne rechenintensive Bereiche des Meshopimizers mit OpenMP parallelisiert wurden.

Nachfolgend soll die Funktion isMainPointDeletable beschrieben werden, da diese für den Test der Dreiecksnormalen verantwortlich ist.

isMainPointDeletable

Dieser Funktion wird ein auf Löschbarkeit zu testender Punkt in der Struktur Point_to_delete übergeben. Zuerst werden für alle Dreiecke, die von den gespeicherten Randpunkten gebildet werden und als Mitglied den Hauptpunkt besitzen, die Normalen über das Kreuzprodukt berechnet. Anschließend werden von allen Normalen die Winkeldifferenzen berechnet und gegen den übergebenen Schwellwert verglichen. Es sei hier am Rande noch erwähnt, dass, wenn nur Winkeldifferenzen benachbarter Normalen verglichen werden würden, eine Kegelspitze vollständig eliminiert werden würde. Diese Funktion wird natürlich hauptsächlich aufgerufen, um einen löschbaren Punkt später auch zu löschen. Sie wird allerdings auch von der Funktion isVertexInBoundaryOfDeletableMainPoint aufgerufen, die ermittelt, ob ein Punkt ein Randpunkt eines zu löschenden Punktes ist. Hierfür wurden schon die Normalen des zu löschenden Punktes berechnet und getestet. Somit kann die Entscheidung, ob der Punkt löschbar ist, gleich zurückgegeben werden.

3.5 DirectX

In diesem Abschnitt wird die Implementation des DirectX–Moduls beschrieben. Dieses Modul ist in zwei Softwaremodule unterteilt. Das DirectX–Interface, das von der GUI angesprochen wird, ist in der DirectX.h und der DirectX.cpp implementiert. Hier wird die Funktionalität der Kamerafahrt und Rotation um das Objekt implementiert. Weiterhin befinden sich hier die Funktionen, mit denen ein Handle auf ein trianguliertes Objekt angefordert und wieder freigegeben werden kann. Es ist zuerst ein Handle anzufordern, und dann die triangulierten Daten für das zu rendernde Objekt zu setzen.

Die TriangulationObject.h und die gleichnamige cpp–Datei implementieren das Rendering und die Verwaltung der triangulierten Objektdaten. Für jedes triangulierte Objekt wird eine Instanz dieser Klasse erzeugt.

3.5.1 DirectX.h

Hier sind alle zu exportierenden Funktionen des DirectX–Moduls nach außen hin sichtbar definiert. Mit der Präprozessordefinition LOAD_DIRECTX_EXPLICIT wird festgelegt, ob das Modul zur Laufzeit entladbar sein soll. Ist dies nicht definiert, wird das DirectX–Modul wie auch das CGAL–Modul beim Programmstart geladen und bleibt im Speicher, bis das Programm beendet wird. Für diesen Fall generiert der Linker universelle Namen für die exportierten Funktionen. Für den Fall, dass das Module zur Laufzeit entladbar sein soll, kann der Linker angewiesen werden, selbst definierte Namen für die exportierten Funktionen zu vergeben, die dann mit der Windows–API–Funktion GetProcAddress verwendet werden können. Nachfolgend soll diese Fallunterscheidung einmal übersichtlich dargestellt werden:

explizites Laden und Entladen des Moduls

Die folgende Linkeranweisung hat in der entsprechenden Funktion zu stehen, damit nur der Funktionsname in die Exporttabelle aufgenommen wird:
#pragma comment(linker, “/EXPORT:“__FUNCTION__“=“__FUNCDNAME__“,PRIVATE“).
Das importierende Modul muss sich die Funktionsadresse mit der Windows–API–Funktion GetProcAddress geben lassen.

Laden beim Programmstart und Entladen beim Programmende

Jede zu exportierende Funktion muss vor ihrer Signatur folgende Deklaration besitzen: __declspec (dllexport). Ein importierendes Modul muss diese Funktion mit __declspec (dllimport) deklarieren. Der Linker trägt diese Daten in die Importtabelle des imporierenden Moduls ein.

Da die DirectX.h einerseits von außen und andererseits intern verwendet wird, ist intern DX_INTERNAL_INCLUDE definiert, um die Exportdeklarationen vorzunehmen. Wird diese Headerdatei von einem anderen Modul includiert, sind die Importdeklarationen definiert. Im Fall, dass das DirectX–Modul zur Laufzeit geladen wird, entfallen somit die Importdeklarationen wie oben erklärt wurde.

3.5.2 DirectX.cpp

Folgende globale Daten sind im Modul deklariert:

- std::vector<p_CTriangulation> g_TriangulationObjectHandleCache

  In diesem STL–Vektor werden die Instanzen der Klasse CTriangulation gespeichert. Alle nicht mit ReleaseTriangulationObjectHandle freigegebenen Instanzen werden beim Entladen der DLL gelöscht.

- CRITICAL_SECTION g_criticalSection

  Dieses Mutex dient der Synchronisation des Zugriffs der Threads auf die globale Variable g_TriangulationObjectHandleCache.

- LPCRITICAL_SECTION g_DXUnloadCriticalSection

  Dies ist der Pointer auf das Mutex für das synchronisierte Entladen der DirectX–DLL. Siehe hierzu 3.3.2 auf Seite 45.

- FLOAT g_xzangle, g_heightangle, g_radius

  Diese Variablen dienen der Speicherung der Position der Kamera. Das zu rendernde Objekt befindet sich im Nullpunkt des Koordinatensystems, und die Kamera bewegt sich auf einer imaginären Sphäre um den Ursprung.

- int g_xold, g_yold

  Diese Koordinaten werden für die Berechnung der neuen Position der Kamera benötigt. Sie entsprechen den alten Mauskoordinaten.

- BOOL g_captureMouse

  Diese Variable bestimmt, ob die Windowmessages der MouseMove Events für die Drehung der Kamera um das Objekt bei Verlassen des Mauscursors des DirectX–Fensters verarbeitet werden müssen. Mit dieser Technik ist eine benutzerfreundlichere Steuerung möglich, da das DirectX–Fenster nach der Initialisierung erstmal relativ klein gehalten wurde.

- HWND g_DXUT_HWND

  Der Wert dieses Windowhandles wird von der Funktion GetDirectXWindowHandle zurückgegeben, falls das Fenster von außerhalb des DirectX–Moduls bekannt sein muss. Ein Beispiel hierfür ist das Beenden des gesamten Programms, während dessen eine Nachricht zu dem Fenster gesendet wird. Siehe hierzu GetDirectXWindowHandle auf Seite 68.

- extern D3DXMATRIX g_TriangulationMatView, g_TriangulationMatProj

  Diese Matrizen sind für die D3D–Pipeline und den Vertexshader notwendig und sind in der TriangulationObject.cpp deklariert, da sie dort direkt genutzt werden müssen. Sie werden in DirectX.cpp für die Rendervorgänge initialisiert.

Die folgenden Funktionen werden vom DirectX–Modul exportiert:

GetTriangulationObjectHandle

Intern werden die Daten der triangulierten Objekte in Instanzen der Klasse CTriangulation gekapselt. Es wird nur ein sogenanntes Handle nach außen gegeben, dass diese Funktion als Rückgabewert besitzt. Dieses Handle wird intern als Index für den globalen STL–Vektor g_TriangulationObjectHandleCache verwendet, das somit das triangulierte Objekt identifiziert. Der Zugriff auf den Vektor wird synchronisiert.

ReleaseTriangulationObjectHandle

Das mit GetTriangulationObjectHandle angeforderte Handle wird hiermit zerstört und die Objektdaten gelöscht. Auch hier wird der Zugriff auf den globalen Vektor synchronisiert.

SetRenderStateOnTriangulationObject

Der Status, ob das gesamte Objekt gerendert - also dargestellt - werden soll, wird festgelegt. Standardmäßig wird gerendert. Dies ermöglicht die Einzelbetrachtung der Objekte, obwohl mehrere vorhanden sein können.

GetRenderStateOnTriangulationObject

Gibt den Status, ob das Objekt gerendert wird, zurück.

SetRotateObjectStateOnTriangulationObject

Mit dieser Funktion lässt sich für jedes Objekt festlegen, ob es sich von alleine drehen soll.

GetRotateObjectStateOnTriangulationObject

Diese Funktion gibt den Status, ob sich das Objekt dreht, zurück.

SetBuiltTriangulationData_exp

Über diese Funktion werden die triangulierten Daten der einzelnen Objekte gesetzt. Der Aufruf erfolgt also nach der Anforderung eines Handles und der Triangulation. Siehe hierzu auch die Memberfunktion der Klasse CTriangulation::SetBuiltTriangulationData auf Seite 71.

Initialize

Dies ist die eigentliche Funktion zur Initialisierung des DirectX–Moduls. Es wird der Pointer auf das Mutex für die Synchronisation während des späteren Entladens und das DirectX–Utility initialisiert. Über das DirectX–Utility wird das Renderfenster und das D3D–Device erzeugt. Weiterhin werden die Callbackfunktionen registriert und der Renderstate für die Beleuchtung ausgeschaltet.

GetDirectXWindowHandle

Das Handle für das DirectX–Fenster wird für die WM_DESTROY Message benötigt. Somit wird das Fenster von der GUI geschlossen, wenn die GUI zuerst beendet wird. Siehe Abschnitt 3.3.2 auf Seite 47.

Folgende Funktionen werden nicht von dem DirectX–Modul exportiert und sind bis auf die DLLMain Callbackfunktionen des DirectX–Utilities:

DeviceLostNotification

Diese Callbackfunktion wird aufgerufen, wenn das D3D–Device den Zustand eines Lostdevices angenommen hat. Sie wird vor der Resetfunktion des D3D–Devices aufgerufen, um zu signalisieren, dass alle Ressourcen verloren gegangen sein könnten. Siehe hierzu die Abschnitte 2.2.7 und 2.2.3.

DeviceResetNotification

Diese Callbackfunktion wird nach dem Aufruf der Resetfunktion des D3D–Devices aufgerufen um zu signalisieren, dass alle Ressourcen, die verlorengegangen sind, jetzt wieder neu geladen werden können. Zu beachten ist hierbei, dass nicht vergessen werden darf, die Renderstates, die von den Standardwerten abweichen, jetzt abermals gesetzt werden müssen. Siehe hierzu die Abschnitte 2.2.7 und 2.2.3.

KeyboardEventNotifikation

In dieser Callbackfunktion wird registriert, ob die Pfeil–Oben oder die Pfeil–Unten Tasten gedrückt wurden. Über diese Tasten wird der Abstand der Kamera zum Objekt (Ursprung) berechnet.

MessageFilter

Diese Callbackfunktion des DirectX–Utilities ist der letzte Ausweg, um alle Windowmessages zu erhalten. Sie musste initialisiert werden, da keine WM_MOUSEMOVE Events an die MouseEventNotifikation Callbackfunktion gesendet werden. Aus der vom letzten Event gespeicherten Mausposition und der aktuellen übergebenen Position wird der Offset der Winkel zu der Horizontalebene und zu der Vertikalebene berechnet. Diese Winkel werden global

gespeichert und von der Funktion FrameMove für die Berechnung der Transformationsmatrizen verwendet. Falls ein WM_MOUSEMOVE ohne gedrückten linken Mausbutton auftritt, wird das Mauseventcapturing außerhalb des DirectX–Fensters deaktiviert.

MouseEventNotifikation

Falls der linke Mausbutton gedrückt wurde, wird das Mauseventcapturing außerhalb des DirectX–Fensters aktiviert. Dies hat zur Folge, dass es möglich ist, die Objekte mit Hilfe der WM_MOUSEMOVE Message weiterhin zu drehen. Wird diese Technik nicht benutzt und die Maus verlässt die Oberfläche des DirectX–Fensters, würden keine WM_MOUSEMOVE Events mehr für die Berechnung der Winkel empfangen werden. Das Objekt würde einfach nicht mehr weitergedreht werden.

FrameMove

Diese Funktion bedeutet eine Aufforderung seitens DirectX, dass die gesamte Szene aktualisiert werden muss. Es soll aber noch nicht gerendert werden, daher wird auch kein Pointer auf die Deviceinstanz übergeben. Die Aktualisierung der Szene beinhaltet unter anderem die Objektverschiebung, Objektskalierung und Neuausrichtung der Kamera. Die globalen Transformationsmatrizen werden hier initialisiert und aktualisiert. Die Funktion SetTransform setzt die Transformationsmatrizen für die Fixed–Function–Pipeline. Die Viewtransformationsmatrix entspricht hier der Beschreibung Kameraposition. Der Vektor vEyePt bedeutet die aktuelle Position der Kamera im Raum. Dieser Vektor wird mit dem Abstand g_radius initialisiert. Danach wird diese Position um den aus den Mausevents berechneten Winkel g_xzangle in der Horizontalebene um die Y–Achse gedreht. Anschließend wird von der Position aus um den Winkel g_heightangle in der Vertikalebene gedreht. Für diese Berechnung ist zu beachten, dass die Drehachse auf keiner der Grundachsen liegt und somit über das Kreuzprodukt der Horizontalebenenposition und der Y–Achse diese berechnet werden muss. Die D3DX–Funktion D3DXMatrixRotationAxis stellt diese Berechnung zur Verfügung. Da die Welttransformationsmatrizen privat für jedes Objekt sind, werden zum Schluss alle vorhandenen Instanzen zur Aktualisierung aufgefordert.

Render

Hier wird nur das Rendertarget (letztendlich das DirectX–Fenster) gelöscht und die einzelnen Instanzen zum Rendern ihrer triangulierten Objekte aufgefordert.

DllMain

Diese Funktion wird vom System aufgerufen, wenn das Modul von einem Prozess geladen oder ein neuer Thread erzeugt wird. Für das Anbinden an einen Prozess müssen unter anderem Mutexe initialisiert werden. Beim Entladen werden alle vorhandenen Instanzen gelöscht.

3.5.3 TriangulationObject.h

Diese Headerdatei deklariert die Klasse CTriangulation. Sie ist als Container für die triangulierten Vertexdaten inklusive die für die Darstellung mit DirectX benötigten Datentypen anzusehen . Für die Darstellung mit DirectX werden die folgenden privat deklarierten directxspezifischen Datentypen benötigt:

- LPDIRECT3DVERTEXBUFFER9 *m_pD3DTriangulationVertexBuffer
- LPDIRECT3DVERTEXDECLARATION9 m_pTriangulationVertexDeclaration
- LPDIRECT3DVERTEXSHADER9 m_pTriangulationShader

Die Membervariable m_builtTriangulation hält die originalen triangulierten Vertexdaten.

Für die Beschreibung der Vertexdaten mit Hilfe der FVF–Codes wurden folgende Konstanten verwendet:

- D3DFVF_DIFFUSE

  Legt fest, dass ein Vertex zusätzlich eine Komponente für diffuses Licht besitzt.

- D3DFVF_XYZ

  Legt fest, dass die Vertexkoordinaten als untransformierte Koordinaten vorliegen

Mit der Definition VERTEX_ELEMENTS_PER_BUFFER wird aufgrund der begrenzten Vertexpuffergröße die Maximalgröße festgelegt. Folglich werden die Vertexdaten in der Membervariablen m_builtTriangulation auf mehrere erzeugte LPDIRECT3DVERTEXBUFFER9 verteilt, falls ein einzelner Puffer nicht ausreicht.

3.5.4 TriangulationObject.cpp

Der Konstruktor und Destruktor sowie die Getter und Setter werden hier nicht weiter beschrieben, da sie triviale Funtionalität besitzen.

CTriangulation::SetBuiltTriangulationData

Diese Funktion dient unter anderem zum Initialisieren der D3D–Vertexpuffer des Objektes. Diese Daten werden der Membervariablen m_builtTriangulation zugewiesen. Dessen Puffer wird dann auf die entsprechende Anzahl von D3D–Vertexpuffern verteilt, damit das Objekt gerendert werden kann. Die Vertexpuffer werden als Standardressourcen angefordert (siehe hierzu Abschnitt 2.2.7 auf Seite 35). Für den Zugriff auf einen D3D–Vertexpuffer muss dieser für die Zeit des Kopierens der Vertexdaten mit der Funktion Lock gesperrt werden. Diese Sperre wird mit der Funktion Unlock wieder aufgehoben, so dass der Puffer für die Rendervorgänge zur Verfügung steht. Weiterhin wird der Vertexshader mit der D3DX–Funktion D3DXAssembleShaderFromFile aus einer Datei in einen Puffer assembliert und dieser der Funktion CreateVertexShader zur Erzeugung des eigentlichen Shaders übergeben. Sollten Syntaxfehler in der Shaderprogrammierung vorhanden sein, enthält der Puffer, der D3DXAssembleShaderFromFile übergeben wurde, einen String, der den Fehler beschreibt. Falls keine Unterstützung für Vertexshader vorhanden ist, gibt die Funktion SetVertexShader in der Memberfunktion RenderTriangulation einen Fehler zurück. Mit der Funktion CreateVertexDeclaration ist die Erzeugung der Vertexdeklaration möglich, die den Aufbau der Vertexdaten beschreibt und in der globalen Variablen g_pTriangulationDecl deklariert ist.

CTriangulation::DeleteRessources

Diese Funktion wird von der DXUT–Callbackfunktion DeviceLostNotification aufgerufen. Hier werden alle Ressourcen gelöscht, die für das Rendern benötigt wurden. Diese Funktion ist Teil der Implementation der Reaktion auf ein Lostdevice (Seite 30).

CTriangulation::ReloadRessources

Diese Funktion wird von der DXUT–Callbackfunktion DeviceResetNotification aufgerufen. Nachdem alle Ressourcen freigegeben wurden, müssen hier alle Ressourcen, die weiterhin für das Rendern notwendig sind, wieder neu geladen werden, um im Videospeicher präsent zu sein. Für das Neuladen wird einfach die schon vorhandene Funktion SetBuiltTriangulationData verwendet, die auch für die erste Initialisierung zuständig ist. Diese Funktion ist Teil der Implementation der Reaktion auf ein Lostdevice (Seite 30).

CTriangulation::TriangulationFrameMoveNotifikation

Diese Funktion aktualisiert die private Weltmatrix des Objektes. In der Weltmatrix wird eine Drehung um die X–Achse mit anschließender Drehung um die Y–Achse berechnet. Es wird allerdings mit doppelter Geschwindigkeit um die Y–Achse gedreht, damit alle Flächen des Objektes sichtbar werden.

CTriangulation::RenderTriangulation

Diese instanzgebundene Memberfunktion initialisiert für die Darstellung des eigenen Objektes den Renderer. Alle Setupfunktionen müssen zwischen den Aufrufen BeginScene/EndScene aufgerufen werden. Falls Vertexshader unterstützt werden, werden die Konstanten für den Shader von der Funktion SetVertexShaderConstantF für die angegebenen Register initialisiert. Der assemblierte Shadercode wird mit der Funktion SetVertexShader gesetzt. Weiterhin werden der FVF–Code und die Vertexdeklaration gesetzt. Mit der Funktion SetStreamSource wird der aktuelle D3D–Vertexpuffer als Quelldatenstrom gesetzt und anschließend mit DrawPrimitive die Daten gerendert. Für das Rendern von Alphashapes ist das Abschalten des Backfacecullings notwendig, da im CGAL–Modul nicht erkannt werden kann, wo für das Dreieck „außen“ in Bezug auf das gesamte Objekt ist. Informationen zum Cullmode sind im Abschnitt 2.2.3 auf Seite 29 zu finden. Die Erklärung der Extraktion der Alphashapedaten ist auf Seite 57 beschrieben.

3.5.5 Der Vertexshader

Der Vertexshader ist in der Datei TriangulationVertexShader.vsh programmiert worden. Falls diese Datei nicht gefunden wurde, erklärt eine Meldung, in welchem Verzeichnis versucht wurde, sie zu finden. Der Vertexshader setzt die Fixed–Function–Pipeline außer Kraft, so dass alle Transformationen vom Shaderprogramm übernommen werden müssen. Hierzu werden die Transformationsmatrizen dem Shader als Konstanten übergeben. Als Eingangsdaten werden die untransformierten Koordinaten der Vertices und deren Farbe zur Verfügung gestellt. Der Shader führt die Transformationen für jeden Vertex durch und setzt dessen Farbe. Das Besondere an diesem Shader ist, dass er die transformierten Weltkoordinaten für eine neue Farbberechnung nutzt. Wäre dies nicht mit einem Shader möglich, so müsste diese Transformation auf der CPU ausgeführt werden, um die neue Farbe berechnen zu können. Die Fixed–Function–Pipeline ist aufgrund der Nichtprogrammierbarkeit überhaupt nicht in der Lage, diese Berechnung durchzuführen. Anzumerken sei hier noch, dass die Shader–ALUs eine wesentlich höhere Rechenleistung (FLOPS) besitzen als Hauptprozessoren.

3.6 Anhang

3.6.1 Projekteinstellungen in Visual Studio 2005

Dieses Softwareprojekt ist gegen das DirectX–SDK vom April 2007 und gegen die CGAL–Library in der Version 3.2.1. gelinkt. Für Visual Studio 2005 wurde das Service Pack 1 installiert. Die Projekteinstellungen wurden so vorgenommen, dass bei der Compilierung dieser Arbeit gegen die statisch gebauten Libraries von DirectX und CGAL gelinkt wird. Nachfolgend wird auf die zu beachtenden Einstellungen für die Erstellung dieser externen Libraries eingegangen.

CGAL Library compilieren

Zuerst müssen CGAL in der Version 3.2.1. und Boost in der Version 1.33.1. installiert werden. Im Ordner “CGAL–Installationsverzeichnis“/src/CGAL ist die vcproj–Datei zu öffnen und für VS2005 zu konvertieren. Folgende Projekteinstellungen sind vorzunehmen:

- Debugeinstellungen:

  Configuration Properties/(C/C++)/Code Generation/Runtime Library: /MTd
  Configuration Properties/(C/C++)/Preprocessor/Preprocessor Definitions:
  folgende Zeilen eintragen:
  _SECURE_SCL=0
  _HAS_ITERATOR_DEBUGGING=0
  _CRT_SECURE_NO_DEPRECATE
  Configuration Properties/Librarian/Output File: ../../lib/msvc7/CGALd.lib

- Releaseeinstellungen:

  Configuration Properties/(C/C++)/Code Generation/Runtime Library: /MT
  Configuration Properties/(C/C++)/Preprocessor/Preprocessor Definitions:
  folgende Zeilen eintragen:
  _SECURE_SCL=0
  _HAS_ITERATOR_DEBUGGING=0
  _CRT_SECURE_NO_DEPRECATE

In VS2005 im Menü Tools/Options/Projects and Solutions/VC++ Directories ist der Includefiles–Pfad “CGAL–Installationsverzeichnis“/auxiliary/gmp/include
und der Libraryfiles–Pfad “CGAL–Installationsverzeichnis“/auxiliary/gmp/lib
einzutragen.

DirectX Library compilieren

Nach der Installation des SDKs liegt das vorkonfigurierte Projekt für das DirectX–Utility im Order „SDK–Installationsverzeichnis“/Samples/C++/DXUT/Core. Hier ist die sln–Datei für VS2005 zu öffnen. Die folgenden Einstellungen sind vorzunehmen:

- Debugeinstellungen:

  Configuration Properties/(C/C++)/Code Generation/Runtime Library: /MTd
  Configuration Properties/Librarian/Output File: DXUTd.lib

- Releaseeinstellungen:

  Configuration Properties/(C/C++)/Code Generation/Runtime Library: /MT

Anschließend können die statischen Libraries gebaut werden. VS2005 kann nun geschlossen werden und die gebauten Libraries in den Ordner „SDK–Installationsverzeichnis“/Lib/x86 kopiert werden.

Compiler- und Linkereinstellungen des Projektes in der VS2005 IDE

Nachdem die CGAL- und DirectX–Libraries gebaut sind, können die letzten Einstellungen im Projekt selbst vorgenommen werden. Die einzigsten Einstellungen sind im CGAL–Modul–Projekt vorzunehmen:

- Debugeinstellungen:

  Configuration Properties/(C/C++)/Language/OpenMP Support entsprechend einstellen
  Configuration Properties/Manifest Tool/Input and Output/Additional Manifest Files:
  „VS2005 Installationsverzeichnis“/VC/redist/Debug_NonRedist/x86/
  Micorosft.VC80.DebugOpenMP/Microsoft.VC80.DebugOpenMP.manifest

- Releaseeinstellungen:

  Configuration Properties/(C/C++)/Language/OpenMP Support entsprechend einstellen Configuration Properties/Manifest Tool/Input and Output/Additional Manifest Files:
  „VS2005 Installationsverzeichnis“/VC/redist/x86/
  Microsoft.VC80.OPENMP/Microsoft.VC80.OpenMP.manifest

Microsoft ist bei der Distribution des Servicepacks 1 für VS2005 der Fehler unterlaufen, die Versionsnummern der OpenMP DLLs in den Manifestdateien nicht anzupassen. Dies muss manuell vorgenommen werden, und die Manifestdateien müssen inklusive der zugehörigen DLLs im selben Order liegen wie das compilierte Projekt; sonst bricht der Windowsprogrammlader mit dem Fehler der nicht ladbaren DLL den Start des Programms ab.

4 Schluss

Die Zielsetzung dieser Arbeit bestand darin, einen Weg zu finden, den Körper, den eine Punktmenge beschreibt, darzustellen. Diese Zielsetzung wurde mit dem Triangulationsalgorithmus von Delaunay erreicht. Es wurde beschrieben, wie eine delaunaykonforme Triangulation berechnet wird. Weiterhin wurde das Ergebnis dieser Triangulation bewertet und dargestellt, wie mit der Berechnung von Alphashapes dieses Ergebnis unter Umständen verbessert werden kann. Neben der Triangulation wurde auch die Beziehung zu Voronoidiagrammen kurz vorgestellt. Insgesamt konnte für diese Aufgaben die freie Bibliothek CGAL verwendet werden, die auch weitaus mehr Funktionalität zur Verfügung stellt, als für diese Arbeit benötigt wurde.

Für die Darstellung wurde die DirectX–API verwendet. Hierfür wurden alle notwendigen Hintergrundinformationen für die Einbindung dieser API vorgestellt. Um den immer wiederkehrenden programmtechnischen Aufwand für zum Beispiel das Messagehandling zu ersparen, wurde auf das DirectX–Utility, das von Microsoft kostenlos zur Verfügung gestellt wird, zurückgegriffen. Es wurde auch der Aspekt der heute programmierbaren Grafikhardware beschrieben und eine Lösung zur Nutzung dieser Funktionalität vorgestellt.

Um den Kompatibilitätskonflikt zwischen CGAL und DirectX zu lösen, wurde ein modulares Design der Software entwickelt. Dieses verstärkt die Übersichtlichkeit der Software. Es wurde auch der Grundstein für einen Austausch des DirectX–Moduls gegen eine andere Darstellungsimplementation gelegt und ein Lösung vorgestellt, wie zur Laufzeit ein Modul bequem entladen und geladen werden kann. Weiterhin wurde darauf Wert gelegt, die Rechenkraft zukünftiger Multiprozessorsysteme vollständig nutzen zu können. Daher wurde die Funktionalität, mehrere Triangulationen gleichzeitig berechnen zu können, implementiert. Um nicht nur die Rechenlast auf mehrere Hauptprozessorkerne zu verteilen, wurde eine Lösung implementiert, um auch solche Lasten auf die Grafikhardware zu verlagern. Für die Verteilung der Rechenarbeiten auf mehrere vorhandene Hauptprozessoren wurde OpenMP verwendet und vorgestellt. Weitere Eckpfeiler dieser Arbeit sind die Aufbereitung der Daten von CGAL für DirextX inklusive eines geschickten Speichermanagements und Export der Triangulationen im Wavefrontformat. Dies ermöglicht die Nachbearbeitung der Objekte mit bekannten 3D–Bearbeitungsprogrammen wie zum Beispiel dem frei nutzbaren Tool Blender. Für die Datenreduktion wurde der Meshoptimizer entwickelt, dessen Funktionalität und Kriterien dargestellt wurden.

Index

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 13. August 2007	
Ort, Datum	Unterschrift