Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Torben Könke

Untersuchung algorithmischer Ansätze zur Vereinfachung von Polygonnetzen

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Torben Könke

Untersuchung algorithmischer Ansätze zur Vereinfachung von Polygonnetzen

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

Betreuender Prüfer : Prof. Dr. Philipp Jenke
Zweitgutachter : Prof. Dr. Stefan Sarstedt

Torben Könke

Thema der Arbeit

Untersuchung algorithmischer Ansätze zur Vereinfachung von Polygonnetzen

Stichworte

Polygonnetz Vereinfachung, Mesh Vereinfachung, Multiresolutions-Rendering, Detaillierungsgrad, Level Of Detail

Kurzzusammenfassung

Diese Arbeit stellt verschiedene Ansätze zur Vereinfachung von Polygonnetzen vor. Die Verfahren werden vor dem Hintergrund eines gewählten Anwendungsfalls verglichen und das am besten geeignete Verfahren anschließend prototypisch implementiert.

Torben Könke

Title of the paper

A Survey of Algorithmic Approaches for Simplifying Polygon Meshes

Keywords

mesh simplification, multiresolution modeling, level of detail, geometric modeling, hierarchical model approximation, level of detail

Abstract

This thesis researches distinct approaches that have been proposed for the purpose of mesh simplification. The researched algorithms are evaluated in the context of a defined use case and the seemingly best-suited approach is subsequently implemented as a prototype.

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

1.1. Motivation

In vielen Bereichen der Computergrafik spielt eine möglichst Detail- und realitätsgetreue Darstellung komplexer dreidimensionaler geometrischer Figuren eine wichtige Rolle. Solche Objekte können in speziellen Modellierungsprogrammen per Hand angefertigt werden, oder durch automatisierte Analyse von Bilddaten, wie beispielsweise die Aufnahmen eines 3D-Scanners, erstellt werden.
Die Komplexität bzw. der Detailgrad solcher Objekte wird dabei generell in der Anzahl ihrer Polygone, d. h. ihrer kleinsten Bestandteile, gemessen, und es ist leicht nachvollziehbar, daß sich ein linearer Zusammenhang zwischen Polygonanzahl eines Objekts und seinem benötigten Verarbeitungsaufwand durch die Grafikhardware herstellen lässt. Trotz der steten Zunahme der Leistungsfähigkeit moderner Grafikkarten existiert so naturgemäß eine Obergrenze an darstellbaren Polygonen, die nicht überschritten werden kann, ohne die Leistungsfähigkeit des Systems zu mindern. Dies ist insbesondere bei interaktiven Anwendungen wie Simulationen oder Computerspielen problematisch, da hier oft eine konstante Bildrate für das gesamte Nutzungserlebnis ausschlaggebend ist.

Auf den Punkt gebracht lässt sich also sagen, daß es in vielen Anwendungsbereichen ein Dilemma zwischen hochauflösender Darstellung auf der einen und vertretbarem Rechenaufwand auf der anderen Seite gibt. In der Praxis hat sich gezeigt, daß es oft möglich ist, die Polygonanzahl hochauflösender Objekte drastisch zu reduzieren, ohne daß die visuelle Wahrnehmung für den Beobachter dabei merklich beeinträchtigt wird. Aus dieser Feststellung heraus hat sich ein Forschungsgebiet entwickelt, das sich mit der Frage der systematischen Vereinfachung von Polygonnetzen beschäftigt.

1.2. Zielsetzung

Ziel dieser Arbeit ist es, unterschiedliche Verfahren zur Reduktion der Polygonanzahl dreidimensionaler Objekte zu recherchieren und wissenschaftlich zu analysieren. Die

Funktionsweisen der verschiedenen Ansätze sollen nachvollzogen und in Bezug auf ihre praktische Einsetzbarkeit hin miteinander verglichen werden.

Der Bezug zur Praxis soll durch die Entwicklung einer Software zur multi-resolutionalen Darstellung von dreidimensionalen Objekten hergestellt werden. Die Software soll dabei als Grundstein für aufbauende Arbeiten zur Konzeption einer Visualisierung eines bestehenden Logistiksystems dienen.

1.3. Aufbau der Arbeit

Die Arbeit gliedert sich in mehrere Kapitel, die aufeinander aufbauen.

Das Kapitel „Grundlagen“ führt elementare Begriffe und Konzepte der Computergrafik ein, die für ein Verständnis der übrigen Kapitel notwendig sind. Im darauf folgenden Kapitel „Analyse“ werden ausgewählte Verfahren zur Vereinfachung von Polygonnetzen im Detail betrachtet und untersucht. Am Ende dieses Kapitels stehen eine Auswertung der untersuchten Verfahren und eine begründete Wahl des Verfahrens, das in der zu entwickelnden Softwarelösung zum Einsatz kommen soll. Im Kapitel „Implementierung“ werden anschließend einige wichtige Details der Implementierung der Software aufgezeigt und erläutert. Die Arbeit schließt ab mit einer Zusammenfassung der erlangten Erkenntnisse, gefolgt von einem Ausblick auf mögliche weiterführende Forschungsansätze.

2. Grundlagen

2.1. Definitionen

Dieser Abschnitt führt einige grundlegende Begriffe aus dem Bereich der Computergrafik ein, die für ein Verständnis der weiteren Kapitel notwendig sind.

2.1.1. Vertex

Ein Vertex (Plural: *Vertices*) beschreibt einen Punkt im Raum, der Schnittpunkt zweier oder mehrerer Geraden bzw. Kanten ist. In dieser Arbeit wird mit dem Begriff Raum generell der euklidische Vektorraum über $\mathbb{R}^3$ bezeichnet, so daß die Position eines Vertex durch einen Spaltenvektor $\vec{x}$ der Form $\begin{pmatrix} x & y & z \end{pmatrix}^T$ angegeben werden kann.

Neben der eigentlichen Position können mit einem Vertex beliebige weitere Attribute wie Farbwerte, Texturkoordinaten oder Oberflächennormalen assoziiert werden.

2.1.2. Kante

Eine Kante (engl.: *Edge*) ist die kürzeste Verbindung zweier Vertices einer geometrischen Figur, wobei zwischen gerichteten und ungerichteten Kanten unterschieden wird. Eine ungerichtete Kante wird durch die Menger zweier Vertices definiert, während eine gerichtete Kante prinzipiell zwischen einem Start- und einem Zielvertex unterscheidet.

2.1.3. Polygon

Ein Polygon ist eine geschlossene geometrische Figur, die durch eine Menge von mindestens drei Vertices, welche durch Kanten miteinander verbunden sind, definiert wird. Polygone, deren Vertices alle in einer gemeinsamen Ebene liegen, werden als planare

Polygone bezeichnet. Da eine Ebene im Raum durch genau 3 Punkte definiert wird, ergibt sich daraus, daß dreiseitige Polygone stets planar sind.

2.1.4. Facette

Bei der Beschreibung von Polygonnetzen werden die Begriffe Facette (engl.: *facette*) und Polygon oft synonym benutzt. Diesem Ansatz folgend wird der Begriff Facette in dieser Arbeit mit dem Begriff Polygon gleichgesetzt.

2.1.5. Normalenvektor

Als Normalenvektor bezeichnet man einen Vektor der orthogonal, d. h. senkrecht auf einer Fläche bzw. einer Ebene steht und über eine Norm von eins verfügt.

Normalenvektoren spielen in vielerlei Anwendungsbereichen der Computergrafik wie beispielsweise der Ausleuchtung von Objekten oder der Berechnung von Schattenwürfen eine wichtige Rolle.

2.2. Polygonnetze

Ein Polygonnetz (engl.: *polygon mesh*) ist ein dreidimensionales Objekt, das durch eine Menge von Polygonen definiert wird und damit die Gestalt eines Polyeders beschreibt. Polygonnetze werden in den unterschiedlichsten Bereichen der Computergrafik eingesetzt um beliebige, oftmals komplexe Gegenstände und Strukturen darzustellen. So lassen sich beispielsweise Gebäude, Fahrzeuge oder Bauteile technischer Industrieanlagen durch Polygonnetze modellieren. Gleichzeitig werden Polygonnetze in medizinischen und wissenschaftlichen Bereichen eingesetzt, um Iso-Oberflächen zu extrahieren oder komplexe Molekülverbindungen zu visualisieren.

Die Einsatzgebiete sind also äußerst vielfältig und umfassen beinahe alle Anwendungsbereiche der modernen 3D-Computergrafik.

2.2.1. Aufbau

Ein Polygonnetz wird durch eine Menge von Polygonen definiert, die über gemeinsame Kanten miteinander verbunden sind, so daß geschlossene Oberflächen entstehen und somit beliebige dreidimensionale Objekte abgebildet werden können. Da Polygone bzw. Facetten wiederum als Mengen von Vertices und Kanten zwischen diesen aufgefasst werden können, müssen in einem Polygonnetz folglich diese 3 Komponenten verwaltet werden:

- Vertices
- Kanten
- Facetten

Zur effizienten Verwaltung und Speicherung dieser Komponenten haben sich eine Reihe unterschiedlicher Datenstrukturen etabliert, die, je nach Anwendungsfall, verschiedene Vor- und Nachteile aufweisen. Einige davon sollen nachfolgend kurz vorgestellt werden.

Eine naheliegende Möglichkeit der Speicherung besteht darin, die Vertices eines Polygonnetzes in einem Feld abzulegen und die Facetten anschließend als Listen von Indexzeigern in eben dieses Feld zu speichern. Diese als Knotenliste bezeichnete Art der Verwaltung ist äußerst speichereffizient und bietet den impliziten Vorteil, daß diese Organisation der Daten gemeinhin von der Grafikhardware erwartet wird. Zu Zwecken der Darstellung können die entsprechenden Speicherblöcke daher direkt ohne weitere Umwandlung in den Speicherbereich der Grafikkarte übertragen werden.

Eine gänzlich andere Form zur Speicherung bzw. zur Verwaltung eines Polygonnetzes sind die sog. Winged-Edge und Half-Edge Datenstrukturen. Diese von Bruce Baumgart [Bau72] bzw. Charles M. Eastman [EK81] entworfenen Datenstrukturen zeichnen sich durch eine explizite Verwaltung der Kanten des Polygonnetzes aus. Für jede Kante werden hierbei Informationen wie die zugehörigen Vertices, Zeiger auf die nachfolgenden bzw. gegenüberliegenden Kanten sowie Zeiger auf die anliegenden Facetten gespeichert. Der Vorteil dieser Datenorganisation ist, daß hierbei Adjazenzbeziehungen zwischen den Komponenten des Polygonnetzes implizit in der Datentruktur festgehalten werden. Dies ermöglicht beispielsweise das Auffinden von benachbarten Eckpunkten und anliegenden Polygonen mit einer Laufzeit von O(1), was bei der Implementierung vieler algorithmischer Verfahren, die auf Polygonnetzen operieren, einen wichtigen Faktor darstellt.

Neben den hier erwähnten Datenstrukturen, gibt es noch viele weitere mögliche Repräsentationen zur Datenorganisation von Polygonnetzen. Für eine ausführlichere Darstellung sei an dieser Stelle auf Smith [Smi06] verwiesen.

2.2.2. Topologie

Der Begriff Topologie oder auch Konnektivität bezeichnet den „inneren Aufbau“ eines Polygonnetzes. Gemeint sind hiermit Informationen darüber, wie die einzelnen Komponenten eines Polygonnetzes miteinander zusammenhängen, beispielsweise welche Facetten benachbart sind, sich also eine gemeinsame Kante teilen oder welche Vertices die Nachbarn eines bestimmten Vertex sind. Die Topologie sollte nicht mit der Geometrie, also gewissermaßen dem „äußeren Aufbau“ eines Polygonnetzes verwechselt werden. Es ist durchaus möglich, daß zwei Polygonnetze über identische Geometrie, aber unterschiedliche Topologie verfügen und umgekehrt. Die folgenden zwei Abbildungen verdeutlichen den Unterschied.

2.2.3. Dreiecksnetze

Ein Dreiecksnetz (engl.: *triangle mesh*) ist ein Polygonnetz mit der Besonderheit, daß es lediglich aus dreieckigen Polygonen besteht. Diese Einschränkung auf Dreiecke, also die einfachste Art von Polygon, als „Baustein“ für ein Polygonnetz ermöglicht eine Reihe von Vereinfachungen und Optimierungen, weshalb diese Art von Polygonnetzen in der modernen Computergrafik am häufigsten anzutreffen ist. So haben dreieckige Polygone den Vorteil, daß sie stets in einer Ebene liegen, da eine Ebene im Raum durch 3 Punkte definiert wird. Diese Eigenschaft der Planarität wiederum ermöglicht es, skalare Attribute entlang der Oberfläche des Polygons linear zu interpolieren, so daß sich beispielsweise Berechnungen zur Ausleuchtung oder zur Texturierung der Fläche des Polygons äußerst effizient durchführen lassen.

Ein weiterer Vorteil ist in der effizienten Speicherauslastung dreieckiger Polygone begründet. Insbesondere benötigt man zur Darstellung eines Dreiecksnetzes ab dem ersten Dreieck für jedes weitere Dreieck lediglich die Angabe eines einzelnen Vertex, unter der Voraussetzung, daß das Dreiecksnetz in der Form eines sog. *Triangle Strips* vorliegt. Aufgrund dieser und weiterer Vorteile operiert moderne Grafikhardware beinahe ausschließlich auf dreiseitigen Polygonen. Dies stellt jedoch keine ernsthafte Einschränkung dar, da sich jedes Polygonnetz in ein äquivalentes Dreiecksnetz überführen lässt (vgl. [DO11, S. 4]), ein Vorgang der als Triangulierung bezeichnet wird. Grafikkarten verfügen zu eben diesem Zweck oftmals über hochspezialisierte Hardware.

3. Analyse

3.1. Vertex Decimation

3.1.1. Einführung

Der Ansatz des Vertex Decimation wurde erstmalig von Schroeder et al. in ihrer Veröffentlichung „Decimation of Triangle Meshes“ 1992 beschrieben [SZL92]. Ausgehend von der Feststellung, daß systematisch erzeugte Polygonnetze oftmals aus sehr vielen Polygonen bestehen, sollte ein Algorithmus entwickelt werden, der die Komplexität dieser Polygonnetze reduziert, ohne dabei die ursprüngliche Topologie zu verfälschen. Insbesondere wird hier das sog. Marching Cubes Verfahren erwähnt (vgl. [LC87]), welches u.a. eingesetzt werden kann, um Polygonnetze aus computertomografischen Bildaufnahmen zu extrahieren. Polygonnetze, die auf diese Weise erzeugt werden, sind häufig sehr hochauflösend und daher für eine direkte Weiterverarbeitung dementsprechend ungeeignet.

Das Prinzip des Algorithmus basiert auf dem iterativen Löschen einzelner Vertices, bis der gewünschte Reduktionsgrad erreicht ist. Der grundlegende Ablauf lässt sich wie folgt beschreiben: Es wird mehrfach über die Vertices des Polygonnetzes iteriert. In jedem Iterationsdurchgang wird dabei jeder Vertex einer eindeutigen Kategorie zugeteilt und anschließend nach einem der Kategorie entsprechendem Dezimierungskriterium als potentieller Löschkandidat evaluiert. Am Ende dieses Vorgangs steht der Vertex, der aus dem Polygonnetz gelöscht werden soll. Neben besagtem Vertex werden ebenfalls alle Dreiecke, die diesen Vertex als Eckpunkt nutzen, gelöscht. Das so entstandene Loch im Polygonnetz wird anschließend durch eine lokale Triangulierung geschlossen. Hiernach startet die nächste Iteration. Dieser Vorgang wird so lange wiederholt, bis das gewünschte Abbruchskriterium erreicht wird, beispielsweise eine Prozentzahl, die den gewünschten Reduktionsgrad angibt, oder die angestrebte Anzahl an Polygonen, die das vereinfachte Polygonnetz nicht überschreiten soll.

Schroeder et al. haben sich bei ihrem Ansatz auf Dreiecksnetze beschränkt. Der Algorithmus lässt sich jedoch auf beliebige Polygonnetze verallgemeinern, da sich jedes

Polygonnetz in ein Dreiecksnetz überführen lässt (vgl. hierzu [Del34], [FM84] oder [EET93]).

3.1.2. Betrachung des Verfahrens

Im ersten Schritt der Evaluierung eines Vertex wird dieser einer von fünf Kategorien zugeordnet. Je nach erfolgter Einteilung werden im weiteren Verlauf unterschiedliche Dezimierungskriterien auf den Vertex angewandt. Abbildung 3.1 veranschaulicht die charakteristischen Eigenschaften der verschiedenen Kategorien, die im Folgenden kurz erläutert werden.

Quelle: [SZL92]

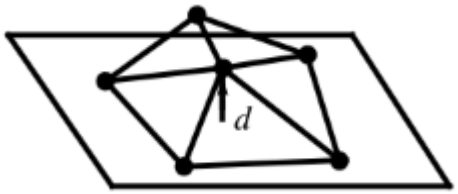
Abbildung 3.1.: Kategorisierung von Vertices nach Schroeder et al.

Ein Vertex wird als *Simple* bezeichnet, wenn um ihn herum ein geschlossener Ring aus Dreiecken existiert. Insbesondere muss hierzu jede inzidente Kante des Vertex von exakt zwei benachbarten Dreiecken geteilt werden. Existieren hingegen Dreiecke, die den Vertex zwar als Eckpunkt haben, jedoch nicht Teil des geschlossenen Rings sind, so wird der Vertex als *Complex* bezeichnet. Dies sind gemeinhin Fälle, in denen die Eigenschaft der 2-Mannigfaltigkeit verletzt wird. Sind die inzidenten Dreiecke um den Vertex herum in einem geschlossenen Halbkreis angeordnet, wird der Vertex als sog. *Boundary Vertex* klassifiziert. Das Umfeld des Vertex stellt in diesem Fall gewissermaßen eine Grenzfläche der Oberfläche des Polygonnetzes dar. Die übrigen zwei Kategorien sind Sonderfälle der ersten Kategorie; Verfügt ein *Simple Vertex* über anliegende benachbarte Dreiecke, deren Diederwinkel, also der Winkel zwischen den Normalen der Ebenen, in denen die jeweiligen Dreiecke liegen, einen zuvor festgelegten Schwellenwert überschreitet, so wird die gemeinsame Kante dieser Dreiecke als *Feature Edge* bezeichnet. Liegen nun wiederum zwei *Feature Edges* an einem *Simple Vertex* an, so wird dieser als *Interior Edge Vertex* bezeichnet und der entsprechenden Kategorie zugeordnet. Verfügt ein Vertex über mehr als zwei *Feature Edges*, wird er schließlich als *Corner Vertex* eingeordnet.

Nach abgeschlossener Kategorisierung des Vertex kann nun im nächsten Schritt das eigentliche Dezimierungskriterium angelegt werden, um die Eignung des Vertex als

potentieller Löschkandidat zu beurteilen, wobei Vertices der Kategorie *Complex* grundsätzlich nicht als Löschkandidaten in Betracht gezogen werden. Die Aufgabe eines Dezimierungskriteriums ist es, eine Metrik zu definieren, nach der potentielle Löschkandidaten geordnet werden können. Mathematisch betrachtet kann man ein Dezimierungskriterium folglich als eine Funktion auffassen, die jedem Vertex einen eindeutigen Kostenwert zuordnet. Hierdurch wird es möglich, genau den Vertex eines Polygonnetzes zu finden, dessen Entfernen die Kosten für die Vereinfachung des Polygonnetzes minimiert, also gewissermaßen den entstandenen „Schaden“ so gering wie möglich hält. Je nach Kategorie des untersuchten Vertex ziehen Schroeder et al. leicht abgewandelte Dezimierungskriterien heran, wobei es prinzipiell möglich ist, an dieser Stelle des Algorithmus beliebige Kriterien zu definieren.

Vertices der Kategorie *Simple* werden nach dem sog. *Distance-to-Plane* Kriterium ausgewertet. Die Idee dabei ist, aus den gewichteten Flächen der anliegenden Dreiecke eine gemittelte Ebene zu bestimmen und anschließend den Abstand des Vertex von dieser Ebene zu messen, wie in Abbildung 3.2 gezeigt.



Quelle: [SZL92]

Abbildung 3.2.: Distance-to-Plane Kriterium

Je geringer der Abstand, desto mehr Erfolg verspricht eine potentielle Löschung des Vertex und der anliegenden Dreiecke, da sich eine solche Region aufgrund ihrer relativen Planarität gut retriangulieren lässt. Seien mit A_i, $\vec{n}_i$ und $\vec{p}_i$ die Fläche, der Normalvektor und der geometrische Schwerpunkt des i-ten Dreiecks eines Vertex *v* bezeichnet, der über *k* inizidente Dreiecke verfügt, so gilt für den flächen-gewichteten Normalenvektor $\vec{n}_{avg}$ (vgl. [Muk12, S. 195]):

$$\vec{n}_{avg} = \frac{\sum_{i=1}^{k} A_i \vec{n}_i}{\sum_{i=1}^{k} A_i}$$

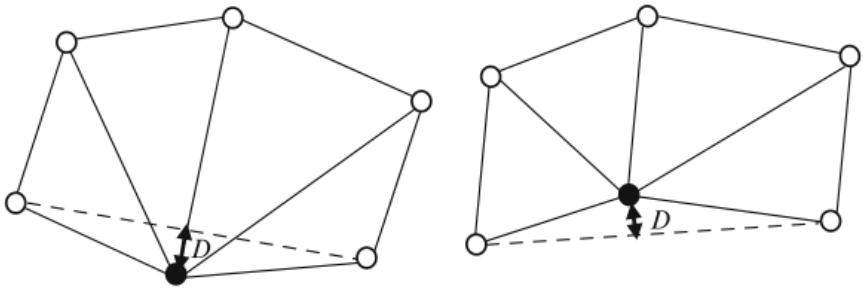
Für den gemittelten geometrischen Schwerepunkt $\vec{p}_{avg}$ gilt entsprechend:

$$\vec{p}_{avg} = \frac{\sum_{i=1}^{k} A_i \vec{p}_i}{\sum_{i=1}^{k} A_i}$$

Dieser so erhaltene Punkt $\vec{p}_{avg}$ definiert nun zusammen mit dem Normalenvektor $\vec{n}_{avg}$ die gemittelte Ebene, die sich über die Normalenform angeben lässt. Der Wert der Dezimierungsfunktion an der Stelle $\vec{v}$ lässt sich dann als Distanz des Vertex v zur Ebene angeben:

$$D = \frac{(x_v - x_p)x_n + (y_v - y_p)y_n + (z_v - z_p)z_n}{\sqrt{x_n^2 + y_n^2 + z_n^2}} = \frac{(\vec{v} \cdot \vec{n}) + d}{|\vec{n}|}$$

Für Vertices der *Boundary*, *Interior Edge* und *Corner* Kategorien kommt eine leicht abgewandelte Form der Dezimierungsfunktion zum Einsatz. Hier wird nicht die Distanz des Vertex zur Ebene gemessen, sondern die Distanz des Vertex zu der Geraden, die durch die zwei Vertices definiert wird, welche die *Feature Edge* bilden (Abbildung 3.3).



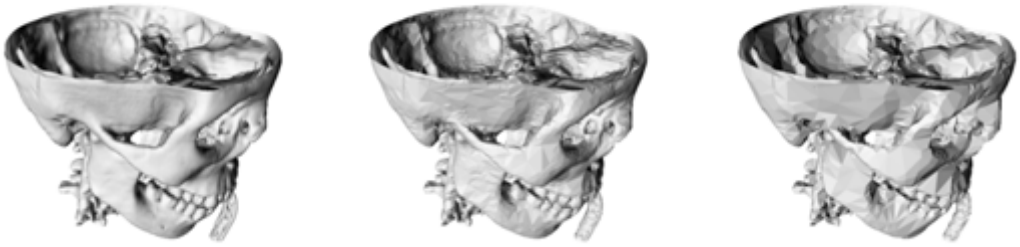
Quelle: [Muk12, S. 196]

Abbildung 3.3.: Fehlermetrik für Boundary Vertices

Nachdem durch Minimierung der Dezimierungsfunktion ein Vertex zur Entfernung ausgewählt wurde, muss vor der eigentlichen Löschung geprüft werden, ob das so entstehende Loch in der Geometrie durch eine Triangulierung geschlossen werden kann. Wenn festgestellt wird, daß das Löschen des betrachteten Vertex eine Änderung der Topologie des Polygonnetzes zur Folge hätte, wird der betroffene Vertex beibehalten und die Suche nach einem geeigneten Kandidaten fortgesetzt. Andernfalls wird der Vertex entfernt und eine lokale Triangulierung durchgeführt. Schroeder et al. nutzen hierbei einen Ansatz, der das Polygon iterativ in zwei Hälften teilt. Dieser Vorgang wiederholt

sich, bis eine Hälfte nur noch aus drei Vertices besteht, aus denen anschließend ein neues Dreieck erstellt werden kann. Nach der Triangulierung startet der Algorithmus die nächste Iteration, sofern das Abbruchkriterium noch nicht erreicht wurde.

Abbildung 3.4 zeigt eine Sequenz von durch den Algorithmus erzeugten Vereinfachungen. Die Aufnahme auf der linken Seite zeigt dabei eine durch ein CT extrahierte Iso-Oberfläche eines menschlichen Schädels. Dieses ursprüngliche Polygonnetz verfügt über 569.000 Polygone, während die abgebildeten Vereinfachungen in der Mitte und auf der rechten Seite über 142.000 bzw. 57.000 Polygone verfügen, also um etwa 75% bzw. 90% in der Anzahl ihrer Polygone reduziert wurden. Wie in der Abbildung zu sehen, sind die Konturen und die Hauptmerkmale der Oberfläche auch in den vereinfachten Ausfertigungen noch gut zu erkennen.



Quelle: [SZL92]

Abbildung 3.4.: Durch Vertex Decimation erzeugte Vereinfachungen

3.2. Pair Contracting

3.2.1. Einführung

Während bei den bisher betrachteten Verfahren der Erhalt der ursprünglichen Topologie des Polygonnetzes eine konkrete Zielsetzung war, beschreiben Garland und Heckbert in ihrer Arbeit *Surface Simplification Using Quadric Error Metrics* 1997 [GH97] einen Algorithmus, der diese Anforderung aufhebt. Der Gedanke hierbei ist, daß der Erhalt der Topologie für viele Anwendungsfälle nicht relevant ist und durch Weglassen dieses Kriteriums eine bessere geometrische Ähnlichkeit zum Ausgangsobjekt erzielt werden kann.

Der Algorithmus funktioniert nach dem Prinzip der iterativen Kontraktion, d. h. es werden paarweise Vertices betrachtet und in einem Vereinfachungsschritt zu einem Vertex

zusammengefasst. Ähnliche Verfahren wurden u. a. bereits von Hoppe [HDD+93], Ronfard et al. [RR96a] und Guéziec [Gué95] entwickelt, allerdings beschränken sich diese unter dem Begriff *Edge Collapse* bekannten Verfahren auf die Kontraktion von Vertices, die durch Kanten unmittelbar miteinander verbunden sind. Garland und Heckbert erweitern diese Idee nun dahingehend, daß auch Vertices zusammengefasst werden können, die nicht durch eine Kante miteinander verbunden sind, ein Vorgang den sie *Aggregation* nennen.

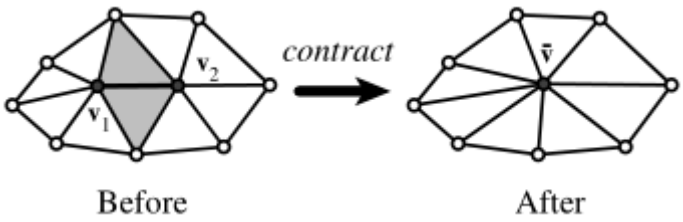
3.2.2. Betrachtung des Verfahrens

Die atomare Operation des Algorithmus ist die sog. *pair contraction*, also das Verschmelzen eines Paares von Vertices. Formal lässt sich dieser Vorgang als

$$(v_1, v_2) \rightarrow \bar{v}$$

ausdrücken und hat dabei folgende Bedeutung: Die beiden Vertices v_1 und v_2 werden an die Position $\bar{v}$ verschoben. Anschließend werden alle inzidenten Kanten von v_2 dem Vertex v_1 hinzugefügt und v_2 wird aus dem Polygonnetz gelöscht. Im Zuge dieses Vorgangs werden nun degenerierte Polygone entfernt.

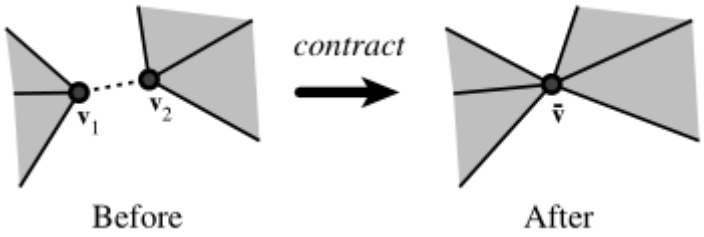
Abbildungen 3.5 und 3.6 demonstrieren das Prinzip.



Quelle: [GH97]

Abbildung 3.5.: Kantenkontraktion

Abbildung 3.5 zeigt eine gewöhnliche Kantenkontraktion, wie sie auch bei Hoppe [HDD+93] und ähnlichen Verfahren zum Einsatz kommt. Bei der gezeigten Kontraktion degenerieren die zwei grau schattierten Dreiecke und werden aus dem Polygonnetz entfernt. Abbildung 3.6 zeigt eine Kontraktion von nicht-verbundenen Vertices, ein Vorgang bei dem also auch unterschiedliche Zusammenhangskomponenten des Polygonnetzes miteinander verbunden werden können und der mit einer Änderung der Topologie einhergeht.



Quelle: [GH97]

Abbildung 3.6.: Kontraktion zweier nicht-verbundener Vertices

In einem ersten Schritt in der Initialisierungsphase des Algorithmus werden alle Vertexpaare ermittelt, die als potentielle Kandidaten für eine Kontraktion in Frage kommen. Hierfür legen Garland und Heckbert zwei Kriterien an. So sind Vertices, die durch eine Kante miteinander verbunden sind, prinzipiell potentielle Kandidaten für eine Kontraktion. Das zweite Kriterium ist ein vom Benutzer konfigurierbarer Schwellwert t, der die maximale räumliche Distanz angibt, die zwei Vertices zueinander haben dürfen, um als Kontraktionspaar berücksichtigt zu werden. Für den Fall, dass $t = 0$ gewählt wird, verhält sich der Algorithmus demnach wie ein gewöhnliches Kantenkontraktionsverfahren. Zu große Werte für t hingegen können zu drastischen und unerwünschten Änderungen der Geometrie führen.

Nach der Ermittlung aller potentiellen Kontraktionspaare kann nun die eigentliche iterative Reduktionsphase des Algorithmus starten. Wie bei anderen Algorithmen muss auch hier eine Auswahl getroffen werden, welcher Kandidat bzw. welches Kandidatenpaar während einer Iteration kontraktiert werden soll. Es muss also eine Kostenfunktion aufgestellt werden, die eine eindeutige Ordnung zwischen den Vertexpaaren definiert. Garland und Heckbert assoziieren zu diesem Zweck mit jedem Vertex v eine symmetrische 4x4 Matrix Q, mit deren Hilfe der geometrische Fehler an der Position von v wie folgt modelliert werden kann:

$$\Delta(v) = v^T Q v \tag{3.1}$$

Um die Matrixmultiplikation in 3.1 zu ermöglichen, wird v hierbei zu einem homogenen Vektor $\begin{pmatrix} x & y & z & w \end{pmatrix}^T$ erweitert, wobei die w-Komponente stets 1 ist. Für die Matrix $\bar{Q}$ eines neuen Vertex $\bar{v}$, der bei einer Aggregation zweier Vertices v_1 und v_2 entsteht, werden die Matrizen der beiden Vertices aufsummiert:

$$\bar{Q} = Q_1 + Q_2 \tag{3.2}$$

Der geometrische Fehler an der Stelle $\bar{v}$ stellt nun die Kosten für die Kontraktion der beiden Vertices v_1 und v_2 des Vertexpaares p zu dem neuen Vertex $\bar{v}$ dar. Es stellt sich nun noch die Frage, an welcher Position im Raum der neue Vertex $\bar{v}$ in das Polygonnetz eingefügt werden soll. Naheliegende Kandidaten hierfür sind die Positionen von v_1 oder v_2 oder der Mittelpunkt der Strecke zwischen v_1 und v_2. Um jedoch die optimale Position zu ermitteln, kann die Funktion aus 3.1 als Kostenfunktion aufgefasst und minimiert werden. Das Minimierungsproblem lässt sich dabei auf folgendes lineares Gleichungssystem zurückführen:

$$\begin{bmatrix} q_{11} & q_{12} & q_{13} & q_{14} \\ q_{21} & q_{22} & q_{23} & q_{24} \\ q_{13} & q_{23} & q_{33} & q_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix} \bar{v} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{3.3}$$

Unter der Voraussetzung, dass die Matrix aus 3.3 invertierbar ist, ergibt sich für $\bar{v}$ dann die Lösung:

$$\bar{v} = \begin{bmatrix} q_{11} & q_{12} & q_{13} & q_{14} \\ q_{21} & q_{22} & q_{23} & q_{24} \\ q_{13} & q_{23} & q_{33} & q_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{3.4}$$

Ist die Matrix aus 3.3 nicht invertierbar, wird auf eine der anfangs erwähnten Positionen zurückgegriffen. In einer Iteration wird nun das Vertexpaar als nächstes ausgewählt, durch dessen Kontraktion die Gesamtkosten in Bezug auf das Polygonnetz minimiert werden.

Im Nachfolgenden soll nun die Bedeutung und die Berechnung der im letzten Absatz eingeführten Matrix Q eingehender betrachtet werden. Wie aus Gleichung 3.1 ersichtlich, muss die Matrix Q eine Art Heuristik für die Abweichung einer Position v von der ursprünglichen Position des betreffenden Vertex des Polygonnetzes darstellen. Für die Berechnung dieser Heuristik machen sich Garland und Heckbert dabei die bereits von Ronfard [RR96b] eingesetzte interessante Tatsache zu Nutze, daß ein Vertex v eines Polygonnetzes als Schnittpunkt genau der Ebenen aufgefasst werden kann, die durch seine inzidenten Facetten aufgespannt werden. Es lässt sich nun eine Funktion aufstellen, die für eine Position v die Summe der quadrierten Abstände zu besagten Ebenen bestimmt und somit eine Fehlermetrik aufstellt:

$$\Delta(v) = \Delta([v_x \quad v_y \quad v_z \quad 1]^T) = \sum_{p \in Ebenen(v)} (p^T v)^2 \tag{3.5}$$

Durch Umformung lässt sich diese Gleichung auch schreiben als:

$$\Delta(v) = \sum_{p \in Ebenen(v)} (v^T p)(p^T v)$$

$$= \sum_{p \in Ebenen(v)} v^T (p p^T) v$$

$$= v^T \left(\sum_{p \in Ebenen(v)} K_p \right) v$$

mit

$$K_p = \begin{bmatrix} a^2 & ab & ac & ad \\ ab & b^2 & bc & bd \\ ac & bc & c^2 & cd \\ ad & bd & cd & d^2 \end{bmatrix}$$

Diese symmetrische Matrix K_p ist nun ein Maß für den Abstand eines Punktes von der Ebene $p = ax + by + cz + d = 0$. Es kann nun für jede inzidente Facette eines Vertex v eine solche Matrix K_p nach obiger Methode bestimmt werden. Durch Aufsummieren dieser Matrizen gelangt man schließlich zu der gesuchten Matrix Q. Als logische Schlussfolgerung ergibt sich, daß die so erhaltenen Heuristiken für alle ursprünglichen Vertices eines Polygonnetzes 0 ergeben, da jeder Vertex genau in den Ebenen aller seiner inzidenten Facetten liegt.

Als vereinfachter Pseudocode lässt sich der Algorithmus von Garland und Heckbert zusammenfassend wie folgt formulieren:

Procedure Pair-Contract

 # Initialisierungsphase
Bestimme Q für jeden Vertex
Bestimme alle gültigen Vertexpaare

foreach *Vertexpaar* p **do**
  Bestimme Kosten der optimalen Position des Vertex $\bar{v}$, der das Vertexpaar p bei einer Kontraktion ersetzt.
end

 # Reduktionsphase
while *Reduktionsziel nicht erreicht* **do**
  Wähle aus Vertexpaaren das Paar p mit den niedrigsten Kosten.
  Kontraktiere Vertexpaar p zu $\bar{v}$.
  Berechne die Kosten aller von der Kontraktion beeinflussten Vertexpaare neu.
end

Abbildung 3.7 zeigt eine Sequenz progressiver, durch den Algorithmus erzeugten Vereinfachungen. Das originale Objekt auf der linken Seite der Abbildung verfügt dabei über 5804 Polygone, während die nachfolgenden Varianten in abnehmender Reihenfolge nur noch über jeweils 994, 532, 248 und 64 Polygone verfügen. Bemerkenswert ist, daß markante Merkmale wie beispielsweise die Hörner der abgebildeten Kuh selbst bei vergleichsweise niedriger Polygonzahl noch deutlich zu erkennen sind.

Quelle: [GH97]

Abbildung 3.7.: Durch Pair Contracting erzeugte Vereinfachungen

3.3. Surface Re-Tiling

3.3.1. Einführung

Etwa zeitgleich mit Schroeder et al. [SZL92] veröffentlichte Turk unter dem Titel *Re-Tiling Polygonal Surfaces* [Tur92] einen gänzlich anderen Ansatz zur Vereinfachung von Polygonnetzen. Während bei vielen Methoden eine Reduzierung der Polygonzahl durch iteratives Ausführen eines Vereinfachungsschrittes erreicht wird, verfolgt Turk den Ansatz, die strukturelle Oberfläche eines Polygonnetzes zu analysieren und anschließend eine vereinfachte Variante unter Beibehaltung der ursprünglichen Topologie komplett neu zu erzeugen.

Die Idee des Algorithmus ist es, in einem ersten Schritt eine Anzahl neuer Vertices auf der Oberfläche eines existierenden Polygonnetzes gleichmäßig zu verteilen, ein Vorgang, den Turk als *Re-Tiling* bezeichnet. Das Resultat dieser Phase ist ein Zwischenprodukt, die sog. *Mutual Tessellation*, das aus den Vertices des ursprünglichen Polygonnetzes und den neu hinzugefügten Vertices besteht. In einem weiteren Schritt werden nun die ursprünglichen Vertices entfernt und dabei lokale Retriangulierungen durchgeführt, so daß die Eigenschaften der ursprünglichen Oberfläche bewahrt bleiben. Laut Turk ist das Verfahren insbesondere bei kurvenreichen Oberflächen effektiv und eignet sich daher beispielsweise für Isoflächen aus dem medizinischen Bereich sowie für per Hand erstellte Modelle von Menschen oder Tieren.

3.3.2. Betrachtung des Verfahrens

In der ersten Phase des Algorithmus wird eine vom Benutzer wählbare Anzahl an neuen Vertices erzeugt und zufällig auf der Oberfläche des Polygonnetzes verteilt. Der Gedanke hierbei ist, daß die ursprünglichen Vertices des Polygonnetzes möglicherweise nicht optimal platziert sind und durch eine Neuverteilung sichergestellt werden soll, daß am Ende des Vorgangs ein wohl-geformtes Dreiecksnetz entstehen kann. Der eigentliche Verteilungsprozess basiert auf der zufälligen Auswahl eines Polygons des Polygonnetzes und platziert nach dieser Auswahl einen neuen Vertex an einer zufälligen Position innerhalb der durch das Polygon begrenzten Fläche. Um eine möglichst gleichmäßige Dichte bei der Verteilung zu gewährleisten, berücksichtigt der Algorithmus bei der Auswahl eines Polygons jedoch seine jeweilige Fläche. Konkret bedeutet dies, daß Polygone mit großem Flächeninhalt eine höhere Wahrscheinlichkeit haben, ausgewählt zu werden, als Polygone mit kleinem Flächeninhalt. Formal lässt sich dies ausdrücken als:

Für ein Polygonnetz M bestehend aus n Polygonen $\{P_1, P_2, ..., P_n\}$ mit den zugehörigen Flächeninhalten $\{A_1, A_2, ..., A_n\}$ ist die Wahrscheinlichkeit für die Auswahl eines Polygons P_i mit $0 < i \leq n$ gegeben durch

$$p(P_i) = \frac{A_i}{A_{Total}}$$

mit

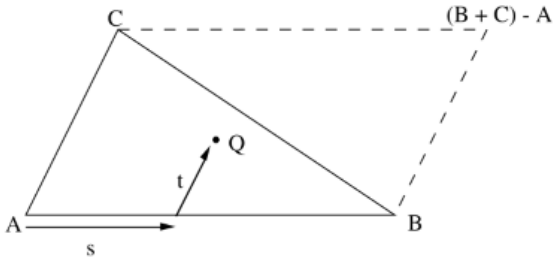
$$A_{Total} = \sum_{i=1}^{n} A_i$$

Nach Auswahl eines Polygons erfolgt nun die zufällige Platzierung des neuen Vertex innerhalb der Fläche des ausgewählten Polygons nach folgendem in [Tur90] beschriebenem Prinzip. Ausgehend von einem Dreieck mit den 3 Eckpunkten $\{A, B, C\}$ werden zwei Zufallswerte s und t im Intervall $[0, 1]$ erzeugt. Mit Hilfe Baryzentrischer Koordinaten lässt sich nun ein zufälliger Punkt Q bestimmen, der innerhalb des Parallelogramms liegt, welches durch die Eckpunkte $\{A, B, C, (B + C) - A\}$ aufgespannt wird:

$$\begin{aligned} a &= 1 - s - t \\ b &= s \\ c &= t \end{aligned}$$

$$Q = aA + bB + cC$$

Abbildung 3.8 demonstriert das Prinzip:



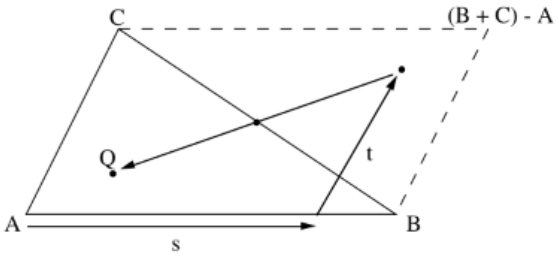
Quelle: [Tur90, S. 26]

Abbildung 3.8.: Bestimmung eines zufälligen Punktes Q

Um zu garantieren, dass ein so erzeugter Punkt innerhalb des Dreiecks $\{A, B, C\}$ liegt, muss lediglich sichergestellt werden, daß stets $s + t \leq 1$ gilt. Dies lässt sich durch folgende simple Erweiterung des Algorithmus erreichen, die anschaulich eine Punktspiegelung des Punktes Q durch den Mittelpunkt des Parallelogramms $\{A, B, C, (B + C) - A\}$ darstellt, falls der erzeugte Punkt außerhalb des Dreiecks $\{A, B, C\}$ liegt:

```
# Punkt auf Fläche des Dreiecks beschränken
if s + t > 1 then
    s = 1 − s
    t = 1 − t
end
```

Abbildung 3.9 veranschaulicht den Fall, daß die Summe der ursprünglichen Werte s und t größer 1 ist.

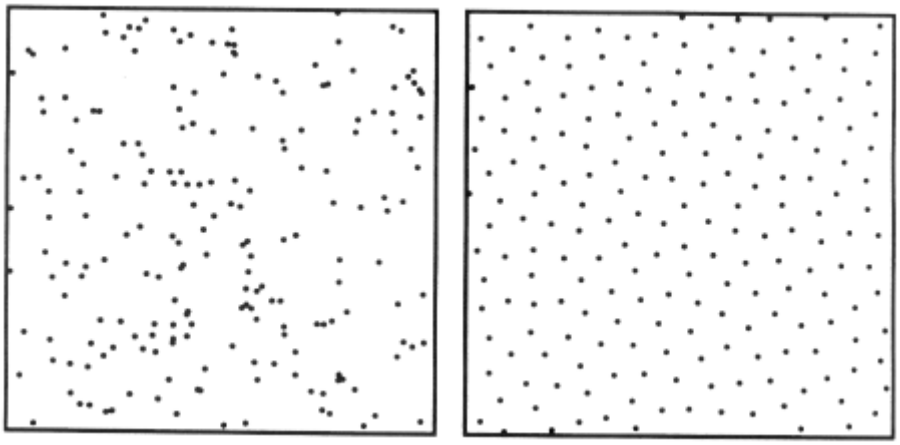


Quelle: [Tur90, S. 26]

Abbildung 3.9.: Punktspiegelung des Punktes Q

Nach Verteilung der Vertices gemäß vorangegangener Beschreibung erfolgt nun in einem zweiten Teilschritt ein Vorgang, der eine möglichst gleichmäßige Positionierung der Vertices gewährleisten soll, ein Prozess, den Turk als *Point Relaxation* bezeichnet [Tur94, S. 52]. Turk führt hierzu eine Abstoßungskraft ein, die jeder Vertex auf seine benachbarten Vertices ausübt. In gewisser Weise ähnelt dieses Prinzip dem Verhalten von Punktladungen gleicher Polarität in der Elektrostatik, die sich aufgrund ihrer wechselseitig wirkenden Abstoßungskräfte stets in einer Gleichgewichtsverteilung anordnen. Um den Rechenaufwand gering zu halten und aus der Beobachtung heraus, daß weit voneinander entfernt liegende Vertices nur zu vernachlässigende kleine Kräfte aufeinander ausüben, wird ein Radius r festgelegt, der das Wirkungsfeld der Abstoßungskraft effektiv beschränkt. Innerhalb dieses abgesteckten Wirkungsfelds nimmt die Kraft mit zunehmendem Abstand von der Position des Vertex linear ab.
In Abbildung 3.10 wird der Vorgang mit einem Vorher-Nachher-Vergleich dargestellt. In der linken Bildhälfte sind zufällig verteilte Vertices in der Ebene dargestellt. In der

rechten Bildhälfte sind die gleichen Vertices abgebildet, nachdem der *Point Relaxation* Prozess durchgeführt wurde.



Quelle: [Tur94, S. 52]

Abbildung 3.10.: Vertices in der Ebene vor und nach Point Relaxation

Um nun die Größe der von einem Vertex p ausgehenden, auf einen Vertex q wirkenden Abstoßungskraft zu bestimmen, wird eine Funktion benötigt, die den Abstand zwischen p und q entlang der Oberfläche des Polygonnetzes angibt. Wenn sich p und q innerhalb desselben Polygons befinden, d. h. in derselben Ebene liegen, kann hierzu der euklidische Abstand zwischen den beiden Vertices herangezogen werden. Liegen p und q hingegen auf zwei benachbarten Polygonen A und B, wird der Vertex q durch eine Rotation um die gemeinsame Kante der Polygone in die durch Polygon A definierte Ebene projiziert. Anschließend kann die euklidische Länge zwischen p und dem projizierten Vertex ermittelt werden. Liegen die beiden Vertices auf Polygonen, die nicht unmittelbar miteinander benachbart sind, so kommt ein ähnliches Verfahren zum Einsatz, welches aus Effizienzgründen jedoch lediglich einen Nährungswert liefert. Für eingehendere Details sei an dieser Stelle auf Turk [Tur94, S. 53-54] verwiesen.

Ausgerüstet mit dieser Distanzfunktion, lässt sich die Berechnung der auf einen Vertex wirkenden Abstoßungskraft nun durch folgenden Pseudocode prägnant beschreiben:

Procedure Compute-Force

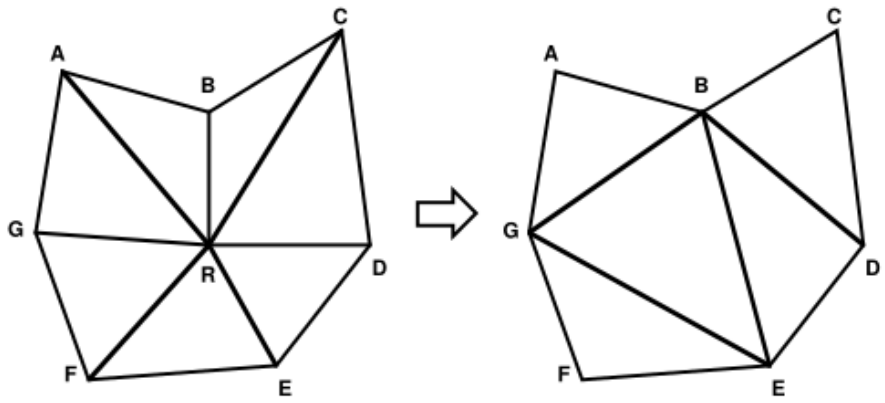
```
# Bestimmt die Kraft, die auf Vertex p einwirkt
F_Abstossung = 0
foreach Vertex q in der Nähe von p do
    if Ebene(p) != Ebene(q) then
        # Vertex q in Ebene von p projizieren
        r = ProjiziereInEbene(q, Ebene(p))
    else
        r = q
    end
    d = Distanz(p, r)   # Kraft ist anti-proportional zur Distanz
    v = Norm(r - p)     # Richtung der resultierenden Kraft
    if d < radius then
        F_Abstossung = F_Abstossung + (radius - d) * v
    end
end
```

Nach Berechnung der kumulierten Abstoßungskraft, die auf einen Vertex einwirkt, wird dieser entsprechend der Richtung und Größe der Kraft entlang der Oberfläche des Polygonnetzes verschoben. Hierbei müssen ähnliche Probleme wie bereits bei der Distanzbestimmung berücksichtigt werden. Wenn ein Vertex beispielsweise über eine begrenzende Kante eines Polygons hinaus verschoben wird, muss er entsprechend rotiert werden, um in die Ebene des angrenzenden Polygons projiziert zu werden. Am Ende dieses Vorgangs hat der Algorithmus einen festgelegten Satz neuer Vertices erzeugt und gleichmäßig verteilt. Im nächsten Schritt wird aus diesen und den ursprünglichen Vertices nun durch Tesselierung ein Dreiecksnetz erzeugt.

Das Erzeugen der von Turk als *Mutual Tessellation* bezeichneten Triangulierung lässt sich recht einfach zusammenfassen. Das Ziel ist es, aus den Polygonen des Polygonnetzes, die nicht zwangsläufig in Dreiecksform vorliegen müssen, unter Berücksichtigung der neu platzierten Vertices ein tesselliertes Dreiecksnetz zu erzeugen. Hierzu wird in einem iterativen Vorgang jedes Polygon betrachtet und es werden die Vertices bestimmt, die innerhalb der vom Polygon begrenzten Fläche liegen. Anschließend dienen diese und diejenigen Vertices, welche die Eckpunkte des betrachteten Polygons darstellen, als Eingabe für den Tessellierungsschritt. Der Algorithmus für das Erzeugen eines Dreiecksnetzes aus dieser Punktmenge ist hierbei beliebig wählbar und wird an

dieser Stelle nicht näher betrachtet. Nach Durchführung der Tesselierung bleibt nun noch im letzten Schritt das Entfernen der ursprünglichen Vertices aus dem erzeugten Dreiecksnetz.

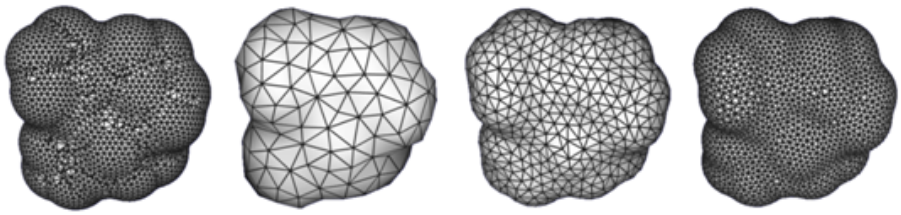
Beim Entfernen der ursprünglichen Vertices des Polygonnetzes bedient sich Turk eines ähnlichen Ansatzes wie bereits Schroeder et al. [SZL92]. Für jeden potentiellen Kandidaten werden die umliegenden Vertices und anliegenden Dreiecke ermittelt. Anschließend wird geprüft, ob der Kandidat gelöscht und das so entstandene Loch durch eine lokale Triangulation geschlossen werden kann. Abbildung 3.11 zeigt das Entfernen eines Vertex und eine mögliche anschließende Re-Triangulierung.



Quelle: [Tur92, S. 4]

Abbildung 3.11.: Re-Triangulierung nach Entfernen eines Vertex

Nach Beendigung dieses Vorgangs hat der Algorithmus das endgültige, vereinfachte Polygonnetz erzeugt, wobei Turk [Tur92, S. 6 ff.] noch einige Sonderfälle und Optimierungsmöglichkeiten in Bezug auf kurvenreiche Oberflächen anführt, die an dieser Stelle jedoch nicht näher betrachtet werden sollen. Zum Abschluß der Betrachtung zeigt Abbildung 3.12 eine Sequenz von durch den Algorithmus erzeugten Vereinfachungen eines dreidimensionalen Modells eines Moleküls. Das ursprüngliche Polygonnetz ist in der Abbildung links dargestellt und verfügt über 3675 Polygone, während die restlichen dargestellten Ausführungen über 201, 801 bzw. 3767 Polygone verfügen.



Quelle: [Tur92]

Abbildung 3.12.: Durch Surface Re-Tiling erzeugte Vereinfachungen

3.4. Evaluierung

Wie in der Einleitung erwähnt, soll im praktischen Teil dieser Arbeit eines der untersuchten Verfahren implementiert werden. Die Implementierung soll dabei als „Grundstein“ für mögliche spätere Arbeiten zur Entwicklung der Visualisierung einer Logistikanwendung dienen. Das Hauptaugenmerk liegt hierbei auf dem Rendering, genauer gesagt, auf der multi-resolutionalen Darstellung komplexer Szenen, wie beispielsweise einer belebten Hafenlandschaft oder dem Be- und Entladen großer Containerschiffe. Vor diesem Hintergrund sollen im Folgenden die Vor- und Nachteile der verschiedenen Verfahren summarisch gegeneinander abgewägt werden; Anschließend soll das am besten geeignete Verfahren zur Implementierung ausgewählt werden.

Ein Vorteil von *Vertex Decimation* gegenüber den anderen Verfahren ist, daß die Vertices der vereinfachten Modelle echte Teilmengen der Vertices des Originals sind. Dies ist insofern nützlich, als es so möglich wird, assoziierte Attribute wie Normalen oder Texturkoordinaten ohne weitere Anpassung zu übernehmen. Nachteilig hingegen ist, daß der Algorithmus eine u. U. ungleichmäßige Tessellierung erzeugt. Auch wird die visuelle Qualität der erzeugten Vereinfachungen durch das Kriterium des Topologieerhalts eingeschränkt. *Surface Re-Tiling* hat diesbezüglich ähnliche Einschränkungen und ist zudem insbesondere für kurvenreiche Oberflächen geeignet. Laut Turks eigenem Bekunden erzielt die Methode für Modelle mit scharfen Kanten, wie sie z. B. bei Gebäuden bestehen, keine optimalen Ergebnisse. So schreibt er [Tur92, S. 2]: *This technique is poorly suited to models that have well-defined corners and sharp edges such as buildings [...].*

Aus den eingangs erwähnten Anforderungen lässt sich schließen, daß der Erhalt der Topologie einzelner Modelle nicht zwingend erforderlich ist. Vielmehr ist bei der Vereinfachung eine möglichst gute visuelle Ähnlichkeit zu den jeweiligen Originalen wünschenswert. Während also prinzipiell alle Verfahren einsetzbar sind, so scheint doch

Garland und Heckberts *Pair Contracting* für diesen speziellen Anwendungszweck am besten geeignet. Im Gegensatz zu Vertex Decimation und Surface Re-Tiling ist der Algorithmus nicht an das Topologiekriterium gebunden und ist nach Ansicht der Autoren daher in der Lage visuell hochwertigere Vereinfachungen von Modellen zu erzeugen. Da der Algorithmus in der Lage ist, inkrementelle Sequenzen von Vereinfachungen zu erzeugen [GH97, S. 2], eignet sich das Verfahren auch zur Kombination mit Datenstrukturen wie *Progressive Meshes* [Hop97]. Aus diesem Grund und wegen der erwähnten Einschränkungen der anderen Verfahren soll im folgenden Kapitel eine exemplarische Implementierung von Garland und Heckberts *Pair Contracting* Algorithmus realisiert werden.

4. Implementierung

4.1. Übersicht

Die Entwicklung eines Systems für das multi-resolutionale Rendering von Objekten lässt sich in zwei separate Schritte einteilen, die unabhängig voneinander umgesetzt werden können:

1. Das Erstellen der Vereinfachungen der bestehenden hochauflösenden Objekte. Hier kommt der eigentliche Vereinfachungsalgorithmus zum Einsatz. Der Vorgang lässt sich im Vorfeld ausführen und liefert als Ergebnis für jedes Objekt eine spezielle Datenstruktur, die progressive Übergänge zwischen beliebigen Detailstufen ermöglicht. Die Berechnung im Vorfeld bietet sich auch deswegen an, da der Vorgang unter Umständen sehr rechenintensiv sein kann.
2. Das eigentliche Rendering der Objekte. Hierzu werden die im ersten Schritt erstellten Datenstrukturen benutzt. Die Polygonzahl eines Objekts lässt sich nun dynamisch regulieren, beispielsweise in Abhängigkeit zur Distanz des Betrachters.

Das Bindeglied zwischen diesen beiden Schritten ist die sog. *Progressive Mesh* [Hop96], eine Datenstruktur, die sich als Nebenprodukt der Vereinfachung eines Objekt erzeugen lässt. Die Idee hierbei ist es, für jeden iterativen Vereinfachungsschritt seine Umkehrung festzuhalten. Das Ergebnis ist dann ein vereinfachtes Polygonnetz und eine Liste von Umkehreinträgen, aus denen sich das originale Polygonnetz und beliebige Detailstufen zwischen Vereinfachung und Original wiederherstellen lassen. Hoppe, der die Datenstruktur in seinem Artikel *Progressive Meshes* [Hop96] vorstellt, tauft die Umkehrung eines Vereinfachungsschrittes *Vertex Split* oder kurz *vsplit*, also die „Aufspaltung“ eines Vertex in zwei neue Vertices.

Im nächsten Abschnitt werden die wichtigsten Implementierungsdetails des Programms *MeshSimplify* erläutert, das für die Umsetzung des ersten Teilschritts und für die Erzeugung der *Progressive Mesh* verantwortlich ist. Der darauf folgende Abschnitt beschreibt die Implementierung der eigentlichen Rendering-Software *MultiRes3d*.

4.2. MeshSimplify

Das Programm *MeshSimplify* ist als Konsolenanwendung realisiert und erwartet als Eingabe ein Polygonnetz sowie einen Satz von Parametern; Ausgabe ist das vereinfachte Polygonnetz bzw. eine *Progressive Mesh* Repräsentation. Eine detaillierte Anleitung zur Benutzung des Programms kann in Anhang A.1 gefunden werden.

Als Implementierungssprache und -umgebung wurden C# und das .NET Framework verwendet.

Die Programmausführung in der Main()-Methode beginnt mit dem Parsen der Kommandozeilenparameter. Erwähnenswert hierbei ist, daß der für die Vereinfachung zu benutzende Algorithmus als Parameter übergeben werden kann. Das Programm erzeugt dann per Reflection eine Instanz der entsprechenden Implementierung. Hierdurch ist es möglich, das Programm mit geringem Aufwand um weitere Vereinfachungsalgorithmen zu erweitern.

```
static void Main() {
  // Kommandozeilenparameter auslesen und verarbeiten.
  var args = ProcessArguments();
  if(args == null)
    return;
  [...]
  try {
    var mesh = ObjIO.Load(args.InputFile);
    // Instanz per Reflection erzeugen.
    using (var algo = InstantiateAlgorithm(args.Algorithm,
       args)) {
      [...]
    }
  } catch (FileNotFoundException) {
    Console.WriteLine("Couldn't find file '{0}'.",
       args.InputFile);
  }
}
```

Listing 4.1: Einstiegspunkt von MeshSimplify

De facto muss für die Implementierung eines neuen Verfahrens lediglich eine neue Klasse erstellt werden, die sich von der abstrakten Basisklasse *Algorithm* ableitet und deren abstrakte Simplify()-Methode implementiert, die über folgende Signatur verfügt:

```
// Vereinfacht die angegebene Eingabemesh.
//
// input
//   Die zu vereinfachende Mesh.
// targetFaceCount
//   Die Anzahl der Facetten, die die erzeugte Vereinfachung
//   anstreben soll.
// createSplitRecords
//   true, um VertexSplit Einträge für die Mesh zu
//   erzeugen; andernfalls false.
// verbose
//   true, um diagnostische Ausgaben während der
//   Vereinfachung zu erzeugen.
// @returns
//   Die erzeugte vereinfachte Mesh.
public abstract Mesh Simplify(Mesh input, int targetFaceCount,
   bool createSplitRecords, bool verbose);
```

Listing 4.2: Signatur der abstraken Methode Simplify

Die Umsetzung des Verfahrens von Garland et al. erfolgt gemäß der Beschreibung des Algorithmus im vorangegangen Kapitel. Der grundlegende Ablauf wird aus der eben beschriebenen Simplify-Methode() ersichtlich, die die Klasse *PairContract* implementiert:

```
// Implementiert den 'Pair-Contract' Algorithmus nach Garland's
// "Surface Simplification Using Quadric Error Metrics"
// Methode.
public override Mesh Simplify(Mesh input, int targetFaceCount,
  bool createSplitRecords, bool verbose) {
  // Wir starten mit den Vertices und Faces der Ausgangsmesh.
  faces = new List<Triangle>(input.Faces);
  for (int i = 0; i < input.Vertices.Count; i++)
    vertices[i] = input.Vertices[i].Position;
  // 1. Die initialen Q Matrizen für jeden Vertex v berechnen.
  Q = ComputeInitialQForEachVertex(options.Strict);
  // 2. All gültigen Vertexpaare bestimmen.
  pairs = ComputeValidPairs();
  // 3. Iterativ das Paar mit den geringsten Kosten
  //    kontraktieren und entsprechend die Kosten aller davon
  //    betroffenen Paare aktualisieren.
  while (faces.Count > targetFaceCount) {
    var pair = pairs.First();
```

```
    Print("Contracting pair ({0}, {1}) with contraction cost =
       {2}",
      pair.Vertex1, pair.Vertex2, pair.Cost.ToString("e"));
    ContractPair(pair);
    Print("New face count: {0}", faces.Count);
  }
  // 4. Neue Mesh Instanz erzeugen und zurückliefern.
  return BuildMesh();
}
```

Listing 4.3: Konkrete Implementierung der Methode Simplify

Die Kontraktionspaare werden hierbei in einer Instanz vom Typ SortedSet<Pair> verwaltet und implementieren das Interface IComparable<Pair> in der Art, daß Paare absteigend nach ihren Kontraktionskosten einsortiert werden. So kann in jeder Iteration einfach das erste Paar entnommen werden, anstatt eine lineare Suche durchführen zu müssen, um das optimale Paar herauszusuchen.

Die eigentliche Berechnung der minimalen Kosten und der optimalen Vertexposition für ein Kontraktionspaar erfolgt in der Methode ComputeMinimumCostPair(). Dies lässt sich in C# aufgrund von Sprachelementen wie dem Überladen von Operatoren, anonymen Typen und der LINQ-Erweiterung recht elegant umsetzen:

```
// Bestimmt die Kosten für die Kontraktion der angegebenen
// Vertices.
//
// s, t
//   Die beiden Vertices des Paares.
// @returns
//   Eine Instanz der Pair-Klasse.
Pair ComputeMinimumCostPair(int s, int t) {
  Vector3d target;
  double cost;
  var q = Q[s] + Q[t];
  // Siehe [Gar97], Abschnitt 4 ("Approximating Error With
     Quadrics").
  var m = new Matrix4d() {
    M11 = q.M11, M12 = q.M12, M13 = q.M13, M14 = q.M14,
    M21 = q.M12, M22 = q.M22, M23 = q.M23, M24 = q.M24,
    M31 = q.M13, M32 = q.M23, M33 = q.M33, M34 = q.M34,
    M41 = 0,     M42 = 0,     M43 = 0,     M44 = 1
  };
  // Wenn m invertierbar ist, lässt sich die optimale Position
     bestimmen.
```

```
  if (m.Determinant != 0) {
    // Determinante ist ungleich 0 für invertierbare Matrizen.
    var inv = Matrix4d.Invert(m);
    target = new Vector3d(inv.M14, inv.M24, inv.M34);
    cost = ComputeVertexError(target, q);
  } else {
    // Ansonsten den besten Wert aus Position von Vertex 1,
       Vertex 2 und Mittelpunkt wählen.
    var v1 = vertices[s];
    var v2 = vertices[t];
    var mp = (v1 + v2) / 2;
    var candidates = new[] {
      new { cost = ComputeVertexError(v1, q), target = v1 },
      new { cost = ComputeVertexError(v2, q), target = v2 },
      new { cost = ComputeVertexError(mp, q), target = mp }
    };
    var best = (from p in candidates
                orderby p.cost
                select p).First();
    target = best.target;
    cost   = best.cost;
  }
  return new Pair(s, t, target, cost);
}
```

Listing 4.4: Berechnung der minimalen Kontraktionskosten

Die für die Erstellung einer *Progressive Mesh* benötigten *VertexSplit*-Einträge werden in der AddSplitRecord()-Methode erstellt. Neben dem Index des betroffenen Vertex und den Positionen der ursprünglichen beiden Vertices muss hier auch festgehalten werden, welche Facetten des Vertex nach der Aufspaltung dem neuen Vertex zugeteilt werden sollen. Erschwert wird dies dadurch, daß die Indices der Vertices der späteren *Progressive Mesh* generell nicht mit den Indices der Vertices des ursprünglichen Polygonnetzes übereinstimmen. Daher wird während der iterativen Vereinfachung eine Zuordnungstabelle aufgebaut, mit deren Hilfe die korrekten Facettenindices für die *VertexSplit*-Einträge später bestimmt werden können.

```
// Erzeugt und speichert einen VertexSplit Eintrag für
// das angegebene Paar.
//
// p
//   Das Paar für welches ein VertexSplit Eintrag erstellt
//   werden soll.
```

```
void AddSplitRecord(Pair p) {
  contractionIndices.Push(p.Vertex2);
  var split = new VertexSplit() {
    S = p.Vertex1, SPosition = vertices[p.Vertex1],
        TPosition = vertices[p.Vertex2]
  };
  foreach (var f in incidentFaces[p.Vertex2]) {
    // -1 wird später durch eigentlichen Index ersetzt,
    // wenn dieser bekannt ist.
    split.Faces.Add(new Triangle(
      f.Indices[0] == p.Vertex2 ? -1 : f.Indices[0],
      f.Indices[1] == p.Vertex2 ? -1 : f.Indices[1],
      f.Indices[2] == p.Vertex2 ? -1 : f.Indices[2]));
  }
  splits.Push(split);
}
```

Listing 4.5: Erstellen der Vertex Splits

Ergänzend lässt sich zu der Implementierung festhalten, daß die Ausführungsgeschwindigkeit durch Umorganisation der vom Algorithmus benötigten Daten enorm gesteigert werden konnte. So hat es sich als sehr wirkungsvoll erwiesen, die Nachbarschaftsbeziehungen zwischen den Vertices des Polygonnetzes zu Beginn der Ausführung zu ermitteln. Auf diese Weise ist der Zugriff auf diese Informationen mit vergleichsweise geringem Rechenaufwand möglich, ähnlich wie bei Half-Edge oder ähnlichen Datenstrukturen. Während erste, naive Implementierungsversuche bei Polygonnetzen mit mehreren hunderttausend Polygonen nahezu zum Erliegen kamen, stellen Polygonzahlen dieser Größenordnung für die jetzige Implementierung kein Problem dar. Auf ähnliche Art ließ sich die Wiederherstellung einer *Progressive Mesh* in das ursprüngliche Polygonnetz beschleunigen. So benötigte eine erste naive Implementierung für die Wiederherstellung einer auf 1000 Polygone reduzierten *Progressive Mesh* zu dem 50.000 Polygon fassenden Original gute einhundert Sekunden. Nach beschriebener Optimierung konnte der gleiche Vorgang in weniger als einer Sekunde durchgeführt werden.

4.3. MultiRes3d

Die Anwendung *MultiRes3d* stellt den zweiten Teil der Implementierung dar und ist für die Visualisierung der durch *MeshSimplify* erzeugten Daten verantwortlich. Einige Beispielausgaben des Programms können in Anhang A.2 betrachtet werden.

Implementiert wurde die Anwendung unter .NET in C#. Als Programmierschnittstelle für die Grafikausgabe wird Direct3D eingesetzt, das Teil des DirectX Frameworks ist. Um aus der .NET Umgebung auf die DirectX bzw. die Direct3D API zugreifen zu können, wird die Wrapper-Bibliothek SlimDX[1] genutzt.

Das Programm ermöglicht das Laden einer zuvor erstellen *Progressive Mesh* und die anschließende dynamische Regulierung der Polygonzahl. Um den Detailgrad zu erhöhen, werden die in der *Progressive Mesh* kodierten *VertexSplit*-Einträge genutzt. Damit der Detailgrad des Polygonnetzes auch wieder vermindert werden kann, erstellt das Programm aus den Informationen eines *VertexSplit*-Eintrags vor dessen Anwendung einen sog. *Contraction*-Eintrag, der die Umkehroperation zu besagtem *VertexSplit* darstellt. Auf diese Weise kann die Polygonzahl dynamisch erhöht oder vermindert werden. Die Anwendung verwaltet hierfür zwei Stacks: Einen Stack, der *VertexSplit*-Einträge enthält, und einen Stack, der *Contraction*-Einträge enthält. Nachdem eine Progressive Mesh geladen wurde, stellt sich die Situation wie in Abbildung 4.1 dar.

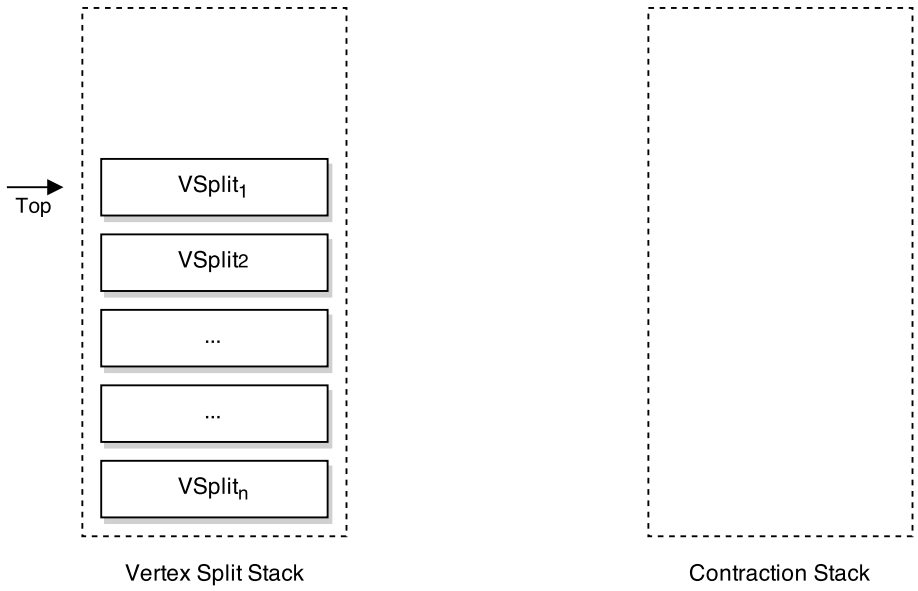


Abbildung 4.1.: VSplit- u. Contraction-Stacks im Ausgangszustand

Die *Progressive Mesh* befindet sich also im maximal vereinfachten „Grundzustand“ und der *VertexSplit*-Stack enthält alle in der *Progressive Mesh* kodierten *VertexSplit*-Einträge. Um nun den Detailgrad des Polygonnetzes zu erhöhen, entnimmt das Programm dem *VertexSplit*-Stack den obersten Eintrag und wendet diesen an. Der Ablauf des Vorgangs wird in Abbildung 4.2 demonstriert.

[1] http://www.slimdx.org/

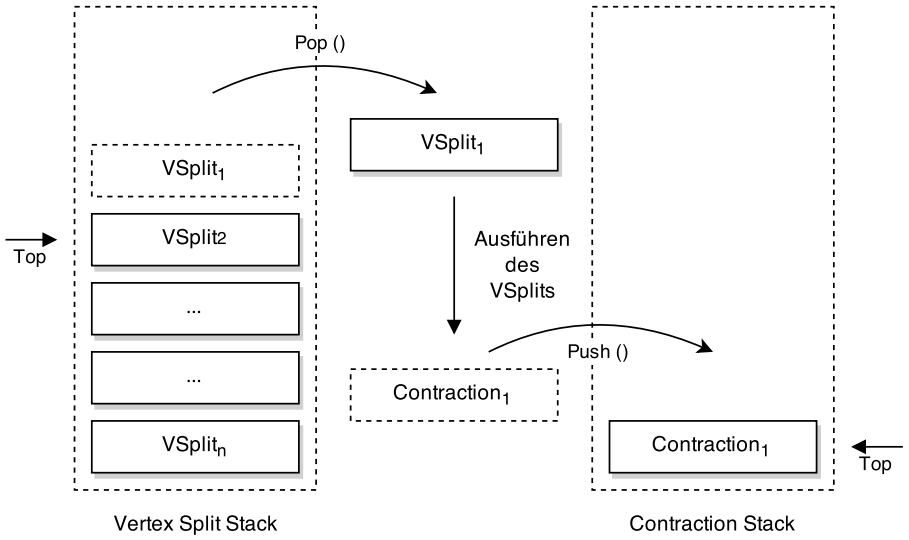


Abbildung 4.2.: Anwendung eines Vertex-Splits

Während der Verarbeitung des *VertexSplits* wird ein *Contraction*-Eintrag erstellt und auf dem *Contraction*-Stack abgelegt. Alle hierfür notwendingen Informationen sind bereits vorhanden. So müssen lediglich die Indices der beiden Vertices sowie die ursprüngliche Position des Vertex, der durch den *VertexSplit* „gespalten“ wird, festgehalten werden. Der Ablauf in gegenteilige Richtung, was einer Vereinfachung des Polygonnetzes entspricht, erfolgt entsprechend analog, wie in Abbildung 4.3 gezeigt wird.

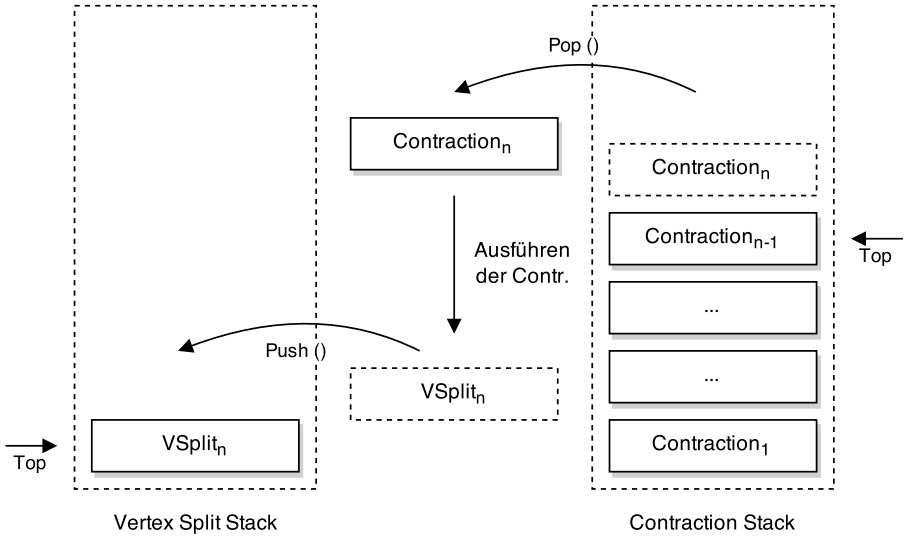


Abbildung 4.3.: Anwendung einer Contraction

Während der Verarbeitung des *Contraction*-Eintrags wird wiederum die Umkehroperation, also der entsprechende *VertexSplit*, auf dem *VertexSplit*-Stack abgelegt. Ein *Contraction*-Eintrag erhält dazu bei seiner Erstellung bereits eine Referenz auf den ursprünglichen *VertexSplit*, dessen Umkehrung er darstellt. Wie aus den Abbildungen und der Beschreibung ersichtlich, bleibt die Summe von VertexSplit- und Contraction-Einträgen stets konstant. Der gegenteilige Fall zu Abbildung 4.1 wäre also ein leerer *VertexSplit*-Stack und ein *Contraction*-Stack mit n Einträgen, was grafisch der maximalen Detailstufe der *Progressive Mesh* entspräche.

Ein weiterer erwähnenswerter Aspekt der Implementierung liegt in der Diskrepanz zwischen dem hohen Abstraktionsgrad von Hochsprachen wie C# und der relativen Hardwarenähe von Grafikschnittstellen wie Direct3D begründet. Während Hochsprachen auf der Basis von Objekten arbeiten, erwarten Grafikschnittstellen gemeinhin Daten, die als flache Arrays primitiver Datentypen vorliegen, wie in Abbildung 4.4 dargestellt wird.

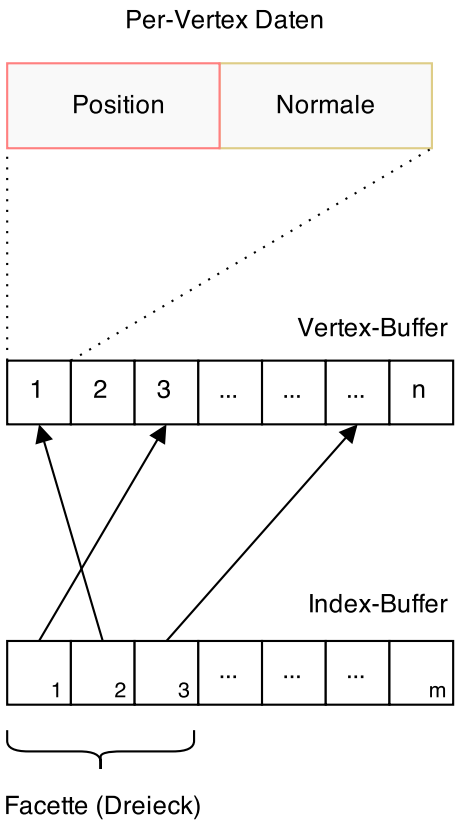


Abbildung 4.4.: Vertex- und Index-Buffer

Die Vertexdaten werden hierbei in einem sog. Vertex-Buffer gespeichert, dessen Inhalt eine Sequenz der Attribute der Vertices des Polygonnetztes sind. Die Facetten-Indices werden in einem separaten Index-Buffer abgelegt, dessen Einträge Offsets in den Vertex-Buffer darstellen. Je drei aufeinander folgende Elemente des Index-Buffers weisen dann eine Facette des Polygonnetzes aus.
Das Problem besteht also in der Serialisierung von Listen von Objekten in Arrays primitiver Typen. Da diese Operationen sehr häufig, u.U. sogar jeden Frame durchgeführt werden müssen, sollten sie so effizient wie möglich umgesetzt werden. Ein denkbarer Ansatz wie in Listing 4.6 ist daher nicht empfehlenswert.

```
uint[] IndexBufferFromFaces(IList<Face> faces) {
  // Es wird bei jedem Aufruf ein neues Heap Objekt angelegt.
  uint[] indexBuffer = new uint[Faces.Length * 3];
  // Es muss in jedem Aufruf über die gesamte Liste
  // iteriert werden.
  for(int i = 0; i < faces.Length; i++) {
    for(int c = 0; c < 3; c++) {
      indexBuffer[i * 3 + c] = faces[i].Indices[c];
    }
  }
  return indexBuffer;
}
```

Listing 4.6: Serialisierung von Facetten-Indices

In der Implementierung wird daher stattdessen einmalig ein Array angelegt, welches die maximale Anzahl an Facetten-Indices aufnehmen kann. Wird nun eine neue Facetten-Instanz erstellt, bekommt diese in ihrem Konstruktor als Eingabe eine Referenz auf das Array und einen Offset, der angibt, an welcher Position die Indices der Facette in dem Array abgelegt werden sollen. Auf diese Weise wird durch Manipulation einer Facette automatisch auch das darunterliegende Array aktualisiert, welches nun als direkte Eingabe für den Index-Buffer dienen kann. Das Verfahren ist in Abbildung 4.5 skizziert.

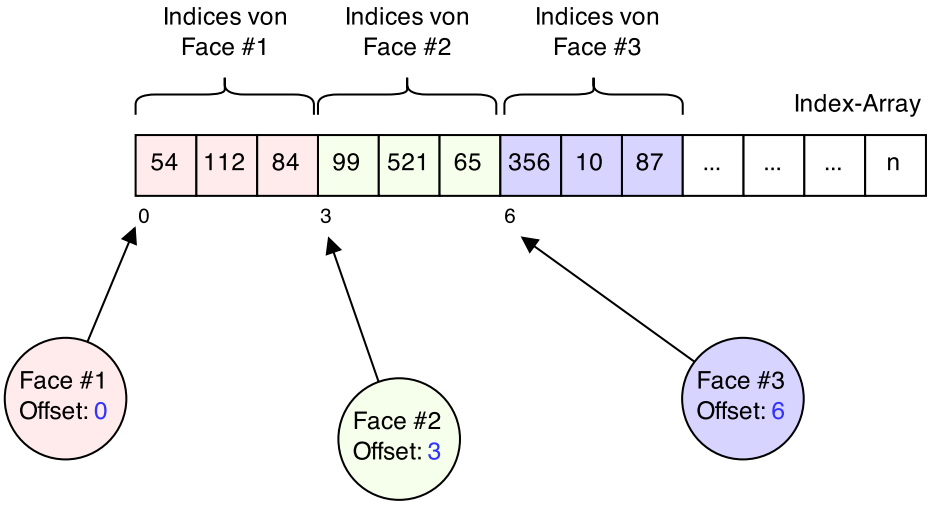


Abbildung 4.5.: Speicherung von Facetten-Indices

Durch Überladen des Index-Operators in der Face-Klasse lässt sich dennoch wie gewohnt auf die Indices der Facette zugreifen, ohne daß sich das Prinzip nach Außen offenbart.

```
// Repräsentiert eine Facette der Progressive Mesh.
public class Face {
  uint[] storage;
  int offset;

  // Überladener Index-Operator zum Zugriff auf die Indices
  // der Facette.
  //
  // @returns
  //  Der jeweilige Index der Facette.
  public uint this[int index] {
    get {
      return storage[offset + index];
    }
    set {
      storage[offset + index] = value;
    }
  }

  // Initialisiert eine neue Instanz der Face Klasse.
  //
  // @indices
```

```
  //  Die Indices der Vertices, die die Facette ausmachen.
  // @storage
  //  Der zugrundeliegene Speicher, in dem die Indices
  //  gespeichert werden.
  // @offset
  //  Der Offset in den Speicher an dem die Indices
  //  gespeichert werden.
  public Face(uint[] indices, uint[] storage, int offset) {
    this.storage = storage;
    this.offset = offset;
    for(int i = 0; i < 3; i++) {
      storage[offset + i] = indices[i];
    }
  }
}
```

Listing 4.7: Ausschnitt der Face-Klasse

5. Schlussbetrachtung

5.1. Zusammenfassung

Das Ziel dieser Arbeit war es, Ansätze zur Vereinfachung von Polygonnetzen zu recherchieren und methodisch zu analysieren. Durch die Untersuchung unterschiedlicher Ansätze wurde dem Leser ein Einblick in dieses weitreichende Forschungsgebiet gewährt. Im Zuge der Untersuchung wurden die verschiedenen Ansätze im Hinblick auf einen realen Anwendungsfall miteinander verglichen und ihre jeweiligen Vor- und Nachteile aufgezeigt.
Durch die anschließende Implementierung eines der Verfahren konnte die praktische Einsetzbarkeit der theoretischen Ausführungen demonstriert werden. Gleichzeitig wurde ein erster möglicher Baustein für die Entwicklung einer Visualisierungssoftware gelegt, auf dem zukünftige Arbeiten aufbauen können.

5.2. Ausblick

Trotz des steten Anstiegs der Rechenleistung moderner Grafikhardware werden Techniken zur Vereinfachung von Polygonnetzen und andere *Level-Of-Detail* Ansätze auch zukünftig eine wichtige Rolle in vielen Bereichen der Computergrafik spielen. Man kann deshalb davon ausgehen, daß auch weiterhin auf diesem Gebiet aktiv geforscht werden wird.

Ausgehend von den gewonnen Erkentnissen dieser Arbeit lassen sich viele weitere Fragestellungen ableiten. Ein noch näher zu untersuchender Aspekt ist z.B. die Bewahrung der oftmals mit Polygonnetzen assoziierten Attribute wie Oberflächennormalen, Farbwerte oder Texturkoordinaten während der Vereinfachung. Ansätze hierzu finden sich etwa bei Garland [GH98] und bei Hoppe [Hop96, S. 6, *Preserving scalar attributes*]. Ein weiterer Aspekt ist der Einsatz der vorgestellten Verfahren zur Vereinfachung offener, weiträumiger Landschaften. Da solche Polygonnetze zumeist sehr groß sind, sich aber in der Regel zu jedem Zeitpunkt nur ein kleiner Ausschnitt im Sichtfeld des Betrachters befindet, wären hier selektive Vereinfachungstechniken wünschenswert, die nur die

momentan sichtbaren Bereiche des Polygonnetzes vereinfachen. Einige Ideen zu diesen Themen finden sich u.a. bei Luebke und Erikson [LE97], Hoppe [Hop97] und bei Xia und Varshney [XV96].

A. Anhang

A.1. MeshSimplify User's Guide

Synopsis

MeshSimplify [*options*] [*input-file*]...

Description

MeshSimplify is a program for polygon mesh simplification. It can be used to create simplified variants of an input mesh with arbitrary numbers of faces. In addition, the program is able to produce progressive-mesh representations as part of the simplification process which can be used for smooth multi-resolutional rendering of 3d objects.

Using `MeshSimplify`

At its most basic level `MeshSimplify` expects the desired target number of faces as well as an input mesh to create the simplification from. Input meshes must generally be triangular and must be provided as wavefront .obj files. For specifying the target number of faces, use the `-n` option:

```
MeshSimplify -n 1000 bunny.obj
```

By default, MeshSimplify produces an output file with the same name as the input file suffixed by _out, so the above example would produce the simplified variant of the input file as bunny_out.obj. To specify a different output location, you can use the `-o` option:

```
MeshSimplify -n 1000 -o output.obj bunny.obj
```

In order to have MeshSimplify add vertex-split records to the resulting output file and produce a progressive-mesh, you can specify the `-p` option:

```
MeshSimplify -n 1000 -p -o pm-bunny.obj bunny.obj
```

Finally, MeshSimplify is also able to restore the original mesh from a progressive-mesh representation or to expand the progressive-mesh to an arbitrary number of faces by specifying the `-restore-mesh` option. So, to expand the previously created progressive-mesh to a polycount of 20000, you could use:

```
MeshSimplify -n 20000 --restore-mesh pm-bunny.obj
```

For a detailed description of all available options, please refer to the next section.

Options

`-a` **ALGORITHM,** `--algorithm` **ALGORITHM** Specifies the simplification algorithm to use. Currently the only valid value for this option is `PairContract` which is an implementation of M. Garland's ''Surface Simplification Using Quadric Error Metrics" approach. Additional algorithms may be supported in the future.

`-d` **DISTANCE,** `--distance-threshold` **DISTANCE** Specifies the distance-threshold value for pair-contraction. This option is only applicable for the `PairContract` algorithm and defaults to 0.

`-h, --help` Shows usage message.

`-n, --num-faces` Specifies the number of faces to reduce the input mesh to.

`-o` **FILE** Specifies the output file.

`-p, --progressive-mesh` Adds vertex-split records to the resulting .obj file, effectively creating a progressive-mesh representation of the input mesh.

`-r, --restore-mesh` Expands the input progressive-mesh to the desired face-count. If this option is specified, the `-n` option refers to the number of faces to restore the output mesh to.

`-s, --strict` Specifies strict mode. This will cause simplification to fail if the input mesh is malformed or any other anomaly occurs.

`-v, --verbose` Produces verbose output.

`--version` Prints version information.

Authors

License

A.2. MultiRes3d Beispielausgaben

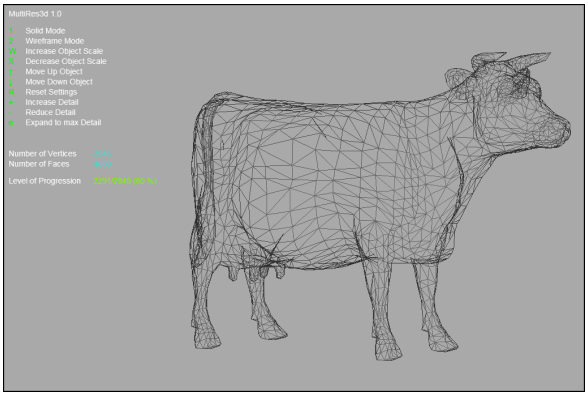


Abbildung A.1.: Progressive Mesh als Gittermodell

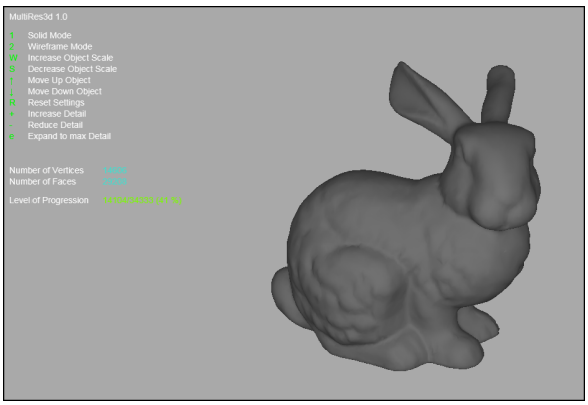


Abbildung A.2.: Progressive Mesh zu 41 % expandiert

A.3. Anwendungsbeispiel

Dieser Abschnitt beinhaltet eine Schritt-für-Schritt Anleitung für die Erstellung einer *Progressive Mesh* aus einer bestehenden Wavefront .obj Datei. Die in diesem Beispiel verwendete .obj Datei kann im beigefügten Datenträger unter dem Dateipfad *meshes/spot_triangulated.obj* gefunden werden.

Generelle Voraussetzungen

1. Die Eingangsmesh liegt als Wavefront .obj Datei vor.
2. Die Eingangsmesh besteht ausschließlich aus dreiseitigen Polygonen.

In diesem Beispiel soll eine *Progressive Mesh* erzeugt werden, die im unexpandierten Zustand über einhundert Polygone verfügt. Der entsprechende Aufruf von MeshSimplify lautet:

```
MeshSimplify --num-faces 100 --progressive-mesh spot_triangulated.obj
```

oder in Kurzschreibweise:

```
MeshSimplify -n 100 -p spot_triangulated.obj
```

Als Resultat wird die Datei *spot_triangulated_out.obj* erstellt, welche die erzeugte *Progressive Mesh* beinhaltet. Die erzeugte Datei kann anschließend mit dem Programm MultiRes3d betrachtet werden.

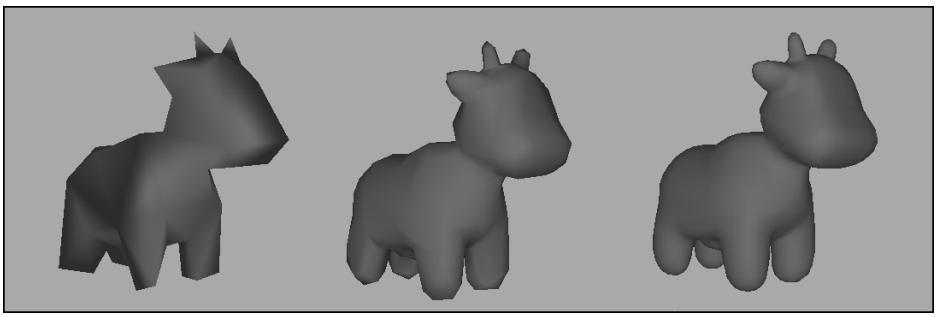


Abbildung A.3.: Erzeugte Progressive Mesh in MultiRes3d

Literaturverzeichnis

[Ada13] M.D. Adams. *Multiresolution Signal and Geometry Processing: Filter Banks, Wavelets, and Subdivision (Version: 2013-09-26):*. Michael Adams, 2013.

[Bau72] Bruce G. Baumgart. Winged edge polyhedron representation. Technical report, Stanford, CA, USA, 1972.

[BKP+10] Mario Botsch, Leif Kobbelt, Mark Pauly, Pierre Alliez, and Bruno Levy. *Polygon Mesh Processing*. AK Peters, 2010.

[CVM+96] Jonathan Cohen, Amitabh Varshney, Dinesh Manocha, Greg Turk, Hans Weber, Pankaj Agarwal, Frederick Brooks, and William Wright. Simplification envelopes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pages 119–128, New York, NY, USA, 1996. ACM.

[Del34] Boris N. Delaunay. Sur la sphère vide. *Bulletin of Academy of Sciences of the USSR*, (6):793–800, 1934.

[DO11] Satyan L. Devadoss and Joseph O'Rourke. *Discrete and Computational Geometry.* Princeton University Press, 2011.

[EET93] Hossam ElGindy, Hazel Everett, and Godfried Toussaint. Slicing an ear using prune-and-search. *Pattern Recogn. Lett.*, 14(9):719–722, September 1993.

[EK81] C. M. Eastman and Preiss K. *A Unified View of Solid Shape Modeling Based on Consistency Verification*. Carnegie-Mellon University, September 1981.

[FM84] A. Fournier and D. Y. Montuno. Triangulating simple polygons and equivalent problems. *ACM Trans. Graph.*, 3(2):153–174, April 1984.

[GH97] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 209–216, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[GH98] Michael Garland and Paul S. Heckbert. Simplifying surfaces with color and texture using quadric error metrics. In *Proceedings of the Conference on Visualization '98*, VIS '98, pages 263–269, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[Gué95] André Guéziec. *Surface simplification with variable tolerance*. Second Annual Intl. Symp. on Medical Robotics and Computer Assisted Surgery (MRCAS 1995), November 1995.

[HDD+93] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh optimization. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '93, pages 19–26, New York, NY, USA, 1993. ACM.

[Hop96] Hugues Hoppe. Progressive meshes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pages 99–108, New York, NY, USA, 1996. ACM.

[Hop97] Hugues Hoppe. View-dependent refinement of progressive meshes. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 189–198, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[LC87] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, pages 163–169, New York, NY, USA, 1987. ACM.

[LE97] David Luebke and Carl Erikson. View-dependent simplification of arbitrary polygonal environments. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 199–208, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[LT97] Kok-Lim Low and Tiow-Seng Tan. Model simplification using vertex-clustering. In *Proceedings of the 1997 Symposium on Interactive 3D Graphics*, I3D '97, pages 75–ff., New York, NY, USA, 1997. ACM.

[Lue01] David P. Luebke. A developer's survey of polygonal simplification algorithms. *IEEE Comput. Graph. Appl.*, 21(3):24–35, May 2001.

[Muk12] Ramakrishnan Mukundan. *Advanced Methods in Computer Graphics - With examples in OpenGL.* Springer, 2012.

[RB93] Jarek Rossignac and Paul Borrel. Multi-resolution 3d approximations for rendering complex scenes. In Bianca Falcidieno and Tosiyasu L. Kunii, editors, *Modeling in Computer Graphics*, IFIP Series on Computer Graphics, pages 455–465. Springer, 1993.

[RR96a] Rémi Ronfard and Jarek Rossignac. Full-range approximation of triangulated polyhedra. *Computer Graphics Forum*, 15(3):67–76, August 1996. Special Issue: Proceedings of Eurographics '96, Poitiers, France. Jarek Rossignac and François Sillon eds.

[RR96b] Rémi Ronfard and Jarek Rossignac. Full-range approximation of triangulated polyhedra. *Computer Graphics Forum*, 15(3):67–76, 1996.

[SL96] Marc Soucy and Denis Laurendeau. Multiresolution surface modeling based on hierarchical triangulation. *Comput. Vis. Image Underst.*, 63(1):1–14, January 1996.

[Smi06] Colin Smith. *On Vertex-vertex Systems and Their Use in Geometric and Biological Modelling*. PhD thesis, University of Calgary, Calgary, Alta., Canada, 2006. AAINR19574.

[SSKLK13] Dave Shreiner, Graham Sellers, John M. Kessenich, and Bill M. Licea-Kane. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*. Addison-Wesley Professional, 8th edition, 2013.

[SW03] Scott Schaefer and Joe Warren. Adaptive vertex clustering using octrees. In *SIAM Geometric Design and Computing*, 2003.

[SZL92] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of triangle meshes. In *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '92, pages 65–70, New York, NY, USA, 1992. ACM.

[Tur90] Greg Turk. Graphics gems. chapter Generating Random Points in Triangles, pages 24–28. Academic Press Professional, Inc., San Diego, CA, USA, 1990.

[Tur91] Greg Turk. Generating textures on arbitrary surfaces using reaction-diffusion. In *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '91, pages 289–298, New York, NY, USA, 1991. ACM.

[Tur92] Greg Turk. Re-tiling polygonal surfaces. In *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '92, pages 55–64, New York, NY, USA, 1992. ACM.

[Tur94] Greg Turk. Texturing surfaces using reaction-diffusion. Technical report, Chapel Hill, NC, USA, 1994.

[XV96] Julie C. Xia and Amitabh Varshney. Dynamic view-dependent simplification for polygonal models. In *Proceedings of the 7th Conference on Visualization '96*, VIS '96, pages 327–ff., Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.

Versicherung über die Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §16(5) APSO-TI-BM ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen habe ich unter Angabe der Quellen kenntlich gemacht.

Ort, Datum	Unterschrift