Humboldt-Universität zu Berlin
Institut für Informatik

Polygonreduktion eines in der Half-Edge-Datenstruktur abgebildeten mit dem Silhouettenschnitt rekonstruierten 3D Objektes

Aleksander Gudalo
31. Mai 2012

Lehrstuhl Visual Computing
Prof. Dr. Peter Eisert

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe. Weiterhin erkläre ich, eine Studienarbeit in diesem Studiengebiet erstmalig einzureichen.

Berlin, den 31. Mai 2012 .....

Inhaltsverzeichnis

1 Einleitung

Ein Bild soll als technisches Medium häufig ein Abbild des Realen darstellen. Das Abbild dient somit primär der Speicherung, aber auch der Informationsverarbeitung des realen Gegenstandes. Seit den Anfängen der Höhlenmalerei vor über 30000 Jahren versucht man das abgebildete Modell immer naturgetreuer darzustellen. Dies galt lange in der Kunst als Maßstab für Qualität, so wie es heute im technischen und wissenschaftlichen Umfeld gilt. Modelle, die am Computer modelliert werden haben Abbildungsfehler, die je nach Einsatzgebiet Auswirkungen auf das Ergebnis haben können. Auch die selektive Wahrnehmung des Gestalters führt oft zu einer geringen objektiven Darstellung eines Modells. Viele weitere Schwächen bei der manuellen Rekonstruktion sind uns bekannt.

Deshalb ist es sinnvoll, Modelle direkt von der realen Umgebung objektiv abzuleiten ohne selbst in den Prozess einzugreifen. Dieser grundlegende Unterschied in der Erzeugung eines Modells kann jedoch zu einer enormen Datenmenge führen, die nicht mehr handhabbar ist. Darum bedarf es nach der Generierung eines komplexen Modelles einer Vereinfachung dieses Objektes ohne dabei seine charakteristische Form zu verändern.

Ausgehend von einem realen Objekt sollte in dieser Arbeit ein dreidimensionales Computermodell anhand von Fotografien rekonstruiert werden. Um das Objekt von allen Seiten fotografisch festzuhalten wurde ein Drehteller eingesetzt. Ein Teil dieser Arbeit ist deshalb die Implementierung der Steuerung des Drehtellers.

Aus einem Satz von Bildern wurde ein Voxelmodell erstellt, welches dann in ein polygonales Modell umgewandelt werden konnte. Bei der Erzeugung des dreidimensionalen Objektes fielen große Datenmengen an, die visualisiert werden mussten. Deshalb wurde das dabei entstandene Datenvolumen zugunsten einer einfacheren Handhabung reduziert, ohne dabei wichtige Informationen des Objektes zu verlieren.

Das von Michael Garland vorgestellte Verfahren zur Vereinfachung von polygonalen Netzen ist die Grundlage des hier implementierten Verfahrens. Dieser Algorithmus bietet neben einer schnellen Datenverarbei-

tungsleistung auch eine geringe Abweichungen von dem originalen Modell.

1.1 Aufgabenstellung

In dieser Arbeit wird beschrieben, wie sukzessiv aus einem realem Objekt ein polygonales dreidimensionales Modell erzeugt wird. Dabei liegt der Schwerpunkt dieser Arbeit bei der Simplifikation der komplexen Eingabe.

Man kann die einzelnen Stufen dieser prototypischen Rekonstruktion wie folgt unterscheiden. Im ersten Schritt wurde eine Fotoserie, die das Objekt allseitig erfasst, angelegt und in einer graphischen Anwendung angepasst. Aus diesem hochaufgelösten Fotomaterial wurde dann im zweiten Schritt ein einfaches Voxelmodell erzeugt. Danach wurde im dritten Schritt eine Triangulation des Voxelmodells vorgenommen. Das polygonale Netz wurde daraufhin, im letzten Schritt, für die weitere Nutzung vereinfacht und anschließend in ein gängiges Dateiformat überführt.

Es wurde bei der Implementierung der verschiedenen Algorithmen Wert darauf gelegt, dass diese leicht wieder verwertet werden können. Es mussten einige Verfahren an die konkreten Bedürfnisse angepasst werden, die jedoch über Schalter steuerbar sind. Um die mögliche Nutzung der verschiedenen Algorithmen auch in weiteren Projekten zu gewährleisten wurde auf bekannte *C++* Bibliotheken zurückgegriffen.

1.2 Aufbau der Arbeit

Das Kapitel *Grundlagen* soll die theoretischen Grundlagen, die für das allgemeine Verständnis der erarbeiteten Algorithmen nötig sind, beschreiben. Es wird allgemein erläutert, wie ein Modell in der Computergraphik repräsentiert wird und welche Vorteile diese Darstellungsweisen haben. Außerdem wird beschrieben wie die verwendeten Algorithmen aufgebaut sind und welche Vorzüge sie anderen bekannten Verfahren gegenüber besitzen.

Im Kapitel *Datenerfassung* werden die ausgeführten Schritte, die zur Erzeugung des Modells durchlaufen wurden, beschrieben. Auf die Implementierung der Drehtellersteuerung wird in diesem Kapitel ausführlich eingegangen.

Das Kapitel *Datenverarbeitung* beschreibt die Algorithmen, die eingesetzt wurden, um das Modell zu bearbeiten. Hier wird die Implementierung wichtiger Komponenten des Projektes betrachtet und die Arbeitsweise des Verfahrens bewertet.

Die Resultate werden im folgenden Kapitel besprochen und es findet eine ausführliche Bewertung statt um dann im letzten Kapitel auf mögliche Verbesserungen der Verfahren eingehen zu können.

1.3 Motivation

Diese Arbeit wurde angeregt durch die aufwendige und nur mäßige Qualität bei der manuellen Erstellung von dreidimensionalen Modellen. Es wurde nach einem einfachen und günstigen Verfahren gesucht um Objekte naturgetreu in virtuelle Computermodelle zu überführen, die dann in weiteren Anwendungen benutzt werden können.

Die Suche nach einer effizienten Methode, die erfassten Daten zu komprimieren, um das Modell auch in einfachen Anwendungen nutzen zu können, war die Motivation zur Implementierung des *Quadric-Based*-Algorithmus, der für Effizienz und hohe Qualität sorgt. Dabei wurde besonders auf die allgemeine Anwendbarkeit des Verfahrens Wert gelegt um beliebige Objekte bearbeiten zu können.

2 Grundlagen

2.1 Dreidimensionale Objektrepräsentation

Dreidimensionale Computermodelle können verschiedenartig beschrieben und erzeugt werden. Sie definieren, neben der Geometrie, auch verschiedene andere Eigenschaften eines konkreten oder abstrakten Objektes. Bei der Modellierung sollten laut Bender und Brill einige Kriterien beachtet werden, die Auswirkungen auf die Qualität haben [Bender 2006]. Es gibt ein reales Objekt, von dem das Computermodell abgeleitet werden kann, und das man zur Beurteilung des Modells heranziehen kann. Die Genauigkeit orientiert sich am Anwendungsgebiet. Dabei schränkt die Verarbeitungsgeschwindigkeit eine beliebig genaue Repräsentation ein.

Die geometrische Datenverarbeitung, ein Teilgebiet der Praktischen Informatik, aus dem die Computergraphik letztendlich hervorgegangen ist [Bender 2006], beschreibt vier grundlegende Arten der Modellrepräsentation. Die vier Repräsentationsformen sind die Punktwolke, das Drahtmodell, das Flächenmodell und das Volumenmodell [Zdunczyk 2006]. Diese elementaren Modelle weisen eine Art Hierarchie auf. Die Punktwolke stellt dabei die einfachste Form dar, gefolgt von dem Drahtmodell, dass die einzelnen Punkte miteinander in Verbindung setzt. Danach kommt das Flächenmodell, welches die Kanten durch Flächen verbindet. Am Ende der Hierarchie steht somit das Volumenmodell, das wiederum das vorherige Modell erweitert. Einige der Modelle lassen sich ineinander überführen. So kann etwa aus dem Volumenmodell ein Flächenmodell entwickelt werden [Lorenson 1987] .

In dieser Arbeit werden das Volumenmodell und das Flächenmodell ausführlicher erläutert, da sie Bestandteil des entwickelten Verfahrens sind. Die Punktwolkendarstellung ist als einfachstes Modell nicht ausreichend um ein Modell genau genug zu beschreiben. Je dichter die Punkte jedoch sind, desto genauer wird das dargestellte Objekt repräsentiert. Aus dieser Punktwolke kann heuristisch eine Oberfläche generiert werden, die häufig jedoch nicht das gewünschte Ergebnis liefert [Delaunay 1934].

Beim Drahtmodell repräsentieren die Kanten zwischen den Punkten

das eigentliche Modell. Diese Darstellungsform ist aussagekräftiger, bietet aber auch keine eindeutige Interpretation der aufgespannten Flächen.

2.1.1 Volumenmodell

Ein Volumenmodell wird durch seine Volumina beschrieben. In dieser Arbeit wird die Voxeldarstellung genutzt um ein Modell zu repräsentieren. Der Begriff Voxel leitet sich aus den Begriffen *Volumetric* und *Pixel*-Element ab [Otto 2005]. Ein Voxel ist das dreidimensionale Pendant zu einem Pixel im zweidimensionalen Raum. Aneinander gereihte regelmäßig aufgeteilte Würfelzellen mit gleicher Kantenlänge beschreiben dabei ein räumliches Objekt. In den Würfelmodellen wird die Position eines Voxels in kartesischen Koordinaten abgebildet und beschreibt den Mittelpunkt einer Zelle. Je nach gewünschter Genauigkeit wird das Modell durch entsprechend viele Voxel repräsentiert. Die Voxel, die im repräsentierten Körper liegen, werden dann in einer Liste gespeichert. Oft wird dafür ein hierarchisches Verfahren verwendet [Häuser 1998].[1]

Anwendung findet diese Form der Datenrepräsentation vor allem in den bildgebenden Verfahren der Medizin [Lehmann 2005]. Sie werden beispielsweise in der Computertomographie (CT) oder Magnetresonanztomographie (MRT) verwendet.

Es existieren unterschiedliche Verfahren anhand derer Volumendaten in Echtzeit visualisiert werden können. Eine Art der Visualisierung ist die Verwendung polygonaler Schnittflächen, ein anderer Ansatz besteht im Ray-Casting [Otto 2005].

[1] z. B. Octree oder BSP-Baum

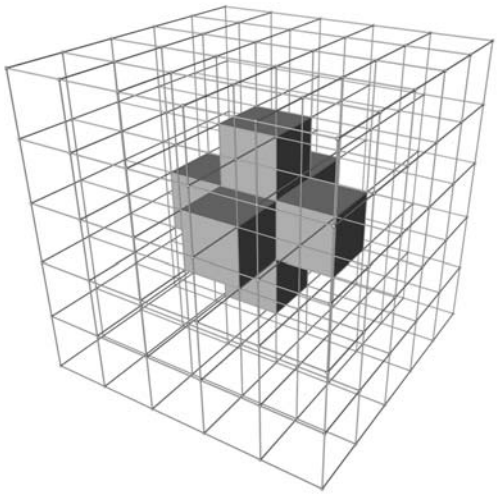


Abbildung 1: Beispiel eines einfachen Voxelmodells

2.1.2 Polygonmodell

Ein Polygonmodell als Flächenmodell wird über seine Außenflächen bestimmt und ist innen ausgehöhlt. Planare polygonale Netze bestimmen die Oberfläche und werden besonders bei sehr komplexen und stark gekrümmten Objekten verwendet. Die einfachste und meist verbreitetste Form des Polygons ist die des Dreiecks. Dieses ist immer planar und konvex und eignet sich wegen seiner Einfachheit besonders gut zur Darstellung von Polygonmodellen. Da es besonders effiziente Algorithmen und eine gute Integration in der Grafikhardware gibt, kann damit auch besonders effektiv gearbeitet werden. Auch Vierecke sind häufig verwendete Polygonformen, die besonders in Kombination mit funktional definierter Geometrie eingesetzt werden. Es ist aber grundsätzlich jede Polygonform möglich um Flächenmodelle zu erzeugen.

Polygone werden über ihre Eckpunkte $V_0 \dots V_n$, die sogenannten Vertices definiert. Stimmt der erste Eckpunkt mit dem letzten überein $V_0 = V_n$, so spricht man von einem *geschlossenen Polygonzug* [Zdunczyk 2006] oder einer *geschlossenen Polyline* [Bender 2006]. Dieser besteht aus mindestens drei Eckpunkten und drei verbundenen Kanten und bildet somit im einfachsten Fall ein Dreieck. Ein *offener Polygonzug* ist hingegen da-

durch gekennzeichnet, dass Anfangs- und Endpunkt nicht identisch sind $V_0 \neq V_n$.

Die Polygone werden mit einer festen Orientierung beschrieben, die anhand der Normalen die Vorder- und Rückseite eines Polygons festlegt. Die Normale ist der Vektor, der senkrecht auf dem Polygon steht und somit durch seine Richtung die Seiten bestimmt. Erst dadurch sind die Außen- und Innenseite eines Objektes fest definiert.

Es gibt verschiedene Datenstrukturen, die sich zur Speicherung von Polygonnetzen eignen und die man in drei Kategorien einteilen kann. Bei der polygonbasierenden Datenstruktur werden die Polygone in einer ungeordneten Liste von Koordinaten der Eckpunkte gespeichert. $P = \big((x_1, y_1, z_1), (x_2, y_2, z_2), \ldots, (x_n, y_n, z_n)\big)$ beschreibt diese Liste, bei der es zu Redundanzen der Vertices kommen kann.

Die speichersparende knotenbasierende Datenstruktur ist aber wohl die in der Praxis am häufigsten verwendete Form. Hier werden die Vertices $V = \big((x_1, y_1, z_1), (x_2, y_2, z_2), \ldots, (x_n, y_n, z_n)\big)$ nur einmal gespeichert und durch Indizes mit einer meist festen Orientierung über die Polygone $P = \big((1,2,3), (1,2,4), \ldots\big)$ referenziert.

Daneben gibt es noch die kantenbasierende Datenstruktur, die neben einer Vertexliste $V = \big((x_1, y_1, z_1), (x_2, y_2, z_2), \ldots, (x_n, y_n, z_n)\big)$, auch eine Kantenliste mit weiteren zusätzlichen Informationen über die Polygone besitzt.

2.1.3 Half-Edge Datenstruktur

Eine kantenbasierende Datenstruktur, die sich besonders für den Einsatz der Mesh-Reduktionen eignet, ist die *Half-Edge*-Struktur. Dabei sind alle Verbindungsinformationen des Polygonnetzes in Beziehung zu den Kanten gesetzt. Allerdings werden nicht einfach die Kanten des Meshes gespeichert sondern jeweils zwei gerichtete gegenläufige Halbkanten. Durch die Richtung der Halbkanten wird zusätzlich die Orientierung des Polygons festgelegt.

In Abbildung 2 sind die Kanten h und die dazu entgegengesetzte Kante o die Halbkanten, die die beiden Vertices v_0 und v_1 verbinden. Zu sehen

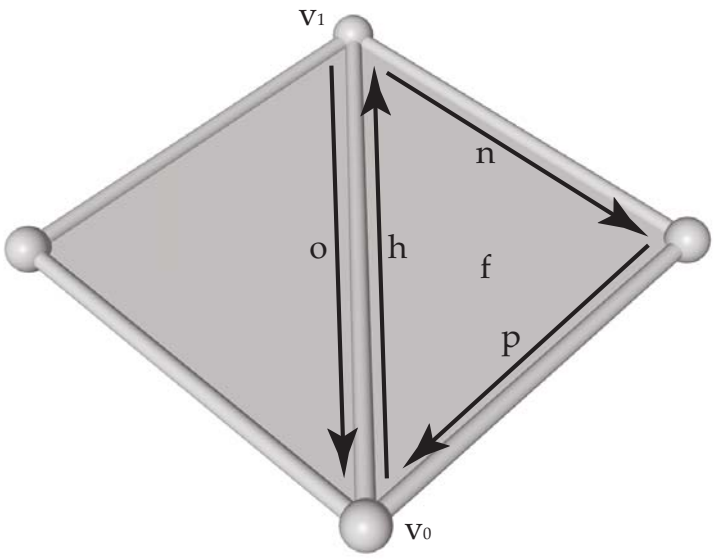


Abbildung 2: *Half-Edge*-Struktur

ist auch die Vorgängerhalbkante p und die nachfolgende Halbkante n, die zusammen ein Polygon f aufspannen.

Neben der Vertex- und Faceliste muss bei dieser Datenstruktur auch eine Half-Edge-Liste gespeichert werden. Die Vertexliste (Quelltext 1) führt neben den Koordinaten des Punktes auch eine indexierte ausgehende Halbkante.

```
struct half_edge_vertex {
 float x;
 float y;
 float z;

 half_edge*          he;
};
```

Quelltext 1: Halbkanten Vertex Struktur

In der Faceliste (Quelltext 2) wird zusätzlich ein Index auf eine angrenzende Halbkante des Polygons gespeichert.

In der Half-Edge-Liste (Quelltext 3) ist das Vertice auf das sie verweist, das angrenzende Polygon, der nächste Half-Edge innerhalb dieses Poly-

```
struct half_edge_face {
 half_edge*          he;
};
```

Quelltext 2: Halbkanten Polygon Struktur

gons, die gegenläufige Halbkante und optional die Vorgänger-Halbkante des Polygons gespeichert [OpenMesh 2012].

```
struct half_edge {
 half_edge_vertice  vertice;
 half_edge_face     face;
 half_edge          opposit;
 half_edge          next;
 half_edge          prev;
};
```

Quelltext 3: Halbkanten Struktur

Mit dieser Struktur ist es leicht durch das Polygonnetz zu navigieren und Nachbarschaftsbeziehungen zu suchen, da die Vorgängerkante, die Nachfolgerkante und die angrenzenden Flächen eindeutig bestimmt sind. Diese Operationen können in konstanter Zeit unabhängig von der Meshgröße implementiert werden. Ausgehend von der Halbkante können die Eckpunkte und das angrenzende Polygon mit einer Laufzeit von $\mathcal{O}(1)$ abgefragt werden. Von einem Eckpunkt aus, kann eine eingehende oder ausgehende Kante und ein angrenzendes Polygon in $\mathcal{O}(n_v)$ ermittelt werden, wobei n_v die Anzahl der angrenzenden Kanten dieses Eckpunktes wiedergibt. Startet man bei einem Polygon, so können alle Kanten und Eckpunkte in einer Laufzeit von $\mathcal{O}(n_e)$ ermittelt werden, wobei n_e die Anzahl der angrenzenden Kanten des Polygons wiedergibt.

Auch eine sogenannte *Edge-Collapse*-Funktion, bei der zwei Vertices einer Kante zu einem Vertex verschmelzen, ist einfach zu realisieren und wird im folgenden Abschnitt 2.4.1 ausführlicher besprochen.

Von Nachteil ist neben der Eigenschaft, dass nur mannigfaltige Polygonnetze gespeichert werden können, vor allem der erhöhte Speicherverbrauch.

2.2 Dreidimensionale Rekonstruktion

Nach Bungartz ergeben sich grundsätzlich drei wesentliche Arbeitsschritte bei der graphischen Datenverarbeitung [Bungartz 2002]. Zuerst muss ein dreidimensionales Modell, meist von einem realen Vorbild ausgehend, erzeugt werden. Danach findet eine Abbildung dieses Modelles in einem zweidimensionalen Bildraum statt, um diesen dann am Ende auf einem Ausgabegerät wiederzugeben. In dieser Arbeit wird nur der erste Arbeitsschritt ausgeführt und erläutert. Eine Darstellung wird über externe Programme durch die Ausgabe in ein gängiges Dateiformat ermöglicht.

Die geometrische Modellierung des realen Gegenstandes findet hier wiederum in zwei Schritten statt. Aus einer Serie von Fotografien wird ein Voxelmodell mit der *Silhouetten-Schnitt*-Technik erzeugt. Daraus wird dann, mit dem *Marching-Cube*-Algoritmus, im folgenden Schritt ein Polygonmodell abgeleitet.

Die Erläuterung dieser beiden Arbeitsschritte bezieht sich auf die Vorarbeit von Ziskel [Ziskel 2004].

2.2.1 Silhouetten-Schnitt-Technik

Bei der Silhouetten-Schnitt-Technik oder dem sogenannten *Shape-From-Silhouette*-Verfahren (SFS) wird aus der Silhouette des realen Objektes die Form schrittweise berechnet. Dafür werden Bildaufnahmen aus verschiedenen Perspektiven (Abbildung 3), mit bekannten Drehwinkel und für jedes Bild erzeugt, um sukzessiv ein Voxelmodell zu extrahieren [Szeliski 1993]. Für jede Perspektive $v_n \in V$ wird eine Silhouette Δv_n ausgeschnitten. Für jeden Pixel p_m des Fotos wird getestet, ob dieser ein Teil des Objektes ist. Ist der Pixel innerhalb des Objektes $p_m \bigcap_{v_n}$, so wird ein Voxel markiert. Die rekonstruierte visuelle Hülle h_p ist dann die Approximation der Punkte, die sich innerhalb des Objektes befinden.

Szeliski weist allerdings schon darauf hin, dass konkave Objekte nicht einfach rekonstruiert werden können, da diese Punkte nicht in der Silhouettenhülle erscheinen. Außerdem ist die Qualität des Verfahrens stark abhängig von der genauen Kamerakalibrierung [Eisert 2004]. Das Objekt

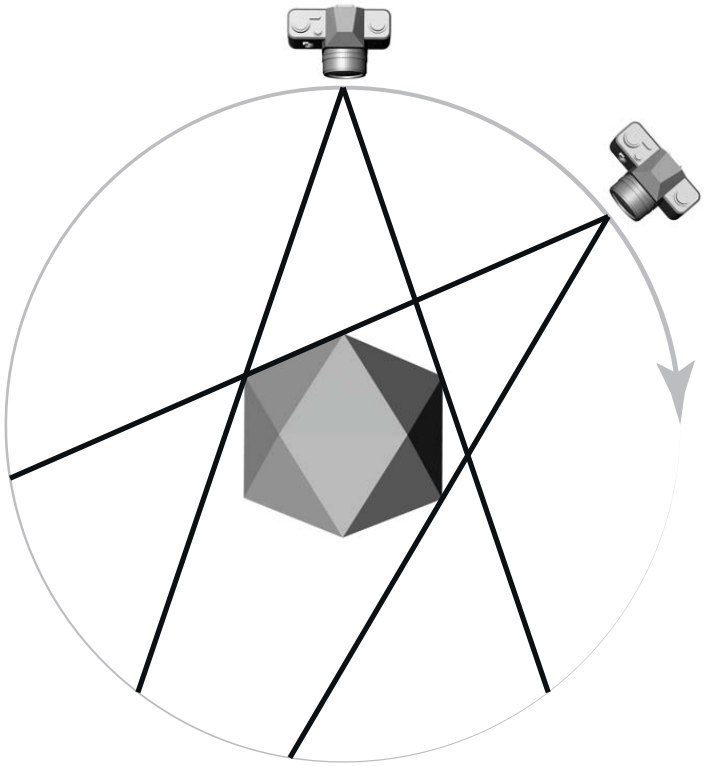


Abbildung 3: Fotografien aus verschiedenen Perspektiven

sollte sich stark von Hintergrund abheben um die Formen eingrenzen zu können. Ein farblicher Schwellenwert bestimmt dann, ob ein Farbpunkt zu einem Objekt gehört oder ob dieser außerhalb des Objektes liegt.

Aus diesem Voxelmodell kann dann ein Polygonnetz erzeugt werden.

2.2.2 Marching-Cubes-Algorithmus

Um das Voxelobjekt weiter bearbeiten zu können ist es nötig, dieses in ein geschlossenes Dreiecksmesh umzuwandeln. Dafür wird das *Marching-Cubes*-Verfahren eingesetzt. Dieses erzeugt eine sinnvolle Dreiecksoberfläche an den Übergängen zu den Voxeln. Es werden von jedem Voxel alle sechs möglichen Nachbarn betrachtet. Daraus ergibt sich für die acht Eckpunkte eine Auswahl von $2^8 = 256$ möglichen Kombinationen. Durch Rotationen und Spiegelungen kann man diese auf 15 verschiedene Kombinationen reduzieren, die in einer *Lookup*-Tabelle (Abbildung 4) abgelegt

werden. Dadurch erhält man für die Rekonstruktion eine konstante Laufzeit von $\mathcal{O}(n)$, mit n der Anzahl an Voxeln.

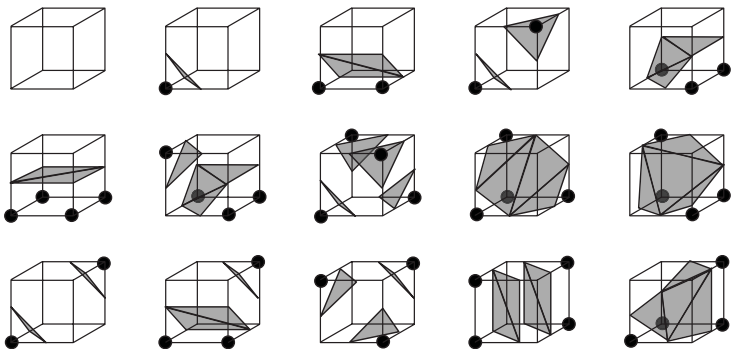


Abbildung 4: *Lookup*-Tabelle der 15 verschiedenen Kombinationen

Es gibt einige Mehrdeutigkeiten, die ein degeneriertes Polygonnetz erzeugen können. Durch Erweiterung des Verfahrens und Vergrößerung der *Lookup*-Tabelle kann dies jedoch verhindert werden [Chernyaev 1995].

Um weichere Oberflächenstrukturen zu bekommen kann mit Schwellenwerten linear interpoliert werden.

2.3 Qualitätsverbesserung durch Glättung

Da nach der Erzeugung des Polygonnetzes durch die bereits beschriebenen Verfahren ein stark eckiges Modell entsteht, sollte dieses vor der weiteren Bearbeitung verbessert werden. Es ist sinnvoll eine Gitternetzglättung vorzunehmen um weiche Übergänge zwischen den Polygonen zu erzeugen.

Ein einfaches aber effizientes heuristisches Verfahren mit einem geringen Speicherverbrauch ist die *Laplace-Smoothing*-Technik. Dabei werden die Vertices $v \in M$ des Polygonnetzes inkrementell so verschoben, dass ein Mittelwert aller adjazenten Vertices N für die neue Vertexposition $\bar{v}$ berechnet wird [Hansen 2005]. Es gilt also für M:

$$\bar{v}_i = \frac{1}{N_i} \sum_{j \in N_i} v_j$$

mit N_i der Valenz von v_i.

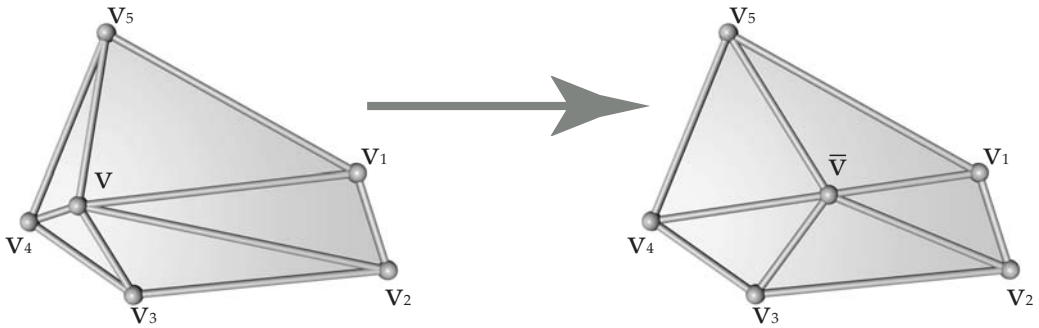


Abbildung 5: Laplace-Algorithmus

Die Vorteile des Verfahrens sind neben der effizienten Ausführung von $\mathcal{O}(n)$, mit n_v der Anzahl der Nachbarn, vor allem darin zu sehen, dass das Polygonnetz in seiner Struktur nicht verändert wird. Es werden also keine Kanten vertauscht oder Vertices zu dem Mesh hinzugefügt, so dass die Topologie unverändert bleibt. Der Speicherverbrauch bleibt unabhängig von der Netzgröße linear, da nur die direkten Nachbarn des Vertex lokal betrachtet werden müssen und die globale Struktur des Polygonnetzes nicht berücksichtigt werden muss. Das Verfahren kann mehrfach hintereinander ausgeführt werden um die Glattheit zu erhöhen.

Von Nachteil ist allerdings, dass bei mehrfachem Durchlauf des Algorithmus eine Verkleinerung des Polygonnetzes stattfindet und das bei unendlich vielen Iterationsschritten, das Objekt zu einem Punkt konvergiert [Zhou 2000]. Es gibt allerdings viele Verbesserungen des ursprünglichen Verfahrens, die deutlich vorteilhaftere Gewichtungsfunktionen implementieren und auch diesen Nachteil ausgleichen.

2.4 Vereinfachung des Polygonnetzes

Bei den Polygonreduktionsverfahren wird die Anzahl der Polygone in einem Mesh um eine bestimmte vorgegebene Größe verringert. Es wird dabei iterativ vorgegangen, so dass die Anzahl der Durchläufe letztendlich die Größe bestimmt. Es soll also ein Polygonnetz erzeugt werden, welches möglichst wenig Abweichung vom Ausgangsnetz hat, aber weniger Polygone besitzt. Diese Art der Kompression soll sowohl bei grafischen

Applikationen den Rechenaufwand verringern, als auch das Datenvolumen senken.

Seit Beginn der neunziger Jahre wurden dafür sehr viele verschiedene Algorithmen entwickelt, welche sich laut Campagna grob in drei Gruppen kategorisieren lassen [Campagna 1998]. Es handelt sich dabei um die *Wavelet*-Methoden, die *Retiling* und die *Clustering*-Techniken. Bei der ersten Gruppe von Verfahren dienen die von Lounsberry entwickelten *Wavelet*-Theorien als Grundlage der Reduktionsmethode [Lounsberry 1994]. Bei dem *Retiling*-Verfahren wird das Modell neu abgetastet und es werden zu dem bestehenden Polygonnetz weitere Vertices, besonders in Regionen mit starker Krümmung, integriert. Danach werden, bis das erwünschte Ergebnis erreicht wird, sukzessiv Vertices des originalen Netzes entfernt [Campagna 1998]. Bei der *Clustering*-Technik wird das Ausgangsmesh in homogene Regionen eingeteilt. Die Vertices innerhalb eines solchen Clusters werden daraufhin zusammengefasst [Rossignac 1993].

Es gibt unzählige Methoden für jedes dieser Verfahren, die an dieser Stelle jedoch nicht alle vorgestellt werden können. Es wird in dieser Arbeit ein iteratives Verfahren, welches von Michael Garland entwickelt wurde, eingesetzt und das in Abschnitt 2.4.2 ausführlich besprochen wird.

2.4.1 Geometrische Verfahren

Betrachtet man die Verfahren nach der Art des geometrischen und topologischen Eingriffs, so muss man drei wesentlich verschiedene Methoden des Eingriffs differenzieren. Diese beziehen sich auf die geometrischen Grundobjekte des Polygonnetzes und man kann zwischen der Vertexdezimierung, der Kantenkontraktion und der Flächendezimierung unterscheiden. Diese Operationen sind die Grundlage der bereits angesprochenen iterativen Verfahren.

Bei der Vertexdezimierung werden einzelne Vertices ausgewählt um sie zu entfernen. Es wird bei der Wahl der zu löschenden Vertices erst die Topologie analysiert um sie dann an den Reduktionskriterien zu prüfen [Schroeder 1992]. Alle Flächen, zu denen der entfernte Vertex gehörte,

werden ebenfalls aus dem Mesh entfernt. Es entsteht nun ein Loch im Polygonnetz, dass durch lokale Triangulation wieder geschlossen werden muss. Bei eckigen Oberflächen liefert dieser Ansatz naturgemäß bessere Ergebnisse als bei weichen Übergängen.

Bei der Kantenkontraktion kann man zwei Verfahren unterscheiden [Campagna 1998]. Die reguläre Kantenkontraktion kann beim verschmelzen eine beliebig neue Position für den zusammengezogenen Vertex setzen. Diese neue Position kann dann über verschieden Qualitätskriterien ermittelt werden. Bei dem *Half-Edge-Collapse* (Abbildung 6) wird dagegen ein Vertex in einen zweiten $e(v_i, v_j) \rightarrow \bar{v}$, über eine Kante verbundenen Vertex, geschoben. Dabei degenerieren die anliegenden Polygone und müssen nach der Kontraktion entfernt werden. Es gibt aber auch die Möglichkeit zwei Vertices zu verschmelzen ohne das diese über eine Kante miteinander verbunden sind. So können etwaige Löcher in einem Mesh geschlossen werden.

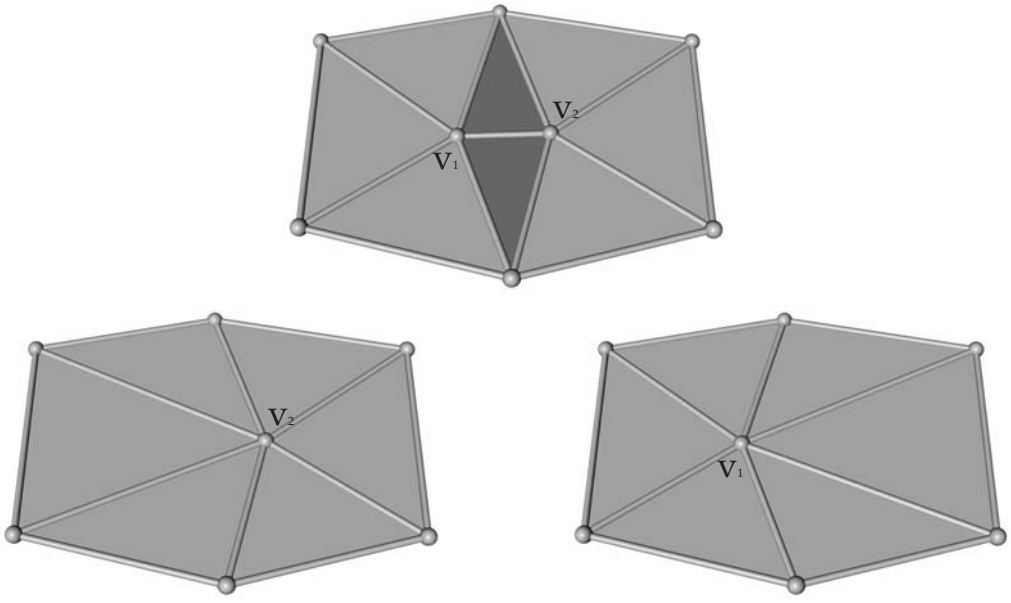


Abbildung 6: (oben) Ausgangsmesh mit der ausgewählten Kante $e(v_1, v_2)$ für den *Half-Edge-Collapse*, (links) $e_h(v_1, v_2) \rightarrow v_2$, (rechts) $e_h(v_2, v_1) \rightarrow v_1$

Bei der Flächendezimierung, die eine Erweiterung der Kantenkontraktion darstellt, wird ein Polygon in einem Vertex verschmolzen. Dabei wer-

den alle Punkte des Polygons zusammengezogen und es wird eine neue Position für diesen Vertex ermittelt.

2.4.2 Quadric-Based-Simplifikation

Bei dem von Garland entwickelten iterativen Verfahren der Kantenkontraktion werden zur Simplifizierung zwei Vertices in einer dritten Position zusammengeführt. Dabei wird der Fehler der Kontraktion Q aus der Summe der quadrierten Abstände der neuen Vertexposition zu den anliegenden Kanten-Polygonen berechnet [Garland 1999]. Es wird bei der Kontraktion der Vertices (v_i, v_j) zuerst eine neue Position $\bar{v}$ ermittelt, wodurch sich die Geometrie des Netzes verändert. Danach müssen die degenerierten adjazenten Polygone entfernt werden, weshalb sich die Zusammenhänge im Mesh ändern.

Die Kostenfunktion bewertet jedes mögliche Vertexpaar des Netzes M, so dass in jedem Iterationsschritt das Paar mit den geringsten Kosten entfernt wird. Der Abstand einer Fläche, die über $n^T v + d = 0$ mit der Normalen $n = [a\ b\ c]^T$ beschrieben wird, zu einem Vertex $v = [x\ y\ z]^T$ wird in der Gleichung

$$D^2(v) = (n^T v + d)^2 = (ax + by + cz + d)^2$$

beschrieben. Es gilt also für jeden Vertex v, dass die Summe der Fehler zu allen Polygonen, die Kosten E wiedergibt.

$$E(v) = \sum_i D_i^2(v) = \sum_i (n_i^T v + d_i)^2$$

Eine effizientere Abschätzung dieses Abstandes erreicht Garland indem er die Gleichung in eine neue Form bringt:

$$\begin{aligned} D^2(v_i) &= (n^T v + d)^2 \\ &= (v^T n + d)(n^T v + d) \\ &= (v^T n n^T v + 2dn^T v + d^2) \\ &= v^T (nn^T) v + 2(dn)^T v + d^2 \end{aligned}$$

Dabei entspricht nn^T folgender Matrix:

$$A = nn^T = \begin{bmatrix} a^2 & ab & ac \\ ab & b^2 & bc \\ ac & bc & c^2 \end{bmatrix}$$

Garland definiert das *Quadric* Q als Tripel $Q = (A, b, c)$ mit der Matrix A, einem Vektor b und einem Skalar c, der jedem Vertex zugeordnet wird und der Gleichung $Q(v) = v^T Av + 2b^T v + c$ entspricht. Eine alternative Repräsentation verwendet vierdimensionale Vektoren, wodurch die Gleichung kompakter dargestellt werden kann.

Um eine ideale Position für den neuen Vertex $\bar{v}$ einer Kante (v_i, v_j) zu finden, kann ein dreistufiges Verfahren angewendet werden. Zuerst versucht man die allgemein optimale Position $\bar{v} = -A^{-1}b$ mit dem Fehler von $Q(\bar{v}) = b^T\bar{v} + c$ zu finden. Wenn es diese eine ideale Position nicht gibt, dann kann auf der Kante $e = (v_i, v_j)$ nach der besten Position gesucht werden. Wenn auch hier kein eindeutiges Ergebnis gefunden werden kann, dann wird der bessere Wert von v_i und v_j ausgewählt.

Dies alleine führt aber noch nicht zu brauchbaren Ergebnissen, da nicht geprüft wird ob das Mesh nach der Kontraktion degeneriert. Es muss mindestens geprüft werden, ob die benachbarten Polygone invertieren oder ob es zu Überlappungen kommt (Abbildung 7). Dafür müssen Tests implementiert werden, die bei einer negativen Bewertung die Kontraktion verhindern.

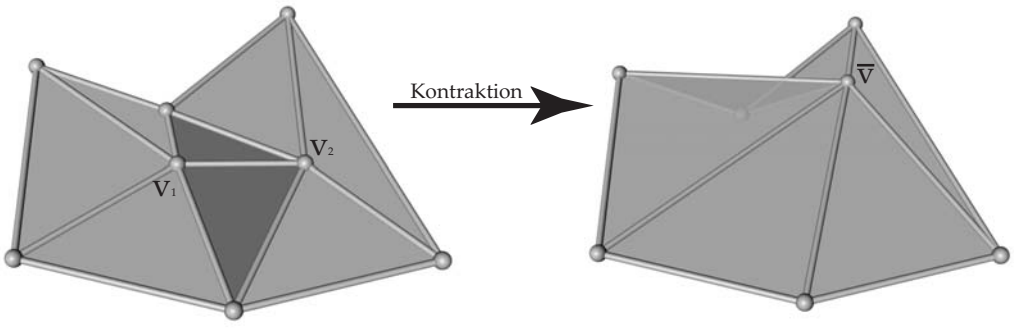


Abbildung 7: (links) Mesh mit der Kante $e(v_i, v_j)$ vor der Kantenkontraktion, (rechts) danach mit neuem Vertex $\bar{v}$ und überdeckter Kante

Ein einfacher und bewährter Ansatz um die Invertierung von Polygonen zu vermeiden besteht darin, dass man die Normalen n_{f_i} der angrenzenden Polygone vor und nach dem Zusammenschluss vergleicht. Wenn die Normale über einem vorgegebenen Schwellenwert liegt und dadurch ein Wechsel der Orientierung vorliegt, wird die Kontraktion nicht ausgeführt.

Der Algorithmus besteht aus vier Stufen, die durchlaufen werden:

1. Es wird ein Kantenpaar $e(v_i, v_j)$ ausgesucht.
2. Für jedes mögliche Paar werden die Kosten Q berechnet.
3. Diese Paare werden nach der Höhe dieser Kosten sortiert.
4. Folgende Schritte werden so lange wiederholt, bis das erwünschte Ergebnis erreicht ist.
   (a) Kontrahiere das Kantenpaar $e(v_i, v_j) \rightarrow \bar{v}$ mit den niedrigsten Kosten Q.
   (b) Berechne die Kosten neu $Q = Q_i + Q_j$ für alle beteiligten Nachbarn.

Das Verfahren ermöglicht es ohne großen Aufwand andere zusätzliche Gewichtungsfunktionen hinzuzurechnen. So könnte man die Länge der Kanten bei der Kontraktion berücksichtigen oder den Winkel der Normalen der angrenzenden Dreiecke hinzunehmen. Wegen dieser guten Anpassungsfähigkeit und der effizienten Ausführung ist das *Quadric-Based*-Verfahren so beliebt und wurde auch für dieses Projekt ausgewählt.

3 Datenerfassung

3.1 Fotoaufnahmen

Abbildung 8: Rundumaufnahme einer Löwenskulptur

Die Fotos der Objekte wurden mit einer Kamera auf einem Stativ vor schwarzem Hintergrund aufgenommen. Auf einem gut ausgeleuchteten Drehteller wurde das Motiv dann stufenweise gedreht. Die Bilder müssen mit einer guten Qualität bei gleichbleibenden Aufnahmebedingungen aufgenommen werden um die Rekonstruktion zu verbessern.

Je nach Formkomplexität des Objektes werden dann entsprechend viele Aufnahmen gemacht. Die Aufnahmen werden dann für die weitere Bearbeitung im JPG[2]-Format gesichert.

In der Arbeit wurden 36 Aufnahmen von dem Objekt (Abbildung 8) angefertigt, in einer Auflösung von 21 Megapixel bei 5616×3744 Bildpunkten. Da bei den Aufnahmen neben dem schwarzen Hintergrund noch der Raum an den Seiten zu sehen war, mussten die Bilder manuell nachbearbeitet werden (Abbildung 9). Dafür wurde eine Schnittmaske benutzt, um den Hintergrund vollkommen schwarz zu färben.

Abbildung 9: Schnittmaske der Löwenskulptur

Bei der Rekonstruktion bildeten sich Artefakte an den Rändern des Objektes, die durch die kaum sichtbaren Farbübergänge erzeugt wurden. Die JPEG-Kompression führte an der Rändern zu diesen Überschwingern, die aber in einem späteren Schritt (Abschnitt 5) herausgefiltert wurden. Deshalb wäre es sinnvoller ein Rohdatenformat für die Bildspeicherung zu benutzen.

[2]Joint Photographic Experts Group

3.2 Drehtellersteuerung

Um eine 360° Aufnahme zu ermöglichen wurde die Kamera auf einem Stativ befestigt und das Objekt wurde gedreht. Dafür wurde ein einachsiger Drehteller der Firma Kaidan benutzt, der um die vertikale Achse rotiert. Da die mitgelieferte Software nicht den Ansprüchen genügte, musste eine eigene Bibliothek für die Steuerung entwickelt werden.

Abbildung 10: Drehteller der Firma Kaidan (Quelle:www.kaidan.com)

Die Steuereinheit des Drehtellers kann über die serielle Schnittstelle angeschlossen werden und kommuniziert über RS-422[3]. Die Kommandos werden dabei im ASCII-Format[4] übertragen. Diese dürfen maximal 15 Zeichen beinhalten und sind *case insensitiv*. Der genaue Aufbau einer Kommadozeile ist in EBNF-Form im Quelltext 4 dargestellt und der Programmablaufplan wird in der Abbildung 11 verdeutlicht.

Bei einer Kommandozeile wird zuerst die Achse angegeben, gefolgt von einem Befehl für diese und einem möglichen Funktionswert. Jeder Schritt wird über einen *linefeed*[5] abgeschlossen und von der Steuereinheit über ein Echo oder ein Ergebniswert bestätigt. Eine einfache Kommandosequenz über ein Terminal könnte also folgendermaßen aussehen:

```
X+1000<LF>XZ<LF>
```

Die Ausgabe der Gegenstelle würde daraufhin folgende Form besitzen:

```
X+1000<LF>XZ1000<LF>
```

[3] Schnittstellen-Standard
[4] American Standard Code for Information Interchange
[5] Zeilenvorschub

```
NUM     ::= "0" | "1" | "2" | "3" | "4" | "5" |
            "6" | "7" | "8" | "9" ;
NUMB    ::= ["+"|"-"] NUM*
CHAR    ::= "A" | "B" | "C" | "D" | ... | "Z" |
            "a" | "b" | ... | "z" ;
SPACE   ::= " "
SEP     ::= SPACE | ",";
END     ::= "\n" | "^J" | "^Enter";
COMMAND ::= CHAR [SPACE] (NUMB|CHAR)
                         [SEP (NUMB|CHAR)] [END];
```

Quelltext 4: EBNF der Kommadozeile

Der Drehteller wird also an Position 1000 gefahren und danach wird die aktuelle Position von der Steuereinheit abgefragt und mit 1000 beantwortet.

3.2.1 TurnTable-Bibliothek

Um die Steuerung des Drehtellers betriebssystemunabhängig zu gestalten wurde die Kommunikationsschnittstelle mit der quelloffenen *Boost*-Bibliothek umgesetzt. Diese setzt sich aus einer Sammlung von Bibliotheken zusammen, in der unter anderem auch eine Schnittstelle für die Kommunikation über den seriellen Port vorhanden ist. Diese *Boost.Asio*[6] Bibliothek abstrahiert die Betriebssystemschnittstellen zur Ein- und Ausgabe und bietet auch eigene *Timer*-Funktionen an.

Bei der Erzeugung der Klasse der neu entwickelten *TurnTable*-Bibliothek wird zuerst die Kommunikationsschnittstelle initialisiert. Dabei wird die Baudrate[7], die Anzahl der Datenbits und die Parität gesetzt. Erst danach kann die Steuereinheit des Drehtellers initialisiert werden, wo die Auflösung, die Geschwindigkeit und andere Parameter gesetzt werden können.

Da die Bibliothek so angelegt wurde, dass sie auch in anderen Projekten weiter benutzt werden kann, wurden verschiedene Möglichkeiten

[6] Asynchrone Ein- und Ausgabe Bibliothek
[7] Einheit für die Symbolübertragungsrate

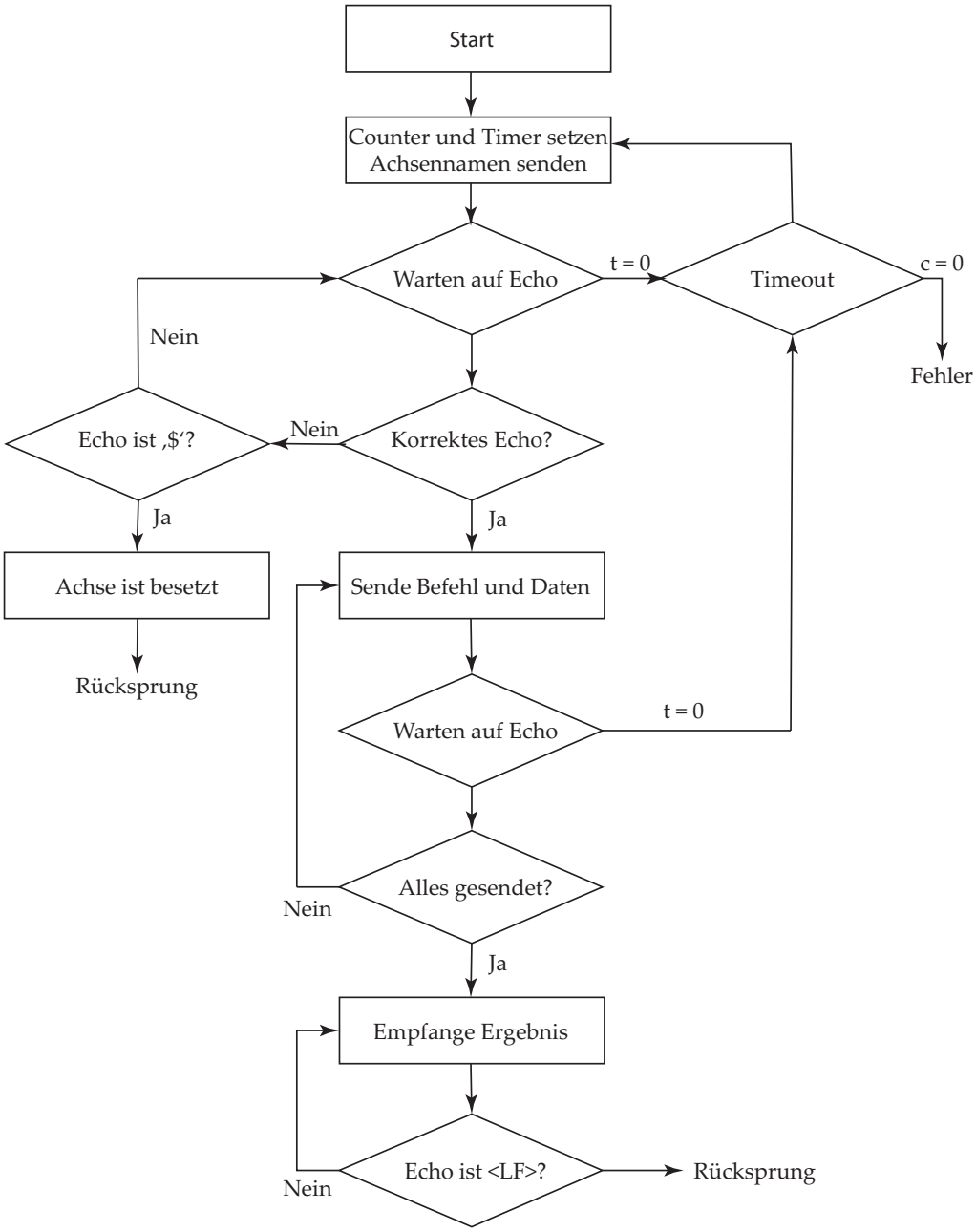


Abbildung 11: Programmablaufplan für die Drehtellersteuerung

zur Steuerung implementiert. So kann eine absolute Position angesteuert werden, eine relative Position, oder man kann einen Winkel angeben der angefahren wird. Man kann auch den Teller konstant drehen lassen, bis

man ihn manuell anhält. Außerdem ist es möglich den aktuellen Status und die aktuelle Position auszulesen.

Der Aufbau einer Kommandoschleife ist in Schaubild 11 dargestellt und soll hier in dem vereinfachten Quelltext 6 aufgezeigt werden. Dabei wird zuerst der Achsenname so oft gesendet und auf die Antwort gewartet bis der *Timeout* den Abbruch erzwingt. Sollte eine korrekte Antwort eintreffen, wird der Rest der Befehlszeile gesendet. Jedes Zeichen, welches gesendet wird muss eine Antwort von der Drehtellersteuerung bekommen. Das Echo wird, ebenso wie das Kommando, mit dem *Linefeed*-Zeichen abgeschlossen.

```
TurnTable tt("COM1");
if (tt.isInitialised()) {
  tt.Setup(KAIDAN_RESOLUTION, 200, 500, 5, 500,
     KAIDAN_SRHALF);
  tt.MoveRelative(1000, true);
  std::cout << tt.ReadPosition() << std::endl;
}
```

Quelltext 5: Einfaches Beispielprogramm

Diese Methode kann direkt benutzt werden um Befehle zu versenden oder man nimmt eine fertige Methode zur Steuerung, wie z. B. die *MoveRalative*-Funktion (Quelltext 7), mit der man sich um eine bestimmte vorgegebenen Anzahl an Schritten weiterbewegen kann. Diese ruft dann *SendCommand* (Quelltext 6) mit den nötigen Parametern auf. Damit kann dann ein einfaches Steuerprogramm (Quelltext 5) für den Drehteller implementiert werden.

```
int SendCommand(char *pWriteDataBuffer, char
   *pReadDataBuffer, unsigned int
   nReadBufferSize, unsigned int nBytesToWrite) {
  while (pReadDataBuffer[0] !=
     pWriteDataBuffer[0]) {
    if (nCounter >= KAIDAN_TIMEOUT_COUNT) {
      SendChar('\n');
      return 0;
    } else nCounter++;
    if (SendChar(*pWriteDataBuffer) == 1)
      ReceiveChar(pReadDataBuffer);
  }
  nBytesReceived++;
  for (int i=1; i < nBytesToWrite; i++) {
    if (SendChar(pWriteDataBuffer[i]) == 1)
      if (int n =
         ReceiveChar(&pReadDataBuffer[i])) {
        nBytesReceived += n;
      } else {
        SendChar('\n');
        return 0;
      }
  }
  while (pReadDataBuffer[nBytesReceived-1] !=
     '\n' && nBytesReceived < nReadBufferSize)
    nBytesReceived +=
       ReceiveChar(&pReadDataBuffer[nBytesReceived]);
  return nBytesReceived;
}
```

Quelltext 6: Methode zum Senden eines Kommandos

```
bool MoveRelative(int nSteps) {
  sprintf_s(chCommand, "X%s%i\n",
     nSteps<0?"":"+", nSteps);
  if (SendCommand(chCommand, strlen(chCommand))
     == 0) return false;
  return true;
}
```

Quelltext 7: Methode für die relative Bewegung

4 Datenverarbeitung

4.1 OpenMesh

Die *OpenMesh*-Bibliothek implementiert eine, sowohl in der Zeit als auch im Speicherverbrauch, effiziente *Half-Edge*-Datenstruktur zur Berabeitung von Polygonnetzen in C++. Die Bibliothek wurde an der RWTH Aachen[8] von der *Computer Graphics Group* entwickelt.

In der Dokumentation wird sie als flexibel, effizient und einfach zu benutzen beschrieben [OpenMesh 2012]. Sie kann an eine Vielzahl von Algorithmen leicht angepasst werden und wurde deshalb auch für dieses Projekt eingesetzt.

Die *OpenMesh*-Bibliothek kann die benötigten Dreiecksnetze sehr effizient [Botsch 2002] abbilden und hat die Möglichkeit gängige Dateiformate einzulesen und zu speichern. Die Daten werden in Arrays und Listen vom *OpenMesh-Kernel* dynamisch verwaltet. Auf alle benötigten elementaren Daten kann einfach zugegriffen werden und die *Half-Edge*-Datenstruktur ermöglicht einen schnellen Verweis auf die Nachbarelemente. Es gibt für jede elementare Objektklasse einen iterativen Zugriff auf alle Daten. Daneben können eigene Attribute definiert und den elementaren Objekten zugeordnet werden. Einfache geometrische und topologische Operationen sind ein Bestandteil der Bibliothek und lassen sich leicht erweitern.

Eine einfache Implementierung der Fehlerquadrics ist auch ein Teil der Bibliothek, die jedoch nicht alle Vorteile des *Quadric-Based*-Algorithmus verwirklicht und deshalb neu umgesetzt werden musste.

[8] Rheinisch-Westfaelische Technische Hochschule Aachen

4.2 Quadric-Based-Simplifikation Implementierung

Die Implementierung der *MeshReduction*-Bibliothek ist der Schwerpunkt dieser Arbeit. Es wird im folgenden nur auf den wichtigsten Teil des Quelltextes eingegangen, der hier auf die wesentliche Funktionalität reduziert wird um ihn besser erklären zu können.

Die *Half-Edge*-Struktur wurde schon in Abschnitt 2.1.3 beschrieben und diese wird durch die *OpenMesh*-Bibliothek zur Verfügung gestellt. Funktionen, die auf elementare Daten zugreifen oder über diese navigieren, sind ebenfalls dort implementiert und werden hier nicht weiter beschrieben.

Der wichtigste Teil des Algorithmus ist die Behandlung der Fehlermetrik. Die *OpenMesh*-Bibliothek besitzt eine Template-Klasse *QuadricT*, die jedoch angepasst werden musste. Ein Quadric wird repräsentiert in der homogenen Matrixform und die Werte können *float* oder *double* annehmen. Die Matrix (Quelltext 8) ist symmetrisch, so dass nur zehn Werte für die Bearbeitung gespeichert werden müssen.

```
template <class Scalar>
class QuadricT {
  Scalar a_, b_, c_, d_,
             e_, f_, g_,
                 h_, i_,
                     j_;
}
typedef QuadricT<float> Quadricf;
typedef QuadricT<double> Quadricd;
```

Quelltext 8: Quadric Klasse

Um einen Quadric zu berechnen, besitzt die Klasse eine *evaluate*-Methode, die $Q(v) = v^T Av + 2b^T v + c$, wie in Abschnitt 2.4.2 dargelegt, auswertet. Es musste noch die Methode *optimize* (Quelltext 9) implementiert werden um eine optimale Fehlerposition $\bar{v}$, falls diese existiert, berechnen zu können.

Bei der Initialisierung der *MeshReduction*-Klasse werden alle Attribute der Objekte zurückgesetzt und es wird die Methode *collect_quadrics*

```
bool optimize(Vec3d& v, Vec3d& v1, Vec3d& v2) {
  Vec3d d = v1 - v2;
  Matrix3 A = tensor();

  Vec3d Av2 = A*v2;
  Vec3d Ad = A*d;
  double denon = 2*(d|Ad);
  if (fabs(denon) < 1e-12) return false;
  double a = (-2 * (vector()|d) - (d|Av2) -
     (v2|Ad)) / (2*(d|Ad));
  if (a < 0.0) a=0.0;
  if (a > 1.0) a=1.0;
  v = a*d + v2;
  return true;
}
```

Quelltext 9: Quadric Methode *optimize*

(Quelltext 10) aufgerufen um für alle Polygone die Quadrics initial zu erzeugen. Diese werden dann für jedes Vertex berechnet und zusammengefasst um sie dann mit der Dreiecksfläche zu gewichten. Danach werden sie zu jedem Vertex des Polygons aufaddiert.

Der eigentliche Reduktionsdurchlauf besteht im Wesentlichen aus zwei fundamentalen Schritten. Zuerst werden alle erlaubten Vertexkombinationen berechnet um sie auf einem *Heap* zu sortieren Danach wird der oberste Eintrag des Heaps sukzessiv bis zu einem vorgegebenen Abbruchkriterium entfernt. Die *decimate*-Funktion durchläuft also zuerst alle Vertices um daraufhin die *compute_target_placement*-Methode (Quelltext 11) für die Berechnung der besten Position für alle ausgehenden *Half-Edges* zu setzen. Es wir also die ideale neue Vertexposition $\bar{v}$ für jede mögliche Kombination, durch Aufaddieren der Quadrics, errechnet und die beste dann ausgewählt. Dafür stehen verschiedene Verfahren zur Verfügung. Die einfachste Variante nimmt den Endvertex einer Halbkante als neue Vertexposition an. Eine andere Möglichkeit ist es den Mittelwert auf der Kante zu ermitteln um diesen dann auszuwählen. Es kann aber auch wie in Abschnitt 2.4.2 beschrieben nach einem optimalen Wert ge-

```
void collect_quadrics() {
  for (FaceIter f_it = faces_begin(); f_it !=
     faces_end(); ++f_it) {
    Vec3d n = (v2-v1) % (v3-v1);
    double farea = n.norm() * 0.5;
    n.normalize_cond();

    Quadricd q(n[0], n[1], n[2], -(point(vh1) |
       n));
    q *= farea;

    property(quadrics, vh1) += q;
    property(quadrics, vh2) += q;
    property(quadrics, vh3) += q;
  }
}
```

Quelltext 10: Quadric Methode *collect_quadrics*

sucht werden, der im Idealfall entweder frei im Raum zu finden ist oder aber auf der Kante liegt.

Im zweiten Schritt wird der *Heap* abgearbeitet und es wird getestet, ob das Zusammenlegen der Vertices überhaupt möglich ist oder ob Fehler im *Mesh* erzeugt werden. So wird beispielsweise in der Funktion *check_faceflip* (Quelltext 12) getestet, ob nach dem Zusammenschluss alle um einen Vertex anliegenden Polygone die gleiche Orientierung haben. Dafür werden die Normalen der adjazenten Polygone vor und nach der Vereinigung berechnet und miteinander verglichen. Wenn alle Kriterien erfüllt sind, kann die *collapse*-Funktion aufgerufen werden und die Kante mit den dazugehörigen Polygonen wird entfernt.

```
void compute_target_placement(VertexHandle vh) {
  double best_error = DBL_MAX;
  HalfedgeHandle best_collapse_target;
     best_collapse_target.reset();
  Vec3d best_collapse_vertex_position;
  for (VertexOHalfedgeIter vhiter(m_mesh, vh);
     vhiter; ++vhiter) {
    Quadricd q = property(quadrics, vh);
    q += property(quadrics,
       to_vertex_handle(vhiter));

    double error = findBestPosition(vhiter,
       new_position);

    if (error < best_error) {
      best_collapse_target = vhiter;
      best_collapse_vertex_position =
        new_position;
      best_error = error;
    }
  }
}
```

Quelltext 11: Reduktionsmethode *compute_target_placement*

```
float check_faceflip(HalfedgeHandle heh, Vec3d
   vnew) {
  float delta = 1.0;
  for (VertexFaceIter vfiter = vf_begin; vfiter
     != vf_end; ++vfiter) {
    Vec3f nb = calc_face_normal(vfiter.handle());
    FaceVertexIter fviter =
       fv_iter(vfiter.handle());
    Vec3f p0 = point(fviter.handle()); ++fviter;
    Vec3f p1 = point(fviter.handle()); ++fviter;
    Vec3f p2 = point(fviter.handle());
    if (p0 == point(from_vertex_handle(heh))) p0
       = vnew;
      else if (p1 ==
         point(from_vertex_handle(heh))) p1 =
         vnew;
        else if (p2 ==
           point(from_vertex_handle(heh))) p2 =
           vnew;
    Vec3f p1p0 = p0;  p1p0 -= p1;
    Vec3f p1p2 = p2;  p1p2 -= p1;
    Vec3f na = cross(p1p2, p1p0);
    float norm = na.length();
    if (norm != 0.0) na *= (1.0/norm);
      else na[0]=na[1]=na[2]=0;
    if ((nb | na) < delta) delta = nb | na;
  }
  return delta;
}
```

Quelltext 12: Reduktionsmethode *check_ faceflip*

4.3 Entfernung von Artefakten

Ein Problem waren die in Abschnitt 3.1 besprochenen Artefakte an den Rändern des Objektes. Diese bildeten nach der Rekonstruktion eigene kleine Objekte an der Oberfläche, die herausgefiltert werden mussten. Dafür wurde ein Tiefensuchverfahren (Quelltext 13) umgesetzt, welches rekursiv aufgerufen wird und alle zusammenhängenden Objekte numerisch markiert. Zuerst werden alle Markierungen zurückgesetzt um dann alle Verbindungen rekursiv zu durchlaufen. Ist ein Objekt abgedeckt, so wird das nächste durchlaufen und der Index wird inkrementiert. Wenn eine Objektgruppe kleiner als ein vorgegebener Schwellenwert ist, dann wird im zweiten Schritt das Objekt aus dem Mesh entfernt. Dabei wird zuerst die Größe des Subobjektes bestimmt, und anschließend werden alle Vertices dieses Objektes entfernt.

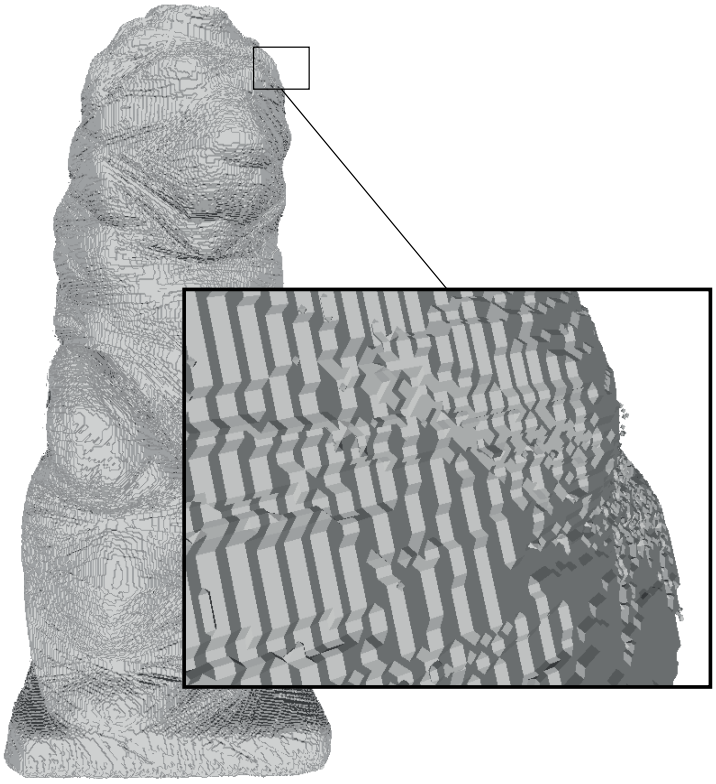


Abbildung 12: Artefakte über der Oberfläche

```
void RemoveSmallObjects(int nMinSize) {
  int nObjectID = 0;
  for (VertexIter viter = vertices_begin();
     viter != vertices_end(); ++viter) {
    if (status(viter).tagged()) {
      nObjectID++;
      RemoveSmallObjectsDFS(viter, nObjectID);
    }
  }
  for (int i = 1; i<=nObjectID; i++) {
    int nObjSize = 0;
    for (VertexIter viter = vertices_begin();
       viter != vertices_end(); ++viter)
      if (property(ObjectID, viter) == i)
        nObjSize++;
    if (nObjSize < nMinSize)
      for (VertexIter viter = vertices_begin();
         viter != vertices_end(); ++viter)
        if (property(ObjectID, viter) == i)
          delete_vertex(viter);
  }
}

void RemoveSmallObjectsDFS(VertexHandle v, int
   nObjectID) {
  if (!status(v).tagged()) {
    status(v).set_tagged(true);
    property(ObjectID, v) = nObjectID;
    for (vhiter(m_mesh, v); vhiter; ++vhiter) {
      RemoveSmallObjectsDFS(to_vertex_handle(vhiter),
         nObjectID);
    }
  }
}
```

Quelltext 13: Algorithmus zum Auffinden von kleinen Objekten

4.4 Oberflächenglättung

Durch die *Silhouetten-Schnitt*-Technik und die Voxelbeschreibung ist das Polygonnetzes nach dem *Marching-Cubes*-Verfahren sehr kantig. Um dieses eckige Modell zu verbessern und glattere Formen zu bekommen, die dem ursprüngliche Objekt näher kommen, musste eine Glättung vor der Reduktion angewendet werden. Dafür stellt die *OpenMesh*-Bibliothek eine eigene *Smoothing*-Klasse zur Verfügung, die auch die *Laplace-Smoothing*-Technik enthält. Die Glättung kann dann in mehreren Durchläufen angewendet werden ohne die Topologie zu verändern. Dadurch bleibt die Silhouette unverändert und nur die Übergänge werden weicher (Abbildung 13).

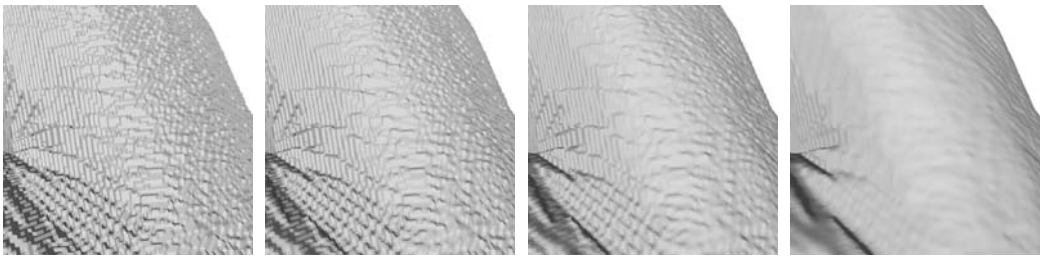


Abbildung 13: Polygonnetz im orginal, nach einfachem, nach dreifachem und nach siebenfachem Durchlauf des *Laplace-Smoothing*-Verfahrens

5 Resultate

5.1 Laufzeit und Speicherverbrauch

Differenziert man, wie im vorherigen Kapitel, den Algorithmus in zwei Phasen, so ist die erste Phase die Initialisierung und die zweite der eigentliche Dezimierungsprozess. Es ist in der ersten Phase mit einer Laufzeit von $\mathcal{O}(n)$ für das Auswählen geeigneter Paare und $\mathcal{O}(n)$ für die Berechnung der initialen Quadrics zu rechnen. Daraufhin müssen die Datenpaare auf den Heap gesetzt werden, was zu einer Laufzeit von $\mathcal{O}(n\ log\ n)$ führt. Insgesamt kommt man also zu einer Laufzeit von $\mathcal{O}(n\ log\ n)$ für den gesamten Initialisierungsvorgang [Garland 1999].

Der zweite Teil des Algorithmus, der das günstigste Paar aussucht und die Kante mit den angrenzenden Polygonen entfernt, hat eine Laufzeit von $\mathcal{O}(log(n - ik))$, wobei die Konstante k die Anzahl der angrenzender Polygone wiedergibt und bei einem mannigfaltigen Mesh stets zwei beträgt [Garland 1999]. Daraus errechnet sich dann eine Gesamtkomplexität von $\mathcal{O}(n\ log\ n - m\ log\ m)$ für den zweiten Teil.

Der Speicherverbrauch ist durch die *Half-Edge*-Struktur vorgegeben und für jedem Vertex wird zusätzlich ein Quadric, eine neue geometrische Position und eine Position im Heap gespeichert. Daraus ergibt sich ein Gesamtwert von $3f + i + 3 * (3f + i)$ für die Datenstruktur und nochmal $10f + 3f + i$ für die zusätzlichen Attribute, mit f als *Float* oder *Double* und i als *Integer*. Außerdem gehören noch die *Pointer* für die einzelnen Listen dazu.

Eine Erhöhung der Polygonanzahl löst also einen linearen Anstieg des Speicherplatzes und einen etwas niedrigeren Anstieg der Arbeitsgeschwindigkeit aus. Die Anzahl der Polygone beeinflusst die Qualität des Objektes, so dass eine Reduktion nicht beliebig ausgeführt werden kann.

5.2 Auswertung

Um zu testen wie gleichmäßig der Algorithmus arbeitet, wurde zu Beginn eine Kugel als Ausgangsobjekt verwendet und in drei Schritten mit der einfachen *collapse*-Funktion reduziert. Es ist zu erkennen wie in den ersten Schritten die Kontraktion gleichmäßig erfolgt. Allerdings ist bei 100 Vertices kaum noch die ursprüngliche Form zu erfassen und es entsteht ein Objekt, das nur noch annähernd als Kugel zu erkennen ist. Die Tabelle 1 gibt die Werte für die einzelnen Reduktionsschritte wieder, wobei der erste Eintrag die ursprünglichen Werte darstellt.

Vertices	Kanten	Polygone	Initialisierung	Reduktion
7566	22692	15128		
3500	10494	6996	0,0425 s	0,2294 s
1000	2994	1996	0,0417 s	0,3241 s
100	294	196	0,0408 s	0,3529 s

Tabelle 1: Reduktion einer Kugel

Um den Algorithmus an komplexeren Objekten zu testen wurden einige Objekte aus dem *Stanford-3D-Scanning*-Projekt[9] ausgewählt. Die Ergebnisse sind in den folgenden Tabellen und Bildern dokumentiert.

Vertices	Kanten	Polygone	Initialisierung	Reduktion
34834	104228	69451		
15000	44871	29868	0,1741 s	1,3434 s
3500	10425	6922	0,1868 s	2,0092 s
350	1039	686	0,1820 s	2,1809 s

Tabelle 2: Reduktion eines Hasenmodells

Da die Reduktionen gute Werte lieferten, wurde der Algorithmus am selbst erstellten Löwen-Modell getestet. Bei der Größe des Objektes viel zum ersten mal auf, dass der Ladevorgang und die Initialisierung der

[9] http://graphics.stanford.edu/data/3Dscanrep/

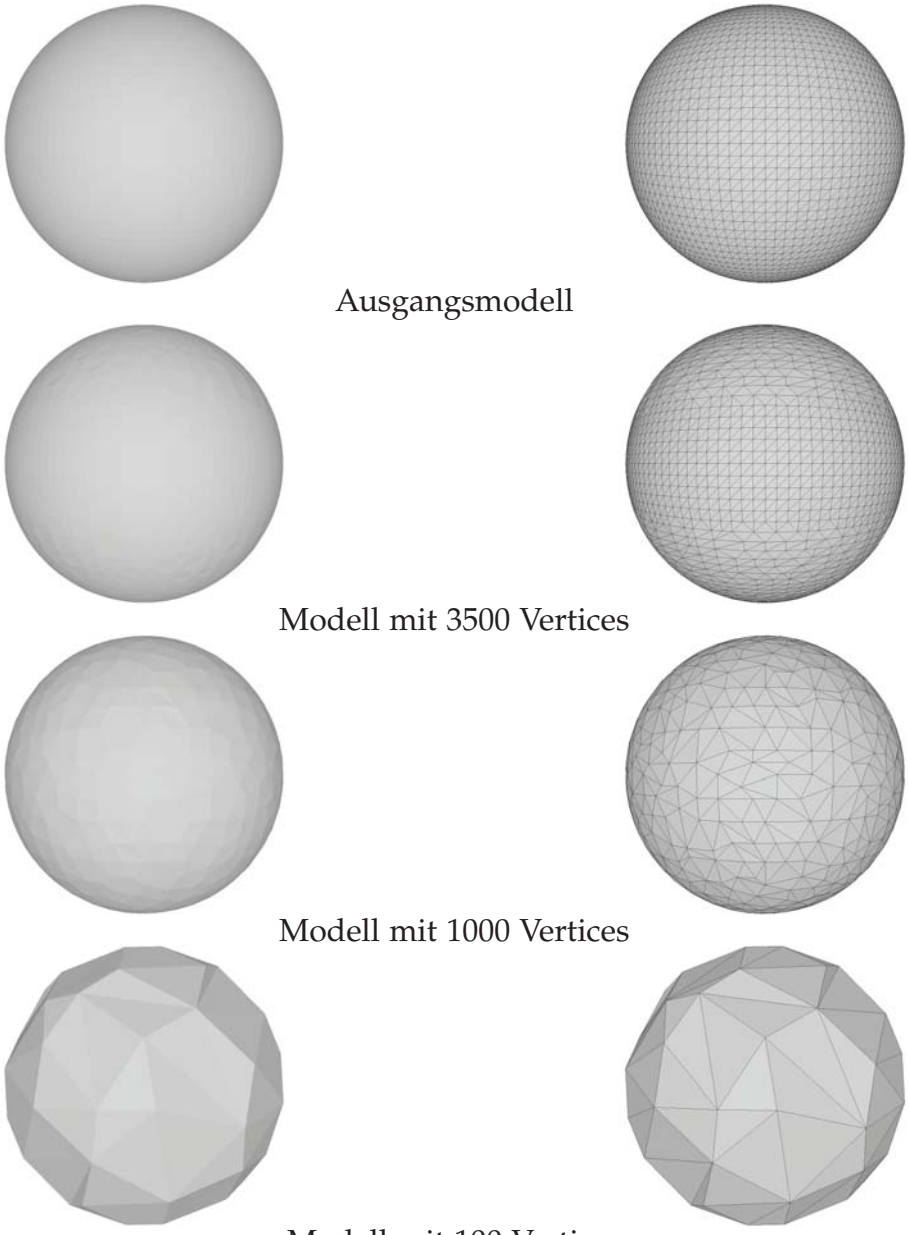
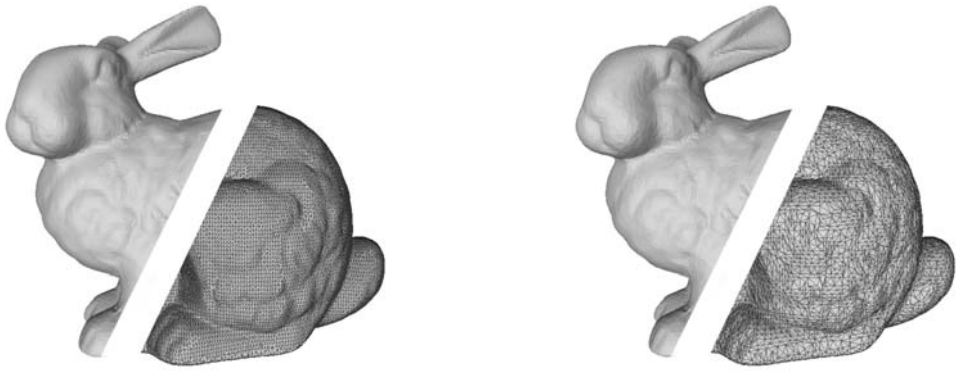
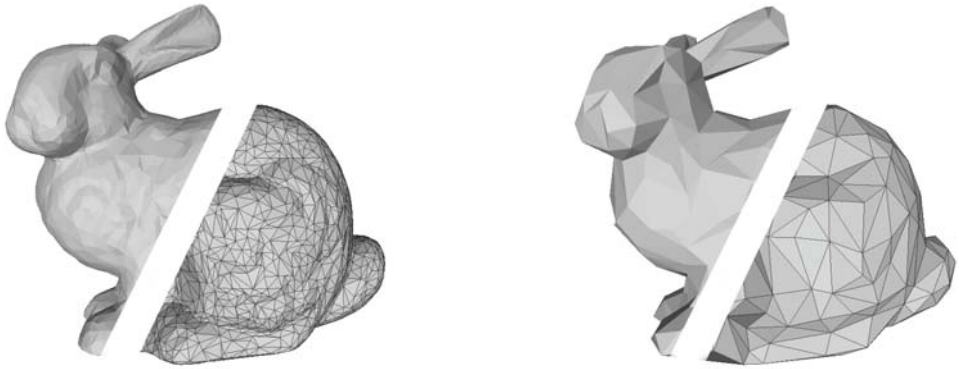


Abbildung 14: Reduktion einer Kugel

Half-Edge-Datenstruktur durch die *OpenMesh*-Bibliothek sehr kostspielig ist und über sechs Minuten gedauert hat.



(links) Ausgangsmodell, (rechts) Modell mit 35000 Vertices



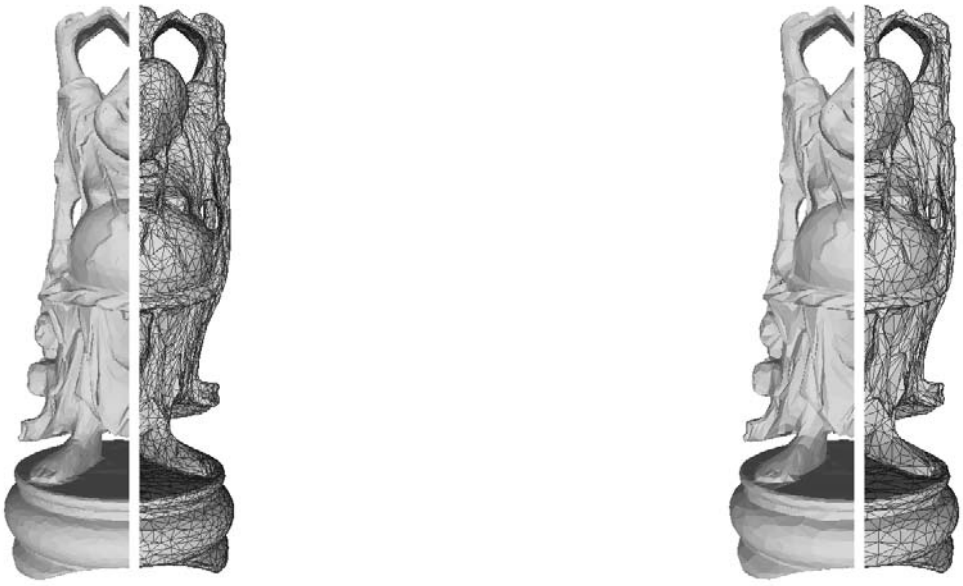
(links) Modell mit 3500 Vertices, (rechts) Modell mit 350 Vertices

Abbildung 15: Reduktion eines Hasenmodells

Vertices	Kanten	Polygone	Initialisierung	Reduktion
543652	1631574	1087716		
50000	150618	100412	2,7073 s	34,7339 s
10000	30618	20412	2,5191 s	37,5813 s
5000	15618	10412	2,4375 s	38,1816 s

Tabelle 3: Reduktion des *Happy-Buddha*-Modells

(links) Ausgangsmodell, (rechts) Modell mit 50000 Vertices



(links) Modell mit 10000 Vertices, (rechts) Modell mit 5000 Vertices

Abbildung 16: Reduktion des *Happy-Buddha*-Modells

Vertices	Kanten	Polygone	Initialisierung	Reduktion
1395662	4189807	2792840		
100000	303709	202404	27,5229 s	89,3747 s
50000	153787	102482	26,8881 s	92,7334 s
5000	18891	12586	33,1114 s	97,7237 s

Tabelle 4: Reduktion des Löwenmodells

5.3 Erweiterung und Verbesserung

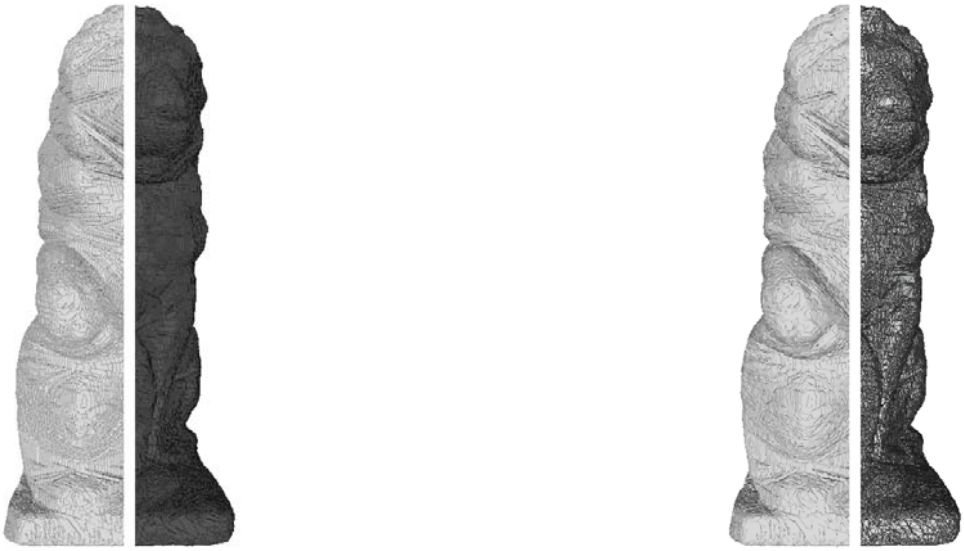
Bei der Reduktion tauchten die in Abschnitt 3.1 beschriebenen Artefakte (Bild 12) auf, die kleine Objekte an der Oberfläche bildeten. Diese wurden in einem weiteren Schritt herausgefiltert. In Abbildung 18 sind diese kleinen Objektartefakte rot markiert und deutlich zu sehen.

Bei der ersten Rekonstruktion stellte sich heraus, dass einige Oberflächenbereiche Aushöhlungen hatten, die hinter der Oberfläche kleine Hohlräume bildeten. Diese sind auch in der Abbildung 18 als dunklere Bereiche gut zu erkennen. Der *Marching-Cubes*-Algorithmus hat nicht wie erwartet gearbeitet. Die Artefakte an der Oberfläche beeinflussten demnach auch die Oberflächenstruktur.

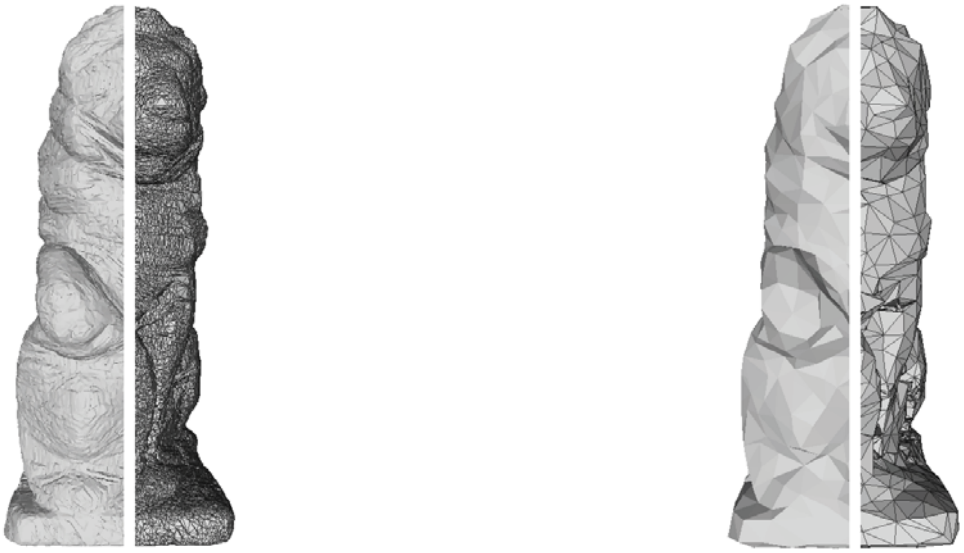
Um die kantige Oberfläche nach der Rekonstruktion zu glätten wurde vor der Reduktion noch ein weiterer Schritt ausgeführt und die *Laplace-Smoothing*-Technik angewandt. In Abbildung 19 ist deutlich zu erkennen wie die Oberfäche weicher wird, wodurch auch die Ergebnisse der Reduktion verbessert werden konnten.

Um die Qualitätsverbesserung zu verdeutlichen, werden in der folgenden Abbildung 20 drei der ausgearbeiteten Verfahren nacheinander aufgezeigt. Die Tabelle 5 gibt die Hausdorff-Distanz, den Mittelwert und das Quadratisches Mittel der verschiedenen Verfahren an. Dabei ist zu erkennen, dass die Hausdorff-Distanz als probates Messverfahren die Qualitätsverbesserung deutlich wiederspiegelt. Die Messwerte wurden hier an der Diagonalen der Bounding-Box ausgerichtet um stabile Vergleichswerte zu erhalten.

Die Tabelle 6 gibt die Messwerte nach der Anzahl der Glättungsdurch-



(links) Ausgangsmodell, (rechts) Modell mit 100000 Vertices



(links) Modell mit 50000 Vertices, (rechts) Modell mit 5000 Vertices

Abbildung 17: Reduktion des Löwenmodells

läufe wieder. Hier ist klar zu erkennen, dass jeder weitere Glättungsschritt zu einer Verschlechterung der Werte führt. Die Laplace-Glättung kann also nur moderat angewendet werden um die Qualität nicht stark zu verschlechtern.

Distanz	Einfach	Mittelwert	Dreistufig
Hausdorff	0,003658	0,003200	0,002906
Mittelwert	0,000213	0,000229	0,000225
Quadratisches Mittel	0,000312	0,000312	0,000313

Tabelle 5: Reduktion des Löwenmodells auf 100000 Vertices nach drei verschiedenen Verfahren

Distanz	ohne	einfach	dreifach	siebenfach
Hausdorff	0,002906	0,003480	0,003512	0,003707
Mittelwert	0,000225	0,000164	0,000295	0,000342
Quadratisches Mittel	0,000313	0,000234	0,000374	0,000428

Tabelle 6: Reduktion des Löwenmodells auf 100000 Vertices mit unterschiedlichen Glättungsdurchläufen

Abschließenden wurde das Testprogramm *decimator* entwickelt, welches die vorgestellte *ReduceMesh*-Bibliothek benutzt. Über Parameter kann dabei die Anzahl der zu dezimierenden Vertices, das Glätten, ein maximaler Fehlerwert und die Entfernung von kleinen Subobjekten eingestellt werden. Die Dezimierungsstrategie wird, so wie es Garland vorschlägt, dreistufig ausgeführt [Garland 1999]. Zuerst wird geprüft ob eine ideale Position für die zusammengezogene Kante bestimmt werden kann. Wenn diese nicht gefunden werden kann, dann wird eine neue Position auf der Kante gesucht. Falls auch hier keine Position gefunden werden kann, wird der kostengünstigere der beiden Vertices als neuer Endvertex ausgesucht.

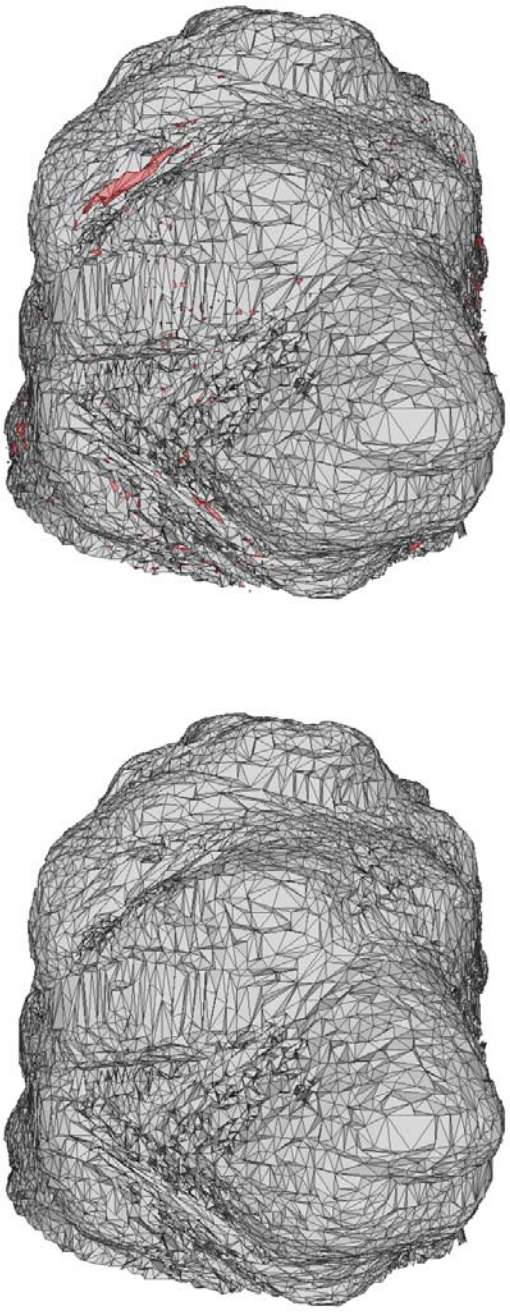
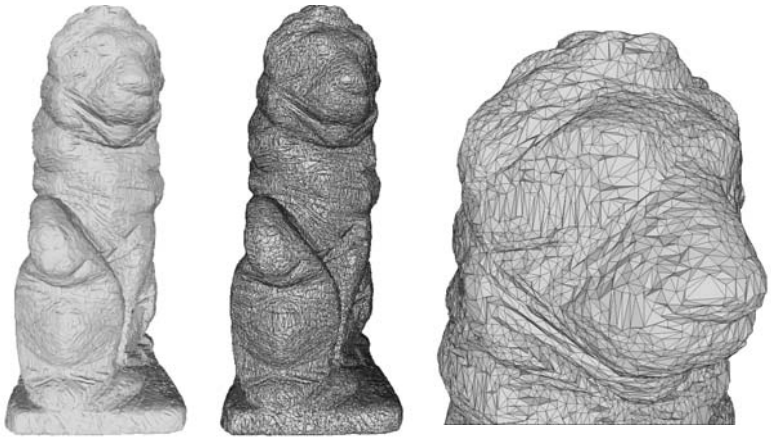
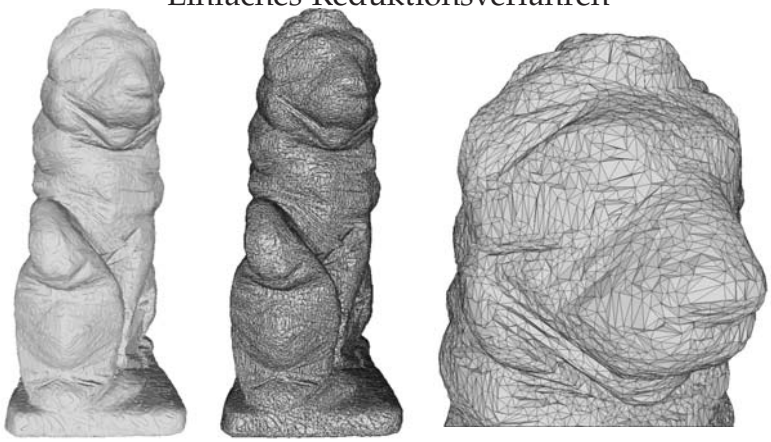


Abbildung 18: Kopf des Löwenmodells nach der Dezimierung; (oben) rot markiert sind die Artefakte; (unten) nach Anwendung des Algorithmuses

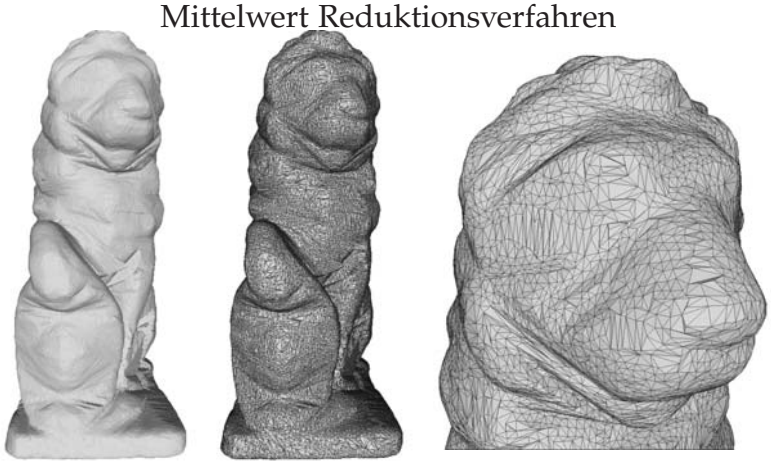
Abbildung 19: Kopf des Löwenmodells nach der Dezimierung; (oben) ohne Glättung; (unten) mit siebenfacher Laplace-Glättung



Einfaches Reduktionsverfahren



Mittelwert Reduktionsverfahren



Dreistufiges Reduktionsverfahren mit Glättung

Abbildung 20: Reduktion des Löwenmodells mit verschiedenen Verfahren

6 Zusammenfassung und Ausblick

Die Aufgabenstellung, ein Objekt einzuscannen um es dann in eine handhabbare Größe zu bringen, wurde erfüllt.

Eine Bibliothek wurde in *C++* realisiert, welche die Drehtellersteuerung übernimmt. Daneben lag der Schwerpunkt dieser Arbeit bei der Implementierung eines Polygonreduktionsverfahrens. Hier wurde das Verfahren von Garland implementiert und an die bestehenden Bedürfnisse angepasst. So wurden eigene Verfahren zur Fehlererkennung bei der Reduktion, wie etwa ein rekursives Verfahren zur Erkennung abgesplitterter Objekte, implementiert und eigene Reduktionsstrategien erprobt.

Einige Bereiche der Arbeit können für konkrete Anwendungen ausgeweitet werden. So wäre es etwa vorteilhaft eine GUI[10] zur Bedienung der Drehtellersteuerung zu implementieren. Außerdem könnte der ganze Ablauf automatisiert werden, so dass alle Komponenten transparent hintereinander ausgeführt werden und der Benutzer nur ein Objekt auf den Drehteller stellen muss um den Vorgang zu starten.

Es traten einige Schwierigkeiten, bei der Implementierung des *Quadric-Based*-Verfahrens und bei der Integration in die *Open-Mesh*-Umgebung auf, die jedoch beseitigt werden konnten. Allerdings wäre es sinnvoller einen Teil der entworfenen Erweiterung direkt in das *Open-Mesh*-Projekt zu integrieren oder aber das ganze Reduktions-Template unabhängig von der Bibliothek neu zu implementieren.

Die *Half-Edge*-Struktur eignet sich gut, auch wenn die Initialisierung sehr kostspielig ist, für das *Quadric-Based*-Verfahren von Garland. Allerdings kann das Verfahren noch erweitert und an die entsprechenden Bedürfnisse angepasst werden. Bei der Auswahl der zu dezimierenden Vertices könnten weitere Kriterien überprüft und in den Algorithmus integriert werden. So wäre es bei manchen Objekten von Vorteil, die Abweichung der Normalenwinkel zwischen den Polygonen zu messen um die Übergänge besser zu erkennen. Um gleichmäßige Polygone zu erzeugen könnte die Kantenlänge auch als Kriterium zur Gewichtung benutzt werden . Die Auswertung der Winkel innerhalb eines Polygons könnte

[10] Grafische Benutzeroberfläche

auch sinnvoll sein. Um die Polygonqualität zu verbessern könnten auch *Edge-Flips* angewendet werden. Außerdem könnten, wie Garland schon vorschlägt, Vertices, die nicht direkt miteinander verbunden sind, zusammengefügt werden. Ebenso sinnvoll wäre es, wenn einzelne auserwählte Vertices oder Vertexgruppen, die etwa in besonders kritischen Regionen des Polygonnetzes liegen, von der Reduktion ausgeschlossen werden könnten.

Die Fehlererkennung der Polygonorientierung kann ebenfalls noch verbessert werden. So kann das einfache Verfahren, welches hier angewendet wird, erweitert werden um die genaue Position des neuen Vertices innerhalb eines vorgegebenen Raums zu überprüfen. Weitere Tests um auch Objekte hinter der sichtbaren Oberfläche oder Aushöhlungen aufzuspüren sind über ein Strahlenverfolgungsverfahren denkbar.

Diese Eingriffe sollten jedoch immer an das Objekt angepasst werden um die effiziente Verabeitungsgeschwindigkeit nicht zu verschlechtern. Für dieses Projekt waren die vorgestellten und implementierten Verfahren jedoch sinnvoll ausgewählt.

Literatur

[Bender 2006] Bender, M., Brill, M.: *Computergrafik, Ein anwendungsorientiertes Lehrbuch*, 2006, Carl Hanser Verlag, München

[Botsch 2002] Botsch, M., Steinberg, S., Bischoff, S., Kobbelt, L.: *OpenMesh: a generic and efficient polygon mesh data structure*, 2002, OpenSG Symposium

[Bungartz 2002] Bungartz, H.J., Griebel, M., Zenger, C.: *Einführung in die Computergraphik*, 2002, Vieweg Verlag, Wiesbaden

[Campagna 1998] Campagna, S.: *Polygonreduktion zur effizienten Speicherung, Übertragung und Darstellung komplexer polygonaler Modelle*, 1998, Herbert Utz Verlag, München

[Chernyaev 1995] Chernyaev, E.V.: *Marching Cubes 33: Construction of Topologically Correct Isosurfaces*, 1995, Technical Report CERN

[Delaunay 1934] Delaunay, B.: *Sur la sphère vide*, Bulletin of Academy of Sciences of the USSR 7, 1934

[Eisert 2004] Eisert, P.: *3D Geometry Enhancement by Contour Optimization in Turntable Sequences* 2004, IEEE International Conference on Image Processing (ICIP), Singapur

[Eisert 2011] Eisert, P.: Vorlesungsskript *Computer Graphik*, Wintersemester 2011/2012, Berlin

[Garland 1999] Garland, M.: *Quadric-Based Polygonal Surface Simplification* 1999, Carnegie Mellon University, Pittsburgh

[Gotsman 2002] Gotsman, C., Gumhold, S., Kobbelt ,L.: *Simplification and compression of 3D-meshes* 2002, Tutorials on multiresolution in geometric modeling, Springer

[Hansen 2005] Hansen, A., Douglass, R.W., Zardecki, A.: *Mesh enhancement: selected elliptic methods, foundations and applications*,2005 ,Imperial College Press, London

[Häuser 1998] Häuser, S.: *Generierung, Darstellung und Interaktion mit Voxel*, 1998, Stuttgart

[Kobbelt 1998] Kobbelt, L., Campagna, S., Seidel, H.P.: *A General Framework for Mesh Decimation* 1998, in Graphics Interface

[Lehmann 2005] Lehmann, T. M.: *Handbuch der medizinischen Informatik*, 2005, Carl Hanser Verlag, München

[Lounsberry 1994] Lounsbery, J.M.: *Multiresolution Analysis for Surfaces of Arbitrary Topological Type*, 1994, University of Washington

[Lorenson 1987] Lorensen, W. E., Cline, H. E.: *Marching Cubes: A high resolution 3D surface construction algorithm*, 1987, Computer Graphics, Vol. 21

[Newman 2006] Newman, T.S., Yi, H.: *A survey of the marching cubes algorithm*, 2006, Computers & Graphics

[Otto 2005] Otto, F.: *Prototypische Realisierung eines 3D Modelling Systems nach dem Sculpturing Ansatz*, 2005, Berlin

[OpenMesh 2012] OpenMesh Dokumentation: *The Halfedge Data Structure*, http://openmesh.org/Documentation/OpenMesh-Doc-Latest/mesh_hds.html, Konsultationsdatum: 05. März 2012

[Rossignac 1993] Rossignac, R., Borrel, P.: *Multi-resolution 3D approximations for rendering complex scenes*, 1993, Geometric Modeling in Computer Graphics, Springer Verlag, Berlin

[Schroeder 1992] Schroeder, W.J., Zarge, J.A., Lorensen, W.E.: *Decimation of Triangle Meshes*, 1992, Proceedings ACM SIGGRAPH 26, 65-70

[Szeliski 1993] Szeliski, R.: *Rapid Octree Construction from Image Sequences*, 1993, CVGIP: Image Understanding Vol. 58

[Zdunczyk 2006] Zdunczyk, B.: *Analyse von Repräsentationen für 3D-Modelle aus Sicht der Softwaretechnik*, 2006, Koblenz

[Ziskel 2004] Ziskel, L.: *Hochauflösende 3D-Rekonstruktion von Objekten aus kalibrierten Einzelansichten*, 2004, Berlin

[Zhou 2000] Zhou, T., Shimada, K.: *An Angle-Based Approach to Two-Dimensional Mesh Smoothing*, 2000, 9th International Meshing Roundtable, Sandia National Laboratories

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis