Bachelorarbeit

Constrained Mesh Simplification in CGAL

Maximilian Röhrl
Matnr: 77965

betreut von
Prof. Dr. Tomas Sauer

11. Juni 2019

Kurzfassung

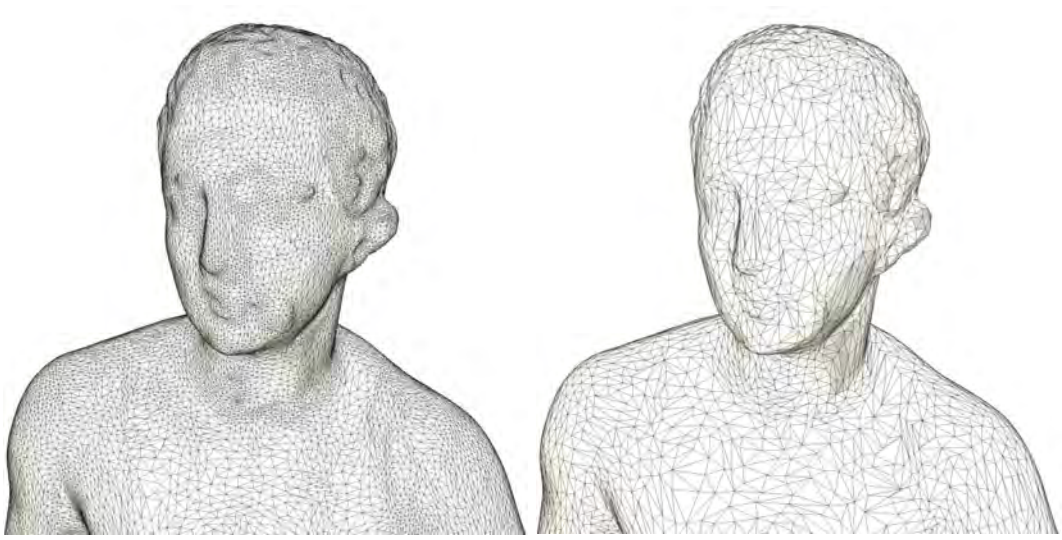
Durch die Verbesserung von Laserscan-Verfahren können Meshes mit sehr vielen Punkten erzeugt werden. Diese Meshes können durch Vereinfachung ausgedünnt werden, was die Komplexität unter Beibehaltung wichtiger geometrischer Merkmale verringert. In dieser Arbeit wird ein Ansatz vorgestellt, bei dem bestimmte Merkmale der Vertices und Faces wie Farbe bei der Vereinfachung berücksichtigt werden.

Inhaltsverzeichnis

Kapitel 1

Einführung

Für diese Arbeit wurde ein Algorithmus erstellt, der die Anzahl der Kanten eines Meshes bis zu einer gewissen Anzahl verringert. Dazu wurde sich an der CGAL Bibliothek orientiert, welche auch eine vorgegebene Funktion für diesen Zweck bietet. Der Algorithmus versucht die Eigenschaften des Meshes so gut wie möglich zu erhalten. Dabei können Kanten bei der Ausdünnung ignoriert werden, welche sich an Stellen befinden, an denen sich das Mesh hinsichtlich der Farbe stark unterscheidet. Es wird der Algorithmus von Lindstrom und Turk aus [LT98] verwendet, der in CGAL implementiert ist, um zu bestimmen, welche Kanten wie entfernt werden. Somit kann der Speicherbedarf des Meshes stark verringert werden, ohne dass große Qualitätseinbußen erlitten werden.



(a) Ursprüngliches Mesh (b) Ausgedünntes Mesh

Abbildung 1.1: Ein Ausschnitt eines Meshes vor und nach der Vereinfachung durch den Algorithmus

Im ersten Kapitel werden wichtige Eigenschaften von Meshes und die Theorie hinter Mesh Simplification erläutert. Im zweiten Kapitel werden die mathematischen Hintergründe des gewählten Algorithmus und Details zur Implementierung beschrieben. Im dritten Kapitel werden die Ergebnisse und die Komplexität des Algorithmus betrachtet und bewertet.

Kapitel 2

Mesh Simplification

2.1 Eigenschaften von Meshes

Ein Polygon Mesh besteht aus Vertices, Edges und Faces, also Eckpunkten, Kanten und Facetten, welche die Form eines Objektes in der Computergrafik beschreiben. Ein Edge ist immer die Verbindung zwischen zwei Vertices. Man kann ein Edge auch als zwei entgegengesetzte Halfedges betrachten, wobei jedes Halfedge zu einem der beiden Vertices der Kante zeigt.
Ein Face ist durch mehrere Edges begrenzt. Dabei sind die Faces meist Dreiecke, aber können auch Vierecke oder andere Polygone sein. Dreiecke bieten den Vorteil, dass sie immer konvex sind, also dass zwischen zwei beliebigen Punkten innerhalb eines Dreiecks immer auch die Verbindungsstrecke Teil des Dreiecks ist. Die Mesh Topologie beschreibt, welche Mesh Elemente nebeneinander liegen, also zum Beispiel welche Vertices und Edges eines Faces benachbart liegen.
Ein Mesh ist 2-mannigfaltig beziehungsweise zweidimensional mannigfaltig, wenn das Mesh an bestimmten Kanten aufgetrennt werden kann und so in eine zweidimensionale Fläche entfaltet werden kann, ohne dass es zu Überschneidungen kommt. Dies ist nur möglich, wenn folgende Eigenschaften gelten [vgl. Mano9]:

- *face count property*: Jede innere Kante hat zwei benachbarte Faces und jede äußere Kante ein benachbartes Face.
- *local disc property*: Jeder Schnitt einer ϵ-Umgebung und dem Mesh ist homöomorph zu einer planaren Scheibe oder einer planaren Halbscheibe an den Rändern.
- *edge ordering property*: Die direkt anliegenden Vertices eines Vertex müssen eindeutig im Uhrzeigersinn aufzählbar sein

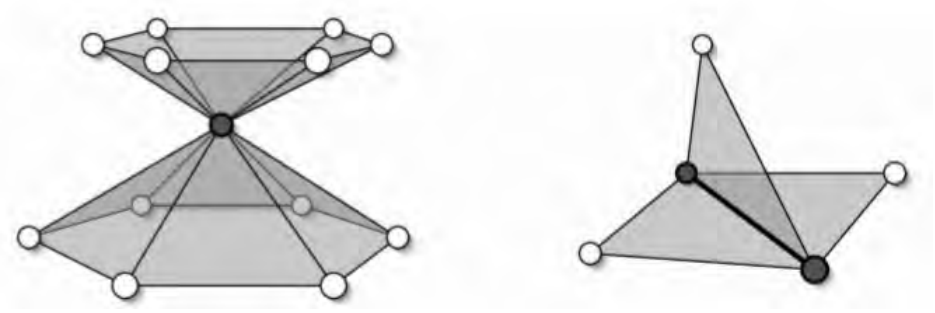


Abbildung 2.1: Ein nicht mannigfaltiges Vertex, welches die *edge ordering property* verletzt, (links) und eine nicht mannigfaltige Kante, welche die *face count property* verletzt, (rechts). [Siehe Bot+10, S. 12]

Ein mannigfaltiges Mesh heißt orientierbar, wenn alle benachbarten Faces eine entgegengesetzte zyklische Ordnung der Vertices eines einzelnen Faces zulassen.

2.2 Incremental Decimation Algorithmen

Dieses Kapitels basiert auf den Erkenntnissen aus [Bot+10, S. 112, 115 f.]. Die Vereinfachung von Meshes beziehungsweise Mesh simplification beschreibt eine Klasse von Algorithmen, die ein bestehendes Mesh in ein anderes Mesh mit weniger Vertices, Edges und Faces umwandelt. Es werden vom Benutzer festgelegte Kriterien beachtet, damit das neue Mesh gewisse Eigenschaften von dem ursprünglichem Mesh besitzt bezüglich geometrischer Abweichung und Form. Dabei kann man diese Reduktion der Komplexität eines Meshes als Einzelschrittverfahren oder iterative Operation ansehen. Da der iterative Ansatz in CGAL verwendet wird und auch der geschriebene Algorithmus ein Incremental Decimation Algorithmus ist, werden anschließend diese Algorithmen genauer erklärt.
Bei Incremental Decimation Algorithmen wird eine Kante beziehungsweise ein Vertex nach dem anderen entfernt. Dazu können die Entfernungsoperationen aus Kapitel 2.3 verwendet werden. Die Reihenfolge, in der die Kanten entfernt werden, kann vom Benutzer angepasst werden und wird in Kapitel 3 von den Kosten einer Kante bestimmt. Um die Reihenfolge der Kandidaten für die Entfernung beizubehalten, wird meist eine modifizierbare *heap data* Struktur verwendet, bei der der nächste Kandidat oben auf liegt.
Immer nachdem eine Entfernungsoperation stattgefunden hat, verändert sich in einer bestimmten Umgebung die Geometrie des Meshes und somit auch die Qualitätskriterien und die Kosten des betroffenen Edge. Diese Neuberechnung ist rechenmäßig der aufwendigste Teil des Algorithmus. Bei einer solchen Neuberechnung werden alle Betroffenen aus der Struktur entfernt und nach der Berechnung neu eingefügt am richtigem Platz. Damit erhöht sich der Zeitaufwand für diese Neuberechnung bei großen Meshes nur um $\mathcal{O}(\log n)$, falls die Kriterienberechnung selbst konstante Komplexität besitzt. Dabei ist der Rechenaufwand eines Incremental Decimation Algorithmus mit $\mathcal{O}(n \log n)$ höher als bei anderen Algorithmen und kann auch $\mathcal{O}(n^2)$ erreichen, wenn eine globale Fehlertoleranz eingehalten werden soll.

Der große Vorteil von Incremental Decimation Algorithmen ist allerdings, dass diese eine hohe Qualität des entstehenden Meshes bieten. Andere Klassen wie Vertex Clustering Algorithmen bieten meist eine niedrigere Qualität, aber dafür eine bessere Laufzeit von $\mathcal{O}(n)$ abhängig von der Anzahl der Vertices.

2.3 Topologische Operationen

Dieses Kapitels basiert auf den Erkenntnissen aus [Bot+10, S. 116–119]. Es gibt mehrere Möglichkeiten, um eine Entfernungsoperation durchzuführen. Dabei ist zu beachten, dass immer nur ein Vertex oder eine Kante pro Operation entfernt wird und keine größeren Teile des Meshes entfernt werden. Durch Hintereinanderausführung vieler einfacher Operationen wird ein größerer Teil des Meshes vereinfacht. Wenn die Topologie des Meshes nicht verändert werden soll, muss eine Euler Operation verwendet werden. Diese Operationen verändern die Euler Charakteristik des Meshes nicht. Die Euler-Pioncaré Charakteristik lautet:

$$v - e + f - h = 2(b - g)$$

Dabei ist v die Anzahl der Vertices, e die Anzahl der Edges, f die Anzahl der Faces, h die Anzahl der Windungen, b die Anzahl der Hüllen des Meshes und g das Geschlecht des Meshes, also die Anzahl der Löcher oder Griffe [vgl. Str06, S. 83]. Somit werden keine Löcher im Mesh geschlossen. Die Anzahl der Windungen h ist 0 und die Anzahl der Hüllen b ist 1 für 2-mannigfaltige Meshes, die keinen Rand besitzen und somit geschlossen sind [vgl. Man09]. Anschließend werden drei Euler Operationen und eine nicht Euler Operationen betrachtet.

2.3.1 Vertex Removal

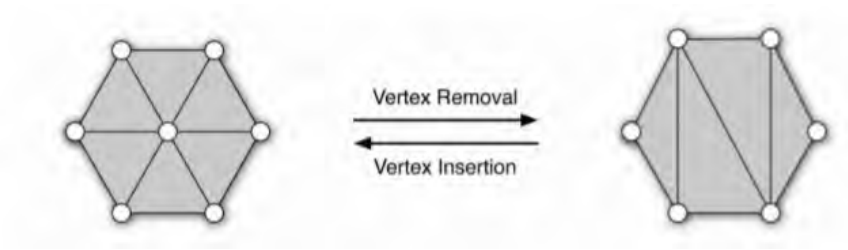


Abbildung 2.2: Eine *Vertex Removal* Operation [Siehe Bot+10, S. 117]

Bei dieser Methode wird ein Vertex und alle anliegenden Dreiecke entfernt. Dabei bleibt für ein Vertex mit k ausgehenden Edges ein k-seitiges Loch im Mesh. Dieses Loch kann durch einen Triangulierungsalgorithmus repariert werden. Die Anzahl an entstehenden Dreiecken ist immer $k - 2$ und damit wird die Anzahl der Dreiecke um 2, die Anzahl der Kanten um 3 und die Anzahl der Vertices um 1 vermindert.

2.3.2 Edge Collapse

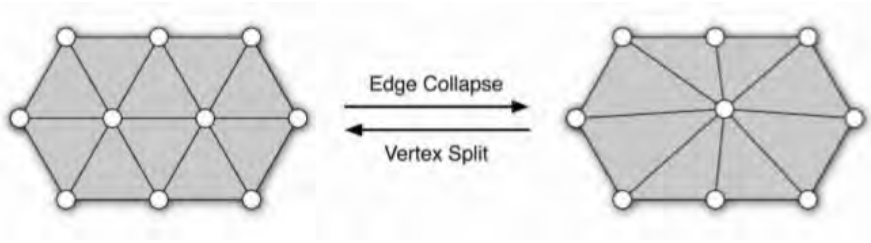


Abbildung 2.3: Eine *Edge Collapse* Operation [Siehe Bot+10, S. 117]

Beim *Edge Collapse* wird die Kante zwischen zwei benachbarten Vertices zusammengezogen bzw. beide Vertices werden zu einer neuen Position verschoben. Dabei werden zwei Dreiecke degeneriert und können vom Mesh gelöscht werden. Es werden 2 Dreiecke, 3 Kanten und 1 Vertex entfernt. Die Position des neuen Vertex kann frei bestimmt werden. Diese Positionierung des neuen Vertex wird in den folgenden Kapitel als *Placement* bezeichnet.

2.3.3 Halfedge Collapse

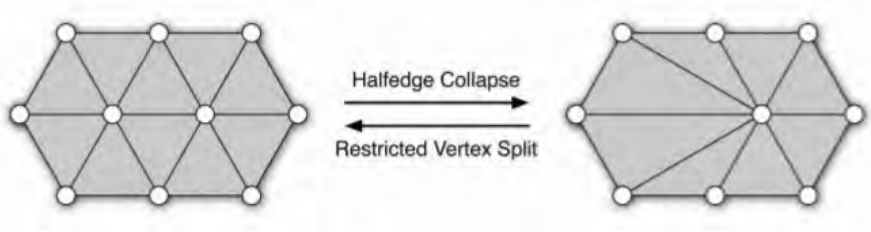


Abbildung 2.4: Eine *Halfedge Collapse* Operation [Siehe Bot+10, S. 117]

In einem geordnetem Paar zweier Vertices v_0 und v_1, die ein Halfedge (v_0, v_1) beschreiben, wird v_0 zur Position von v_1 geschoben. Dies kommt einem *Edge Collapse* gleich, wobei die Position des neuen Vertex gleich v_1 ist. Man kann dies aber auch als *Vertex Collapse* betrachten, bei dem das entstehende Loch so trianguliert wird, dass jedes benachbarte Vertex von v_0 mit einer Kante zu v_1 verbunden wird.
Bei dieser Operation hat man im Vergleich mit den anderen keine Entscheidungsfreiheit, was die neue Vertex Position angeht. Der Vorteil ist hierbei, dass das Design von Mesh Simplification Algorithmen einfacher gestaltet wird, indem diese Operation einem keine Entscheidungsfreiheit über das *Placement* überlässt.

Es können bei *Edge Collapse* und *Halfedge Collapse* Operatoren auch topologisch inkorrekte Meshes entstehen. Deswegen müssen für eine valide Operation folgende Kriterien eingehalten werden:

- Falls v_0 und v_1 Rand-Vertices sind, muss das Edge (v_0, v_1) auch ein Rand-Edge sein.

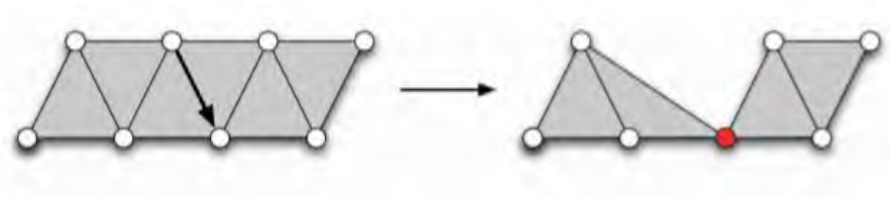


Abbildung 2.5: Ein *Halfedge Collapse*, welcher die Topologie des Meshes verändert. Eine innere Kante zwischen zwei äußeren Vertices wird zusammengezogen und es entsteht ein nicht mannigfaltiges Vertex, welches rot markiert ist. [Siehe Bot+10, S. 119]

- Für alle Vertices v_2 neben v_0 und v_1 muss es ein Dreieck (v_0, v_1, v_2) im Mesh geben.

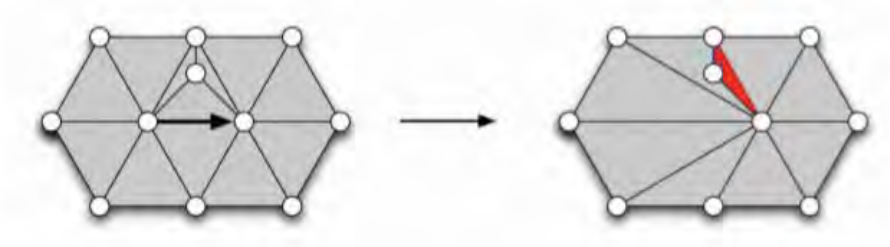


Abbildung 2.6: Ein *Halfedge Collapse*, welcher die Topologie des Meshes verändert. Es gibt kein Dreieck im Mesh für ein benachbartes Vertex und es entsteht eine nicht mannigfaltige Kante und ein sich überlagerndes Dreieck, welches rot markiert ist. [Siehe Bot+10, S. 119]

Wenn beide Kriterien eingehalten werden, ändert sich die Topologie des Meshes nicht nach einer Operation. Keine Löcher in der Oberfläche des Meshes werden geschlossen und keine verbundenen Komponenten des Meshes werden getrennt.

2.3.4 Vertex Contraction

Damit auch die Topologie eines Meshes geändert werden kann, finden Operationen wie *Vertex Contraction* Verwendung. Dabei werden zwei Vertices v_0 und v_1 an die Position eines neuen Vertex v verschoben, wobei v_0 und v_1 keine gemeinsame Kante haben müssen. Die Anzahl der Vertices wird um eins verringert und die Anzahl der Dreiecke bleibt gleich. Es wird eine flexible Datenstruktur benötigt, damit nicht mannigfaltige Meshes dargestellt werden können, da die Umgebung von v nach dem Zusammenziehen nicht unbedingt homöomorph zu einer planaren Scheibe ist.

Kapitel 3

Mesh Simplification in CGAL

Der praktische Teil der Bachelor Arbeit besteht darin, einen Algorithmus in C++ zu implementieren, der ein vorgegebenes Mesh soweit vereinfacht, dass Bereiche mit verschiedenen Eigenschaften bei der Ausdünnung ausgelassen werden. Zum Beispiel in Bereichen, in denen sich die Textur Farben sehr stark unterscheiden, sollen mehr Kanten erhalten bleiben. Dazu wurde die CGAL Bibliothek verwendet, welche im nächsten Abschnitt genauer beschrieben wird. Anschließend wird in diesem Kapitel der Algorithmus zur Constrained Mesh Simpilification beschrieben.

3.1 Die CGAL Bibliothek

Die *Computational Geometry Algorithms Library* ist eine Software Bibliothek, die in C++ geschrieben ist. CGAL stellt Funktionen und Datentypen zur Erstellung, Speicherung und Bearbeitung von verschiedenen Mesh Formaten bereit.
Dabei bietet CGAL eine sehr hohe Funktionsvielfalt und verwendet verschiedene Namensräume, um die Funktionen und Klassen sinnvoll zu unterteilen und zu gruppieren. Die Dokumentation lässt sich online abrufen unter `doc.cgal.org` und dabei ist auch jedes Paket in der Online Dokumentation mit mehreren kurzen Beispielen beschrieben. Es stehen dabei auch viele Demos und Beispiele zur Verfügung, von denen die *Polyhedron3* Demo benutzt wurde, um Mesh Dateien öffnen und darstellen zu können. Für diese Arbeit wurde ein Plugin geschrieben, so dass man über das User Interface dieser Demo eine Mesh Vereinfachung aufrufen kann, bei der man verschiedene Parameter über einen Dialog anpassen kann.

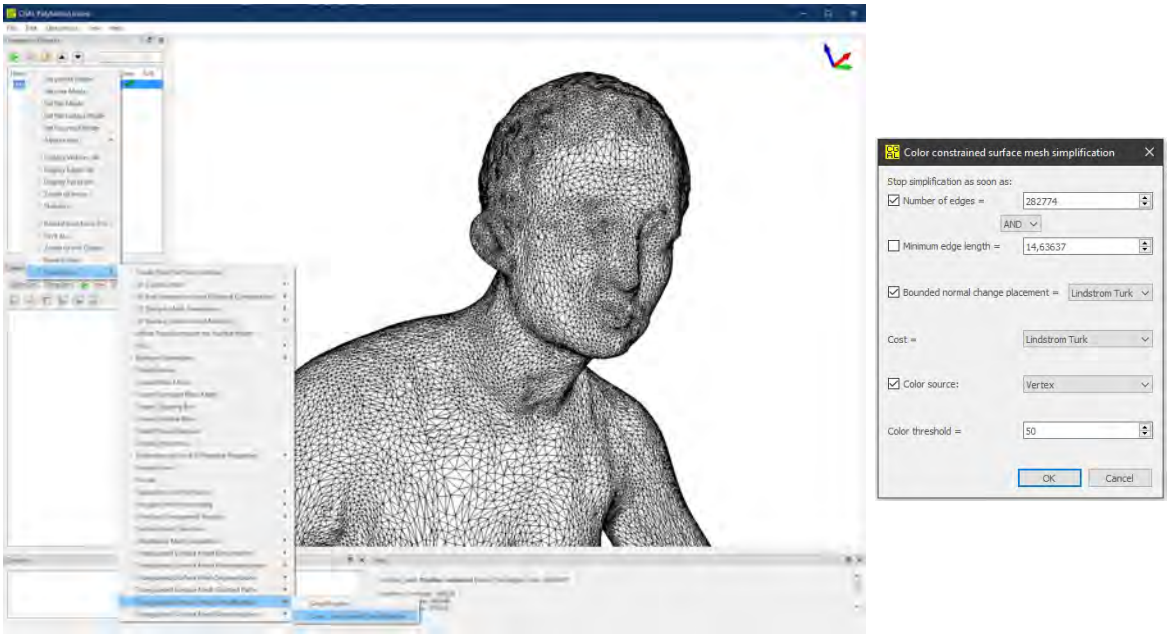


Abbildung 3.1: Links ist ein Screenshots der *Polyhedron3* Demo mit geöffnetem Menü zu sehen und rechts der Dialog zum Anpassen des Simplification Algorithmus.

CGAL hängt von der *Boost* und der *Eigen* Bibliothek ab. Einige Demos benötigen auch zusätzlich noch die *Qt5* Bibliothek zur Darstellung einer Benutzeroberfläche wie sie zum Beispiel die *Polyhedron3* Demo bietet.
Im allgemeinen Design von CGAL lassen sich drei Schichten erkennen, die Schicht der Algorithmen und Datenstrukturen, die Kernel Schicht und die Schicht der Arithmetik und Algebra [vgl. HKS18]. Dabei finden C++ Templates Verwendung, um CGAL so generisch wie möglich zu gestalten. Die Algorithmen und Datenstrukturen können somit parametrisiert werden mit konkreten Template Argumenten, wenn diese die entsprechenden syntaktischen und semantischen Anforderungen des Algorithmus oder der Datenstruktur erfüllen. Damit die Menge an Parametern nicht unübersichtlich wird, verwendet man *Traits* Klassen, um viele Parameter zu gruppieren. Diese Art von Klassen findet man in der Kernel Schicht.

3.2 Die CGAL Mesh Simplification Funktion

CGAL hat die freie Template Funktion `CGAL::Surface_mesh_simplification::edge_collapse()`, welche zur Vereinfachung eines Meshes benutzt werden kann. Diese Funktion hat dabei als Hauptparameter das Mesh, welches vereinfacht wird, und das Stop Prädikat, welches bei jeder Iteration im Algorithmus aufgerufen wird, um festzustellen, wann der Algorithmus genug Kanten entfernt hat. Das Mesh muss bei dieser Funktion ein Model der Klassen `MutableFaceGraph` und `HalfedgeListGraph` sein. Zum Beispiel wurde im praktischem Teil der Arbeit die Klasse `CGAL::Surface_mesh<Point_3>` verwendet, welche besagte Voraussetzungen erfüllt und eine Halfedge Datenstruktur ist.

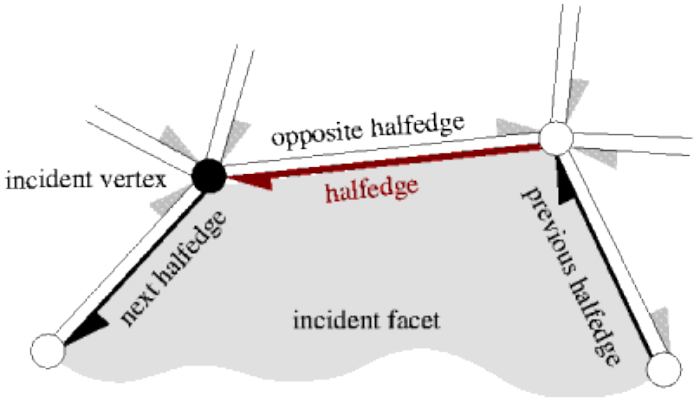


Abbildung 3.2: Aufbau einer Halfedge Datenstruktur [Siehe Ket18]

Eine Halfedge Datenstruktur ist eine doppelt verkettete Kantenliste [vgl. Ket18]. Ein Halfedge besitzt ein entgegengesetztes Halfedge, mit der es ein Edge bildet. Diese in CGAL verwendete Datenstruktur ist auch bekannt als *doubly connected edge list* (DCEL) und ist beschränkt auf orientierbare 2-mannigfaltige Meshes. Zu jedem Halfedge werden Referenzen auf folgende Elemente gespeichert [vgl. Bot+10, S. 25]:

- das Vertex, auf das gezeigt wird
- das anliegende Face, welches ein Nullpointer bei einem Halfedge am Rand des Meshes ist
- das nächste Halfedge im Face oder am Rand im entgegengesetzten Uhrzeigersinn
- das vorherige Halfedge im Face
- das gegenüberliegende Halfedge

Jedes Face enthält eine Referenz auf eines seiner Halfedges und jedes Vertex speichert seine Position und das Halfedge, das von dem Vertex ausgeht.

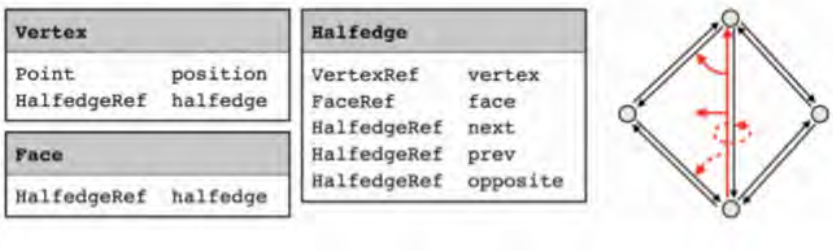


Abbildung 3.3: Die in einer Halfedge Datenstruktur gespeicherten Informationen über die Verbindungen der Mesh Elemente. [Siehe Bot+10, S. 25]

Die Vereinfachung wird *in Place* durchgeführt, dass heißt, es wird das Mesh direkt geändert und es wird kein neues Mesh erstellt, um Speicher zu sparen. Die Vereinfachungsfunktion ist im *policy-based* Design implementiert, welches ermöglicht, dass einige Aspekte des Algorithmus angepasst werden können, indem *Policy*

Objekte der Funktion übergeben werden. Dies passiert mit Hilfe einiger optionaler Parameter und wird mit *Boost Graph Library Named Parameters* umgesetzt. Somit können nicht benötigte Parameter ausgelassen werden und bleiben dann auf dem Standartwert. Da nur der Name eines Parameters angegeben werden muss, ist die Reihenfolge der Parameter irrelevant.

Diese optionalen Parameter sind:

- `vertex_index_map`: Eine *property map*, die den Index jedes Vertex für Debugging Zwecke enthält
- `vertex_point_map`: Eine *property map*, die jedem Vertex einen Punkt im Raum zuweist.
- `halfedge_index_map`: Eine *property map*, die den Index eines jeden Halfedges enthält
- `edge_is_constrained_map`: Eine *property map*, die für jede Kante die nicht geändert werden darf, `true` enthält und ansonsten `false`.
- `get_cost`: Das *Policy* Objekt, das für jede Kante die Kosten liefert
- `get_placement`: Das *Policy* Objekt, das für jede Kante die optimale Position des neuen Vertex nach dem Zusammenziehen der Kante zurückliefert
- `visitor`: Das *Visitor* Objekt, das an gewissen Punkten des Algorithmus aufgerufen wird und somit zum Beispiel Statistiken zum Vereinfachungsprozess zu sammeln.

Die Kosten einer Kante sind ein Fehlermaß und bestimmen die Reihenfolge, in der die Kanten für den *Edge Collapse* in Betracht gezogen werden. Dabei leiten sich die Kosten von dem *Placement* des Vertex ab, das die jeweilige Kante ersetzt. Um die *Collapse* Kosten und das *Placement* zu bestimmen, wird eine Kosten- und Platzierungsstrategie verwendet, die per Parameter mit einem *Policy* Objekt übergeben werden kann.

Im ersten Schritt des Algorithmus werden alle Kanten gesammelt und mit ihren Kosten in einer *Priority Queue* gespeichert. Somit kann die Kante mit den niedrigsten Kosten im nächsten Schritt leicht ausgewählt werden und wird als potenzieller Kandidat für ein *Edge Collapse* betrachtet. Dabei kann die Kante aber auch abgelehnt werden, wenn keine Position für das neue Vertex berechnet werden kann oder andere Bedingungen nicht erfüllt sind (siehe Kapitel 3.5.2). Bei jeder weiteren Iteration werden alle verbleibenden Kanten betrachtet und die mit den niedrigsten Kosten wird aus der *Priority Queue* ausgewählt. Die Kosten werden dabei auch zu jeden Vertex in der *Priority Queue* gespeichert.

Das *Placement*, welches einen dreidimensionalen Punkt entspricht, wird aber nicht zu jeder Kante in der Warteschlange mitgespeichert, da sonst der zusätzliche Speicherbedarf verdreifacht werden würde [vgl. Cac18]. Da sich aber die Kosten von der Platzierung des neuen Vertex ableiten, muss im Endeffekt das *Placement* noch einmal berechnet werden, wenn eine Kante dann entfernt wird, um die neue Position des verbleibenden Vertex zu berechnen. Dies ist ein Kompromiss an Laufzeit, um Speicher zu sparen. Laut [Cac18] reduziert es aber nicht die Laufzeit,

wenn man das *Placement* zu jeder Kante mitspeichert, da sich dieses ändert, wenn Kanten in der Nähe entfernt werden, und somit immer nur die Berechnung des *Placement* direkt beim *Edge Collapse* eingespart wird.

Es wird das *Edge Collapse* Verfahren aus Kapitel 2.3.2 verwendet, bei dem ein Vertex, zwei Faces und drei Edges entfernt werden. Die Position des neuen Vertex wird durch die *Placement* Strategie bestimmt.

In CGAL sind zwei *Placement* Strategien implementiert, die *Lindstrom Turk* Strategie und die *Edge Length* Strategie, bei welcher der Mittelpunkt der betrachteten Kante als neue Vertex Position verwendet wird. Die letztgenannte Strategie ist sehr trivial und viel schneller als die *Lindstrom Turk* Strategie, aber liefert weniger akkurate Ergebnisse [vgl. Cac18].

3.3 Lindstrom Turk Kosten- und Platzierungsstrategie

In diesem Abschnitt wird die *Lindstrom Turk* Kosten- und Platzierungsstrategie von Peter Lindstrom und Greg Turk erläutert, welche in CGAL standardmäßig verwendet wird, wenn keine andere per *Policy* Objekt übergeben wird. Diese Erkenntnisse basieren auf dem Paper von [LT98], wobei einige Formeln auch aus der CGAL Implementierung der *Lindstrom Turk* Strategie [siehe CGA18] abgeleitet wurden. Die wichtigste Eigenschaft dieser Strategie ist es, dass das Mesh nicht bei jedem Schritt mit dem ursprünglichen oder in vorherigen Schritten entstandenem Mesh verglichen werden muss. Somit muss immer nur das aktuelle Mesh gespeichert werden und die Vereinfachung wird auch *memoryless* genannt.

Es wird das Kreuzprodukt verwendet, welches einen Vektor liefert, der senkrecht auf a und b steht:

$$a \times b = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \times \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{bmatrix}$$

Die Länge dieses Vektors entspricht dem Flächeninhalt des von a und b aufgespannten Parallelogramms.

3.3.1 Bestimmung des Placement

Die Position des neuen Vertex wird hier aus der Lösung eines Systems aus drei linear unabhängigen linearen Gleichungen berechnet. Es wird versucht die Veränderung einiger geometrischer Eigenschaften wie Volumen und Fläche zu minimieren. Dabei wird dies in drei *Constraints* ausgedrückt, welche Flächen beschreiben. Falls zwei dieser Flächen linear abhängig sind, führen kleine Abweichungen in den Koeffizienten zu großen Abweichungen in der Lösung, da es somit häufiger zu Rundungsfehlern kommt.

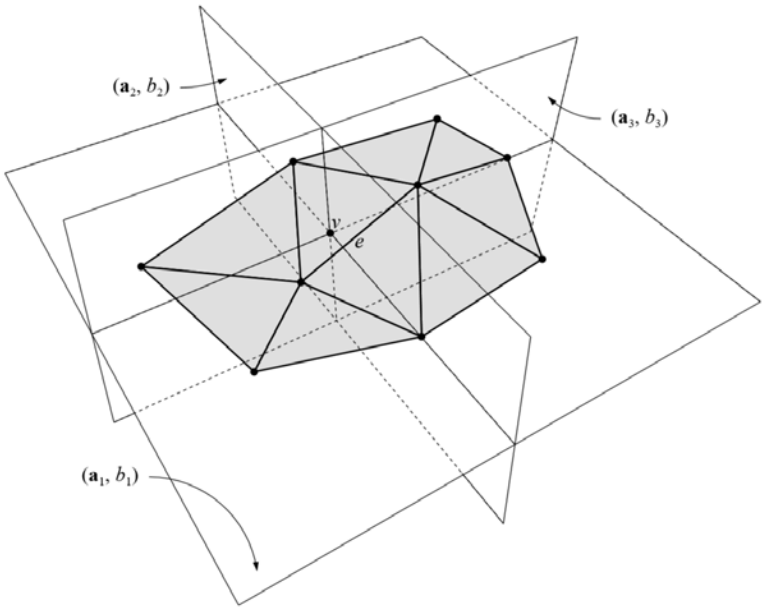


Abbildung 3.4: Die optimale Vertex Position v für den *Edge Collapse* von e ist der Schnittpunkt der drei Flächen, die von den *Constraints* (a_i, b_i) beschrieben werden. [Siehe LT98, S. 4]

Es werden *Constraints* nur dann hinzugefügt, wenn die Normale a_n der Fläche, die ein neues *Constraint* beschreibt, nicht unter einem bestimmten Winkel α zu den Linearkombinationen der Normalen $\{a_i : 0 < i < n\}$ der durch die bisherigen Gleichungen bestimmen Flächen liegt. Bei $n-1$ bisherigen *Constraints* ($n \leq 3$) wird (a_n, b_n) nur akzeptiert, falls folgendes erfüllt ist:

$$\underline{n = 1:}\ a_1 \neq 0$$
$$\underline{n = 2:}\ (a_1^T a_2)^2 < (\|a_1\|\|a_2\|\cos(\alpha))^2$$
$$\underline{n = 3:}\ ((a_1 \times a_2)^T a_3)^2 > (\|a_1 \times a_2\|\|a_3\|\sin(\alpha))^2$$

Falls die jeweilige Bedingung erfüllt ist, nennt man (a_n, b_n) *α-compatible* zu den bisherigen *Constraints*. Dabei kann α auf 1° gesetzt werden wie in [LT98] vorgegeben. Nun lässt sich das *Placement* des neuen Vertex v bestimmen durch:

$$\underbrace{\begin{bmatrix} a_1^T \\ a_2^T \\ a_3^T \end{bmatrix}}_{A} v = \underbrace{\begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}}_{b}$$

wobei a_i^T der i-ten Reihe der Matrix A entspricht und b_i den i-ten Eintrag des Vektors b für $i \in \{1, 2, 3\}$. Somit müssen drei *Constraints* festgelegt werden, welche untereinander *α-compatible* sind, damit die Lösung für das *Placement* v eindeutig bestimmt werden kann.

Es werden folgende *Constraints* in Betracht gezogen, wobei die Formeln aus der CGAL Implementierung von *Lindstrom Turk* [siehe CGA18] abgeleitet wurden. Die

Auswahl, welches Vertex v_0, v_1 und v_2 entspricht, kann Auswirkungen auf das Vorzeichen der berechneten Werte haben und ist durch die CGAL Implementierung eindeutig bestimmt.

- **Boundary Preservation**: Es wird versucht die Fläche, die der Rand des Meshes festlegt, nicht zu verändern.

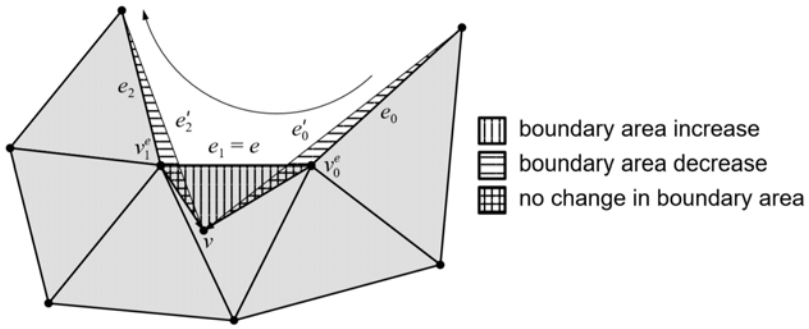


Abbildung 3.5: Das Edge e wird durch *Edge Collapse* mit der neuen Vertex Position v entfernt. Der Pfeil gibt die Orientierung der *Boundary* Edges e_1, e_2 und e_3 an. [Siehe LT98, S. 4]

Die Summe in den folgenden Termen summiert über die Vertices $v_{0,i}$ und $v_{1,i}$ der entsprechenden Boundary Edges auf, die in der Umgebung der betrachteten Kante liegen, wie sie in Abbildung 3.5 zu sehen ist. Es wird das *LT* Produkt aus dem Paper von Lindstrom und Turk verwendet, welches hier aber schon in ausmultiplizierter Form angegeben ist, welche in CGAL verwendet wird um Rechenoperationen zu sparen.

$$LT(\begin{bmatrix} x \\ y \\ z \end{bmatrix}) = \begin{bmatrix} y^2+z^2 & xy & xz \\ xy & x^2+z^2 & yz \\ xz & yz & x^2+y^2 \end{bmatrix}$$

$$E = \{\text{Indices der Rand Kanten in der Umgebung}\}$$

$$H = LT(\sum_{i \in E} v_{1,i} - v_{0,i})$$

$$c = (\sum_{i \in E} v_{1,i} - v_{0,i}) \times (\sum_{i \in E} v_{0,i} \times v_{1,i})$$

- **Volume Preservation**: Es wird versucht das Volumen des Meshes so wenig wie möglich zu verändern, um die Form zu erhalten. Es wird über die Vertices $v_{0,i}, v_{1,i}, v_{2,i}$ der einzelnen Faces summiert, die an der betrachteten Kante anliegen, wobei $v_{1,i}$ immer ein Vertex der betrachteten Kante ist. Hier kann (a_n, b_n) direkt ermittelt werden.

$$F = \{\text{Indices der Dreicke, die an der Kante angrenzen}\} \tag{3.1}$$

$$a_n = \sum_{i \in F} (v_{1,i} - v_{0,i}) \times (v_{2,i} - v_{0,i})$$

$$b_n = \sum_{i \in F} (v_{0,i} \times v_{1,i})^T v_{2,i}$$

- **Boundary Optimization**: Es wird im Vergleich zu *Boundary Preservation* der gesamte Fehler, der durch Veränderungen des Randes eingeführt wurde, betrachtet. Verschiedene Änderungen können sich dann nicht gegenseitig aufheben. v_0 und v_1 bezeichnen die Vertices der betrachteten Kante. Die Gewichtung $w_{Boundary}$ ist in CGAL standardmäßig $\frac{1}{2}$, aber kann vom Benutzer angepasst werden.

$$
\begin{aligned}
w &= 9\, w_{Boundary} \|v_0 - v_1\|^2 \\
H &= w \sum_{i \in E} LT(v_{1,i} - v_{0,i}) \\
c &= w \sum_{i \in E} (v_{1,i} - v_{0,i}) \times (v_{0,i} \times v_{1,i})
\end{aligned}
$$

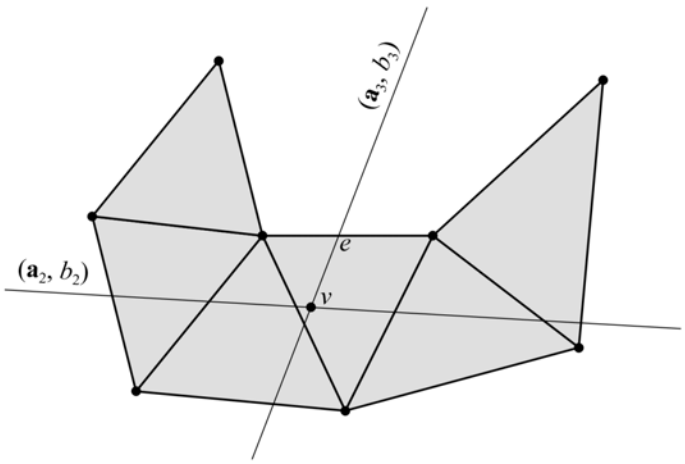


Abbildung 3.6: Das *Boundary* Edge e wird durch *Edge Collapse* mit der neuen Vertex Position v entfernt. (a_2, b_2) ist die Fläche der *Boundary Preservation Constraint* und (a_3, b_3) die der *Boundary Optimization Constraint*. Die nicht eingezeichnete Fläche der *Volume Preservation Constraint* (a_1, b_1) ist parallel zu der Fläche der Dreiecke. [Siehe LT98, S. 5]

- **Volume Optimization**: Es wird im Vergleich zu *Volume Preservation* der gesamte Fehler betrachtet, der durch die Änderung des Volumens entsteht. Auch hier können sich dann verschiedene Änderungen nicht gegenseitig aufheben. Die Gewichtung w_{Volume} ist in CGAL standardmäßig $\frac{1}{2}$, aber kann auch angepasst werden.

$$
\begin{aligned}
H =& w_{Volume} \sum_{i \in F} ((v_{1,i} - v_{0,i}) \times (v_{2,i} - v_{0,i}))((v_{1,i} - v_{0,i}) \times (v_{2,i} - v_{0,i}))^T \\
c =& - w_{Volume} \sum_{i \in F} ((v_{0,i} \times v_{1,i})^T v_{2,i})((v_{1,i} - v_{0,i}) \times (v_{2,i} - v_{0,i}))
\end{aligned}
$$

- **Triangle Shape Optimization**: Es wird versucht, dass die entstehenden Dreiecke möglichst gleichseitig sind. Dieses Constraint wird nur als Notmaßnahme benutzt, wenn die *Volume* und *Boundary Optimization* Terme beide sehr nah an 0 liegen. In CGAL wird dieses *Constraint* nicht benutzt, da die Gewichtung dieses Constraint 0 ist. Es werden alle Vertices in der Umgebung

der Kante betrachtet. Diese entsprechen den Vertices der Faces, die bei *Volume Optimization* betrachtet werden.

$$V = \{\text{Indices der Vertices in der Umgebung}\}$$
$$H = \begin{bmatrix} |V| & 0 & 0 \\ 0 & |V| & 0 \\ 0 & 0 & |V| \end{bmatrix}$$
$$c = -\sum_{i \in V} v_i$$

Die *Constraints* werden in folgender Reihenfolge betrachtet. *Volume* und *Boundary Optimization Constraints* werden zusammen hinzugefügt, indem die beiden Matrizen H und die beiden Vektoren c vorher addiert werden und gewichtet sind.

#	*Constraint*	Anzahl an bisherigen *Constraints*
1	Volume Preservation	≤ 1
2	Boundary Preservation	≤ 2
3	Volume/Boundary Optimization	≤ 3
4	Triangle Shape Optimization	≤ 3

Tabelle 3.1: Die Reihenfolge, in der die *Constraints* angewendet werden [Siehe LT98, S. 5]

Um aus H und c ein oder mehrere *Constraints* (a_i, b_i) berechnen zu können, werden folgende Regeln verwendet, abhängig von der Anzahl der bisherigen *Constraints* n:

$\underline{n = 0:}$
$$a_i = H_i \text{ für } i \in \{1,2,3\}$$
$$b_i = -c_i \text{ für } i \in \{1,2,3\}$$
$\underline{n = 1:}$
$$Q(\begin{bmatrix} x \\ y \\ z \end{bmatrix}) = \begin{bmatrix} -\frac{z}{x} & 0 & 1 \\ 0 & -\frac{z}{y} & 1 \\ 1 & 0 & -\frac{x}{z} \end{bmatrix}$$
$$j = \text{Index des im Betrag maximalen Eintrags von } a_1$$
$$q_1 = Q(a_1)_j$$
$$q_2 = a_1 \times q_1$$
$$a_i = Hq_i \text{ für } i \in \{2,3\}$$
$$b_i = -(q_i^T c) \text{ für } i \in \{2,3\}$$
$\underline{n = 2:}$
$$a_3 = H(a_1 \times a_2)$$
$$b_3 = -((a_1 \times a_2)^T c)$$

Mit H_i bzw. Q_i wird die i-te Reihe der Matrix H bzw. Q und mit c_i der i-te Eintrag des Vektors c notiert. Ein *Constraint* kann aber dennoch nur verwendet werden, wenn er *α-compatible* zu den bisherigen *Constraints* ist.

3.3.2 Bestimmung der Kosten

Die Kosten C eines *Edge Collapse* der betrachteten Kante können erst über das vorher berechnete *Placement* bestimmt werden und sind eine gewichtete Kombination aus Form-, Rand- und Volumenkosten. v ist das vorher berechnete Placement des Vertex für den *Edge Collapse*.

Die einzelnen Kostentypen berechnen sich nach der CGAL Implementierung von *Lindstrom Turk* [siehe CGA18] wie folgt:

- ***Boundary* Kosten**:

$$C_B = \frac{1}{4} \sum_{i \in E} ((v_{0,i} - v_{1,i}) \times ((v_{0,i} \times v_{1,i}) - v)))^2$$

- ***Volume* Kosten**:

$$C_V = \frac{1}{36} \sum_{i \in F} (((v_{1,i} - v_{0,i}) \times (v_{2,i} - v_{0,i}))^T v - (v_{0,i} \times v_{1,i})^T v_{2,i})^2$$

- ***Shape* Kosten**:

$$C_S = \sum_{i \in V} (v_i - v)^2$$

Die Gesamtkosten einer Kante lassen sich somit bestimmen:

$$\begin{aligned} C = & \; w_{Volume} \cdot C_V + \\ & + w_{Boundary} \cdot C_B \cdot \|v_0 - v_1\|^2 + \\ & + w_{Shape} \cdot C_S \cdot \|v_0 - v_1\|^4 \\ \text{für } & w_{Volume} + w_{Boundary} + w_{Shape} = 1 \end{aligned}$$

Der Benutzer kann die Gewichtungen auch selbst festlegen über ein Parameter Objekt. Die Volumen und Boundary Gewichtung ist in CGAL $\frac{1}{2}$. Die Shape Gewichtung ist 0 und somit werden die Shape Kosten standardmäßig ignoriert, da sonst Kanten die gute Werte für *Volume* und *Boundary Optimization* haben, benachteiligt werden als potenzielle Kandidaten für den *Edge Collapse*.

3.4 Begrenzte Änderung der Normalen der Dreiecke

Da auch bei *Lindstrom Turk* nicht garantiert wird, dass sich keine Dreiecke überschneiden, können die Normalen der umliegenden Dreiecke vor und nach dem *Edge Collapse* betrachtet werden.

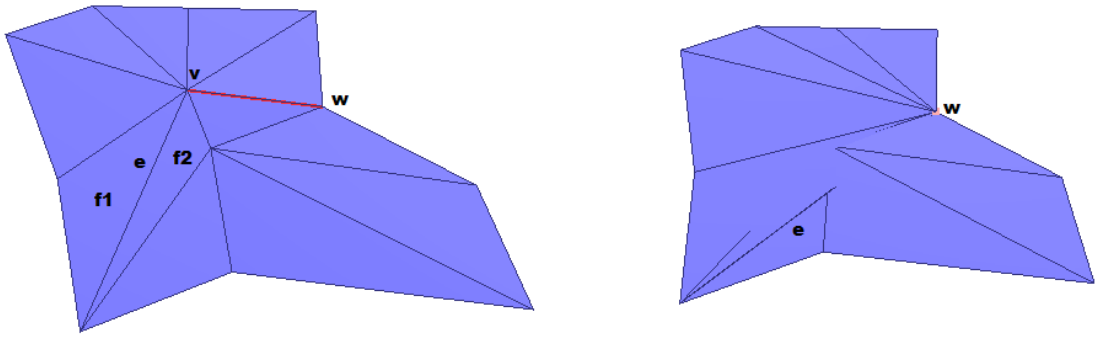


Abbildung 3.7: Ein Mesh vor und nach der Vereinfachung. Die Kante (v,w) wird nach w gezogen. Die Normalen von f1 und f2 sind fast gleich, aber nach dem *Edge Collapse* sind sie entgegengesetzt. [Siehe Cac18]

Dazu wird die Klasse `Bounded_normal_change_placement` als *Placement* eingesetzt, welche *Edge Collapses* ablehnt, wenn bei einem anderem *Placement* wie zum Beispiel *Lindstrom Turk* die Normalen nicht mehr in die selbe Richtung zeigen im Vergleich zu vor dem *Edge Collapse* bzw. der Winkel zwischen den beiden Normalen zwischen 90° und 180° liegt.

Falls das folgende Skalarprodukt negativ ist, ist der Winkel zwischen den Normalen eines Dreiecks vor und nach dem *Edge Collapse* zwischen 90° und 180°.

$$((v_{0,i} - v_{1,i}) \times (v_{2,i} - v_{1,i}))^T((v_{0,i} - v) \times (v_{2,i} - v)) < 0 \text{ für } i \in F$$

Dabei ist v die durch den anderen *Placement* Algorithmus bestimmte Position für den *Edge Collapse*, die Vertices $v_{0,i}$, $v_{1,i}$, $v_{2,i}$ sind die Vertex Punkte eines Dreiecks in der Umgebung der betrachteten Kante, wobei $v_{1,i}$ immer ein Vertex der betrachteten Kante ist. F ist so definiert wie in Formel 3.1.

3.5 Constrained Mesh Simplification

Um zu verhindern, dass bestimmte Vertices mit dem *Edge Collapse* Verfahren zusammengezogen werden, wird eine *Constrained Edges Map* verwendet. Diese Map ist ein `struct` welche eine `get` Methode besitzt, die für eine Kante `true` zurückliefert, falls diese Kante *constrained* ist und somit nicht vereinfacht werden darf. Der Vereinfachungsfunktion wird dann diese Map per *Named Parameter* übergeben. Somit werden alle Kanten festgelegt, die sicher im vereinfachten Mesh zu finden sind. Das *Placement* der Vertices, die zu diesen Kanten gehören, kann sich aber noch ändern. Um dies zu verhindern, wird die Klasse `Constrained_placement` verwendet, welches bei Kanten, die an *constrained* Edges angrenzen, das gemeinsame Vertex als *Placement* verwendet [vgl. Cac18]. Somit verändert sich die Position von *constrained* Edges nicht während dem Vereinfachen.

Es wurde die Farbunterscheidung nicht in die Kostenfunktion integriert. Mit dieser Änderung würden Kanten mit höheren Farbunterschied weiter unten in der *Priority Queue* landen als Kanten mit geringeren Unterschied. In dieser Implementierung werden alle Kanten durch die *Constrained edges map* komplett ignoriert, welche einen zu hohen Farbunterschied aufweisen.

3.5.1 Farbinformationen

Die Farbe einzelner Vertices bzw. Faces kann über die *Boost graph library property maps* `v:color` bzw. `f:color` der Surface Mesh Klasse abgerufen werden. Dabei werden diese in der *Polyhedron3* Demo nur dann befühlt, wenn als Plugin zum Einlesen der OFF Dateien mit Vertex bzw. Face Farben das `surface_mesh_io_plugin` Plugin ausgewählt ist.

Die Farben sind im RGB Format gespeichert und somit lassen sich der rote, grüne und blaue Farbanteil einfach abrufen. Wenn man die Vertex Farbe betrachtet, ist eine Kante (v_0, v_1) *constrained* falls:

$$(r_{v_0} - r_{v_1})^2 + (g_{v_0} - g_{v_1})^2 + (b_{v_0} - b_{v_1})^2 > t^2$$

Wobei r den roten, g den grünen und b den blauen Farbanteil der Vertices v_0 und v_1 beschreibt. t ist der vom Benutzer spezifizierte Grenzwert des Farbunterschieds, unter dem eine Kante nicht *constrained* ist. Analog dazu können die Face Farben verglichen werden, indem statt v_i das Face f_i betrachtet wird, auf das das Halfedge, auf das v_i zeigt, eine Referenz hat.
Um zu ermitteln welche Kanten *constrained* sind, wird folgender Algorithmus verwendet.

Algorithm 1: Der Algorithmus zur Bestimmung der *constrained edges*

Data: edge, threshold, use_vertex_color
Result: is_constrained
1 **if** *use_vertex_color*:
2 v_0 = first vertex of the edge
3 v_1 = second vertex of the edge
4 c_0 = v_0.color()
5 c_1 = v_1.color()
6 **else:**
7 h_0 = first halfedge of the edge
8 h_1 = second halfedge of the edge
9 f_0 = face of halfedge h_0
10 f_1 = face of halfedge h_1
11 c_0 = f_0.color()
12 c_1 = f_1.color()
13 red = c_0.red() - c_1.red()
14 green = c_0.green() - c_1.green()
15 blue = c_0.blue() - c_1.blue()
16 is_constrained = $(\text{red}^2 + \text{green}^2 + \text{blue}^2 > \text{threshold}^2)$

Die Funktionen red(), green() und blue() geben den entsprechenden Farbanteil einer Farbe von 0 bis 255 zurück. Die color() Funktion gibt die Farbe eines Vertex bzw. Faces zurück.

In der Software *Meshlab* ist es möglich aus der Textur die Farbe einzelner Vertices zu ermitteln. Unter `Filters > Texture > Transfer: Texture to Vertex`

`Color (1 or 2 meshes)` kann ein Dialog geöffnet werden, in dem das Quell Mesh mit einer Textur und das Ziel Mesh ausgewählt werden kann. Nach Bestätigen des Dialogs werden die Vertex Farben aus den umliegenden Textur Farben berechnet.

Falls das Mesh schon Vertex Farben enthält, kann auch für jedes Face eine Farbe mitgespeichert werden. Die Bestimmung einer Face Farbe aus den Vertex Farben des Faces wurde in das selbst erstellte Plugin integriert. Es wurde ein Algorithmus basierend auf den Vorschlag in [Ave15] implementiert. Es werden die RGB Farben erst quadriert, bevor der Durchschnitt gebildet wird, da somit die logarithmische Wahrnehmung von Farben besser berücksichtigt wird. Wenn dies nicht gemacht wird, werden die berechneten Farben dunkler als man intuitiv erwarten würde [vgl. Ave15]. Es wird davon ausgegangen, dass das Mesh nur aus Dreiecken besteht und man somit bei der Durchschnittsberechnung immer durch drei dividieren muss, da ein Face drei Vertices mit Farben besitzt. Der folgende Algorithmus wird automatisch dann aufgerufen, wenn keine Face Farben gefunden wurden, aber Vertex Farben zu Verfügung stehen, um eine Face Farbe zu approximieren.

Algorithm 2: Der Vertex to Face Color Algorithmus

Data: vcolor_map, fcolor_map
foreach *face in the mesh*:
  red = 0, green = 0, blue = 0
  foreach *vertex around the face*:
    vertex_color = vcolor_map[vertex]
    red += vertex_color.red()^2
    green += vertex_color.green()^2
    blue += vertex_color.blue()^2
  face_color = Color($\sqrt{\frac{\text{red}}{3}}$, $\sqrt{\frac{\text{green}}{3}}$, $\sqrt{\frac{\text{blue}}{3}}$)
  fcolor_map[face] = face_color

3.5.2 Der iterative Simplification Algorithmus

In diesem Abschnitt wird der Vereinfachungs Algorithmus beschrieben, der *constrainted* Edges nicht entfernt. Dabei werden mit v_0 und v_1 die beiden Vertices des gerade betrachteten Halfedge (v_0, v_1) bezeichnet.

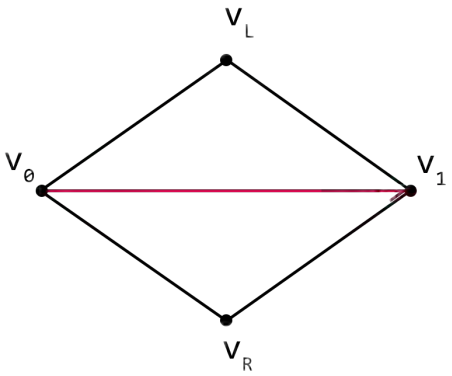


Abbildung 3.8: Das linke und rechte Face des Halfedge (v_0, v_1)

Der erste Schritt besteht aus dem Sammeln der Halfedges in der *Priority Queue* `PQ`. Die Kosten eines Halfedge werden in `PQ` mit der Referenz auf das Halfedge gespeichert. Das Halfedge mit den niedrigsten Kosten liegt oben auf `PQ`. Dabei werden gleich die Kanten mit der Länge 0 beseitigt, damit diese später nicht mehr behandelt werden müssen. Im zweiten Schritt wird `PQ` iterativ abgearbeitet und bei jedem Kandidat geprüft, ob eine *Collapse* Operation mit dem berechneten *Placement* valide ist. Nach jeder *Collapse* Operation werden die Kosten der benachbarten Halfedges aktualisiert. Dies geht solange bis das Stop Prädikat `Should_stop()` erfüllt ist und der Algorithmus terminiert.

Algorithm 3: Der Constrained Simplification Algorithmus

```
   Data: zero_length_list, PQ
 1 # Sammle Halfedges in der Priority Queue
 2 foreach non constrained halfedge in the mesh:
 3     if v_0 == v_1:
 4         Insert halfedge into zero_length_list
 5     else:
 6         cost = Get_cost(halfedge)
 7         Insert halfedge and cost into PQ with the order determined by cost
 8 # Entferne Kanten mit der Länge o aus dem Mesh
 9 foreach halfedge in zero_length_list:
10     Remove halfedges from the PQ which are incident and part of the
        degenerated triangle
11     Collapse the halfedge with v_0 as placement
12 # Verarbeite jedes Halfedge in der Priority Queue
13 while hEdge = pop_PQ():
14     if Should_stop():
15         # Stop Prädikat ist erfüllt
16         break
17     # Prüfe, ob eine nicht mannigfaltige Situation vorliegt
18     if is_collapse_topologically_valid(hEdge):
19         placement = Get_placement(hEdge)
20         # Prüfe, ob nach dem Edge Collapse der maximale Winkel zwischen den
            Normalen der Dreiecke einen Grenzwert nicht überschreitet
21         if is_collapse_geometrically_valid(hEdge, placement):
22             if left face of hEdge exists:
23                 Remove halfedge (v_L, v_0) from PQ or (v_1, v_L) if first halfedge
                    is constrained
24             if right face of hEdge exists:
25                 Remove halfedge (v_R, v_1) from PQ or (v_0, v_R) if first halfedge
                    is constrained
26             Collapse hEdge with placement as placement
27             # Update the cost and so the order in the PQ of all edges around each
                vertex adjacent to the new vertex
28             Update_neighbours(hEdge)
```

Der Algorithmus kann durch ein Konsolenfenster verfolgt werden, indem man die aktuelle Anzahl der gesammelten und noch abzuarbeiteten Kanten sehen kann. Nachdem genug Kanten entfernt wurden und der Algortihmus beendet wird, werden noch einige Statistiken zur Vereinfachung angezeigt, wie zum Beispiel wie viele Kanten als Kandidat für einen *Edge Collapse* in Betracht gezogen wurden, aber wegen anderen Kriterien des Algorithmus abgelehnt wurden.

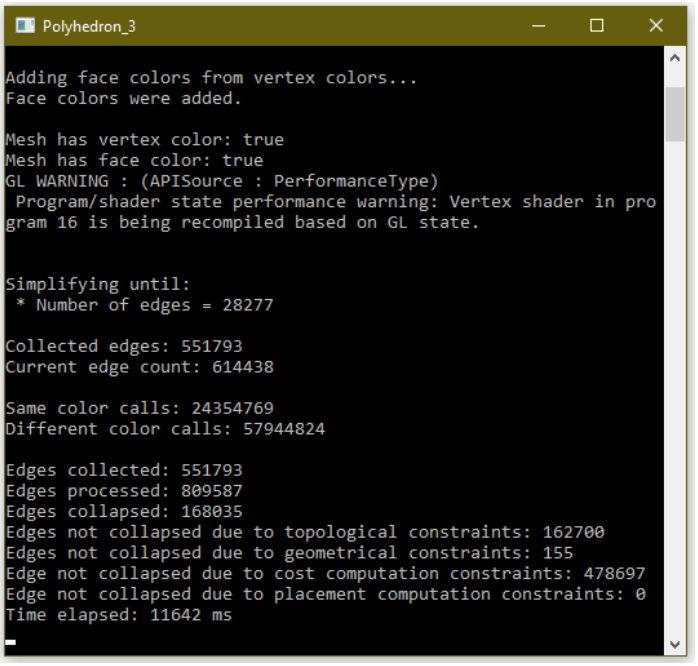


Abbildung 3.9: Das Konsolenfenster zeigt die Log Nachrichten der CGAL Polyhedron3 Demo.

Da diese Ausgabe, während der Algorithmus läuft, sehr langsam ist, werden nur immer nach 50000 abgearbeiteten Kanten die Ausgabe aktualisiert. Dies entspricht ungefähr einen Update pro Sekunde. Somit lässt sich auch erkennen, wenn der Algorithmus festhängt. Dies kann bedingt sein durch einen zu niedrigen Farb-Threshold und tritt nur bei manchen Meshes auf.

Kapitel 4

Besprechung der Ergebnisse

4.1 Informationen zu den Test Meshes

In diesem Kapitel wird der Algorithmus an zwei Meshes mit Farbinformationen getestet. Zu diesen Meshes gab es auch eine Textur (siehe Abbildung 4.1) und mit *Meshlab* wurden die Vertex Farben aus der Textur ermittelt, wie in Abschnitt 3.5.1 beschrieben.

(a) Merkur (b) Burgfräulein

Abbildung 4.1: Meshes mit Textur

Es wurden diese Meshes gewählt, da beim Merkur Mesh auf der Unterseite vier Zahlen zu sehen sind, welche sich nur über die Textur Farbe erkennen lassen und diese nicht in die Figur eingraviert worden sind. Beim Burgfräulein Mesh hat man im Gegensatz dazu mehr Farbunterschiede, über die sich auch die strukturelle Beschaffenheit des Meshes abhebt.

Mesh Name	Edges	Vertices	Faces
Merkur	6565575	2188551	4377050
Burgfräulein	5058749	1687230	3371790

Tabelle 4.1: Informationen zu den Meshes

4.2 Laufzeitunterschiede

Um die Laufzeitunterschiede für die verschiedenen Kosten- und Platzierungsstrategien zu ermitteln, wurden die Zeiten von jeweils drei Durchläufen gemittelt. Wenn das *Bounded Normal Change Placement* (siehe Abschnitt 3.4) vor dem eigentlichen Placement geschaltet wird, erhöht sich die Zeit um durchschnittlich 9,3% bei Lindstrom Turk und um 17,3% bei *Edge Length* Kosten mit *Midpoint Placement*. Lindstrom Turk zu verwenden ist hier im Schnitt 71% zeitaufwändiger als *Edge Length* Kosten mit *Midpoint Placement* zu verwenden. Das Sammeln der Halfedges in der *Priority Queue* und somit das einmalige Berechnen der Kosten für jedes Halfedge dauert in jedem Fall nur wenige Sekunden in dieser Mesh Komplexitätsklasse. Die meiste Zeit wird damit verbracht, nach einem *Edge Collapse* die Kosten der umliegenden Halfedges zu aktualisieren.

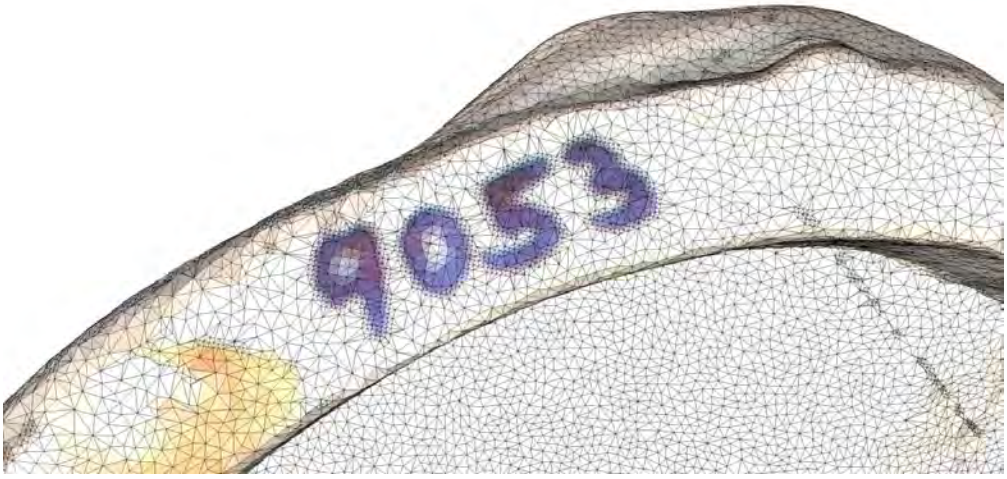
Mesh	Edges	Edges	Lindstrom Turk		Edge Length	
Name	vorher	nachher	mit BNC	ohne BNC	mit BNC	ohne BNC
Merkur	6565575	656557	110481	100599	68207	57230
Burgfräulein	5058749	505874	87713	80592	52194	45173

Tabelle 4.2: Laufzeit des Algorithmus für verschiedene Kosten- und Platzierungsstrategien mit und ohne *Bounded Normal Change Placement* in Millisekunden. Die Anzahl der Edges wird um 90% verringert.

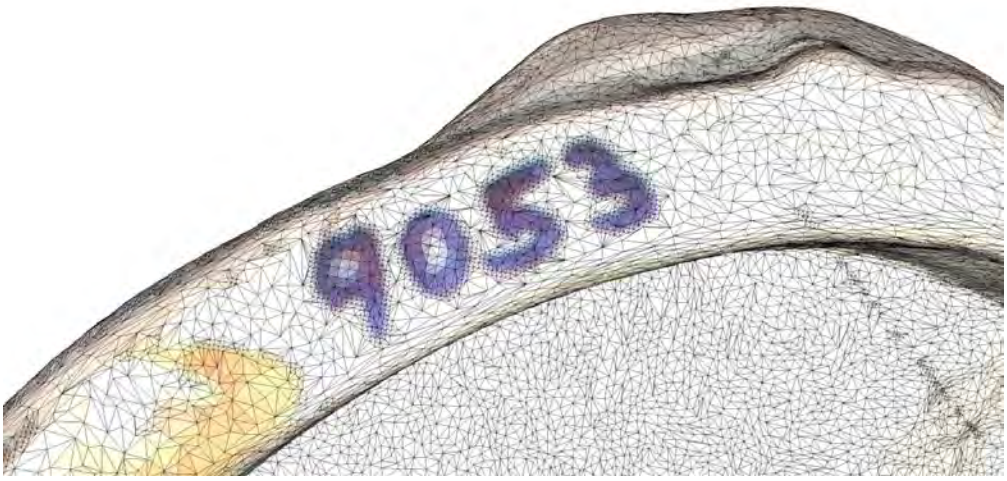
4.3 Visueller Vergleich

Beim visuellen Vergleich der beiden Strategien lassen sich in diesem Fall bei den äußeren beiden Zahlen im Merkur Mesh Unterschiede feststellen. Mehr spitzere Dreiecke sind bei Abbildung 4.2c als bei 4.2b entstanden, da mit der Standardgewichtung der verschiedenen Kosten bei *Lindstrom Turk* die Dreiecksform ignoriert wird und nur das Volumen und der Rand optimiert wird (siehe Abschnitt 3.3.2). Bei Abbildung 4.3 lässt sich dies auch besonders bei den Fingern beobachten. Beim Übergang zwischen den Zahlen im Merkur Mesh kann man eine erhöhte Anzahl an Faces nach der Vereinfachung feststellen. Die Farbübergänge im Burgfräulein Mesh sind aber nicht mit einer solchen Dichte an Faces zu erkennen. Bei der Verwendung von Face Farben anstatt Vertex Farben (siehe Abbildung 4.4) sind bei 90 prozentiger Ausdünnung nur minimale Unterschiede zu erkennen.

(a) Original

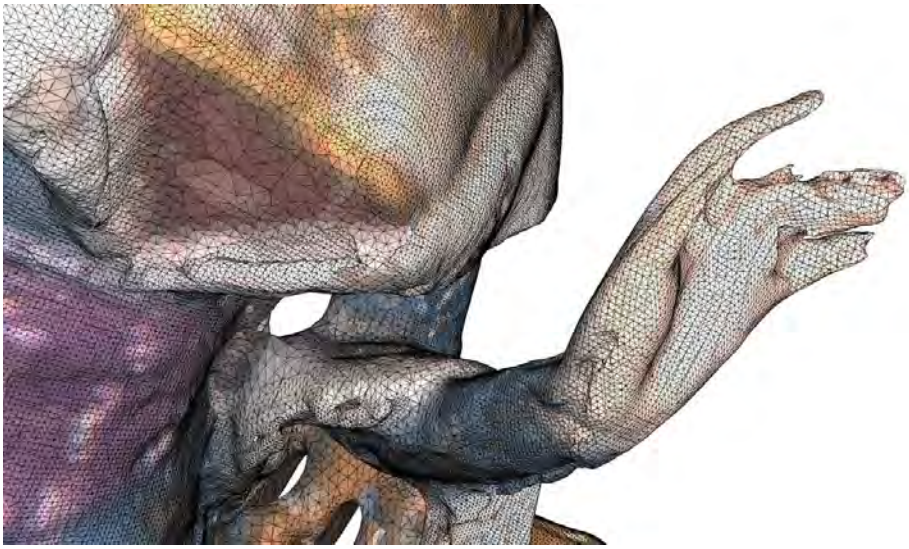


(b) Ausgedünnt mit *Edge Length* Kosten und *Midpoint Placement*

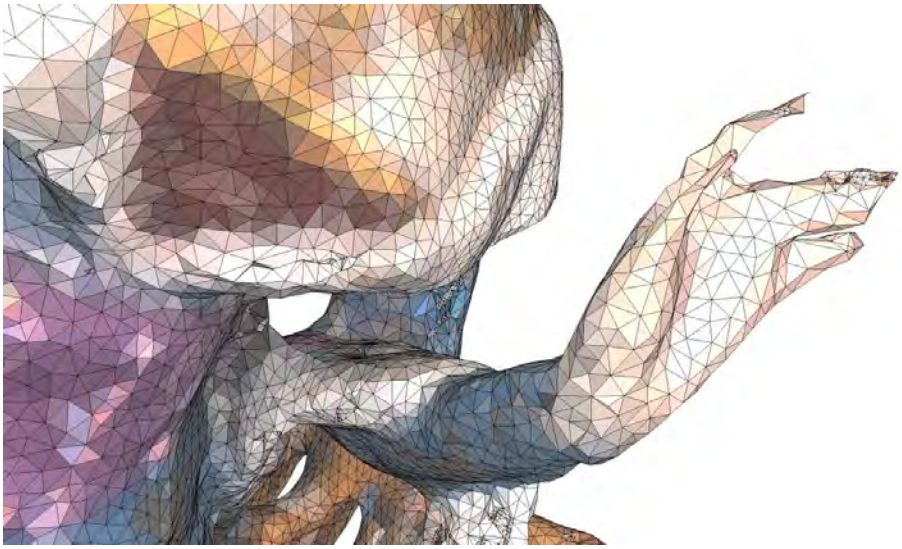


(c) Ausgedünnt mit Lindstrom Turk Kosten und Platzierung

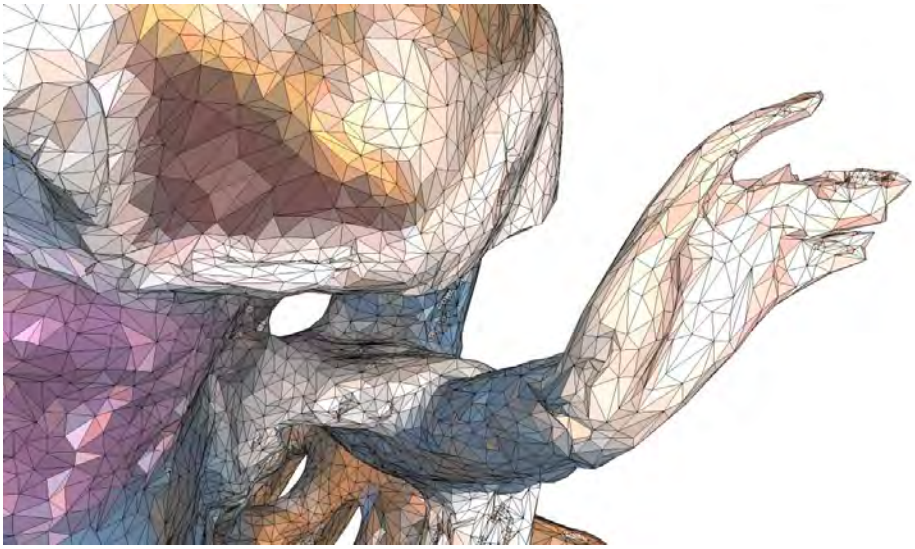
Abbildung 4.2: Unterseite vom Merkur Mesh nach Reduzierung der Edges um 90% mit Vertex Farbunterschiedstoleranz von 50



(a) Original



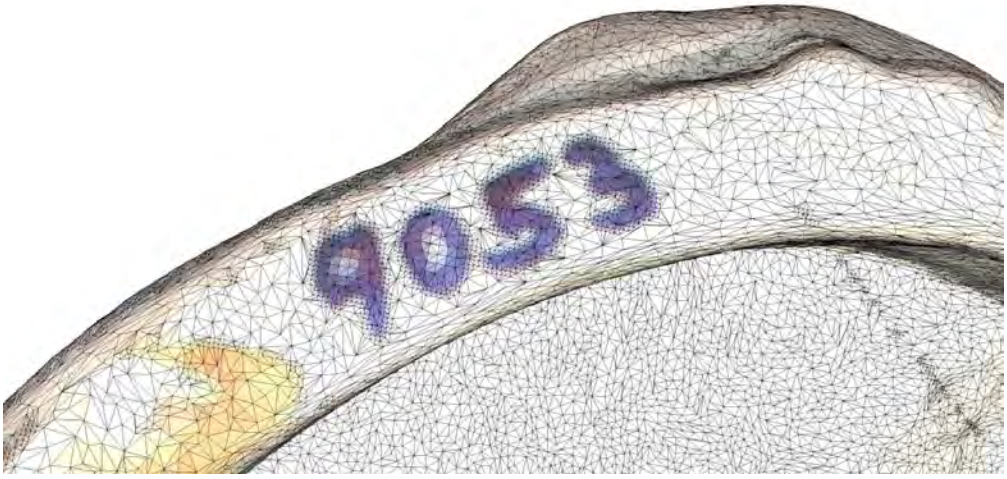
(b) Ausgedünnt mit *Edge Length* Kosten und *Midpoint Placement*



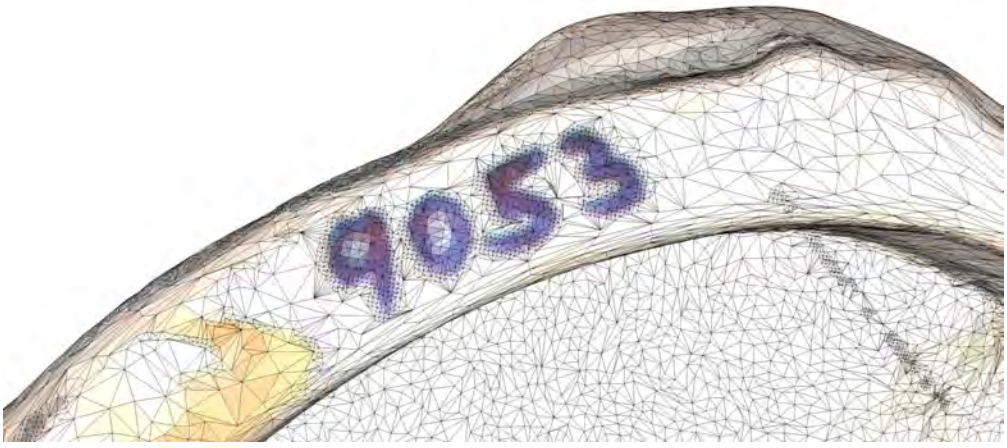
(c) Ausgedünnt mit Lindstrom Turk Kosten und Platzierung

Abbildung 4.3: Vorderseite vom Burgfräulein Mesh nach Reduzierung der Edges um 90% mit Vertex Farbunterschiedstoleranz von 150

(a) Original



(b) Ausgedünnt mit Lindstrom Turk Kosten und Platzierung mit Vertex Farben



(c) Ausgedünnt mit Lindstrom Turk Kosten und Platzierung mit Face Farben

Abbildung 4.4: Unterseite vom Merkur Mesh nach Reduzierung der Edges um 90% mit Vertex Farbunterschiedstoleranz von 50

Kapitel 5

Fazit

Durch den Algorithmus, der in Form eines Plugins für die CGAL *Polyhedron3* Demo realisiert wurde, lassen sich Meshes in relativ kurzer Zeit ausdünnen. Dabei ist die Qualität des entstehenden Meshes auch gut, solange man keine zu kleine Anzahl an Kanten, bis zu der vereinfacht wird, festlegt. Bei Meshes mit vielen Farbübergängen, wie beim Burgfräulein Mesh, kann der Algorithmus festhängen, wenn eine zu kleine Anzahl an Kanten gewählt wird, da die Farbübergänge irgendwann keine weiteren *Edge Collapses* mehr zulassen. Dies erfordert ein lockereres Limit für den maximal zulässigen Farbunterschied bei dem eine Kante nicht *constrained* ist.

Außerdem unterstützt CGAL seit der Version 4.14 auch Surface Mesh Approximation, welches Variational Shape Approximation [siehe Bot+10, S. 122 ff.] benutzt, um ein Mesh zu vereinfachen. Bei der Erstellung dieser Arbeit wurde bisher nur Surface Mesh Simplification mithilfe von Incremental Mesh Decimation in CGAL unterstützt. Um diesen Algorithmus zu nutzen, müssten aber sehr viele Änderungen am Code vorgenommen werden.

Literatur

[Ave15] Average colors. *What is the best way to average two colors that define a linear gradient?* https://stackoverflow.com/a/29576746/5355871. [Onlinezugriff 11.10.2018]. 2015.

[Bot+10] Mario Botsch u. a. *Polygon Mesh Processing*. A K Peters/CRC Press, 2010.

[Cac18] Fernando Cacciola. „Triangulated Surface Mesh Simplification". In: *CGAL User and Reference Manual*. 4.13. [Onlinezugriff 04.10.2018]. CGAL Editorial Board, 2018. URL: https://doc.cgal.org/latest/Surface_mesh_simplification/index.html.

[CGA18] CGAL. *Lindstrom Turk Implementierung*. https://github.com/CGAL/cgal/tree/releases/CGAL-4.13-branch/Surface_mesh_simplification/include/CGAL/Surface_mesh_simplification/Policies/Edge_collapse/Detail. [Onlinezugriff 23.03.2019]. 2018.

[HKS18] Susan Hert, Menelaos Karavelas und Stefan Schirra. „CGAL Developer Manual Introduction". In: *CGAL User and Reference Manual*. 4.13. [Onlinezugriff 19.11.2018]. CGAL Editorial Board, 2018. URL: https://doc.cgal.org/latest/Manual/devman_intro.html.

[Ket18] Lutz Kettner. „Halfedge Data Structures". In: *CGAL User and Reference Manual*. 4.13. [Onlinezugriff 17.11.2018]. CGAL Editorial Board, 2018. URL: https://doc.cgal.org/latest/HalfedgeDS/index.html.

[LT98] Peter Lindstrom und Greg Turk. *Fast and Memory Efficient Polygonal Simplification*. https://www.cc.gatech.edu/~turk/my_papers/memless_vis98.pdf. [Onlinezugriff 15.10.2018]. 1998.

[Man09] Mannigfaltigkeit. *Mesh*. https://wiki.delphigl.com/index.php/Mesh. [Onlinezugriff 14.12.2018]. 2009.

[Str06] Ian Stroud. *Boundary Representation Modelling Techniques*. Springer, London, 2006. Kap. 4, S. 83–92.

Erklärung

Ich erkläre hiermit eidesstattlich, dass ich die vorliegende Arbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Passau, 11. Juni 2019

Maximilian Röhrl