Tesselierung von Oberflächen

Studienarbeit

Vorgelegt von
Hendrik Wiebel

Institut für Computervisualistik
Arbeitsgruppe Computergraphik

Betreuer und Prüfer:
Prof. Dr.-Ing. Stefan Müller

Juni 2004

Inhaltsverzeichnis

1 Einleitung **2**

2 Aufbereitung der Eingabedaten **2**

3 Der *DelaunayWall (DeWall)* Algorithmus **3**
3.1 Das *DelaunayKriterium* 3
3.2 Grundprinzip des *DeWall* Algorithmus 3
3.3 Simplexgenerierung 4
3.4 Optimierung durch das *UniformGrid* 6
3.5 Der *DeWall* Algorithmus im Detail 7

4 Rekonstruktion durch γ-*Indikatoren* **8**
4.1 Der γ-*Indikator* 8
4.2 Löschen von Simplices 12
4.3 Der *Constriction* Algorithmus 13

5 Implementierung **14**
5.1 Framework 14
5.2 Elementare Datenstrukturen 15
5.3 Importieren und Aufbereiten von Daten 16
5.4 Der *DeWall* Algorithmus 18
5.5 Optimierung durch das *UniformGrid* 22
5.6 Die *WingedEdge* Datenstruktur 23
5.7 Der *Constriction* Algorithmus 24
5.8 Smoothing 27

6 Fazit **28**

1 Einleitung

Die folgenden Ausführungen befassen sich mit der Behandlung eines immer noch recht aktuellen Problems. In vielen Bereichen des täglichen Lebens spielen computergestützte Repräsentationen von Strukturen aus der realen Welt eine wichtige Rolle. Ein spezieller Fall ist dabei die Abtastung der Oberfläche eines Objektes und dessen anschliessende Rekonstruktion sowie dreidimensionale Präsentation durch den Rechner. Dabei wird durch die (meist optische) Abtastung eine Menge von Schnittpunkten mit dem Objekt zurückgeliefert. Zwischen diesen Punkten werden Flächen (Polygone) aufgespannt und evtl. mit photographischen Texturen versehen, um einen möglichst originalgetreuen Eindruck zu erhalten. Entscheidend ist natürlich, zwischen welchen Punkten man welche Fläche erzeugt. Oft wird als Grundelement das Dreieck gewählt, man spricht dann von einer Triangulierung. Eine Triangulierung lässt sich als Graph verstehen, der sich aus den Punkten und ihren Beziehungen untereinander zusammensetzt. Die Schwierigkeit bei der Rekonstruktion liegt darin, die richtigen Beziehungen herzustellen, d.h. nur die Flächen zu erzeugen, welche die ursprünglichen Formen approximiert. Im idealen Fall sind die Daten gleichmässig und sehr dicht gesampelt. Dann können relativ einfach die korrekten Flächen rekonstruiert werden, weil man die Seitenlängen der zu erzeugenden Dreiecke bereits vorher kennt. Nach dem Abtasttheorem muss die Abtastrate grösser als die doppelte Grenzfrequenz sein, um ein kontinuierliches Signal verlustfrei zu rekonstruieren. Dementsprechend sollte die Samplingdichte hoch genug sein, um alle Variationen in der Oberfläche, die von Interesse sind, zu erfassen. In vielen Fällen liegt jedoch keine gleichmässige Abtastung vor, so dass man sich bei der Auswahl geeigneter Flächen kaum auf absolute Grössen verlassen kann. Es gilt dann andere Auswahlverfahren zu finden, die eine Selektion z.B. mit Hilfe von Winkelbeziehungen vornehmen. Den Schwerpunkt dieser Studienarbeit macht die Verarbeitung solcher ungleichmässig gestreuter Datensätze aus. Dabei wird auf alle Prozesse von der Datenaufbereitung bis zur Visualisierung des Objektes eingegangen.

2 Aufbereitung der Eingabedaten

Gegeben ist eine Liste mit Punkten, die jeweils durch ihre x-, y- und z-Koordinate festgelegt sind. Die Ausdehnung und Position der Punktewolke ist bei jedem Datensatz unterschiedlich und es bietet sich an, vorher eine Normalisierung durchzuführen. Dabei werden zunächst die auftretenden Extrema für alle Koordinatenachsen ermittelt, um danach die Wolke in den Koordinatenursprung zu verschieben. Das Volumen wird im Anschluss daran so skaliert, dass seine längste Achse einen vorher festgelegten Wert (z.B. 1) annimmt. Somit liegen alle Koordinaten innerhalb eines bekannten Wertebereichs. Punkte, die mehrfach auftreten oder zu nah beieinander liegen, sollten zu einem einzigen zusammengefasst werden, um spätere Inkonsistenzen und unnötigen Mehraufwand zu vermeiden. Unter "zu nah" versteht man einen Abstand, der um ein Vielfaches kleiner ist als die durchschnittliche Entfernung zwischen zwei nächsten Nachbarn und damit nicht zu einer Erhöhung des Detailgrades beiträgt. Je nach Häufigkeit und Umfang dieser Ansammlungen werden eine Reihe von sehr kleinen Polygonen erzeugt, die man zwar nicht wahrnimmt, aber trotzdem Performanz kosten.

Um Mehrdeutigkeiten bei der Triangulierung im Vornherein zu verhindern, müssen

die Daten einem weiteren Prozess unterworfen werden. Es wird an späterer Stelle auf das sogenannte $DelaunayKriterium$ eingegangen, nach welchem man aus 4 Punkten einen Tetraeder konstruiert, falls sie alle auf der Oberfläche einer imaginären Kugel liegen und diese Kugel leer ist. Es kann jedoch sein, dass 5 oder mehr Punkte beteiligt sind und sich damit alle gleich gut für die Generierung eines solchen Tetraeders eignen. Man spricht dann von degenerierten Fällen, die besonders bei regelmässigen Objekten mit Symmetrieeigenschaften auftreten. In der vorliegenden Implementierung hat die Nichtbeachtung dieser Fälle zur Folge gehabt, dass der *DelaunayWall* Algorithmus nicht terminierte. [1] stellt ein Verfahren zum Beseitigen von Degenerierungen vor. Grundlage dafür ist die Annahme, dass ein degenerierter Datensatz zu einem nicht-degenerierten umgewandelt werden kann, indem man die Werte minimal verändert. Diese Veränderung darf natürlich nicht ihre Beziehung untereinander stören. Ausserdem wird diese Perturbation nur symbolisch durchgeführt und für die weitere Verwendung die originalen Belegungen beibehalten.

3 Der *DelaunayWall (DeWall)* Algorithmus

3.1 Das *DelaunayKriterium*

Man geht zunächst von einer Punktemenge P in einem d-dimensionalen Raum E^d aus. Ein *k-Simplex*, $k <= d$, ist dann definiert als konvexe Kombination von $k+1$ nicht colinearen Punkten in P. Sie nennen sich Vertices (oder Eckpunkte) des Simplex. In der Graphentheorie ist "Vertex" auch ein Synonym für "Knoten". Unter einem *s-Face* versteht man die Untermenge von $s+1$ Punkten eines Simplex. Ein *2-Simplex* entspricht folglich einem Dreieck und der *3-Simplex* wird durch einen Tetraeder repräsentiert. Jede Seitenfläche des Tetraeders ist ausserdem ein *2-Face*, jede Kante ein *1-Face* und jeder Punkt ein *0-Face*. Eine Triangulierung Σ auf einer Punktemenge P in E^d wird definiert als Menge von *d-Simplices* für welche laut [2] folgende Bedingungen gelten:

1. Ein Punkt p in E^d ist Vertex eines Simplex σ in Σ, falls $p \in P$
2. Die Schnittmenge zwischen 2 Simplices in Σ ist entweder leer oder ein gemeinsames Face
3. Die Menge Σ ist maximal, d.h. es kann kein weiterer Simplex σ hinzugefügt werden, ohne eine der vorherigen Regeln zu verletzen

Eine *Delaunay Triangulierung (DT)* zeichnet sich dadurch aus, dass die Hyperkugel, welche jeden Simplex umgibt, keine Punkte aus der Menge P enthält. Die *DT* ist nur dann eindeutig, wenn niemals mehr als $d+2$ Vertices auf einer Hyperkugel liegen. Es handelt sich ansonsten um die bereits in Abschnitt 2 angesprochenen Degenerierungen.

3.2 Grundprinzip des *DeWall* Algorithmus

Es gibt eine Reihe von unterschiedlichen Ansätzen, um eine *DT* auf einer Punktemenge P durchzuführen. Der Gedanke von *DeWall* beruht auf dem sogenannten *Divide &*

Conquer (D&C) Paradigma. Dabei wird die Punktemenge rekursiv aufgesplittet und lokal trianguliert. Anschliessend werden die Teilmengen zusammengefügt, wodurch man Σ erhält. Mit dieser einfachen Form des *D&C* lässt sich eine *DT* allerdings nur für den zweidimensionalen Fall effizient lösen. Für die Umsetzung in beliebigen Dimensionen müssen die Phasen von Aufsplittung und Vereinigung differenzierter erfolgen. Bei *DeWall* wird deshalb zuerst der Bereich trianguliert, welcher die beiden Teilmengen verbindet. Man nennt jene Region auch *SimplexWall*. Dieses Prinzip wendet man rekursiv auf die Teilmengen an bis Σ maximal wird. Die Aufteilung von P erfolgt mittels einer Schnittebene α, welche die Punktemenge in P_1 und P_2 aufteilt. Ausserdem wird durch sie die Triangulierung Σ in drei disjunkte Teilmengen zerlegt. Σ_1 und Σ_1 liegen jeweils komplett in einem der beiden Halbräume, die durch α definiert werden. Die dritte Teilmenge Σ_α besteht aus den Simplices, die α schneidet. Ihre Vertices befinden sich dementsprechend in beiden Halbräumen und damit sowohl in P_1 als auch in P_2. Grob zusammengefasst besteht der *DeWall* Algorithmus aus folgenden Schritten:

- Wähle die Schnittebene α, teile P in zwei Teilmengen P_1, P_2 und erzeuge Σ_α
- Führe den ersten Schritt, ausgehend von Σ_α, rekursiv auf den Teilmengen P_1, P_2 durch, um Σ_1 und Σ_2 zu erzeugen
- Gib die Vereinigung von Σ_1, Σ_2 und Σ_α zurück

Bisher wurde der *DeWall* Algorithmus für beliebige Dimensionen erläutert. Für die zugrundeliegende Aufgabenstellung ist nur die Betrachtung des dreidimensionalen Falles (E^3) notwendig. Zur Vereinfachung werden statt $3-Simplex$ und $2-Face$ die Begriffe Simplex und Face verwendet. Nimmt man an, dass der erste Tetraeder der *SimplexWall* Σ_α bereits existiert (näheres dazu in Abschnitt 3.3), so werden folglich mehrere seiner Flächen bzw. Faces von α geschnitten. Ausgehend von diesen wird im vorderen Halbraum nach geeigneten Punkten gesucht, um weitere Tetraeder von Σ_α zu erzeugen. Einige Faces gehen nicht durch α, also liegen ihre Vertices komplett in P_1 oder P_2. Diese Faces werden frühestens in der nächsten Rekursion für die Simplexgenerierung herangezogen, weil diese Tetraeder zum aktuellen Zeitpunkt keine Elemente von Σ_α wären.

Deshalb werden entstandene Faces klassifiziert und in 3 verschiedene Listen, sogenannte *ActiveFaceLists (AFL)*, eingetragen. In AFL_1 und AFL_2 kommen also diejenigen Faces, die sich am Rande der *SimplexWall* befinden. Die genaue Zuordnung ist abhängig vom jeweiligen Halbraum. AFL_α ist dagegen nur für Faces vorgesehen, die von α geschnitten werden. Auf diese Art und Weise wird garantiert, dass in jeder Rekursion nur die SimplexWall generiert wird. AFL_1 und AFL_2 werden bei den nächsten rekursiven Funktionsaufrufen als Parameter AFL übergeben und dort neu klassifiziert, bevor man weitere Simplices erzeugt. Beim ersten Aufruf ist AFL logischerweise leer. Es bleibt zu klären, wie die Konstruktion eines Simplex vonstatten geht.

3.3 Simplexgenerierung

Zunächst soll die Erzeugung des ersten Simplex erläutert werden, der sich von den nachfolgenden etwas unterscheidet. Zu Anfang existieren keinerlei Faces, die als Grundlage für den Aufbau eines Tetraeders dienen könnten. Deshalb muss zuerst ein Dreieck konstruiert werden. Dazu wird der Punkt $p_1 \in P$ gesucht, dessen Entfernung zur Schnittebe-

ne α am geringsten ist. Ausserdem sucht man auf der anderen Seite von α ein $p_2 \in P$, für den das gleiche gilt. Nun wird p_3 so gewählt, dass der Umkreis zwischen allen dreien minimal wird (Siehe Berechnungen 1 und 2). Die anschliessende Prozedur ist für den ersten wie auch für jeden weiteren Simplex ähnlich. Ausgehend vom Face f eines Simplex wird in dessen vorderen Halbraum nach einem Punkt p_4 gesucht, um den benachbarten Tetraeder zu erzeugen. p_4 wählt man so, dass der *Delaunayabstand (DelaunayDistance dd)* der dadurch definierten Umkugel minimal wird. Für die Generierung des ersten Simplex dehnt sich die Suche nach geeigneten Punkten auf beide Halbräume aus, weil f anfangs zu keinem Simplex gehört, durch den eine Vorderseite festgelegt wäre. Der *Delaunayabstand* wird für die Suche nach dem besten p_4 folgendermaßen ermittelt:

$$dd(f,p) = \begin{cases} r & \text{falls } c \in \text{Halfspace(f,p)} \\ -r & \text{sonst} \end{cases}$$

Befindet sich das Zentrum c der Umkugel auf der gleichen Seite wie p, so sind *Delaunayabstand* und Radius identisch. Sobald c aber auf der anderen Seite von f liegt, wird als *Delaunayabstand* der negative Radius gesetzt. Somit werden teilweise Punkte bevorzugt, die zwar enorme Umkugelradien erzwingen, aber durch die Negation einen sehr kleinen *Delaunayabstand* mit sich bringen. Da in diesem Fall das Kugelzentrum nicht im Halbraum von p_4 liegt, befindet sich ein Grossteil des Kugelvolumens in einer Region, in welcher keine relevanten Punkte zu erwarten sind. Nur der Bereich der Kugelkappe ist von Bedeutung. Diese wird mit zunehmend negativem Radius immer flacher, d.h. auch der Simplex in ihr flacht ab. An späterer Stelle im Text wird von den Funktionen *makeSimplex* und *makeFirstSimplex* die Rede sein. In ihnen sind die Operationen zur Simplexgenerierung gekapselt. Existieren im vorderen Halbraum von Face f keine weiteren Punkte, die zur Simplexkonstruktion herangezogen werden können, so ist f ein Bestandteil der konvexen Hülle des Objektes. In diesem Fall gibt *makeSimplex null* zurück.

Die nachfolgenden Formeln dienen zum Berechnen von Radius r und Mittelpunkt c eines Kreises, der durch drei Punkte a, b, c, ($a, b, c \in E^3$) eindeutig festgelegt ist. $|a|$ steht dabei für die euklidische Norm von a. $a \times b$ ist das Kreuzprodukt der Vektoren a, b.

$$r = \frac{\Big||c-a|^2\Big((b-a)\times(c-a)\Big)\times(b-a) \;+\; |b-a|^2(c-a)\times\Big((b-a)\times(c-a)\Big)\Big|}{2|(b-a)\times(c-a)|^2} \tag{1}$$

$$c = a + \frac{|c-a|^2\Big((b-a)\times(c-a)\Big)\times(b-a) \;+\; |b-a|^2(c-a)\times\Big((b-a)\times(c-a)\Big)}{2|(b-a)\times(c-a)|^2} \tag{2}$$

Für die Berechnungen zur Umkugel der Punkte a, b, c und d, ($a, b, c, d \in E^3$) existieren folgende analoge Berechnungen. ax, ay und az stellen die Komponenten von a dar.

$$r = \frac{\left|\,|d-a|^2\Big((b-a)\times(c-a)\Big) \;+\; |c-a|^2\Big((d-a)\times(b-a)\Big) \;+\; |b-a|^2\Big((c-a)\times(d-a)\Big)\,\right|}{2\begin{vmatrix} bx-ax & by-ay & bz-az \\ cx-ax & cy-ay & cz-az \\ dx-ax & dy-ay & dz-az \end{vmatrix}} \tag{3}$$

$$c = a + \frac{|d-a|^2\Big((b-a)\times(c-a)\Big) \;+\; |c-a|^2\Big((d-a)\times(b-a)\Big) \;+\; |b-a|^2\Big((c-a)\times(d-a)\Big)}{2\begin{vmatrix} bx-ax & by-ay & bz-az \\ cx-ax & cy-ay & cz-az \\ dx-ax & dy-ay & dz-az \end{vmatrix}} \tag{4}$$

Diese Formeln sind unter dem Gesichtspunkt der numerischen Stabilität nicht optimal, falls der Nenner annähernd 0 wird. Jedoch treten diese Fälle nur auf, wenn es sich um nahezu degenerierte Simplices handelt, d.h. die Vertices (fast) in einer Ebene liegen. Es existieren sicherere Methoden (siehe [4]), die jedoch mit einem enormen Mehraufwand einhergehen.

3.4 Optimierung durch das *UniformGrid*

Die Berechnungen zu Umkugel,Umkreis innerhalb der *makeSimplex* Funktion sind aufgrund etlicher Vektoroperationen ziemlich aufwändig. Hinzu kommt, dass diese Aufrufe sehr oft erfolgen, da für jedes Face f alle potentiellen Vertices (im vorderen Halbraum) der aktuellen Punktemenge P geprüft werden müssen. Auch wenn P mit jeder Rekursionsstufe kleiner wird, erweisst sich der *DeWall* Algorithmus in seiner naiven Implementierung als nicht besonders performant. Mit Hilfe einer zusätzlichen Datenstruktur lässt sich die Effizienz jedoch um ein Vielfaches steigern. Es handelt sich dabei um ein dreidimensionales Raster, das sog. *Uniform Grid (UG)*, welches über die Punktewolke gelegt wird. Jedem Feld werden die Punkte zugeordnet, die in ihm liegen. Somit ist es möglich, spezifisch in bestimmten Regionen nach Punkten zu suchen. Für *makeSimplex* bieten sich dadurch Optimierungsmöglichkeiten. Nehmen wir an, dass für f ein potentieller Vertex p_4 zur Bildung eines Simplex gefunden wurde. Durch p_4 wird Mittelpunkt c und Radius r einer Umkugel festgelegt.

Alle weiteren Punkte, die sich ausserhalb dieser Umkugel befinden, kommen nicht mehr als alternativer p_4 in Frage, da der resultierende *Delaunayabstand* grösser wäre. Sie müssen deshalb auch nicht überprüft werden. Die *makeSimplex* Funktion wird jetzt so gestaltet, dass mit der Punktesuche im Zentrum von f begonnen wird. Die Zellen des *UG* werden werden um den Startpunkt herum abgearbeitet und der Suchradius erweitert bis ein erster geeigneter Vertex gefunden wird. Anschliessend setzt sich die Suche fort, wobei man sich auf die Zellen beschränkt, welche die Umkugel überlagern. Jeder neue Kandidat schränkt den Suchraum weiter ein, bis alle seine Zellen auf ihre Punkte geprüft wurden. Die Anzahl der gesparten Überprüfungen ist abhängig von der Auflösung des *UG*. Eine zu kleine Auflösung macht das Prinzip des *UG* zunichte, da dann kaum lokal gesucht werden kann. Wird die Auflösung höher, steigt auch der Verwaltungsaufwand. Die korrekte Berechnung der Zellen, die eine Kugel überlagern, stellt einen zusätzlichen Implementierungsaufwand dar. Es ist einfacher, die komplette Boundingbox der Kugel einzubeziehen, obwohl dadurch natürlich mehr Punkte als notwendig getestet werden.

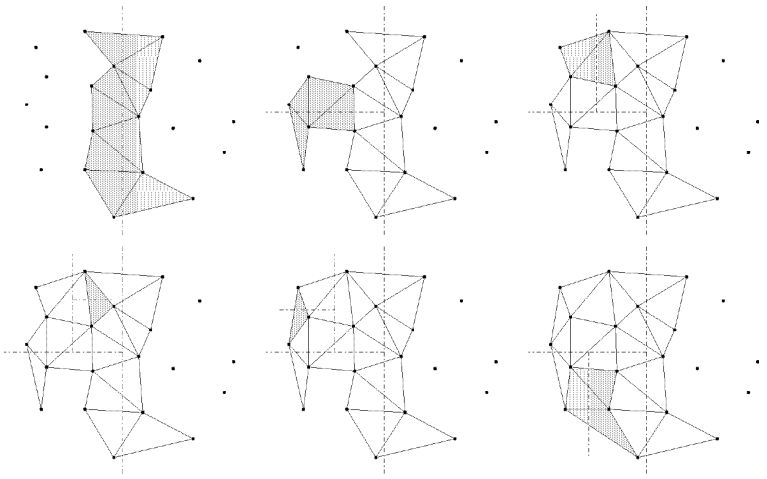

Abbildung 1: einige Schritte des *DeWall* Algorithmus für eine Punktemenge in $2D$ [2]

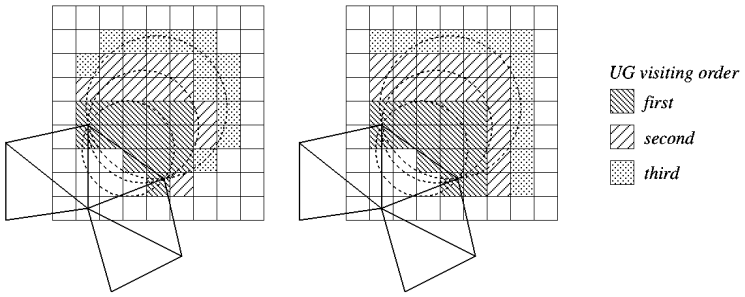

Abbildung 2: Abarbeitungsreihenfolge im *UniformGrid* [2]

3.5 Der *DeWall* Algorithmus im Detail

Der folgende Pseudocode wurde aus [2] übernommen und stellt die Funktionsweise des *DeWall* Algorithmus dar.

Die Funktion *DeWall* besitzt 2 Parameter: Die Punktemenge P sowie die *FaceList* AFL. Beim ersten Aufruf umfasst P die Gesamtheit aller Punkte und AFL ist leer. In diesem Fall wird durch die Funktion *makeFirstSimplex* der erste Simplex t der Triangulierung Σ erzeugt und seine Faces in AFL eingetragen. Alle anderen Schritte sind für jeden weiteren Aufruf identisch. Die Faces von AFL werden einem Schnittest mit der Ebene α unterzogen und bei Erfolg in der Liste AFL_α vermerkt. α wird zu Beginn übrigens so gewählt, dass sie parallel zur xy-, yz-, oder xz-Ebene liegt und P gleichmässig aufteilt. In diesem Schritt wird P in P_1 und P_2 partitioniert. Bei Misserfolg des Tests wird das jeweilige Face f den Listen AFL_1, bzw. AFL_2 zugeordnet. Die Zuordnung hängt davon ab, in welcher Teilmenge von P sich die Vertices von f befinden. Im Anschluss daran wird für jedes Face von AFL_α *makeSimplex* aufgerufen. In P startet dann die Suche nach Punkten zur Simplexkonstruktion. Jeder erfolgreich generierte Simplex wird zu Σ hinzugefügt und seine Faces f' klassifiziert. Davon ausgenommen bleibt jenes Face, von dem aus der Simplex erzeugt wurde. Die Klassifizierung wird wieder mittels Schnittest vollzogen, wobei diesmal mit Hilfe der Prozedur *Update* geprüft wird, ob f' bereits in der vorgesehenen Liste existiert. Dieser Fall tritt auf, wenn sich der Zwischenraum zweier Simplices mit einem neuen Simplex füllt. f' wird dann aus der

```
Function  DeWall (P:  point_set, AFL:  (d-1)face_list):  d-simplex_list;
     var f:  (d-1)face; AFL_α, AFL_1, AFL_2:  (d-1)face_list;
         t:  d-simplex; Σ:  d-simplex_list; α:  splitting_plane;
     begin
          AFL_α, AFL_1, AFL_2:= emptylist;
          Pointset_Partition(P, α, P_1, P_2);
     /* Simplex Wall Construction */
          if AFL = ∅ then
               t:= MakeFirstSimplex(P, α);
               AFL:= (d-1)faces(t); Insert(t,Σ);
          for each f ∈ AFL do
               if IsIntersected(f, α) then Insert(f, AFL_α);
               if Vertices(f) ⊂ P_1 then Insert(f, AFL_1);
               if Vertices(f) ⊂ P_2 then Insert(f, AFL_2);
          while AFL_α ≠ ∅ do
               f:= Extract(AFL_α);
               t:= MakeSimplex(f,P);
               if t ≠ null then
                    Σ:= Σ ∪ {t};
                    for each f': f' ∈ (d-1)faces(t) and f' ≠ f do
                      if IsIntersected(f', α) then
                      Update(f', AFL_α);
                      if Vertices(f') ⊂ P_1 then
                      Update(f', AFL_1);
                      if Vertices(f') ⊂ P_2 then
                      Update(f', AFL_2);
          /* Recursive Triangulation */
          if AFL_1 ≠ ∅ then Σ:= Σ ∪ DeWall(P_1, AFL_1);
          if AFL_2 ≠ ∅ then Σ:= Σ ∪ DeWall(P_2, AFL_2);
          DeWall:= Σ;
     end;

Procedure Update(f:face, L:face_list);
     begin;
          if Member(f,L) then Delete(f,L);
          else Insert(f,L);
     end;
```

Abbildung 3: *DeWall* Algorithmus

jeweiligen Liste entfernt und nicht erneut eingetragen. Ansonsten findet die gewohnte Zuordnung statt. Bis zum jetzigen Zeitpunkt wurden nur Simplices der *SimplexWall* erzeugt, welche entlang von α verläuft. Nun erfolgen rekursive Aufrufe von *DeWall* für die Faces in AFL_1 und AFL_2. Ihr Rückgabewert ist eine Liste aus Simplices, welche zusammen mit den Simplices der aktuellen Rekursion die maximale Triangulierung Σ ergeben. Funktionsabbruch und Rückgabe erfolgen, wenn beide Listen leer sind.

4 Rekonstruktion durch γ-*Indikatoren*

4.1 Der γ-*Indikator*

Der *DeWall* Algorithmus liefert eine Menge von Simplices, welche die konvexe Hülle des betrachteten Objektes füllen. Die Grundlage ihrer Erzeugung ist eine *Delaunaytriangulierung*, d.h. unter anderem finden keinerlei Überschneidungen statt und jeder Punkt ist Vertex eines Tetraeders. Das Ziel ist die Extrahierung derjenigen Faces, die das Objekt besonders gut annähern. Diese gewünschte Teilmenge wird gewonnen, indem man schrittweise alle überflüssigen Simplices und Faces löscht. Dazu benötigt man Methoden, um Simplices bzw. ihre Faces dementsprechend zu klassifizieren. Im Folgenden wird für Punkte die Bezeichnung v verwendet und nicht mehr wie bisher p. Man

kann das damit erklären, dass es sich jetzt um Elemente des Graphen handelt, während es vor der Triangulierung nur einzelne Punkte waren.

In [3] wird der sogenannte γ-*Graph* eingeführt, der eine Vielzahl an geometrischen Graphen in sich vereinigt. Betrachtet man k Punkte im k-dimensionalen Raum, die nicht in einer $(k-2)D$ Ebene liegen, so gibt es unendlich viele kD Hyperkugeln, auf deren Hülle sich alle k Punkte befinden. Als Grundlage für die Definition einer Nachbarschaft N_γ im γ-*Graph* dienen 2 Hyperkugeln B_0 und B_1, die einen gewissen Raum ausfüllen, der frei von Punkten sein muss. Ihre Radien lassen sich mit den Gleichungen 5 und 6 berechnen.

$$r_0 = r(v_1, ..., v_k) \ / \ (1 - |c_0|) \tag{5}$$
$$r_1 = r(v_1, ..., v_k) \ / \ (1 - |c_1|) \tag{6}$$

$r(v_1, ..., v_k)$ ist der Radius der kleinstmöglichen Umkugel der Punkte $v_1, ..., v_k$ und c_0, c_1 stellen Parameter zum Manipulieren von B_0, B_1 dar. Es gelten folgende Bedingungen:

- $c_0, c_1 \in [-1, 1]$
- $|c_0| \leq |c_1|$
- falls $c_0 c_1 < 0$, so liegen die Zentren von B_0, B_1 auf der selben Seite der Hyperebene durch $v_1, ..., v_k$
- falls $c_0 c_1 > 0$, so liegen die Zentren von B_0, B_1 auf unterschiedlichen Seiten der Hyperebene durch $v_1, ..., v_k$
- falls $c_1 \in [-1, 0]$, so besteht die Nachbarschaft $N_\gamma(c_0, c_1)$ aus dem Inneren von $(B_0 \cap B_1)$ und dem Inneren des $(k-1)$-Simplex aus $v_1, ..., v_k$
- falls $c_1 \in [0, 1]$, so besteht die Nachbarschaft $N_\gamma(c_0, c_1)$ aus dem Inneren von $(B_0 \cup B_1)$ und dem Inneren des $(k-1)$-Simplex aus $v_1, ..., v_k$

Ein $\gamma(c_0, c_1)$-*Graph* ist dadurch definiert, dass die Nachbarschaft $N_\gamma(c_0, c_1)$ seiner Vertices $v_1, ..., v_k$ leer ist. Abbildung 4 veranschaulicht den Zusammenhang zwischen Parametern und Nachbarschaft für den Fall in $2D$.

Es existiert eine Form des γ-*Graph*, die durch vier statt zwei Parameter beschrieben wird. In diesem Fall werden nämlich nicht feste c_0, c_1 gewählt, sondern Wertebereiche für sie angegeben. Für einen $\gamma([c_0, c_1], [c_2, c_3])$-*Graph* gilt: $N_\gamma(c_0', c_1')$ mit $c_0' \in [c_0, c_1]$, $c_1' \in [c_2, c_3]$ ist frei von Vertices, d.h. leer. Die Grenzen der Nachbarschaft sind fliessend, also ist die Menge der eingeschlossenen geometrischen Graphen variabel. Ein γ-*Graph* der Form γ([-1,1],[0,1]) entspricht der *DT*, weil dadurch die Nachbarschaft durch 2 leere Umkugeln mit beliebigen Radien festgelegt wird, auf denen sich die Vertices befinden. Damit ist das *Delaunaykriterium* erfüllt.

Eng verbunden mit der Idee des γ-*Graph* sind die γ-*Indikatoren*. Sie repräsentieren Formeigenschaften eines Simplex und bieten eine Möglichkeit Simplices miteinander zu vergleichen. Für jedes der vier Faces lässt sich dieser Indikator berechnen, indem Gleichung 7 zu 8 umgeformt wird.

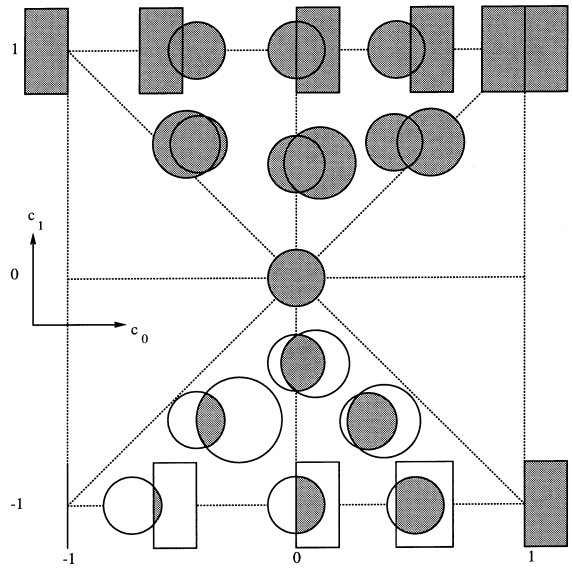

Abbildung 4: Spektrum an $2D$-Nachbarschaften $N_\gamma(c_0, c_1)$ [3]

$$R(v_i, v_j, v_k, v_l) \;=\; r(v_i, v_j, v_k) \;/\; (1 - |c_0|) \tag{7}$$

$$|c_0| \;=\; 1 \;-\; r(v_i, v_j, v_k) \;/\; R(v_i, v_j, v_k, v_l) \tag{8}$$

Dabei beschreibt R den Radius der Kugel, welche durch die Vertices v_i, v_j, v_k, v_l definiert wird und r steht für den Radius des Kreises, dessen Grundlage v_i, v_j, v_k bilden. Der γ-*Indikator* von Face $v_i v_j v_k$ ist $|c_0|$, falls sich das Zentrum der Kugel im selben (durch das Face definierten) Halbraum wie der Vertex v_l befindet. Im anderen Fall ist er $-|c_0|$.

Ein sehr flacher Tetraeder, bei dem v_l fast im Mittelpunkt von Face $v_i v_j v_k$ liegt, besitzt einen relativ grossen R. Unabhängig vom Umkreis r des Face wird der Quotient r/R relativ klein und damit $|c_0|$ annähernd 1. Da das Zentrum in diesem Fall weit auf der anderen Seite liegt, erhalten wir einen Indikator, der gegen -1 geht. Verschiebt man v_l entlang der Normalen von $v_i v_j v_k$, so wird der Tetraeder zunehmend spitzer, R kleiner und der Indikator damit grösser. Ab einer bestimmten Entfernung liegt auch das Zentrum wieder im anderen Halbraum, wodurch der Indikator in den positiven Wertebereich gelangt. Demnach ist der γ-*Indikator* ein Mass für den Öffnungswinkel φ der Tetraederspitze, ausgehend von v_l. Kleine Winkel ziehen einen grossen Indikator mit sich und umgekehrt. Das Verhältnis beider Werte ist also umgekehrt proportional. Diese Beobachtungen lassen sich auch machen, falls v_l seinen Abstand zur Ebene beibehält, die durch $v_i v_j v_k$ definiert wird. Dies ist der Fall, wenn man den v_l des flachen Tetraeders entlang der Schenkel von $v_i v_j v_k$ verschiebt. Je weiter sich v_l von seiner ursprünglichen Position entfernt, desto schiefer wird der Tetraeder. v_l bewegt sich auf die Hülle der minimalen Umkugel von v_i, v_j, v_k zu, deren Radius gleich dem Umkreisradius ist. Dadurch verringert sich R und erreicht sein Minimum, wenn v_l auf dieser Hülle liegt. Danach steigt er wieder an. Trotzdem wird der γ-*Indikator* weiterhin grösser, weil das Kugelzentrum die Seite gewechselt hat. Die erläuterten Beispiele werden durch Abbildung 5 veranschaulicht.

Je nachdem für welches Face eines Simplex man den Indikator berechnet, wird ein anderer Vertex als Tetraederspitze interpretiert. Der Indikator macht keine Aussagen über die absolute Grösse eines Tetraeders und ist deshalb geeignet, um Simplices einer *DT* einem Vergleich zu unterziehen, auch wenn deren Grundlage ungleichmässig gestreute Punkte sind.

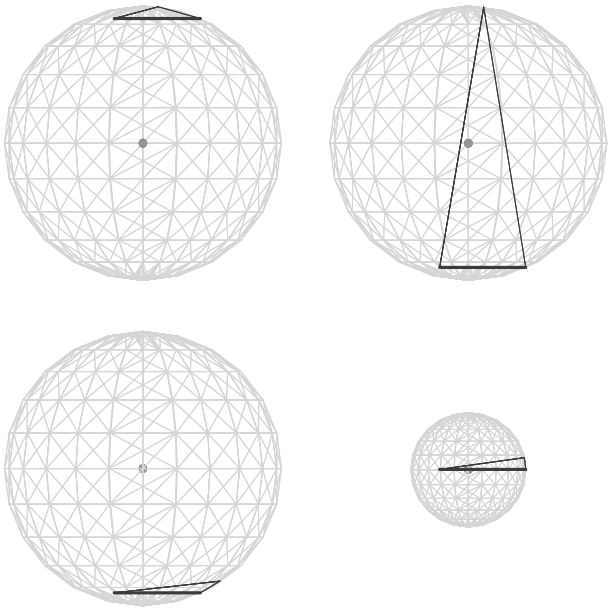

Abbildung 5: Abhängigkeit des γ-*Indikators* eines Face (hervorgehoben) von Position und Radius der Umkugel. Oben links: $\gamma \approx -0.6$ Oben rechts: $\gamma \approx 0.6$ Unten links: $\gamma \approx 0.6$ Unten rechts: $\gamma \approx 0$

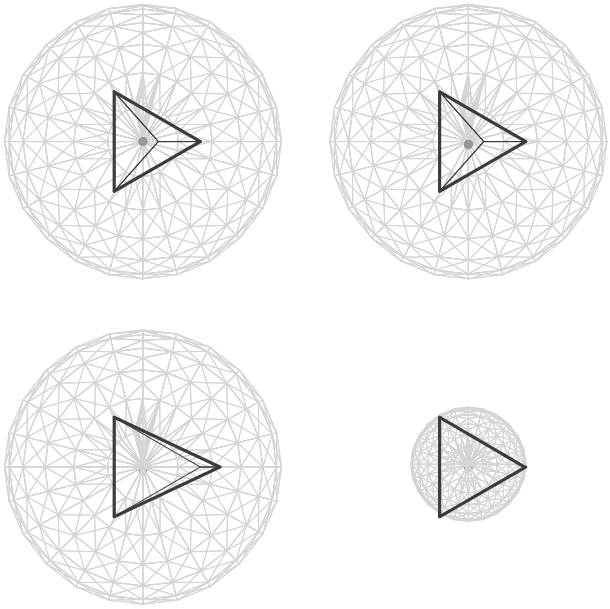

Abbildung 6: Abbildung 5 aus der Draufsicht

4.2 Löschen von Simplices

Mit Hilfe des γ-*Indikators* lässt sich also die Form eines Simplex beurteilen. Es bleibt zu klären, welche Kriterien letztendlich erfüllt sein müssen, damit er gelöscht werden kann. Zunächst einmal gehen wir davon aus, das die Gesamtanzahl der Vertices grösser als vier ist, d.h. $N_v > 4$. Ein Tetraeder kann dann bis zu drei Faces besitzen, die sich auf der Oberfläche des Objektes befinden. Solche Faces werden im Folgenden als *BoundaryFaces* bezeichnet. Im Falle von $N_v = 4$ gäbe es sogar ausschliesslich *BoundaryFaces*. Die Tetraeder im Inneren eines Objektes sind von ihren Nachbarn komplett eingeschlossen, wodurch ihre Anzahl an *BoundaryFaces* gleich 0 ist. Unabhängig von der Form des Tetraeders muss analysiert werden, ob seine Faces überhaupt gelöscht werden dürfen. Dazu dienen folgende Regeln:

- Es sind niemals 3 *BoundaryFaces* eines Tetraeders löschbar
- 2 *BoundaryFaces* $v_iv_jv_k$, $v_jv_kv_l$ sind löschbar, falls die Kante (Edge) v_iv_l nicht bereits auf der Oberfläche liegt
- 1 *BoundaryFace* $v_iv_jv_k$ eines Simplex $v_iv_jv_kv_l$ ist löschbar, falls v_l nicht bereits auf der Oberfläche liegt.

Obige Bedingungen sollen verhindern, dass unerwünschte Löcher im Objekt oder einzelne Vertices ohne zugehöriges Face entstehen. Betrachten wir einen Simplex mit einem löschbaren *BoundaryFace*. Entfernt man dieses, so gelangen drei neue Faces und ein Vertex an die Oberfläche. Diese Faces gehören nicht nur zum betrachteten Simplex, sondern gleichzeitig auch zum jeweiligen Nachbarn. Bei einer Visualisierung wird der Tetraeder nicht mehr als solcher wahrgenommen und man sieht die Faces instinktiv als Bestandteil der umgebenden Simplices. Hatte das gelöschte Face f einen sehr kleinen γ-*Indikator* nahe -1, so war der Tetraeder extrem flach. Bei einem degenerierten Simplex, d.h. alle Vertices befinden sich in einer Ebene, ist f allein so gross wie Summe der anderen Flächen. Je flacher der Tetraeder ist, desto grösser wird also die Fläche von f im Vergleich zu den drei übrigen. Der Winkel zwischen den letzteren ist weit geöffnet. Dadurch ändert sich das Erscheinungsbild dieser Region nach der Löschung von f nicht gravierend, weil je nach Beleuchtungsverhältnisssen immer noch der Eindruck einer ebenen Fläche gegeben ist. Für Faces mit grossem γ-*Indikator* lassen sich gegenteilige Beobachtungen machen. Ihre Löschung führt zu starken Veränderungen in der Oberfläche. Ausserdem enthalten sie mehr Information, da sie einen geringeren Teil der Gesamtfläche des Tetraeders ausmachen. Die konvexe Hülle eines Objektes enthält übrigens viele Faces mit kleinen Indikatoren. Da die gewünschte Oberfläche meist wenige gemeinsame Flächen mit der konvexen Hülle hat, haben diese Faces oberste Priorität beim Löschen.

Falls zwei *BoundaryFaces* gelöscht werden können, treten auch zwei neue Faces an die Oberfläche. Die Gesamtanzahl an *BoundaryFaces* im Graphen ändert sich demnach nicht. An dieser Stelle soll darauf hingewiesen werden, dass wirklich beide Faces entfernt werden müssen. Jedes für sich allein ist nämlich nicht löschbar, da der gegenüberliegende Vertex auf der Oberfläche liegt. Zwischen den neuen Faces wird wieder ein Winkel aufgespannt, der in Zusammenhang mit den γ-*Indikatoren* der gelöschten Faces steht. Um eine möglichst gute Oberflächenrekonstruktion zu erhalten, werden schrittweise diejenigen löschbaren Faces entfernt, deren Summe von γ-*Indikatoren* am kleinsten ist. D.h. bei einem Tetraeder mit zwei betroffenen *BoundaryFaces* werden die

Werte aufaddiert. Somit wird ein lokales Mass (γ-*Indikator*) mit globaler Information (das Minimum) kombiniert.

4.3 Der *Constriction* Algorithmus

```
Function  Constrict() {
      graph Graph;
      (k-1)-simplex Face, NewFace;
      int N_bv;
      heap Heap;

      Construct-γ-Graph(graph);
      N_bv = InitialBoundary(Graph, Heap);
      while(N_bv < N_v && Heap != ∅)
      {
            Face = Root(Heap);
            if(Removable(Face))
            {
                  N_bv += Delete(Face, Graph);
                  for(each NewFace on the boundary)
                  {
                        if(Removable(NewFace)) Insert(NewFace, Heap);
                  }
            }
      }
      ReportBoundary(Graph);
}
```

Abbildung 7: Der *Constriction* Algorithmus

Der Pseudocode in Abbildung 7 ist aus [3] entliehen und setzt die Überlegungen von Abschnitt 4.2 um. Dabei wird zunächst der γ-*Graph* erzeugt und die γ-*Indikatoren* für alle Faces berechnet. In unserem Fall existiert bereits ein Graph $\gamma([-1,1],[0,1])$, der durch *DeWall* erzeugt wurde. Es bietet sich an, auch die γ-*Indikatoren* während der Triangulierung zu generieren. Im nächsten Schritt werden alle *BoundaryFaces* der initialen Oberfläche erfasst und geordnet nach γ-*Indikator* in einem *Heap* eingetragen. Die anschliessende Schleife läuft solange, wie die Anzahl der Oberflächenvertices N_{bv} kleiner als die Gesamtmenge der Vertices N_v ist und der *Heap* noch Faces enthält. Zu Beginn des Loops entnimmt man das Face (bzw. die Faces) mit dem kleinsten γ aus dem *Heap* und überprüft die Löschbarkeit. Bei positivem Ergebnis wird es (bzw. werden sie) aus dem Graphen entfernt und N_{bv} inkrementiert, falls ein weiterer Vertex an die Oberfläche gelangt ist.

Für jedes der neuen *BoundaryFaces* wird wiederum entschieden, ob sie löschbar sind. Falls ja, werden sie im *Heap* abgelegt. Auch wenn der *Heap* zu Anfang nur löschbare Faces enthält und danach ausschliesslich Faces eingetragen werden, die ebenfalls entfernt werden dürfen, so muss trotzdem jedes entnommene Face erneut geprüft werden. Einige von ihnen verlieren diese Eigenschaft nämlich, weil die Löschung benachbarter Faces neue Bedingungen mit sich bringen. Nach dem Terminieren der Schleife kann der reduzierte Graph zurückgegeben werden. Das Abbruchkriterium ist übrigens keineswegs perfekt gewählt. Selbst wenn sich alle Vertices auf der Oberfläche befinden, so heisst das nicht, dass ein Objekt "gut" rekonstruiert wurde. Die besten Ergebnisse erzielt man, wenn für jeden individuellen Datensatz weiterhin manuell Faces aus dem *Heap* gelöscht werden, bis das gewünschte Ergebnis erreicht ist. Je nach Komplexität des Objektes bietet es sich an, pro Schritt gleich mehrere Faces zu löschen und nicht einzeln vorzugehen.

5 Implementierung

5.1 Framework

Die in den vorherigen Kapiteln behandelte Theorie zur Rekonstruktion von Oberflächen wird im Programm *Tesselation* (teilweise) praktisch umgesetzt. Für die Programmierung in *C++* kam das *VisualStudio 6.0* von *Microsoft* und *QT 2.3 for Windows* von *Trolltech* zum Einsatz. *QT* ist ein Framework zur plattformunabhängigen Programmierung von Benutzeroberflächen (GUI). Es bietet eine Reihe von nützlichen Funktionen, von denen die folgenden bei der Programmierarbeit genutzt wurden.

- Anlegen von Menüstrukturen
- Dialoge zum Laden/Speichern von Dateien
- Einlesen von Daten aus einer Datei
- Abfrage von Eingabegeräten
- Unterstützung für *OpenGL*

Das Konzept der Kommunikation zwischen den Komponenten verdient dabei besonderes Augenmerk. Für jede Klasse lassen sich *SIGNALS* und *SLOTS* definieren. *SLOTS* sind spezielle Methoden, die ausgeführt werden, falls ein *SIGNAL* von einem Objekt eintrifft. Ein *SIGNAL* enthält meist einen oder mehrere Parameter und kann per *emit* innerhalb einer Methode ausgelöst werden. Das Objekt selbst weiss jedoch nicht, wer dieses *SIGNAL* empfängt, bzw. welchen Ursprung ein ankommendes *SIGNAL* hat. Die Verbindung wird an einer anderen Stelle hergestellt, indem ein *SLOT* des einen mit dem *SIGNAL* des anderen Objektes verknüpft wird. Somit erhalten nur ausgewählte Komponenten eine bestimmte Art von Nachrichten, deren Transportweg ihnen aber unbekannt bleibt. Der *QT* spezifische Code wird von einem Preprozessor verarbeitet, der ihn kompatibel für den eigentlichen Kompiliervorgang macht.

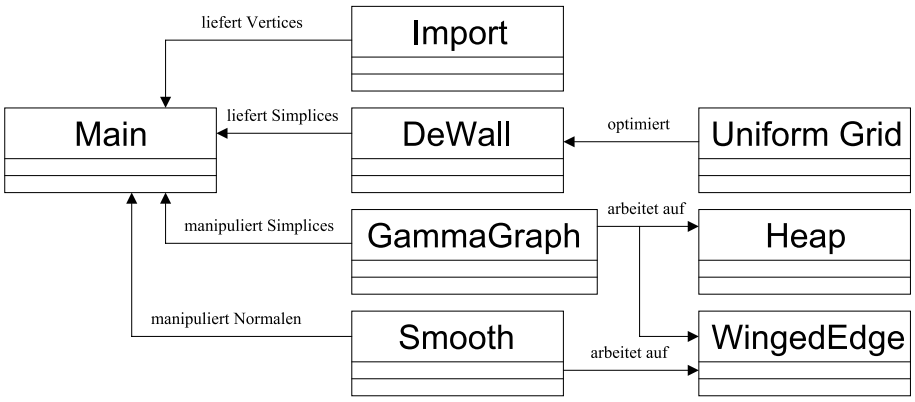

Abbildung 8: Struktur des Programmes

Abbildung 8 zeigt die wichtigsten Elemente des Programmes und ihre Beziehungen untereinander in einem einfachen Klassendiagramm. Es sei darauf hingewiesen, dass auch die detaillierteren Diagramme in den kommenden Abschnitten teilweise stark abstrahiert wurden. Einige Hilfsfunktionen fehlen, Datentypen besitzen andere Bezeichnungen und die Parameter, sowie Rückgabewerte stimmen nicht immer mit der Imple-

mentierung überein. Diese Abstriche dienen der Übersichtlichkeit und dem besseren Verständnis. Das Zusammenspiel der Komponenten lässt sich wie folgt zusammenfassen:

1. Ausgehend von *Main* wird ein Objekt *Import* erzeugt, welches Punktedaten aus einer Datei einliesst, aufbereitet und in einen Vektor (dynamisches Array) schreibt.
2. Die Punkte liefern die Grundlage für die Triangulierung durch *DeWall*. Das Ergebnis ist eine *DT* in Form einer Liste von Simplices, bzw. Tetraedern. Mit Hilfe des *UniformGrid* wird der Triangulierungsprozess beschleunigt.
3. Innerhalb der Methode *constrict()* von *GammaGraph* wird die Triangulierung modifiziert, indem Faces von einigen Simplices gelöscht werden. Dazu sind die Strukturen *Heap* und *WingedEdge* notwendig. Zurückgeliefert wird eine Liste mit den Faces der rekonstruierten Oberfläche.
4. Die Klasse *Smooth* ist für die Neuberechnung der Normalen auf den Vertices verantwortlich, um weiche Übergänge zwischen den Faces zu gewährleisten. Wieder kommt *WingedEdge* zum Einsatz.
5. Anschliessend erfolgt die Darstellung der Faces durch *OpenGL* von *Main* aus.

5.2 Elementare Datenstrukturen

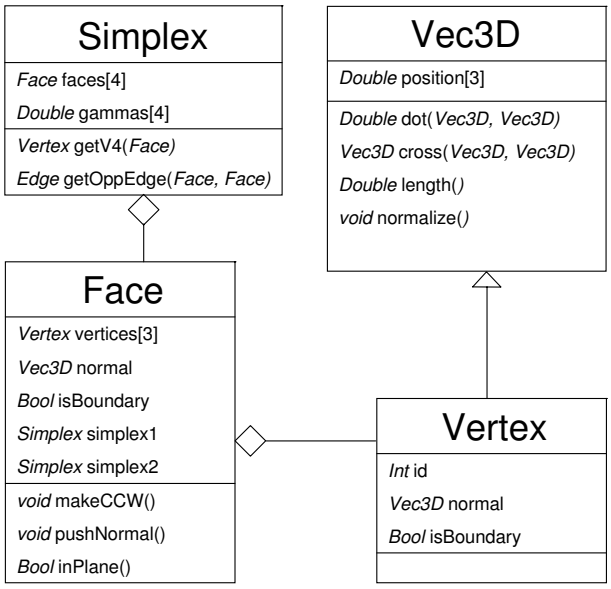

Abbildung 9: Klassendiagramm für die grundlegenden geometrischen Datenstrukturen

Die vorgestellten Verfahren basieren auf geometrischen Strukturen wie Vektoren, Vertices, Faces oder Simplices in E^3. Ihre Implementierungen bieten die notwendigen Operatoren, auch wenn sie nicht in Abbildung 9 aufgeführt sind. Die kleinste Einheit bildet die Klasse *Vec3D*, welche drei Koordinaten als *double* Wert speichert. Für sie existieren Methoden zur Berechnung der Länge und Normalisierung des Vektors. Zwischen zwei Vektoren kann das Kreuzprodukt und das Skalarprodukt gebildet werden. *Vertex* leitet sich aus *Vec3D* ab und wird um die Attribute *id*, *normal* und *isBoundary* erweitert. Eine ID ist für die Triangulierung eigentlich nicht notwendig, da die Vertices über

Pointer eindeutig identifiziert werden können. In Bezug auf die *WingedEdge* Struktur erweisst sie sich jedoch als hilfreich. Die Normale ist entscheidend für die Beleuchtung des resultierenden Objektes und *isBoundary* gibt lediglich an, ob sich ein Vertex auf der Objektoberfläche befindet. Initial steht dieser Wert auf *false*.

Die nächst höhere Struktur wird durch die Klasse *Face* repräsentiert, deren wichtigste Bestandteile ihre Pointer auf drei Vertices sind. Sie besitzt eine eigene Normale, die sich von denen der Vertices unterscheiden kann und ihr gehört ebenfalls ein Attribut *isBoundary*. Wenn sich alle Vertices auf der Oberfläche befinden, gilt das nicht automatisch für das Face, da es trotzdem innerhalb des Objektes liegen kann. Ein Face kann Bestandteil von maximal zwei Simplices sein, d.h benachbarte Simplices teilen sich eine Fläche. Für jedes Face existieren deshalb zwei Pointer, die auf zugehörige Simplices verweisen. *makeCCW()* ist eine Methode, welche die Vertices gegen den Uhrzeigersinn sortiert. Als Voraussetzung dafür muss die korrekte Normale gegeben sein. Mittels *pushNormal()* wird eben diese an die Vertices weitergegeben und dort gesetzt. Um festzustellen, ob ein beliebiger Vertex in der Ebene liegt, die durch das Face definiert wird, kann *inPlane()* verwendet werden.

Die Attribute der Klasse *Simplex* setzen sich aus vier Pointern auf Faces und ebensovielen γ*-Indikatoren* zusammen. Die γ*-Indikatoren* wurden dem Simplex zugeordnet, weil ein Face zwei davon besitzen kann und dieser Wert immer vom gesamten Tetraeder abhängt. An einigen Stellen im Programm wird der einem Face $v_1v_2v_3$ gegenüberliegende Vertex v_4 im Simplex $v_1v_2v_3v_4$ gesucht. Zu diesen Zweck existiert die Methode *getV4()*. *getOppEdge()* liefert dagegen jene Kante v_1v_4, die den Faces $v_1v_2v_3$ und $v_2v_3v_4$ entgegengesetzt ist. Übrigens wird eine Kante (Edge) als *std::pair<Vertex*, Vertex*>* modelliert. Alle erweiterten Datentypen, die keine eigene Klasse besitzen, sind in *Utility.h* definiert.

5.3 Importieren und Aufbereiten von Daten

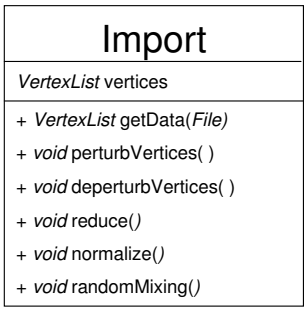

Abbildung 10: Klassendiagramm *Import*

Die Methode *getData()* liest Daten aus .ASE Dateien ein, die Objektdaten im ASCII Code beinhalten. Alle Beispieldateien wurden mit *3D Studio Max 5.0* erstellt und exportiert. Zwar enthalten die Dateien neben den Punkten auch eine Reihe anderer geometrischer Information, jedoch werden diese Abschnitte von der Importroutine ignoriert. Mit Hilfe der *QT* Methoden zum Analysieren und Bearbeiten von Strings können die Koordinaten unkompliziert eingelesen werden. Dabei ist vorallem die automatische Eliminierung der Leerstellen (Whitespaces) hilfreich. Abbildung 11 zeigt einen relevanten

Abschnitt einer .ASE Datei. Für jeden Punkt wird ein Objekt *Vertex* angelegt und in einen Vektor *std::vector<Vertex>* geschrieben.

```
*MESH_VERTEX_LIST {
*MESH_VERTEX    0 1.0628 41.6480 27.5815
*MESH_VERTEX    1 -39.0697 21.6534 22.9715
*MESH_VERTEX    2 -38.6336 -21.6478 22.1863
*MESH_VERTEX    3 1.7697 -44.9545 26.3524
*MESH_VERTEX    4 0.3024 45.2638 -22.4185
*MESH_VERTEX    5 -38.5114 17.7382 -27.0285
*MESH_VERTEX    6 -35.2081 -25.5630 -27.8137
*MESH_VERTEX    7 -1.0954 -41.3387 -23.6476
}
```

Abbildung 11: Auszug aus einer .ASE Datei

Anschliessend wird P durch *normalize()* normalisiert. Dazu ermittelt man zuerst die Ausdehnung der Punktewolke und verschiebt sie in den Koordinatenursprung. Anschliessend wird ein Skalierungsfaktor berechnet, indem man die längste Seite des Volumens durch eine feste Konstante *SCALE* dividiert. Alle Punkte werden mit diesem Faktor multipliziert.

Die Punktedaten liegen sortiert vor und werden auch in der selben Reihenfolge in die Liste geschrieben. Um "reale Verhältnisse" zu schaffen, muss verhindert werden, dass *makeSimplex()* durch die Sortierung profitiert. Auch wenn nicht explizit *nearestNeighbor* - Relationen vorliegen, kann man aus der Sortierreihenfolge oft auf lokale Nähe schliessen. Aus diesem Grund werden die Punkte nach dem Einlesen durch die Methode *randomMixing()* gemischt, indem man für jedes Element eine neue zufällige Position berechnet und die Inhalte miteinander vertauscht.

Es kommt oft vor, dass einige Punkte sehr dicht beieinander liegen oder ihre Position sogar identisch ist. Damit keine degenerierten oder einfach nur unnötigen Simplices generiert werden, löscht *reduce()* alle Punkte, die einen gewissen Mindestabstand zu ihrem *nearestNeighbor* unterschreiten. Zur Festlegung dieser Schwelle sind Informationen über den kleinsten, grössten und mittleren Abstand innerhalb von P hilfreich. Allerdings ergibt sich bei einer naiven Suche ein Aufwand von $O(n^2)$. Um die Kosten etwas einzuschränken existiert eine Konstante *SAMPLE_STEP* mit der man den Umfang der zu prüfenden Teilmenge festlegen kann. Sind die eben genannten Grössen bekannt, wird der Abstand jedes Punktes zu allen anderen ermittelt und folgende Abfrage durchgeführt:

```
if ((vertex1 - vertex2).length / global_average / global_max < RATIO) then
     delete vertex2;
```

RATIO ist dabei eine frei wählbare Konstante, welche als Schwellenwert bei der Löschung fungiert. Dieses Kriterium wurde versuchsweise ermittelt und hat sich bei den Testdatensätzen bewährt. Auch die Konstanten wurden mit Hilfe der Testobjekte abgestimmt. Es wäre auch eine leicht abgeänderte Form denkbar, die statt dem globalen Maximum den kleinsten vorkommenden Abstand verwendet oder diesen zusätzlich in das vorhandene Kriterium integriert. Wie schon erwähnt, ist diese Prozedur in der vorliegenden Implementierung mit einem quadratischen Aufwand verbunden. Ähnlich wie bei *makeSimplex()* könnten die Kosten jedoch mit Hilfe des *UniformGrid* erheblich reduziert werden.

Im Abschnitt 2 wurde kurz auf Mehrdeutigkeiten beim *DelaunayKriterium* eingegangen, die auftreten, falls mehr als vier Punkte auf der Umkugel eines Simplex liegen.

Werden diese degenerierten Fälle nicht behandelt, entstehen ungültige Triangulierungen. Die vorliegende Implementierung des *DeWall* Algorithmus läuft als Konsequenz daraus in einer Endlosschleife. Die Verfahren zur "Simulation von Einfachheit" aus [1] bieten Lösungsansätze für dieses und ähnliche Probleme. Die grundlegende Idee liegt darin, dass die Punkte eines degenerierten Datensatzes geringfügig verschoben werden, bis die Degenerierungen aufgehoben sind. Allerdings findet diese Verschiebung nur symbolisch statt. Die Methode *perturbVertices()* basiert ansatzweise auf dieser Idee, aber implementiert nicht die umfangreichen Operationen von Edelsbrunner, die für eine korrekte Umsetzung eigentlich notwendig sind. Ausgangspunkt von *perturbVertices()* ist der kleinste Abstand zwischen zwei Punkten in P. Dieser gibt eine Obergrenze für den Betrag des Verschiebungsvektors δ an, damit die Konstellation der Punkte untereinander nicht verändert wird. Danach berechnet man für jeden Punkt ein δ, indem per Zufallsgenerator seine Komponenten ermittelt und entsprechend der Grenze angepasst werden. Alle δ bekommen ihren Eintrag in einer Liste und werden auf die Punkte aufaddiert. Nach der Triangulierung wird die Translation mit Hilfe der Liste wieder rückgängig gemacht. Diese Aufgabe übernimmt *dePerturbVertices()*. Sehr ressourcenschonend ist eine solche Liste natürlich nicht, da sie für jeden Vertex einen entsprechenden Verschiebungsvektor mit Komponenten des Typs *double* enthält. Ausserdem entstehen bei allen Berechnungen mit den perturbierten Punkten Fehler, die durch eine Rückverschiebung allein nicht korrigiert werden. So erfolgt die Berechnung der Normalen z.B. während der Triangulierung. Dies geschieht unter Verwendung der manipulierten Vertices, da die Normalen für Abfragen bzgl. des Halbraumes usw. hinzugezogen werden und deshalb nicht auf originalen (und damit evtl. degenerierten) Daten basieren dürfen. Korrekterweise müssten nach der Ausführung von *dePerturbVertices()* alle vorher berechneten Grössen erneut generiert werden. Sind die gewählten δ jedoch klein genug, so ist die Abweichung der Normalen ebenfalls gering und vom Betrachter nicht erkennbar. Allerdings gibt dieses Verfahren keine absolute Garantie für die Beseitigung degenerierter Fälle. Da die Translation der Vertices auf Zufallsgrössen basiert, ist es durchaus möglich, dass ungünstige Konstellationen beibehalten werden oder sogar neu entstehen. Es wird jedoch erst während der Triangulierung deutlich, ob dies der Fall ist. Das Programm ist in dieser Beziehung verbesserungswürdig.

5.4 Der *DeWall* Algorithmus

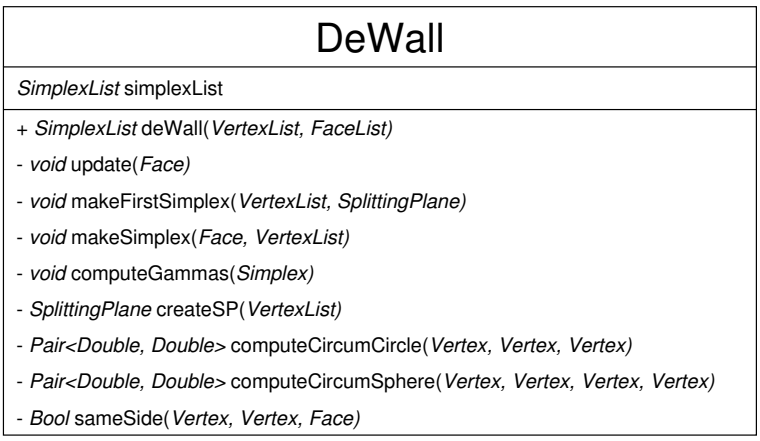

Abbildung 12: Klassendiagramm *DeWall*

Der eigentliche Algorithmus, wie er in 3 als Pseudocode vorgestellt wird, ist in der Methode *deWall()* implementiert. Der Parameter vom Typ *VertexList* erhält die Bezeichnung *PointSet* und stellt eine Liste aus Pointern auf Vertices dar. Diese umfasst beim ersten Funktionsaufruf alle Vertices. Die übergebene *FaceList* nennt sich *afl (ActiveFaceList)*. Sie besteht aus Pointern auf Faces und ist zu Anfang leer. Der genaue Ablauf von *DeWall* wurde ja bereits im theoretischen Teil erklärt, deshalb wird in den weiteren Ausführungen primär auf Implementierungsdetails eingegangen. Für die neuen Punktemengen P_1, P_2 werden dynamische Objekte $p1$, $p2$ angelegt, damit sie an späterer Stelle manuell gelöscht werden können. Die *SimplexList* Σ wird ebenfalls dynamisch erzeugt. Die Modellierung der Schnittebene α erfolgt über die Klasse *SplittingPlane*, deren Attribute sich aus einem Wert $axis \in [0, 2]$ zur Identifizierung der Achse und einer zugehörigen Koordinate $value$ zusammensetzen. Ein einfacher Vektor hätte zur Repräsentation auch genügt, jedoch müsste in diesem Fall immer geprüft werden, welche Komponente ungleich 0 ist. Dagegen gibt $axis$ sofort Aufschluss darüber, welche Koordinatenachse von der Ebene geteilt wird. Die Belegung des Objektes $alpha$ erfolgt über *createSP()*. Dort wird die längste Kante des Punktevolumens bestimmt und $alpha$ so gewählt, dass sie es orthogonal zur Kante in der Mitte teilt. Mittels *pointSetPartition()* werden alle Punkte von p in $p1$ kopiert, die sich links von *alpha* bzw. darauf befinden. Alle anderen finden dementsprechend ihren Platz in $p2$. Die Methode ist aufgrund ihrer Einfachheit sehr kurz gehalten und wird in Abbildung 13 aufgeführt.

```
void DeWall::pointSetPartition(PointSet& p, SplittingPlane& s,
                               PointSet *p1, PointSet *p2)
{
        int axis = s.getAxis();
        double value = s.getValue();
        PointSet::iterator iter;
        for(iter = p.begin(); iter < p.end(); iter++)
        {
                if((**iter)[axis] - value <= 0)
                {
                        p1->push_back(*iter);
                }
                if((**iter)[axis] - value > 0)
                {
                        p2->push_back(*iter);
                }
        }
}
```

Abbildung 13: Funktion zur Aufteilung von P in P_1 und P_2

Die Elemente in p bleiben bestehen, weil mit ihnen an späterer Stelle weitergearbeitet wird. Falls afl leer ist, sorgt *makeFirstSimplex()* für die Generierung des ersten Simplex. Dazu wird links von $alpha$ nach dem Punkt $v1$ und rechts nach $v2$ gesucht, so dass die Länge der Kante $v1v2$ minimal wird. Danach wird Ausschau nach einem geeigneten $v3$ gehalten, das den Umkreis des dadurch definierten Dreiecks ebenfalls minimiert. Dazu könnte man *computeCircumCircle()* verwenden (siehe [9]). Es existiert jedoch eine einfachere Funktion, die nur den Radius und nicht das Zentrum ermittelt. Sie trägt die Bezeichnung *cCircleRad()*, wobei die Theorie hinter der Radiusberechnung aus [5] stammt.

Nachdem so das erste Face erzeugt wurde, übergibt man es am Ende von *makeFirstSimplex()* an *makeSimplex()*, wo die eigentliche Simplexgenerierung stattfindet. Dort wird grundsätzlich unterschieden zwischen dem ersten und allen anderen Simplices. Im Falle des ersten besitzt das Face f nämlich noch keine Normale, d.h. der dafür vorgesehene

```
bool DeWall::cCircleRad(Vertex *v1, Vertex *v2, Vertex *v3, double *radius)
{
        //if some points are identical
        if(v1==v2 || v1==v3 || v2==v3) return false;

        Vec3D edge_a = (*v1 - *v2);
        Vec3D edge_b = (*v2 - *v3);
        Vec3D edge_c = (*v3 - *v1);

        double a = edge_a.length();
        double b = edge_b.length();
        double c = edge_c.length();

        //if coplanar
        edge_a.normalize();
        edge_b.normalize();
        if(fabs(dot(edge_a,edge_b)) > DELTA_02) return false;

        *radius = (a*b*c)/sqrt((a+b+c)*(b+c-a)*(c+a-b)*(a+b-c));
        return true;
}
```

Abbildung 14: Funktion zur Berechnung des Umkreisradius bei drei gegebenen Punkten

Pointer ist 0. Dementsprechend wird im vorderen und hinteren Halbraum nach geeigneten Vertices aus dem aktuellen *PointSet* gesucht. Ist eine Normale vorhanden, so wird nur der vordere Halbraum betrachtet. Für diese Abfrage wird der Vektor zwischen dem Zentrum von f sowie dem Vertex v gebildet und danach das Skalarprodukt mit der Normale gebildet. Wenn dieses grösser als 0 ist, befindet sich v im relevanten Halbraum und wird weiter untersucht. Mit Hilfe von *computeCircumSphere()* erhält man das Zentrum c und den Radius r der Umkugel, die durch das Face und den Vertex $v4$ definiert wird. Falls die Punkte coplanar oder identisch sind, gibt die Funktion den Wert $false$ zurück und der betrachtete $v4$ wird übersprungen. Eine weitere Hilfsmethode namens *sameSide()* gibt Auskunft darüber, ob c auf einer anderen Seite von f liegt als $v4$. In 3.3 wurde bereits darauf hingewiesen, dass dann r ein negatives Vorzeichen erhält. Übrigens wird in *sameSide()* wieder jeweils ein Vektor zwischen c bzw. $v4$ und dem Zentrum von f aufgespannt. Anschliessend folgen die Skalarprodukte mit der Normalen, welche dann miteinander verglichen werden. Besitzt das Dreieck noch keine Normale, wird eine temporäre erzeugt, deren Richtung in diesem Fall irrelevant ist. Der Wert von r wird nun mit dem bisherigen minimalen Radius verglichen und ersetzt diesen gegebenenfalls. Der bisher temporäre $v4$ wird dann als vorerst "bester" $v4$ gesetzt.

Wenn kein Kandidat $v4$ gefunden wurde, handelt es sich bei f um ein *BoundaryFace*, d.h. ein Element auf der konvexen Hülle. Das entsprechende Attribut wird dann für f gesetzt und *makeSimplex()* bricht mit dem Rückgabewert 0 ab. Ansonsten werden zwischen den Vertices von f und $v4$ drei neue Faces als dynamische Objekte erzeugt. Von jedem Face aus wird ein Hilfsvektor zum gegenüberliegenden Vertex gezogen, z.B. wäre für das Face $v2v3v4$ der Vektor $v1 - v2$ denkbar. Die Hilfsvektoren sind notwendig, um mit der Funktion *makeCCW()* Normalen zu berechnen und die Vertices im Uhrzeigersinn zu sortieren. Bevor der Simplex zurückgegeben wird, erfolgt noch die Berechnung der γ*-Indikatoren* durch *computeGammas()* auf Grundlage von Gleichung 8. Der Radius R und das Zentrum c der Simplexumkugel sind bereits bekannt und werden als Parameter übergeben. Jedoch müssen die Radien r_i der Umkreise aller 4 Faces noch ermittelt werden. Es kommt dabei ein ähnliches Prinzip wie bei der Umkugelberechnung zum Einsatz. Die γ*-Indikatoren* erhalten ein negatives Vorzeichen, falls sich c im hinteren Halbraum befindet. Für diese Überprüfung kann man den Vertex $v4$ hinzuziehen,

der einem Face $v1v2v3$ gegenüberliegt. Dieser ist nämlich immer im hinteren Halbraum anzutreffen, bzw. wird durch ihn sogar der hintere Halbraum definiert, falls keine Normale vorhanden ist. Übrigens erhält *computeGammas()* bei seinem Aufruf auch ein Array mit Pointern auf alle vier $v4$, welche aufgrund ihrer Position im Array einem Face zugeordnet werden können. Somit entfällt die Suche nach ihnen mittels *getV4()*.

Um auf den Anfang dieses Abschnittes, nämlich die Methode *deWall()* zurückzukommen, so befinden wir uns jetzt an einem Punkt kurz nach der Generierung des ersten Simplex. Für die Übertragung der nächsten paar Operationen vom Pseudocode in *C++* ist nicht viel Aufwand notwendig. Für Container vom Typ *std::vector<>*, welcher für die meisten Listen im Programm verwendet wird, existieren gängige Methoden wie *push_back* und *pop_back*, um das Einfügen und Entfernen von Elementen zu bewerkstelligen. Die nächste auftretende Hilfsmethode ist *isIntersected()* und prüft, ob ein Face der Liste afl von $alpha$ geschnitten wird und damit in die afl_alpha verschoben werden kann. Näheres dazu in Abbildung 15.

```
bool DeWall::isIntersected(Face *f, SplittingPlane& s)
{
        int axis = s.getAxis();
        double value = s.getValue();

        bool isLeft = false;
        bool isRight = false;
        bool isOnPlane = false;

        if((*f)[0][axis]<value) isLeft = true;
        else isRight = true;

        if((*f)[1][axis]<value) isLeft = true;
        else isRight = true;

        if((*f)[2][axis]<value) isLeft = true;
        else isRight = true;

        if (fabs((*f)[0][axis] - value) < DELTA_04 ||
            fabs((*f)[1][axis] - value) < DELTA_04 ||
            fabs((*f)[2][axis] - value) < DELTA_04) isOnPlane = true;

        return ((isLeft && isRight) || isOnPlane);
}
```

Abbildung 15: Funktion für den Schnittest von Ebene s und Face f

An späterer Stelle taucht die Methode *update()* auf, welche ein Face f in afl einträgt, falls es dort noch nicht vorhanden ist und ansonsten aus afl entfernt. Es werden dabei nicht die Pointer (d.h. Speicheradressen) der Faces auf Gleichheit geprüft, sondern wirklich ihre Geometrie, da z.B. die Reihenfolge der Vertices ruhig variieren kann, solange nur die Position identisch ist. Wie in 3.5 angesprochen, tritt letzerer Fall auf, wenn der Raum zwischen zwei Simplices mit einem weiteren Simplex ausgefüllt wurde. An der Verbindungsstelle des bereits existierenden und gerade erzeugten Simplex befinden sich dann zwei Faces, obwohl sie sich eigentlich eines teilen müssten. Demnach muss beim letzten Simplex der Pointer von jenem Face dem anderen angepasst werden, damit sie auf dasselbe verweisen.

Bevor die rekursiven Aufrufe von *deWall()* erfolgen, wird das *PointSet* p gelöscht, weil im Anschluss nur noch $p1$,$p2$ benötigt werden, die ihre eigenen Pointer auf Vertices enthalten. Die Ergebnisse der Rekursionen sind Teilmengen der Triangulierung Σ, d.h. eine *SimplexList*, welche durch *merge()* mit der Liste $sigma$ aus dem aktuellen Funktionsdurchlauf vereinigt und am Ende zurückgegeben wird.

5.5 Optimierung durch das *UniformGrid*

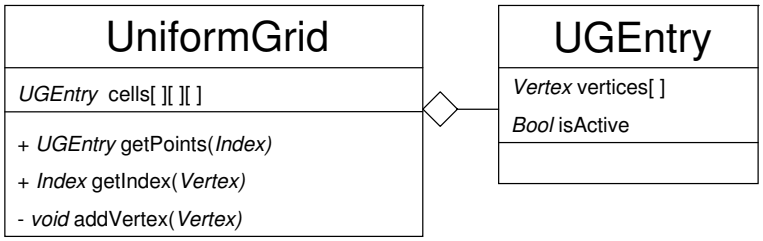

Abbildung 16: Klassendiagramm *UniformGrid*

Der Grossteil der Rechenzeit wird durch *makeSimplex()* in Anspruch genommen, da für alle Vertices in p die Methode *computeCircumSphere()* aufgerufen wird. Die Anzahl der Aufrufe sollten deshalb auf die wirklich nötigen reduziert werden. In Abschnitt 3.4 wurden Optimierungsmöglichkeiten auf Basis des *UniformGrid (UG)* angesprochen.

Die Struktur des *UG* wird durch ein dreidimensionales Array modelliert welches auf Objekte des Typs *UGEntry* verweisst. Ein *UGEntry* besteht aus einem *PointSet* und dem Wert *isActive*, der den Zustand des Feldes beschreibt. Vor dem Aufbau des *UG* muss seine Auflösung festgelegt werden. Dazu existiert zunächst eine Konstante *density* $\in [0, 1]$, welche dann mit der Anzahl der Punkte verrechnet wird, um den Wert für *resolution* festzulegen. Die Zellen wurden nicht als quadratisch angenommen. Ihre Ausdehnung hängt von den Maßen des Punktevolumens ab. Bei der Erzeugung des Arrays enthalten alle Felder zunächst nur den Nullpointer. Danach wird jeder Punkt durch *addVertex()* seinem zugehörigen Feld zugeordnet, indem an der entsprechenden Position im Array ein *UGEntry* erzeugt wird, falls das nicht schon vorher geschehen ist. Das *PointSet* eines *UGEntry* wächst mit jedem hinzugefügten Punkt. Zum Auffinden des richtigen Feldes eines Punktes dient *getIndex()* (siehe Abbildung 17), während *getPoints()* für einen bestimmten Index das jeweilige *UGEntry* zurückliefert.

```
Index3D UG::getIndex(Vertex *v)
{
        int index_x, index_y, index_z;

        if((*v)[0] == m_xMax) index_x = m_resolution - 1;
        else index_x  = ((*v)[0] - m_xMin) / m_cellWidthX;

        if((*v)[1] == m_yMax) index_y = m_resolution - 1;
        else index_y  = ((*v)[1] - m_yMin) / m_cellWidthY;

        if((*v)[2] == m_zMax) index_z = m_resolution - 1;
        else index_z  = ((*v)[2] - m_zMin) / m_cellWidthZ;

        return Index3D(index_x, index_y, index_z);
}
```

Abbildung 17: Funktion zur Ermittlung des Index eines Punktes im *UG*

Die Optimierung von *makeSimplex()* stellte sich als aufwendiger als zunächst erwartet heraus. Das lag u.a. an folgenden Faktoren:

- Für jeden neuen Kandidaten von $v4$ existiert ein neuer Suchraum, der von Zentrum und Radius der entsprechenden Umkugel abhängt. Das Zentrum kann jedoch weit ausserhalb des *UG* liegen und der Radius sehr grosse Werte annehmen (z.B. bei extrem flachen Tetraedern)

- Vorher war die Suchmenge durch ein *PointSet* gegeben, in dem nur Punkte im relevanten Halbraum vorhanden waren. Diese Überprüfung muss jetzt wieder erfolgen.
- Das Suchvolumen sollte von innen nach aussen traversiert werden.

Der erste Schritt ist die Bestimmung des Zentrums c von Face f und des Feldes im *UG*, in welchem c liegt. Die Adresse dieses Feldes bildet den *startIndex* und dort wird zuerst nach geeigneten Vertices gesucht. Findet sich ein $v4$, so wird der *startIndex* für den neuen Suchraum berechnet und durch die Werte *leftborder, rightborder, downborder, upborder, backborder, frontborder* seine Abmessungen gespeichert. Bei einem komplett abgesuchten Feld wird das Attribut *isActive* auf *false* gesetzt. Falls kein passender $v4$ im Feld des *startIndex* gefunden wird, so dehnt sich die Suche auf die Nachbarzellen aus. Das Suchvolumen wird somit schichtweise um sein Zentrum c abgearbeitet, bis die Grenzen erreicht sind. Oft liegt das berechnete c ausserhalb des *UG* - es wird dann das nächstgelegene wirklich existierende Feld als *startIndex* verwendet. Um schnell herauszufinden, welche Punkte sich im richtigen Halbraum befinden, werden *makeSimplex()* die Grenzen des *PointSet* übergeben, welches man in der unoptimierten Version untersucht. Für die Repräsentation dieser Informationen gibt es eine Klasse *PSContainer*. Das Abbruchkriterium ist erreicht, wenn die Grenzen des Suchvolumens erreicht sind, oder alle Felder des *UG* inaktiviert wurden.

Die Implementierung des Optimierungsverfahrens ist nicht abgeschlossen. Aus einem unbekannten Grund werden immer noch zuviele Zellen besucht - die Ersparnis liegt bei höchstens 5%. Der Mehraufwand, welcher mit dem *UG* verbunden ist, fällt zusätzlich ins Gewicht, so dass von einer Optimierung keine Rede sein kann. In den Diagrammen aus [2] ist ein Beschleunigungsfaktor von über 10 keine Seltenheit.

5.6 Die *WingedEdge* Datenstruktur

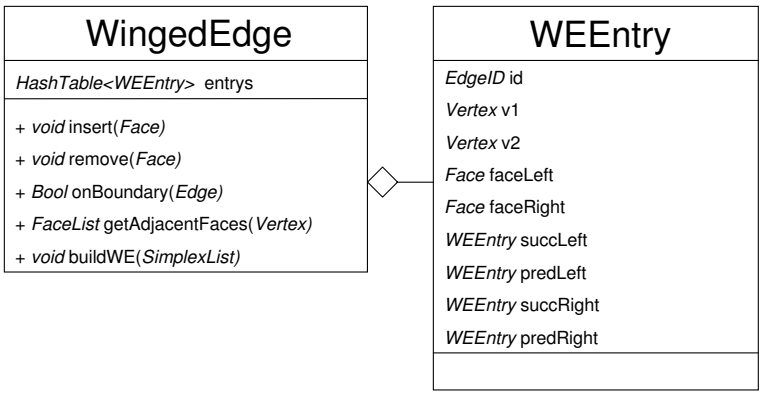

Abbildung 18: Klassendiagramm *WingedEdge*

Bevor auf die Umsetzung des *Constriction* Algorithmus eingegangen werden kann, soll die *WingedEdge* Datenstruktur vorgestellt werden. Diese speichert Nachbarschaftsinformationen basierend auf Kanten. Für jedes Edge werden seine beiden Vertices, die benachbarten Faces sowie die Kanten dieser Faces verwaltet. Für die letzteren verwendet man die Begriffe *Vorgänger links, Nachfolger links, Vorgänger rechts, Nachfolger*

rechts, wobei mit links bzw. rechts das jeweilige Face gemeint ist. Durch diese Parameter wird berücksichtigt, dass dieselbe Kante abhängig vom betrachteten Face zwei unterschiedliche Richtungen haben kann, obwohl sie nur einmal abgespeichert wird. Obige Informationen stammen aus [8].

Mit Hilfe einer Hashtabelle lässt sich die Datenstruktur recht einfach modellieren. Dabei wird ein Array angelegt, dessen Grösse der Anzahl aller Vertices entspricht. Jedes Feld repräsentiert eine Liste aus Elementen vom Typ *WEEntry*. Ein Edge wird durch die IDs seiner Vertices identifiziert, so dass man zum Adressieren des gewünschten Eintrages zunächst das richtige Feld im Array über die erste ID ansteuert. Danach liefert die Suche nach der zweiten ID innerhalb der Liste das Ergebnis.

Mittels *buildWE()* baut sich die Struktur ausgehend von einer *SimplexList* auf. Da die Nachbarschaftsinformationen für den zweidimensionalen Fall ausgelegt sind, werden natürlich nur die *BoundaryFaces* der Simplices aufgenommen. Die Oberfläche liesse sich nämlich aufrollen und auf eine Ebene projizieren. Für die korrekte Eintragung der Faces zeigt sich *insert()* verantwortlich, welches sich wiederum auf *makeEntry()* stützt. Bevor ein Edge seinen Eintrag erhält, muss überprüft werden, ob es bereits als solches oder auch in umgekehrter Richtung in der Struktur existiert. Bei der Erstellung des Eintrages finden die Informationen ihren Platz, welche durch die anderen Edges des Face bekannt sind. Es wird übrigens immer als das linke Face klassifiziert, da bei den Punkten eine Sortierung im Uhrzeigersinn vorliegt. Das Wissen über *Face rechts, Vorgänger rechts, Nachfolger rechts* wird erst bei späteren Aufrufen von *insert()* hinzugefügt, wenn das entsprechende Face aufgenommen wird. Dann nämlich stellt *makeEntry()* fest, dass ein Edge bereits umgekehrt in der Struktur besteht und man somit auf benachbarte Faces gestossen ist.

Falls sich an der Zusammenstellung der *BoundaryFaces* etwas ändert, soll natürlich nicht die gesamte Struktur neu aufgebaut werden. Stattdessen löscht man die Informationen über nicht mehr vorhandene Faces durch *remove()*. Einzelne Verweise der betroffenen Kanten werden dabei auf 0 gesetzt. Isolierte Edges ohne linkes oder rechtes Faces werden komplett gelöscht, d.h. ihr Eintrag wird aus der Hashtabelle entfernt.

Der eigentliche Sinn der Struktur für dieses Programm besteht zum einen darin, die benachbarten Faces eines Vertex zu finden. Zum anderen soll schnell festgestellt werden können, ob ein Edge auf der Oberfläche liegt. Letztere Information liefert *onBoundary()*, indem einfach geprüft wird, ob jenes Edge in irgendeiner Form als Eintrag vorhanden ist. Die Methode *getAdjacentFaces* sucht zunächst nach einem beliebigen Edge, welches den betroffenen Vertex enthält, da keine Einstiegspointer angelegt wurden. Je nach Richtung des aktuellen Edge wird das linke oder rechte Face der Ausgabeliste hinzugefügt und im Uhrzeigersinn das nächste Edge bestimmt. Die Liste kann zurückgegeben werden, wenn man wieder beim ersten Edge angekommen ist.

5.7 Der *Constriction* Algorithmus

In 4.3 ist von einem *Heap* die Rede, in welchem die Werte von *gamma* sortiert werden. Es handelt sich dabei um einen binären Baum aus Elementen, die ein *gamma* speichern und auf den Simplex verweisen, in dem sich das zugehörige Face befindet. Wird ein neues Element eingefügt, so gelangt es in den linken Teilbaum, falls sein *gamma* klei-

```
WEEntry* WingedEdge::makeEntry(Face *f, edge e, edgeID eid, bool *isReverse)
{
        int id1 = eid.first;
        int id2 = eid.second;
        EntryList::iterator iter;

        EntryList *actEntryList = &(m_WEHash[id1]);
        EntryList *actEntryListRev = &(m_WEHash[id2]);
        WEEntry *actEntry = 0;

        //check if the edge already exists in the structure
        for(iter = actEntryList->begin(); iter != actEntryList->end(); iter++)
        {
                if((*iter)->getID().second == id2)
                {
                        //entry and face already exist
                        if((*iter)->m_faceLeft == f) return *iter;
                        //its another face whose normal points in another direction
                        else
                        {
                                (*iter)->m_faceLeft = f;
                                return *iter;
        }       }       }

        //check, if the reverse edge is already in the structure
        for(iter = actEntryListRev->begin(); iter != actEntryListRev->end(); iter++)
        {
                if((*iter)->getID().second == id1)
                {
                        *isReverse = true;
                        return *iter;
        }       }

        actEntry = new WEEntry;
        //add all available information to Entry
        actEntry->m_edgeID = eid;
        actEntry->m_v1 = e.first;
        actEntry->m_v2 = e.second;
        actEntry->m_faceLeft = f;
        //add Entry to its list
        actEntryList->push_back(actEntry);

        return actEntry;
}
```

Abbildung 19: Funktion zum Erstellen eines Eintrages in der *WingedEdge* Datenstruktur

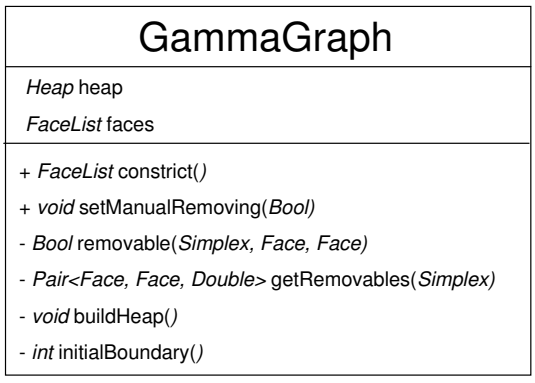

Abbildung 20: Klassendiagramm *GammaGraph*

ner gleich dem der Wurzel ist. Ansonsten wird es in den rechten eingetragen. Dieses Kriterium wendet man rekursiv an, bis ein Knoten keine entsprechenden Nachfolger hat und ein neuer Eintrag erstellt werden kann. Zum Entfernen eines Elementes e, werden seine Kindknoten erneut in den *Heap* eingesetzt, nachdem e gelöscht wurde. Die Me-

thode *extractNext()* steigt entlang der linken äusseren Knoten hinab, um das minimale *gamma* zurückzuliefern und gleichzeitig aus dem Baum zu entfernen.

Die zentrale Methode der Klasse *GammaGraph* ist *constrict()*, deren Funktionsweise anhand des Pseudocodes in Abbildung 4.3 nachvollzogen werden kann. Der *GammaGraph* an sich existiert schon in Form der *DelaunayTriangulierung*, deren Daten in einer *SimplexList* gehalten werden. Nachdem der *Heap* aufgebaut und alle *BoundaryVertices* gezählt wurden, beginnt auch schon die Hauptschleife. Diese terminiert, wenn alle Vertices auf der Oberfläche liegen, oder der *Heap* leer ist. Setzt man den Wert von *manualRemoving* auf *true*, kann man den ersten Teil des Abbruchkriteriums ausser Kraft setzen und weiterhin Faces löschen. Dabei wird ein (frei wählbarer) Wert *stepsize* dekrementiert. Steht dieser auf 0, so wird *manualRemoving* deaktiviert. Der Benutzer kann somit z.B. in einem Schritt weitere 100 Faces entfernen, selbst wenn schon sämtliche Vertices zu *BoundaryVertices* geworden sind.

Nachdem ein Element mit dem kleinsten *gamma* aus dem *Heap* entfernt wurde, untersucht man den jeweiligen Simplex auf seine löschbaren Faces. Durch *getRemovables()* wird geprüft, ob ein Paar von Faces, ein einzelnes Face oder keines für eine Löschung in Frage kommt. Die erste Voraussetzung besteht natürlich darin, dass sie auf der Oberfläche liegen. Ausserdem muss die Analyse einer weiteren Hilfsfunktion namens *removable()* (siehe Abbildung 21) positiv ausfallen. Als Parameter bekommt *removable()* zwar Pointer auf zwei Faces übergeben, aber einer davon kann 0 sein, damit auch der Test eines einzelnen Face möglich ist. Im Fall von zwei löschbaren Faces zeigt sich der Nutzen der *WingedEdge* Datenstruktur. Ohne sie wäre es sehr aufwendig ein Edge darauf zu überprüfen, ob es an der Oberfläche liegt.

```
bool GammaGraph::removable(Simplex *s, Face *f1, Face *f2)
{
        Vertex *v4;
        if(f1 == 0 && f2 == 0) return false;

        //check if one face is removable
        if(f1 == 0)
        {
                //the vertex v4 of the simplex s, which does not belong to face f2
                v4 = s->getV4(f2);
                //v4 must not lie on the boundary
                return !(v4->onBoundary());
        }
        if(f2 == 0)
        {
                v4 = s->getV4(f1);
                return !(v4->onBoundary());
        }

        //check if two faces are removable
        if(f1 != 0 && f2 != 0)
        {
                //faces are removable, if the edge on the other side of the
                //simplices does not lie on the boundary
                return !(m_wingedEdge->onBoundary(s->getOppEdge(f1,f2)));
        }

        return false;
}
```

Abbildung 21: Funktion zur Überprüfung von Faces auf Löschbarkeit

Kommen wir zurück zu *constrict()*. Die löschbaren Faces und die Summe ihrer *gammas* wurden soeben ermittelt und nun muss diese Summe mit dem Wert des entnommenen

Heapelementes verglichen werden. Es ist nämlich möglich, dass durch die Manipulation anderer Simplices ein Eintrag seine Aktualität verliert, da sich die Anzahl der löschbaren Faces des Simplex verändert hat. Falls die neue Summe grösser als die alte ist, so wird ein neues *Heapelement* erzeugt und eingefügt. Anschliessend folgt ein neuer Schleifendurchlauf - schliesslich könnte bereits ein Element existieren, dessen *gamma* kleiner als die korrigierte Summe ist.

Wenn es dann wirklich an die Löschung von Faces geht, müssen die beteiligten Simplices angepasst werden. Da jedes Face einen Verweis auf seinen (oder seine) Tetraeder besitzt, ist dies schnell erledigt. Ausserdem benötigt die *WingedEdge* Datenstruktur ein Update und der Speicher für die Faces sollte freigemacht werden. Falls nur ein Face entfernt wird, erhöht sich die Zahl der *BoundaryVertices* um 1, da drei neue Faces an die Oberfläche geraten. Die neuen Faces sind ja nicht nur Elemente des betrachteten Simplex, sondern ebenso anderer Simplices. Werden diese ermittelt, so ist der erstere nicht mehr notwendig, wodurch die Verweise der Faces auf ihn und umgekehrt gelöscht werden können. Man kann ihn sich dann als einen unsichtbaren Simplex vorstellen. Die Normalenrichtung der neuen Faces war bisher nicht eindeutig definiert. Mit Unterstützung eines Hilfsvertex sorgt *reorientNormal()* für die korrekte Orientierung. Dieser Hilfsvertex ist jeweils der $v4$ der Simplices, da er sich immer in dem Halbraum befindet, in den die Normale nicht zeigen soll. Danach werden die Punkte mit *makeCCW()* korrekt sortiert, *WingedEdge* entsprechend der neuen Faces aktualisiert und das Attribut *isBoundary = true* für jedes Face gesetzt. Anschliessend erfolgt die Überprüfung der Simplices auf löschbare Faces, die dann wieder im *Heap* ihren Platz finden. Nach Ablauf der Hauptschleife werden Pointer auf alle *Boundaryfaces* innerhalb der *SimplexList* in eine Liste kopiert und diese zurückgegeben.

An diesem Punkt der Implementierung angekommen müsste es also möglich sein, eine Punktemenge zu triangulieren und danach alle überflüssigen Faces zu löschen, so dass die ursprüngliche Form des Objektes möglichst genau angenähert wird. Die Resultate entsprechen leider nicht den Erwartungen, wie man z.B. in den Abbildungen 25 und 28 sehen kann. Es muss ein Fehler in der Implementierung vorliegen, da es sich nicht etwa nur um kleinere Unregelmässigkeiten handelt, sondern das Objekt als solches teilweise nicht erkennbar ist. An sehr simplen Objekten mit wenigen Polygonen sind die gemachten Schritte nachvollziehbar und das Ergebnis scheint korrekt zu sein. Wird die Geometrie allerdings komplizierter, treten Probleme auf. Etliche "gute" Faces werden dann gelöscht und "schlechte" beibehalten. Die bisherigen Überprüfungen haben keine Hinweise auf die Ursache geliefert. Evtl. ist es sogar möglich, dass der Fehler nicht beim *Constriction* Algorithmus liegt, sondern schon während der Triangulierung auftritt. Allerdings würde das in vielen Fällen zu einer Endlosschleife in *deWall()* führen.

5.8 Smoothing

Unter Smoothing werden hier keine Verfahren zur Neutriangulierung verstanden, wie man sie verwendet, um einem *Mesh* mit unterschiedlich stark aufgelösten Bereichen eine gleichmässige Struktur mit weichen Übergängen zu verleihen. Mit Hilfe der Klasse *Smooth* werden lediglich die Normalen auf den Vertices so angepasst, dass sie dem arithmetischen Mittel der umgebenden Flächennormalen entsprechen. Erst dann werden die Vorteile des *GouraudShadings* gegenüber dem *FlatShading* sichtbar, da die Vertices

eines Face individuelle Farben (bzw. Helligkeitswerte) besitzen, zwischen denen interpoliert werden kann. Somit wirken die Übergänge an den Flächenkanten weicher und dem Objekt wird mehr Plastizität verliehen.

Eine wichtige Voraussetzung liegt darin, dass die Flächen um einen Vertex herum schnell gefunden werden. Dafür sorgt die Methode *getAdjacentFaces()* der *WingedEdge* Datenstruktur. Der Code für die Normalenmittelung findet sich in Abbildung 22.

```
void Smooth::doSmoothing(WingedEdge *we, FaceList *fl, Vertices *v)
{
        FaceList::iterator fIter;
        Vertices::iterator vIter;

        FaceList *temp_fl = 0;

        for(vIter = v->begin(); vIter != v->end(); vIter++)
        {
                //only smooth boundary
                if(!vIter->onBoundary()) continue;

                //get faces surrounding vertex v
                temp_fl = we->getAdjacentFaces(vIter);

                Vec3D temp_n;
                int count = 0;

                //sum all normals of the faces
                for(fIter = temp_fl->begin(); fIter != temp_fl->end(); fIter++ )
                {
                        assert((*fIter)->getNormal() != 0);
                        temp_n = temp_n + *((*fIter)->getNormal());
                        count++;
                }

                temp_n = temp_n / count;
                (*vIter).setNormal(temp_n);
        }
}
```

Abbildung 22: Funktion zur Mittelung der Normalen

6 Fazit

Die Rekonstruktion von ungleichmässig dicht gesampelten Datensätzen erweist sich als nicht einfach. Selbst mit einer funktionierenden Implementierung wie sie in [3] vorgestellt wurde, gibt es keine Garantie für ein zufriedenstellendes Ergebnis. In vielen Fällen muss der Benutzer in den Rekonstruktionsprozess eingreifen, indem nach Abschluss der automatischen Löschung weitere Faces entfernt werden, bis ein gutes Resultat erreicht ist. Man muss jedoch darauf hinweisen, dass die betreffende Literatur über 10 Jahre alt ist und mittlerweile zuverlässigere Verfahren existieren. So wirken die Resultate in [7] oder [6] um einiges ausgereifter, wenn auch die Umsetzung aufwendiger erscheint. Doch auch für den *Constriction* Algorithmus wären Erweiterungsmöglichkeiten denkbar. Eines seiner Prinzipien ist ja die Grössenunabhängigkeit der Faces. Setzt man jedoch einen einigermassen gleichmässig gesampelten Datensatz voraus, könnte eine Einbeziehung absoluter Grössen in die Analyse bessere Ergebnisse liefern. Dieses hybride Verfahren wäre natürlich bei stark gestreuten Punktewolken kaum von Vorteil. Letztendlich ist es wie bei allen Verfahren - je höher die Qualität des Input, desto bessere Ergebnisse liefert der Output.

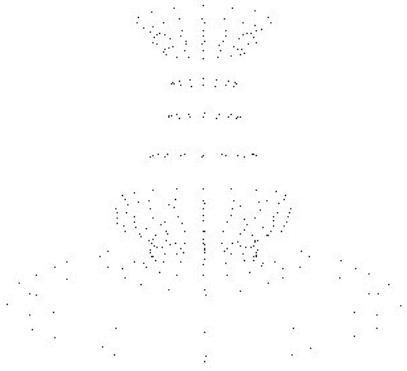

Abbildung 23: Objekt *Vase*: 306 Vertices

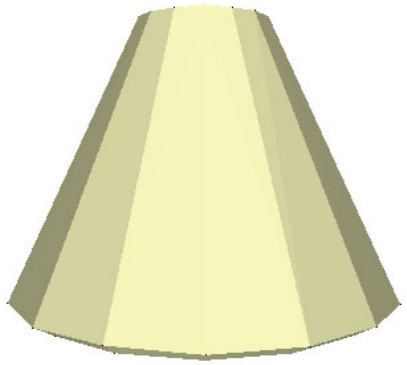

Abbildung 24: Objekt *Vase*: DeWall abgeschlossen: 92 *BoundaryFaces*

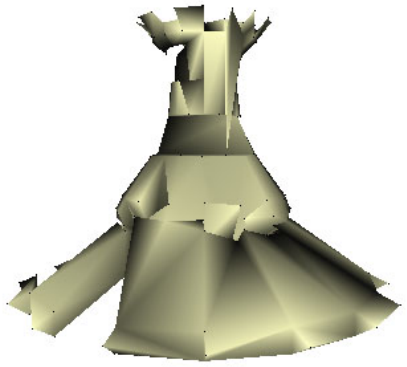

Abbildung 25: Objekt *Vase*: Constriction abgeschlossen: 606 *BoundaryFaces* (smoothed)

Abbildung 26: Objekt *Reiter*: 728 Vertices

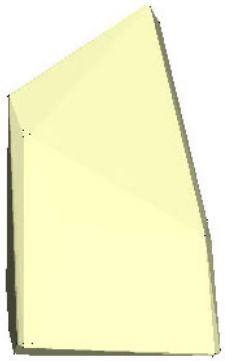

Abbildung 27: Objekt *Reiter*: DeWall abgeschlossen: 140 *BoundaryFaces*

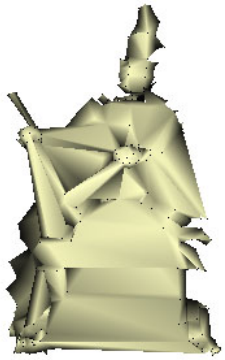

Abbildung 28: Objekt *Reiter*: Constriction abgeschlossen: 1452 *BoundaryFaces* (smoothed)

Literatur

[1] *Edelsbrunner, H.; Mücke, E.P.* (1990): "Simulation of Simplicity: A technique to Cope with Degenerate Cases in Geometric Algorithms."

[2] *Cignoni, P.; Montani, P.; Scopigno, R.* (1997): "DeWall: A Fast Divide & Conquer Delaunay Triangulation Algorithm in E^d."

[3] *Veltcamp, C.* (1994): "Closed Object Boundaries from Scattered Points." Berlin: Springer-Verlag

[4] *Shewchuk, J.R.* (1996): "predicates.c"(Sourcecode)
http://www.cs.cmu.edu/ quake/robust.html

[5] *Weisstein, E.W.* (1999): "circumradius"
http://mathworld.wolfram.com/Circumradius.html

[6] *Dey, T.K.; Giesen, J.; Hudson, J.*: "Delaunay Based Shape Reconstruction from Large Data."

[7] *Tobor, I.; Reuter, P.; Schlick, C.* (2003): "Efficient Reconstruction of Large Scattered Geometric Datasets using the Partition of Unity and Radial Basis Functions."

[8] *Müller, S.* (2002): "Computergraphik 2"(Vorlesung)

[9] *Shewchuk, J.R.* : "circumsphere"(Sourcecode)
http://www.ics.uci.edu/ eppstein/junkyard/circumcenter.html