Einleitung

Die heutige Computergrafik bedient sich verschiedener Techniken zur effizienten Objektrepräsentation, wie zum Beispiel in Form von Polygonen, oder aber auch durch die Definition von impliziten Oberflächen in volumetrischen Datensätzen. Unabhängig von Effizienz und Güte der einzelnen Techniken, stellt sich die Frage nach geeigneten Schnittstellen zwischen den verschiedenen Ansätzen. So wäre es unter bestimmten Umständen wünschenswert einen volumetrischen Datensatz mithilfe von CT-Scans zu erzeugen und diesen als traditionelles Mesh zu rendern. 3-Dimensionale Scanner entwickeln sich zu einer sehr angenehmen Art Modelle zu gewinnen, da sich der Prozess des Modellierens von Hand vergleichsweise aufwendig gestaltet. Schon seit geraumer Zeit finden sie Anwendung in der Medizin und der Visualisierung im Allgemeinen. Allerdings ist die Überführung eines solchen Scans in ein qualitativ hochwertiges Mesh im Allgemeinen nicht trivial.

Natürlich gibt es bereits verschiedene Techniken zur Umwandlung der volumetrischen Datensätze, der bekannteste ist der simple und robuste Marching Cubes Algorithmus. Allerdings erzeugt gerade dieser meist eine ineffiziente Objektrepräsentation, mit unnötig vielen und schlecht positionierten Dreiecken. Auch gehen durch die Festlegung auf ein festes Raster Details unwiederbringlich verloren. In vielen Bereichen, in denen es auf möglichst hohe Effizienz ankommt, ist es allerdings nötig nur Meshes von bester Qualität zu verwenden.

Ziel dieser Arbeit ist es daher einen Algorithmus zu entwickeln, der die Umwandlung eines volumetrischen Datensatzes in ein traditionelles Mesh erlaubt und den folgenden Kriterien genügt:

* Minimaler Verlust von Details bei minimaler Anzahl von Dreiecken . Mithilfe von adaptivem Triangulieren der Oberfläche sollen Details bewahrt werden, indem die lokale Größe der Dreiecke antiproportional zur Krümmung angepasst wird. Gleichzeitig können auf diese Weise große Fläche mit einer geringen Zahl von Dreiecken trianguliert werden.
* Beliebige Komplexität der Approximation und minimale Anforderungen an den Nutzer. Der Algorithmus soll nur auf zwei Parametern, p und n, basieren. p kontrolliert die Genauigkeit der Approximation und damit auch die Anzahl der Dreiecke. n beschreibt das maximale Verhältnis der Länge von adjazenten Kanten und bestimmt somit die Änderungsrate der lokalen Dreiecksgröße.

Stand der Wissenschaft

Blub.

Definition der Oberfläche

Blub.

Guidance Field Konstruktion

Blub.

Triangulierung

Nachdem das Guidance Field erstellt wurde und die Oberfläche ausreichend gesampelt ist, kann nun mit der eigentlichen Triangulierung des Datensatzes begonnen werden. Dazu werden allerdings noch geeignete Seed-Fronten benötigt. In der vorliegenden Arbeit wurde lediglich ein Sample ausgewählt, ausgewertet und eine neue Kante mit der entsprechenden idealen Kantenlänge auf die Oberfläche projiziert. Offensichtlich ist diese Verfahren nicht in der Lage komplexere Modelle, die aus mehreren einzelnen Teilen bestehen, zu triangulieren. Hierzu wäre ein Verfahren nötig, welches garantiert, dass jedes Teilobjekt zumindest einen Seed erhält. Allerdings ist der Algorithmus in seinem aktuellen Zustand bereits in der Lage eine beliebig Anzahl von Seeds auf beliebig vielen Teilobjekten korrekt zu verarbeiten, daher müsste nur die Identifizierung der Seeds verbessert werden.

Die erzeugten Seed-Fronten werden nun dem Front Manager übergeben, welcher sich über die Dauer des Algorithmus um das Hinzufügen und Entfernen von Fronten kümmern wird. Jede Front hat eine eindeutige ID, genauso jedes Front-Element. Solange der Front Manager Fronten enthält werden die folgenden Schritte ausgeführt:

1. Identifizierung eines Front-Elements
   1. Wähle eine zufällige Front *f*
   2. Falls *f* nur aus drei Elementen besteht und keine Seed-Front ist, entferne sie und füge an ihrer Stelle ein neues Dreieck zur Triangulierung hinzu.
   3. Sonst ist *e* die Kante zwischen einem zufälligen Element aus f und dessen Vorgänger
2. Konstruktion eines neuen, idealen Dreiecks
   1. Evaluierung des Guidance Fields an den Vertices von e liefert die jeweilige ideale Kantenlänge
   2. Falls die Kreise idealer Kantenlänge sich schneiden, ist der Schnittpunkt unser neues Vertex *vNew*
   3. Projektion von *vNew* auf die Oberfläche
3. Erkennung und Auswahl von Kollisionen
   1. Transformation der umliegenden Elementen in den durch das neue Dreieck aufgespannten Raum
   2. Testen des neuen Dreiecks im 2D Space gegen die transformierten Elemente
   3. Falls es eine Kollision gab, identifiziere die beste Kollision
4. Front Management
   1. Falls es in 3. Keine Kollision gab, füge *vNew* zu *f* hinzu und erstelle ein neues Dreieck aus *vNew* und *e*
   2. Sonst,
      1. Falls die Kollision mit der eigenen Front ist, splitte f in zwei neue Fronten
      2. Sonst merge die andere Front mit f

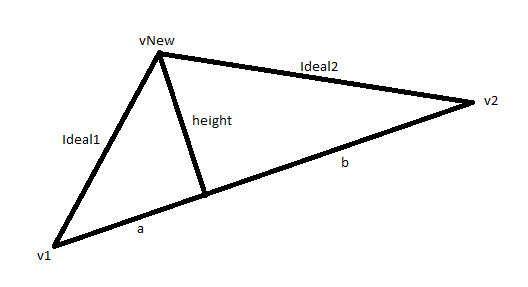
Im Folgenden soll jeder der Teilschritte 1-4 detailliert erläutert werden.

# Identifizierung eines Front-Elements

Obwohl der Code praktisch selbsterklärend ist, gibt es doch einige interessante Details. In der vorliegenden Implementierung werden sowohl die Front, als auch das Front-Element komplett zufällig ausgewählt. Der Algorithmus würde aber prinzipiell mit jedem anderen Auswahlkriterium funktionieren. So könnte man kleinere Fronten bevorzugen, in der Hoffnung den Speicherbedarf gering zu halten. Oder man könnte innerhalb der Fronten zunächst solche Front-Elemente bevorzugen, welche zu Kollisionsfreien, idealen Dreiecken führen und somit die gesamte Dreiecksqualität verbessern. Es wäre auch denkbar die neuen Front-Elemente in einem bestimmten Quadranten des Raums auszuwählen, um Multithreading für den Algorithmus zu ermöglichen.

Wie oben bemerkt wird eine Front entfernt, falls sie nur 3 Elemente enthält und keine Seed-Front ist. Offensichtlich macht es in einem solchen Fall keinen Sinn die Front weiter zu entwickeln, da sie bereits ein fertiges Dreieck darstellt. Es sei angemerkt, dass dieser Schritt die einzige Möglichkeit für den Algorithmus zum Terminieren ist. Der Algorithmus terminiert nämlich gerade dann, wenn keine Fronten mehr vorhanden sind, was weder durch Split noch Merge erreicht werden kann.

# Konstruktion eines neuen Dreiecks



Bei der Konstruktion des neuen Dreiecks kommt das Guidance Field endlich zum Einsatz. Durch Auswerten an den Stellen v1 und v2 gewinnen wir zwei Kreise idealer Kantenlänge. Falls der Abstand d zwischen v1 und v2 kleiner ist als die Summe der idealen Kantenlängen oder der Betrag der Differenz der beiden Kantenlängen ist gibt es keinen Schnittpunkt. Die Erzeugung des neuen Dreiecks ist somit gescheitert und ein neues Front-Element muss gewählt werden. Falls dies allerdings nicht der Fall ist, werden wie in der obigen Abbildung sichtbar die Werte für height, a und b durch Trigonometrie bestimmt. Die normalisierte Summe der Normalen an v1 und v2 liefert uns die Normale der Kante. Das Kreuzprodukt zwischen dieser Normalen und der Richtung der Kante zwischen v1 und v2 liefert uns die Richtung in der vNew liegt. Da wir nun sowohl Richtung als auch Distanz errechnet haben, können wir vNew leicht bestimmen.

Wir nutzen hierbei die Tatsache, dass das Kreuzprodukt sich für ein rechts- oder linkshändiges System immer gleich verhält. Bei konsistenter Anwendung werden also alle neuen Vertices von der Front wegzeigen, genau wie gewünscht.

Im Moment liegt vNew nicht gezwungenermaßen auf der Oberfläche des zu triangulierenden Objektes, da dieses im Allgemeinen gekrümmt sein wird. Wir müssen vNew also geeignet projizieren. Dazu projizieren wir vNew aus X, Y und Z Richtung auf die Oberfläche und wählen den Punkt, der am nächsten am ursprünglichen Punkt lag. Hier wäre Potential für eine Verbesserung des Algorithmus, da man idealerweise in Richtung der Normalen bei vNew oder der ursprünglichen Kante projizieren sollte.

Genau wie die Erzeugung des neuen Vertex kann auch die Projektion scheitern. Da die Projektion in der vorliegenden Implementierung auf Newton basiert, ist sie an sehr ebenen Stellen eher fehleranfällig (f‘(x) = 0, bei Newton problematisch!) und kann somit zu keinem Ergebnis führen. In einem solchen Fall wird der aktuelle Schritt abgebrochen und es wird an einer anderen Stelle weitergemacht.

Wir haben nun ein neues Dreieck gebildet, welches aus Sicht der aktuellen Kante ideal ist. Allerdings ist diese Sicht natürlich sehr limitiert. Es ist leicht möglich, dass vNew sehr nah an anderen Vertices oder Kanten liegt und somit in den Folgeschritten gezwungenermaßen Dreiecke schlechter Qualität erzeugt werden müssen, um die vorherigen Fehler zu kompensieren. Dadurch wird der Ablauf des Algorithmus nicht gefährdet, aber das erzeugte Mesh ist von schlechter Qualität. Im Ausblick der Arbeit werden daher mögliche Strategien zur besseren Auswahl von neuen Vertices diskutiert.

# Erkennung und Auswahl von Kollisionen

Für die Terminierung des Algorithmus ist robuste Kollisionserkennung absolut kritisch: Falls eine Kollision nicht erkannt oder unter mehreren falsch ausgewählt wird, terminiert der Algorithmus in den allermeisten Fällen nicht. Stattdessen bildet sich eine neue Schicht von Fronten auf dem bereits triangulierten Bereich und der Algorithmus beginnt von neuem die Oberfläche zu triangulieren. Auch wenn dieser zweite Anlauf terminieren sollte, wäre er völlig unbrauchbar. Gleichzeitig nehmen die Kosten mit höherer Sicherheit stark zu, weshalb ein ausreichender Mittelweg gefunden werden muss.

Die Kollisionserkennung besteht außer der eigentlichen Findung der Kollisionen auch aus der Auswahl einer idealen Kollision, welche nicht zu neuen Kollisionen führt. Auch dieser Schritt ist extrem kritisch.

## Erkennung

In der vorliegenden Implementierung wurde die Kollisionserkennung zur Verbesserung der Performanz in einem geeigneten 2D-Space durchgeführt. Zunächst wird also eine geeignete Transformation berechnet.

/\*

Code: Get2DTransformation

\*/

Das neue Koordinatensystem hat seinen Ursprung in vNew und wird durch die Vektoren von der ursprünglichen Kante zu vNew aufgespannt. Die Z-Koordinaten der Vertices des neuen Dreiecks sind also alle Null.

In der Methode *TestTriangle* wird unser neues Dreieck nun auf alle möglichen Arten von Kollisionen geprüft. Es müssen allerdings zunächst alle relevanten Kanten, mit denen es zu Kollisionen kommen könnte, gefunden werden. Dazu werden einfach alle Kanten auf ihre Distanz zum neuen Vertex geprüft. Liegt die Kante in einem von der idealen Kantenlänge abhängigen Radius und entspricht nicht der ursprünglichen Kante aus v1 und v2, so ist diese Kante für uns relevant. Schließlich werden alle gefunden Kanten, sowie die Kanten des neuen Dreiecks in den zuvor berechneten 2D-Space transformiert.

Nun werden die gefunden Kanten gegen die zwei neuen Kanten ((vNew, v1); (vNew, v2)) des Dreiecks getestet. Ein entscheidender Aspekt ist hier der Test auf die Z-Koordinaten der einzelnen Kanten. Durch die Projektion kann es bei Objekten mit mehreren, nahe liegenden Schichten zu ungewünschten Kollisionen zwischen den Schichten kommen. Um Dieses Phänomen zu bekämpfen werden die Z-Koordinaten, welche nun direkt den Abstand von der Dreiecksebene angeben, getestet. Falls eine Kante Z-Koordinaten größer und kleiner Null aufweist ist sie für uns relevant, da sie die Dreiecksebene schneidet. Falls sie in einem Bereich der Akzeptanz liegt, ist sie ebenfalls relevant. Dieser Bereich muss mit Bedacht gewählt werden und eventuell für verschiedene Volumen angepasst werden. Ist er zu groß kann es sehr leicht zu Kollisionen zwischen Schichten führen, welche nicht erwünscht sind. Ist er allerdings zu klein kann es in Gebieten hoher Krümmung dazu kommen, dass die Oberfläche mehrfach trianguliert wird, da fälschlicherweise keine Kollision erkannt wird.

/\*

Code: TestZCoordinate

\*/

Der eigentliche Test auf Kollision zwischen zwei Kanten entspricht weitgehend den gängigen Formeln. Man könnte sich hier noch zusätzlich die Tatsache zu Nutze machen, dass vNew immer im Ursprung liegt und somit einige Element der Ausdrücke direkt wegfallen würden.

/\*

Code: GetIntersection

\*/

In seltenen Fällen kann unser neues Dreieck eine komplette Front umschließen. In einem solchen Fall wird der obige Test keine Kollision liefern, was zur doppelten Triangulierung des entsprechenden Bereichs führen würde. Daher testen wir die Vertices der gefundenen Kanten darauf, ob sie sich innerhalb des neuen Dreiecks befinden. Alle auf diese Weise gefunden, neuen Kollisionen werden speziell gekennzeichnet. Diese Kollisionen werden im folgenden Abschnitt interessant für uns werden, da sie uns eine konservative, sichere Triangulierung ermöglichen.

## Auswahl

Zu diesem Zeitpunkt haben wir alle relevanten Elemente gefunden und auf alle möglichen Kollisionen getestet. Falls die Menge der Kollisionen Null ist, beendet die Methode hier und ein neues Dreieck wird der Triangulierung hinzugefügt. Falls dies nicht der Fall ist, muss ein zu einer Kollision zugehöriges Front-Element ausgewählt werden, welches den folgenden Kriterien genügt:

* Das Element darf nicht Teil des neuen Dreiecks, oder direkt adjazent sein.
* Der Winkel zwischen der Kante zum Element der ursprünglichen Kante muss in [0°, 180°] liegen.
* Die Distanz zu den Elementen der ursprünglichen Kante sollte minimal sein, wobei im Zweifelsfall Kollisionen innerhalb des neuen Dreiecks immer bevorzugt werden und ansonsten Split-Operationen Merge-Operationen vorgezogen werden.

/\*

Code: FindBestIntersection

\*/

Ausblick und Diskussion

Blub.