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

Der Schlüssel zu einer effizienten Triangulierung liegt in ihrer Adaptivität. Die Größe von Dreiecken sollte in irgendeiner Form antiproportional zur lokalen Krümmung sein. Einige frühere Implementationen verließen sich darauf, die lokale Krümmung zu berechnen und direkt zu verwenden, um die Dreiecksgröße zu bestimmen [1, 13, 26]. Allerdings kann eine solche Implementation nicht die Erfassung aller Features garantieren, da die Größe angepasst werden muss **bevor** der kritische Bereich erreicht wird. Daher wurden Algorithmen entwickelt, die sich der Heuristik bedienen, um früh genug die Dreiecksgröße zu reduzieren [26].

In der vorliegenden Implementierung verwenden wir ein anderes Konstrukt, das durch Schreiner et al. eingeführte Guidance Field [45]. Prinzipiell ist es als eine Skalar-Funktion zu verstehen, welche im ganzen Raum definiert ist und jedem Punkt eine ideale Kantenlänge zuordnet. Seine Konstruktion teilt sich in die folgenden Schritte auf.

1. Implementierung der theoretischen Grundlagen
2. Sampling
3. Auswertung

# Theoretische Grundlagen

Wir beginnen mit der Definition einer Funktion welche uns die ideale Kantenlänge am Punkt s liefert.

l(s) = (2\*sin(p/2))/Kmax(s)

Die Funktion setzt die ideale Kantenlänge auf den durch den Winkel p beschrieben Teil auf dem minimalen Kreis an der Stelle s. p ist durch den Anwender definiert, die lokale, maximale Krümmung Kmax müssen wir bestimmen. Dies ist allerdings verhältnismäßig teuer, weshalb das Guidance Field nur an diskreten Sample-Punkten errechnet wird. Aus diesem Grund müssen wir eine Funktion definieren, die sinnvoll zwischen den einzelnen Samples interpoliert. Diese Funktion ist gs(x) mit

gs(x) = (1-n^-1) \* |x-s| + n-^1 \* l(s).

Die Funktion beschreibt einen Kegel, dessen Zentrum in s liegt. Es ist zu beachten, dass gs(x) < l(s). Da die lokale Dreiecksgröße möglichst konservativ und damit durch das Sample mit der kleinsten idealen Kantenlänge definiert wird setzen wir schließlich das Guidance Field g(x) fest als

g(x) = min (s € L) gs(x).

Das Guidance Field an der Stelle x ist also das Minimum aller induzierten gs(x). Da die relevanten Samples in der unmittelbaren Nähe von x liegen müssen (|x-s|) ist die Auswertung des von g(x) am besten über Kd oder Octree Anfragen zu berechnen [1].

Es bleibt die Frage nach der eigentlich entscheidenden Information, der Krümmung. Kindlmann et al. [27] zeigen in ihrer Arbeit wie man den sogenannten Tensor G einer impliziten Oberfläche berechnet. Sei f(x) eine geeignete Beschreibung der Oberfläche, wie zum Beispiel die im vorherigen Kapitel vorgestellten Splines, dann ist mit

n = (DELTA f)/ |DELTA f|

und

P = I – nn^T

Sowie dem Hessian

H = […]

Der Tensor G definiert durch

G = PHP / | DELTA f|

G enkodiert die gesamte Krümmungsinformation, daher können wir über Matrix Invarianten Kmax an der Stelle s bestimmen und haben somit ein Sample gefunden [1, 27].

/\*

Code: Guidance Field

\*/

# Sampling

Wie bereits angemerkt ist es nötig die Oberfläche diskret zu sampeln. Daher erzeugen wir an zufälligen Positionen in der Nähe der Oberfläche Samples und projizieren sie mit dem zuvor definierten Projektions-Operator auf die Oberfläche. In der vorliegenden Implementation wurden zu diesem Zweck nur solche Zellen verwendet, welche tatsächlich Splines enthalten die nicht über die komplette Zelle 0 sind. In diesen Zellen wird eine bestimmte Anzahl Samples in X-, Y- und Z-Richtung auf die Oberfläche projiziert um eine weitgehend gleiche Verteilung der Samples zu ermöglichen. Idealerweise würde man die Samples an arbiträren Axen projizieren.

/\*

Code: Projektion

\*/

# Auswertung

Unglücklicherweise kann das Guidance Field nicht komplett vor dem eigentlichen Algorithmus berechnet werden, da die induzierten gs(x) von x abhängig sind. Die effiziente Auswertung ist allerdings entscheidend, da man das Guidance Field für jedes neue Vertex auswerten muss. Naiv könnte man einfach alle gs(x) € L für ein gegebenes x berechnen, dies wäre allerdings viel zu teuer [1]. Daher wurde in dieser Arbeit ein Octree mit Range-Queries verwendet. Im Fall einer Auswertung wird einfach eine Anfrage an den Octree mit der entsprechenden Reichweite gemacht. Dieser liefert alle Samples in der gegebenen Entfernung und errechnet dann das minimale gs(x).

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 dieses 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. Wiederhole 3.2 und 3.3 mit dem Dreieck, dass durch die ursprüngliche Kante und die beste Kollision gebildet werden würde.
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

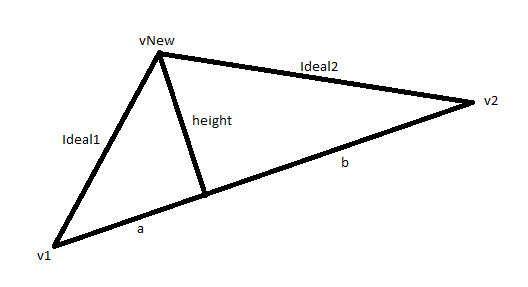
Im Folgenden soll jeder der Teilschritte 1-4 detailliert erläutert werden.

# 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 zu einem der Vertices der ursprünglichen Kante sein.
* Der Winkel zwischen der Kante zum Element und 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.

Im Moment liegen uns die Kollisionen aber nur als eine List von Kanten und der Koordinate des Schnittpunktes auf dieser Kante vor. Um eine Korrekte Merge- oder Split-Operation durchführen zu können müssen wir uns für eines der Elemente der Kanten, im Folgenden bestes Kollisions-Element, entscheiden.

/\*

Code: FindBestIntersection

\*/

Der Prozess der Identifizierung des Besten Kollisions-Elements teilt sich im Wesentlichen in zwei Schritte auf. Zunächst muss geprüft werden, ob die aktuelle Kollision besser ist, als die bisherige Beste. Dabei werden Kollisionen innerhalb des neuen Dreiecks immer und Kollisionen mit der eigenen Front über solche mit fremden Fronten bevorzugt. Dies dient dazu den Algorithmus konservativer und sicherer zu machen. Falls eine Kollision den obigen Kriterien genügt so werden im nächsten Schritt ihre beiden Vertices auf die Winkel- und Adjazenzbedingung geprüft. Im Allgemeinen wird in diesem Schritt einfach das nähere Element der Kollisions-Kante verwendet.

## Abschließende Tests

Das Dreieck aus der zuvor gefunden besten Kollision und der ursprünglichen Kante wird in den allermeisten Fällen korrekt sein. Allerdings gibt es einige, wenige Fälle in denen dieses neue Dreieck wiederum zu neuen Kollisionen führt. Daher testen wir abschließend dieses Dreieck nach genau den gleichen Kriterien wie zuvor.

# Front Management

Wir haben nun ein definitiv gültiges, neues Front-Element gefunden, haben seine eventuelle Beste Kollision identifiziert und können mit dieser Information unsere Front erweitern. Es sei angemerkt, dass die gesamte Front Klasse kein Verständnis von Dreiecken oder dem Raum als solchem benötigt. Die Fronten sind ausschließlich logische Strukturen.

Falls die vorherigen Schritte nicht zu einer Kollision geführt haben, so fügen wir das neue Front-Element in der entsprechenden Front ein und fügen der Triangulierung ein neues Dreieck hinzu.

Falls dies nicht der Fall ist, prüfen wir, ob die Kollision mit unserer eigenen oder einer fremden Front stattgefunden hat. Im ersteren Fall müssen wir unsere Front splitten, sonst verbinden wir sie mit der anderen. Im Folgenden sei *origin* das Front Element unserer ursprünglichen Kante und *intersect* das beste Kollisions-Element.

## Merge

Die Merge Operation stellt neben der Front aus nur drei Elementen die einzige Möglichkeit dar, eine Front zu entfernen. Für den Merge kopieren wir einfach alle Elemente der anderen Front von *intersect* bis zu ihrem Ende direkt vor *origin*. Dann kopieren wir alle verbleibenden Elemente, nämlich vom Beginn der anderen Front bis zu ihrem Ende ebenfalls direkt vor *origin*. Da der Merge zu keinen neuen Kanten führen soll, müssen wir eine Kopie von *intersect* zu unserer Front hinzufügen, wiederum direkt vor *origin*. Man beachte, dass durch diesen Schritte keine neuen Dreiecke entstanden sind, nur die Fronten haben sich verändert.

## Split

Die Split-Operation ist geringfügig komplexer, basiert aber auf den gleichen Prinzipien wie der Merge. Neue Fronten können ausschließlich durch Split-Operationen entstehen. Für den Split erstellen wir zunächst eine neue Front *fNew*. Nun entfernen wir die Elemente von intersect bis origin aus unserer Front und fügen sie zu *f2* hinzu. Sollten wir zwischendurch das Ende unserer Front erreichen, so müssen wir zusätzlich die Elemente vom Ende bis *origin* zu *f2* hinzufügen.

Um den jeweiligen Kreis wieder zu schließen fügen wir unserer Front eine Kopie von *intersect* direkt vor *origin* und *f2* eine Kopie von *origin* direkt vor *intersect* hinzu. Sowohl Split als auch Merge sollten nie zu Fronten führen, die weniger als 3 Elemente enthalten. Falls dies geschieht wurde eine Kollision ausgewählt, die nicht den gewünschten Kriterien entspricht.

Optimierung und Erweiterung

Aufgrund zeitlichen Mangels wurden einige gewünschte Features nicht komplett oder nicht ideal implementiert. Dieses Kapitel soll einen Überblick über mögliche Erweiterungen und Verbesserungen der vorliegenden Implementierung liefern. Besonderes Augenmerk wurde auf die Aspekte der Performanz und Qualität der Triangulierung gelegt.

# Performanz

## Splines

In der aktuellen Form teilt sich die Berechnungszeit hauptsächlich in zwei große Bereiche auf: Die Berechnung der Splines und die eigentliche Triangulierung, wobei die Splines deutlich mehr Zeit beanspruchen. Glücklicherweise ist gerade bei diesen sehr viel Potential für Optimierung denkbar. Zunächst einmal ist festzustellen, dass ein einzelner Spline keine Information seines Nachbarn benötigt, man könnte die Berechnung also problemlos parallelisieren. Als nächstes wird bei der Untersuchung der Multivariaten Polynome klar, dass die Implementierung sehr objektorientiert und nicht unbedingt auf Effizienz ausgelegt ist. Die Implementierung der Multivariaten Polynome auf der GPU würde vermutlich zu einer erheblichen Leistungssteigerung führen. Aber auch aus logischer Sicht ist Optimierung denkbar. Betrachten wir noch einmal die Definition der Splines. Die Spline Funktion f wird in einem Gitter der Größe 4^3 mit den Samples pijk definiert durch

f(x, y, z) = Sum\_ijk(pijk \* bi(x) \* bj(y) \* bk(z))

wobei

bi(x) = ([x^3, x^2, x, 1] M)i

Da die einzelnen bi allerdings nur kubische Polynome sind können wir die obere Gleichung in die folgende Form

f(x, y, z) = Sum\_ijk(cijk \* x^i \* y^j \* z^k)

überführen. F hängt jetzt nur noch von den einzelnen Koeffizienten cijk ab, welche wiederum nur von den pijk und M abhängen. Würde man einmal die Berechnungsvorschrift für ein gegebenes M errechnen, so fiele praktisch die komplette Polynomrechnung weg [Schreiner Dissertation].

## Guidance Field

Auch hier bietet sich Parallelisierung sehr an: Die einzelnen Samples können unabhängig voneinander auf die Oberfläche projiziert und danach auch getrennt berechnet werden. Wie in der Arbeit von Schreiner et al. gezeigt können aber auch logische Verbesserungen vorgenommen werden, die zu einer effizienteren Auswertung des Guidance Field führen. Bei der Betrachtung der Auswertung wird klar, dass manche Samples andere in ihrem Umkreis stark dominieren. Ein Sample an einer Stelle mit hoher Krümmung wird für seine unmittelbare Umgebung meistens bestimmend sein. Die Idee ist daher nach dem Sampling solche Proben wieder zu entfernen, die in jedem Fall für die Auswertung nie relevant wären. Das senkt den Speicherverbrauch und die benötigte Zeit zur Auswertung. Prinzipiell könnten wir jedes Sample gegen jedes andere auf Dominanz testen, allerdings wäre dies bei Millionen von Samples mit einer Laufzeit von O(n^2) vermutlich eher ein Leistungsverlust.

Schreiner et al. stellen daher ein Verfahren vor, welches alle Samples effizient auf Dominanz prüft. Zunächst stellen wir fest, dass jedes Sample einen Kegel in R^n+1 definiert, wobei n die Dimension des Space ist und in unserem Fall 3. Jedes unserer gs(x) definiert also einen solchen Kegel im Space [ x, y, z, r]. Wir stellen nun fest, dass falls ein Sample s1 ein anderes Sample s2 dominiert, dann liegt s2 im Kegel von s1. Außerdem ist die Relation transitiv. Falls s2 ein Sample s3 dominiert, so dominiert s1 auch s3. Würden wir also eine geeignete, hierarchische Datenstruktur finden, so könnten wir komplette Teilbereiche der Samples schnell entfernen [1].

Eine solche Struktur ist ein 4-Dimensionaler kd-Baum, wobei jedes der Samples im Baum die Koordinaten [sx, sy, sz, sr] hat. Sr beschreibt den Wert der Funktion gs(x) an der Stelle s. Anschaulich spannt jedes der Samples also einen Kegel an der Stelle [sx, sy, sz, sr] und der Achse [0, 0, 0, 1] auf. Der Winkel zur Achse wird durch gs(x) gegeben und beträgt tan^-1(n/n^-1). Es sind also außer den Beiden bekannten Parametern p und n keine zusätzlichen Informationen nötig. Somit reduziert sich die Suche nach den durch ein Sample s dominierten Samples auf eine einfach Range-Query Anfrage an den kd-Baum [1].

Die Schnelligkeit der Reduktion der Samples hängt allerdings davon ab wie geschickt wir unsere Samples testen. Manche Tests führen dazu, dass sehr große Teile des Baumes wegfallen, anderen haben überhaupt keinen Effekt. Offensichtlich gute Kandidaten sind allerdings solche mit hoher Krümmung, also niedrigem gs(x). Daher sortieren wir zunächst alle Samples nach den sr und führen die Reduktion dann durch [1].

/\*

Grafik Schreiner Tabelle Culling

\*/

Schreiner et al. erreichen durch die oben vorgestellte Prozedur eine Reduktion der Samples um bis zu 95% und Leistungsgewinne zwischen 20% und 80%. Vor allem bei Datensätzen mit Millionen von Samples sind die Leistungssteigerungen signifikant.

## Triangulierung

Der eigentliche Algorithmus ist im Vergleich zur restlichen Implementierung bereits relativ performant, aber auch hier sind Optimierungen möglich. Allerdings ist Vorsicht geboten, da in diesem Bereich der Fokus unbedingt auf Lesbarkeit und Verständlichkeit des Codes gelegt werden sollte. Eine offensichtliche Verbesserung wäre eine bessere Datenstruktur zur Speicherung der Front-Elemente. Da die Anzahl der gleichzeitig existierenden Front-Elemente im Vergleich zu beispielsweise der Zahl der Samples verhältnismäßig gering ist, werden bei der Suche nach kollidierenden Kanten zur Zeit alle Kanten auf ihre Distanz zum zu prüfenden Element getestet. Sinnvoll wäre hier ein Octree oder eine vergleichbare Datenstruktur, welche Range-Queries erlaubt.

Es wäre sehr wünschenswert bei der Durchführung der Triangulierung Parallelisierung einsetzen zu können. Allerdings ist ein Schritt des Algorithmus alles andere als atomar, weshalb Maßnahmen ergriffen werden müssen um paralleles Triangulieren zu ermöglichen. Zunächst einmal wäre eine räumliche Trennung denkbar. Durch geschicktes Wählen der Seeds, könnte jeder Thread zunächst seinen eigenen Bereich triangulieren. Wird Nähe zu einem fremden, bereits triangulierten Bereich festgestellt, so fügt einer der Threads seine Fronten dem anderen hinzu und gibt die Arbeit auf. Auf diese Weise untätig gewordene Threads könnten wieder eingesetzt werden indem neue Seeds in bisher nicht triangulierten Bereichen hinzugefügt werden. Die Distanz zwischen arbeitenden Threads sollte großzügig gewählt werden, da sonst eventuell deutliche Kanten zwischen den Teilbereichen sichtbar werden. Die Parallelisierung des Algorithmus könnte auch durch eine geschicktere Auswahl der zu bearbeitenden Front-Elemente erleichtert werden: Zurzeit werden einfach zufällig Elemente ausgewählt. Allerdings könnte man auch gezielt Elemente aus bestimmten Blocks auswählen. Die Blocks sollten natürlich nicht überlappen, sondern wiederum eine genügend große Zone zur Verbindung freilassen. Sobald alle adjazenten Blöcke eines Blockes trianguliert sind kann dann ein einzelner Thread die Triangulierung des Bereiches leicht beenden indem ihm einfach die offenen Fronten des Blockes zugwiesen werden. Verbindung und Schließung der Lücken sind in diesem Fall sehr dankbar, da sie bereits Teil des eigentlichen Algorithmus sind.

# Qualität der Triangulierung

Ausblick und Diskussion

Blub.