UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Echte Displacement Mapping Verfahren mit Hilfe des Geometry Shaders

Diplomarbeit

zur Erlangung des Grades eines Diplom-Informatikers
im Studiengang Computervisualistik

vorgelegt von
Wolfram Meffert

Erstgutachter:	Prof. Dr.-Ing. Stefan Müller (Institut für Computervisualistik, AG Computergraphik)
Zweitgutachter:	Dipl.-Inform. Niklas Henrich (Institut für Computervisualistik, AG Computergraphik)

Koblenz, im Januar 2009

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

.....
(Ort, Datum) (Unterschrift)

Institut für Computervisualistik
AG Computergraphik
Prof. Dr. Stefan Müller
Postfach 20 16 02
56 016 Koblenz
Tel.: 0261-287-2727
Fax: 0261-287-2735
E-Mail: stefanm@uni-koblenz.de

UNIVERSITÄT
KOBLENZ · LANDAU

Fachbereich 4: Informatik

Aufgabenstellung für die Diplomarbeit
Wolfram Meffert
(Matr.-Nr. 203 110 026)

Thema: **Echte Displacement Mapping Verfahren mit Hilfe des Geometry Shaders**

Die Idee des Displacement Mappings ist 30 Jahre alt und wurde im Laufe der Zeit immer weiter entwickelt, verbessert und realistischer. Die meisten Verfahren basieren auf der Simulation von Geometrie, d.h. das Displacement Mapping Verfahren rechnet auf speziellen Texturen, um den Eindruck von Tiefe zu verstärken. Echte Displacement Mapping Verfahren verändern tatsächlich die Geometrie, doch davon wurde in der Vergangenheit Abstand genommen, da die Berechnung auf der CPU zu aufwändig war bzw. das Ergebnis nicht den Aufwand rechtfertigte. Dazu mussten bisher die Punkte von der Applikation im geeigneten Detailgrad erzeugt werden. Die Verfahren der Geometrieerzeugung sind daher weniger performant. Außerdem können die texturbasierten Simulationsverfahren bereits blickwinkelabhängige Effekte, Selbstverschattungen und auch Selbstverdeckungen darstellen. Als Nachteil sind die Randprobleme bei diesen Verfahren zu nennen. Dennoch hat Displacement Mapping mit diesen Verfahren zunehmend Einzug in die Spieleindustrie gehalten, für die es bisher zu rechenaufwändig war. Hier bieten die neusten Erweiterungen der GPU interessante Potentiale, da mit Geometry Shadern die Möglichkeit besteht, echte Geometrie beschleunigt und adaptiv zu erzeugen.

Ziel dieser Arbeit ist es, verschiedene Verfahren für echtes Displacement Mapping mit dem Geometry Shader zu untersuchen. Dabei wird neue Geometrie aus einer Textur heraus erzeugt. Hierbei werden Punkte und Polygone generiert, wobei bei Polygonen noch ein weiterer Meshing-Schritt folgt, der die Punkte zu Dreiecken verbindet. In der Textur werden neben den Normalen noch die Höhenwerte im Alphawert gespeichert. Außerdem wird noch ein Verfahren untersucht, welches Volumeninformationen simuliert, um Überhänge und schwebende Objekte zu erzeugen.

Schwerpunkte:

1. Literaturrecherche und Analyse der existierenden Ansätze.
2. Konzeption und Entwurf
3. Implementation
4. Test und Bewertung
5. Dokumentation der Ergebnisse

S. Müller

Koblenz, den 09.06.08

Prof. Dr. Stefan Müller

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Motivation . . . 1
- 1.2 Problemstellung . . . 1
- 1.3 Zielsetzung . . . 1

2 Grundlagen **3**
- 2.1 Geschichte des Displacement Mapping . . . 3
  - 2.1.1 View-Dependent Displacement Mapping . . . 3
  - 2.1.2 Parallax Mapping . . . 4
  - 2.1.3 Parallax Occlusion Mapping . . . 4
  - 2.1.4 Direct X 10 Displacement Mapping . . . 6
- 2.2 Shadermodel 4.0 . . . 6
  - 2.2.1 Vertex Buffer Objekt . . . 9
  - 2.2.2 Transform Feedback . . . 9
- 2.3 Bump Mapping . . . 11
- 2.4 Level of Detail . . . 12

3 Verfahren **15**
- 3.1 Einzelshader ohne Transform Feedback . . . 15
- 3.2 Einzelshader nur mit Punkten . . . 16
- 3.3 Zwei Shader mit Transform Feedback . . . 17
- 3.4 Volumen . . . 18
  - 3.4.1 Datenstrukturen . . . 19
  - 3.4.2 Meshing . . . 21
  - 3.4.3 Marching Cubes auf der GPU mit Einzelpunktinput . 25
- 3.5 LoD . . . 25
  - 3.5.1 Spalten und Schürzen . . . 29

4 Implementierung **31**
- 4.1 Framework . . . 31
- 4.2 Fps-Messung . . . 31
- 4.3 Geometrie . . . 33
- 4.4 Shaderverwaltung . . . 33
- 4.5 Transform Feedback . . . 34
- 4.6 Umsetzung der Displacement Mapping Verfahren . . . 35
  - 4.6.1 Einzelner Shader ohne Transform Feedback . . . 35
  - 4.6.2 Punktshader . . . 37
  - 4.6.3 Subdivision mit Transform Feedback . . . 38
- 4.7 Bump Mapping . . . 39
- 4.8 Volumen . . . 40
- 4.9 LoD . . . 41
  - 4.9.1 Spalten und Schürzen . . . 43

4.9.2 In Bezug auf Volumen 44

5 Ergebnisse und Bewertung **45**
5.1 Testbedingungen 45
5.2 Ergebnisse 45
5.3 Punktvolumen 52

6 Ausblick **55**

Abbildungsverzeichnis

1 Displacement Mapping mit Grundgeometrie in rot 2
2 Skizze des Parallax Occlusion Mappings 5
3 Beispiele für Treppcheneffekte [Zin] und Silhouettenproblem [BT04] 5
4 Direct X 10 Displacement Mapping [Tri] 6
5 Skizze Rendering Pipeline in Shadermodel 4.0 7
6 Dreieck mit Nachbarschaftsinformationen 8
7 Texturbeispiel Tangentenraum 12
8 Texturbeispiel Objektraum[Kre] 12
9 Nahaufnahme von Schürzen an der Oberfläche 13
10 Beispiel für Lücken und Schemaskizze mit Schürze 14
11 Beispiel für Punktausgabe 16
12 Skizze zur Interpolation und Subdivision 18
13 Volumen Displacement Mapping mit Punkten(links), zugehörige Textur(rechts) 19
14 Unterschied der Textur aus Abb. 13 in Schichten(links) und ineinander(rechts) 21
15 Beispiel für Farbnachbarschaft 24
16 Die Bruteforce Version der Schürzen für Level of Detail . . . 27
17 Skizze der LoD Grenze 29
18 Klassendiagramm 32
19 Skizze zu Variablen und Schleifenstruktur des LoopShaders 37
20 Skizze zur besseren Nutzung von redundanten Punkten . . 42
21 Die beiden Texturen für die Tests 45
22 Tabelle 47
23 Diagramm zur Tabelle Teil 1 48
24 Diagramm zur Tabelle Teil 2 49
25 Schürzen ohne Folgeunterteilung 50
26 Die Shürzen wurden auf sinnvolle Bereiche beschränkt . . . 51
27 Vergleich von Parallax Occlusion Mapping [Zin](links) mit Displacement Mapping(rechts) 52
28 Nahaufnahme der Probleme bei Punktvolumen 53
29 Beispiel für sichtbare Grundgeometrie 56

1 Einleitung

1.1 Motivation

In modernen Grafikengines werden mehr und mehr Techniken eingesetzt um eine höhere Geometrieauflösung darzustellen als tatsächlich vorhanden ist. Angefangen mit verschiedenen Bump Mapping Verfahren wie Normal Mapping bis hin zu Parallax Occlusion Mapping. Diese Verfahren bieten bereits blickwinkelabhängige Beleuchtung, Selbstverschattung und Selbstverdeckung bei guter Performanz. Doch andererseits behalten all diese Verfahren ihre ursprünglich zugrunde liegenden Oberflächen, was zu unerwünschten Grafikeffekten führt. Eine Kugel zum Beispiel behält die Silhouette einer Kugel, egal wie uneben sie auch bei senkrechtem Betrachtungswinkel wirkt. Beispielhaft in Abb. 3 (rechts) zu sehen. Diese Nachteile müssen entweder in irgendeiner Weise kaschiert werden oder die Anwendungsbereiche bleiben eingeschränkt. Ein anderer Ansatz liegt im echten Displacement Mapping, welches tatsächliche Geometrie benutzt, um im Endeffekt sogar die Silhouette einer Kugel zu verändern. Durch die Verwendung echter Geometrie können auch moderne Partikelsimulationen darauf angewendet werden.

1.2 Problemstellung

Das Problem bei Displacement Mapping ist allerdings, dass bisher keine neue Geometrie erzeugt werden konnte und daher die nötige Geometrieauflösung schon vorher vorhanden sein musste. Wenn die notwendige Geometrie aber schon bestehen muss, besteht keine Notwendigkeit für Displacement Mapping bzw. die Eckpunkte erst zur Laufzeit an die richtige Position zu bringen. Daher ist Displacement Mapping bisher keine akzeptable Alternative gewesen. Doch mit der Einführung des Shadermodels 4.0 stehen Grafikprogrammierern ganz neue Möglichkeiten offen. So können erstmals mit dem Geometry Shader aus einem einzigen Punkt viele neue generiert werden. Allerdings ist die GPU Programmierung vielen Einschränkungen unterlegen, die beachtet werden müssen.

1.3 Zielsetzung

Ziel dieser Diplomarbeit ist es, die Schwierigkeiten und Möglichkeiten im Umgang mit dem Geometry Shader zur Implementierung eines echten Displacement Mapping Verfahrens zu untersuchen. Die Informationen, die nötig sind um die neue Geometrie zu erzeugen, sollen dabei wie in ähnlichen Verfahren allgemein üblich, in einer Textur abgespeichert werden. Es werden mögliche Verfahren sowohl mit Dreiecken als auch ausschließlich mit Punkten untersucht, ob und wie ein Displacement Mapping möglich ist,

während der Geschwindigkeitsverlust akzeptabel bleibt. Außerdem welche Probleme dabei auftreten können, wenn gängige Verfahren wie Level of Detail oder Phongbeleuchtung implementiert werden. Zuletzt wird noch ein Verfahren untersucht, welches Volumeninformationen simuliert, um Überhänge und schwebende Objekte zu erzeugen.

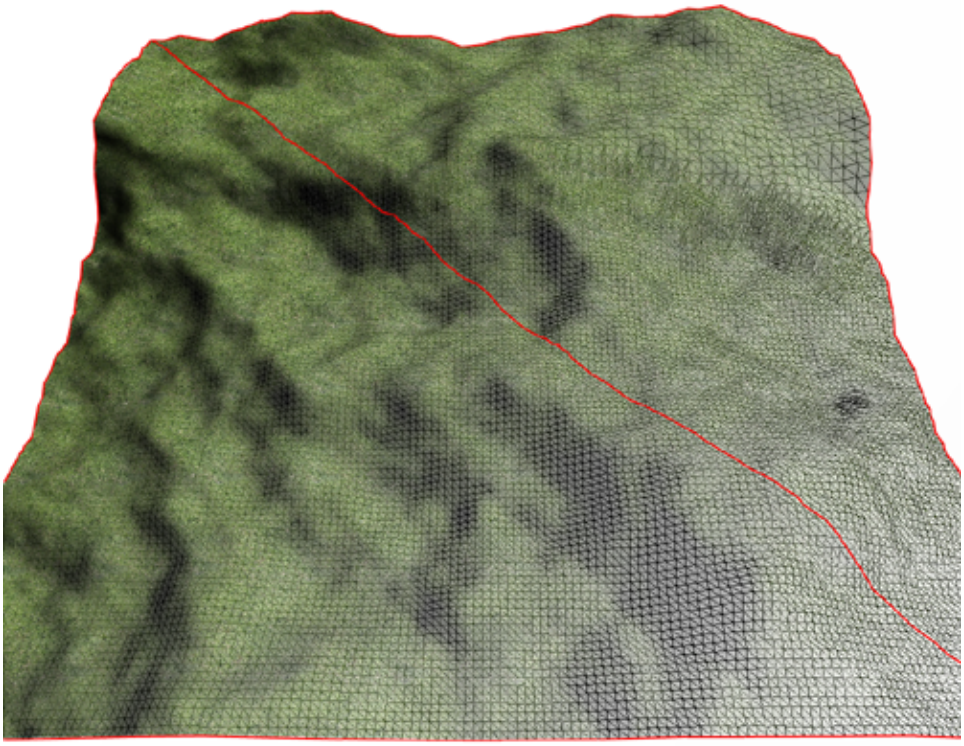

Abbildung 1: Displacement Mapping mit Grundgeometrie in rot

2 Grundlagen

In diesem Kapitel werden die benötigten Vorraussetzungen sowohl von Hardware und Software, als auch die theoretischen Grundlagen der Arbeit erörtert. Dazu gibt es auch eine Übersicht über bisherige bekannte Ansätze und deren Vor- und Nachteile.

2.1 Geschichte des Displacement Mapping

Das erste echte Displacement Verfahren wurde 1984 von Cook beschrieben. [Coo84] Dabei werden Heightmaps verwendet, um Geometrie entlang ihrer Normalen, entsprechend eines Höhenwertes zu verschieben. Allerdings muss das Mesh vor dem Beginn des Renderpasses die entsprechende Auflösung besitzen. Damals wurde die Geometrie ohnehin sehr fein unterteilt, weil es kein Textur Mapping gab. Heute bedeutet dies aber eine hohe Belastung für die Rechenleistung und den Speicherverbrauch, so dass eine Verwendung für Echtzeitsysteme nicht in Frage kommt. Beziehungsweise könnte ein solches Mesh, welches sich nicht dynamisch ändert, in diesem Fall ebenso gut in einem Vorverarbeitungsschritt berechnet werden, um dann zur eigentlichen Laufzeit ohne die Berechnungen zur Verschiebung dargestellt werden zu können. Außerdem können den Vertices vor Übergabe an die Pipeline keine Beleuchtungsnormalen zugewiesen werden, denn durch das Displacement ist unklar wie die Tangente aussehen wird, bzw. in welcher Neigung sich das zugehörige Dreieck zur Lichtquelle befinden wird. Daher müssen die Normalen nachträglich berechnet werden. Seit der Erfindung des Normal Mapping kann dieser Schritt auch durch Vorberechnung beschleunigt werden. Aufgrund der hohen Anforderungen von Displacement Mapping zielten andere Ansätze entgegen Cook bisher darauf ab, zusätzliche Geometrie nur zu simulieren, um die aufwändige Bearbeitung hoher Mengen an Vertices zu sparen. Diese Verfahren stützen sich in der Regel auf eine Form von Raytracing.

2.1.1 View-Dependent Displacement Mapping

Dieses Verfahren verwendet eine Form von Raytracing im Fragment Shader, um den Abstand entlang der Blickrichtung von einer Referenzoberfläche zur simulierten Oberfläche zu bestimmen. Mit den ermittelten Werten werden dann im Fragmentshader die Texturkoordinaten so verschoben, dass ein besserer Eindruck von Geometrie entsteht. Selbstverdeckung oder eine höhere Amplitude der Heightmap sind somit möglich. Sogar Silhouetten und Selbstverschattung können dargestellt werden. Allerdings müssen für verschiedene Blickrichtungen und Oberflächenkrümmungen insgesamt fünf Dimensionen bzw. fünf Werte abgespeichert werden. In einer regulären 2D Textur sind aber nur vier Floats verfügbar. Außerdem

verbraucht eine Textur mit 128x128 Pixel und 32x8 Blickwinkeln und 16 Krümmungen bereits 64MB Speicherplatz auf der Grafikkarte. Dieser Speicherplatzverbrauch kann durch Singulärwertzerlegung auf ca. 4MB reduziert werden auf Kosten zusätzlicher Einschränkungen. Die Rekonstruktion der komprimierten Daten muss im Shader durch eine geringe Anzahl an Eigen-Funktionen durchgeführt werden, so dass viele unterschiedliche hohe Frequenzen in der Karte kaum dargestellt werden können. Der Algorithmus ist aufgrund der Hauptlast im Fragmentshader stark von der Anzahl der Pixel der zu verschiebenden Objekte abhängig und daher nur bei vergleichsweise geringen Auflösungen echtzeitfähig.[WWT+03]

2.1.2 Parallax Mapping

Beim einfachen Parallax Mapping, ob mit oder ohne Offset Limiting, wird nur mit Hilfe des Blickvektors und einer Heightmap ein Offset berechnet, um die Texturkoordinate zu verschieben. Der Mittelwert der Heightmap bildet dabei die Oberflächenebene. Doch auch hier werden Artefakte stärker je flacher der Blickwinkel auf die Fläche ist. Es ist aufgrund der einfachen Berechnung nur sehr wenig langsamer als Bump Mapping, bietet aber etwas bessere Ergebnisse bezüglich des Blickwinkels. Selbstverdeckung und Selbstverschattung sind damit nicht möglich. Für die Selbstverschattung fehlt der Bezug zur Lichtquelle und die Textur wird durch den Offset mehr oder weniger stark gestaucht, so dass keine wirkliche Selbstverdeckung entsteht. [KTea01]

2.1.3 Parallax Occlusion Mapping

Diese Technik ist mächtiger als Bump- oder Parallax Mapping. Diese Simulationen oder Techniken konnten nur die Beleuchtung im Phongmodell durch eine Normalmap verändern, die Texturkoordinaten leicht verschieben. So dargestellte Geometrie beschränkte sich auf Furchen oder Beulen etc., also Strukturen die eine niedrige Amplitude besitzen. Doch dadurch war es nicht möglich eine Selbstverschattung oder sogar eine Selbstverdeckung zu erzeugen. Mit Parallax Occlusion Mapping jedoch ist dies beides möglich. Dazu wird wie auch vorher eine Normalmap und eine Heightmap benötigt, die hierbei allerdings im Fragment Shader benutzt wird um ein pro Pixel Raytracing durch die Geometrie zu erlauben. Dadurch wird prinzipiell die volle Auflösung gewährleistet ohne neue Geometrie zu erzeugen. Auf der normalen Geometrie wird zunächst der normale Schnittpunkt bestimmt um dann abhängig von Blickwinkel und Sampleschrittgröße durch die Textur zu laufen und den Schnittpunkt mit der Hightmap zu suchen. Denn die simulierte Geometrie wird als unter der Ursprungsoberfläche liegend angenommen. Wenn der Schnittpunkt mit der Hightmap gefunden ist, wird an dieser Stelle der Textur die Normale und Farbe zur

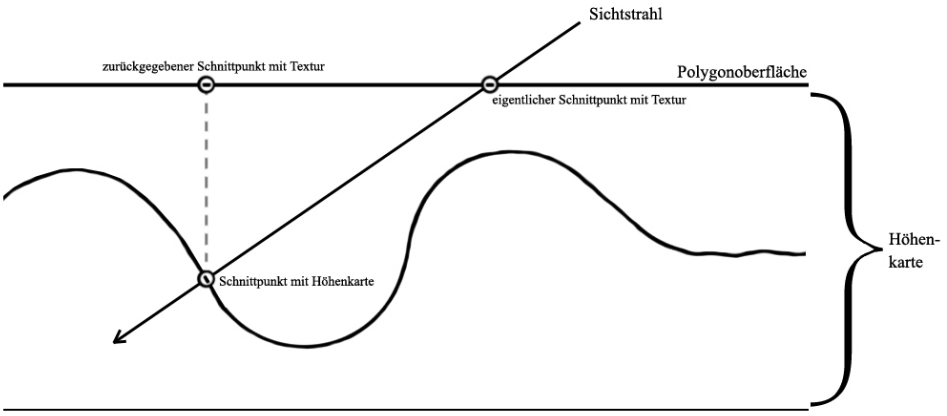

Abbildung 2: Skizze des Parallax Occlusion Mappings

Beleuchtung ausgelesen. Dies ist zur Veranschaulichung in Abbildung 2 zu sehen. Dadurch wird auch erreicht, dass sich die simulierte Geometrie in der Textur selbst verdecken und bei entsprechender Anwendung von oder zu der Lichtquelle auch selbst verschatten kann. Probleme dabei sind vor allen Dingen die Sampleschritte des Strahls, da kein herkömmlicher Schnitttest möglich ist. Bei zu groß gewählter Schrittgröße können kleine Details oder sogar die Fläche an sich verfehlt werden. Bei kleinen Schritten leidet dagegen mehr und mehr die Performanz. Eine weitere Grenze bildet der Rand des Geometrieprimitivs welcher immer noch nicht die Sihouette der dargestellten Geometrie lliefern kann (Abb. 3(rechts)). Treppcheneffekte wie in Abbildung 3(links) können mittlerweile gut vermieden werden.

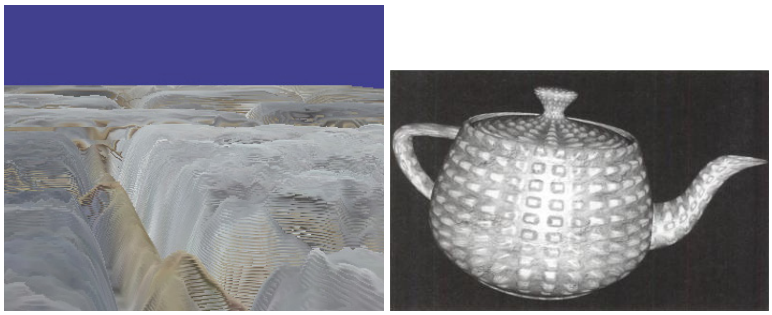

Abbildung 3: Beispiele für Treppcheneffekte [Zin] und Silhouettenproblem [BT04]

Spheretracing bildet da eine gute Möglichkeit. Dabei wird bei dem ersten Samplepunkt die kürzeste Entfernung zur Geometrie gesucht. Falls diese größer als 0 ist, wird mit dem errechneten Abstand als Entfernung der

nächste Samplepunkte auf der Sichtlinie gewählt. So passt sich die Anzahl und Genauigkeit der Sampleschritte der jeweiligen Situation an. Bei schrägem Auftreffen des Sehstrahls muss verhindert werden das die Samples lediglich zum Schnittpunkt konvergieren. [BT04]

2.1.4 Direct X 10 Displacement Mapping

Bei diesem Verfahren wird Prinzipiell die gleiche Technik verwendet wie auch bei Parallax Occlusion Mapping. Es wird jedoch erst dadurch mächtiger, dass mit Hilfe des Geometry Shaders die Eckpunkte des zugrunde liegenden Dreiecks extrudiert werden. Dadurch entsteht ein Prisma block, welcher in drei Tetraeder unterteilt wird, in denen sich die Heightmap dann über der ursprünglichen Ebene im Raum befindet. Dadurch findet das Raytracing nicht in bzw. unter der Ebene statt, sondern darüber (in bzw. unter dem neuen Block) und bietet so die Möglichkeit aus dem Objekt herausragende Geometrie zu simulieren. Allerdings ist durch die bloße Simulation von Geometrie immer noch keine Partikelsimulation darauf möglich. Lediglich die Beschränkung auf die ursprüngliche Ebene wird aufgehoben und die Ränder wirken nicht mehr flach.[WTL+04]

Abbildung 4: Direct X 10 Displacement Mapping [Tri]

2.2 Shadermodel 4.0

Das Shadermodel 4.0 ist in Bezug auf die Funktionalität eine Obermenge des Shadermodels 3.0 mit der Ausnahme, dass es nicht die Features aus Shadermodel 1.0 unterstützt. Es ist mit Grafikkarten ab der Geforce 8000er Reihe oder der ATI HD 2000er Reihe einsetzbar. Aber auch ohne moderne Ausstattung lässt sich das Shadermodel 4.0 mit ältern Karten nutzen. Dazu müssen die fehlenden Fähigkeiten der Karten emuliert werden. Allerdings ist diese Möglichkeit aufgrund fehlender entsprechender Hardware sehr viel langsamer. Die wichtigsten Änderungen im Vertexshader ist die Aufstockung der Anweisungsanzahl von 512 auf 65536 wobei die maximale Anzahl an ausführbaren Anweisungen jedoch bei 65536 bleibt.

Der Unterschied zwischen Anweisungen und ausführbaren Anweisungen ist der, dass es z.B. durch Schleifen mehr Anweisungen auszuführen gibt als explizit im Shaderprogramm angegeben. Also einmal die Anweisungen die im Code stehen und einmal die Anweisungen die zur Laufzeit ausgeführt werden. Im Fragmentshader gibt es ebenso umfangreiche Änderungen. Das Anweisungslimit wurde ebenfalls auf 65536 erhöht und die maximal ausführbaren Anweisungen sind theoretisch unbegrenzt. Die größte und wichtigste Änderung jedoch im Shadermodel 4.0 ist die Einführung des Geometry Shaders. Dieser ist bei der Verwendung von Shadern optional. Er befindet sich in der Pipeline hinter dem Vertex Shader und vor perspektivischer Division und Clipping. Er bearbeitet erstmals komplette

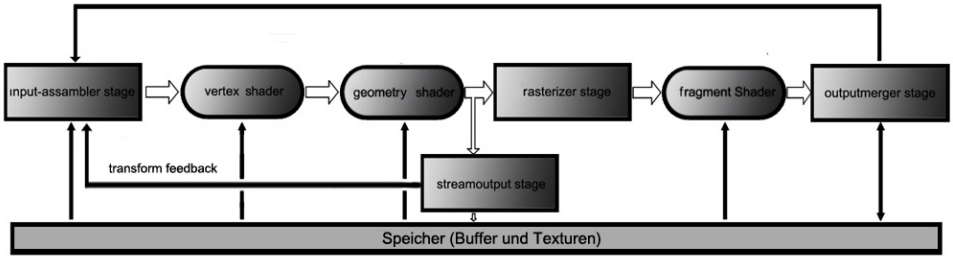

Abbildung 5: Skizze Rendering Pipeline in Shadermodel 4.0

Geometrie Primitive wie:

- Punkte
- Linien
- Linien mit Nachbarschaft
- Dreiecke
- Dreiecke mit Nachbarshaft

GL_LINESTRIP_ADJACENCY und GL_TRIANGLE_ADJACENCY sind dabei diejenigen Modi, welche Nachbarschaftsinformationen besitzen. Die Linien bekommen vor dem ersten Punkt und nach dem letzten einen weiteren. Das Dreieck mit Nachbarschaft besteht aus vier Dreiecken bzw. aus dem eigentlichen zentralen Dreieck und drei Nachbarpunkten, die diese vier Dreiecke repräsentieren. Abb. (6) Es können also bis zu sechs Punkte als Eingabe für einen Geometry Shader dienen. Die Ausgabe ist noch etwas flexibler als die Eingabe. Hier können neben den selben Möglichkeiten wie in der Eingabe, zusätzlich noch umfangreichere Primitive ausgegeben werden.

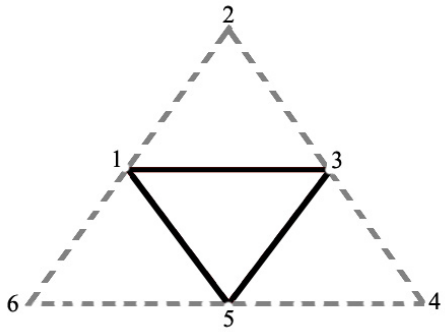

Abbildung 6: Dreieck mit Nachbarschaftsinformationen

- Linienstreifen
- Linienstreifen mit Nachbarschaft
- Dreieckstreifen
- Dreieckstreifen mit Nachbarschaft

Das bedeutet, dass neue und vor allem mehr Geometrie als ursprünglich vorhanden erzeugt werden kann. Zum Beispiel kann als Eingabe ein einzelner Punkt dienen, während die Ausgabe aus fünf Dreiecken und deren Nachbarpunkten besteht. Allerdings ist das Erzeugen von neuen Punkte bzw. Dreiecken nicht unbegrenzt möglich. Denn eigentlich ist der Geometry Shader nicht dafür vorgesehen lediglich Unmengen an neuer Geometrie zu produzieren. Das Maximum an Punktoutput hängt von verschiedenen Faktoren ab. Einmal muss mit GEOMETRY _VERTICES _OUT _EXT zwingend festgelegt werden wieviele Punkte maximal ausgegeben werden dürfen. Dieses Maximum darf aber nicht MAX _GEOMETRY _OUTPUT _VERTICES _EXT und MAX _GEOMETRY _TOTAL _OUTPUT _COMPONENTS _EXT überschreiten, welche Hardware- und Treiberabhängig sind. Das heißt, wenn einem Punkt im Geometry Shader weitere Werte zugewiesen werden wie z.B. Farbe oder Normale, sinkt das effektive Punktmaximum. Das Komponentenmaximum muss mindestens 1024 betragen und liegt bei dieser Arbeit bei eben diesem Limit, womit selbst bei Ausgabe ohne zusätzliche Attribute, höchstens 256 Eckpunkte ausgegeben werden können.[NVI08] Da neue Geometrie erzeugt werden kann, ist die Beleuchtung im Geometry Shader ebenfalls durchführbar. Auch Texturesampling ist im Geometry Shader möglich und das Anweisungsmaximum könnte vollständig für Texturzugriffe ausgeschöpft werden. [Mic]

2.2.1 Vertex Buffer Objekt

Ein so genanntes Vertex Buffer Objekt(VBO) speichert Floatwerte zu Eckpunkten in einem Array, die als Geometrie interpretiert werden können und die Vorteile von Displaylisten und Vertex Arrays ohne deren Nachteile vereinen. Ein Vertex Array kann die Anzahl der Funktionsaufrufe und redundante Benutzung der Vertices reduzieren. Allerdings müssen die Daten bei jeder Referenzierung an den Server geschickt werden, da sich die Vertex Array Funktionen im Client-Zustand befinden. Die Bandbreite zwischen Server und Client ist beschränkt, so dass sich dieser Übertragungsweg mit steigender Datenmenge schnell zu einem Nachteil auswächst. Die Displaylisten auf der anderen Seite sind serverseitige Funktionen und müssen daher nicht übertragen werden. Dafür kann eine Displayliste nach abgeschlossener Erstellung nicht wieder modifiziert werden. Vertex Buffer Objekte sind aber Buffer Objekte für Vertex Attribute auf dem RAM Speicher der Grafikkarte bzw. auf der Serverseite, die dennoch die selben Zugangsmöglichkeiten besitzen wie auch die Vertex Arrays. Aber entgegen Displaylisten kann ein VBO ausgelesen und aktualisiert werden, indem es in den Speicher des Clients gemapped wird. Dadurch ist es ein weiterer Vorteil des VBO, dass sich mehrere Clients das Objekt teilen können, genau wie Displaylisten oder Texturen. Vorausgesetzt, es wird mit dem entsprechenden Bezeichner abgerufen. Der Speichermanager des VBO bringt das Buffer Objekt auf den besten Platz im Speicher durch Benutzerhinweise wie target- oder usage-Modus. Hierfür kann der Speichermanager die Buffer zwischen Systemspeicher, AGP Speicher und Videospeicher balancieren. In einem VBO im Interleaved-Modus werden die Floatwerte in einer von OpenGL fest vorgeschriebenen Reihenfolge eingetragen. Die Interpretationsmöglichkeiten für das VBO beinhalten die Position, Farbe, Normale und Texturkoordinate des Vertices. An der Stelle an der das VBO benutzt werden soll, z.B. wenn die Daten in die Pipeline übergeben werden sollen, muss mit einem Befehl wie z.B. GL_T2F_C4F_N3F_V3F mitgeteilt werden, in welcher Reihenfolge die Floats die einzelnen Vertices beschreiben. Dieser Befehl hier besagt, dass vier Eigenschaften in der Reihenfolge Texturkoordinate mit zwei Floats, Farbe mit vier Floats, Normalemit drei Floats und Vertexposition mit drei Floats gespeichert sind. Da die Floats ohne zusätzliche Informationen oder einen Index im Buffer abgelegt sind, können für einzelne Vertices keine Eigenschaften ausgelassen werden. Alternativ können auch bis zu vier einzelne VBO's erstellt werden, die jeweils nur eine Eigenschaft speichern. Diese Art wird Seperated-Modus genannt.

2.2.2 Transform Feedback

Die Transform Feedback Extension bietet einen neuen Modus, der die Vertexattribute von Primitiven speichert, die von der Pipeline bearbeitet wur-

den. Die gewählten Attribute werden in ein oder mehr Vertex Buffer Objekte geschrieben. Dies geschieht entweder indem ein Buffer für jedes Attribut verwendet wird oder je Vertex alle Attribute nacheinander in einen Buffer geschrieben werden. Wenn ein Shader aktiv ist, das kann auch ein Geometry Shader sein, werden die Primitive aufgezeichnet die der Shader ausgibt. Andernfalls, werden die Primitive des Fixed-Function Vertex Processing aufgezeichnet. Der Fragmentshader bleibt außen vor bzw. die Rasterisierung kann sogar ganz deaktiviert werden für die Dauer des Feedbacks. So könnte es auch als verkürzter Renderpass bezeichnen werden, wie in Abbildung 5 zu sehen. Um auf den Daten in den Vertexbuffern richtig arbeiten zu können, müssen sowohl die Varying Variables in allen verwendeten Shadern des Renderpasses in der selben Reihenfolge ein und ausgegeben werden, wie auch die Reihenfolge im Buffer zugewiesen wurde. Das gleiche gilt analog für die Verwendung von mehreren Buffern. Durch die Verwendung der Buffer Objekte können die Daten direkt auf der GPU von den Shadern verarbeitet werden, ohne zunächst zur CPU transferiert zu werden. Außerdem können auch andere Buffer Objekte benutzt werden wie z.B. Pixelbuffer, Parameterbuffer oder jedes andere Buffer Objekt. Die Buffer Objekte können zwar theoretisch zwischen den Feedback Schritten verändert werden, um z.B. Eckpunkte die im Shader nicht mehr bearbeitet werden müssen für den nächsten Durchgang zu sparen. Jedoch würde dadurch der Geschwindigkeitsvorteil des Transform Feedbacks verloren gehen, da dies mit Hilfe der CPU geschiet und die Daten erst von der Grafikkarte dorthin transferiert werden müssen.
Neu bei der Extension ist auch das Queryobject, welches einem erlaubt den Feedbackmodus auch asynchron zu nutzen oder die Anzahl der verarbeiteten Primitive zu zählen, um z.B. Abbruchkriterien zu finden oder sie als Kontrolle für korrektes arbeiten zu benutzen. In dieser Arbeit wurde ein so genanntes Interleaved Array verwendet in dem alle Vertexattribute gespeichert wurden und ein Query als Kontrollelement, welches die gezeichneten Primitive zählt.

Um mit dem Transform Feedback zu arbeiten muss zunächst die zu bearbeitende Geometrie in einen VBO übergeben werden. Dies kann mit den Regulären Zeichenbefehlen von OpenGL geschehen die wahlweise schon mit Shadern in das VBO gerendert werden. Andererseits kann auch Initial ein VBO mit Geometrie erstellt werden welches direkt verwendet werden kann. Mit dem gefüllten Buffer kann eine Feedbackoperation mehrfach in einer Schleife oder verschiedene Operationen nacheinander ausgeführt werden, während die Rasterisierung deaktiviert ist, bevor der Renderpass beendet wird. Um die durchgeführten Operationen schließlich darzustellen, reicht es mit der Extension einen weiteren Zeichenbefehl durchzuführen mit eingeschalteter Rasterisierung. Dieser Schritt kann selbstverständlich auch wieder andere Shader benutzen, muss allerdings seine Ausgabe in der Darstellungsroutine der Pipeline haben. Das heißt, wenn nur ein

Geometry Shader benötigt wird, verlangt das Transform Feedback dennoch einen Vertex Shader und Fragment Shader. Mit diesem Paket könnte im Anschluss gleich die Rasterisierung durchgeführt werden.[NVI08]

2.3 Bump Mapping

Bump Mapping ist ein Überbegriff für verschiedene Verfahren, die die Beleuchtung verbessern. Das bereits erwähnte Normal Mapping ist eines davon. Dabei werden in einer Textur Normalen abgelegt, die im Fragmentshader eine pro Pixel Beleuchtung ermöglichen, welche über die Interpolation der Eckpunktnormalen hinaus geht. So kann die durch das DPM verschobene und / oder genauer unterteilte Geometrie korrekt beleuchtet werden, ohne dass die Normalen jedes mal zur Laufzeit von der Heightmap abgeleitet werden müssen. Aber auch ohne DPM verbessert es die Qualität der Beleuchtung. Daher ist es das erste und am weitesten verbreitet Verfahren, um bessere Grafikqualität zu erhalten ohne die Geometriekomplexität zu erhöhen. Es sieht korrekt schattiert aus, solange dabei die selben Regeln wie für die Heightmap berücksichtigt werden. Das heißt, da es nur einen Höhenwert pro Pixel gibt, sind keine senkrechten Flächen möglich und ebenso gibt es auch nur eine Normale pro Pixel für die Beleuchtung.Beim Normal Mapping gibt es ebenfalls weitere Unterscheidungen. Zum einen wäre hier das Normal Mapping im Tangentenraum zu nennen. Dabei werden die Normalen in der Textur so interpretiert, dass sie nur von dem Dreieck in Richtung der Flächennormale zeigen. Die drei Komponenten der Normale werden in den Farbkanälen der Textur gespeichert. Rot bestimmt die Intensität entlang der Tangente zum Dreieck, blau bestimmt die Intensität der Flächennormale des Dreiecks und grün bestimmt die Intensität der Binormale, welche aus dem Kreuzproduckt der Flächennormale und der Tangente berechnet wird. Da die meisten Normalen hauptsächlich von dem Dreieck weg zeigen statt in eine Richtung geneigt zu sein, ist der Blaustich der Normalmap charakteristisch für die Verwendung des Tangentenraumes. Der Vorteil an diesem Verfahren ist, dass ein und die selbe Textur unabhängig von der Ausrichtung des Dreiecks und mehrfach in der Geometrie verwendet werden kann für gleiche oder symmetrische Teile. Neben dem Tangentenraum gibt es noch den Objektraum. Dabei wird die Normalmap sozusagen um ein Objekt gewickelt und enthält alle Oberflächennormalen des Objektes und nicht nur die für ein Dreieck. Damit ist es nicht so leicht wieder verwendbar, aber leichter in der Implementierung. Denn die gespeicherten Normalen können ohne weitere Verarbeitung zur Beleuchtungsberechnung verwendet werden. Diese Art der Normalmap ähnelt in ihrem Aussehen eher eine Farbpalette, in der die verschiedenen Grund- und Mischfarben an unterschiedlichen Stellen unterschiedlich stark hervortreten.

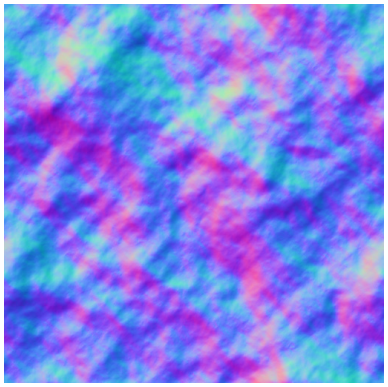

Abbildung 7: Texturbeispiel Tangentenraum

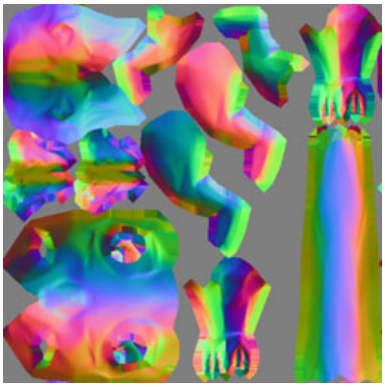

Abbildung 8: Texturbeispiel Objektraum[Kre]

2.4 Level of Detail

Level of Detail(LoD) beschreibt eine abgestufte qualitative und quantitative Veränderung der wahrnehmbaren Eigenschaften eines Datensatzes. Das kann einerseits die Textur sein, dann würde traditionell zu Mip-Mapping gegriffen, welches von OpenGL z.B. automatisch durchgeführt werden kann. Das kann andererseits aber auch die Geometriekomplexität betreffen. Durch den Einsatz von LoD bzw. die Reduzierung der Details mit der Entfernung, kann der Aufwand der Berechnnung reduziert und die Darstellung insgesamt beschleunigt werden. Im Falle der Reduzierung von Geometrie, gibt es unterschiedliche Möglichkeiten. Große Datensätze werden in einzelne Quadrate, wie z.B. im Fall von Terrain, unterteilt, die je nach Entfernung unterschiedlich fein aufgelöst werden. Dabei sind in der Regel an den Übergängen der Quadrate Schürzen (Abb. 9) angebracht, um Lücken zu verdecken, die entstehen, wenn durch die unterschiedliche Auflösung nicht alle Eckpunkte an den Übergängen die gleiche Position haben. Denn wenn ein benachbarter Bereich durch das LoD feiner unterteilt ist, besitzt er zwischen den Punkten der gröberen Stufe weitere Punkte, die nicht zwangsläufig auf der Verbindungslinie der gröberen Punkte liegt. Dann ist das Mesh nicht mehr geschlossen und Sehstrahlen treffen nichts mehr bzw. das Falsche. Dadurch entstehen ungewollte Artefakte die es gilt zu verhindern. Die Einsatz zusätzlicher Dreiecke für einen Mesh nahen Übergang, ist dabei eine bekannte Technik und Alternative zu Schürzen.
Kleinere Datensätze von Einzelobjekten z.B. können damit komplett in unterschiedlichen Auflösungen dargestellt werden und brauchen keine Schürzen. Dennoch können die Übergänge von einer Stufe in die andere, wie auch bei den anderen LoD Bereichen, recht plötzlich passieren. Daher werden die Übergänge in einer Entfernung gewählt in der es nicht mehr auffällt, oder die Geometrie bewegt sich nach Überschreiten der Grenze lang-

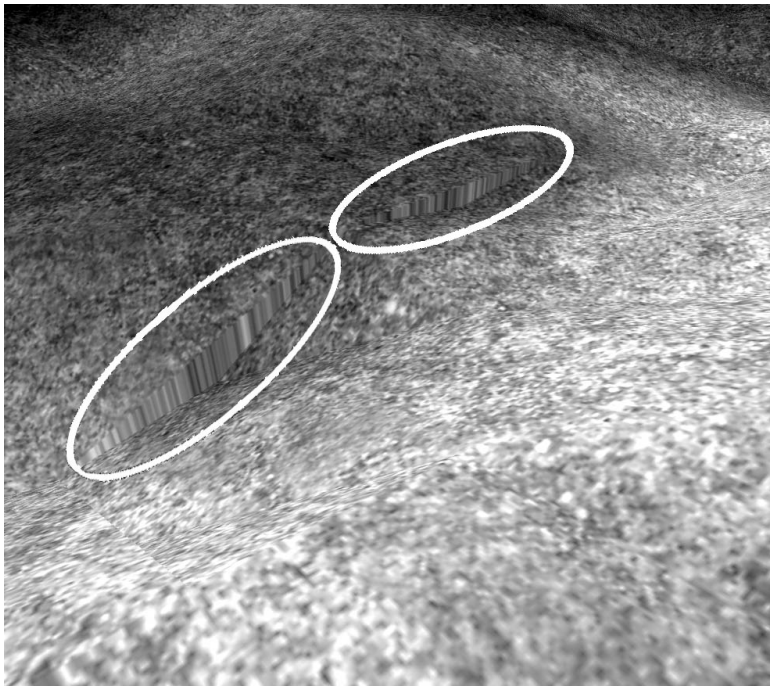

Abbildung 9: Nahaufnahme von Schürzen an der Oberfläche

sam in die entsprechende Position. Geschieht dies unabhängig von der Position des Betrachters kann es so aussehen als ob sich die Geometrie von alleine bewegt. In Verbindung mit der Entfernung allerdings kann ein Interpolationsschritt zwischen den beiden Stufen festgelegt werden, so dass nur mit der Bewegung des Betrachters auch die Geometrie angepasst wird. Wenn die Übergänge bei Texturen verborgen werden müssen, wird meist von einer in die andere Stufe übergeblendet. Da es bei der Entscheidung, wann etwas in seinem Detailgrad reduziert werden kann, hauptsächlich darum geht, wieviele Pixel es auf dem Bildschirm einnimmt, spielt neben der Entfernung auch die Neigung des Primitivs eine wichtige Rolle. So wird oft eine Art von Formfaktor benutzt, um zu bestimmen wieviel Platz z.B. ein Dreieck auf dem Bildschirm einnimmt.

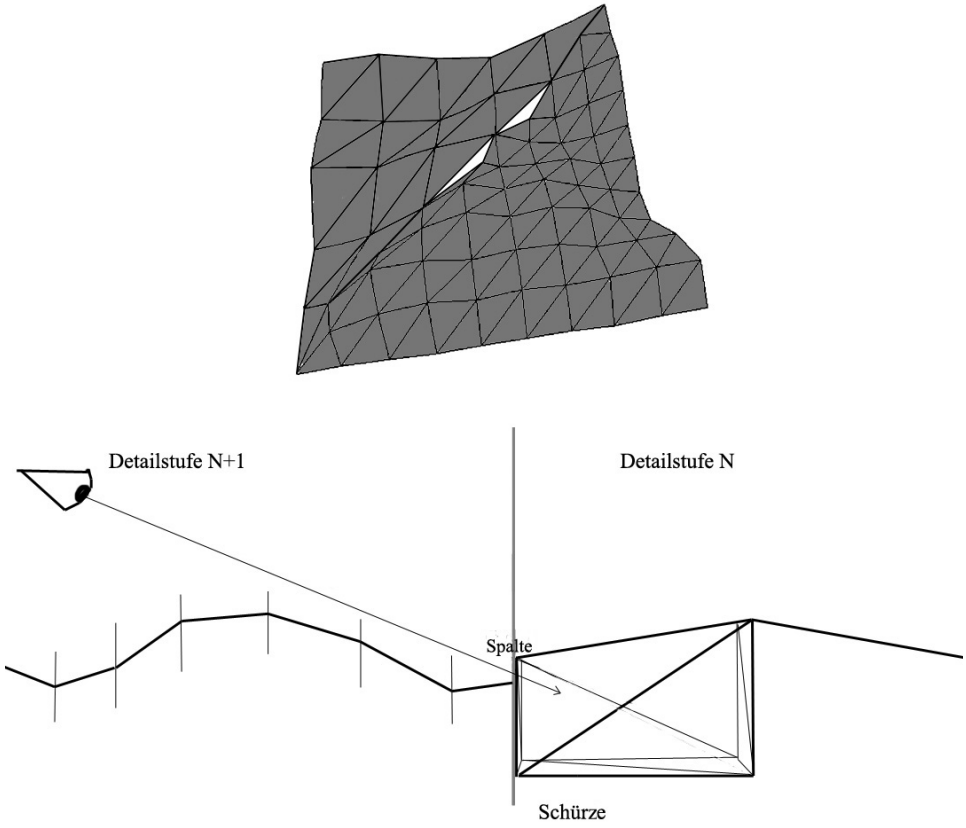

Abbildung 10: Beispiel für Lücken und Schemaskizze mit Schürze

3 Verfahren

In diesem größten Kapitel dieser Arbeit geht es darum, die unterschiedlichen Verfahren genau zu erklären und Vor- und Nachteile aufzuzeigen.
Da Normalen und Höhenwerte in Texturen abgelegt werden sollen, stellt sich vielleicht auch die Frage, ob ganze Floatwerte in der Textur besser zu bearbeiten sind oder Werte zwischen Null und Eins. Aber das kommt auf den Programmierer und die gewünschten Eigenschaften des Programms an. Durch Skalierung lassen sich die Werte ohnehin leichter an die anderen Textureigenschaften oder Geometrie anpassen.

3.1 Einzelshader ohne Transform Feedback

Bei diesem Verfahren wird versucht, alles Nötige für Displacement Mapping in einem einzigen Shader und in einem einzigen Durchgang zu erledigen. Da Subdivision nicht rekursiv in einem Shader stattfinden kann, muss hier das zugrunde liegende Dreieck mit einer zweidimensionalen Schleife rasterisiert werden. Eine Möglichkeit ist die Interpolation zwischen den Eckpunkten für die äußere und innere Schleife. Dabei muss die genaue Ausrichtung der Textur auf dem Dreieck nicht bekannt sein. Die Nachbarschaft des Dreiecks kann auch außer Acht gelassen werden. Angrenzende Dreiecke hätten bei gleicher LoD Stufe automatisch die unterteilten zusätzlichen Eckpunkte an den gleichen Stellen. Nur für unterschiedliche Stufen müssten Schürzen angebracht werden. Diese wären mit minimal erforderlichen Dreiecken realisierbar, da alle Informationen über die Subdivision im Shader verfügbar sind. Das größte Problem dieses Shaders ist der, dass das Punktlimit hier 128 beträgt und somit keine hohen Auflösungen erreicht werden können. Es besteht natürlich die Möglichkeit mehr kleinere Dreiecke mit diesem Shader zu unterteilen, aber einerseits kämen bei einer Implementierung noch Eckpunkte für Schürzen hinzu, um Lücken bei LoD Übergängen zu verhindern, anderseits müsste die Ursprungsgeometrie schon feiner unterteilt sein, was mit dieser Arbeit umgangen werden soll. Sollte dieser Ansatz doch weiter verfolgt werden, wäre LoD über die Variable steps als Uniformparameter, integrierbar. Eine Schleifenvariante mit festem Koordinatensystem, um auf jedem Texturpixel ein Punkt zu erzeugen, wäre eine andere Möglichkeit. Um dies mit gleicher Adaptivität wie mit Interpolation zu realisieren, muss im Schader genau bestimmt werden, welcher der Eckpunkte welche Welt- bzw. Texturkoordinaten hat. Außerdem muss bestimmt werden, welche davon die größte und kleinste ist und wie innerhalb der Textur, je nachdem ob es über s oder t Reihen geht, das Dreieck für das Displacement Mapping rasterisiert wird. Das führt zu einem erheblichen Mehraufwand bei der Berechnung dieser Werte, da dies nicht über min() und max() Funktionsaufrufe zu realisieren ist, sondern nur mit If-Anweisungen. Denn je nachdem wie die Textur auf dem Dreieck

liegt, müssen die Schleifen anders ausgerichtet werden. Durch die obige Interpolation spielt die Ausrichtung der Textur keine Rolle.

3.2 Einzelshader nur mit Punkten

Der Shader ist hauptsächlich eine Vereinfachung des vorangegangenen. Der Punktshader versucht eine Oberfläche ganz ohne Linien oder Flächen

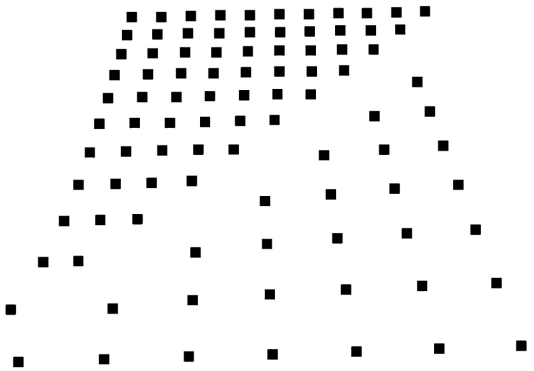

Abbildung 11: Beispiel für Punktausgabe

nur mit Hilfe von Punkten zu erzeugen. Deshalb muss kein Meshing Algorithmus verwendet werden und der Code wird einfacher, kürzer und damit auch schneller. Wie in Abb 11 zu sehen, muss ab einer gewissen Nähe zu einer Oberfläche aus Punkten darauf geachtet werden, das diese je näher sie kommt zunächst Lücken bekommt und weniger wie eine Oberfläche aussieht. Die Lücken können bis zu einem bestimmten Grad über größere Punkte geschlossen werden, hauptsächlich aber über die Punktanzahl. Dabei muss aber auf das Erzeugungsmaximum des Geometry Shaders geachtet werden. Dennoch bleibt das Problem, dass Die Punkte eine feste Größe auf dem Bildschirm einnehmen und unabhängig von der Entfernung der Kamera gleich bleiben. Das heißt, sie werden quasi größer im Raum wenn sie sich von der Kamera entfernen. Dadurch gibt es bei dreidimensionalen Objekten praktisch keine perfekte Entfernung zur Kamera oder Ausrichtung im Raum. Auf einer Ebene passen die Kanten der Punkte nahtlos aneinander, davor jedoch entstehen Lücken und dahinter schieben sie sich übereinander und reduzieren schnell die erkennbaren Details. Die Kompensation dieses Problems durch noch so geschickt gewählte LoD Grenzen bringt auch keine zufrieden stellende Besserung. Die Punktgröße selbst kann auch nicht innerhalb eines Shaders gewechselt werden, so dass nur der Wechsel zu echten Dreiecken bleibt, die automatisch in Abhängigkeit zur Entfernung von der Kamera unterschiedlich groß auf dem

Bildschirm erscheinen. Dort gibt es wiederum mehrere Möglichkeiten geeignete Geometrien zu wählen um einen Punkt zu ersetzen, wie z.B. eine Kugel, ein Quadraht, einen Zylinder oder Billboards. Grundsätzlich wird der Algorithmus durch das Einsetzen von mehr Geometrie für einen einzelnen Punkt langsamer. Wobei eine Kugel und ein Zylinder zwar noch die besten Beleuchtungsergebnisse erzielen, aber gleichzeitig auch die meisten Eckpunkte brauchen. Ein Quadrat passt perfekt in ein virtuelles Voxelraster, um Lücken zu vermeiden um mit den richtigen Normalen nur geringe Beleuchtungs- und Überlappungsartefakte erzeugen. Billboards zu guter letzt erfordern am wenigsten neue Geometrie, erzeugen aber auch die größten Artefakte. Es würden Überlappungen auftreten bei einem schrägen Betrachtungswinkel auf eine Ebene, da sich die Billboards zum Betrachter drehen und dabei gleich groß bleiben. Angefangen mit Treppcheneffekten bis hin zur vollständigen Verdeckung einer ganzen Ebene durch die vorderste Reihe an Billboards. Die richtige Wahl der Methode hängt dabei immer vom spezifischen Verwendungszweck und Programmstruktur des Zielsystems ab.

3.3 Zwei Shader mit Transform Feedback

Um das Problem zu umgehen, dass der Geometry Shader mehrere Outputmaxima hat, kann für die Unterteilung der Geometrie das Transform Feedback verwendet werden. Danach erst werden die Eckpunkte entsprechend der Displacementmap verschoben. Das Transform Feedback wiederholt dazu die Subdivision gleich der Stufe des LoD. Diese ist im Kern sehr einfach aufgebaut. Es reicht zunächst ein Dreieck als Eingabe ohne Nachbarschaft. Zwischen den Eckpunkten werden auf halber Strecke neue Punkte erzeugt, so dass insgesamt sechs zur Verfügung stehen. Mit diesen werden aus dem ursprünglichen Dreieck vier kleinere neue Dreiecke erstellt. Diese Dreiecke können einzeln erzeugt werden, so dass insgesamt zwölf Eckpunkte ausgegeben werden. Die Anzahl der ausgegebenen Eckpunkte spielt zwar für das VBO keine größere Bedeutung als die Dreiecke, aber durch die Ausgabe als Trianglestrip können Anweisungen gespart werden. Dadurch wird die Darstellung beschleunigt und er Code kürzer gehalten. Mit dem Trianglestrip werden nur noch acht Anstelle von zwölf Eckpunkten benötigt. Während dem Vorgang müssen außerdem noch die Normalen der Eckpunkte interpoliert werden. Durch die spätere Verschiebung entlang der Interpolierten Eckpunktnormalen anstelle der Flächennormale werden Lücken an Polygonkanten geschlossen und die entstehende Verzerrung der Displacementmap auf das gesamte Dreieck verteilt. Dieses Vorgehen reicht jedoch nicht aus um die ursprüngliche Form der Geometrie zuverlässig zu kaschieren.
Bei jedem Durchgang des Transform Feedbacks werden die Dreiecke wieder in vier kleinere unterteilt, so dass sich der Aufwand mit jedem Durch-

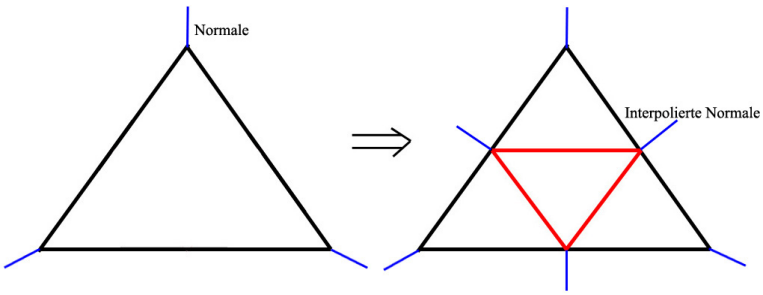

Abbildung 12: Skizze zur Interpolation und Subdivision

gang vervierfacht. Ist die Unterteilung abgeschlossen, werden die Geometriedaten an den Displacement Shader weitergegeben. Dieser bestimmt zunächst den Tangentenraum bevor er die Normalen an der Normalmap ausrichten kann. Außerdem müssen vorher die Eckpunkte verschoben werden, da sonst die Normalen schon die Werte für die Beleuchtung hätten, anstatt für die Verschiebung. Wenn das beachtet wird, kann für die Normalen die selbe Varying Variable verwendet werden. Die Unterteilung in zwei Shader ist notwendig. Wird die Verschiebung auch noch im ersten Shader durchgeführt, muß jedes Mal geprüft werden, ob die letzte Iteration des Transform Feedbacks erreicht wurde. Denn würde die Verschiebung nicht erst im letzten Schritt durchgeführt, sondern bei jedem, würden die Punkte aufgrund der Art der Subdivision unterschiedlich oft Verschoben. Das würde auf eine ungleichmäßige und Falsche Verschiebung hinauslaufen, wodurch automatisch auch zahlreiche unnötige Anweisungen entstehen, die die Performanz senken. Jedes Dreieck bräuchte eine Variable, die speichert ob es Final ist oder nicht. Diese muss bei jedem Transform Feedback Durchgang für jedes Dreieck geprüft werden. Aber der weit größte Anteil entsteht, wenn für jeden Transform Feedback Durchgang die Rasterisierung aktiviert bleibt. Sie kann zwar wieder auf den letzten Durchgang beschränkt werden, aber wenn schon der Teil der Rasterisierung vom eigentlichen Feedback getrennt werden muss, ist die Lösung mit zwei unterschiedlichen Shadern besser. Der Zweite Shader garantiert, dass die komplette Geometrie nur ein einziges Mal verschoben und gerendert wird, die beiden Shader einzeln betrachtet übersichtlicher und einfacher und somit auch effizienter sind.

3.4 Volumen

Echtes Displacement Mapping war bisher nicht praktikabel oder zu langsam, aber wenn die Ressourcen schon bereitgestellt werden, liegt es nahe zusätzlich noch nach den neuen Grenzen zu suchen. Hier geht es um ei-

ne Idee das Displacement Mapping um eine Dimension zu erweitern und somit dreidimensionale Objekte darzustellen und nicht nur eine zweidimensionale Heightmap. So könnten zum einen senkrechte Flächen erzeugt werden, z.B. für Häuserwände oder viel Komplexere Strukturen. Da wären zunächst einfache Überhänge oder Schwebende Objekte, aber auch Höhlen. Alle Weltkoordinaten stammen dabei aus nur einer Textur. Die Erwei-

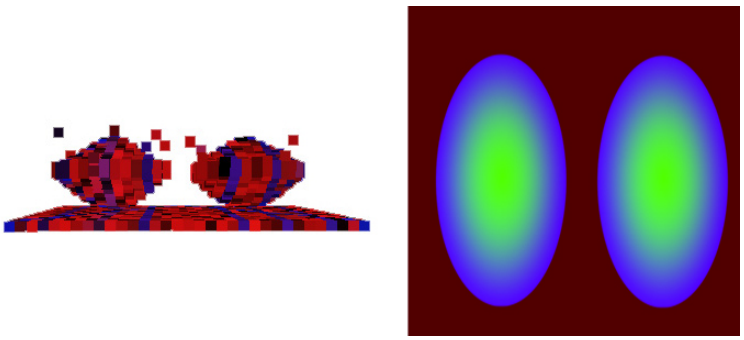

Abbildung 13: Volumen Displacement Mapping mit Punkten(links), zugehörige Textur(rechts)

terung um eine Dimension ist natürlich nicht einfach und mit höherem Rechenaufwand und Speicherplatz behaftet. Daher war die erste Idee, möglichst viele der Datenstrukturen zu simulieren ohne dabei auf echte Geometrie zu verzichten. Dazu sollten mehrere Punkte übereinander in einem Texturpixel einer zweidimensionalen Textur gespeichert werden. Wie diese Punktwolke gespeichert oder ein Mesh darüber gelegt wird, gestaltet sich dabei allerdings schwierig.

3.4.1 Datenstrukturen

Die erste Frage ist, wie die Dreidimensionalen Daten am besten gespeichert werden. Datenstrukturen wie das regular grid, rectangular grid und structured grid werden hauptsächlich in Bereichen angewendet, bei denen durch äußere Umstände keine in allen drei Raumachsen regelmäßige Anordnung möglich ist. Magnet Resonanz Scan z.B. müssen die Scheibendicke zusätzlich speichern. Dabei würde dann das regular grid Anwendung finden. Bei manuell oder automatisch erzeugten Testdaten können solche Unregelmäßigkeiten verhindert werden. Eine andere Methode die Daten zu speichern, wären Radiale Basisfunktionen. Diese müssen aus bestehnden 3D Punkten vorberechnet werden und können dann aber auch mehr Punkte als vorher für die Erzeugung benötigt wurden erzeugen und benötigen außerdem zur Ablage im Speicher keine Gitterstruktur. Durch Interpolation können auch so schon zusätzliche Punkte zwischen den Daten erzeugt werden, doch die RBF sind ähnlich wie NURBS besser an die Tatsächli-

che Oberfläche angenähert als Interpolation. Nur wird das Umwandeln zurück in Punkte extra Zeit zur Laufzeit kosten, da sie danach doch erst mit Texturen in die Shader gebracht werden, während die Punkte in der Textur direkt gespeichert werden können. Der Vorteil ist also die verbesserte Genauigkeit der erhaltenen Punkte für verschiedene Detailstufen. Der Nachteil ist der erhöhte Rechenaufwand für die Umwandlung der RBF in für Shader auswertbare Punkte. Das einfache Cartesian Grid benötigt zwar viel Speicherplatz, ist aber die einfachste und gleichzeitig gleichmäßigste Repräsentation. Dreidimensionale Texturen sind quasi ein Cartesian Grid und werden auch ohne Erweiterung von der Grafikkarte unterstützt. Diese benötigen allerdings auch enorm viel Speicherplatz. Um das Speicherplatzproblem zu umgehen, könnte eine einfache zweidimensionale Textur verwendet werden. So besteht immerhin die Möglichkeit, vier Punkte übereinander zu speichern. Es liegen sozusagen vier Höhenkarten in einer Textur. Jedoch sollen dabei diese vier Punkte nicht auf vier übereinander liegenden Schichten beschränkt bleiben. Die vier Punkte können in einem theoretischen Koordinatenraum beliebig gesetzt werden. Es dürfen nur maximal Vier Punkte übereinander liegen. Eine andere Alternative wäre es die Punkte für die Beschreibung von zwei Intervallen zu benutzen innerhalb dieser dann der Raum als voll bzw. massiv betrachtet wird. Dazu wird für das Meshing am besten ein Marching Cubes Algorithmus verwendet. Doch bevor das Meshing behandelt wird, zunächst noch einige andere Punkte bezüglich der Datenstrukturen. In Analogie zu den übereinander liegenden Höhenkarten kann natürlich diese Anordnung verwendet werden, um z.B. runde Objekte zu erzeugen. Der Vorteil dabei ist, dass eine insgesamt höhere vertikale Auflösung vorhanden ist. Das liegt daran, dass mit jedem Farbkanal ein Float zur Verfügung steht und die vertikale Auflösung somit mit vier Float dargestellt werden kann. Diese Art wurde für das Beispiel in Abbildung 13 verwendet. Der Nachteil ist, dass Wendepunkte, an denen z.B. die Kugeloberfläche eine senkrechte Tangente hat, nur an den Schnittpunkten der Kanalschichten liegen können. Dies kann auf Kosten der vertikalen Auflösung umgangen werden, indem die Kanalschichten ineinander liegen. So bleibt nur eine vertikale Auflösung von einem Float und theoretisch können durch Numerische Ungenauigkeiten oder Interpolation dich beieinander liegende Punkte ineinander Abgebildet werden. Aber wie gesagt, die Lage der Wendepunkte ist frei wählbar. Derartige Texturen im allgemeinen sind nur schwer wie herkömmliche zweidimensionale Texturen in einem Bildbearbeitungsprogramm zu bearbeiten oder zu erstellen. Die automatische Generierung aus dreidimensionalen Modellen von Editoren wie etwa Maya oder 3D Max wäre naheliegend. Zu guter letzt die unstructured grid genannte Struktur, welche ganz unterschiedliche Zelltypen Speichern kann wie z.B. kleine, große, schiefe, etc. Diese Struktur für ein regelmäßiges Gitter zu verwenden, wäre reine Verschwendung. Für die zusätzlichen Eigenschaften der Unterschiedlichen Zellen wird natür-

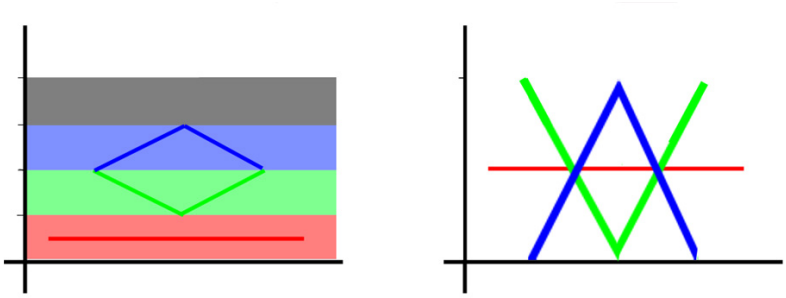

Abbildung 14: Unterschied der Textur aus Abb. 13 in Schichten(links) und ineinander(rechts)

lich grundsätzlich erst einmal mehr Speicherplatz verbraucht. Es erscheint aber in Zusammenhang mit detailabhängigen Meshes viel versprechend, da damit in einer ebenen Fläche wie z.B. einer Tischplatte keine Überflüssigen Punkte gespeichert werden müssten.

3.4.2 Meshing

Dreidimensionales echtzeitfähiges Meshing im Shader ist ein großes Problem. Einerseits schon ohne die Zusatzeinschränkung Shader, andererseits weil meist nur Einzelobjekte behandelt werden und nicht mehrere getrennte. Eine Möglichkeit mehrere getrennte Objekte gleichzeitig zu bearbeiten ist der Marching Cubes Algorithmus [LC87]. Dieser arbeitet auf einem dreidimensionalen Voxelgitter bei dem für jedes Voxel gespeichert ist, ob es in einem Objekt liegt oder nicht. Unter Berücksichtigung der Nachbarvoxel kann entschieden werden ob und wie ein Polygon in das Voxel gelegt werden muss um die Oberfläche am besten darzustellen. Der Algorithmus arbeitet dabei lokal ohne Wissen über die gesamte Geometrie. Für ein Voxel bestehen 256 Möglichkeiten wie ein Polygon gelagert sein kann. Diese Vielfalt kann nur eingeschränkt vereinfacht werden, durch eine Lookuptabelle jedoch können Berechnungen gespart werden. Da es hier darum geht die Geometrie aus einer Fläche zu erzeugen, die zunächst keine dreidimensionalen Daten liefert, kann der Marching Cubes Algorithmus nicht ohne weiteres eingesetzt werden. Es wäre z.B. möglich einen modifizierten Marching Cubes Algorithmus auf simulierten Voxeln anzuwenden. Dazu werden vertikale Mindestabstände zwischen den Punkten in der Textur gewählt, um somit die Voxelgröße bestimmen zu können. Die horizontalen Nachbarn finden sich dabei in den Nachbartexeln, wobei diese ebenso wie die vertikalen Nachbarn, mit dem vertikalen Mindestabstand bestimmt werden müssen. Der Algorithmus kann dabei sowohl auf der GPU als auch auf der CPU laufen. Da jedoch die Anzahl an vertikal übereinanderliegen-

den Voxeln beschränkt ist, entstehen bei größeren Objekten Hohlräume, die durch ihre Innenseite zusätliche Dreiecke produzieren. Dabei wird es schwierig, ungewollt hohl von gewollt hohl zu unterscheiden. Andererseits ist bei dem Algorithmus eine Interpolation zwischen den Texeln zu vermeiden, da sonst die Punkte nicht mehr im simulierten Gitter liegen und nicht mehr leicht zu unterscheiden ist ob ein Voxel in oder außerhalb eines Objektes liegt. Bei einem groben Mesh und feiner Textur, können Punkte übersprungen werden und unkontrolliert Lücken im Objekt auftreten. Die Adaptivität ist also schwierig mit Interpolation zu vereinbaren. Der größte und wichtigste Vorteil ist der, dass es ein bewährtes Verfahren zur dreidimensionalen Mesherzeugung ist, gut dokumentiert und vor allem auch Verschiedene Objekte innerhalb einer Punktwolke voneinander getrennt halten kann. Im Gegensatz zur Delauney Triangulierung. Hinsichtlich der Umsetzung innerhalb eines einzelnen Geometry Shaders wie in Kapitel 3.1 oder 3.2, bei denen es darum geht alles in einem Durchgang darzustellen, ist der Marching Cubes Algorithmus zu stark eingeschränkt. Zu der zusätzlichen Unterteilung des Grunddreiecks kommt, dass bei der Meshgenerierung wieder neue Dreiecke erzeugt werden, wodurch das Outputmaximum schneller erreicht wird. Eine Hochrechnung der verlorenen Details, schränkt die Einsatzgebiete stark ein. Und auch wenn das Anweisungsmaximum im Geometry Shader mit Shadermodell 4.0 bei 65536 liegt, könnte Marching Cubes dort ebenfalls an Grenzen stoßen.
Ein anderer Ansatz ist die dreidimensionale Anwendung des Delauney Algorithmus. Dabei werden wie auch bei der zweidimensionale Variante, rekursiv zu einer Kante der nahste Punkt mit geringstem Winkel gesucht, um ein neues Dreieck zu bilden. In der dreidimensionalen Variante werden allerdings neue Tetraeder gebildet mit dem nahsten Punkte zu einer Dreiecksfläche. Das Problem dabei ist, das eine Konvexe Hülle um die Punktwolke gelegt wird. Das ist einmal bei konkaven Objekten problematisch und außerdem bei mehreren getrennten Objekten. Eine Möglichkeit Objekte von einander zu trennen ist es die Kantenlänge für neue Tetraeder zu beschränken. In eingeschränktem Maß vermeidet das auch eine konvexe Hülle. Allerdings muss dann bei der Generierung der Textur auf diese Grenze geachtet werden und kleinere Lücken als diese Grenze sind natürlich nicht möglich. Die Frage wie mit dem Punktlimit bzw. dem Outputlimit des Geometry Shaders umgegangen wird, stellt sich auch hier. Es spielt aber eine geringere Rolle als bei den Marching Cubes, denn bei Delauney bleibt ein Punkt ein Punkt und wird nicht zu mehreren neuen. Dennoch ist für komplexere Formen eine Unterteilung notwendig. Prinzipiell ist die Unterteilung kaum mit mehr Aufwand verbunden für Erzeugung und Verbindung der Elemente, aber ein großes Problem ergibt sich aus der Funktionsweise des Algorithmus im Allgemeinen. Der zu beginn erwähnte rekursive Charakter des Verfahrens lässt sich nicht leicht mit Geometry Shader und auch kaum mit Transform Feedback verbinden. Zwar ist eine

eingeschränkte Form der Rekursion möglich, jedoch nur wenn die Rekursionstiefe vorher schon bekannt ist. Im fall einer Implementierung sollte auf Subdivision verzichtet werden, da bereits viele Rekursionen für Delauney gebraucht werden. Dabei bleibt jedoch das Problem, dass die Punkte aus der Textur ausgelesen werden während die erzeugten Dreiecke im VBO gespeichert werden. Mit den begrenzten Mitteln der GPU ist es schwierig bzw. Zeitaufwändig die bereits verwendeten Punkte von den noch zu bearbeitenden zu unterscheiden. Annehmebar ist auch, dass zu viele unnötige Dreiecke im inneren der Punktwolke erzeugt werden, da genau wie bei den Dreiecken alle Lücken geschlossen werden. Aber zu viele Dreiecke entstehen bei guten Texturen und der Beschränkung der Kantenlänge weniger, da die Objekte hohl sein sollten und eine geringe Manteldicke der Objekte viele überflüssige Tetraeder im Inneren verhindert.
Weiterhin gibt es die Möglichkeit durch weitere Einschränkung der Verbindungen zwischen den Punkten das Meshing zu beschleunigen und Punkte nur mit direkt benachbarten Punkten zu verbinden. Dazu kann neben der Nachbarschaft in der Textur bzw. horizontale Nachbarschaft, auch die Nachbarschaft in den Farbkanälen bzw. vertikale Nachbarschaft herangezogen werde. Die erfordert aber nicht das im oberen Abschnitt Datenstrukturen erwähnte Modell der gestapelten Farbkanäle. Unter Berücksichtigung der Kantenlänge kann dann auch zwischen benachbarten Farben eine Verbindung verhindert werden. Bei dem Nachbarschaftsmodell ist nur die Kantenlänge zwischen unterschiedlichen Farben zu überprüfen, nicht jedoch innerhalb eines Kanals, da nur benachbarte Texel verbunden werden. In der horizontalen Ausdehnung ist die Kantenlänge damit schon beschränkt, während vertikal noch größere Abstände möglich sind. Derartige Texturen sind in Bildbearbeitungsprogrammen nicht mehr effektiv zu bearbeiten, da Werte im dreidimensionalen Raum gesetzt werden müssen, während Bildbearbeitungsprogramme für zweidimensionale Bearbeitung optimiert sind. Die möglichen Verbindungen sehen bei den Farbkanälen wie folgt aus. Von rot kann nur eine Verbindung zu einem roten oder einem grünem Nachbarpunkt in den angrenzenden Texeln hergestellt werden, was 17 Möglichkeiten ergibt. In der roten Ebene bzw. Farbkanal nur acht, weil der Ausgangspunkt selbst in der Mitte liegt. In der Ebene darüber bzw. dem grünen Farbkanal neun und darunter liegt kein Farbkanal. Von grün aus, besteht nur die Möglichkeit zu roten, grünen und blauen Punkten eine Verbindung herzustellen und daher müssen in den angrenzenden Texeln 26 Punkte überprüft werden. Bei dem blauen Farbkanal als Ausgangspunkt sieht es ähnlich aus. 26 angrenzende Punkte in den Kanälen grün, blau und alpha. Beim Alphakanal ist es wiederum ähnlich wie schon beim Rotkanal. 17 Angrenzende Punkte in blau und alpha. Natürlich gibt es auch die Möglichkeit, dass kein Punkt in einem Nachbartexel liegt. Da der Texel aber einen Farbwert hat, muss ein bestimmter Farbwert oder Intensität als fehlender Punkt gewertet werden. Kanten sind damit

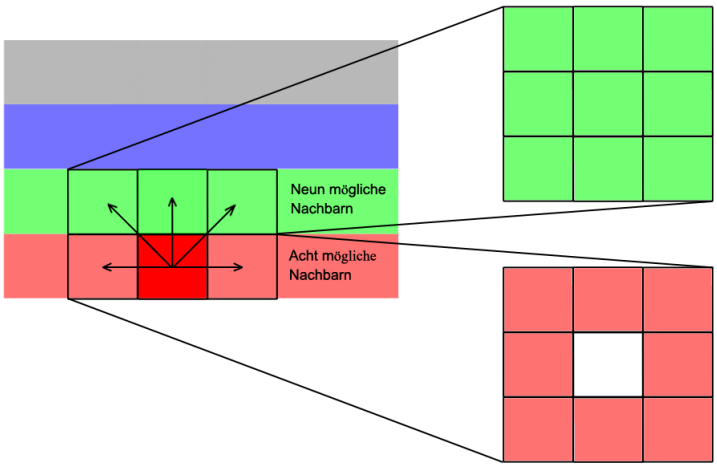

Abbildung 15: Beispiel für Farbnachbarschaft

leicht zu erzeugen, aber Dreiecke bereiten größere Schwierigkeiten. Selbst drei Punkte die untereinander Verbunden sind, müssen nicht zwangsläufig ein Dreieck darstellen. Die Lösung dieses Problems hängt davon ab, wechle Technik für die Dreiecke angewandt wird.
Der naivste Ansatz mit Farbnachbarschaft, ist es zu versuchen eine Schleife mit GL_TRIANGLESTRIP über alle Punkte in der Textur zu gehen und nur die gewünschten Kanten zu erzeugen und unterschiedliche Objekte getrennt zu halten. Dazu müssten viele Zusatzinformationen an den Shader geliefert werden, um z.B. zu wissen welcherlei Art die Objekte in der Textur sind und wie viele es sind. Denn eine Kugel ließe sich noch recht gut mit einem einzelnen Dreieckstreifen darstellen während eine Ebene in jeder Reihe neu angesetzt werden sollte. Auftretende Fehler könnten durch Höhenunterschiede verursacht werden, so dass ein Dreieck quer über die gesamte Fläche gezogen wird. Neue Objekte brauchen ohnehin mindestens einen eigenen Triangletrip da sonst Verbindungsdreiecke vom vorherigen Objekt oder zum nächsten Objekt unvermeidbar wären. Diese Variante der Farbnachbarschaft wäre auch ohne Beschränkung auf Nachbartexel möglich. Dann muss allerdings wieder zusätzlich die horizontale Ausdehnung überprüft werden. Eine andere Möglichkeit ist, die Schleife ohne Dreieckstreifen der Reihe nach über jedes Texel laufen zu lassen. Dabei müssen die Objektanzahl und Art der Objekte nicht gespeichert werden, sondern welche Punkte schon behandelt wurden. Denn von einem Punkt aus können mehrere Dreiecke entstehen, so dass ein Dreieck nur zwischen den Punkten sicher erstellt werden kann, die schon behandelt wurden. Dafür ist die Schleife allerdings besonders einfach. Außerdem bleibt es dem Ersteller der Textur überlassen keine Dreiecke bzw. Punkte zu erzeugen die vom Algo-

rithmus falsch interpretiert werden könnten und es werden keine Punkte durch den Trianglestrip gespart.

Natürlich sind all diese Verfahren um weitere vier Punkte pro zusätzlicher Textur erweiterbar, zum Preis von höherem Rechen- und Verwaltungsaufwand, wenn Texturunits frei sind. Aufgrund der vielen anderen Aufgaben die Texturen zu erfüllen haben erscheint dies jedoch unrentabel. Je nach Implementierung werden für jeden der vier Punkte bzw. für jeden Kanal eine Normalmap benötigt zusätzlich zur jeweiligen Colormap. Also bis zu acht weiterer Texturen.

3.4.3 Marching Cubes auf der GPU mit Einzelpunktinput

Die Einzelpunkte werden dabei zunächst im Vertex Shader durch eine Dichtefunktion in die für den Marching Cubes erforderliche Form gebracht und in einer dreidimensionalen Textur gespeichert. Danach wird jedes Voxel innerhalb der Textur einzeln überprüft und anhand von Lookuptabellen entschiedn ob und wie Dreiecke generiert werden. In einem Vertex Buffer gespeichert, müssen diese Dreiecke nicht neuberechnet werden solange sie sichtbar sind, sondern lediglich neu gerendert. [Gei07]

3.5 LoD

Die Umsetzung eines LoD Verfahrens ist unerlässlich angesichts des Rechenaufwandes und anderer Probleme. Für die Umsetzung gab es zwei grundlegende Ideen. Erstens sollte der Grad der Geometrieunterteilung selbst als LoD dienen. Dazu sollte die Schleifenvariable des Transform Feedbacks, welche die Anzahl der Feedbackdurchgänge für alle Dreieck gleichzeitig steuert, entsprechend der Entfernung zur Kamera manipuliert werden. Also würde sich die Anzahl der Dreiecke innerhalb eines Grunddreiecks bei jedem Schritt vervierfachen und LoD Grenzen nur an Rändern der Grunddreiecke bemerkbar machen. Dies ist alles außerhalb der Shader realisierbar. Das erste Problem dabei ist, das dann allerdings um jedem Dreieck den angemessenen LoD zu geben, sämtliche Dreiecke nacheinander in einem eigenen Transform Feedback bearbeitet werden müssen und nicht zusammen. Denn die unterschiedliche Anzahl an Wiederholungen des Feedbacks, machen es unmöglich alle Grunddreiecke in einem zu bearbeiten. Außerdem sind die Grunddreiecke im Verhältnis zur dargestellten Geometrie so groß, das bei diagonaler Schnittkante zweier Dreiecke in einem Quadrat, das vordere Dreieck hinten zwar genauso weit entfernt ist wie das hintere, aber dennoch feiner unterteilt ist. Das kann zu einer gezackten LoD Grenze führen, die bei Bewegung über die Diagonale Achse des Quadrats auch sprunghaft die Seite wechseln kann. Das zweite Problem sind die Schürzen, aber dazu mehr im Unterabschnitt Spalten und Schürzen. Die zweite Idee zur Realisierung sieht vor, das LoD komplett

im Geometry Shader unterzubringen. Zusammen mit den Subdivisionsanweisungen kann somit das LoD sogar innerhalb des Grunddreiecks variiert werden. Dafür sind allerdings If-Anweisungen nötig, um alternative Anweisungsstränge auszuführen je nachdem ob sich das aktuelle Dreieck vor oder hinter der Grenze des LoD befindet. Um die Grenze für das LoD berechnen zu können, muss die Kameraposition mit Hilfe eines Uniformparameters an den Shader übergeben werden. Die Kameraposition allein reicht allerdings nicht aus um die Entfernung zu berechnen. Dazu ist aber entgegen möglicher Erwartungen aus anderen Bereichen wie z.B. der Radiosity, die Drehung des Dreiecks im Raum von keinerlei Bedeutung. Während bei den meisten anderen Verfahren, flache Betrachtungswinkel dazu führen, dass ein eigentlich großes Dreieck nur noch wenige Pixel auf dem Bildschirm einnimmt, liegt der Fall beim echten Displacement Mapping anders. Durch die Unterteilung und Verschiebung der Geometrie, kann ein eigentlich flacher Betrachtungswinkel doch sehr viele Pixel auf dem Bildschirm einnehmen. Daher spielt es für das LoD keine Rolle ob ein Dreieck senkrecht oder waagerecht zur Kamera steht. Die Entfernung allein ist entscheidend. Doch hier darf nicht einfach ein beliebiger Eckpunkt gewählt werden. Dadurch könnte schnell ein Dreieck, welches eigentlich weiter entfernt ist in der Berechung weiter vorne behandelt werden. Der Mittelpunkt des Dreiecks muss errechnet werden, um für jede mögliche Lage des Dreiecks etwa die gleiche Entfernung zu erhalten. Die Entfernung allein jedoch reicht noch nicht aus, um das LoD in geeignetem Maße zu bestimmen. Kleine Dreiecke werden an den Entfernungsgrenzen genauso oft wie große Dreiecke unterteilt. Dadurch entstünde der Eindruck unterschiedlicher LoD in gleicher Entfernung oder sogar benachbarter Dreiecke. Um dies zu verhindern muss die Größe der Dreiecke mit in die Kalkulation des LoD einfließen. Eine einzelne Kantenlänge oder auch der Gesamtumfang eines Dreiecks ist hierzu unzureichend. Kantenlängen und Umfang variieren besonders bei flachen, aber auch bei relativ gleichmäßigen Dreiecken im vergleich zur Fläche. Daher ist es erforderlich eine einfache Lösung für die Berechnung der Fläche zu erhalten. Die gängige Formel($Breite * Hoehe/2$)ist dabei keineswegs die leichteste Lösung. Die Höhe steht in rechtem Winkel zu einer beliebigen Kante und müsste erst kompliziert errechnet werden. Relativ unkompliziert dagegen ist der Satz des Heron. Mit dessen Hilfe kann die Fläche aufgrund der Positionen der Eckpunkte berechnet werden. Genau die Information die der Shader auch in dieser Form besitzt.

$Flaeche = (S(S-a)(S-b)(S-c))^{1/2}$ wobei $S = (a+b+c)/2$

Die so erechnete Fläche kann zusammen mit der Entfernung in der Formel für das Lod kombiniert werden. Diese Formel kann wie schon erwähnt im Shader für Subdivision verwendet werden. Damit wird nach jedem Un-

terteilungsschritt im Shader neu entschieden ob das Dreieck weiter unterteilt wird oder nicht. So ist das LoD auch innerhalb der Grunddreiecke adaptiv. Auf dem Bild 16 ist die Grenze, zur Verdeutlichung von oben und im

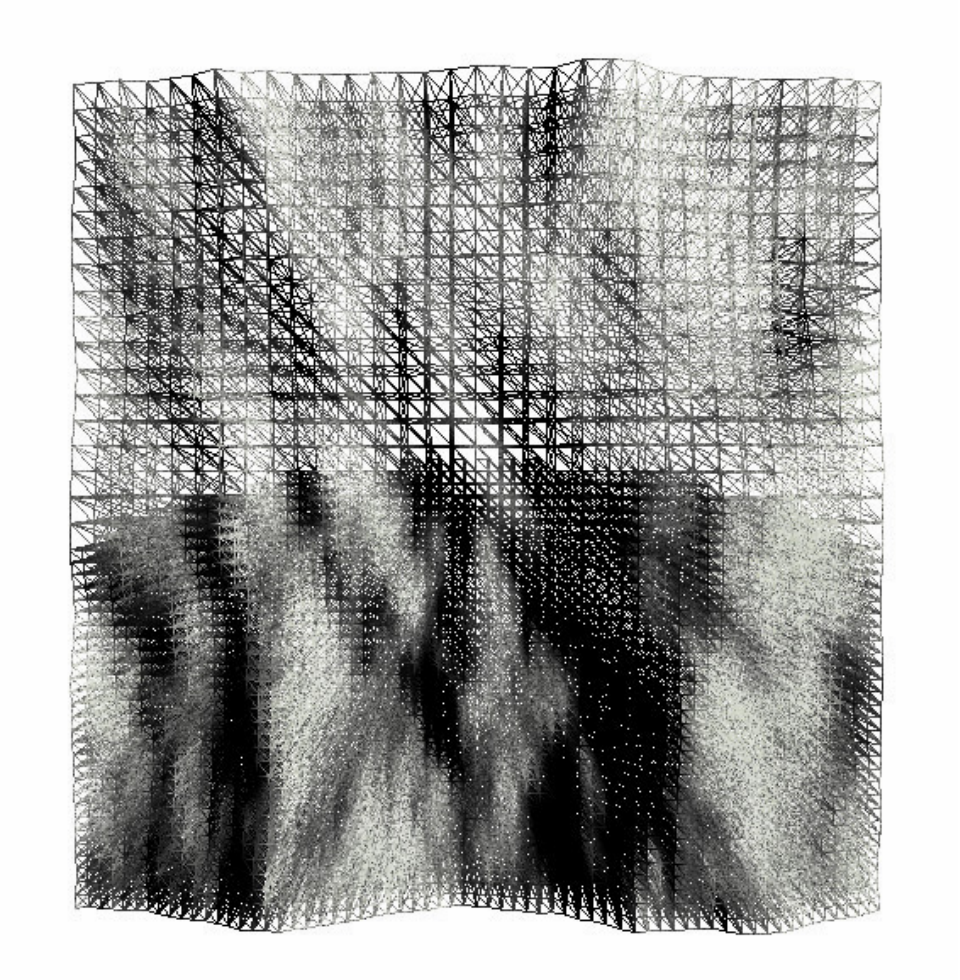

Abbildung 16: Die Bruteforce Version der Schürzen für Level of Detail

Drahtgittermodus zu sehen. Oder mit anderen Worten, der Bezugspunkt für die Entfernung liegt vom Betrachter aus unterhalb der Geometrie. Die Grundgeometrie besteht aus zwei einfachen Dreiecken, daher ist an der Biegung im Subdivisionsunterschied gut zu erkennen, wie die Grenze innerhalb der ursprünglichen Dreiecke verläuft. Der Geometry Shader für Subdivision bearbeitet jedes Dreieck, das übergeben wird und kann natürlich auch entscheiden, das ein Dreieck schon klein genug ist und nicht weiter unterteilt werden muss. Diese Dreiecke werden allerdings auch wieder im VBO abgelegt und mit dem nächsten Durchgang des Transform Feed-

backs neu geprüft. Dadurch entsteht natürlich eine Anzahl an Aufrufen die nichts weiter bewirken und es wäre besser die Finalen Dreiecke aus der weiteren Bearbeitung zu entfernen. Jedoch ist es nicht möglich dies zu tun. Es ist zwar möglich in mehrere Buffer Objekte zu schreiben, aber diese sind nur für einzelne Eigenschaften aller Dreiecke wie z.B. die Normalen der Dreiecke. Dreiecksgruppen können damit nicht aussortiert werden. Dazu müsste das oder die VBO's außerhalb der GPU bearbeitet werden. Wie in der Erklärung des Transform Feedbacks bereits erwähnt, würde diese den Geschwindigkeitsvorteil negieren. Ist ein Dreieck final muss diese Information gespeichert werden. Andernfalls kann der Geometry Shader diese Inforamtion im nächsten Durchgang des Transform Feedback nicht abrufen. Nachdem die Dreiecke unterteilt wurden ist ihre Fläche geschrumpft. Diese wird erneut gegen die Entfernung getestet. Ist die Entfernung zur Kamera kleiner als die Fläche des Dreiecks, wird wieder Unterteilt. Denn dann ist das Dreieck trotz kleinerer Fläche immer noch nah genug an der Kamera, um zu grob zu erscheinen. Die maximale Anzahl der Subdivisionsschritte kann eingestellt werden. Die LoD grenzen dagegen sind unendlich, können aber anhand der Formel justiert werden. Nähere Dreiecke werden häufiger unterteilt, weil der Faktor Entfernung für sie sehr klein ist und die Fläche daher im Laufe der Durchgänge lange größer bleibt. Weiter entfernte Dreiecke werden weniger häufig unterteilt weil der Faktor Fläche schnell kleiner ist als die Entfernung. Ebenso werden größere Dreiecke öfter unterteilt, diesmal allerdings, weil die Fläche größer ist und häufiger geviertelt werden kann, bevor sie kleiner als die Entfernung ist. Die Verteilung der Grenzen für LoD können zwar mit Hilfe der Formel relativ frei gewählt bzw. wie eine Kurve modeliert werden, aber das Sinnvollste ist eine Verteilung die den nicht linear skalierten Werten nach der perspektivischen Division im Z-buffer, nahe kommt. Eine lineare Verteilung ist nah an der Kamera zu langsam und weit entfernt von der Kamera zu schnell.
Die möglichen Details können natürlich ebenfalls durch eine geringer aufgelöste Textur reduziert werden, da durch die lineare Interpolation für die Erzeugung der Eckpunkte, Abweichungen unvermeidbar sind. Doch dies bringt keinen Geschwindigkeitsvorteil. Dazu müsste die Subdivision ebenfalls angepasst werden. Die passende Auflösung der Textur hängt ganz von dem Einsatzzweck ab. Für große Terrains wird z.B. auch eine große Textur benötigt. Für geradlinige einfache Formen oder kleine Bereiche kann entsprechend auch eine deutlich kleinere Textur eingesetzt werden. Da ist z.B. auch eine Auflösung von 32x32 denkbar. Zu Optimierungszwecken der optischen Qualität kann auch so weit gegangen werden, die Auflösung der Textur, genau an das maximal eingestellte LoD anzupassen. Denn werden bei Detailreichen Texturen im Schnitt mehr Eckpunkte erzeugt als Texel vorhanden sind, bringt die hohe Subdivision keinen nennenswerten optischen Mehrgewinn. Wirklich relevant für die Geschwindigkeit wird die Texturauflösung erst, wenn Eckpunkte pro Texel erzeugt werden.

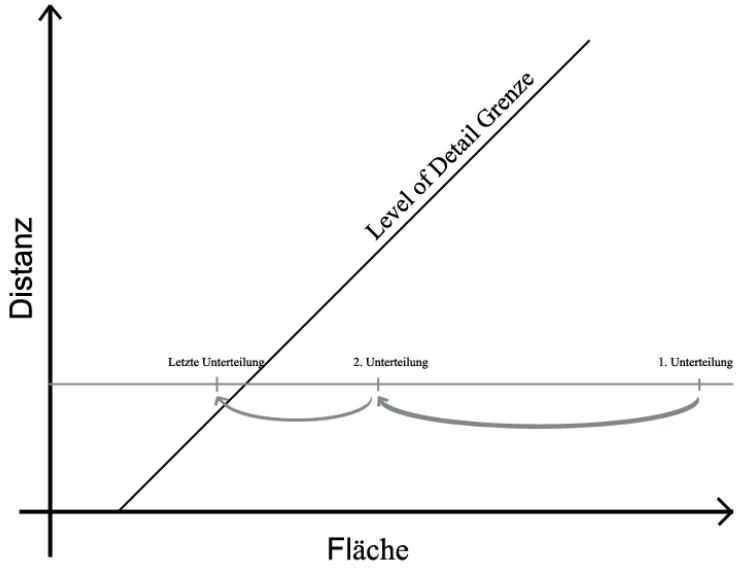

Abbildung 17: Skizze der LoD Grenze

3.5.1 Spalten und Schürzen

Das passgenaue Einfügen zusätzlicher Dreiecke ist eine gute Möglichkeit mit möglichst wenigen Dreieck und möglichst wenigen Artefakten Lücken zu schließen. In Bezug auf das sich hier herrauskristallisierende Verfahren bedeutet das, das jedes Dreieck für jede Kante ermitteln muss, ob das angrenzende Dreieck passt oder nicht und wie ein Dreieck in die Lücke einzufügen ist. Nachbarschaftsinformationen sind zwar mit dem Geometry Shader eingeschränkt erhältlich, aber ein Dreieck hat keinen Zugriff auf Informationen über bereits bearbeitete Dreiecke im aktuellen Durchgang. Wird bei dem LoD Algorithmus jedes Dreieck einzeln innerhalb der Subdivision bearbeitet, bringt der Trianglestrip adjecency nicht die benötigten Informationen. Dazu müsste es möglich sein auf ein und dem selben VBO arbeiten zu können, aber es muss von einem gelesen und auf einen anderen geschrieben werden. Die angrenzenden Dreiecke des Trianglestrip adjecency, müssten alle zusätzlich bearbeitet werden, damit die Information über deren LoD Stufe verfügbar wäre und die geeignetn Schritte für das zentrale Dreieck eingleitet werden können. Die über die Nachbarn gewonnenen Informationen könnten allerdings nicht sinnvoll erhalten werden, da die Dreiecke durch den Trianglestrip adjecency zwar mit ausgegeben werden, aber nochmal als zentrales Dreieck aus dem input VBO bearbeitet werden, ohne Zugriff auf die vorherige Berechnung. Das würde zumindest für den Letzten Transform Feedback Durchgang den vierfachen Aufwand bedeuten.

Für den Einsatz von Schürzen dagegen, gibt es mehrere Möglichkeiten. Die nächsten drei Techniken betreffen den Fall, das innerhalb des gesamten Grunddreiecks die gleiche Stufe an Unterteilung besteht.
In diesem Sinne ist die einfachste Lösung, das Grunddreieck an Ort und Stelle zu kopieren. Dies funktioniert nicht, wenn das Displacement negative Werte annehmen kann, bzw. dies müsste bei der Positionierung des kopierten Dreiecks berücksichtigt werden. Andernfalls könnte es in den sichtbaren Bereich hinein ragen. Probleme entstehen dabei wenn die Ränder des Dreiecks hoch über der Grundfläche liegen und der Blickwinkel flach durch eine Spalte tief unter die Oberfläche geht. Bei starken Bewegungen und unregelmäßigen Texturen kann dies dazu führen, das sich der sichbare Teil der Schürze stark mit der Bewegung verändert. Ebenso wenn die aneinander grenzenden Grundflächen nicht eben sind.
Eine weitere Möglichkeit ist die Erstellung einer großen Schürze jeweils um ein ganzes Grunddreieck. Dies wäre vergleichbar mit den standard Schürzen für viele Verfahren auf der CPU.Das Problem dabei ist, das schnell die Unterteilung innerhalb des Dreiecks eine so große Dreiecksanzahl erreicht hat, das die Schürze wegen des Outputlimits zu groß wird, um mit dem Geometry Shader dargestellt zu werden.
Dann gibt es noch die Möglichkeit zu allerletzt an jedes einzelne Dreieck rundherum eine Schürze zu hängen. Der Nachteil dabei ist offensichtlich. Die hohe Anzahl an resultierenden Dreiecken für Schürzen belegen den Speicher. Werden diese noch während des Transform Feedbacks angelegt, fällt dies noch schwerer ins Gewicht, da das VBO eine beschränkte Größe besitzt. Die Schürze für ein einzelnes Dreieck besteht aus sechs neuen Dreiecken. Mit Hilfe des Trianglestrips werden dafür acht Eckpunkte benötigt, ohne Trianglestrip 18. So werden mehr Schürzen als Oberflächendreiecke erstellt. Dies ist in Abb. 16 zu sehen auch wenn hier innerhalb der Dreiecke unterschiedliche LoD gibt. Um diese Anzahl zu verringern, können während bzw. vor dem letzten Subdivisionsschritt die Schürze um die vorletzte Stufe der Subdivision gelegt werden. Dadurch werden zumindest die Schürzen um das jeweils innere der vier neuen Dreieck gespart. Die durch die Subdivision entstandenen Interpolierten Mittelpunkte der Kanten müssen mit in die Schürze eingebaut werden, damit die Schürzen nicht teilweise zu tief oder zu hoch angebracht werden. So oder so muß auch die Information verfügbar sein, ob die Subdivisionsstufe eines Dreiecks final ist oder nicht, damit es nicht weiter unterteilt wird und der Shader merkt wann die Schürze angehängt werden muss. Wird nun noch innerhalb des Dreiecks unterschiedlich unterteilt, können nicht mehr nur am Rand des Dreieck Lücken auftreten, sondern auch überall darin. Mit Hilfe der in diesem Fall ohnehin für das LoD gewonnenen Informationen für die finalen Dreiecke und den Grenzen für das LoD, kann diese Technik auch hier implementiert werden.

4 Implementierung

In diesem Kapitel soll der Code der Arbeit erläutert werden. In dieser Arbeit hat das zugrunde liegende Mesh sehr wenige Details und erhält möglichst viele erst durch das Displacement Mapping. Für die Implementierung wurde OpenGL verwendet wegen seiner Plattformunabhängigkeit, freien Verfügbarkeit und weiten Verbreitung in der Lehre. Passend dazu wird auch die Shadersprache GLSL verwendet, welche seit OpenGL 2.0 fester Bestandteil desselben geworden ist. Als zusätzliche Bibliotheken wurden glut, für Fenster und Steuerungsoptionen sowie SOIL zum Laden der Texturen eingesetzt.

4.1 Framework

Im Framework wird kein Bild ohne Shader gerendert. Daher ist die Initialisierung der Shader und ihrer Texturen fest in der Main init implementiert. Der Code dafür steht für Shader und Texturen in verschiedenen Initialisierungsmethoden getrennt in der Klasse ShaderPacket. Umfassende Steuerungsmöglichkeiten sind in der Klasse Main enthalten, um die Tests zu erleichtern und geeignete Bilder der Features zu erhalten. Die Drehung der Geometrie wird mit glRotate durchgeführt, der Zoom dagegen durch Verschieben der Kameraposition. Das liegt daran, das die Geometrie, die an die Klasse TransformFeedback übergeben wird, nicht mit rotiert wird, damit der Blickwinkel auf die Geometrie unabhängig von den implementierten Verfahren frei gewählt werden kann. Die Kameraposition wird aber benötigt für das LoD und diese lässt sich leichter als Uniformparameter an die Shader übergeben im Gegensatz zu einer Transformation. Ist das Transform Feedback aktiv, wird es mit jedem neuen Display-Aufruf resettet, die Geometrie neu in das durch das Resetten leeren VBO geschrieben und danach erst das Transform Feddback an sich durchgeführt. Das Resetten ist eine verkürzte Neuinitialisierung um die Zeit des Löschen und Erstellens der Buffer, Queries und anderer Variablen zu sparen.

4.2 Fps-Messung

Derzeit wird die glutTimerFunc zur Zeitmessung verwendet. Jede Sekunde wird die Anzahl der gerenderten Frames ausgegeben. Dies geschieht in der Main Klasse. Der erste Aufruf von glutTimerFunc steht in der init, alle weiteren sind rekursiv in der TimerFunc myTimer. Durch das Zählen der Frames pro Sekunde allerdings keine Kommazahl errechnet. Um das zu erreichen könnte ein Mittelwert über mehrere Sekunden errechnet oder die Zeit für das Rendern eines Frames gemessen werden. Da jedoch echtzeitfähige Frameraten angestrebt werden, sind Kommastellen nur wenig mehr Aussagekräftig.

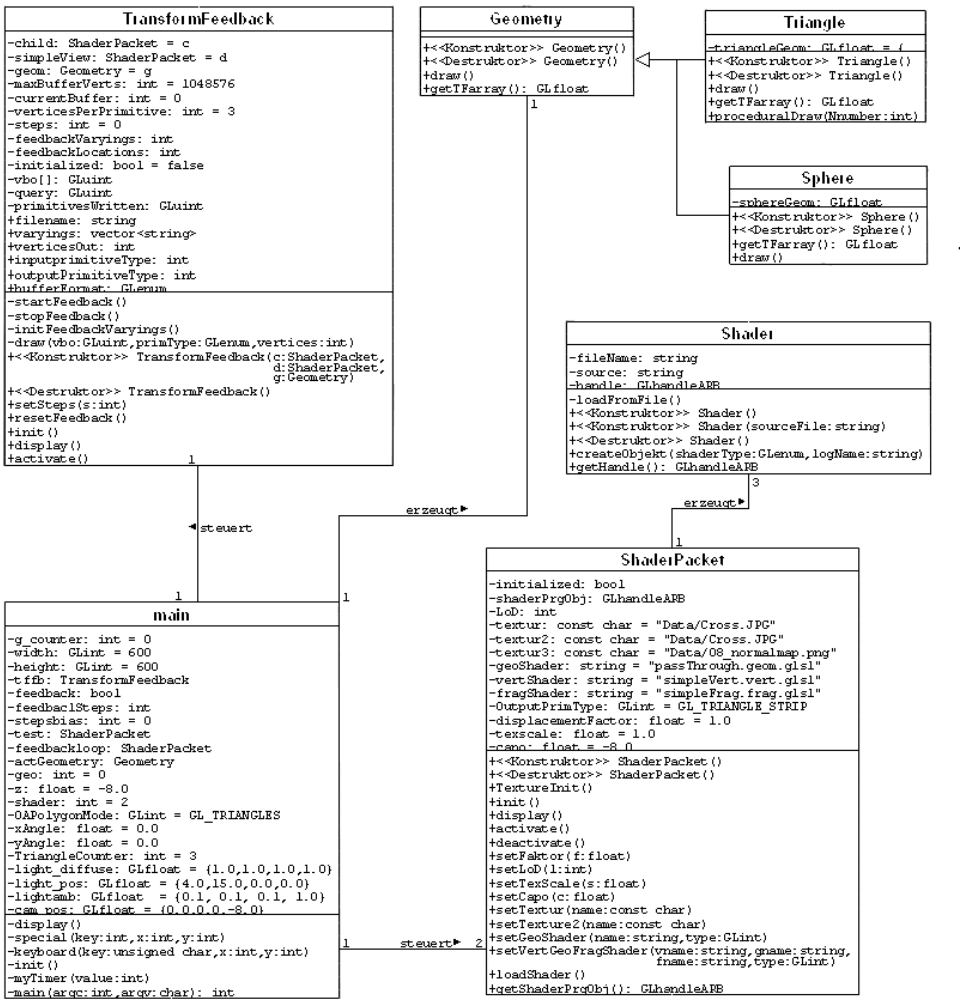

Abbildung 18: Klassendiagramm

4.3 Geometrie

Hier werden jeweils für ein geometrisches Objekt verschiedene Methoden zur Darstellung und Variation grundlegender Eigenschaften des Objektes bereitgestellt. Bisher gibt es die Klassen Triangle und Sphere. Triangle besitzt noch eine Methode, die zur Laufzeit die Anzahl der angezeigten Dreiecke erhöhen oder senken kann, um damit leichter Vergleichstest für Performanz durchzuführen. Die Klasse Geometry selbst enthält hauptsächlich abstrakte Methoden, die in den abgeleiteten Klassen Triangle und Sphere überschrieben werden. Damit ist es vor allem für die Klasse Transform Feedback leichter, die Geometrie zu initialisieren. Spätere Codeänderungen werden auch erleichtert, da nur an wenigen Stellen im Code eine neue Geometrieklasse berücksichtigt werden müsste.

4.4 Shaderverwaltung

Diese Klasse basiert grundsätzlich auf den Shaderklassen aus der Studienarbeit von Andreas Arb [vA08], wurde aber einerseits vereinfacht und andererseits erweitert. So gibt es mehrere Mehtoden, die dazu dienen, die Shader oder aber die Uniformparameter zu ändern, zu denen auch die Texturen gehören. Die Methode setFactor ändert den Wert mit dem die Höhe des Displacement Mappings skaliert werden kann. Bei setTexScale handelt es sich um die Skalierung der Texturkoordinaten. Mit setTextur1 und setTextur2 können neue Texturen übergeben werden, was allerdings eine Neuinitialisierung erfordert. Durch setCapo wird die Kameraposition im Shader aktualisiert, da diese für die Bestimmung des LoD sehr wichtig ist. Einzelne Shader oder alle drei gleichzeitig können z.B. mit setVertGeoFragShader oder setVertShader gewechselt werden. In diesem Fall handelt es sich allerdings nicht um einen Wechsel des Shader Programmobjekt, sondern um neue Shader, so dass diese neu geladen und aktiviert werden müssen. Die Klasse ShaderPacket lädt die Texturen in der Methode TextureInit mit Hilfe der Soil Bibliothek. Des Weiteren werden hier die Shader eingelesen, gebunden und übergeben. Da mehrere Shaderpackete in einem Renderpass benötigt werden können macht es daher Sinn, mehrere Instanzen dieser Klasse zu erstellen, anstatt alles während des Renderpasses neu zu initialisieren. Die verschiedenen Pakete bzw. Shaderprogrammobjekte müssen dann nur deaktiviert oder aktiviert werden, was schneller ist. Die verschiedenen Shader Programmobjekte können so schnell gewechselt werden, dies wird jedoch nur in Kernbereichen angewandt. Um Texturen oder einzelne Shader zu wechseln werden die vorhandenen Programmobjekte verändert und müssen dann neu initialisiert werden. Das verlangsamt den Programmablauf für kurze Zeit. Daher gibt es für die Initialisierung der verschiedenen Attribute eigene Initialisierungsmethoden. Die Init bindet die Shaderdateien an ein Shader Programmobjekt, TextureInit läd die

Texturen in Units und activate aktiviert das Shader Programmobjekt und übergibt die Textur an Uniformparameter. Die Uniformparameter für die Texturen sowie die für Skalierungsfaktoren und Licht und Kameraposition werden nach der Aktivierung der Shader initialisiert, da sonst kein Multitexturing möglich ist. Dieses wird benötigt, da in einer Textur Heightmap und Normalmap gespeichert sind und eine zweite Textur für die Farbwerte benötigt wird.

4.5 Transform Feedback

In dieser Arbeit wurde das Transform Feedback wie folgt realisiert. Dem Konstruktor werden die zu verwendenden Shaderpackete bzw. Shader Programmobjekte und die zu behandelnde Geometrie übergeben. Das ist ein Shader Programmobjekt für Subdivision und LoD, sowie eines für Displacement und Beleuchtung. Danach muss die init ausgeführt werden, um zunächst die zwei Vertex Buffer Objekte und das Query zu erzeugen. Außerdem wird noch die Methode initFeedbackVaryings aufgerufen, welche die Varyings lädt und sofort überprüft, bevor der Initialaisierungsstatus auf True gesetzt wird. Wenn auch nur eine Varying Variable nicht richtig geladen oder gefunden werden kann funktioniert das Transform Feedback nicht. In den Shadern müssen daher beim Vertex Shader angefangen bis zum Fragment Shader die Varyings vorhanden und richtig gesetzt werden. Wird z.B. eine Varying Variable die vom Geometry Shader in den Fragment Shader weitergereicht wird, dort nicht benutzt, erkennt der Compiler dies, so dass die Variable dann nicht gefunden werden kann. Selbst eine Multiplikation mit 0 wird als inaktiv gewertet, so dass die betroffenen Varyings zumindest minimal in eine Berechnung einfließen müssen. Dieser Fall kann dann auftreten, wenn z.B. eine Varying Variable im Geometry Shader mit Transform Feedback genutzt wird, also die Ausgabe im nächsten Durchgang zur Eingabe wird, der Fragment Shader jedoch praktisch keine Verwendung für die an ihn weitergereichte Variable hat. Um selbst die minimalste Fehlereinstreuung zu verhindern, wird daher der Fragment Shader, der die Varyings im Transform Feedback aktiv hält, nicht für den dem Feedback folgenden Durchgang mit Rasterisierung benutzt. So bleibt die fehlerhafte Einbeziehung eigentlich unbenutzer Daten in einem Fragment Shader, der aufgrund deaktivierter Rasterisierung während dem Feedback nicht eingesetzt wird. Nachdem das Transform Feedback jetzt bereit ist, können von hier ab die drei Methoden resetFeedback, activate und display in der Klasse Main ausgeführt werden. Erstere setzt die Daten in den beiden VBO wieder auf Null zurück. Das ist am Anfang noch nicht nötig, da die Buffer noch leer sind. Aber da die Display Methode der Klasse Main eine Dauerschleife ist und nach dem ersten Durchgang wieder auf der Grundgeometrie gearbeitet werden soll, notwendig. Die Methode activate überträgt danach die Eckpunkte aus der an die Klasse übertragenen

Geometrie in den ersten Buffer. Auf dieser Grundlagen kann dann die Methode Display in einer Schleife den Subdivisions Geometry Shader mehrfach hintereinander auf die Geometrie anwenden. Dafür wird zunächst das gewünschte Shader Programmobjekt aktiviert, die Aufzeichnung der Daten in den zweiten Buffer mit startFeedback gestartet, die im ersten VBO gespeicherte Geometrie mit dem Shader bearbeitet und anschließend die Aufzeichnung wieder mit stopFeedback beendet. Durch den Tausch der VBO's ist die Schleife danach wieder für den nächsten Durchgang bereit. Start- und stopFeedback aktivieren und deaktivieren neben dem eigentlichen Aufzeichnen der Daten auch noch das Query und die Rasterisierung. Da einerseits der Shader im Feedback und andererseits die ausgeschaltete Rasterisierung noch keine Darstellung der Geometrie vorsehen, wird als letztes in der Methode Display noch einmal das Shader Programmobjekt gewechselt und mit eingeschalteter Rasterisierung die modifizierte Geometrie gerendert.

4.6 Umsetzung der Displacement Mapping Verfahren

Generell werden Höhenwerte im Alphawert der Textur abgespeichert. Der Geometry Shader liest die Farbe aus der Textur und erzeugt an der Stelle, an der der Farbwert der Textur auf dem Dreieck liegt, einen neuen Punkt in der Welt mit dem Alphawert als Höhe über dem Dreieck. Da alle Koordinaten mittels Interpolation bestimmt werden, wird in den meisten Fällen kein Texturpixel direkt getroffen. Daher werden die umliegenden Pixel anteilig eingerechnet.

4.6.1 Einzelner Shader ohne Transform Feedback

Der hier beschriebene Shader diente als Erkenntnisgrundlage, Prototyp und als Bruteforce Version von Displacement Mapping im Geometry Shader. Der loopShader besitzt eine Input Varying und eine Output Varying jeweils für Texturkoordinaten. Weitere Varyings gibt es nicht, da diese nur für Features benötigt würden, die in diesem Shader nicht implementiert wurden. Er übernimmt sowohl die Unterteilung der Geometrie als auch die Verschiebung der Eckpunkte. Dabei wurde das Hauptaugenmerk auf Adaptivität gelegt. Wie schon in Kapitel 3.1 erwähnt, wird dazu zwischen den Eckpunkten Interpoliert, um Start- und Endpunkte für die ineinander liegenden Schleifen zu erhalten. Durch die Verwendung der gleichen Schrittgröße bei der Interpolation für Weltkoordinaten und Texturkoordinaten ist gewährleistet, dass die erzeugten Punkte genau an der Stelle in der Welt liegen, wo auch die Information aus der Textur in der Welt liegt. Die Texturkoordinaten bilden so etwas wie die x und y Koordinaten während die Höhenkarte im Alphakanal der Textur die z Koordinate liefert. Das Zusammenfügen der Eckpunkte zu Dreiecken wird mit Trianglestrips

realisiert. Dadurch müssen weniger Punkte ausgegeben werden, als bei der Ausgabe von einzelnen Dreiecken. Die Trianglestrips müssen in der inneren Schleife erzeugt werden und jedes mal etwas länger werden, da die äußere Schleife von einem Eckpunkt über die Kanten zu den beiden anderen läuft und die innere zwischen den Kanten des Dreiecks verläuft. Außerdem brauchen die Strips zwei Reihen an Punkten, die abwechselnd ausgegeben werden müssen, was die Bestimmung der jeweiligen Interpolationsschritte verkompliziert. Abb. 19 zeigt exemplarisch einen Streifen im Dreieck.

Listing 1: LoopShader Pseudocode

```
1  main()
2  {
3    Initialisierung der Variablen
4    for(i = 1, i < 10, i++)
5    {
6      Interpolation der Welt- und
7      Texturkoordinaten für innere Schleife
8      for(j = 0, j <= i, j++)
9      {
10       Interpolation der Welt- und
11       Texturkoordinaten für genaue Position
12       Displacement und Ausgabe Punkt Reihe 2
13       Displacement und Ausgabe Punkt Reihe 1
14     }
15     Displacement und Ausgabe letzter Punkt Reihe 2
16   }
17 }
```

Es ist zu erkennen, dass die blaue Reihe nur Zwei Punkte besitzt, während die grüne Drei Punkte hat. Die innere Schleifenvariable j gibt die Anzahl der Interpolationsschritte für die innere Schleife an und hängt dierkt von der äußeren Schleifenvariable i ab. Um die Schrittgröße entsprechend der aktuellen Position innerhalb beider Schleifen zu skalieren, wird j durch i dividiert. Dabei ist in dieser Formel die erste Reihe durch i und die zweite Reihe durch i+1 zu berechnen. Die Variable i stellt sowohl in der äußeren als auch der inneren Schleife die Schrittanzahl für die Interpolation dar. Wie im Pseudocode zu sehen, läuft die innere Schleife von 0 bis zur aktuellen äußeren Schleifenvariable i und läuft damit für jeden weiteren äußeren Durchgang einmal häufiger als zuvor. Das heißt, dass die Schrittgröße umgerechnet in Weltkoordinaten gleich bleibt, aber die Schrittgröße für die Interpolationsmethode ständig kleiner wird. Denn sie muss zu Beginn 0 und 1 sein, da für das Kopfdreieck kein Zwischenschritt benötigt wird, in den darauffolgenden Schleifendurchgängen jedoch jeweils eine weitere Unterteilung durchführen. Bei der erzeugung der Eckpunkte wird zunächst der

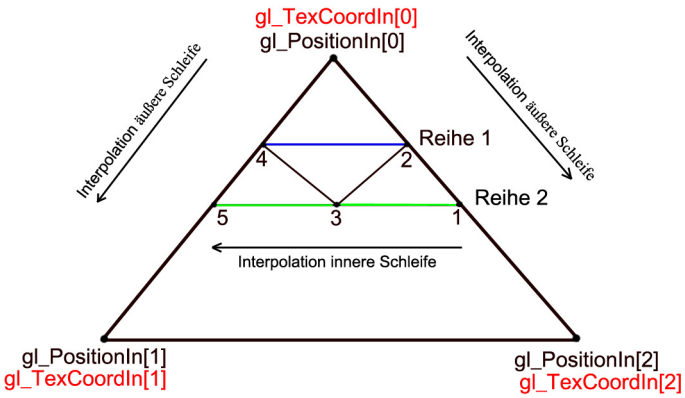

Abbildung 19: Skizze zu Variablen und Schleifenstruktur des LoopShaders

Farbwert aus der Textur gelesen, um den neuen Weltpunkt entsprechend des Alphawerts in Richtung der Flächennormale zu verschieben. Da die zwei Reihen an Punkten in einer Schleife durchlaufen werden, kann nur eine gerade Anzahl an Punkten für den Dreieckstreifen eingegeben werden. Die Dreieckstruktur erfordert aber, dass die zweite Reihe immer einen Punkt mehr enthält als die vorherige. Deshalb wird der letzte Punkt in Reihe zwei in die äußere Schleife ausgelagert. Das Punktlimit des Geometry Shaders ist der Grund, warum hier keine weiteren Features, wie z.B. Bump Mapping für Beleuchtung und Normaleninterpolation implementiert wurden. Die Anzahl der Oberflächendreiecke steigt exponentiell an, während Schürzen nur linear ansteigen. Dabei könnten mit 128 Punkten maximal 49 Oberflächendreiecke angezeigt werden, damit die Schürzen noch alles umschließen. Bei 64 Oberflächendreiecken stehen bei den Schürzen schon zwei Eckpunkte zuwenig zur Verfügung.

4.6.2 Punktshader

Dieser Shader wurder implementiert um die Stärken und Schwächen der Punktlösung aufzuzeigen, die schon in Kapitel 3 erläutert wurden. Der Code dieses Shaders ist in vielerlei Hinsicht einfacher, als der des vorherigen Abschnitts. Erstens daher, dass er nur mit Punkten und nicht mit Dreiecken arbeiten muss. Dadurch braucht er zwar immernoch eine zweidimensionale Schleife, aber die innere Schleife muss keine zwei Reihen mehr erzeugen und somit müssen auch weniger Interpolationen durchgeführt werden. Außerdem können effektiv mehr Punkte ausgegeben werden, da keine Punkte mehr für nahtlose Trianglestrips übereinander liegen müssen. Zweitens sind dadurch die Berechnungen einfacher geworden. Aufgrund

des Problems mit der konstanten Punktgröße, wird hier vor der Interpolation die Kantenlängen des zugrunde liegenden Dreiecks berechnet, um die Interpolationschritte der Länge der Kante und der Punktgröße anpassen zu können. So entstehen in langgestreckten Dreiecken keine unterschiedlich großen Lücken zwischen horizontalen und vertikalen Reihen.

Listing 2: Punktshader

```
1  main()
2  {
3    Initialisierung der Variablen
4    Anpassung der Schrittgröße an Kantenlänge
5    for (i = 0, i<Schritte, i++)
6    {
7      Interpolation der Welt- und
8      Texturkoordinaten für innere Schleife
9      for (j = 0, j<i, j++)
10     {
11       Interpolation der Welt- und
12       Texturkoordinaten für genaue Position
13          Displacement und Ausgabe eines Punktes
14     }
15   }
16 }
```

4.6.3 Subdivision mit Transform Feedback

Dieses Verfahren ist hauptsächlich auf zwei Shader aufgeteilt, welche nacheinander im Transform Feedback ausgeführte werden. Der erste Steuert die Subdivision und erzeugt innerhalb des Eingabedreiecks vier kleinere neu, wie in Abb. 12 zu sehen. Der zweite Shader arbeitet auf der unterteilten Geometry um diese zu verschieben. Dazu verwendet eine Textur mit Displacementmap und einen Skalierungsfaktor für die Stärke des Displacements. Von den Varying Variables aus dem Subdivisions Shader müssen nicht mehr alle beibehalten werden, da dieser Shader nicht im Kreislauf des Transform Feedbacks steht. So entfällt die Variable für Vertices, die stattdessen per gl_PositionIn abgerufen werden. In der Main Methode werden zusätzlich zu dem Displacement auch die Normalen für Phongbeleuchtung im Fragment Shader vorbereitet. Dazu müssen die Stützvektoren des Tangentenraums bestimmt und für jeden Eckpunkt mit den Werten aus der Normalmap multipliziert werden, damit sie ihre Entsprechung im Weltkoordinatensystem erhalten. Die eingegebene Normale wird erst durch die Normale für die Beleuchtung überschrieben, nachdem die verschobene Eckpunktposition berechnet wurde.

Listing 3: Displacement

```
1  main()
2  {
3    Initialisierung der Variablen
4    Berechnung der Tangente
5    for (i = 0, i<3, i++)
6    {
7      Berechnung der Normale
8      Berechnung der Bi-Normale
9      Verschiebung und Skalierung der Position
10     Berechnung der Beleuchtungsnormale
11     Ausgabe des Eckpunktes
12   }
13   Zusammenfügung von drei Punkten zu einem Dreieck
14 }
```

Die Eckpunkte wurden nicht schon im Vertexshader transformiert, sondern erst nach dem Displacement. Dies ist nötig, da neue Geometrie hinzugefügt wird. Das Hinzufügen neuer Geometrie nach der Transformation durch die Matrizen Modelview und Projection führte zu Problemen, wie zu früh ausgeblendete Flächen bei einer Drehung weg von Kamera, obwohl die Beleuchtung korrekt berechnet wurde.

4.7 Bump Mapping

In dieser Arbeit wird der Tangentenraum verwendet, da dies besser zu dem Konzept der Adaptivität passt, das auch einige andere Aspekte der Shader entschieden hat (Interpolation oder Koordinatenbestimmung, LoD in Abhängigkeit der Dreieckgröße,...). Die Hightmap nimmt also nur einen Kanal in Anspruch und die Normalmap drei Kanäle. Vier gibt es in einer Textur, daher wurden diese beiden Maps zusammen in eine Textur gelegt. Da das zu rendernde Objekte ohnehin eine Colormap braucht, ist für das Displacement Mapping nur eine zusätzliche Textur erforderlich. Während des Displacement Shaders wird die Varying Variable für die Normale noch für das Verschieben der Eckpunkte benötigt. Also wird zunächst der Tangentenraum bestimmt und nachdem die Normale für das Displacement genutzt wurde, wird ihr aus der Normalmap die neue Beleuchtungsnormale zugewiesen. Dazu werden die Werte aus den Kanälen mit den entsprechenden Vektoren des Tangentenraumes multipliziert. Bevor sie ausgegeben werden kann, muss sie noch normalisiert und mit der Normalmatrix multipliziert werden. Nachdem die neue Normale an den Fragmentshader weitergereicht wurde, kann sie dort für Phongbeleuchtung eingesetzt werden.

4.8 Volumen

Insgesamt ist Volumen Displacement Mapping auf verschiedene Arten möglich, aber gleichzeitig bisher durch die zusätzliche Dimension und damit zusätzlich zu berücksichtigenden Randbedingungen in größeren Detailgraden nicht echtzeitfähig. Die Idee das Meshing über Farbnachbarschaft und Kantenlänge zu steuern enthielt zu viele Schwierigkeiten bei der Erstellung einer Textur und wurde aufgrund der gesenkten Adaptivität des Algorithmus durch die Einschränkung mit Farbnachbarshaft und Entfernung, nicht umgesetzt.
Zu Testzwecken wurde jedoch ein Verfahren implementiert, welches keine Dreiecke sondern nur Punkte erzeugt. Die Vor- und Nachteile wurden bereits im Abschnitt 3.2 erörtert. Weitere Nachteile durch die zusätzliche Dimension gibt es bei den Punkten allerdings nicht. Obwohl der Versuch das Volumen mit Punkten darzustellen einige Schwierigkeiten aufweist, wurde es doch den anderen Verfahren vorgezogen, da es kein Meshing erfordert und leicht aus dem Verfahren in 3.3 entwickelt werden kann. Ein Mesh könnte zwar mit Hilfe der Positionen der Punkte und Marching Cubes erzeugt werden, dies wurde jedoch auf Grund der Probleme und zu erwartenden Artefakte durch die häufig eingesetzte Interpolation während der Subdivision, nicht eingesetzt. Um die erhöhten Anforderungen durch die dreidimensionalen Daten zu bewältigen, kommen 3D Speichergrids mit weniger als drei regelmäßigen Komponenten nicht in Frage. Die Sartbedingungen hier erlauben regelmäßigere Strukturen. Aber durch das Speicherplatzproblem bei dreidimensionalen Texturen wurde die zweidimensionale Textur mit 4 Höhenwerten verwendet. Der Ansatz mit zwei Intervallen aus den vier Punkten wurde allerdings nicht verfolgt, weil das Problem der Meshgenerierung in diesem Fall am besten mit einem Marching Cubes Algorithmus gelöst wird, welcher allerdings, wie schon erwähnt, nicht eingesetzt wird. Der Codes des Subdivision Shaders sieht genau so aus wie auch beim Subdivision Shader aus Kapitel 4.6.3, denn die Funktionsweise der rekursiven Unterteilung basiert auf der Dreiecksstruktur. Daher werden auch hier Dreiecke erzeugt obwohl später nur deren Eckpunkte angezeigt werden. Der Code für das eigentliche Displacement ist natürlich komplexer.

Listing 4: Volumen Displacement

```
1 main()
2 {
3   Initialisierung der Variablen
4   Berechnung der Tangente
5   for (i = 0, i<3, i++)
6   {
7     Berechnung der Normale
```

```
8       Berechnung der Bi-Normale
9       if (Punkt1)
10        Verschiebung und Skalierung der 1. Position
11        Ausgabe des 1. Eckpunktes
12      if (Punkt2)
13        Verschiebung und Skalierung der 2. Position
14        Ausgabe des 2. Eckpunktes
15      if (Punkt3)
16        Verschiebung und Skalierung der 3. Position
17        Ausgabe des 3. Eckpunktes
18      if (Punkt4)
19        Verschiebung und Skalierung der 4. Position
20        Ausgabe des 4. Eckpunktes
21    }
22  }
```

Zum einen ist eine weitere Texturunit hinzugekommen, da die Displacementmap nun vier Kanäle benötigt und die Normalmap in andere Texturen ausgelagert werden muss. Zum anderen werden vier Punkte verschoben in der Schleife. Wie auch schon bei dem einfacheren Displacement Shader werden die Eckpunkte in einer Schleife behandelt. Innerhalb der Schleife muss allerdings, bevor ein Punkt verschoben wird geprüft werden, ob an der Stelle in der Textur überhaupt ein Punkt steht. Daher wird per If-Anweisung der aktuelle Farbkanal mit einem Grenzwert abgeglichen. Liegt der Kanalwert darunter, wird kein Punkt erzeugt. Dadurch müssen für jeden an den Shader übergebenen Eckpunkt vier If-Anweisungen durchgeführt werden. Für jeden Farbkanal muss geprüft werden, ob der Farbwert einen Punkt repräsentieren soll oder nicht. Durch die Struktur der Subdivision ist eine neue Technik möglich geworden, mit der die Lücken zwischen den Punkten vermindert werden können. Da bei der Generierung von vier kleinern Dreiecken viele der neu erzeugten Punkte mehrmals übereinander liegen, können diese ein Stück Richtung Zentrum des Dreiecks verschoben werden. So werden die eigentlich redundanten Punkte zur Schließung der Lücken genutzt und eine minimale Verkleinerung des Grunddreiecks ist nur an den Rändern zu bemängeln. (Abb.20) Aber auch hier sind zusätzliche Steuerungstechniken oder die Verwendung von echter Geometrie nötig, um die Lücken und Überlappungen effektiv zu vermeiden. (siehe 3.2)

4.9 LoD

Dieses Kapitel beschreibt, wie LoD in den zuvor beschrieben Algorithmus mit Transform Feedback exemplarisch integriert wurde, um die Performanz zu verbessern. Die Verwendung verschiedener Texturauflösungen, den Mipmaps, kommt hier nicht in Frage, da durch die Adaptivität mittels Inter-

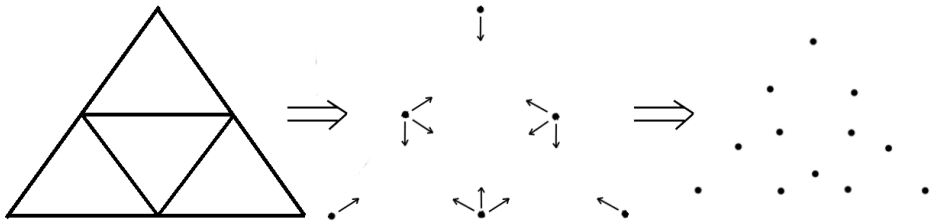

Abbildung 20: Skizze zur besseren Nutzung von redundanten Punkten

polation, die Texturauflösung unabhängig von der Punkterzeugung ist. Es werden bei unterschiedlichen Texturauflösungen immer gleich viele Punkte erzeugt. Die zusätzliche Verwendung von Mipmaps kann nach wie vor für die Farbgebung der Geometrie eingestzt werden. Um in den fest laufenden Durchgängen des Transform Feedback einen Finalen Status vor der letzten Iteration des Geometry Shaders zu erhalten, muss dies in irgendeiner Form für den Shader erkennbar sein. Daher wurde die Information in den bisher ungenutzten Alphakanal der Eckpunktfarbe gespeichert. Die anderen drei Kanäle speichern nach wie vor die Tangente für Normal Mapping. Zu beginn hat ein Eckpunkt den Alphawert null. Sobald es Final wird, wird der Wert auf eins geändert. Erkennt der Shader das das Dreieck Final ist reicht er es unverändert weiter. Im Code wurde das folgendermaßen realisiert. Der Shader musste im vergleich zum einfachen Subdivisions Shader durch If-Anweisungen aufgeteilt werden. Die erste If-Anweisung überprüft ob es sich um ein Finales Dreieck handelt oder nicht. Dazu wird getestet ob der Alphakanal kleiner 0,5 ist. Ist es wahr, also noch nicht Final, wird in einer Zweiten If-Anweisung geprüft ob die aktuelle Fläche kleiner als die Entfernung ist. In diesem Fall wird Unterteilt. Andernfalls das Dreieck als Final markiert, also der Alphawert auf eins geändert und durchgereicht. Tritt zu Beginn der Fall ein, dass Alpha größer als 0,5 ist, wird das Dreieck unverändert durchgereicht.

$$Entfernung - 4 < 250 Flaeche$$

Die zusätzlichen Faktoren in der Formel für LoD verbessern das Ergebnis. Der Bias Wert von -4 dient dazu, die Grenzen insgesamt ein klein wenig von der Kamera zu entfernen, da sie sonst alle im Augpunkt konvergieren und somit hinter der Nearplane verschwinden. Der Multiplikator bringt die Größen in die gewünschten Verhältnisse, da die Dreiecke im direkten Vergleich zu klein waren, um noch unterteilt zu werden. Die genaue Wahl der Faktoren hängt dabei stark von der Implementierung des gesamten Systems ab und können nicht als universell einsetzbar angesehen werden.

Wie in einem Baum wird mit jedem neuen Subdivisionsschritt bzw. wachsender Baumtiefe der Detailgrad erhöht. Im Gegensatz jedoch zum Baum und den meisten anderen Verfahren mit festen Datenstrukturen und Detailstufen, gibt es bei dem hier verwendeten Algorithmus keine Blöcke mit gleicher Detailstufe, denn die neu erzeugten Dreiecke werden alle einzeln auf den zugehörigen LoD Bereich geprüft. Daher verläuft jede Grenze des LoD kreisrund um die Kamera. Abb. 26 Auf die Feineinstellung, das LoD an die Auflösung der Textur anzupassen, wurde in dieser Arbeit verzichtet, da es hauptsächlich um die erreichbare Performanz geht. Wie schon im Kapitel drei erläutert, wird die Texturauflösung für die Geschwindigkeit erst relevant, wenn Eckpunkte pro Texel erzeugt werden, was zwar möglich und verbreitet ist, aber wegen der teureren Adaptivität nicht eingesetzt wurde.

4.9.1 Spalten und Schürzen

Für den implementierten Algorithmus wurde zunächst eine Bruteforce Variante getestet, um zunächst die Machbarkeit zu beweisen. Der Grundgedanke dabei ist, dass möglichst wenig über das Dreieck bekannt sein muss, damit keine zusätzlichen Informationen übergeben und verarbeitet werden müssen. Dabei wurde an jedes Dreieck eine Schürze gehängt. Um diese ungünstige aber funktionierende Lösung zu beschleunigen, wurde die Anbringung von Schürzen auf sinnvolle Bereiche beschränkt. Erstens wird nicht mehr an der letzten Unterteilungsstufe die Schürze angebracht, sondern an der Vorletzten. Denn es wird ohnehin schon geprüft ob ein Dreieck final ist oder nicht. Zweitens sind die Schürzen nur unmittelbar hinter einer LoD Grenze nötig. Davor werden sie von der Kamera nicht gesehen und zur Schließung einer Lücke braucht es nur eine Schürze. Die weiter hinten folgenden Oberflächendreiecke brauchen keine Schürzen. Obwohl der Shader keine direkte Information über die Nachbardreiecke hat, weshalb zunächst jedes Dreieck eine bekommen hat, lassen sich die meisten überflüssigen Schürzen mit einer zusätzlichen If-Anweisung verhindern. Dazu wird die Entfernung des Aktuellen Dreiecks mit der aktuell relevanten LoD Grenze verglichen und eine Schürze erstellt wenn die Entfernung zur Grenze ein Epsilon unterschreitet.

$$|(Entfernung - 4) - 250Flaeche| < \frac{Flaeche}{2} + 4$$

Der Betrag von Entfernung minus Fläche muss kleiner sein als ein Epsilon, das sich den schrumpfenden Bezugswerten anpasst. So bleibt der Bereich, in dem Schürzen gezeichnet werden, klein und nah an den Grenzen. Die Formel ist unabhängig von Transform Feedback Durchgang und Dreiecksgröße gleich, da die Dreiecksgröße in der Formel explizit berücksichtigt ist und die Durchgänge implizit auch. Das Ergebnis ist in Abb. 26

zu sehen. Drittens werden nicht mehr Einzeldreiecke für die Schürzen und Unterteilungen verwendet sondern Trianglestrips.

Um die Schürzen im Shader einzufügen musste der Shader etwas verändert werden. So wird bei der Markierung der finalen Dreiecke ebenfalls unterteilt. Das ist nötig, da an dieser Stelle zusätzlich die Schürze eingefügt wird. Ohne die Unterteilung bzw. mit der vorherigen Shaderstruktur hätte die Schürze nicht am vorletzten Subdivisionsschritt durchgeführt werden können. Die Schürze kann erst angebracht werden wenn sicher ist, dass das aktuelle Dreieck final ist. Die Schürze an sich kann nicht mit dem Shader unterteilt werden, obwohl sie im selben vbo gespeichert wird und bei einem eventuell folgenden Durchgang mit geprüft wird. Sie wird sofort bei Erstellung selbst als Final markiert. Allerdings könnte ohne dieses Wissen die Schürze angebracht werden, obwohl das Dreieck noch weitere male Unterteilt wird und somit alle vorherigen Schürzen an diesem Dreieck überflüssig werden.

4.9.2 In Bezug auf Volumen

Da nur die Variante mit Punkten zu Testzwecken implementiert wurde, gibt es dort zwar LoD, aber keine Schürzen. Der Shader für die Subdivision in dem auch das LoD bearbeitet wird, ist im Prinzip der gleiche wie auch bei dem vorherigen Verfahren. Es besteht außer den fehlenden Schürzen kein Vorteil für Geschwindigkeit oder Struktur aufgrund der Verwendung von Punkten für das Displacement Mapping, da diesbezüglich alles erst im anschließend angewendeten Geometry Shader implementiert ist. Die Subdivision findet also auf Basis von Dreiecken statt, während der Umstieg auf Punkte erst später beim eigentlichen Displacement Mapping geschieht.

5 Ergebnisse und Bewertung

In diesem Kapitel werden die erzielten Ergebnisse mit einem puren Rendering ebenso vieler Dreiecke verglichen, um auch ohne Nachstellen der Testbedingungen eine Vorstellung der Verhältnisse erhalten zu können.

5.1 Testbedingungen

Das pure Rendering bedient sich keinerlei Beschleunigungstechniken außer dem Trianglestrip. Die Dreiecke werden im Immediate Mode geschrieben, also ausschließlich mit glBegin(), glEnd(), glVertex3f() etc. bzw. ohne Displaylisten oder Vertex Buffer Objekte. Es wird ebenfalls kein Displacement Mapping bei der puren Version durchgeführt, es zeichnet lediglich Dreiecke. Der für die Tests verwendete Rechner besitzt einen Intel Core2Duo 6400 2,13GHz mit 1Gb Ram und einer NVidia GeForce 8800 GTX, 768MB, PCIe 16x, v163.75. Die Texturauflösung beträgt 512x512 und variiert nicht bei den Tests, da es für die Geschwindigkeit unerheblich wäre. Die verwendeten Texturen sind Höhen- und Normalmap, die in einer Texturunit zusammengefasst sind und eine Grastextur für die Farbgebung. Die Auflösung des Anzeigefensters betrug für die Test 600x600. Eine Änderung der Fensterauflösung wurde bis 1024x800 getestet, erbrachte aber keinen merklichen Geschwindigkeitsunterschied. Die gerenderte Szene besteht aus zwei Dreiecken, die in einem Quadrat angeordnet sind. Daher werden bei niedrigen Subdivisionsmaxima keine hohen Dreieckszahlen erreicht.

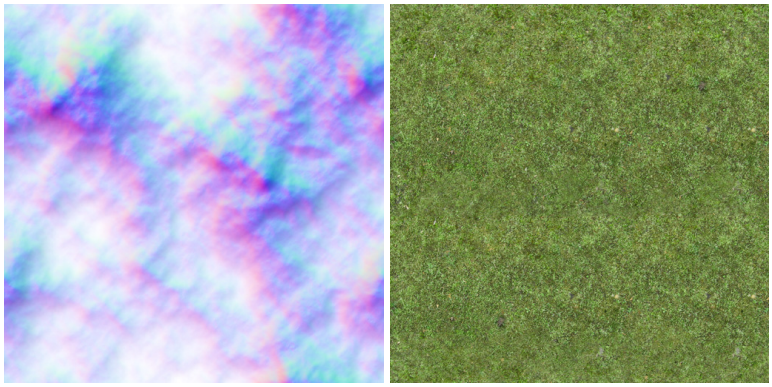

Abbildung 21: Die beiden Texturen für die Tests

5.2 Ergebnisse

Durch die LoD Grenzen mitten durch die Dreiecke, sind nicht immer die exakt gleichen Dreieckszahlen im Vergleich zu den verschiedenen anderen

Subdivisionsmaxima zu erreichen. Daher ist die Tabelle mit den genauen Framezahlen nicht vollständig ausgefüllt. Die Ergebnisse dieser Tests sind in der folgenden Liste und Grafik zu sehen. Die rechten Spalten in der Tabelle zeigen die zweite Variante die zwar Schürzen auf Streifen reduziert, aber immernoch recht viele Schürzendreiecke besitzt. Die Schürzen werden in einem kleinen Bereich hinter der LoD Grenze an jedes Dreieck angehängt. In der Mitte ist die dritte Variante zu sehen, welche im Gegensatz zu der zweiten Variante die Schürzen einen Subdivisionsschritt vor der Finalmarkierung anbringt. Ganz links ist zum Vergleich die Spalte mit den Ergebnissen des puren Renderings. Die zweite Variante mit Schürzen an finalen Dreiecken produzierte viele unnötige Dreiecke. Dadurch, dass die Schürzen am vorletzten Schritt angebracht wurden, konnten an den Außenkanten zwei und in der Mitte des Dreiecks zwölf Eckpunkte gespart werden. Also waren insgesamt nur noch 14 statt 32 Eckpunkten nötig. In Dreiecke zusammengefügt bedeutet das eine Reduzierung von 24 auf 12 Dreiecke. Da die Schürzen allerdings auf einen kleinen Bereich an den LoD Grenzen beschränkt sind und das Oberflächenmesh dazu kommt, lässt sich die Einsparung um die Hälfte nicht auf die Gesamtzahlen übertragen. Dennoch ist eine Einsparung von 30 Prozent in Stufe neun ereicht worden. Aber auf der anderen Seite werden immer noch unnötige Dreiecke berechnet. Durch die Verlagerung des Schürzencodes vor die letzte Subdivision, bedeutet das, dass bei der letzten Stufe des Transform Feedback Schürzen erzeugt werden, die nicht mehr gebraucht werden, da es näher zum Betrachter keine feinere Unterteilung mehr geben kann. Also werden Schürzen erzeugt die keine Lücken abdecken (Abb. 25). Für den Fall, dass dieses Verfahren für die Erzeugung von Gelände verwendet wird, würde das bedeuten, dass sich die überflüssigen Schürzen zunächst nicht vermeiden lassen. Zusätzliche Codeoptimierungen wären nötig. Aber auch ohne dies lässt es sich einsetzen, wenn das Verfahren selektiv eingesetzt wird. Diese Möglichkeit Dreiecke zu sparen ist dennoch sehr effektiv.

Insgesamt ist die dritte Variante zwar etwas langsamer bei gleichen Dreieckszahlen, kommt aber insgesamt mit weniger Dreiecken aus. Gut zu erkennen ist dies vor allem bei acht und neun Subdivisionsschritten. Bei acht, sinkt die Anzahl der Dreieck von ca. 75000 auf ca. 40000, wodurch für das gleiche Oberflächenmesh ca. 262 statt ca. 141 Frames zur erreicht werden. Bei neun Subdivisionschritten sinkt die Dreieckszahl von ca. 200000 auf ca. 135000, wodurch ca. 80 statt ca. 51 Frames erreicht werden. Bei höheren Subdivisionen schwindet der Vorteil, so dass schon bei zehn Subdivisionen trotz weniger Dreiecken kaum mehr Frames erreicht werden. Alle Kurven, aber vor allem die der rechten Spalten, fallen gegen Ende noch einmal merklich ab. Die abfallenden Werte kommen daher, dass am Ende der Kurven bei unterschiedlichen Blickwinkeln die Framezahlen um zehn bis 20 Frames schwanken. Das liegt daran, dass in der letzten Stufe unnötig Schürzen erzeugt werden, obwohl es keine weitere Unterteilung gibt,

Displacement Mapping fps		Mit verbesserten Schürzen Maximale Subdivisionsschritte				Ohne verbesserte Shürzen Maximale Subdivisionsschritte		
Dreiecke	Bruteforce	Subdiv 11	Subdiv 10	Subdiv 9	Subdiv 8	Subdiv 10	Subdiv 9	Subdiv 8
349525	24	21				26		
337609						38		
322712			29			39		
300000	28	29	35			41		
288605			37			43		
250000	33		41			48		
228098						52		
200000	40	39	48			60	51	
190766			50			62	64	
177473			53				69	
169493		44				68		
157340			58			74	76	
150000	54							
141938		49				79	82	
134624			64	80				
125468			68	90		86	90	
100000	80		78	104		104	110	
90000	88		83	113		113	117	
80000	98	67	89	122				
74384			93	128		126	134	141
70000	112							156
65411			98	136				166
60000	130						160	178
57956		79	106	148				
51200						159	180	
50000	154			161				200
42542						180	202	226
41783		96	124	176				
40000	190							
39248					262			
35000		109	141	196	290	204	227	259
30000	242				333			
29000	252	125	159	217				
25000	290						270	
24000	294	148	184	246	374			
23000	308							
22412		156	194	260	390	242		332
22000	328						288	
21000	350	164	205	273	407			
20000	361				418			348
19000	367	177	220	291	429			
18000	383					268	312	368
17000	424							378
16832		196	242	316	455			
16000	481	203	250	326	468			388
15000	505					292	334	396
14000	515	226	275	353	503	305	350	412
13000	537	239	291	374	522	322	366	426
11000	800	268	323	409	560	350	396	456
10000	880	285	341	433	580	374	422	486
9000	970	308	370	460	612	396	446	510
8000	1120	342	410	504	670	428	480	548
7000	1300	378	450	554	719	472	528	600
6000	1660	422	501	613	790	525	584	666
5000	2070	482	566	685	871	592	664	750
4000	2530	559	652	781	972	674	750	844
3000	3260	662	764	900	1100	771	850	946
2000	4700	854	970	1125	1340	946	1040	1160
1000	7240	1210	1348	1510	1730	1266	1380	1520
512		1560	1705	1880	2100	1480	1610	1770
128		2010	2150	2310	2510	1732	1860	2010
32	12000	2190	2340	2500	2680	2060	2230	2380

Abbildung 22: Tabelle

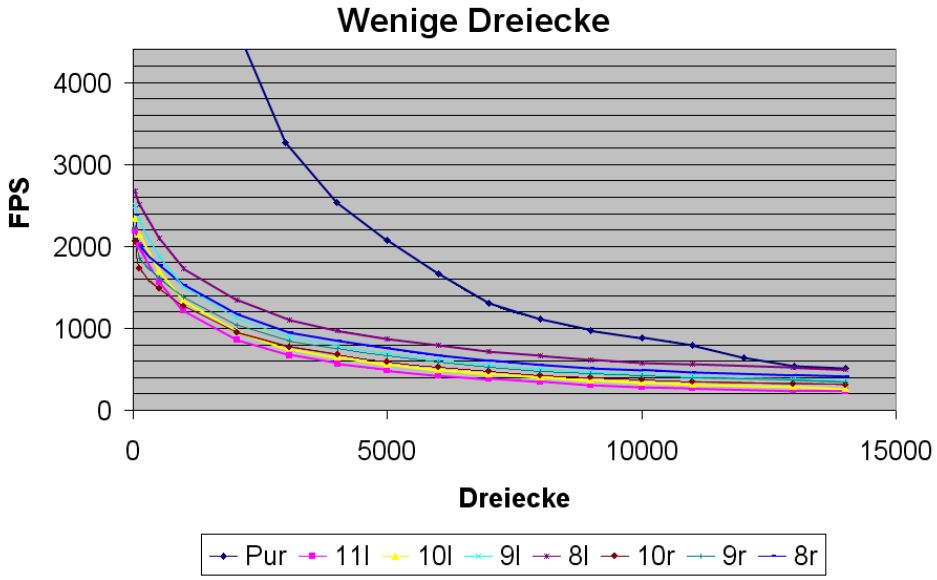

Abbildung 23: Diagramm zur Tabelle Teil 1

die Lücken veruhrsachen könnten. Außerdem arbeitet die Formel, welche die Schürzen auf sinnvolle Bereiche beschränken soll, mit einem fest aufaddierten Wert. Dadurch werden die Schürzenstreifen im Verhaltnis zu der Dreiecksgröße der jeweiligen Stufe breiter. In Kombination mit den kleiner werdenden Dreiecken bedeutet das eine stärker ansteigende Zahl an Schürzen mit zunehmender Subdivision, als ohnehin durch die feinere Unterteilung. Aber ebenso fallen alle Kurven langsamer als das pure Rendering und werden irgendwann schneller als dieses, obwohl bei dem einfachen Rendering kein Displacement Mapping durchgeführt wird.
Die Verbesserung der initialen Implementierung von LoD und Schürzen zu beschleunigten Techniken hat die Performanz um bis zu 130 Prozent gesteigert. Ursprünglich wurden in neun Unterteilungsschritten gerade genug Frames für echtzeitfähiges Rendering erzeugt. Der Umstieg auf Trianglestrips brachte den vergleichsweise geringsten Vorteil, wobei zu beachten ist, dass der Vorteil mit der Anzahl an Dreiecken in geringem Maße steigt. Die Einschränkung der Schürzen auf einen Sinnvollen Bereich dagegen spart, wie in Abb.16 und 26 zu erkennen, deutlich mehr. Außerdem die schon erwähnte Zurücksetzung der Schürzen auf die vorletzte Unterteilungsstufe, welche die Hälfte der Schürzen spart, bzw. insgesamt 30 Prozent. Durch diese Ersparnis ist es jetzt ebenfalls möglich einen weiteren, also zehn Unterteilungsschritte anzuwenden, ohne an das Limit des VBO zu stoßen. Bei gleicher Geschwindigkeit sind daher mehr Oberflächendreiecke nutzbar. Zuvor wurde Stufe 10 nur für die Auslastung des VBO getes-

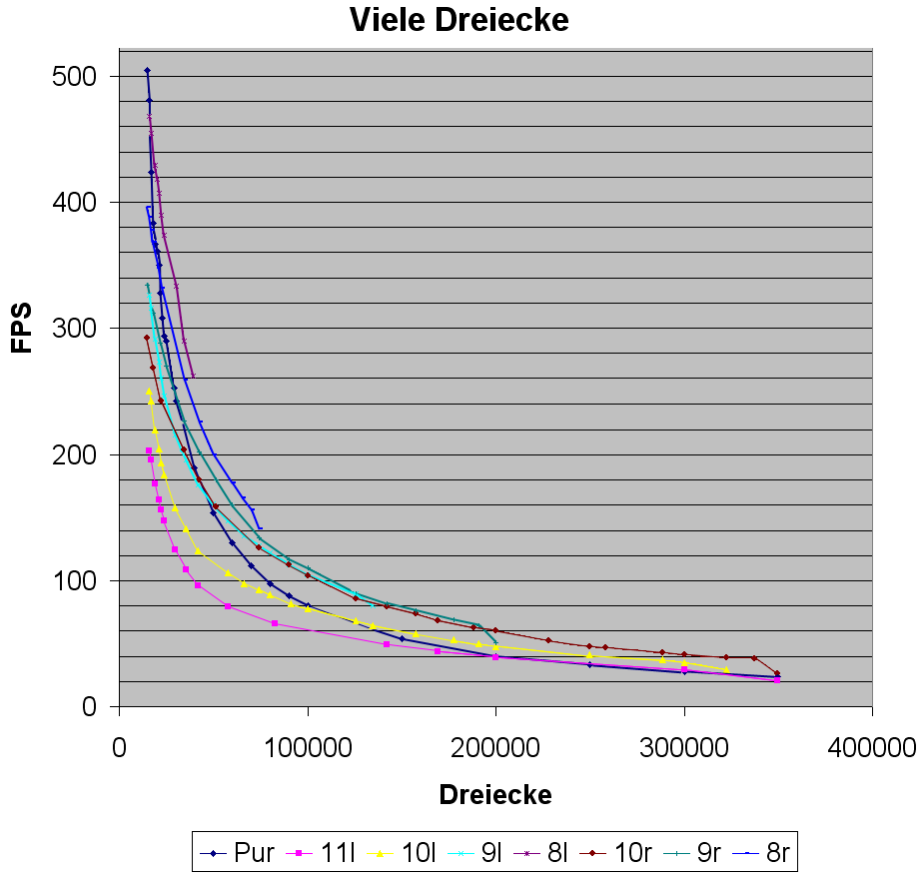

Abbildung 24: Diagramm zur Tabelle Teil 2

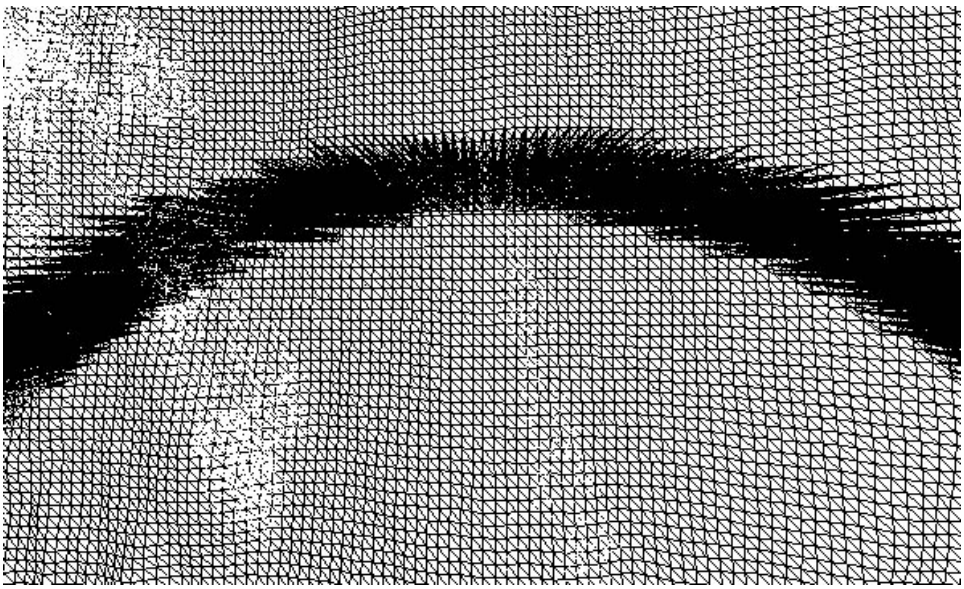

Abbildung 25: Schürzen ohne Folgeunterteilung

tet und Stufe 11 gar nicht. Eine weitere Stufe an Subdivision bedeutet zwar, dass mehr Details dargestellt werden können, aber um nun an das Limit zu stoßen bedarf es elf Subdivisionsschritten. Der Test ergab, dass trotz Beschleunigung das Limit hier nicht mehr echtzeitfähig ausgelastet werden kann, da die Frames zwishen 16 und 26 schwanken. Aber in Hinsicht auf den dennoch erhöhten möglichen Detailgrad mit weniger Subdivisionen, ist das nicht negativ zu beurteilen. Denn die Grenzen für das LoD können auch verschoben werden, um sich den echtzeitfähig darstellbaren Dreiecksszahlen und Stufen anzupassen.
Insgesamt lohnt sich echtes Displacement Mapping mit Hilfe des Geometry Shaders, je nach dem wie oft unterteilt wird, ab ca. 6000-200000 Dreiecken gegenüber der Bruteforce Variante. Die Effizienz hängt in erster Linie von der Rekursionstiefe des Transform Feedback ab, aber nicht zuletzt auch von der Justierung der Formeln, der Komplexität der Grundgeometrie, der verwendeten Soft- und Hardwareumgebung, und einigen anderen Faktoren. Die echtzeitfähige Darstellung von knapp 350.000 Dreiecken aber ist ein Beweis dafür, dass Displacement Mapping durch den Einsatz des Geometry Shaders deutlich effektiver und realisierbar geworden ist. Während echtes Displacement Mapping früher kaum interaktiv war und selbst bei angepassten Anwendungsgebieten seinen Alternativen nicht vorgezogen werden konnte, ist es heute vielleicht nicht für ein modernes PC-Spiel, aber zumindest eingeschränkt nutzbar. Obwohl der hier vorgestellte Code noch weiter optimiert werden kann, zeichnen sich Vorteile gegenüber anderen leistungsfähigen Verfahren ab. Durch die Verwendung des Geometry Sha-

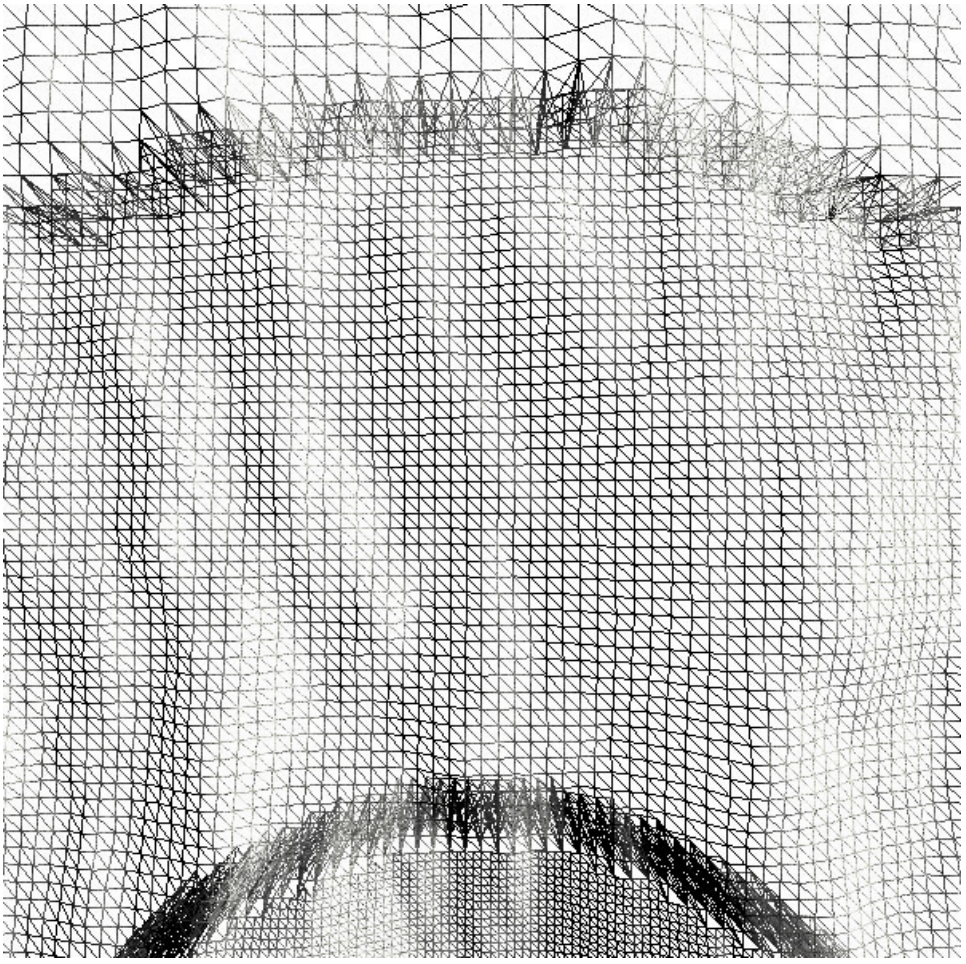

Abbildung 26: Die Shürzen wurden auf sinnvolle Bereiche beschränkt

ders lastet deutlich weniger Arbeit auf dem Fragment Shader, so dass die Bildauflösung fast keine Rolle für die Performanz spielt. Des Weiteren ermöglicht die hohe Adaptivität den Einsatz dieses Verfahrens mit praktisch keiner Vorarbeit an den Texturen und der Geometrie. Außerdem treten weniger Artefakte auf wie z.B. die Treppchen oder Silhouette des Parallax Occlusion Mappings.

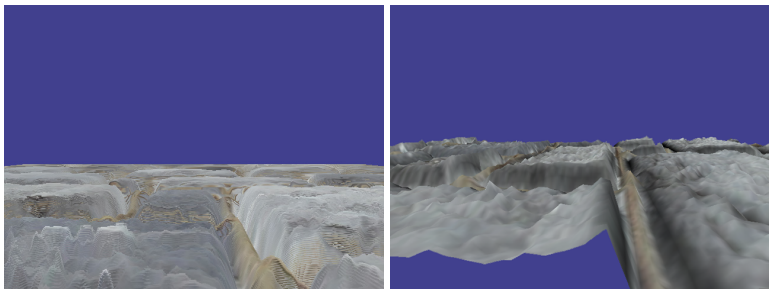

Abbildung 27: Vergleich von Parallax Occlusion Mapping [Zin](links) mit Displacement Mapping(rechts)

5.3 Punktvolumen

Das Volumen Displacement Mapping welches testweise mit Punkten realisiert wurde, benötigt keine Schürzen und produziert somit weniger Primitive. Dadurch ist es zunächst schneller als im Vergleich mit dem zuvor behandelten Verfahren. Aber wird stattdessen die Anzahl der Primitive direkt miteinander verglichen, ist die Punktlösung langsamer. So laufen bei neun Subdivisionen 32330 Punkte mit 108 Fps und 14246 Punkte mit 218 Fps. Bei gleicher Stufe des LoD und der gleichen Stufe an Subdivisionen ist das ca. 40-50 Prozent langsamer als die gleiche Anzahl an Dreiecken, die in diesem Fall etwa 206 und 353 Frames erreichen. Ein weiteres Problem ist die Bleuchtung der Punkte. Selbst wenn sie nur von einer Seite betrachtet werden, muß wenigsten für jede Ebene eine Normalmap und eine Colormap mitgeliefert werden. Somit sind für eine Volumen Displacement Textur pro Eckpunkt 28 Floats nötig, die wenn sie optimal aufgeteilt werden, sieben 2D Texturen beanspruchen. Durch die Aufteilung der Subdivision und des Displacements werden die Höhenunterschiede der vier Ebenen nicht beim LoD berücksichtigt. Die Grenzen bilden sich daher nicht radial wie Kugeln um die Kamera, sondern wie ein Zynlinder. Auf Abb. 28 ist gut zu erkennen, wie im Vordergrund große Lücken im Verhältnis zu den Punkten auftreten, während die Lücken mit zunehmender Entfernung geringer werden, um hinter der nächsten LoD Grenze wieder größer zu werden. Die feste Größe der Punkte lässt es außerdem so aussehen, als ob die Punkte nach hinten größer würden. Ein optischer Effekt der in unserem Gehirn

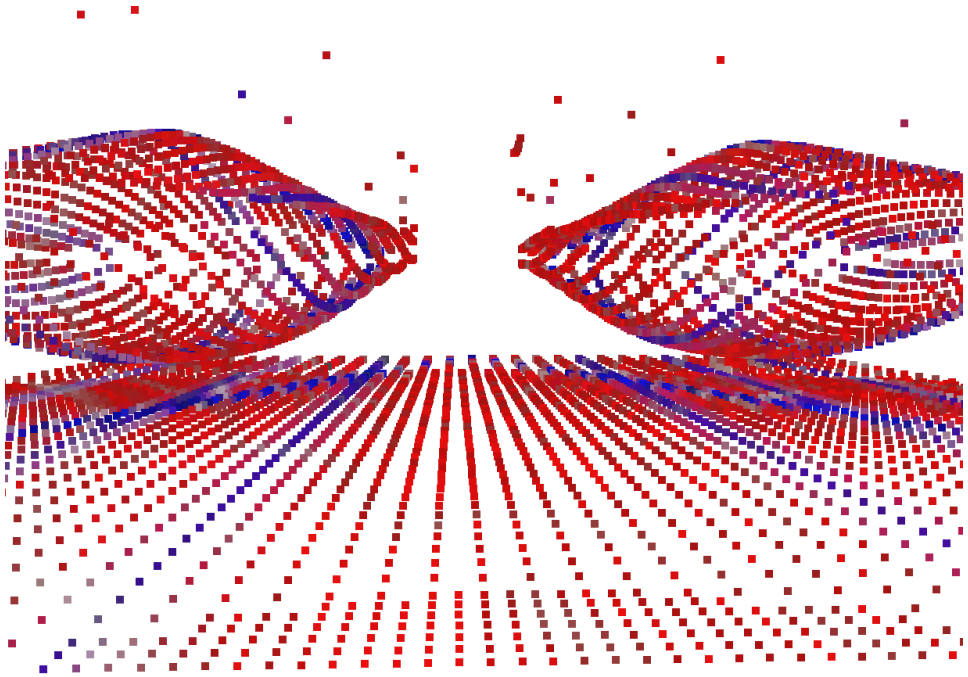

Abbildung 28: Nahaufnahme der Probleme bei Punktvolumen

entsteht und sich nicht abschalten lässt. Die vereinzelten Punkte, die über der Figur schweben, sind Artefakte die an den Rändern der Objekte entstehen können, wenn die Textur mit verlustbehafteten Techniken komprimiert wird. Angesichts dieser Probleme, die zu denen in Kapitel 3.2 und 3.4 erwähnten hinzu kommen, sind Punkte angesichts der starren Größe nicht empfehlendswert. Die Verwendung von 2D Texturen ist ebenfalls nicht zu empfehlen, da diese zu viele Textureinheiten belegen. Sie haben nur den Vorteil viel weniger Speicherplatz als 3D Texturen zu verbrauchen.

6 Ausblick

Um die Performanz der Algorithmen weiter zu steigern bzw. größeren Anforderungen anzupassen ist der nächste Schritt zu versuchen bekannte Techniken wie Viewfrustum Culling oder auch Occlusion Culling zu Implementieren.
Viewfrustum Culling sorgt dafür, dass Geometrie außerhalb des Öffnungswinkels der Kamera nicht mit gerendert werden muss. Dadurch könnten die hier erreichten 350000 Dreiecke mit 36fps immer auf dem Bildschirm dargestellt werden, während drum herum theoretisch noch hunderttausende andere Dreiecke warten. Die Implementierung ist aufgrund der Generierung von Geometrie in den Shadern dabei eine besondere Herausforderung.
Beim Occlusion Culling werden Teile der Geometrie, welche von näher zur Kamera liegender Geometrie verdeckt werden, nicht mit gerendert. Dabei können auch Dreieck im Sichtfeld der Kamera eingespart werden. Auch hierbei muss die im Shader erzeugte Geometrie berücksichtigt werden, weil durch das Displacement eigentlich verdeckte Geometrie doch noch ins Sichtfeld ragen kann. Berge z.B. die durch Displacement Mapping entstanden sind und eine Mauer überragen, deren Grundgeometrie aber von der Mauer verdeckt ist.
Eine Verbesserung der Dreiecksverteilung auf der Fläche könnte durch detailabhängiges Meshing erreicht werden. Dabei werden die Eckpunkte nicht mehr regelmäßig durch Interpolation auf der Grundfläche verteilt, sondern dort, wo nach dem Displacement Mapping Kanten in der Geometrie sein werden. Die Herausforderung ist dabei noch größer als bei Viewfrustum Culling oder Occlusion Culling, da die Kanten der Geometrie unter Berücksichtigung des LoD dafür bekannt sein müssen. Als Speichermodell für die Daten wäre hier das Unstructured Grid eine Möglichkeit. Es kann unterschiedlich große, unterschiedlich ausgerichtete und unterschiedlich geformte Dreiecke speichern. Oder der Algorithmus wird grundsätzlich umgekehrt, indem zunächst die Geometrie vorberechnet und verfeinert wird und erst durch das LoD in der Entfernnung gröber wird. Die verfeinerte Geometrie wäre immer noch durch die Textur entstanden, aber die Berechnungen für die Verfeinerungen würden nicht mehr jedes Frame belasten.
Die nächste große aber wichtige Herausforderung ist die Integration eines Schattenalgorithmus. Normal Mapping bietet keine Selbstverschattung. Die beiden wichtigsten Algorithmen wären Shadow Maps und Shadow Volumes. Während die Shadow Volumes die Generierung neuer Geometrie im Geometry Shader berücksichtigen muß, können die Shadow Maps relativ normal implementiert werden. In den GPU Gems 3[SWK07] in Kapitel elf werden Shadow Volumes auf der GPU beschrieben. Dies vereinfacht die Adaption für das Displacement Mapping.

Ein weiterer Schwachpunkt im bisherigen System ist Displacement Mapping auf eckigen Körpern. Bei scharfen Kanten wie z.B. 90 Grad bei Würfelkanten wird immer noch alles gleichmäßig verschoben, was dazu führt, dass bei dem Versuch aus dem Würfel eine Kugel zu machen, die Würfelform noch gut erkennbar ist. Abb. 29 zeigt einen achteckigen Zylinder von der Seite, dessen Oberfläche mit einer Geländekarte verschoben wurde. Die achteckige Grundstrucktur ist noch deutlich zu erkennen. Dieses Problem

Abbildung 29: Beispiel für sichtbare Grundgeometrie

ist vielleicht lösbar, durch eine differenziertere Skalierung der Verschiebungsintensität. Indem abhängig von der Neigung der Normale einen fester Wert aufaddiert wird. Steht die Normale senkrecht, wird ein hoher Wert aufaddiert, liegt sie dagegen flach in der Ebene des Polygons wird nichts aufaddiert. Ein Multiplikator würde starke unterschiede verursachen, da ohnehin kleine Werte auch relativ klein bleiben und große Werte auch mit einem kleinen Multiplikator relativ viel zulegen. Das Ergebnis würde mehr von der Textur als von der Grundgeometrie abhängen. Trotzdem muss darauf geachtet werden, dass nur ein Dreieck bekannt ist. Daher ist es schwierig, die Position innerhalb der Würfelseite zu bestimmen. Eine andere Möglichkeit wäre es, durch die Verwendung der Subdivision mit Triangle adjacency über die gewonnenen Nachbarschaftsinformationen die Würfelflächen während der Subdivision vorzuwölben. Wodurch die Ursprüngliche Form der Geometrie deutlich schlechter durchdringt oder deutlich einfacher gehalten werden kann.
Des Weiteren können die Schürzen gegen Lücken durch blickpunktabhän-

gige Selektion noch weiter reduziert werden. Dazu müsste jeder an den Geometry Shader übergebene Punkt gegen die Kameraposition getestet werden. Zwischen den beiden am weitesten entfernten Punkten sind keine Dreiecke für Schürzen notwendig. Die Einsparung an Dreiecken für Schürzen bringt nicht ausschließlich mehr Geschwindigkeit für das Rendering. Die gleiche Anzahl an Dreiecken würde etwas langsamer dargestellt, denn der Auswahlprozess der zu rendernden Dreiecke bedeutet einen kleinen Mehraufwand. Allerdings würden mehr Dreiecke tatsächlich Nutzbar.
Um weiter Dreiecke einzusparen kann ohne große Codeänderungen die Formel für die Positionierung der Schürzen verbessert werden. Denn bisher wird ein fester Wert aufaddiert, der je näher sich die Kamera befindet mehr und mehr Schürzen zu viel produziert. Ein adaptiver Wert könnte sich der Dreiecksgröße anpassen und das exponentielle Wachstum verlangsamen.
Eine gewagte Möglichkeit besteht darin für jede Textur bzw. zu erzeugende Geometrie die maximal erforderliche Wiederholung des Transform Feedback festzulegen. Das Problem dabei sind mögliche unterschiedliche Subdivisionstiefen für verschiedene Texturen. Denn ein Transform Feedback ist nicht besonders Variabel während eines Renderpasses und mehrere bedeuten zunächst auch einen mehraufwand. Einfacher wäre es, dem Shader die aktuelle und maximale Stufe des Transform Feedback mitzuteilen. So können bei den Schürzen, die vor einer finalen Subdivision erstellt werden, diejenigen verhindert werden, die keine folgende Unterteilung abdecken würden und somit unnötig wären.

Literatur

[Ban04a] David Banz. High-level shader sprachen. http://www2.inf.fh-rhein-sieg.de/mi/lv/sbmi/ss04/stud/cg.pdf, 2004. Teil 1.

[Ban04b] David Banz. High-level shader sprachen. http://www2.inf.fh-rhein-sieg.de/mi/lv/sbmi/ss04/stud/glsl.pd, 2004. Teil 2.

[Bär06] Robert Bärz. Effizientes rendering von landschaften. Diplomarbeit Uni-Koblenz, FB4, AG Müller, Januar 2006.

[BT04] Z. Brawley and N. Tatarchuk. Parallax occlusion mapping: Self-shadowing, perspective-correct bump mapping using reverse height map tracing. ShaderX3, 2004.

[CBL01] Vittorio Cristini, Jerzy Blawzdziewicz, and Michael Loewenberg. An adaptive mesh algorithm for evolving surfaces: Simulations of drop breakup and coalescence. http://www.math.uci.edu/ cristini/publications/JCompPhys01.pdf, 2001.

[Coo84] Robert L. Cook. shade trees. In Computer Graphics (Processings of Siggraph 84), 1984. 18(3):223-231.

[Gei07] Ryan Geiss. *GPU Gems 3*. Pearson Education, Inc., 2007. Kapitel 1.

[GS] Gpu glsl geometry shader. http://cirl.missouri.edu/gpu/glsl_lessons/glsl_geometry_shader/index.html. Grundlagen.

[Gut04] Stefan Guthe. Compression and visualization of large and animated volume data. Dissertation der Fakultät für Informations- und Kognitionswissenschaften der Eberhard-Karls-Universität Tübingen zur Erlangung des Grades eines Doktors der Naturwissenschaften (Dr. rer. nat.), 2004.

[Hed] Jens Hedrich. Geometry shader tutorials. http://geri.uni-koblenz.de/Projektpraktika/MedVis2/index.php/ Geometry_Shader_Tutorials. Online, Jan 2009.

[Hop96] Huhues Hoppe. *Progressive meshes*. Proceedings of the 23rd annual conference on Computer graphics and interactive techniques, August 1996. p.99-108.

[Kre] Jonathan Kreuzer. Object space normal mapping with skeletal animation tutorial. http://www.3dkingdoms.com/tutorial.htm. Online, Jan 2009.

[KTea01] Tomomichi Kaneko, Toshiyuki Takahei, and et al. Detailed shape representation with parallax mapping. http://vrsj.t.u-tokyo.ac.jp/ic-at/ICAT2003/papers/01205.pdf, 2001. ICAT, 2001.

[LC87] W.E. Lorensen and H.E. Cline. *Marching Cubes: a high resolution 3D surface reconstruction algorithm*. In Computer Graphics, 1987. Vol. 21, No. 4, pp 163-169.

[Mic] Microsoft. Ms directx developer center. http://msdn.microsoft.com/en-us/default.aspx.

[NVI08] NVIDIA. Nvidia opengl extension specifications. http://developer.download.nvidia.com/opengl/specs/GL_EXT_geometry_shader4.txt, 2008.

[Par97] S. Parodat. Marabu - ein werkzeug zur approximation nichtlinearer kennlinien mit radialen basisfunktionen. http://www.pb.izm.fhg.de/mimosys/publish/Status97/3modwerk/modwerk1.pdf, 1997. Online, Jan 2009.

[Ros07] Randi J. Rost. *OpnGL Shading Language Second Edition*. Pearson Education, Inc., September 2007.

[SA08] Mark Segal and Kurt Akeley. The opengl graphics system: A specification. http://www.opengl.org/registry/doc/glspec30.20080811.pdf, 2008. Online, Jan 2009.

[SWK07] Martin Stich, Carsten Wächter, and Alexander Keller. *GPU Gems 3*. Pearson Education, Inc., 2007. Kapitel 11.

[Tri] Damien Triolet. Directx 10 and gpus - behardware. http://www.behardware.com/art/imprimer/631/. Online, Jan 2009.

[vA08] Andreas von Arb. Entwicklung einer beispielapplikation mit hilfe von geometrie-shadern. Studienarbeit Uni-Koblenz, FB4, AG Müller, Mai 2008.

[WTL+04] Xi Wang, Xin Tong, Stephen Lin, Shimin Hu, Baining Guo, and Heung-Yeung Shum. Generalized displacement maps. In Eurographics Symposium on Rendering 2004, 2004. pp. 227-234.

[WWT+03] L. Wang, X. Wang, Xin Tong, Stephen Lin, and et al. View-dependent displacement mapping. acm transactions on graphic. Proceedings of SIGGRAPH 2003 (http://research.microsoft.com/users/lfwang/vdm.pdf), 2003. 22(3):334-339.

[Zin] Jason Zink. A closer look at parallax occlusion mapping. http://www.gamedev.net/columns/hardcore/pom/default.asp. Algorithm Metrics, Online, Jan 2009.