Numerical Geometry and Visualization Group
Interdisziplinäres Zentrum für wissenschaftliches Rechnen
Ruprecht-Karls-Universität Heidelberg

Bachelor-Arbeit

Adaptive Vermaschung und Detailreduzierung von 3D-Punktwolken aus Laserscan-Daten

Christoph Hoppe
christoph.hoppe@iwr.uni-heidelberg.de

22. April 2008

Betreuer:

Prof. Dr. Thomas Ludwig
t.ludwig@computer.org

Dr. Susanne Krömker
kroemker@iwr.uni-heidelberg.de

Ich versichere, dass ich diese Bachelor-Arbeit selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Heidelberg, 22. April 2008

Abstract

3D-laser scanners become more and more popular especially for measuring construction sites, in the field of architecture and for preservation of monuments. As these scanners can only record discrete data sets (*point clouds*), it is neccessary to mesh these sets for getting closed 3D-models and take advantage of 3D-graphics acceleration of modern graphics hardware. The meshing process is a complex issue and in the last years there were lots of algorithms developed to solve this problem.

In this work a continuous Level-of-Detail (LOD) algorithm will be presented, from which a simplified 3D-surface model can be created, which uses only as much triangles as needed. The amount of simplification is determined by the user through an error tolerance similar to the *Hausdorff-Distance*. This algorithm is not a dynamic (view-dependent) LOD-mesh simplification, but an intelligent (non-redundant) approximation of point clouds. It is a simple and extendable meshing algorithm, which is made up of a mixture of some techniques adapted from popular Terrain-LOD algorithms. The algorithm has been implemented in an OpenGL based 3D-point cloud editing tool called *PointMesh*, which is also presented here.

Inhaltsverzeichnis

1 Laserscan-Aufnahmeverfahren....1
1.1 3D-Laser Scanner....1
1.1.1 Grundlegende Funktionsweise des CP 3200 3
2 Approximative Vermaschung von Punktwolken....5
2.1 Das Programm PointMesh....8
2.1.1 Reguläre Vermaschung mit PointMesh....9
2.2 Notwendigkeit eines intelligenteren Vermaschungsverfahrens....11
3 Level-of-Detail Algorithmen....13
3.1 Top-Down und Bottom-Up....13
3.2 Reguläre Gitter und TINs....13
3.3 Der SOAR-Algorithmus von Lindstrom et al....14
3.4 Der ROAM Algorithmus....14
4 Der implementierte Algorithmus....16
4.1 Generierung eines Höhenfelds....16
4.1.1 Erfassen der Höhenwerte....17
4.2 Triangulierung über rekursive Unterteilung (recursive subdivision)....18
4.2.1 4-8 Mesh....18
4.2.2 Nicht-restriktive Verfeinerung....18
4.2.3 Bestimmen des Mittelpunkts der Hypotenuse....20
4.2.4 Adaptive Verfeinerung....21
4.2.5 Fehlerberechnung – Die Hausdorff-Distanz....22
4.2.5.1 Die Implementierung der Fehlerberechung....23
4.3 „Cracks“ und „T-Junctions“....24
4.3.1 Aufbau eines Vertex Sets....28
4.3.1.1 Split Points und Forbidden Points....28
4.4 Exportieren der Modelle im VRML-Format....30
5 Texturierung der Modelle mit Blender....32
6 Vergleich zwischen regulär triangulierten und 4-8 Meshes....33
6.1 Die Grenzen des Verfahrens....37
7 Diskussion und Verbesserungsvorschläge....38
8 Anhang....40
9 Glossar....41
10 Literaturverzeichnis....45

1 Laserscan-Aufnahmeverfahren

Digitale Messverfahren können nur eine diskrete Anzahl von Messwerten unserer Umwelt aufnehmen. Aus dem Grund, dass im Laufe der Zeit die Messinstrumente immer präziser werden, wird auch die Menge der aufgenommenen Messdaten sehr viel größer. Um aus solch großen Datenmengen überhaupt noch Erkenntnisse zu erlangen, ist es mehr denn je nötig, diese Daten am Rechner zu visualisieren.

Der Prozess der 3D-Visualisierung von großen Datenmengen beansprucht eine enorme Rechenleistung. Um solche Datenmengen auch auf nicht so leistungsfähigen Rechnern, wie z.B. Laptops, darstellen zu können, ist es deshalb nötig die Datenmenge zu reduzieren, ohne dabei grundlegende Details zu verlieren.

Gerade im Bereich des Laserscannings ist dies von großer Bedeutung, da hier große redundante Datenmengen anfallen, aber die gescannten Objekte auch mobil, z.B. direkt nach dem Scan, betrachtet werden sollen.

1.1 3D-Laser Scanner

3D-Laserscanner stellen mittlerweile in vielen Bereichen das Grundwerkzeug zur Vermessung und Analyse von realen Objekten und Umgebungen dar. Solche Scanner ermöglichen über die Aussendung eines Laserlichtstrahls Form- und Strukturdaten, sowie auch Daten über die äußere Erscheinung (Farbe etc.) von nahezu beliebigen Objekten aufzunehmen.

Die hauptsächlichen Einsatzgebiete von Laserscannern sind:

- Vermessung (Landschaften, Gebäude, etc.)
- Denkmalschutz für historische Gebäude und Artefakte
- Architektur (Konstruktion / Rekonstruktion)
- Aufnahme von Objekten und Personen für 3D-Filme und PC-Spiele
- Spurensicherung / Forensik / Unfallforschung

[WIKI_SCAN]

Die auf dem Markt erhältlichen Scanner arbeiten, je nach Anwendungsgebiet, mit ganz unterschiedlichen Technologien:

- **Impulslaufzeitmessung:** Misst die Zeitdifferenz zwischen Aussendung und Empfang eines kurzen Laserlichtpulses und bestimmt daraus die Entfernung zum Objekt.
- **Phasenlaufzeitmessung:** Hier wird ein kontinuierlicher Laserstrahl ausgesandt. Die Amplitude des ausgesandten Laserstrahls wird mit mehreren sinusförmigen Wellen unterschiedlicher Wellenlänge moduliert. Der entstehende zeitliche Abstand des empfangenen Signals gegenüber dem gesendeten Signal ist eine Folge der Entfernung zum

Objekt. Bei gleichzeitiger Betrachtung der Phasenlage des gesendeten und des empfangenen Signals ergibt sich eine Phasendifferenz, die die Bestimmung des Objektabstandes erlaubt. [WIKI_SCAN2]

- **Lichtschnitttriangulation:** Bei dieser Methode wird eine Lichtschicht (generiert z.B. durch einen über eine Zylinderlinse aufgeweiteten Laserstrahl) auf die Oberfläche projiziert und von einer Kamera unter einem Triangulationswinkel abgebildet. Bei durch ein geeignetes Kalibrierverfahren gegebener Aufnahme- und Projektionsgeometrie kann damit ein dreidimensionales Profil auf dem Objekt berechnet werden. [PHOTOGRAM]

Jede Technologie hat seine eigenen Vor- und Nachteile, Grenzen und Kosten. Trotz der fortschreitenden Technik kann man mit solchen Geräten aber nicht alle Objekte vermessen, weil optische Laserscanner-Technologien aufgrund ihres Systemaufbaus und der physikalischen Gesetze an Grenzen gebunden sind. So haben sie beispielsweise grundsätzlich Schwierigkeiten, glänzende, schwarze, reflektierende oder transparente Objekte aufzunehmen. Ein Laserstrahl wird von solchen Oberflächen unter Umständen in die falsche Richtung reflektiert, absorbiert oder zu schwach reflektiert und kann somit nicht mehr von dem im Scanner eingebauten Sensor erfasst werden.

Zudem weisen 3D-Laserscanner eine bestimmte Reichweite auf, weshalb man sie auch in die Kategorien *Close-Range* (< 1 m), *Mid-Range* (30 m – 150 m) und *High-Range* (140 m – 6 km) einteilt. Moderne Mid-Range-Scannersysteme erreichen für die ihre maximale Entfernungsmessung (z.B. für Callidus CP 3200: 32 m [CAL]) eine Punktgenauigkeit von bis zu 1 mm. Die Genauigkeit der modellierten Fläche ist außerdem von der kleinsten Schrittweite des Drehkopfes und der Entfernung des Scanners zum Objekt abhängig. Hier erreichen moderne Mid-Range-Scanner eine Genauigkeit von ± 1 mm auf bis zu 30 m Enfernung. [LAS_ACC]

Im Rahmen dieser Arbeit wurde mit den Messdaten eines 3D-Mid-Range-Laserscanners der Firma Callidus gearbeitet, die hauptsächlich zur Gebäudevermessung gewonnen wurden.

Abbildung 1: Der Callidus CP 3200 Mid-Range-Laserscanner. [CAL]

1.1.1 Grundlegende Funktionsweise des CP 3200

Der Scanner arbeitet nach dem Prinzip der Impulslaufzeitmessung. Dabei wird ein sehr kurzer Laserpuls von dem Scanner ausgesandt, trifft auf ein Hindernis, wird von diesem diffus reflektiert und kommt so schließlich wieder zum Scanner zurück, wo er von einem Lasersensor erfasst wird. Die Zeit zwischen dem Aussenden und dem Empfang des Lichtpulses (Lichtlaufzeit) ist proportional zur Distanz zwischen Scanner und Objekt, weshalb hieraus die tatsächliche Distanz zum Objekt berechnet werden kann. Über diese Distanz und den Winkel des Lichtstrahls, können dann für den aufgenommenen Punkt 3D-Raumkoordinaten berechnet und anschließend auf einem Rechner gesichert werden. Zusätzlich wird auch die Reflektanz der Oberfläche über das reflektierte Laserlichts ermittelt, aus der sich Rückschlüsse über die Materialeigenschaften ziehen lassen.
Während der Aufnahme rotiert der im Messkopf integrierte Laserscanner um 360° in der horizontalen Ebene (einstellbare Schrittweiten: 0.0625°, 0.125°, 0.25°, 0.5°, 1.0°). Durch einen rotierenden Drehspiegel wird der Laserstrahl in Form von vertikalen Streifen ausgesandt und kann bis zu 140° seiner Umgebung in der vertikalen Ebene erfassen (Schrittweiten: 0.25°, 0.5°, 1.0°). Hier ist der verschattete Bereich am Boden abhängig von der Installationshöhe des Scanners (siehe Abb. 3). Dadurch, dass als Lichtquelle ein Infrarot-Laser benutzt wird, sind diese Messungen nicht vom Umgebungslicht abhängig. [CAL]

Durch dieses Verfahren wird schließlich eine große Anzahl an Messpunkten aufgenommen, die die Oberfläche des aufgenommenen Objekts approximieren. Eine solche Menge von Punkten in einem Raum (in diesem Fall dreidimensional) wird im Allgemeinen auch als *Punktwolke* bezeichnet. Die Genauigkeit einer solchen *Punktwolke* hängt von der Gesamtzahl der aufgenommenen Messpunkte ab und somit auch von der Anzahl der durchgeführten Scans. Da ein solcher Scanner seine Umgebung radial aufnimmt, sinkt die Punktdichte mit zunehmender Entfernung zum Scanner. Aufgrund dieser Beschränkungen werden zumeist mehrere Aufnahmen von einem Objekt aus unterschiedlichen Scannerpositionen durchgeführt, die im Nachhinein über einen in der Scannersoftware integrierten *Matching*-Algorithmus zu einem Gesamtendatensatz zusammengefügt werden.

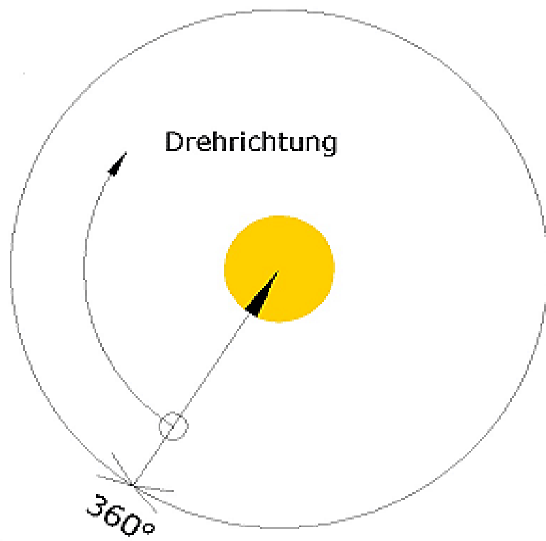


Abbildung 2: Drehwinkel und Drehrichtung des Scanners um dessen vertikale Achse. [CAL]

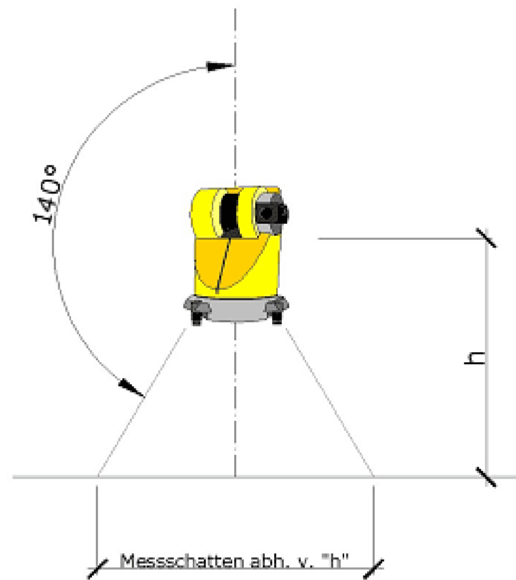


Abbildung 3: Drehwinkel des Scanners um die horizontale Achse. [CAL]

Über eine in das System integrierte CCD-Kamera (mit variabler Brennweite) ist es möglich, sowohl Panorama- als auch Detailaufnahmen der Umgebung aufzunehmen, um das gescannte Objekt entsprechend zu dokumentieren. Diese Aufnahmen bieten außerdem eine Vorlage, um anschließend automatisch den aufgenommenen Punktdaten *RGB-Echtfarben* zuweisen zu können. [CAL]

Zudem besitzt der Scanner einen integrierten Kompass und man kann so im Nachhinein allen Messpunkten eine Himmelsrichtung zuweisen, in die eine Normale der zugehörigen Oberfläche zeigen würde. Diese sogenannten *Kompassfarben* oder besser Richtungsfarben, werden aus den XYZ Komponenten der "Punktnormalen" berechnet: Die Punktnormale eines Punktes wird unter Verwendung einiger benachbarter Punkte (Bildung einer Fläche) bestimmt. Die Berechnung der Farbanteile für Rot, Grün und Blau jeweils im Bereich von 0..255 (RGB-Farben) erfolgt nach der Vorschrift:

Rot = 160 + 90 * X
Grün = 160 – 90 * Y
Blau = 160 + 90 * Z

Da die Werte von x, y und z immer im Bereich von -1.0 bis 1.0 liegen, können die Farbkomponenten Werte von 70 bis 250 annehmen und es ist gewährleistet, dass bei schwarzem Hintergrund (RGB = 0, 0, 0) die Punkte immer sichtbar sind. Bei der Transformation einzelner Scans in ein übergeordnetes (globales) Koordinatensystem werden die lokalen Punktnormalen eines jeden Scans mit transformiert. Somit erhalten später gleiche Flächen gleiche Farben. [CAL] Zur Veranschaulichung siehe auch Abb. 4 links.

2 Approximative Vermaschung von Punktwolken

Dem IWR der Universität Heidelberg liegen seit geraumer Zeit verschiedenste 3D-Laserscanner-Aufnahmen vor. So wurden mit diesen Laserscannern z.B. zahlreiche Aufnahmen des Kirchenrests des „Klosters Lorsch", dem „Heidelberger Fass" und der „Alten Brücke" in Heidelberg vorgenommen. Diese Daten liegen im ASCII-Format (*plain text* und *VRML*) vor. Aufgrund der Fülle von Messpunkten sind diese Daten sehr umfangreich (ca. 200-300 MByte) und lassen sich demnach auch auf aktuellen Rechnern nur bedingt flüssig darstellen und bearbeiten. Die gespeicherten Punktwolken liegen sowohl in *RGB-Echtfarben* (aufgenommen über die Digitalkamera des Laserscanners), als auch in *Kompassfarben* vor.

Zunächst werden diese Punktwolken mit einem Gitter (*Mesh*) approximiert. Diese automatisch generierten *Meshes* bilden ein dichtes reguläres Gitter aus Quadraten, die die Oberfläche approximieren. Im Rahmen meiner HiWi-Tätigkeit und eines Softwarepraktikums am IWR habe ich bereits das Programm *PointMesh* entwickelt, das diese Approximation durchführt. [SWP_MESH] Eine Einführung in den Aufbau und die Funktionsweise des Programms soll im folgenden Kapitel gegeben werden.

Trotz der ansehnlichen Resultate ist es generell nicht sehr effizient, eine mehr oder weniger ebene Oberfläche (Wand) mit tausenden von kleinen Quadraten bzw. Dreiecken darzustellen. Prinzipiell würde hier ein Rechteck genügen und die Struktur der Oberfläche könnte man, wie in der 3D-Grafik allgemein üblich, über eine Textur darstellen.
Durch die vorgenommene Approximation reduziert sich die Punktanzahl um den Faktor von zwei bis zwanzig, je nach der vom Nutzer getroffenen Wahl der *Maschenweite* des Gitters. Um diesen Detailverlust zu kompensieren, werden automatisch Texturen aus den *RGB-Echtfarben* der Punktwolken gewonnen und auf das Gitter gelegt. [SWP_TEX]

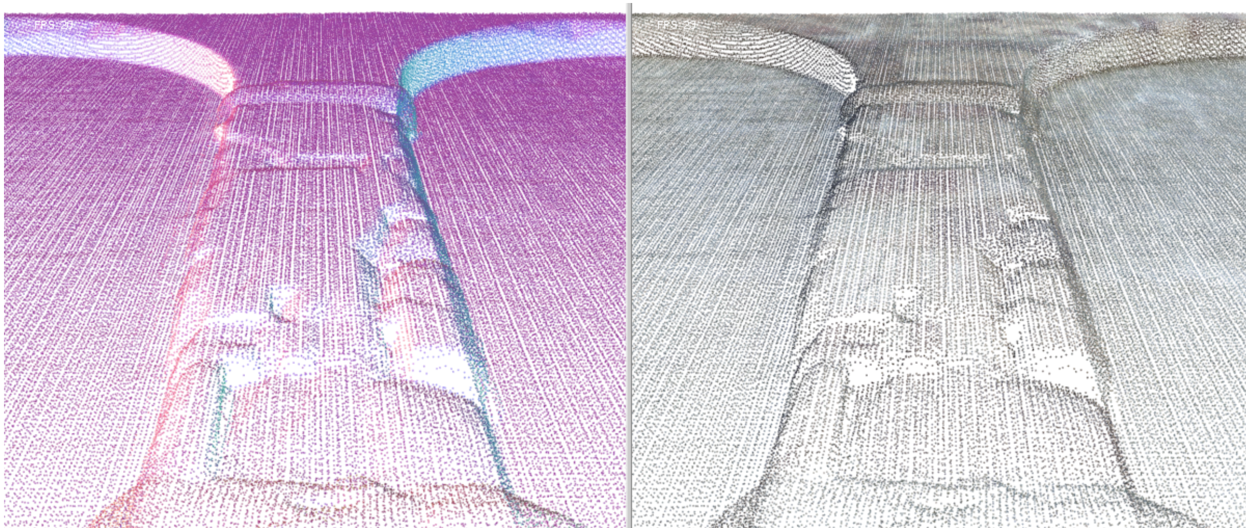


Abbildung 4: Punktwolkenausschnitt eines 3D-Laserscans - links: Kompassfarben; rechts: RGB-Echtfarben

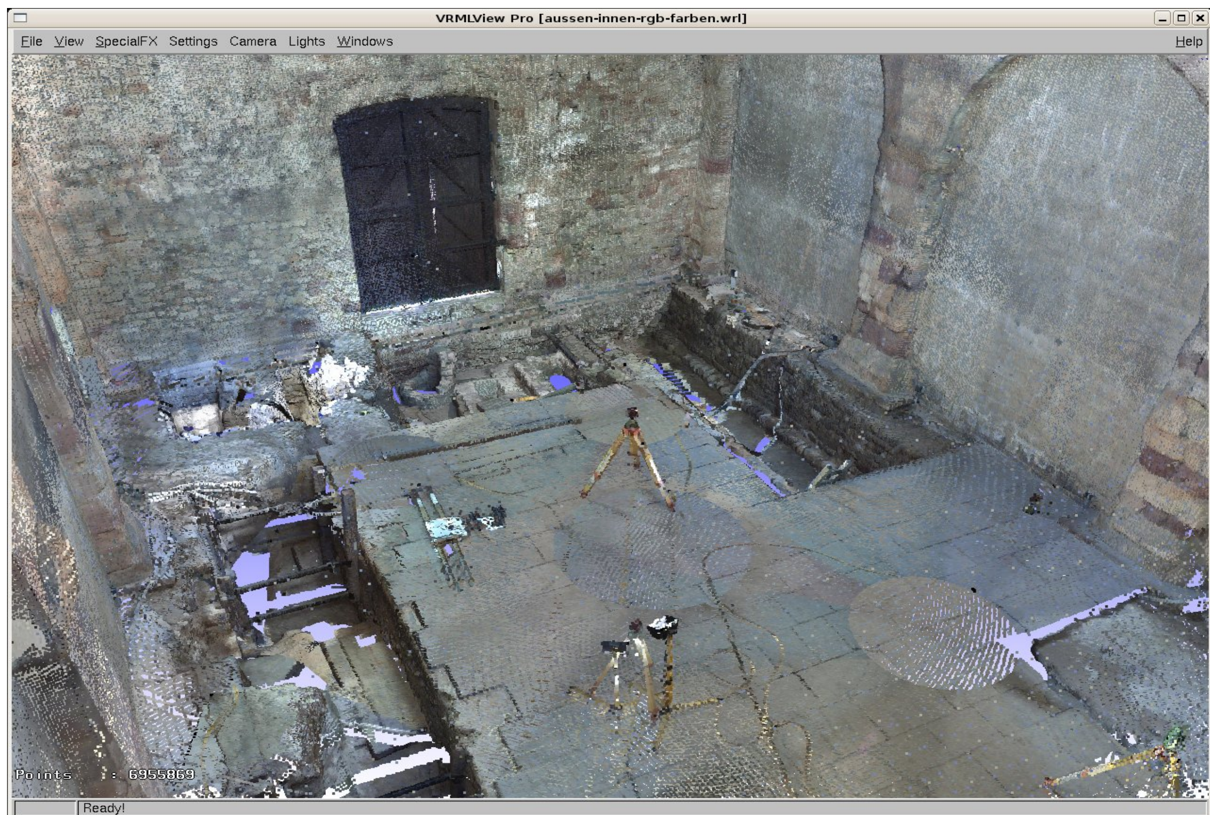


Abbildung 5: 3D-Innenaufnahme des Kirchenrests des Klosters Lorsch. Auf dem Bild sind zahlreiche Kreise zu erkennen, die die unterschiedlichen Scannerstandorte erkennen lassen. Die Punktdichte ist hier durch Verschattungen unterhalb des Scanners enorm verringert, da hier mindestens ein Scan gar keine Werte liefern konnte.

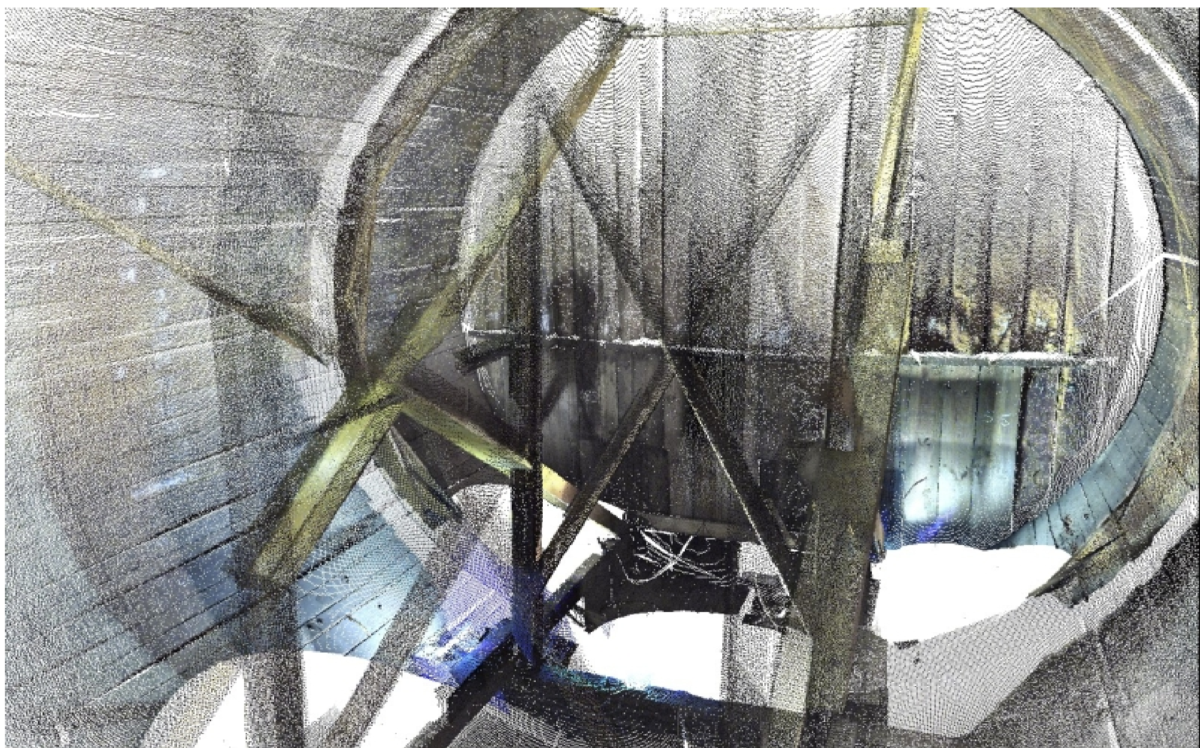


Abbildung 6: 3D-Innenaufnahme des Heidelberger Fasses. Hier kann man ebenfalls anhand der weißen Kreise erkennen, wo der Scanner zum Zeitpunkt der Aufnahmen stand.

Abbildung 7: 3D-Laserscan der Südtürme der „Alten Brücke“ in Heidelberg. Man sieht hier deutlich, dass die Punktdichte zu den Spitzen hin abnimmt, und dort viele Lücken auftreten, da der Scanner während den Aufnahmen nur auf dem Boden positioniert werden konnte.

2.1 Das Programm PointMesh

Mit dem Programm *PointMesh* ist ein Werkzeug entstanden, das eine einfache und intuitive Bearbeitung von Punktwolkendaten ermöglicht. Seit August 2006 ist dieses Programm kontinuierlich weiterentwickelt worden. Zunächst ging es nur darum, Punktwolkendaten im VRML-Format einlesen, zuschneiden und schließlich diese kleineren Ausschnitte wieder abspeichern zu können. Dieses Programm war ursprünglich nur ein kommandozeilen-basiertes Linux-Programm. Es stellte sich heraus, dass es sehr schwierig war, die entsprechenden Koordinaten zum Zuschneiden herauszufinden, ohne zugleich ein Bild vor Augen zu haben. Es musste also um eine grafische Darstellung und Steuerung erweitert werden, die in den darauf folgenden Monaten entwickelt wurde. Als Basis wurde die OpenGL-Programmierschnittstelle und die FreeGLUT-Bibliothek gewählt, damit sich das Programm möglichst einfach auf andere Systeme portieren lässt. Der Code für die grafische Darstellung wurde entsprechend in separate Dateien gldisplay.cc / gldisplay.h ausgelagert. Alle Steuerungsfunktionen über Tastatur und Maus wurden in die Dateien ui.cc / ui.h integriert.

Über eine *bounding box* - Auswahl und *Point-Picking*-Funktionen konnten die Punktwolken sehr viel intuitiver zugeschnitten und abgespeichert werden. Da aber die FreeGLUT-Bibliothek nur ein begrenztes Maß an GUI-Steuerelementen anbietet, wurde Ende 2006 eine eigene GUI mit texturierten Schnellstartknöpfen entworfen, die in die Codedateien ui.cc, textures.cc und buttons.cc integriert wurde.

Durch die gestiegene Komplexität und Funktionalität wurde im Laufe dieses Projekts für das User-Interface die GUI-Library GLUI in das Programm eingebettet. GLUI bietet den Vorteil sehr kompakt zu sein und trotzdem viele der aus *MS Windows* bekannten Steuerelemente und Funktionen für OpenGL und GLUT zu integrieren. Ein weiterer Vorteil ist, dass das Programm dabei trotzdem

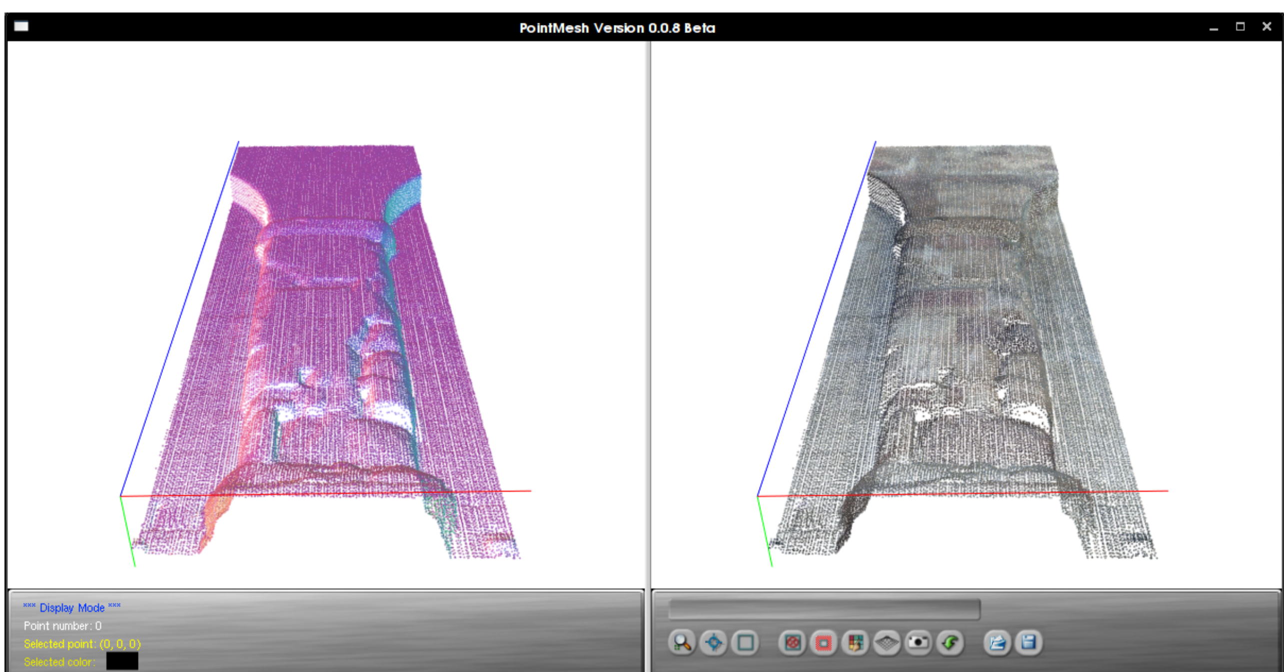


Abbildung 8: Die Benutzeroberfläche des Programms PointMesh.
links: Punktwolkenausschnitt in Kompassfarben; rechts: Punktwolkenausschnitt in RGB-Echtfarben

plattformunabhängig bleibt, weil hier nur auf standardmäßige OpenGL und GLUT-Funktionen zurückgegriffen wird. Aus diesem Grund konnte das Programm unter wenig Aufwand erfolgreich für Macintosh auf *MacOS X* portiert werden.

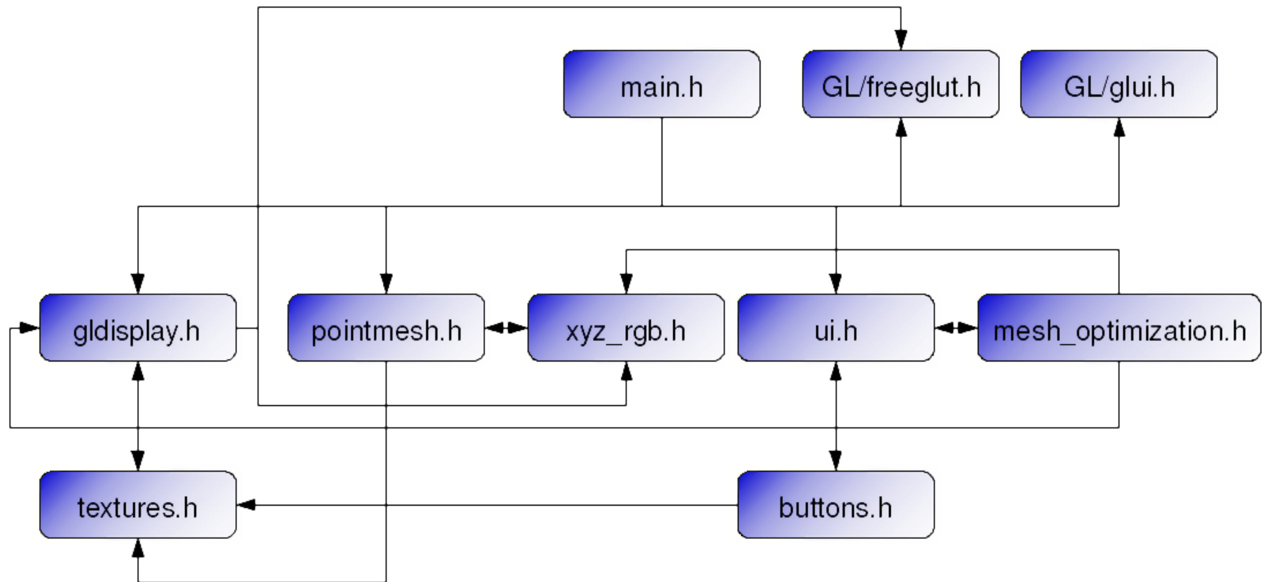


Abbildung 9: Struktureller Aufbau des Programmcodes von PointMesh. Der hier vorgestellte Algorithmus wurde in den neu eingeführten Dateien mesh_optimization.cc / mesh_optimization.h untergebracht.

2.1.1 Reguläre Vermaschung mit PointMesh

Die Punktwolken aus 3D-Laserscans, die uns als Ausgangsdatensätze dienen, besitzen zwei charakteristische Merkmale:

1. Die Messpunkte sind nicht gleichmäßig verteilt. Ihre Dichte nimmt mit zunehmender Entfernung zum Scannerstandort ab und hängt von der Anzahl der vorgenommenen Scans ab.
2. Die Punktwolken besitzen an vielen Stellen „*Löcher*“, die z.B. durch Verschattungen, Oberflächenreflexion und Fenster entstehen können.

Um ein Bauwerk mit einem 3D-Laserscanner aufzunehmen, werden aus diesen Gründen zahlreiche Scans aus unterschiedlichen Positionen durchgeführt, um Löcher und Verschattungen zu vermeiden und insgesamt detailliertere Aufnahmen zu erhalten. Die Messpunkte eines einzelnen Scans sind geordnet, da der Messkopf die Datenwerte fächerförmig von unten nach oben aufnimmt und sich schrittweise um seine vertikale Achse dreht. Ein anschließender Prozess des *Matchings* führt dazu, dass die Punkte aller Scans vereinigt werden und somit im endgültigen Datensatz völlig ungeordnet vorliegen.

Trotz der relativ flüssigen Darstellung von Punktwolken kommen selbst leistungsfähige Rechner und Grafikkarten ins Stocken, wenn man sehr große Punktwolken (> 3 Mio. Punkte) einliest, wie wir sie z.B. von dem Kirchenrest des Klosters Lorsch besitzen.

Aus diesem Grunde wurde an einem regulären Vermaschungsalgorithmus gearbeitet, der im Rahmen eines Softwarepraktikums entwickelt wurde. Der Ansatz, diese Punktwolken zu vermaschen, besteht darin, diese Oberfläche mit triangulierten Quadraten einer fest gewählten Größe zu approximieren [SWP_MESH].

Dieser Algorithmus wählt eine orthographische Projektion eines Teilausschnitts der Punktwolke, dessen approximierte Oberfläche insgesamt in eine bestimmte Richtung weist. Schrittweise detektiert der Algorithmus alle Punkte innerhalb einer quadratischen Region (genannt *„Patch“*) von vorgegebener Größe (genannt *„Patchsize“* oder *„Maschenweite“*) und aligniert über die Berechnung der mittleren Höhe der Punkte ein trianguliertes Quadrat aus zwei Dreiecken daran. Anschließend wird diese Region auf der Punktwolke um deren Größe so lange nach oben verschoben, bis keine Punkte mehr gefunden werden können. Tritt dieser Fall ein, wird die Region wieder auf den unteren Rand des Punktwolkenausschnitts gesetzt und um deren eigene Größe nach rechts verschoben. Insgesamt läuft der Algorithmus so von der linken unteren Ecke der Punktwolke zur rechten oberen Ecke der Punktwolke streifenweise über die Punktwolkenoberfläche und approximiert sie in jedem Schritt durch zwei Dreiecke. Die Größe der Dreiecke und damit auch deren Gesamtzahl ergibt sich aus der zuvor gewählten Größe der Region (Maschenweite / Gitterweite). Zudem wird aus den jeweils in der Region liegenden Punkten der Mittelwert ihrer *RGB-Echtfarben* berechnet und den beiden Dreiecken zugewiesen.

Die so entstandenen Dreiecks-Streifen werden in einem zweiten Durchlauf miteinander verbunden, um schließlich eine geschlossene Oberfläche zu erhalten. Nach der Erstellung eines solchen Meshs kann dieses im VRML-Format gesichert werden, um es mit anderen Programmen weiter nutzen zu können.

Gegen Ende 2007 wurde im Rahmen eines Softwarepraktikums [SWP_TEX] auf Basis dieses Algorithmus das Programm dahingehend erweitert, dass sich aus Ausschnitten von Punktwolken auch Bilddateien im *BMP*-Format abspeichern lassen, um diese als Texturen für die vermaschten 3D-Modelle weiterverwenden zu können. Das funktioniert folgendermaßen:

Zunächst führt man eine möglichst enge Vermaschung der Region über den zuvor beschriebenen Algorithmus durch. Dabei ist die *Maschenweite* so eng zu wählen, dass im Mittel nicht mehr als 3-4 Punkte innerhalb der jeweiligen Region liegen, um so ein möglichst genaues Resultat zu erhalten. Da während des Durchlaufs in jedem Schritt der Mittelwert der Farbe dieser Punkte abgespeichert wird, liegen schließlich auch die Farbwerte aller Quadrate streifenweise sortiert in einem Array vor. Um sie im *Bitmap*-Format (BMP) abspeichern zu können, werden die Farben in der für BMP-Bilder typischen Reihenfolge abgespeichert (BGR = blau, grün, rot) und zeilenweise von unten nach oben.

Zusätzlich wurde ein automatisches *Mapping* dieser Texturen auf die gewonnenen 3D-Modelle implementiert, so dass sich diese auch texturiert im VRML-Format abspeichern lassen (genaueres dazu: siehe [SWP_TEX]).

2.2 Notwendigkeit eines intelligenteren Vermaschungsverfahrens

Die im vorherigen Absatz besprochene Art der Vermaschung ist in der Praxis nicht besonders effizient. Der Reduktionsfaktor der Punktanzahl ist hier abhängig von der Anzahl der in der Vermaschungsregion (*Patch*) eingeschlossenen Punkte. Wieviele Messpunkte während der Vermaschung in der Vermaschungsregion eingeschlossen sind hängt größtenteils von der gewählten *Maschenweite* und der allgemeinen Dichte der Punktwolke, somit also von der Zahl der durchgeführten Scans und der Entfernung des Scanners zur gescannten Oberfläche ab.

Der Reduktionsfaktor der Punktanzahl (RF) durch reguläre Vermaschung lässt sich näherungsweise durch folgende Formel beschreiben:

$$RF = \frac{\#\text{Gitterzeilen} \cdot \#\text{Gitterspalten}}{\#\text{Regionen} \cdot \sum^{\#\text{Regionen}} \frac{\#\text{Punkte pro Region}}{\#\text{Regionen}}}$$

wobei #Regionen = (#Dreiecke / 2).

Beispielsweise ergibt sich bei im Mittel drei eingeschlossenen Punkten pro Region und einem Gitterausmaß von 5x4 Quadraten (entspricht 20 Vertices und 24 Dreiecken) eine Reduktion der gesamten Punktanzahl um ungefähr 44%.

Trotzdem entstehen durch die reguläre Vermaschung bei hauptsächlich ebenen Regionen immer noch viel zu viele unnötige Dreiecke, welche bei großflächigen hochdetaillierten Modellen auch auf schnellen Rechnern zu einer langsamen Darstellung führen können. Die Gründe dafür liegen in der Art und Weise, wie Geometrien im Rechner verarbeitet werden.

Zur Darstellung von triangulierten Oberflächen wird die *3D-Grafikpipeline* genutzt, die darauf optimiert ist, die Dreiecke zu transformieren und mit Texturen zu belegen. Die *Grafikpipeline* besteht aus zwei Hauptstufen: Geometrieverarbeitung und Rasterung (siehe Glossar).

Für gewöhnlich ist der Rasterungsaufwand relativ begrenzt, weil die darzustellende Oberfläche wenig Tiefen-Komplexität besitzt. Im schlechtesten Fall bedeckt das komplette Modell den *Viewport* und die Anzahl der zu füllenden Pixel ist nur ein wenig größer als die der Pixel im *Framebuffer*. Heutige Grafikkarten besitzen eine genügend große Füllrate, um den ganzen *Framebuffer* mehr als ausreichend schnell mit Texturen zu belegen, auch bei Verwendung von komplexen Funktionen, wie anisotropisches Mip-Mapping oder von hochdetaillierten Texturen. [SVDLOD]

Im Gegensatz dazu erweist sich die Geometrieverarbeitung als das eigentliche „*bottleneck*“. Auch high-end Plattformen können in Echtzeit nur einen Bruchteil von mehreren hundert Millionen Dreiecken berechnen, weshalb es notwendig ist, sich mit einer effizienten Darstellung zu beschäftigen.

Eine relativ ebene Oberfläche, wie wir sie oft bei Gebäudeaufnahmen von 3D-Laserscannern erhalten, kann mit einem stark vereinfachten Mesh approximiert werden, ohne dass der Detailgrad der Darstellung sichtlich darunter leidet. Damit ein solches vereinfachtes Mesh räumlich kontinuierlich bleibt, sollte es frei von Brüchen (*Cracks*) und T-Kreuzungen (*T-Junctions*) sein. [SVDLOD]

Im Rahmen dieser Bachelorarbeit soll ein kontinuierliches Level-Of-Detail-Verfahren (LOD) vorgestellt werden, mit dem aus Punktwolken ein 3D-Oberflächenmodell erzeugt werden kann, das möglichst wenige Dreiecke benutzt. Es handelt sich hierbei nicht um eine dynamische (sichtweitenabhängige) LOD-Darstellung, sondern um eine intelligente (nicht-redundante) Approximation von Punktwolkendaten über Höhenfelder.

In dieser Arbeit wird ein Grundgerüst zur Durchführung einer intelligenten Vermaschung vorgestellt. Da die Implementierung eines solchen Verfahrens relativ aufwendig ist, wurde sich aus Zeitgründen nur auf das Verfahren selbst beschränkt. So musste beispielsweise auf die automatische Texturierung und eine Glättung der Modelle verzichtet werden. Um einen besseren Eindruck von den gewonnenen Ergebnissen zu bekommen, wurde stattdessen in Kapitel 5 die Texturierung manuell mithilfe der Open Source 3D-Modelling und Rendering Software *Blender* vorgenommen. [BLENDER]

3 Level-of-Detail Algorithmen

Mit dem Problem einer intelligenten Vermaschung und Detailreduzierung haben sich schon sehr viele Forschungs- und Arbeitsgruppen auseinandergesetzt und sehr leistungsfähige Verfahren entwickelt.

Da eine Punktwolke aus einem 3D-Laserscan nur eine Oberfläche beschreibt, kann man diese als eine Art Landschaft interpretieren, auf die sich *„Terrain“*-LOD-Algorithmen anwenden lassen, weshalb wir uns in dieser Arbeit nur auf solche Verfahren beschränken. Im Folgenden sollen einige populäre Verfahren und deren Unterschiede betrachtet werden.

3.1 Top-Down und Bottom-Up

Im Vergleich zu generellen LOD-Techniken lassen sich Terrain-LOD-Algorithmen nach ihrem Ansatz zur Vereinfachung in zwei Typen gliedern: *top-down* und *bottom-up*. Bei einem *top-down* Algorithmus beginnt man für gewöhnlich mit zwei oder vier Dreiecken für die gesamte Region und fügt anschließend progressiv neue Dreiecke hinzu, bis die gewünschte Auflösung erreicht ist. Diese Techniken werden auch Subdivisions- oder Verfeinerungsmethoden genannt. Im Gegensatz dazu, startet ein *bottom-up*-Algorithmus mit dem höchstaufgelösten Mesh und entfernt iterativ Vertices aus der Triangulierung, bis der gewünschte Grad an Detailreduzierung erreicht ist. Diese Techniken nennt man auch Dezimierungs- oder Vereinfachungsmethoden. *Bottom-up*-Ansätze finden meist die minimale Anzahl an nötigen Dreiecken für eine vorgegebene Genauigkeit, stellen jedoch als Voraussetzung, dass das komplette Modell gleich zu Beginn schon verfügbar ist und benötigen deshalb mehr Speicher und Rechenleistung.

3.2 Reguläre Gitter und TINs

Ein weiteres wichtiges Unterscheidungsmerkmal zwischen Terrain-LOD-Algorithmen, ist die Art der verwendeten Struktur, die das Terrain repräsentiert. Die zwei häufigsten benutzen Ansätze sind regulär vermaschte Höhenfelder (reguläre Gitter) und triangulierte irreguläre Gitter (*triangulated irregular networks = TIN*). Reguläre Gitter benutzen ein Array von Höhenwerten für x- und y-Werte, die alle einen gleichförmigen Abstand besitzen.

Generell können TINs eine Oberfläche genauer approximieren, benötigen also weniger Dreiecke als andere Verfahren. Zum Beispiel können hiermit großflächige ebene Regionen mit einem groben Gitter erfasst werden, während die unebenen Regionen durch ein hochaufgelöstes Gitter dargestellt werden. Im Vergleich dazu sind reguläre Gitter sehr viel ungenauer als TINs, da sie die gleiche Auflösung für die gesamte Region besitzen, ganz gleich, ob die Gebiete eben oder uneben sind. Jedoch bieten reguläre Gitter den Vorteil, dass sie leichter zu speichern und zu manipulieren sind und man dadurch einen schnellen Zugriff auf beliebige Höhenwerte hat, weshalb wir uns in dieser Arbeit auf reguläre Gitter beschränken.

3.3 Der SOAR-Algorithmus von Lindstrom et al.

Einer der ersten Echtzeit LOD-Algorithmen für Höhenfelder war die frühe Arbeit von Lindstrom et al. Dieser Algorithmus benutzt ein reguläres Gitter und eine vom Benutzer definierte *Screen-Space* Fehlertoleranz, um den Grad der Vereinfachung zu steuern [SOAR]. Der Algorithmus ist konzeptionell vom Typ *bottom-up*, beginnt also mit dem kompletten Modell bei seiner höchsten Auflösung und vereinfacht die Dreiecke progressiv, bis die gewünschte Genauigkeit erreicht ist. Jedoch wird in der Praxis das Mesh in rechteckige Blöcke unterteilt und zunächst eine *top-down* Vereinfachung auf diese Blocks angewandt, gefolgt von einer Vertex-Reduzierung innerhalb dieser Blocks. Die *bottom-up* Rekursion entsteht, wenn ein Vertex (Knotenpunkt) von einem aktiven Zustand in einen inaktiven (und umgekehrt) überführt wird. An dieser Stelle müssen die Dreiecke vereinigt oder geteilt (*merge* oder *split*) werden. *Cracks* zwischen aneinanderliegenden Knoten (siehe auch Kapitel 4.2.4) werden über einen Binärbaum entfernt, der die Abhängigkeiten zwischen den Vertices erhält. Um auch *Cracks* zwischen den einzelnen Blöcken zu vermeiden, teilen sich alle benachbarten Blöcke in ihren Grenzregionen die gleichen Vertices. [LOD_3D]

Das Vereinfachungsschema von Lindstrom et al. schlägt eine Vertex-Entfernung vor, bei der jeweils ein Paar von Dreiecken zu einem einzelnen reduziert wird. Dies beinhaltet, dass das gemeinsame Vertex zwischen den Dreiecken identifiziert und entfernt werden muss. Die Entscheidung, wann diese *merge*-Operation durchgeführt werden muss, wird über die Größe des *Screen-Space* Fehlers zwischen den beiden Oberflächen getroffen. In diesem Fall ist das die vertikale Distanz zwischen dem zu entfernenden Vertex und dem Mittelpunkt der neu zu entstehenden Kante. Anschließend wird dieses Segment in den *Screen-Space* projiziert, um den maximal erreichbaren Fehler berechnen zu können. Wenn dieser Fehler kleiner als ein vorgegebener Pixel-Schwellwert liegt, wird die Dreiecks-Vereinfachung weiter durchgeführt. [LOD_3D]

Mittlerweile wurde dieser Algorithmus modifiziert und es entstand daraus die *SOAR-Terrain-Engine* (siehe [SOAR]).

3.4 Der ROAM Algorithmus

Ein Jahr nach Lindstrom et al.s Algorithmus publiziert wurde, wurde von Duchaineu et al. aus den Los Alamos und Lawrence Livermore National Laboratories der ROAM-Algorithmus veröffentlicht [ROAM]. Dieser Algorithmus ist vor allem unter Spieleentwicklern sehr populär geworden. ROAM (*real-time optimally adapting meshes*) benutzt einen inkrementellen prioritätsbasierten Ansatz, mit einer zugrunde liegenden Binärbaumstruktur. Über diese Struktur wird ein kontinuierliches Mesh erstellt, indem eine Reihe von *split-* und *merge*-Operationen auf Dreieckspaare angewandt wird, die sich ihre Hypotenusen teilen, sogenannte *„Diamanten“*.

Der ROAM-Algorithmus benutzt zwei Prioritäts-Warteschlangen, um über die merge- und split-Operationen zu entscheiden. Die erste Warteschlange enthält eine nach Priorität geordnete Liste von *split*-Operationen zwischen den Dreiecken, so dass eine Verfeinerung des Terrains erzielt wird, wenn man in jedem Schritt das Dreieck mit der höchsten Priorität unterteilt. Die zweite Warteschlange enthält eine nach Priorität geordnete Liste von *merge*-Operationen der Dreiecke, um

das Mesh zu vereinfachen. Dies ermöglicht ROAM den Vorteil von *Frame-Kohärenz* zu nutzen, d.h. dass er auf vorherige Frames der Triangulierung zurückgreifen kann und dort inkrementell Dreiecke hinzufügen oder entfernen kann. Die Priorität der *split-* und *merge*-Operationen beider Warteschlangen wird durch verschiedene Fehler-Metriken bestimmt und kann bei Interesse in dem zugehörigen Paper [ROAM] nachgelesen werden.

4 Der implementierte Algorithmus

4.1 Generierung eines Höhenfelds

Da die *Maschenweite* bei dem oben beschriebenen Verfahren zur regulären Triangulierung einheitlich gewählt ist, entsteht dadurch der grundlegende Nachteil, dass in allen Bereichen des Gitters gleich viele Dreiecke sind, auch dort, wo sie nicht vonnöten sind.

Einen Vorteil des Verfahrens, den man sich aber zunutze machen kann, besteht darin, dass man durch dieses Verfahren sehr leicht ein geordnetes Höhenfeld erhält, indem man nacheinander die Höhenwerte der Vertices des Gitters in einem Array abspeichert.
In einem solchen Höhenfeld werden nur Höhenwerte gespeichert, das heißt, die zweidimensionalen Positionen der Werte ergeben sich implizit. Deshalb eignet sich beispielsweise ein dynamisches *float*-Array zur Speicherung dieses Felds, welches im Vergleich zur Speicherung aller drei Raumkoordinaten nur etwa ein Drittel des Speicherbedarfs benötigt. Die einzigen noch notwendigen Parameter sind, der fest gewählte Abstand der Punkte und die Dimensionen (Länge und Breite) des Felds.

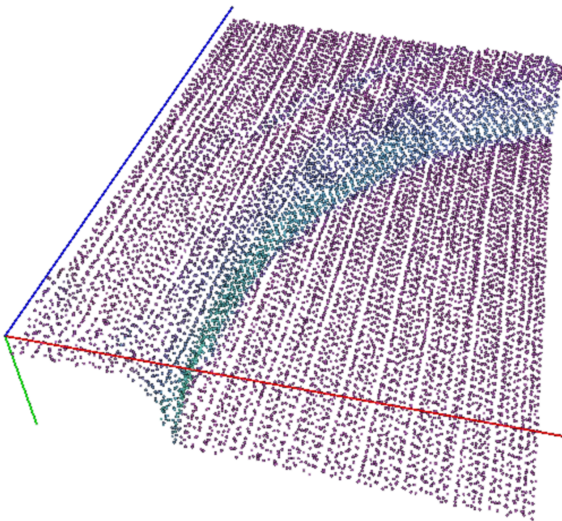


Abbildung 10: Ausgangsdatensatz - Punktwolke eines Arkadenbogens in Kompass-Farben, gewonnen durch 3D-Laserscans.

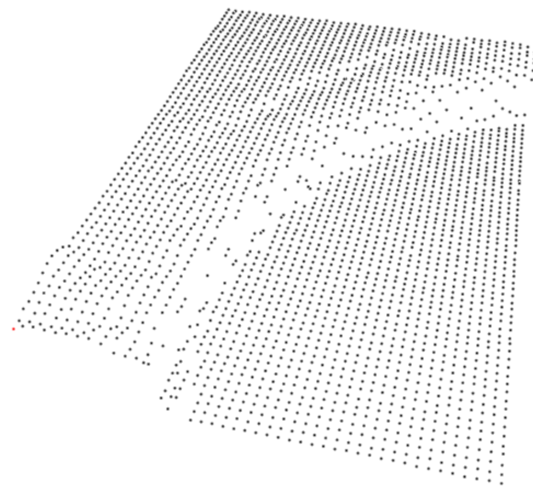


Abbildung 11: Gewonnenes Höhenfeld aus dem Datensatz, bei einer Maschenweite (=Gitterpunktabstand) von 4 cm.

4.1.1 Erfassen der Höhenwerte

Um an einer beliebigen Position innerhalb des Höhenfelds, auch zwischen den Gitterpunkten, Zugriff auf die Höhenwerte zu erhalten, ist es notwendig, den jeweils nächstgelegenen Höhenwert des Gitters zu erfassen. Man könnte an dieser Stelle auch mehrere Nachbarpunkte einbeziehen und zwischen ihnen interpolieren, was exaktere Resultate liefern würde. Aufgrund des erhöhten Rechenaufwands wurde zunächst aber darauf verzichtet.
Für die Erfassung des nächstgelegenen Höhenwerts werden die 3D-Raumkoordinaten des Punktes in einen Index des Höhenfelds umgerechnet.

Die Funktion

get_index(float xpos, float zpos)

löst dieses Problem folgendermaßen:

Die Daten der Punktwolken werden in unserem Fall so geliefert, dass die Länge des Gitters der z-Koordinate entspricht, die Breite der x-Koordinate, entsprechend liegen die Höhenwerte in der positiven y-Richtung vor.
Da die Höhenwerte in unserem Fall spaltenweise von unten nach oben (positive z-Richtung) gespeichert sind und wir die gesamte Zeilen- und Spaltenanzahl des Höhenfelds vorliegen haben, ergibt sich ein folgendes Umrechnungsschema:

```
off_x = (unsigned int)(((xpos-gmin_x)/((gmax_x-gmin_x)/gridwidth))-0.5);
off_z = (unsigned int)(((zpos-gmin_z)/((gmax_z-gmin_z)/gridlength))-0.5);
```

- xpos und zpos definieren die Eingabewerte der Funktion in 3D-Koordinaten.
- gmax_z, gmin_z, gmax_x, gmin_x sind die Minimal- und Maximalwerte des Gitters in 3D-Koordinaten.
- die Variablen gridlength und gridwidth geben die Anzahl der Höhenwerte des Gitters in z- und x-Richtung an.
- gridsize liefert den bei der Erzeugung des Höhenfelds festgelegten Abstand zwischen zwei benachbarten Höhenwerten.

Über *off_x* wird zunächst die Gitterposition in x-Richtung bestimmt, entsprechend über *off_z* die Gitterposition in z-Richtung. Um zu vermeiden, dass am oberen und rechten Rand des Gitters ungültige Werte benutzt werden, werden die beide Offsets bei der Konvertierung in vorzeichenlose Ganzzahlen (unsigned int) abgerundet (-0.5). Sonst könnte es vorkommen, dass am oberen Rand des Höhenfelds der nächste Wert benutzt werden würde, der wiederum am unteren Rand des Felds liegen würde.
Da das Gitter spaltenweise von unten nach oben und von links nach rechts vorliegt, berechnet sich nun die entsprechende Indexpositon über:

```
index = ( off_x * gridlength ) + off_z;
```

So kann also aus zwei beliebigen (x, z)-Raumkoordinaten der nächstgelegenene Index abgerufen werden, womit wir auch Zugriff auf den nächstliegenden Höhenwert erhalten.
Die Funktion

float get_height(float xpos, float zpos)

erledigt genau dies, indem sie auf die Funktion **get_index(...)** zurückgreift, den Index von dieser erhält und im Höhenfeld den entsprechenden Wert abruft.

Die Bestimmung der Höhenwerte zu beliebigen x-z-Koordinaten ist Grundvoraussetzung, um eine Triangulierung mit Approximation durchführen zu können, da das Mesh ohne Zugriff auf die Höhenwerte komplett flach bliebe.

4.2 Triangulierung über rekursive Unterteilung (recursive subdivision)

4.2.1 4-8 Mesh

Ein 4-8 Mesh ist, mathematisch beschrieben, eine reguläre trianguläre Unterteilung einer Teilfläche des Körpers R^2. Es ist ein semi-reguläres Gitter, bei dem die Vertices (Knotenpunkte) eine maximale Wertigkeit von vier oder acht haben, d.h. dass die Knoten im Inneren jeweils vier oder acht und am Rand entweder drei oder fünf Nachbarn besitzen.
Ein solches Mesh eignet sich besonders gut für eine apdaptive Triangulierung, da es sich, im Gegensatz zu anderen Methoden (*face split / vertex split*), durch einfache Bisektion verfeinern lässt

und dabei in jedem Schritt immer wieder ein feineres gültiges 4-8 Mesh entsteht. Dies hat zur Folge, dass bei restriktiver Triangulierung die Topologie des Meshs erhalten bleibt. Das vereinfacht die adaptive Verfeinerung enorm: Wenn ein Mesh nicht gleichförmig verfeinert wird, bleibt das verfeinerte Mesh trotzdem noch gültig.
Ein weiterer Vorteil ist, dass sich die Anzahl der Dreiecke zur nächst höheren Stufe nur um den Faktor zwei erhöht, womit eine feinere Detailabstufung möglich ist, als bei einigen anderen bekannten Verfahren.

4.2.2 Nicht-restriktive Verfeinerung

Um ein 4-8 Mesh zu erzeugen, wird zu Beginn ein grobes Mesh aus zwei Dreiecken aufgebaut, indem die vier Eckpunkte des Höhenfelds identifiziert und zu jeweils zwei Dreiecken verbunden werden. Dieses initiale Mesh bildet also ein einzelnes trianguliertes Viereck (*Quad*).
In den folgenden Schritten wird von jedem Dreieck der Mittelpunkt der Hypotenuse bestimmt und genau an dieser Stelle das Dreieck in zwei kleinere Dreiecke halbiert. Dabei hat man zwischen acht möglichen Ausrichtungen der Dreiecke zu unterscheiden, die bei dieser Art der Unterteilung entstehen können, wie es in Abbildung 13 dargestellt wird.

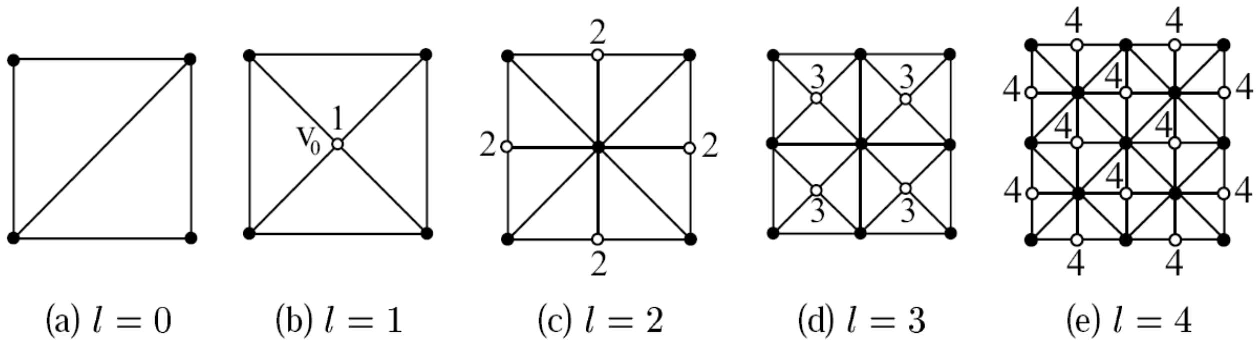


Abbildung 12: Schematische Ansicht der Erzeugung und gleichmäßigen Verfeinerung eines 4-8 Meshs in den unterschiedlichen Stufen (l). Die Ziffern kennzeichnen die in den jeweiligen Stufen eingeführten Vertices. [MESH_OPT]

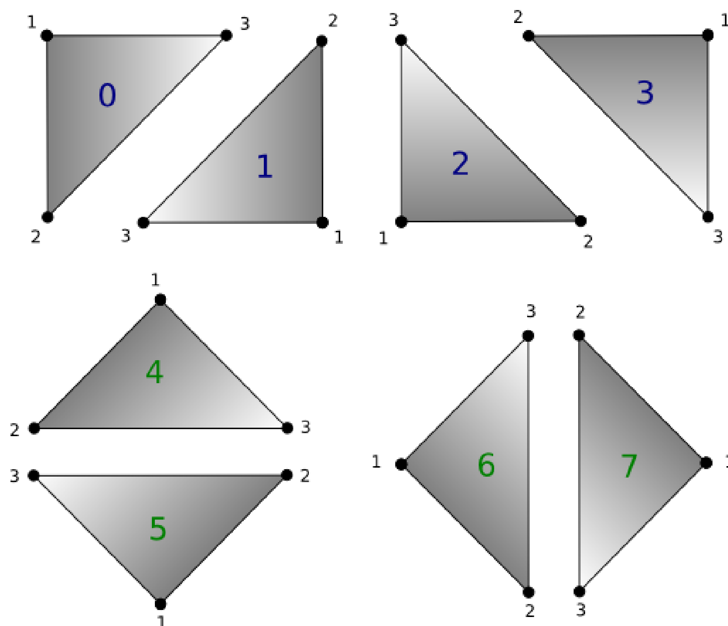


Abbildung 13: Die möglichen Ausrichtungen der Dreiecke im Raum bei rekursiver Unterteilung über den Mittelpunkt der Hypotenuse.

Um eine Ordnung im Mesh zu erhalten, wurde für die Reihenfolge der Eckpunkte (Orientierung der Dreiecke) ein gegenläufiger Uhrzeigersinn gewählt (*Counter-Clockwise (CCW)*), damit auf alle Dreiecke in der gleichen Weise zugegriffen werden kann. Für die graphische Darstellung der Dreiecke ist dadurch die äußere Normale, die die Vorderseite *(Front Face)* definiert, dem Betrachter zugewandt.

4.2.3 Bestimmen des Mittelpunkts der Hypotenuse

Für die Ausrichtungen 0 bis 3 lässt sich die Position des Mittelpunkts der Hypotenuse bestimmen über:

hypo.x = point2.x + (point3.x – point2.x) / 2;
hypo.z = point2.z + (point1.z – point2.z) / 2;

Entsprechend gilt für die Ausrichtungen 4 und 5:

hypo.x = point1.x;
hypo.z = point2.z;

Und schließlich für die Ausrichtungen 6 und 7:

hypo.x = point2.x;
hypo.z = point1.z;

Die y-Koordinate wird aus dem nächstgelegenen Punkt im Höhenfeld bestimmt, das, wie in Abschnitt 4.1.1 beschrieben, die Funktion **get_height(...)** erledigt:

hypo.y = get_height(hypo.x, hypo.z);

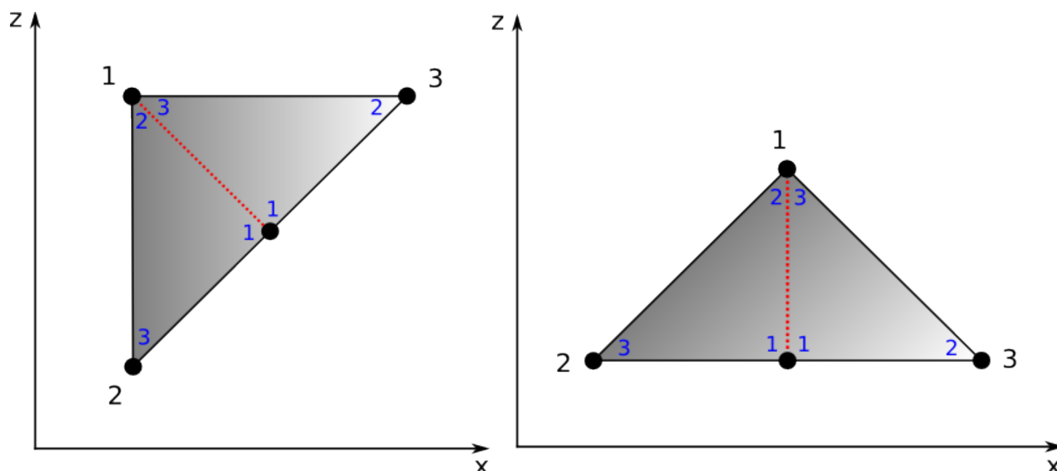


Abbildung 14: Bestimmung des Mittelpunkts der Hypotenuse für die Ausrichtungen 0 und 4. Die rot gestrichelte Linie deutet an, an welcher Stelle nach der Unterteilung eine neue Kante entsteht. Die blauen Ziffern geben die Reihenfolge der Vertices der neu entstandenen Dreiecke an.

Nach der Bestimmung des Mittelpunkts der Hypotenuse wird das Ursprungs-Dreieck geteilt, d.h. an Stelle des alten Dreiecks werden zwei neue Dreiecke ausgegeben.

Die Funktion zur Unterteilung

subdivide(triangle3, triangle3, char, unsigned int level, unsigned int maxlevel)

bearbeitet je zwei Dreiecke in einem Schritt, woraus folgt, dass die Funktion für jeweils zwei Eingabe-Dreiecke vier neue Dreiecke ausgibt.

Ein sogenanntes *draw*-Flag kennzeichnet für jedes Dreieck, ob es zum endgültigen Mesh gehört und ob es somit auf dem Bildschirm gezeichnet und später auch abgespeichert werden soll. In jedem Unterteilungsschritt wird deshalb das *draw*-Flag der beiden Eingabe-Dreiecke auf *false* gesetzt und der vier neuen Ausgabe-Dreiecke auf *true*. Der Vorteil dieser Methode ist, dass auch im Nachhinein auf Dreiecke zugegriffen werden kann, die nicht auf dem Bildschirm gezeichnet werden. Dies ist für die Vermeidung von *Cracks* und *T-Junctions* (siehe Abschnitt 4.3) von Bedeutung. Zu jedem Dreieck wird zudem aufsteigend die Rekursionsstufe gespeichert, beginnend bei Stufe 0 für die beiden initialen Dreiecke. Zur Veranschaulichlichung des ganzen Prozesses, siehe Abb. 29-31 im Anhang (Seite 39).

Die **subdivide(...)**-Funktion ist eine rekursive Funktion, d.h. sie ruft sich in jedem Verfeinerungsschritt so lange wieder selbst auf, bis ein zuvor festgelegter Verfeinerungsgrad erreicht ist. Diesen gewünschten Verfeinerungsgrad (Anzahl der Rekursionsschritte) kann man der Funktion über das Argument *maxlevel* übergeben. Bei der adaptiven Verfeinerung ermittelt die Hilfsfunktion **calc_error(...)** anhand eines übergebenen Schwellwerts automatisch, ob in einem bestimmten Bereich eine weitere Verfeinerung notwendig ist, wie es im folgenden Abschnitt geschildert wird.

4.2.4 Adaptive Verfeinerung

Durch die nicht-restriktive Verfeinerung wird das Mesh an allen Stellen gleichmäßig verfeinert. Das Ziel soll aber sein, das Mesh nur an solchen Stellen weiter zu verfeinern, an denen die von den Dreiecken eingeschlossenen Höhenwerte sich in ihrer Höhendifferenz zum zuvor approximierten Dreieck merklich unterscheiden, d.h. nur innerhalb solcher Regionen, in denen das Höhenfeld noch Unebenheiten aufweist.
Um dies zu erreichen, muss man eine effektive Begrenzung für die weitere Unterteilung eines Dreiecks einrichten. Das heißt, es muss ermittelt werden, wie stark sich die aktuelle Triangulierung von einer optimalen Triangulierung der Höhenpunkte unterscheidet. In dieser optimalen Triangulierung würde jeder Höhenpunkt mit seinen vier unmittelbaren Nachbarpunkten verbunden werden, wäre also im Inneren des Felds ein Eckpunkt für vier Dreiecke.
Es gibt verschiedene Möglichkeiten diese Abweichung (im Folgenden auch *Fehler* genannt) zu berechnen. Eine sehr beliebte Variante in diesem Bereich ist die *Hausdorff*-Distanz.

4.2.5 Fehlerberechnung – Die Hausdorff-Distanz

Die Hausdorff-Distanz ist ein bekanntes Konzept aus der Topologie und findet Anwendung in der Bildverarbeitung, Oberflächenmodellierung und einer Vielzahl weiterer Anwendungsgebiete. Sie ist nur auf Punktmengen definiert. Da man aber eine Oberfläche als eine kontinuierliche Punktmenge beschreiben kann, kann man sie auch hierauf anwenden.
Zu zwei gegebenen Punktmengen A und B, ist die Hausdorff-Distanz H(A,B) das Maximum der minimalen Abstände zwischen den Punkten beider Mengen. Mit anderen Worten: Wir suchen für jeden Punkt der Menge A den nächstgelegenen Punkt in der Menge B und umgekehrt. Anschließend berechnen wir die Distanzen zwischen all diesen Punktpaaren und nehmen davon das Maximum.

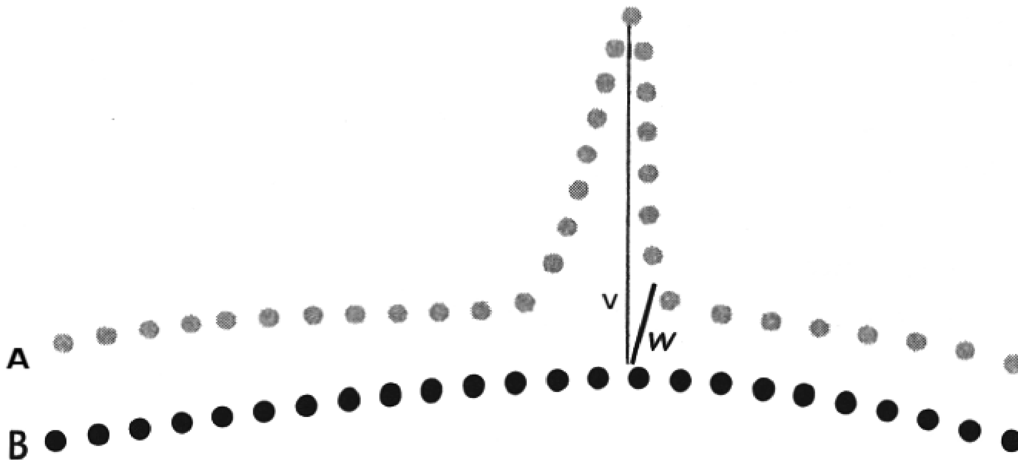


Abbildung 15: Die Hausdorff-Distanz zwischen zwei Oberflächen. Die einseitigen Hausdorff-Distanzen sind h(A,B) = ||v|| und h(B,A) = ||w||. Die beidseitige Hausdorff-Distanz ist H(A,B) = max(||v||,||w||)=||v||. [LOD_3D]

Algebraisch ausgedrückt:

$$H(A,B)=max(h(A,B),h(B,A))$$

wobei

$$h(A,B)=max_{(a\in A)}min_{(b\in B)}\|a-b\|$$

h(A, B) bestimmt für jeden Punkt in A den nächstgelegenen Punkt in B und nimmt davon das Maximum. Diese Funktion, auch einseitige Hausdorff-Distanz genannt, ist nicht symmetrisch. Jeder Punkt in A wird einem einzigen Punkt in B zugeordnet, aber dabei können nicht zugeordnete (und mehrfach zugeordnete) Punkte in B verbleiben. Deshalb ist h(A, B) ≠ h(B, A). Jedoch wird die Hausdorff-Distanz (oder beidseitige Hausdorff-Distanz) so konstruiert, dass sie symmetrisch ist, indem beide einseitigen Hausdorff-Distanzen betrachtet werden und nur das Maximum zurückgegeben wird. Dies sei hier in Abbildung 15 veranschaulicht.

4.2.5.1 Die Implementierung der Fehlerberechung

In *PointMesh* wird aus Gründen der Geschwindigkeit für die Fehlerabschätzung nur eine Näherung der Hausdorff-Distanz berechnet und die Berechnung nur für Dreiecke der Ausrichtungen 0 bis 3 durchgeführt, was bedeutet, dass diese Berechnung nur für Dreiecke in ungeraden Rekursionsschritten der **subdivide(...)**-Funktion durchgeführt wird.

In der Funktion

calc_error(triangle3 input, char orientation)

wurde dies folgendermaßen implementiert:

1. Wähle den linken unteren Punkt des Dreiecks und suche den entsprechenden Gitterpunkt-Index.
2. Berechne die Größe der vom Dreieck eingeschlossenen rechteckigen Region in Form der Anzahl an Gitterpunkten in x- und z-Richtung.
3. Speichere Indizes aller Punkte, die von dieser Region eingeschlossen werden in einem *STL-Vektor*.
4. Bestimme die minimale Höhe der Dreieckspunkte (Vertices) und speichere sie nach min_y.
5. Bestimme die maximale Höhe aller Gitterpunkte innerhalb der rechteckigen Region und speichere sie nach max_height.
6. Berechne den Fehler: error = max_height – min_y

Dieser Fehler ist ausschlaggebend dafür, ob eine weitere Unterteilung durchgeführt wird oder nicht. Als Entscheidungsgrenze wurde als Voreinstellung die Hälfte des Gitterpunktabstands (entspricht der zuvor gewählten Maschenweite) gewählt, damit die Dreiecke nicht zu sehr entarten. Der Nutzer kann den Schwellwert nach eigenem Ermessen ändern. Liegt der Fehler über dem Schwellwert, wird das betreffende Dreieck weiter unterteilt, d.h. die Rekursion wird fortgesetzt. Liegt er darunter, wird die Unterteilung gestoppt und für die aktuellen Dreiecke das *draw*-Flag gesetzt.
Es gibt einen Fall, in dem der Fehler nie klein genug wird, um unter diesen Schwellwert zu gelangen. Dies passiert an solchen Stellen, an denen sehr große Höhendifferenzen zwischen den Gitterpunkten auftreten. Die Region der eingeschlossenen Punkte wird hier durch die Unterteilung immer kleiner und der Fehler bleibt trotzdem über dem Schwellwert. Bei weniger als zwei eingeschlossenen Gitterpunkten ist keine sinnvolle Fehlerberechnung mehr möglich. Hier wird die Rekursion gestoppt, sobald eine Seite des zu unterteilenden Dreiecks kleiner als der zweifache Gitterpunktabstand ist. Anschließend wird das *draw*-Flag für die beiden Eingabe-Dreiecke der Funktion auf *true* gesetzt.

4.3 „Cracks“ und „T-Junctions“

Betrachtet man ein auf solche Weise generiertes Mesh näher, so fallen an vielen Stellen Brüche auf. In der Literatur spricht man auch von *Cracks* (Brüchen) und *T-Junctions* (T-Kreuzungen).
Cracks entstehen an solchen Stellen, an denen zwei Dreiecke aneinander liegen, deren Detailstufen sich um mehr als eine unterscheiden. In diesem Fall ist es möglich, dass Brüche entlang der Kanten erzeugt werden, an denen das höher detaillierte Dreieck ein zusätzliches Vertex einführt, das nicht auf der Kante des niedriger detaillierten Dreiecks liegt. Wird das Bild gerendert, können diese *Cracks* Löcher im Mesh erzeugen, die den Hintergrund durchscheinen lassen.
Ein weiteres ungewünschtes Artefakt ist die *T-Junction*. Diese tritt an den Stellen auf, bei denen ein Vertex des höher detaillierten Dreiecks auf der Höhe der Kante des niedriger detaillierten Dreiecks liegt. Dies kann beim Rendern des Bildes zu „*bleeding edges*“ (ein Flackern zwischen den Kanten) führen, verursacht durch minimale Fließkomma-Rundungsunterschiede.

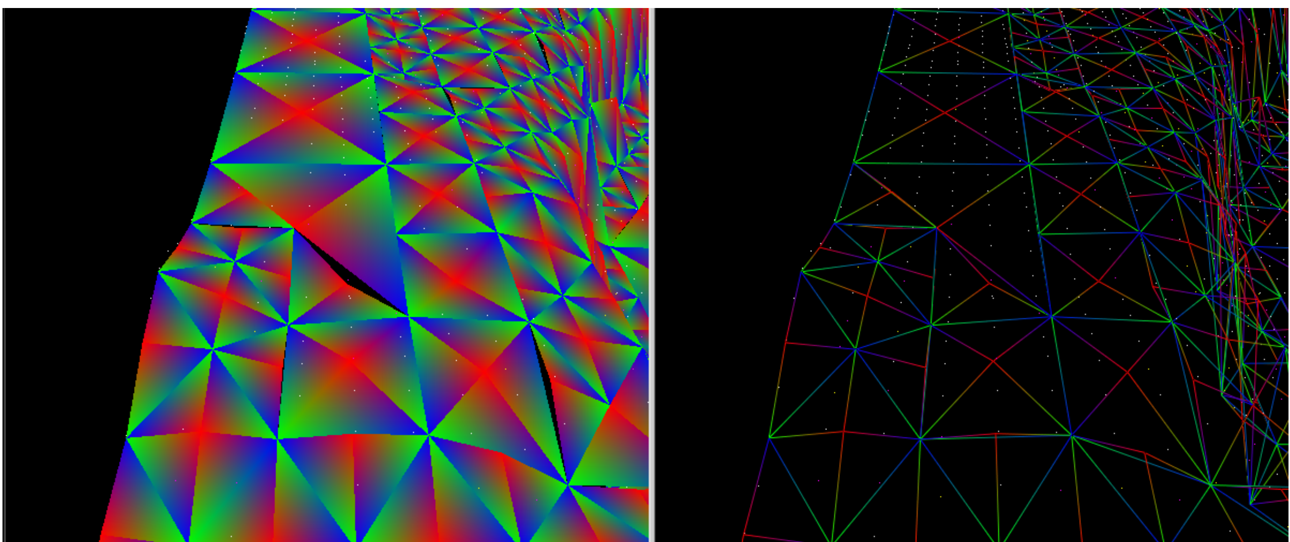


Abbildung 16: "Cracks" und "T-Junctions" - In der Mitte der Abbildungen liegen drei Dreiecke der Rekursionsstufen 7 (oberes großes Dreieck) und 9 (die beiden kleineren Dreiecke darunter) aneinander, mit einem deutlich sichtbarem Crack zwischen ihnen. Diagonal links darüber ist eine „T-Junction“ zu erkennen. Die Vertices der Dreiecke sind in der Reihenfolge rot = 1. Punkt, grün = 2. Punkt und blau = 3. Punkt eingefärbt.

Es gibt mehrere Möglichkeiten, diese Probleme zu lösen:

- Die einfachste Lösung wäre, an diesen Stellen die Höhe des T-Kreuzungspunktes auf die mittlere Höhe der beiden Kantenpunkte des dort liegenden größeren Dreiecks zu setzen. Diese Methode schließt zwar die Lücken, die durch *Cracks* entstehen, verhindert aber trotzdem keine Darstellungsfehler, da hier sämtliche *Cracks* zu *T-Junctions* werden. Der einzige Vorteil dieser Methode ist, dass hier keine zusätzlichen Dreiecke eingeführt werden müssen. [LOD_3D]

- Eine zweite Möglichkeit besteht darin, zwischen den drei Punkten eines *Cracks* oder einer *T-Junction* ein neues Dreieck einzufügen, das die Lücke schließt. Bei großen Höhenunterschieden führt diese Methode aber zu unschönen Klippen, die das Mesh unglaubwürdig und unexakt aussehen lassen. [LOD_3D]

- Die am häufigsten verwendete Lösung ist, die direkt um den *Crack* liegenden Dreiecke rekursiv zu unterteilen (*recursive splitting*), um so eine kontinuierliche Oberfläche zu erhalten. Zwar werden dadurch dem Mesh zusätzliche Dreiecke hinzugefügt, aber schließlich liefert diese Methode die besten Resultate. An den Stellen eines *Cracks* können die zwei kleineren Dreiecke (von höherer Stufe) entweder zu einem großen Dreieck einer Stufe niedriger zusammengefügt werden (*merge-Operation*) oder das große Dreieck in zwei kleinere unterteilt werden (*split*-Operation) und falls nötig auch die daran angrenzenden Dreiecke. [LOD_3D] Im Folgenden wird eine Kombination dieser beiden Lösungen vorgestellt.

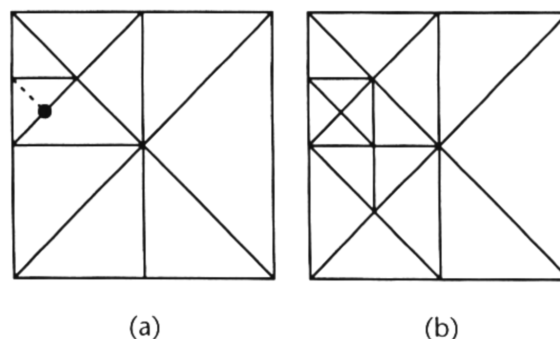


Abbildung 17: Rekursives Splitting: Die Unterteilung des durch die gepunktete Linie gekennzeichneten Dreiecks in (a) würde eine „T-Junction" an dem markierten Punkt einführen. Das Mesh in (b) zeigt das Resultat des rekursiven Splittings, um diese Situation zu umgehen. [LOD_3D]

Die Funktion

void mesh_optimization::update_neighbors(unsigned int level)

löst dieses Problem folgendermaßen:

In jeder ungeraden Rekursionsstufe:

1. Bestimmte und speichere für jedes Dreieck das Nachbardreieck mit gemeinsamer Hypotenuse. Dann gilt, dass dieses Dreieck jeweils Punkt 2 mit Punkt 3 des Nachbardreiecks und Punkt 3 mit Punkt 2 teilt. Die Dreiecke am Rand des Höhenfelds werden von dieser Suche ausgeschlossen, da sie grundsätzlich keine Nachbardreiecke mit gemeinsamer Hypotenuse besitzen.

2. Wenn das Dreieck ein Nachbardreieck mit gemeinsamer Hypotenuse besitzt und es zwar selbst gezeichnet wird (*draw*-Flag = *true*), das Nachbardreieck aber nicht (*draw*-Flag = *false*), so unterteile das Dreieck in zwei neue, setze sie auf die nächsthöhere Rekursionsstufe und kennzeichne sie mittels eines Flags, das in der Implementierung *deptri* („dependency triangle") genannt wurde.
3. Stelle diejenigen Dreiecke der niedrigeren Stufe wieder her (*draw*-Flag = *true*), die eines der durch *deptri* gekennzeichneten Dreiecke als Nachbarn besitzen und kennzeichne diese ebenfalls über das *deptri*-Flag.
4. Lösche in diesem Fall die darüber liegenden Dreiecke der höheren Stufe.

Für den Ausschluss der Dreiecke am Rand des Höhenfelds wird über die Funktion **bool testEdge(triangle3 &tri)** während des Unterteilungsprozesses ermittelt, ob ein Dreieck mit einer seiner Kanten am Rand des Höhenfelds grenzt. Dazu werden alle drei Punkte mit den Randwerten des Höhenfelds verglichen. Liegen mindestens zwei der Punkte im Bereich des Randes, so wird ein *isEdge*-Flag für das Dreieck auf *true* gesetzt, um damit das Dreieck entsprechend zu kennzeichnen.

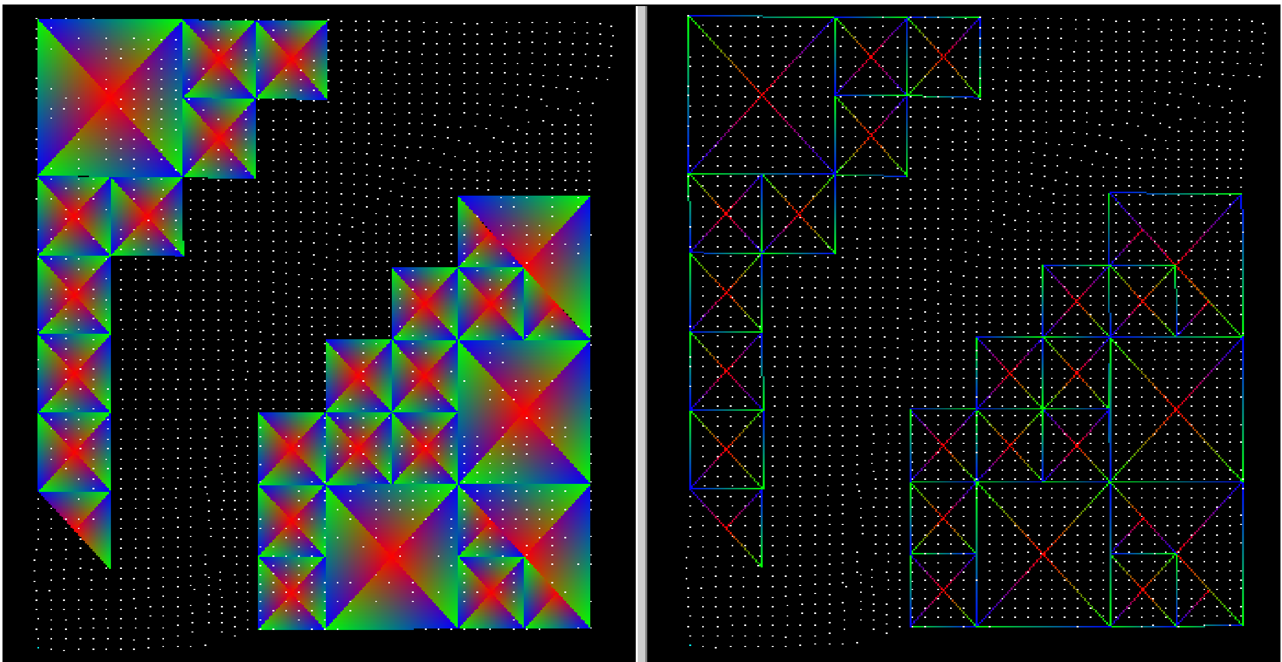


Abbildung 18: Ein unkorrigiertes Mesh während des Aufbaus durch rekursive Unterteilung. Die Abbildung zeigt den Zustand des Meshs während der Rekursionsstufe 7. In allen freien Bereichen läuft die Unterteilung zu diesem Zeitpunkt weiter. Dort existieren zwar auch Dreiecke der Stufe 7, werden aber aufgrund der Restriktionsbedingungen (Toleranzwert > Fehler) nicht gezeichnet.

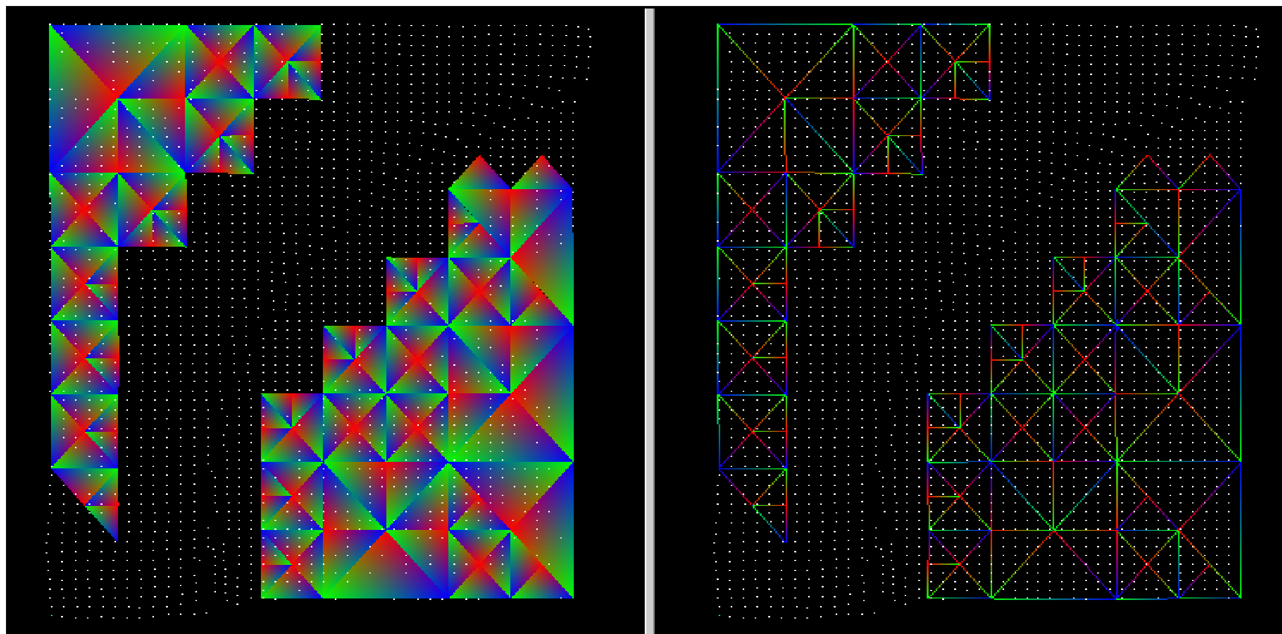


Abbildung 19: Ein durch die Funktion ***update_neighbors(...)*** *korrigiertes Mesh. Durch die strikte Unterteilung der Dreiecke in den Randbereichen zu den noch nicht gezeichneten Dreiecken schließen die Dreiecke der folgenden Unterteilungsstufen lückenlos an, sofern sie horizontal oder vertikal angrenzen.*

Da diese geometrische Korrektur für jede Rekursionsstufe durchgeführt wird, werden alle *Cracks* und *T-Junctions* zwischen nebeneinander liegenden Dreiecken mit gemeinsamer Hypotenuse verhindert. Leider werden durch dieses Verfahren nicht alle Fälle abgedeckt. Beispielsweise grenzen in beiden Abbildungen 18 und 19 rechts unten Dreiecke einer höheren Stufe diagonal an denen einer niedrigeren Stufe. Hier gäbe es auch die Möglichkeit, die Dreiecke der niedrigeren Stufe zu unterteilen. Dies führt aber zu dem Problem, dass man zum einen die Dreiecke eventuell mehrfach unterteilen müsste und zum anderen auch die benachbarten Dreiecke weiter unterteilen müsste, um so unter Umständen neu entstandene *T-Junctions* zu vermeiden.
Aufgrund der hohen Komplexität eines solchen Verfahrens soll an dieser Stelle eine andere Möglichkeit gezeigt werden, die sich im Laufe der Arbeit entwickelt hat.

4.3.1 Aufbau eines Vertex Sets

Die Dreiecke des 4-8 Meshs werden intern jeweils mit allen drei Eckpunkten gespeichert. Das ist nicht sehr effizient, da alle Punkte von mehreren Dreiecken benutzt und somit redundant gespeichert werden. Aber aus dieser Vorgehensweise lässt sich einfach feststellen, welche Punkte von welchen Dreiecken gemeinsam genutzt werden. Diese Zuordnung wurde durch die Implementierung eines *Vertex Sets* in *PointMesh* erreicht.

Das *Vertex Set* ist ein dynamischer *STL-Vektor* eines abstrakten Datentyps *vSet*. In diesem Vektor werden alle Vertices der Dreiecke jeweils nur einmal gespeichert.

Der Datentyp *vSet* integriert zunächst die folgenden Daten:

- Index des *vertexSet*-Eintrags
- Indizes aller Dreiecke, die dieses Vertex benutzen
- Position des Vertex in 3D-Raumkoordinaten

Über die Funktion void **update_vSet(...)** wird nach der Vermaschung ermittelt, welche Dreiecke einen jeweiligen Punkt des *Vertex Sets* nutzen. Deren Indizes werden innerhalb des *Vertex Sets* abgespeichert. Desweiteren werden dem *Vertex Set* spezielle Punkte hinzugefügt, die nicht direkt von Dreiecken genutzt werden, sogenannte *„Split Points“* und *„Forbidden Points“*, die ebenfalls in diesem Datentyp integriert werden.

4.3.1.1 Split Points und Forbidden Points

Bereits beim Erstellen des Meshs werden für jedes endgültig gezeichnete Dreieck zusätzliche Daten abgespeichert. Der *Split Point* definiert den Mittelpunkt der Hypotenuse, an dem ein Dreieck weiter unterteilt werden würde, auch wenn das zum Zeitpunkt der Erstellung des Meshs durch die Funktion **subdivide(...)** nicht mehr getan wird, da die Restriktionsbedingungen erfüllt sind. Wie wir aber im Folgenden sehen werden, kann eine weitere Unterteilung zur Vermeidung von *Cracks* sinnvoll sein.

Darüber hinaus werden während des Unterteilungsprozesses für jedes zu zeichnende Dreieck zwei „verbotene“ Punkte, genannt *„Forbidden Points“*, gespeichert. Diese Punkte liegen genau auf den Mittelpunkten der Katheten und wurden deshalb so benannt, weil an diesen Stellen kein Vertex eines benachbarten Dreiecks liegen darf, da dort sonst ein *Crack* oder eine *T-Junction* entstünde.

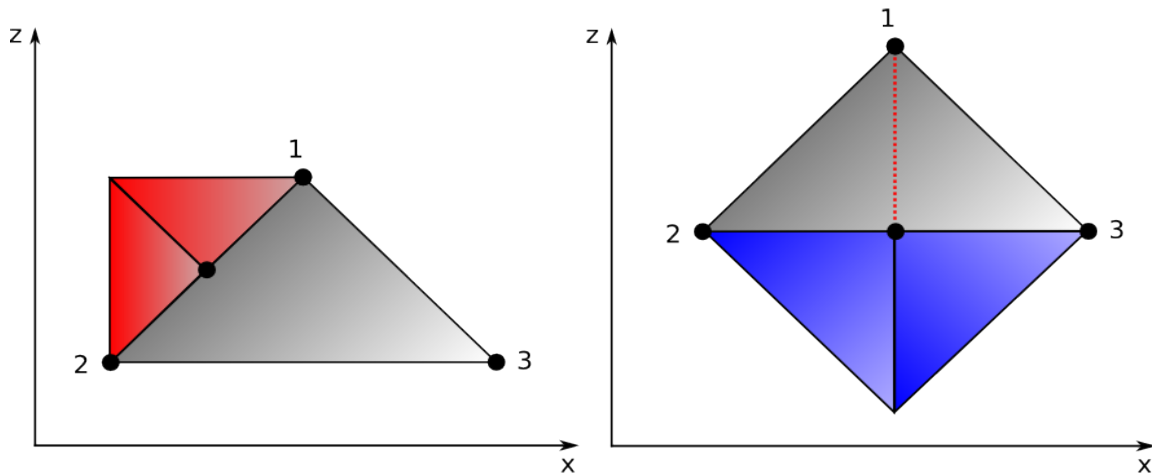


Abbildung 20: Forbidden Points und Split Points.

In Abbildung 20 werden exemplarisch diese speziellen Punkte dargestellt. In der linken Zeichnung liegen zwei Punkte der roten Dreiecke einer höheren Rekursionsstufe an einem verbotenen Punkt des grauen Dreiecks einer niedrigeren Rekursionsstufe. In diesem Fall werden die beiden roten Dreiecke zu einem Dreieck einer Stufe niedriger vereinigt (*merge*-Operation), um die *T-Junction* zu vermeiden. Dadurch geht zwar Detail verloren, dafür erspart man sich aber die mehrfache Unterteilung des grauen Dreiecks und dessen eventueller Nachbarn.

In der rechten Abbildung liegen zwei Punkte der beiden blauen Dreiecke (höhere Rekursionsstufe) an einem *Split Point* des grauen Dreiecks von niedrigerer Stufe. Hier wird das graue Dreieck in zwei neue unterteilt (*split*-Operation), um die *T-Junction* zu entfernen.

Die Funktion **update_vSet(...)** speichert auch diese drei Punkte als separate Einträge im Vertex Set ab, die über die Flags *isSplitPoint* und *isForbiddenPoint* innerhalb des Datentyps *vSet* entsprechend als solche gekennzeichnet werden. Zudem wird auch der Index des Dreiecks gespeichert, zu dem der *Split Point* oder die *Forbidden Points* gehören.

Die Funktion

void check_vSet()

überprüft nach dem Aufbau des *Vertex Sets*, welche Dreiecke ein Vertex des *Vertex Set* benutzen und aktualisiert daraufhin dessen Indexliste. Im gleichen Durchlauf wird überprüft, ob dieses Vertex ein *Split Point* oder ein *Forbidden Point* ist. Falls es ein *Split Point* ist, wird das Dreieck, zu dem dieser *Split Point* gehört, weiter unterteilt. Die Funktion **void splitpoint_split(triangle3 &tri)** führt genau diese Unterteilung durch.

Ist das Vertex ein *Forbidden Point*, werden alle Dreiecke, die diesen Punkt benutzen, gelöscht (*draw*-Flag = *false*) und stattdessen ein daran lückenlos anschließendes Dreieck wiederhergestellt, das auf eine Rekursionsstufe höher als die des Dreiecks des *Forbidden Points* gesetzt wird. Da dieses neue Dreieck wieder zwei *Forbidden Points* besitzt, wird auch für diese unmittelbar danach überprüft, ob sie schon von anderen Dreiecken benutzt werden. Ist dies der Fall, was relativ häufig auftritt, werden die daran angrenzenden Dreiecke über die Funktion **merge_tris(triangle3 &tri1, triangle3 &tri2)** zu einem neuen Dreieck der nächsthöheren Stufe vereint. Im Prinzip wäre es hier nicht nötig gewesen, ein neues Dreieck einzuführen, da genau ein solches schon im *tri_set*-Vektor (ein dynamisches Array, für jedes Dreieck die Vertices und Farben speichert) vorliegt, aber nicht gezeichnet wird. Dieses zu finden, würde aber nur zu unnötigem Rechenaufwand führen, dem der minimal zusätzliche Speicherplatz vorzuziehen ist.

Durch den Aufruf der Funktionen **update_vSet(...)** und **check_vSet()** kann auch die Integration der neu hinzugefügten Dreiecke mit seinen Nachbarn weiter überprüft und gegebenenfalls korrigiert werden, da diese immer am Ende des *tri_set* – Vektors hinzugefügt werden und somit die eventuelle Verwendung von deren *Split Points* und *Forbidden Points* unmittelbar nach dem Hinzufügen der neuen Dreiecke ebenfalls überprüft wird. Das funktioniert aus dem Grunde, da das Ende der Schleife, die diese Operationen durchführt, durch die Größe des *tri_set* - Vektors angegeben wird.

Zudem lassen sich über diese beiden Funktionen und dem *Vertex Set* prinzipiell alle Kontinuitätsprobleme innerhalb des Meshs noch während des Rekursionsprozesses lösen. Da durch die **update_neighbor(...)**-Funktion aber schon der Großteil der *Cracks* eliminiert wird, werden diese beiden Funktionen erst am Schluss aufgerufen, nachdem der Triangulierungprozess bereits abgeschlossen ist.

4.4 Exportieren der Modelle im VRML-Format

Um das komplett vermaschte Modell auch außerhalb von PointMesh benutzen zu können, muss es in einem geeigneten Format abgespeichert werden. Für den Export wurde an dieser Stelle das VRML-Format gewählt. Dieses Format besitzt den Vorteil, dass es ein ASCII-Format, also textbasiert ist und sich deshalb sehr leicht handhaben und erweitern lässt. Zudem lassen sich die Modelle auch online über gängige Webbrowser darstellen, sofern ein Browser-Plugin installiert ist. Der größte Nachteil liegt in dem vergleichsweise hohen Speicherplatzbedarf und der relativ langsamen Ein- und Auslesegeschwindigkeit im Vergleich zu anderen (vor allem binären) Formaten.

Über die GUI lässt sich sowohl das VRML-1.0-Format wie auch das VRML-2.0 Format auswählen. Die Speicherung als VRML-1.0-Format wurde implementiert, da einige 3D-Modelling-Tools (wie z.B. Blender) dieses ältere Format nutzen. Die meisten VRML-Viewer und Browser-Plugins benutzen allerdings das VRML-2.0-Format. Deshalb bleibt es dem Anwender überlassen, in welchem Format das Modell abgespeichert werden soll. *PointMesh* selbst kann zur Zeit nur 3D-Modelle im VRML-2.0-Format einlesen. Für zukünftige Versionen ist auch ein eigenes binäres Format geplant, welches die Daten kompakter speichert.

```
#VRML V2.0 utf8
Shape {
	geometry IndexedFaceSet {
		coord DEF coord0 Coordinate {
			point [
				990.201 505.146 104.902,
				990.241 505.142 104.902,
				.
				.
				.
				999.156 505.804 104.902,
				999.156 505.804 104.902,
				]
		}
		color Color {
			color [
				0.380392 0.3333 0.323529,
				0.380392 0.3333 0.323529,
				.
				.
				.
				0.439216 0.4604 0.484314,
				0.439216 0.4604 0.484314,
				]
		}
		coordIndex [
				0, 1, 2, -1,
				2, 1, 0, -1,
				.
				.
				.
				71645, 71646, 71647, -1,
				71647, 71646, 71645, -1,
				]
		}
}
```

Abbildung 21: Ausschnitt einer VRML-2.0-Datei - Für jedes Vertex werden die Position (point), die Farbe (color) und die Triangulierung (coordIndex) in separaten Abschnitten gespeichert. Im Abschnitt coordIndex sind jeweils die Indizes der Punkte angegeben, die zusammen ein Dreieck bilden. Die Zahl „-1“ dient als Trennzeichen zwischen den einzelnen Dreiecken.

5 Texturierung der Modelle mit Blender

Die automatische Gewinnung von Bilddaten aus Punktwolken wurde bereits im Rahmen eines Softwarepraktikums (siehe Abschnitt 2.1.1) in *PointMesh* integriert, so dass direkt passende Bilder im Bitmap-Format (BMP) erzeugt werden können. Ein komplizierteres Problem stellt das *Mapping* der Texturen auf die 4-8 Meshes dar.

Bei einem *Mapping* werden über eine lineare Abbildung die Pixelkoordinaten eines Bildes auf die einzelnen Dreiecke eines 3D-Modells abgebildet, wozu man eine Texturkoordinatenliste (UV-Koordinaten) bereitstellen muss. Da aber die Dreiecke eines 4-8 Meshs sich in ihrer Größe unterscheiden, ist die automatische Erstellung einer solchen Liste etwas aufwändiger, als bei den normalen regulären Meshes. Die Bearbeitung dieses Themas konnte aus Zeitgründen deshalb nicht in diese Arbeit aufgenommen werden. Glücklicherweise lässt sich die Texturierung auch manuell, über eine 3D-Rendering Software vornehmen. Für die im Folgenden gezeigten Bilder wurde die Open Source 3D-Modeling und Rendering Software *Blender* verwendet. Das *Mapping* funktioniert hier im Prinzip so, dass man ein 3D-Modell importiert und das entsprechende Bild lädt, das als Textur verwendet werden soll. Über einen UV-Image Editor lässt sich dann das Bild an die Oberfläche des Modells anpassen und gleichzeitig visuell kontrollieren. Die entsprechende Texturkoordinatenliste erstellt *Blender* intern, ohne dass der Nutzer sich damit auseinandersetzen muss.

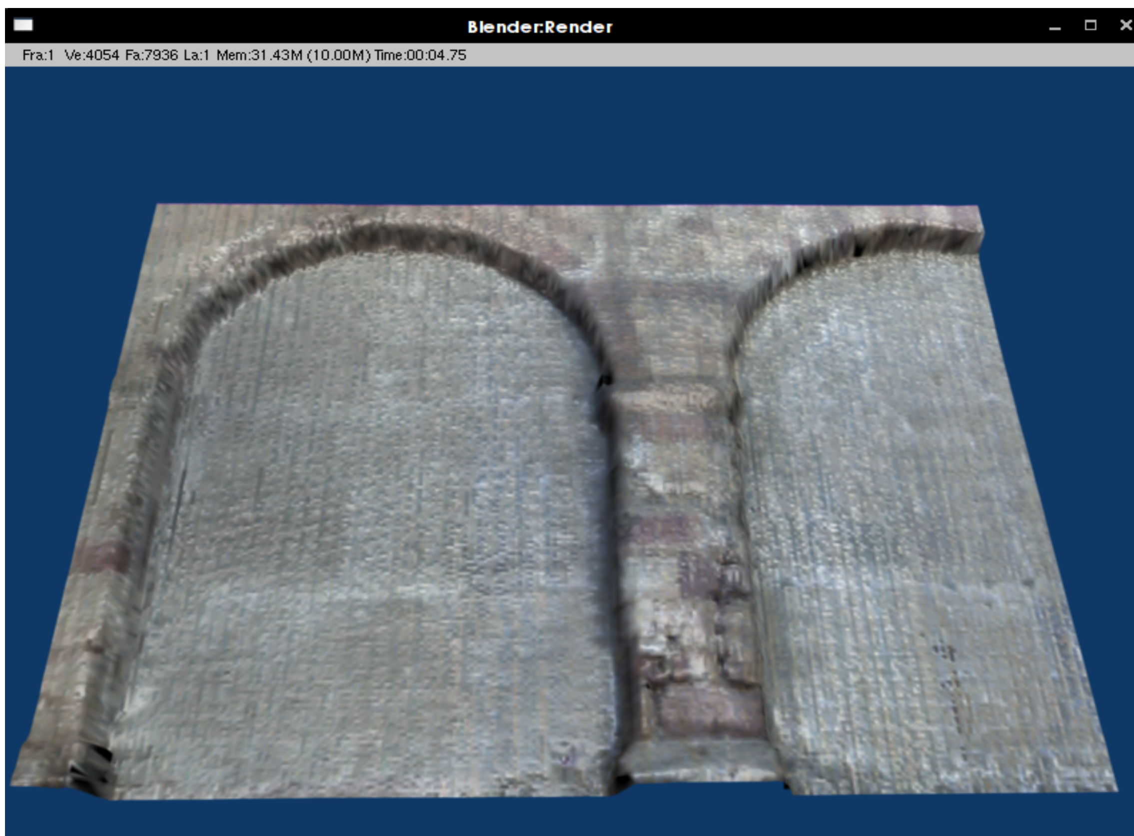


Abbildung 22: Das komplett texturierte Modell eines Ausschnitts der Arkaden des Klosters Lorsch. Hier wurde zusätzlich ein Bumpmapping, ebenfalls aus den mit PointMesh generierten Texturdaten, benutzt, um der Oberfläche den Eindruck von mehr Struktur zu verleihen.

6 Vergleich zwischen regulär triangulierten und 4-8 Meshes

Da wir die 3D-Modelle auch nach der alten regulären Vermaschungsmethode als VRML-Datei abspeichern können, lassen sich die unterschiedlichen Ergebnisse direkt vergleichen. Bei dem Vergleich beider Verfahren kommt es nicht nur auf den visuellen Eindruck an. Die adaptive Triangulierung über 4-8 Meshes sollte im Endeffekt auch Speicherplatz sparen und sehr viel flüssiger von der Grafikkarte dargestellt werden. Bei dem vorgestellten großen Ausschnitt eines Teils der Arkaden des Klosters Lorschs wurde das Modell mit einer Maschenweite von 4 cm vermascht, sowohl für die konventionelle Art der Vermaschung, als auch für die 4-8 Vermaschung. Der Toleranzwert zur Fehlerabschätzung wurde auf 2, 4, 6, 8 und 10 cm gesetzt. Außerdem wurde die mittlere *Framerate* der Darstellung gemessen. Hier ist zu beachten, dass die Darstellungsgeschwindigkeit der 4-8 Meshs noch nicht optimal ist, weil, im Gegensatz zur Darstellung der regulären Vermaschung, noch keine *Triangle-Strips* oder *Triangle-Fans* benutzt werden.

Ausschnitt „Alte Brücke“

mesh type	tolerance [cm]	#triangles	FPS
regular	n/a	24049	70
4-8-mesh	2	6823	69
4-8-mesh	4	5245	82
4-8-mesh	6	4284	94
4-8-mesh	8	3770	99
4-8-mesh	10	3281	121

Großer Ausschnitt „Arkaden“

mesh type	tolerance [cm]	#triangles	FPS
regular	n/a	70295	24
4-8-mesh	2	9060	35
4-8-mesh	4	6003	37
4-8-mesh	6	4999	40
4-8-mesh	8	3287	44
4-8-mesh	10	3130	45

Kleiner Ausschnitt „Grabung“

mesh type	tolerance [cm]	#triangles	FPS
regular	n/a	3677	167
4-8-mesh	2	1677	151
4-8-mesh	4	967	136
4-8-mesh	6	738	149
4-8-mesh	8	607	160
4-8-mesh	10	437	174

Testsystem: Intel(R) Pentium(R) Dual CPU T2330 @ 1.60GHz, 3GB RAM
Grafikkarte: NVIDIA G86GL-850

Bei den FPS-Werten handelt es sich um gerundete Mittelwerte über 60 Sekunden.
Sie wurden mit dem Programmen FRAPS und VRMLView unter Windows Vista erfasst.

Die Dreiecksanzahl korreliert nicht linear mit dem Geschwindigkeitszuwachs und der gesamten Platzersparnis, da für das adaptive Mesh auch eine Textur geladen werden muss.

Unser Hauptaugenmerk sollte aber trotzdem auf die subjektive Darstellung gelegt werden, also wie die Modelle sich visuell für einen Betrachter unterscheiden. Die regulär triangulierten Meshes bilden ein sehr dichtes Gitter aus unterschiedlich gefärbten Dreiecken. Die einzelnen Farbwerte liefern bei der Betrachtung unserem Gehirn wichtige zusätzliche Informationen über die Oberflächenstruktur. Dies sind sehr viel mehr Informationen, als wir aus der eigentlichen Geometrie des Objektes ableiten könnten. In der Computergrafik ist es deshalb üblich, Strukturinformationen von Oberflächen in Form von Texturen, also auf das Modell gelegten Bilddaten, zu übermitteln. Für die detailreduzierten 4-8 Meshes kann dann durch Texturen der visuelle Detailverlust kompensiert werden.

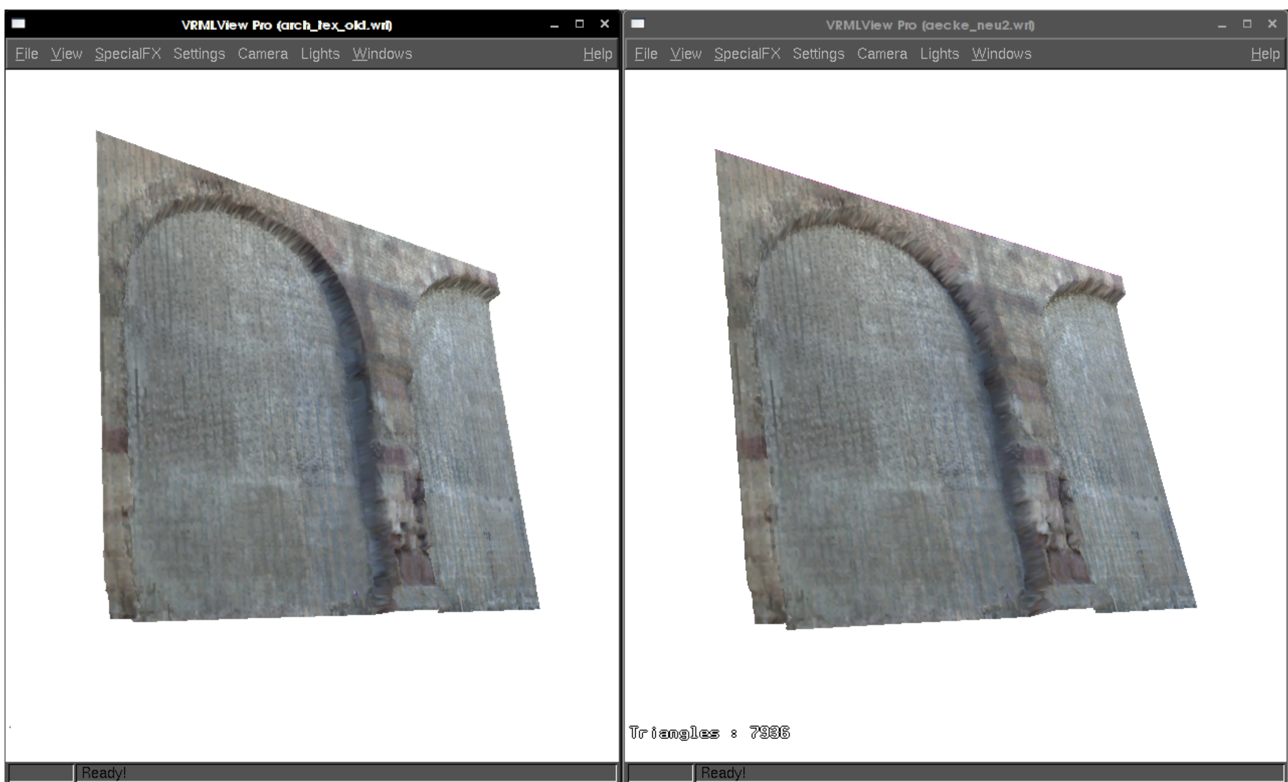


Abbildung 23: Visueller Vergleich zwischen zwei texturierten 3D-Modellen. Das linke Modell wurde regulär trianguliert (44540 Dreiecke), das rechte Modell adaptiv (7936 Dreiecke, Toleranz 2 cm).

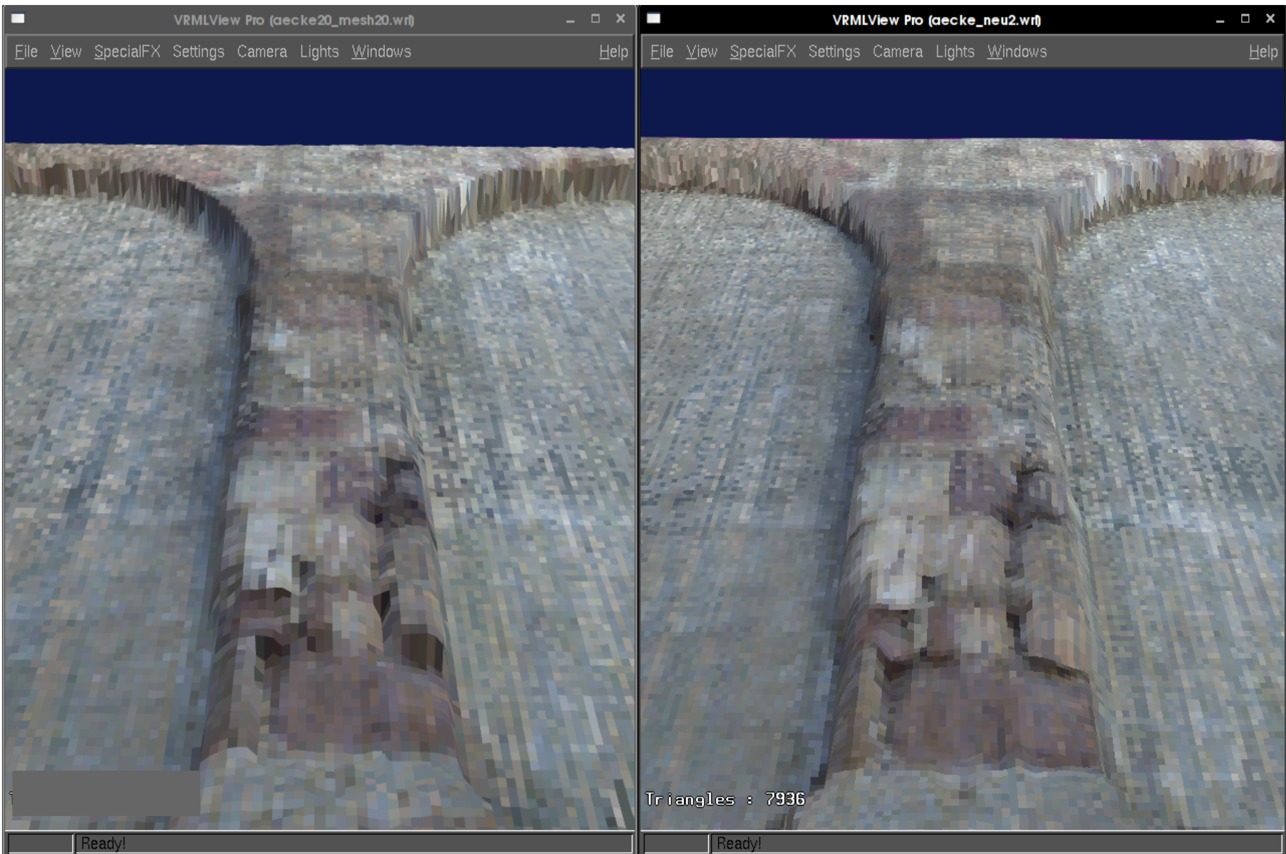


Abbildung 24: Ein vergrößerter Ausschnitt beider Modelle; links regulär, rechts adaptiv.

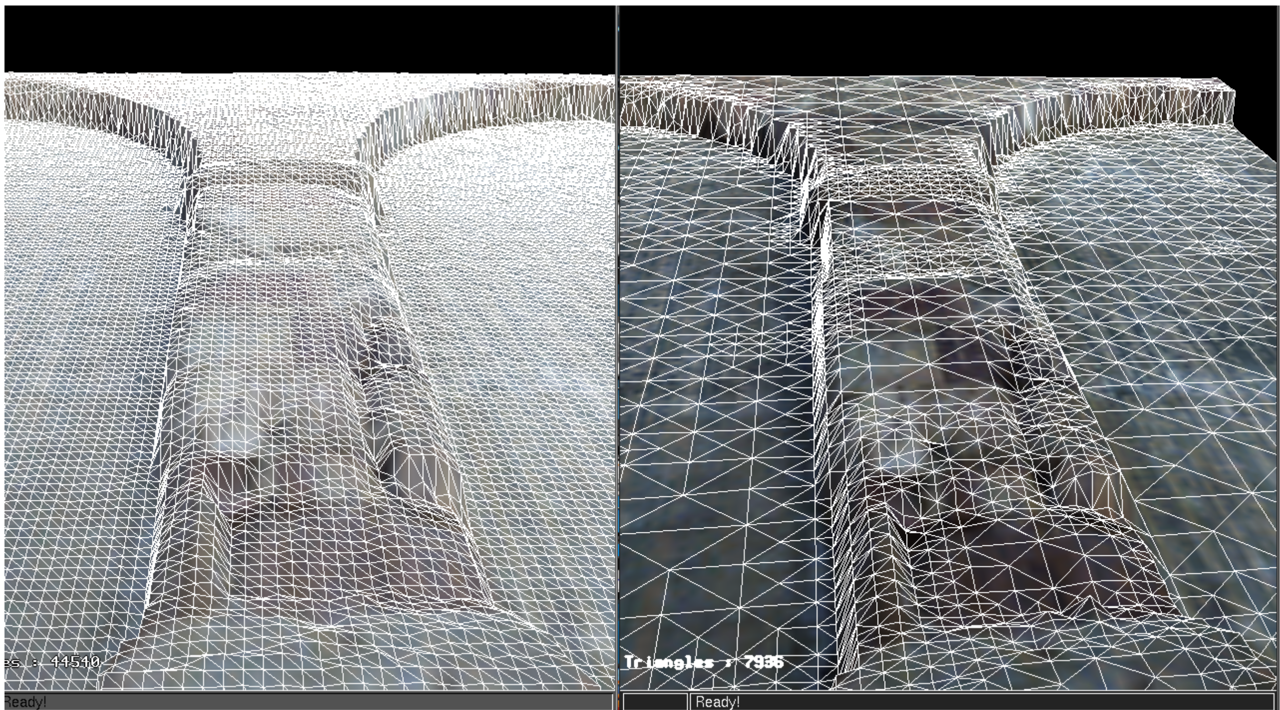


Abbildung 25: In der Drahtgitteransicht (wireframe) werden die Unterschiede zwischen den Modellen ersichtlich.

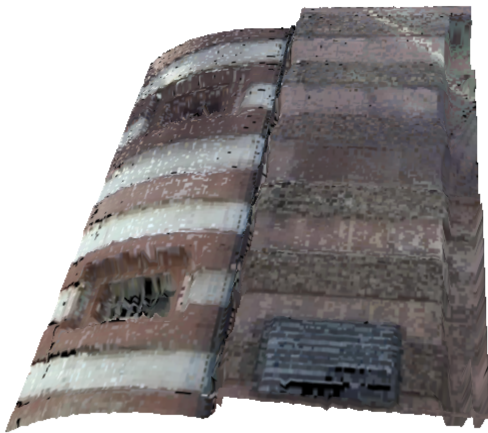
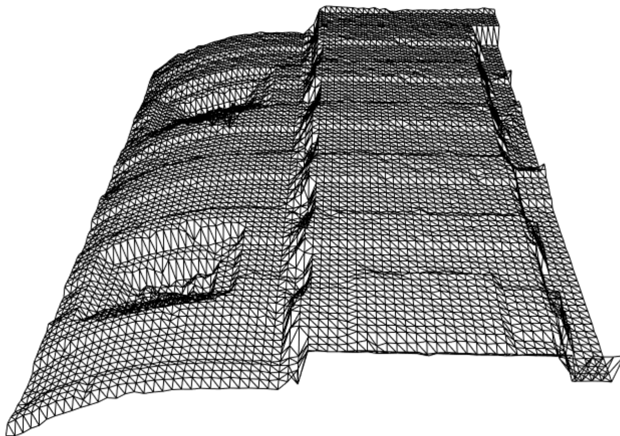


Abbildung 26: Texturierte Modelle (links: regulär – 10752 Dreiecke, rechts: adaptiv – 5015 Dreiecke) eines Ausschnitt der "Alten Brücke". Durch die Texturierung fallen die Unterschiede zwischen den Modellen kaum auf.



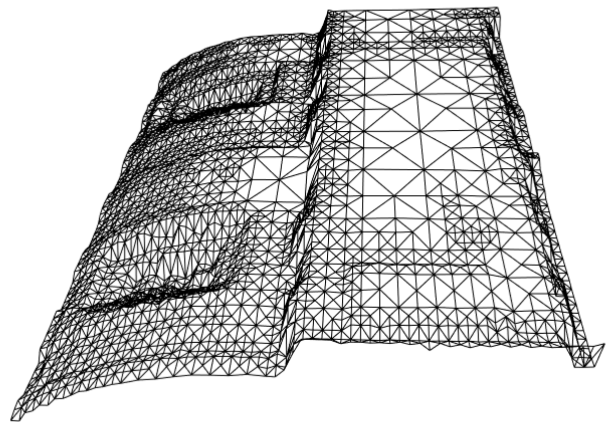


Abbildung 27: Drahtgitteransicht der beiden Modelle. Hier fällt besonders auf, dass in ebenen Regionen sehr viel weniger Dreiecke benutzt werden, als in den gekrümmten und unebenen Regionen. (Toleranzwert: 2 cm)

6.1 Die Grenzen des Verfahrens

Eine adaptive Triangulierung muss nicht notwendigerweise besser sein als eine strikt reguläre Triangulierung. Im schlechtesten Fall, wenn eine Oberfläche sehr viele Unebenheiten aufweist, funktioniert die adaptive Triangulierung kaum effizienter als die strikt reguläre. Um diesen Fall zu demonstrieren, wurde das Modell der folgenden Abbildung 28 aus einer Punktwolke einer Wand der Ausgrabungen im Kloster Lorsch generiert.

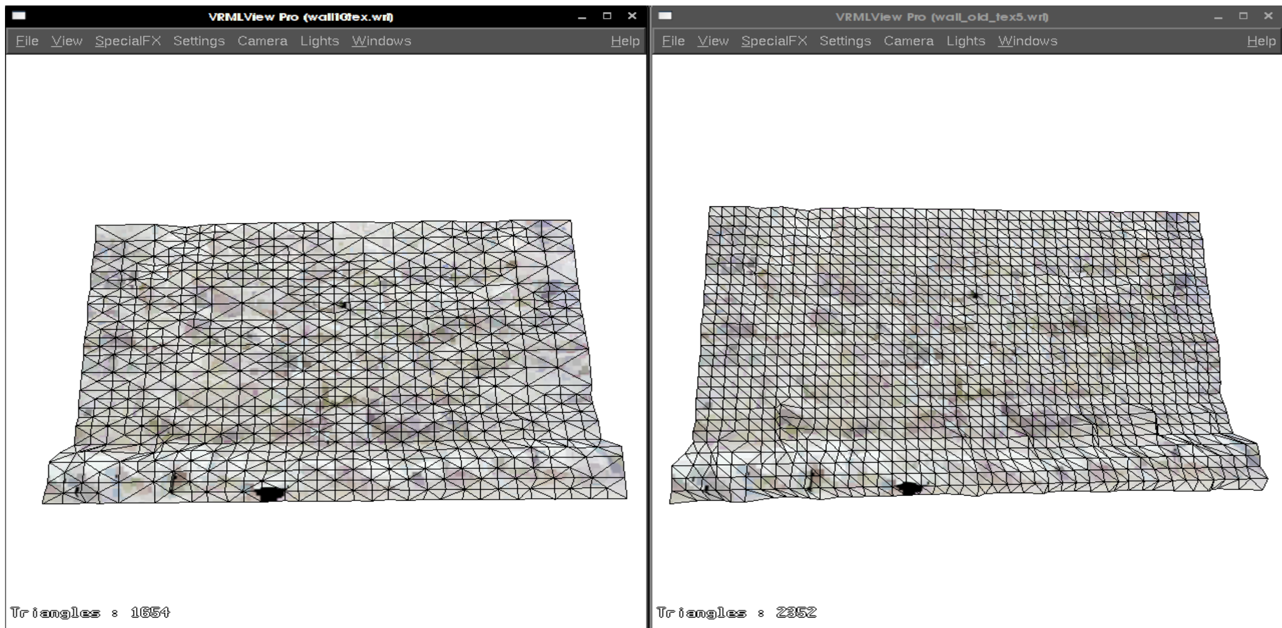


Abbildung 28: Vergleich zwischen adaptiver (links) und regulärer Triangulierung (rechts) auf einer unebenen Fläche. Wählt man den Toleranzwert zu klein (hier: 2cm), reduziert sich die Dreiecksanzahl nur minimal.

Der Detailgrad von beiden texturierten Modelle ist hier in etwa gleich. Aber Differenz der Anzahl der Dreicke beträgt hier nur 701. Das adaptive Verfahren konnte für diese Oberfläche aufgrund der starken Unebenheiten kaum weniger Dreiecke verwenden als das strikt reguläre. Man muss den Schwellwert höher setzen, um eine merkliche Reduzierung der Dreiecksanzahl zu bekommen. Leider gehen dadurch wichtige Details der Oberfläche verloren, weshalb der Nutzer diesen Wert nach seinem eigenen Ermessen setzen muss. Wenn es dem Nutzer auf kleinste Details und Unebenheiten ankommt, die nicht adäquat durch eine Textur kompensiert werden können, sollte in diesem Fall dem regulären Gitter den Vorzug gewährt werden.

7 Diskussion und Verbesserungsvorschläge

Obwohl die Ergebnisse auf den ersten Blick gut aussehen, gibt es doch noch einige Verbesserungsmöglichkeiten. Zunächst wird bei Abruf eines Werts zwischen den Gitterpunkten des Höhenfelds nur der nächste Nachbar genommen. Die Implementierung einer linearen Interpolation wäre wünschenswert und sollte insgesamt glattere Verläufe des Terrains wiedergeben.

Zudem könnte man für jeden Gitterpunkt des Höhenfelds ein Fehler über die *Hausdorff-Distanz* zum jeweiligen Dreieck berechnen, das eine Region umschließt. Dadurch würde sich die Genauigkeit sehr stark erhöhen, aber dementsprechend auch der Speicherbedarf und Rechenaufwand.

Bereits angefangen zu implementieren wurde eine nachträgliche Glättungsfunktion, die zum Ziel hat, Treppenstufen bei zu großen Höhenunterschieden zwischen den zu entfernen und das Mesh insgesamt zu glätten. Zum aktuellen Zeitpunkt ist diese Funktion aber noch nicht korrekt lauffähig.

Insgesamt lässt sich an der Verarbeitungsgeschwindigkeit noch viel verbessern. Die Dreiecke sind in einem Array angeordnet, von denen nach der Vermaschung schließlich kaum noch welche benutzt werden. Für jedes Dreieck werden zwar jeweils der Vorgänger (*Parent)* und beide Nachfolger (*Childs)* gespeichert, aber diese Baumstruktur wird noch nicht effektiv zur Vermeidung der *Cracks* genutzt. Ein sogenanntes „Tree-Pruning“-Verfahren (siehe [MESH_OPT]) könnte das Abhängigkeitsproblem effizienter lösen.

Wie schon in der Einführung angesprochen, wurde die automatische Texturierung von 4-8 Meshs noch nicht implementiert, da sich die Texturkoordinaten nicht so einfach bestimmen lassen, wie für ein reguläres Mesh. Würde man das Mesh in Form einer *Quadric* speichern und als ganzes texturieren, würde sich dieser Schritt aber automatisieren lassen.

Ein weiterer interessanter Punkt ist die Suche nach einem optimalen *Triangle Strip*, durch den das komplette Mesh beschrieben werden kann. In einem *Triangle Strip* wird nur ein Anfangsdreieck gespeichert, für alle folgenden Dreiecke werden jeweils die letzten beiden Vertices des Vorgänger-Dreiecks benutzt und nur jeweils ein neuer Vertex gespeichert. Aufeinander folgende Dreiecke besitzen hier immer eine gemeinsame Kante. Insgesamt würde das den Speicherbedarf um zwei fast Drittel reduzieren und die auch Darstellungsgeschwindigkeit großer Modelle beschleunigen.

Ein alternatives Verfahren wäre die Nutzung von *Triangle Fans*. Hier wird für alle Dreiecke, die mindestens einen gemeinsamen Punkt besitzen, dieser jeweils nur einmal gespeichert. Die anderen beiden Vertices müssen für jedes Dreieck definiert sein, wobei auch hier nacheinander erstellte Dreiecke immer zwei gemeinsame Vertices und eine gemeinsame Kante besitzen. Man kann durch dieses Verfahren den Speicherverbrauch senken und die Geschwindigkeit der Darstellung fast genauso stark beschleunigen, wie mit *Triangle Strips*, weil in einem 4-8 Mesh ein Vertex von bis zu acht Dreiecken gemeinsam benutzt wird.

Für die Zukunft wäre eine Implementierung des Algorithmus in der Sprache *C for Graphics* (Cg) vorstellbar (siehe auch: [GPU_TER]). Über diese Programmiersprache erhält man sehr leicht Zugriff auf die Vertex- und Pixelshader-Einheiten einer modernen Grafikkarte. Der Hauptvorteil der

Nutzung dieser Einheiten ist, dass diese viele Berechnungen parallel ausführen können (bei aktuellen Nvidia Geforce 9 Modellen bis zu 256 Berechnungen pro Taktzyklus [NVIDIA]), was das Verfahren insgesamt enorm beschleunigen würde.

8 Anhang

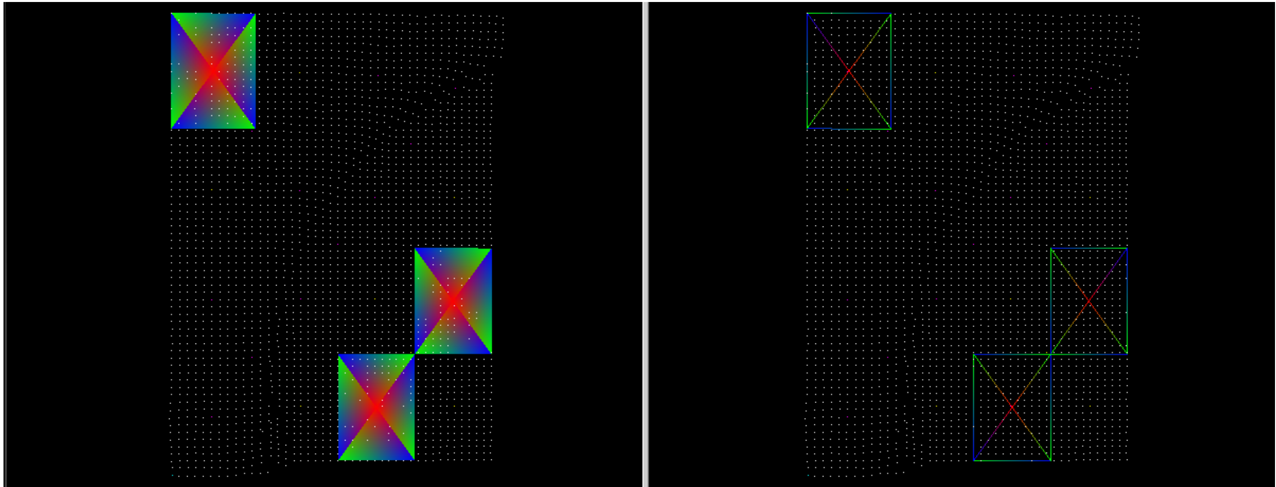


Abbildung 29: Rekursionsstufe 5 - In drei Regionen liegt der Fehler unterhalb des Schwellwerts, die "draw"-Flags der dort liegenden Dreiecke wurden gesetzt und die weitere Unterteilung abgebrochen. In allen anderen Bereichen läuft zu diesem Zeitpunkt die Unterteilung weiter.

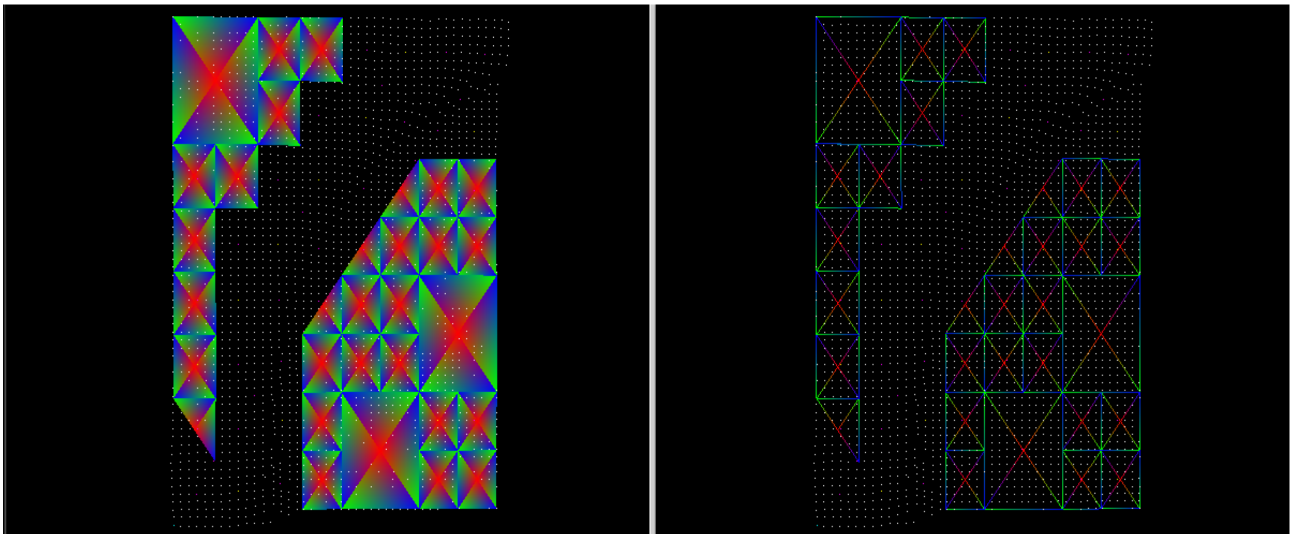


Abbildung 30: Rekursionsstufe 6 - In vielen weiteren, relativ flachen Gebieten wurde die Unterteilung gestoppt und das draw-Flag gesetzt.

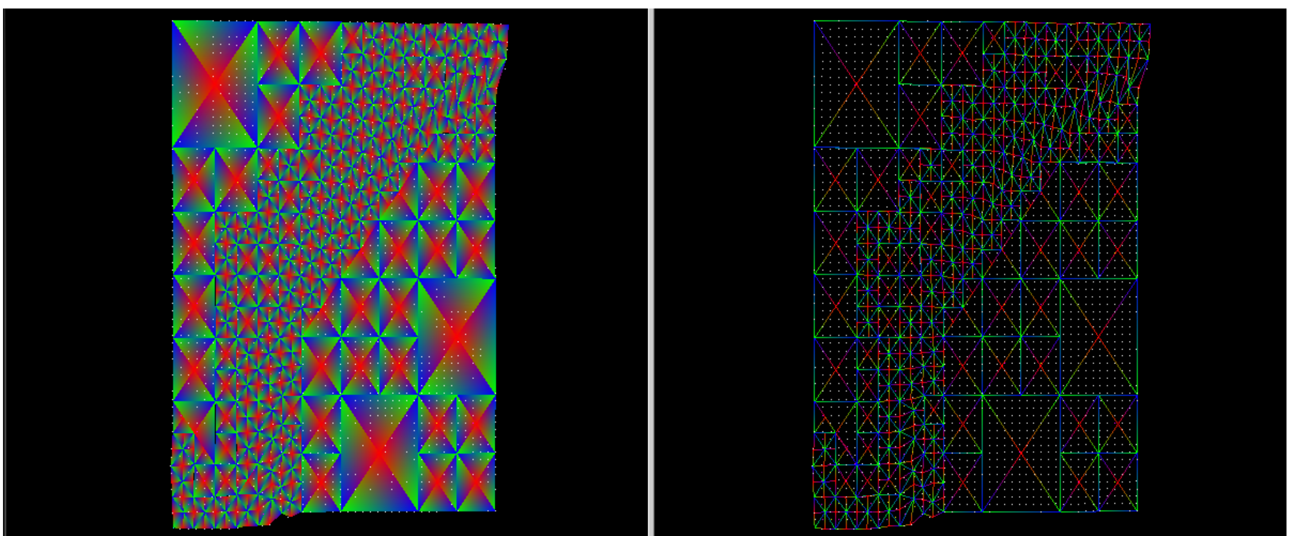


Abbildung 31: Rekursionsstufe 10 - Das Höhenfeld wurde komplett vermascht.

9 Glossar

- **Impulslaufzeit:** Die Zeitdifferenz (dt) zwischen Aussendung und Empfang eines kurzen Laserlichtimpulses, woraus sich die Entfernung zum Objekt ds = c / dt (mit c = Lichtgeschwindigkeit) bestimmen lässt.
- **Reflektanz:** Ein Proportionalitätsfaktor, der das Verhältnis zwischen der von einem beleuchteten Körper zurückgestrahlten Lichtmenge und der Intensität der beleuchtenden Quelle bezeichnet. Die Reflektanz ist über das Lichtspektrum hinweg veränderlich, da unterschiedliche Wellenlängen vom reflektierenden Körper unterschiedlich stark zurückgeworfen werden. [WIKI_DE]
- **Punktwolke:** In der Computergrafik bezeichnend für eine Liste von kartesischen 3D-Koordinaten. [WIKI_DE]
- **Kompassfarben:** *Kompassfarben* oder besser *Richtungsfarben*, werden aus den XYZ Komponenten der "Punktnormalen" berechnet und geben die Himmelsrichtung an, in die ein Punkt der entsprechend approximierten Oberfläche zeigt.
- ***RGB-Echtfarben:*** Das RGB-Farbmodell (Rot, Grün, Blau) ist das Standardmodell für die Darstellung additiver Farbmischung auf Bildschirmen. Es ist auf das Absorptionsmaxima der Sehpigmente des menschlichen Auges ausgerichtet und wird für emittierende Farb-Displays benutzt. Das RGB-Farbmodell bietet den umfangreichsten Farbraum, der mit drei Farben realisiert werden kann. Mit diesem Farbmodell können mehr Farben dargestellt werden als mit dem CMY-Farbmodell; allerdings kann es nicht alle in der Natur vorkommende Farben abbilden.
  [ZD_NET]
- ***OpenGL:*** OpenGL (Open Graphics Library) ist eine Spezifikation für ein plattform- und programmiersprachenunabhängiges API (Application Programming Interface) zur Entwicklung von 3D-Computergrafik. Der OpenGL-Standard beschreibt etwa 250 Befehle, die die Darstellung komplexer 3D-Szenen in Echtzeit erlauben. [WIKI_DE]
- **GUI:** *Graphical User Interface* = grafische Benutzeroberfläche.
- ***GLUT:*** Das *OpenGL Utility Toolkit* (GLUT) ist ein API (Application Programming Interface), dass ermöglicht, Aufgaben wie die Erstellung von Fenstern oder Tastaturein- und -ausgaben plattformunabhängig zu erledigen. Daneben bietet GLUT einige Funktionen zum Zeichnen einfacher geometrischer Objekte wie Würfel, Zylinder, Kugeln oder der Utah-Teekanne. GLUT wird seit geraumer Zeit nicht mehr weiterentwickelt, was dazu führte, dass viele Programmierer auf Alternativen wie das kompatible *FreeGLUT* oder gänzlich andere Programmierschnittstellen wechselten. [WIKI_DE]
- **Bounding Box:** ein kastenförmiges Gebilde (Würfel / Quader), welches mehrere Objekte oder ein bestimmtes Objekt umschreibt. Sie dient oft zur Auswahl oder visuellen Markierung von Objekten.
- **Point-Picking:** Auswahl eines bestimmten Punkts im 3D-Raum über ein Eingabegerät (meist Maus) auf dem Bildschirm .

- **Textur:** Eine Bilddatei, die auf eine Oberfläche (meist 3D) gelegt wird, um deren Struktur zu simulieren.
- **Matching:** Mehrere Punktwolken mit gemeinsamen oder ähnlichen Bereichen werden vereinigt, indem diese Bereiche übereinander gelegt werden.
- **Vermaschung (Meshing):** Meshing bezeichnet eine Gruppe von Verfahren in der Computergrafik. Hierbei wird eine z. B. mathematisch gegebene Oberfläche durch eine Menge kleinerer, meist ebener Elemente angenähert (approximiert). Am häufigsten kommen hier Dreiecks- oder Viereckselemente zur Anwendung. [WIKI_DE]
- **Triangulation:** Ein Dreiecksnetz ist ein ebener oder räumlicher Graph, der nur aus Dreiecken besteht. Das Dreieck wie auch dessen Ermittlung werden Triangulierung genannt. Dreiecksnetze werden in der Technik zur Vermessung und zur Modellierung verwendet. [WIKI_DE]
- **3D-Grafikpipeline:** Eine Grafikpipeline, ist eine Modellvorstellung in der Computergrafik, die beschreibt, welche Schritte ein Grafiksystem zum Rendern, also zur Darstellung einer 3D-Szene auf einem Bildschirm, durchführen muss. Eine Grafikpipeline lässt sich in drei große Schritte aufteilen: Anwendung, Geometrieverarbeitung und Rasterung. [WIKI_DE]
- **Geometrieverarbeitung:** Der Geometrieschritt, der für den Großteil der Operationen mit Polygonen und deren Eckpunkten (Vertices) verantwortlich ist, lässt sich in folgende fünf Aufgaben unterteilen: Transformation von Objekt- in Weltkoordinaten, Beleuchtungsberechnung, Transformation von 3D in 2D Koordinaten, Clipping, Window-Viewport-Transformation. [WIKI_DE]
- **Rasterung:** Im Rasterungsschritt werden alle Primitiven gerastert, also die zu ihnen gehörenden Pixel eingefärbt. Dazu ist es nötig, bei überlappenden Polygonen das jeweils sichtbare, also näher am Betrachter liegende, zu ermitteln. Für diese sogenannte Verdeckungsberechnung wird üblicherweise ein Z-Buffer verwendet. Die Farbe eines Pixels hängt von der Beleuchtung, Textur und anderen Materialeigenschaften des sichtbaren Primitivs ab und wird oft anhand der Dreieckseckpunte interpoliert. Pixel-Shader sind ebenfalls möglich. [WIKI_DE]
- **Viewport:** Der **Viewport** ist ein Ausschnitt aus der 3D Welt. Er ist genau der Bereich der später im Fenster, dem **Window**, dargestellt wird. Man könnte den Viewport somit auch als Sichtfeld des Nutzers bezeichnen. [DGL_WIKI]
- **anisotropisches Mip-Mapping: Anisotropes Filtern** bezeichnet eine Methode der Texturfilterung, speziell für Texturen in verzerrter Darstellung; beispielsweise Flächen in 3D-Szenen, die in einem flachen Winkel betrachtet werden. Laien bezeichnen dies auch als Schärfentiefe, da Texturen die weit entfernt sind, immer noch scharf dargestellt werden. Im Gegensatz zu isotropen Filtern wird hierbei ein nicht-quadratischer Filterkernel (je nach Hardware ein Rechteck oder ein Parallelogramm) angenähert. [WIKI_DE]

- **bottleneck:** dt. „Flaschenhals" - beschreibt den (zeit-)kritischen und die Gesamt-Performance verschlechternden Teil einer Hardware oder Software.
- **Vertex:** Ein Eckpunkt eines Polygons (meist eines Dreiecks), in unserem Fall auch ein Knotenpunkt eines Gitters.
- **Maschenweite:** Der Abstand zwischen zwei orthogonal benachbarten Knotenpunkten in einem regulären Gitter. Wird ein reguläres Gitter aus einem Höhenfeld unter Einbezug aller Höhenwerte erstellt, entspricht die Maschenweite dem Abstand zweier orthogonal benachbarter Höhenwerte.
- **Bitmap-Format (BMP-Format):** Ein zweidimensionales zumeist unkomprimiertes Rastergrafikformat, das für die Betriebssysteme Microsoft Windows und OS/2 entwickelt und mit Windows 3.0 eingeführt wurde. [WIKI_DE]
- **Screen-Space Error:** Der *Screenspace Error* beschreibt für einen Punkt die Abweichung von Zeichenposition zur eigentlichen Position in Pixel. Der *Screenspace Error* bietet einen - für den Menschen - gut vorstellbaren Wert. Bei entsprechend leistungsfähigen Virtual Reality Systemen wird auch des öfteren ein maximaler *Screenspace Error* von < 1 Pixel verwendet, wodurch das gezeichnete Bild immer dem gewollten Bild (wie es ohne Level of Detail aussehen würde) entspricht und zudem *Popping* verhindert werden kann. [DGL_WIKI]
- **Binärbaumstruktur:** Als Binärbaum bezeichnet man in der Graphentheorie eine spezielle Form eines Graphen. Genauer gesagt handelt es sich um einen gewurzelten Baum, bei dem jeder Knoten höchstens zwei Kindknoten besitzt. Oft wird verlangt, dass sich die Kindknoten eindeutig in linkes und rechtes Kind einteilen lassen. Ein anschauliches Beispiel für einen solchen Binärbaum ist die Ahnentafel. Hierbei sind allerdings die Elternteile die Kindknoten. Ein Binärbaum ist entweder leer, oder er besteht aus einer Wurzel mit bis zu zwei Kindknoten, die wiederum Binärbäume sind. [WIKI_DE]
- **merge-Operation:** Zwei oder mehrere aneinander liegende Flächen werden zu einer größeren vereint.
- **split-Operation:** Eine Fläche wird in zwei oder mehrere aneinander liegende Flächen unterteilt.
- **Bisektion:** Unterteilung einer Fläche in genau zwei gleich große kleinere Flächen.
- **STL-Vektor:** Die *STL Vector Class* ist eine der Basisklassen in der Standard Template Library (STL). Ein Vektor ist im Grunde ein dynamisches Array, auf das wahlfreien Zugriff über den [] Operator erlaubt ist. Es lassen sich hier zudem an beliebiger Stelle Elemente einfügen und löschen. Das Einfügen ist aber nur am Ende des Vektors effizient, da sonst der komplette Array-Inhalt nach hinten verschiebt werden muss. [STL]
- **Flag:** Ein boolscher Wert, der einen bestimmten Zustand eines Objekts mit *true* oder *false* (wahr oder falsch) kennzeichnet.

- **Rendering:** Der Prozess des Zeichnens eines Bildes auf dem Bildschirm. Siehe auch: **3D-Grafikpipeline**.
- **Framerate:** Die Anzahl der gezeichneten Bilder pro Sekunde während des Rendering-Prozesses.
- **Quadric:** *Quadriken* (engl.: *Quadrics*) sind Container für geometrische Objekte in OpenGL. Sie werden eingesetzt, wenn ohne großen Aufwand komplexere geom. Objekte (Kugel, Zylinder, ...) gerendert werden sollen. [DGL_WIKI]

10 Literaturverzeichnis

- WIKI_SCAN: *Wikipedia-Artikel: 3D-Scanner*, 2008, http://en.wikipedia.org/wiki/3D_scanner
- WIKI_SCAN2: *Wikipedia-Artikel: Laserscanning*, 2008, http://de.wikipedia.org/wiki/Laserscanning
- PHOTOGRAM: Dr. Hans-Gerd Maas, *Mehrbildtechniken in der digitalen Photogrammetrie*, 1997, Institut für Geodäsie und Photogrammetrie, Technische Hochschule Zürich
- CAL: *Callidus Precision Systems GmbH*, http://www.callidus.de
- RIEGL3D: *Riegl Laser Measurement Systems GmbH*, 2008, http://www.riegl.com
- LAS_ACC: W. Boehler, M. Bordas Vicent, A. Marbs, *Investigating Laser Scanner Accuracy*, 2003
- SWP_MESH: Christoph Hoppe, Dominik Ducati, *Softwarepraktikum: Meshing of Point Clouds from 3D Laser Scans*, WS 2006/2007, IWR, Universität Heidelberg, http://pille.iwr.uni-heidelberg.de/~laserscan01/
- SWP_TEX: Sharon Friedrich, Maik Häsner, *Softwarepraktikum: Automatisch generierte Texturen aus Punktwolken*, WS 2007/2008, IWR, Universität Heidelberg, http://pille.iwr.uni-heidelberg.de/
- BLENDER: *Blender Foundation*, http://www.blender.org
- SVDLOD: Hugues Hoppe, *Smooth View-Dependent Level-of-Detail Control and its Application to Terrain Rendering*, 2004, Microsoft Research, http://research.microsoft.com/~hoppe/
- SOAR: *SOAR Terrain Engine*, http://www.cc.gatech.edu/~lindstro/software/soar/
- LOD_3D: D. Luebke, M. Reddy, D. Cohen, A. Varshney, B. Watson, R. Huebner, *Level of Detail for 3D-Graphics*, 2003, Morgan Kaufmann, ISBN: 978-1-55860-838-9
- ROAM: Duchaineau, Wolinsky, Sigeti, Miller, Aldrich, Mineev-Weinstein, *ROAMing Terrain: Real-Time Optimally Adapting Meshes*, 1997, Proceedings of IEEE Vizualization '97, S. 81-88
- MESH_OPT: Laurent Balmelli, Martin Vetterli, Thomas Liebling, *Mesh optimization using global error with application to geometry simplification*, 2002, Journal of Graphical Models, January 2003
- GPU_TER: Jens Schneider, Rüdiger Westermann, *GPU-Friendly High-Quality Terrain Rendering*, 2006, Journal of WSCG, Vol. 14, 2006
- NVIDIA: NVIDIA Inc., *GeForce 9800 GX2 - Technical Specifications,* http://www.nvidia.com/object/geforce_9800gx2.html
- TERRVIS1: Lourena Rocha, Sérgio Pinheiro, Marcelo B. Vieira, Luiz Velho, *A FRAMEWORK FOR REAL-TIME TERRAIN VISUALIZATION WITH ADAPTIVE SEMI-REGULAR MESHES*, 2006, IMPA, Rio de Janeiro, Brazil
- QLOD: Rodrigo Toledo, Marcelo Gattass, Luiz Velho, *QLOD: An Adaptive Multiresolution Structure for Terrain Visualization*, 1996, http://www.visgraf.impa.br/Data/RefBib/PS_PDF/qlod/article.pdf
- LINDSTROM: P. Lindstrom, V. Pascucci, *Terrain Simplification Simplified: A General Framework for View-Dependent Out-of-Core Visualization*, May 2002, IEEE Transactions on Visualization and Computer Graphics
- LSTV: Renato Pajarola, *Large Scale Terrain Visualization Using The Restricted Quadtree Triangulation*, 1998, Institute of Theoretical Computer Science, ETH Zürich
- OPENGL_PG: Dave Shreiner, Mason Woo, Jackie Neider, *OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL*, Version 1.2, 2006, ISBN: 978-0201604580
- STL_REF: Nicolai M. Josuttis, *The C++ Standard Library: A Tutorial and Reference*, 1999

- COM_GRA: James D. Foley, Andries VanDam, Steven K. Feiner, *Computer Graphics: Principles and Practice in C*, 1995, ISBN-13: 978-0201379266
- WIKI_DE: Deutsche Wikipedia, Wikimedia Foundation, http://de.wikipedia.org/wiki/Hauptseite/
- DGL_WIKI: Delphi OpenGL Wiki, http://wiki.delphigl.com/index.php/Hauptseite
- STL: Standard Template Library, © Hewlett Packard / SGI (Silicon Graphics Incorporated) http://www.sgi.com/tech/stl
- *ZD_NET: Glossar – RGB-Farbmodell: http://www.zdnet.de/glossar/0,39029897,70012721p-39001674q,00.htm*