Ruprecht-Karls Universität Heidelberg
Institute of Computer Science

Bachelor Thesis

Künstliche Terrainmodellierung

Name:	Benjamin Rommel
Matrikelnummer:	2505211
Aufgabensteller:	Herr Prof. Dr. Thomas Ludwig
Betreuerin:	Frau Dr. Susanne Krömker
Abgabe Datum:	07. Juli 2008

Inhaltsverzeichnis

1 Künstliches Terrain - eine Motivation

In vielen grafischen Anwendungen spielt das Gelände eine wichtige Rolle. So kann in virtuellen Simulationen von geplanten Großbauprojekten beispielsweise gezeigt werden, wie sich das Bauvorhaben in das landschaftliche Bild einfügt. In modernen Routenplanern kann die Wegstrecke auch dreidimensional betrachtet werden, inklusive dem dazugehörigen Gelände. In Computerspielen bietet das Gelände dem Spieler häufig zahlreiche taktische und strategische Möglichkeiten, Einfluss auf das Spielgeschehen zu nehmen. Es gibt darüber hinaus noch viele weitere Anwendungsmöglichkeiten von Gelände in grafischen Anwendungen und damit in der Informatik. Da das Gelände ein so wichtiger Bestandteil zahlreicher Grafikanwendungen ist, beschäftige ich mich in dieser Arbeit mit der Frage, wie sich Gelände in Anwendungen effizient umsetzen, d.h. implementieren und rendern lässt, ohne unnötig Rechenzeit und Speicherplatz zu beanspruchen.

Ich beschäftige mich vor allem mit der Frage, wie sich das Gelände effizient visualisieren lässt. Ich nehme dabei an, dass das Gelände, welches visualisiert werden soll, bereits als Datensatz vorliegt, z.B. in Form von Satellitenaufnahmen. Diese Datensätze können jedoch nahezu beliebig groß sein, wenn beispielsweise komplette Landschaften dargestellt werden sollen. Da dies zu einem zu großen Speicherplatzbedarf führen würde, werde ich in dieser Arbeit ein Verfahren vorstellen, wie der Datensatz so reduziert werden kann, dass das Gelände aus dem Datensatz bis auf einen vorher festgelegten Fehler approximiert wird und dabei Speicherplatz eingespart werden kann, siehe dazu Kapitel 2.2. Ich stelle im weiteren Verlauf dieser Arbeit außerdem eine Methode vor, mit welcher künstliches Terrain mit Hilfe von Fraktalen modelliert werden kann, siehe Kapitel 4.

Liegen die Daten für das Gelände vor, entweder durch Reduktion eines gegebenen Datensatzes oder durch Modellierung eines künstlichen Terrains, so stellt sich die Frage nach einem geeigneten Verfahren, das Gelände zu rendern. In Kapitel 2.4 widme ich mich dieser Fragestellung und möglichen Problemen und stelle ein Verfahren vor, welches das Gelände effizient rendert und die möglichen Probleme des Renderprozesses löst. Ich werde mich im Zusammenhang mit dem Rendern ebenfalls der Frage widmen, wie sich das Gelände geeignet texturieren lässt (siehe 3.2) und stelle darüber hinaus ein Verfahren vor, welches als 'View Frustum Culling' bezeichnet wird. Mit diesem Verfahren wird lediglich der für den Betrachter sichtbare Bereich des Geländes gerendert, wodurch Rechenzeit eingespart wird.

Die letzte Fragestellung, welche in dieser Arbeit behandelt wird, ist die, wie man ein bereits gegebenes Gelände während der Laufzeit modifizieren kann, ohne das vollständige Gelände erneut berechnen zu müssen. Ein konkretes Anwendungsbeispiel hierfür sind Krater, welche beispielsweise durch einen Meteoriteneinschlag oder Waffengewalt entstehen können. Trifft ein Meteorit oder ein Explosivgeschoss auf den Boden, so entsteht durch den Krater eine lokale

Veränderung der Landschaft, der Großteil der Landschaft bleibt hiervon jedoch unberührt. In Kapiel 5 widme ich mich diesem Thema und seiner Umsetzung ausführlich. Zum Schluss werde ich noch einen Ausblick auf mögliche Aufgabenstellungen geben, welche in dieser Arbeit nicht behandelt wurden.

Der vollständige Quellcode der Implementierung dieser Arbeit wird gerne auf Anfrage zur Verfügung gestellt. Bei Interesse an den Dateien bitte eine kurze e-Mail an mich (benjamin@fantasy-online.info) schicken.

Eine Homepage zu dieser Arbeit findet sich unter:

`http://pille.iwr.uni-heidelberg.de/~artterrain01/`

2 Generieren des Terrain

2.1 Basisalgorithmen

Bevor mit der Entwicklung einer Datenstruktur, der Triangulierung, oder weiteren wichtigen Bestandteilen des Algorithmus begonnen werden kann, müssen zuerst drei grundlegende Fragen geklärt werden, welche das weitere Vorgehen und die Entwicklung des Algorithmus beeinflussen: die Strategie der Reduzierung des Ausgangsdatensatzes, die Art der Verfeinerung und die Wahl der Hierarchie der Verfeinerung.

2.1.1 Bottom-Up- und Top-Down-Ansatz

Beim Bottom-Up-Ansatz geht man davon aus, dass man zu Beginn den vollständigen Datensatz geladen und bereits vermascht hat. Um den reduzierten Datensatz zu erhalten, wird mit Hilfe eines Verfeinerungskriteriums (siehe 2.2.2) überprüft, welche der Knoten zwingend erforderlich sind. Die übrigen Knoten können entfernt werden, ohne dass der Fehler dabei einen vorher festgelegten Fehlerschwellwert überschreitet. Dieser Ansatz wird üblicherweise nur zur Initialisierung des Geländes verwendet, da bei jeder neuen Berechnung des Geländes der komplette Ausgangsdatensatz in den Speicher geladen und anschließend verfeinert werden muss. Das ist während der Laufzeit aufgrund des hohen Speicher- und Rechenaufwands nicht ratsam. Dagegen lässt sich mit dem Bottom-Up-Ansatz in der Regel die minimale Anzahl an Dreiecken finden, um die gewünschte Genauigkeit beim reduzierten Datensatz zu erhalten.

Beim Top-Down-Ansatz geht man von vier Eckpunkten des Ausgangsdatensatzes aus. Dieses Rechteck wird mit einer (oder zwei) Diagonalen in zwei (oder vier) Dreiecke unterteilt. Nun werden die Dreiecke so weit verfeinert, bis der Fehler jedes Dreiecks unter einem Fehlerschwellwert liegt, d.h. das Ausgangsgelände hinreichend gut approximiert wurde. Der Vorteil dieses Verfahrens besteht in einem geringen Speicherplatzbedarf. Während bei dem Bottom-Up-Ansatz bei jeder neuen Berechnung der vollständige Datensatz geladen werden muss, wird während des Top-Down-Ansatzes weniger Speicherplatz benötigt, da der reduzierte Datensatz, welcher beim Top-Down-Ansatz am Ende vorliegt, kleiner ist, als der Ausgangsdatensatz, welcher beim Bottom-Ansatz bereits zu Beginn vorliegt.

Da es nützlich ist, auch während der Laufzeit Änderungen am Terrain vornehmen zu können, wird für den Algorithmus der Top-Down-Ansatz verwendet. Er ermöglicht im weiteren Programmverlauf das Terrain erneut zu verfeinern, ohne unnötig viel Speicher zu belegen, wie es bei dem Bottom-Up-Ansatz der Fall wäre. [LUEBKE 02]

2.1.2 Vergleich von Regular Grids und TINs

Bei der Art der Verfeinerung existieren zwei grundlegend verschiedene Ansätze: *Triangulated Irregular Networks* (kurz: *TINs*) und *Regular Grids*.

Bei *Regular Grids* wird eine bereits existierende Rechteckfläche dahingehend verfeinert, dass sie in vier regelmäßige, d.h. gleich große, Rechteckflächen aufgeteilt wird. Da für die selbe Fläche neun anstatt vier Knoten verwendet werden, erfolgt so eine verfeinerte Darstellung der Fläche. Die feineren Rechtecksflächen können nun selbst wieder verfeinert werden. Ein Beispiel für die sukzessive Verfeinerung eines Quadrats mit *Regular Grids* zeigt Abbildung 1.

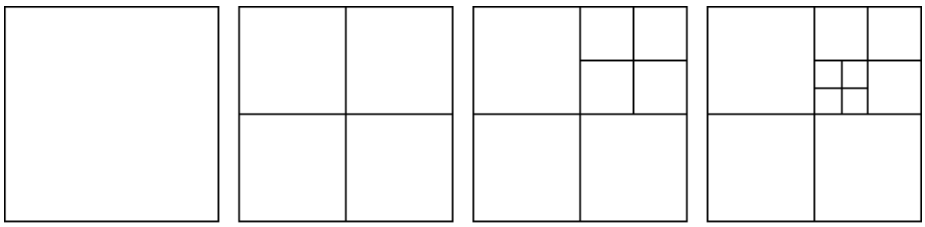


Abbildung 1: Verfeinerung mit Regular Grids

Im Gegensatz zu zu Regular Grids operieren *TINs* auf Dreiecken. Bei jedem Dreieck wird überprüft, ob das Verfeinerungskriterium erfüllt ist und wenn dies der Fall ist, wird es in zwei Teildreiecke zerlegt. Da jedes Dreieck unabhängig betrachtet und verfeinert werden kann, führt dies zu unregelmäßig vernetzten Flächen, wie es Abbildung 2 veranschaulicht.

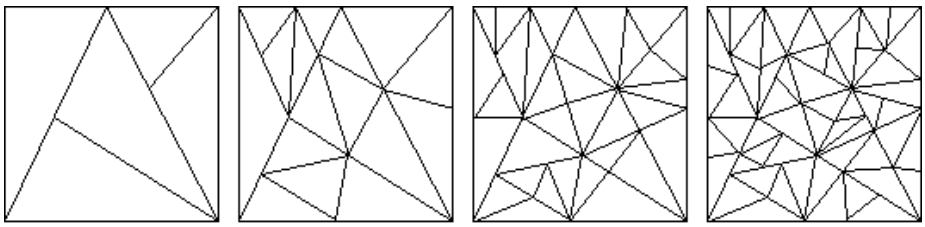


Abbildung 2: Verfeinerung mit TINs

Die beiden Ansätze sind grundverschieden und bieten beide sowohl Vor- als auch Nachteile. *TINs* benötigen in der Regel weniger Polygone und damit Dreiecke, um ein Terrain hinreichend gut zu approximieren. Große, flache Bereiche können durch sehr wenige Dreiecke repräsentiert, feine, detaillierte Bereiche durch viele Dreiecke besser approximiert werden. *Regular Grids* benötigen auch für flache Flächen mehr Dreiecke. Obwohl dies für *TINs* spricht, werden in der Regel *Regular Grids* verwendet. Auch wenn sie mehr Dreiecke für eine vergleichbar genaue Triangulierung benötigen, haben sie den Vorteil, dass sie wesentlich leichter und

speicherplatzsparender zu verwalten und zu speichern sind.

Bei einer Vermaschung des Terrains mit *TINs* muss die genaue Hierarchie der Verfeinerungen gespeichert werden, da anhand der Position eines Dreiecks nicht seine Position in der Hierarchie ersichtlich ist. Dagegen wäre es bei einer Verfeinerung mit *Regular Grids* nicht zwingend erforderlich, die Hierarchie der Verfeinerungen zu speichern, da man alle Koordinaten der Eckpunkte der Rechtecksflächen des nächsten Levels aus dem zu verfeinernden Rechteck bestimmen kann. Da jedoch durch Berechnung der Endpunkte Rundungsfehler entstehen können, wird die Hierarchie dennoch gespeichert, um die Koordinaten der Eckpunkte bei den umliegenden Rechtecken auslesen zu können.

Abbildung 3 soll dies veranschaulichen, unter der Annahme, dass eine quadratische Ausgangsfläche mit 2^k x 2^k Knoten gegeben ist. Ausgehend von einer quadratischen Fläche, welche wir zu Beginn haben, wird diese in vier gleich große Flächen verfeinert. Die Koordinaten der Eckpunkte ergeben sich aus den Eckpunkten der Ausgangsfläche und dem Grad der Verfeinerung. Ist die Kantenlänge der Ausgangsfläche 2^k Knoten lang, so reduziert sich die Kantenlänge der verfeinerten Rechtecke um die Hälfte auf 2^{k-1}, usw. Die y-Koordinaten ergeben sich entweder durch Auslesen einer *Heightmap* (siehe 2.3.1) oder durch Interpolation der Höhenwerte der benachbarten Knoten.

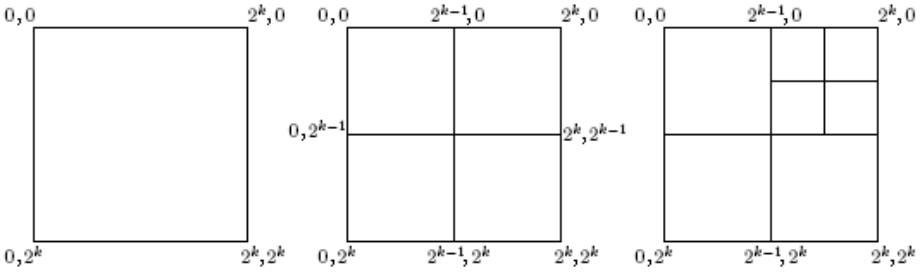


Abbildung 3: Bestimmung der Eckkoordinaten

Weitere Möglichkeiten, welche eindeutig für die Verwendung von *Regular grids* sprechen, sind die unkomplizierte Implementierung von *View Frustum Culling* (siehe 3.3), d.h. *view-dependent refinement*[1], sowie die leichte Manipulation des Geländes. Als Beispiel können hier Krater genannt werden (siehe 5). Diese werden in den aktuellen Computerspielen durch eine Textur dargestellt und nicht als dreidimensionale Krater im Gelände. Für solch eine Kratererzeugung ist es wichtig die Positionen der nachzuverfeinernden Dreiecke zu wissen. Diese Bestimmung der Positionen ist aufgrund der regelmäßigen Struktur bei *Regular Grids* einfacher und schneller zu bewerkstelligen, als bei *TINs*. Kollisionser-

[1]Die Verfeinerung des Geländes ist abhängig von der Position und dem Blickwinkel des Betrachters. Dadurch muss nicht das gesamte Gelände vollständig verfeinert werden, sondern nur der tatsächlich sichtbare Bereich. [LINDSTROM 01]

kennung, ein wichtiger Bestandteil bei vielen grafischen Anwendungen, ist mit *Regular Grids* ebenfalls um einiges einfacher zu implementieren. [LUEBKE 02]

Auch wenn man bei der Verwendung von *TINs* weniger Dreiecke für eine vergleichbare Approximation des Ausgangsgeländes benötigt, überwiegen die Vorteile von *Regular Grids*, weswegen bei dem Algorithmus *Regular Grids* verwendet werden.

2.1.3 Bin- und Quadtrees

Auch wenn das Speichern der Hierarchie für die reine Geländeverfeinerung mit *Regular Grids* nicht zwingend erforderlich ist, wird sie dennoch Bestandteil des Algorithmus, da zu späterem Zeitpunkt u.a. das *View Frustum Culling* (siehe 3.3) und die Triangulierung des Terrains (siehe 2.4.2) eine Hierarchie voraussetzen.

Unsere dritte und letzte fundamentale Frage beschäftigt sich deshalb mit der Frage, wie die Hierarchie aussehen soll. Im wesentlichen haben sich zwei Arten durchgesetzt: binary triangle trees, kurz *Bintrees*, und *Quadtrees*. Bei *Quadtrees* geht man von einer Rechtecksfläche aus, welche in vier kleinere Rechtecksflächen zerlegt wird, welche dann die Kindsknoten des Ausgangsrechtecks werden, was in Abbildung 4 oben veranschaulicht wird. Bei *Bintrees* werden anstatt Rechtecke Dreiecke in zwei Teildreiecke zerlegt, wie es in Abbildung 4 unten zu sehen ist. *Bintrees* haben sich dadurch ausgezeichnet, dass mit ihnen *Cracks* und *T-Junctions* (siehe 2.4.3) vergleichsweise leicht vermieden werden können.[LUEBKE 02]

Für den Algorithmus jedoch werde ich *Quadtrees* verwenden, da alle Operationen auf Rechtecksflächen durchgeführt werden und der *Quadtree* im Vergleich zum äquivalenten *Bintree* eine gerinigere Tiefe hat. *Cracks* und *T-Junctions* werde ich durch die Triangulierung während des Renderns vermeiden (siehe 2.4.4).

2.2 Reduzierung des Datensatzes

2.2.1 Ausgangsdatensatz

Bevor mit dem Top-Down-Ansatz das Terrain verfeinert werden kann, muss zuerst ein Gelände vorgegeben werden, das approximiert werden soll. Da das Terrain aus *Regular Grids* aufgebaut ist, kann der Ausgangsdatensatz durch eine Matrix, bzw. im Programm durch ein zweidimensionales Array dargestellt werden, in welchem die Höhendaten gespeichert werden. Dieses Array wird oft auch als *Heightmap* bezeichnet. Angenommen, die Grundfläche des Ausgangsdatensatzes ist n x m Knoten groß und das zweidimensionale Array `height[n][m]` beinhaltet die Höhendaten, dann kann jedem Knoten anhand seiner x- und z-Koordinaten der entsprechende Höhenwert mit dem Array zugewiesen werden.

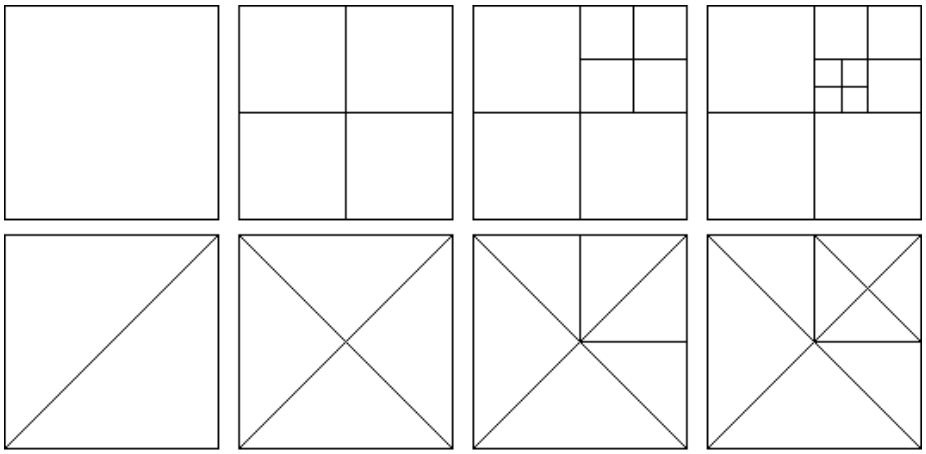


Abbildung 4: Beispiele für Bin- und Quadtreehierarchien

Nehmen wir das Ausgangsrechteck an, welches bei dem Top-Down-Verfahren zu Beginn vorliegt. Die vier Eckpunkte im Uhrzeigersinn haben die Koordinaten (1, 1), (n, 1), (n,m) und (1,m). Um diesen Punkten nun die Höhe zuzuweisen, wird auf den entsprechenden Eintrag im Array zugegriffen, d.h. der oberste, linke Punkt mit den Koordinaten (1, 1) erhält den Höhenwert `height[0][0]` und der rechte, untere Punkt den Wert `[height[n-1][m-1]`.
Die Höhendaten liegen im Programmcode nicht nur als zweidimensionales Array vor, sondern werden in der Regel auch in diversen Datenformaten zweidimensional gespeichert. Im folgenden stelle ich zwei häufig verwendete Quellen für Terraindaten vor, wie sie als Ausgangsdatensatz dienen können.

monochrome Heightmaps Eine Möglichkeit stellen monochrome Heightmaps da. Das sind Bilddateien mit einem einzigen Farbkanal, welcher für Grauwerte verwendet wird, wobei jeder Grauwert einer bestimmten Höhe des Terrains entspricht. Schwarze Bereiche geben hier die tiefstmöglichen und weiße Bereiche die höchstmöglichen Gebiete des Geländes an und die linear abgestuften Grauwerte entsprechen den Werten dazwischen. Das zweidimensionale Array des Programms, welches unsere Höhendaten speichert, kann leicht beschrieben werden, indem der Grauwert jedes Pixels der Bilddatei als Höhenwert in ein Arrayfeld geschrieben wird. Das oberste linke Pixel enspricht beispielsweise dem Arrayeintrag `height[0][0]`. Der einzige Nachteil dieses Verfahrens ist die Beschränkung auf 256 verschiedene Höhenwerte, da mit einer monochromen Bilddatei lediglich $2^8 = 256$ verschiedene Graustufen dargestellt werden können. Man könnte jedoch Bilddateien mit 3 Farbkanälen verwenden, was zu einer Auflösung von $2^{24} \approx 16.77$ Mio. Höhenstufen führt. Allerdings ist nach dem Öffnen einer RGB-Heightmap mit einem Bildbearbeitungsprogramm die Form des Geländes weniger ersichtlich als bei monochromen Heightmaps. Abbildung 5 zeigt ein Beispiel für eine monochrome Heightmap.

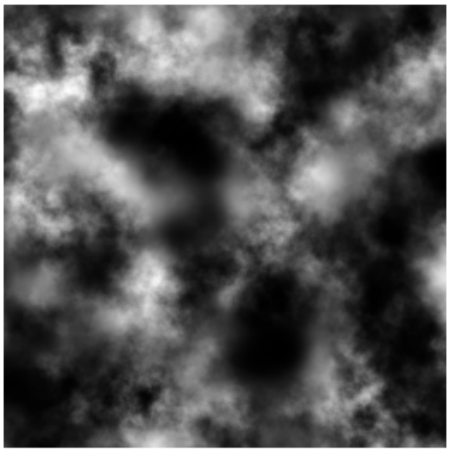


Abbildung 5: monochrome Heightmap
(http://en.wikipedia.org/wiki/Image:Heightmap.png)

Geo-Datensätze Eine weitere, viel verwendete, Quelle sind Geo-Datensätze, welche mittels Satellitenaufnahmen erstellt wurden und ab einer gewissen Auflösung frei im Internet verfügbar sind. Der große Vorteil hierbei ist, dass die Datensätze schon vorliegen und real existierende Gebiete der Erde wie etwa Skandinavien, Japan oder Europa problemlos eingelesen werden können und die Datensätze nicht manuell erstellt werden müssen, wie es beispielswese bei monochromen Heightmaps meist der Fall ist.

Der prinzipielle Aufbau eines Geo-Datensatzes ist ähnlich zu der einer monochromen Heightmap, mit dem Unterschied, dass keine Bilddateien verwendet werden, sondern Textdateien, welche mit einem Header beginnen, in dem alle relevanten Daten stehen und von einer Matrix gefolgt werden, in der die Höhendaten als ganzzahlige Werte stehen. Eines der weit verbreiteten Geo-Datenformate ist *ARC/INFO ASCII Grid*, welches mit einem Header nach folgendem Schema beginnt:

```
ncols         1001
nrows         1001
xllcorner     3600000
yllcorner     5700000
cellsize      0.0001
NODATA_value  -9999
```

Zuerst wird die Anzahl an Zeilen (`ncols`) und Spalten (`nrows`) angegeben, gefolgt von der x- und y-Koordinate des südöstlichsten Punktes. Zuletzt wird noch die Auflösung des Datensatzes angegeben (`cellsize`) und `NODATA_value` gibt den Wert an, der bei einem fehlenden Matrixeintrag zurückgegeben wird. Nach diesem Header folgen durch Leerzeichen getrennte, natürliche Zahlen, wel-

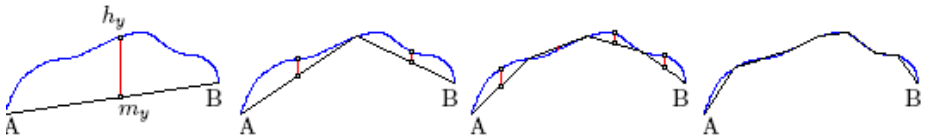


Abbildung 6: Verfeinerung einer Kante

che die Höhendaten angeben und wie auch bei monochromen Heightmap 1:1 in das zweidimensionale Array übertragen werden können. Die Informationen im Header geben lediglich einige zusätzliche Informationen, die zur Gestaltung des Geländes herangezogen werden können, etwa der genaue Abstand zweier Knoten. Andere Geo-Datenformate haben einen ähnlichen Aufbau und sind vergleichbar leicht zu handhaben.

Ich werde für mein Beispielprogramm eine Geodatei im 'ARC/INFO ASCII Grid'-Format verwenden und meine Datenstruktur in 2.3.1 basiert darauf. Eine Verwendung einer anderen Quelle wäre aber ohne nennenswerten Aufwand ebenso möglich.

2.2.2 Verfeinerung

Wir haben nun ein Ausgangsdatensatz und wir wissen, dass wir unser Terrain mit Regular Grids nach dem Top-Down-Ansatz verfeinern möchten. Was noch fehlt ist das Kriterium, nach dem entschieden wird, ob ein Rechteck verfeinert werden soll oder nicht.

Der *Object-Space Approximation Error*[2] ist das einfachste Verfeinerungskriterium und bietet dennoch alles für den Algorithmus nötige. Die Funktionsweise ist wie folgt: Gegeben seien zwei Punkte A und B, welche durch eine Kante mit einander verbunden sind. Nun wird der Mittelpunkt m dieser Kante bestimmt und der Höhenwert m_y durch Interpolation mit den benachbarten Knoten ermittelt. Anschließend wird überprüft, welchen Höhenwert h_y das zweidimensionale `height`-Array für diesen Punkt angibt und die Differenz $h_y - m_y$ ist der Fehler ϵ. Wir legen vor der Verfeinerung einen Fehlerschwellwert τ fest, welcher nicht überschritten werden darf. Eine Kante wird nun geteilt, wenn $\epsilon > \tau$. Abbildung 6 soll dies für die ersten drei Schritte veranschaulichen. Die blaue Linie ist dabei das Ausgangsgelände, die rote Linie stellt den Fehler dar. Da der Algorithmus auf Rechtecken und nicht auf einzelnen Kanten arbeitet, wird der *Object Space Approximation Error* nun so verwendet, dass bei einer Verfeinerung überprüft wird, ob der Fehler $\epsilon = h_y - m_y$ an mindestens einer Kante den Schwellwert τ überschreitet. Ist dies der Fall, wird das gesamte Rechteck verfeinert. Auf diese Art und Weise wird das Ausgangsrechteck sukzessive verfeinert, bis das Ausgangsgelände hinreichend approximiert wurde. Abbildung 7 zeigt einen Aus-

[2] siehe [PAJAROLA 07, Seite 14]

schnitt des Harzes mit verschiedenen Fehlerschwellwerten zwischen 20 und 200.

2.3 Datenstruktur

Nachdem alle grundlegenden theoretischen Fragen geklärt wurden, kommen die praktischen Fragen, d.h. die der Implementierung. Ich werde, wie im Kapitel zuvor angesprochen, für mein Beispielprogramm auf das Geo-Datenformat ARC/INFO ASCII Grid zurückgreifen. In den Basisalgorithmen wurde bereits festgelegt, dass eine *Quadtree*-Hierarchie verwendet wird und das Gelände mit *Regular Grids* unterteilt wird. In den folgenden drei Unterkapiteln beschäftige ich mich mit der Frage, wie die für den Algorithmus notwendigen Daten am effizientesten gespeichert werden können.

2.3.1 Speichern des Terrains

Als erstes stelle ich die Struktur für das Speichern aller wichtigen Geländedaten vor: *MapInfo*.

```
struct MapInfo
{
int  rows;
int  cols;
int  grid_space;
int minY;
int maxY;
int**  height;
};
```

Nachdem die ARC/INFO ASCII Grid Datei erfolgreich ausgelesen wurde, wird die Struktur `MapInfo` mit den notwendigen Informationen gefüllt. `rows` gibt die Anzahl der Knoten in x-Richtung an und entsprechend `cols` (= columns) die Anzahl der Knoten in z-Richtung. `grid_space` ist der Abstand zweier Knoten im Ausgangsdatensatz und die beiden Variablen `minY` und `maxY` geben den minimalen und maximalen y-Wert an, der ausgelesen wurde. Relevant sind die Variablen `minY` und `maxY` nur für höhenabhängige Färbung des Geländes (siehe 3.2.1). Das wichtigste Element zum Schluss: `height`. Das zweidimensionale Array `height` beinhaltet alle Höhendaten, welche aus der Datei ausgelesen wurden, d.h. die ganzzahligen Werte zwischen `minY` und `maxY`.

2.3.2 Grid-Datenstruktur

Bisher wurden als Flächen immer verallgemeinert Rechtecke angenommen, doch in den folgenden Kapiteln wird dazu übergegangen, ausschließlich auf Quadraten zu operieren, da diese sehr leicht zu berechnen und handzuhaben sind. Die Struktur `Quad`, welche verwendet wird, um die relevanten Daten zu speichern, hat folgenden Aufbau und benötigt aufgerundet 16 Byte Speicherplatz pro Quadrat:

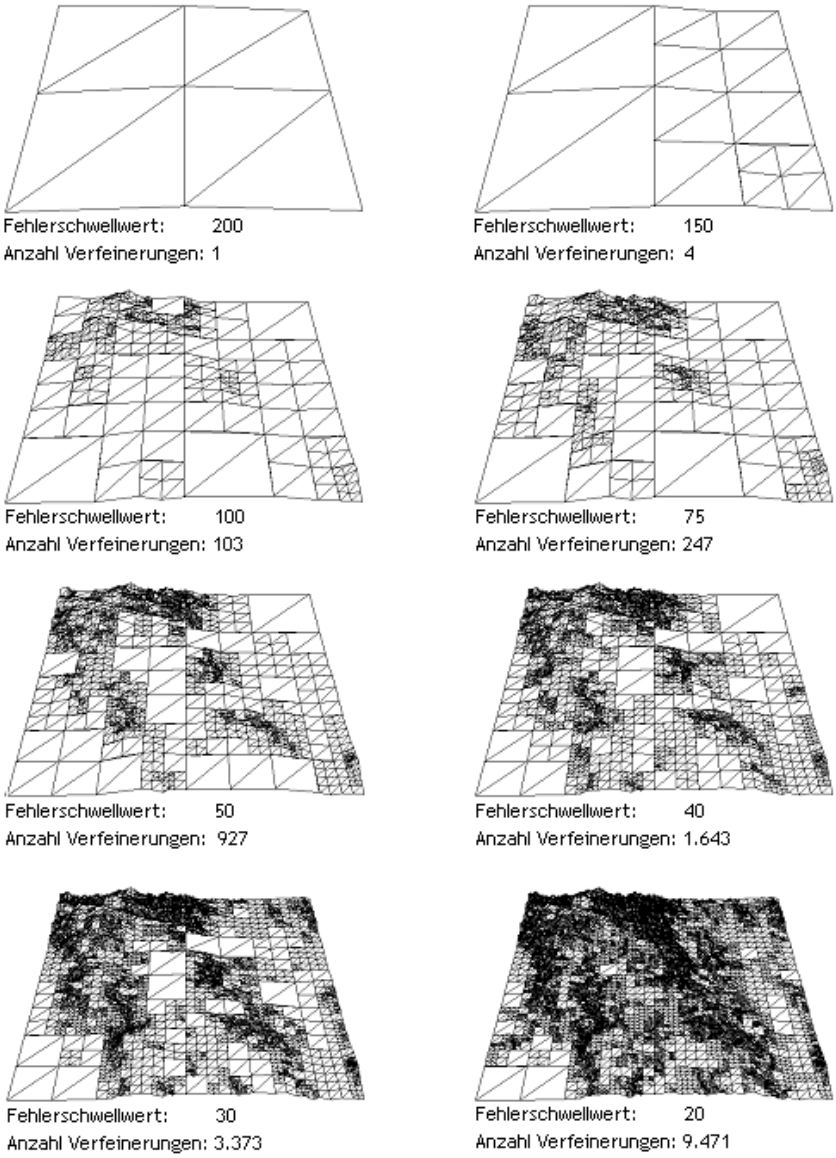


Abbildung 7: Beispiel einer Approximation eines Geländes

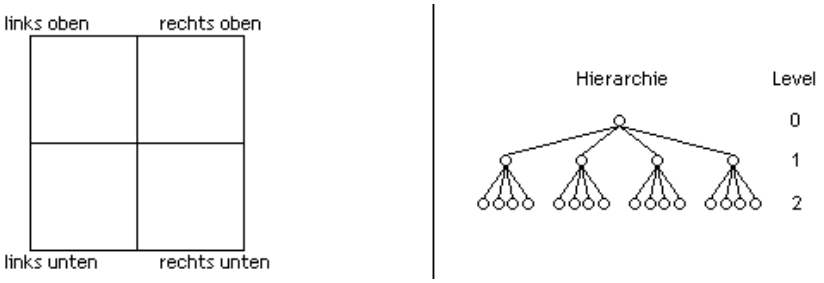


Abbildung 8:
links: Position der Quadrate in der nächsthöheren Hierarchiestufe
rechts: Level der Hierarchie

```
struct Quad
{
short x, y, z;
BYTE level;
bool  position_top, position_left;
bool top, left, bottom, right;
struct Refinement *ref;
struct  Quad *parent;
};
```

Die drei Variablen `x`, `y` und `z` geben die drei Raumkoordinaten, d.h. die Position des oberen, linken Punktes des Quadrats an. Der Wert von `level` beschreibt das Level in der Hierarchie, in der sich das Quadrat befindet (siehe Abbildung 8, rechts). Das Level 0 gehört dabei dem Ausgangsrechteck, welches vor der ersten Verfeinerung vorliegt. Die beiden bool'schen Variablen `position_top` und `position_left` geben die Position des Quadrats in der nächsthöheren Hierarchie wieder. Siehe dazu Abbildung 8, links. Sind beispielsweise `position_left` und `position_top` true, so ist das aktuelle Quadrat in der linken, oberen Ecke des Quadrats der nächsthöheren Hierarchiestufe. Wären beide Variablen false, würde es sich in der rechten, unteren Ecke befinden. Die vier Variablen `top`, `left`, `bottom` und `right` geben an, ob an den Kanten ein weiterer Knoten eingefügt werden muss oder nicht. Dies wird in Kapitel 2.4.2 ausführlich beschrieben. `ref` ist eine Instanz der Refinement-Struktur, welche in 2.3.3 behandelt wird. Der Zeiger `parent` verweist auf das Quadrat der nächsthöheren Hierarchieebene, in welcher sich das aktuelle Quadrat befindet.

2.3.3 Quadtree-Hierarchie

Um nun eine Hierarchie aufbauen zu können, wird die Struktur `Refinement` verwendet. Ist ein Quadrat kein Blattknoten, so verweist der Pointer `ref` in der `Quad`-Struktur auf eine Instanz der `Refinement`-Struktur und mit ihr auf seine Kindsknoten.

```
struct Refinement
{
Quad *top_left;
Quad *top_right;
Quad *bottom_left;
Quad *bottom_right;
};
```

Die vier Pointer `top_left`, `top_right`, `bottom_left`, `bottom_right` weisen auf den jeweils aus dem Namen ersichtlichen Kindsknoten, welcher wieder von der Struktur `Quad` ist.

2.4 Verfeinerung und Triangulierung

Nachdem alle theoretischen Aspekte geklärt und die Datenstrukturen festgelegt sind, beschäftigen wir uns nun mit der Verfeinerung und Triangulierung im Programm und den Problemen, welche dabei vermieden werden müssen.

2.4.1 Erste Verfeinerung

Zu Beginn haben wir das mehrfach angesprochene Ausgangsquadrat, welches sukzessive nach dem Verfeinerungsverfahren aus Kapitel 2.2.2 verfeinert wird. Der Grad der Verfeinerung und der dafür notwendigen Quadrate ist abhängig von dem Fehlerschwellwert, welcher verwendet wird. Abbildung 9 zeigt die Verfeinerung des Beispieldatensatzes 'data_arc_harz', welcher für alle Beispielanwendungen dieser Arbeit verwendet wird, mit einem Fehlerschwellwert von 50.

Deutlich zu sehen sind dabei die verschiedenen Feinheitsgrade im Terrain. Während der nördliche Teil des Geländes wesentlich detaillierter ist, und daher für eine hinreichend gute Approximation mehr Quadrate benötigt werden, ist der südliche Teil weitesgehend flach und entsprechend weniger Quadrate werden für die Annäherung benötigt.

2.4.2 Triangulierung

Um das Terrain darzustellen, wird jedes einzelne Quadrat trianguliert. Es wird dabei so vorgegangen, dass vier Dreiecke jedes Quadrats, welches ein Blatt in der Hierarchie ist, d.h. selbst keine weitere Verfeinerung hat, gerendert werden. Um ein Quadrat triangulieren zu können, müssen die Koordinaten der vier Eckpunkte des Quadrats, sowie die Koordinaten des Mittelpunktes bekannt sein. Die Koordinaten des Mittelpunktes können leicht mit den Koordinaten der vier

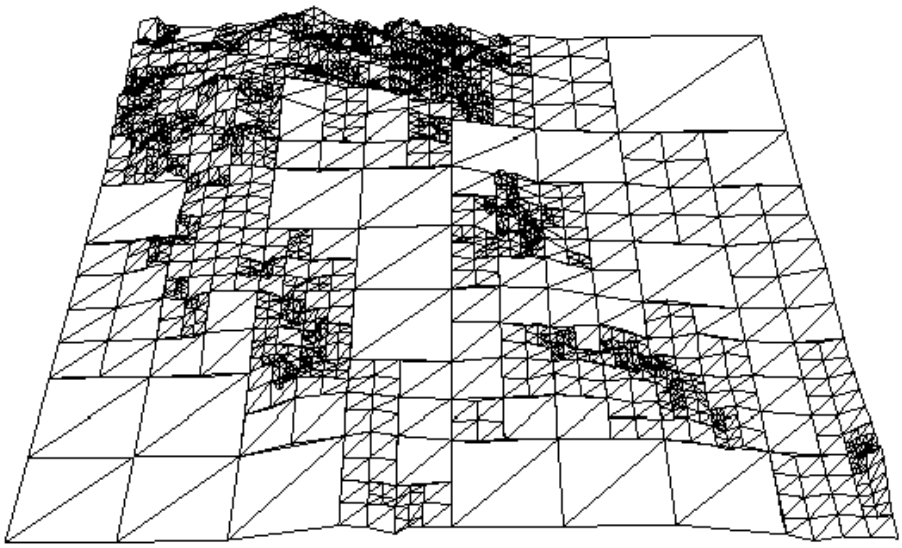


Abbildung 9: Beispiel einer Verfeinerung des Terrains

Eckpunkte berechnet werden. Der y-Wert entspricht dem gemittelten Höhenwert der vier Eckpunkte und die x- und z-Koordinaten liegen mittig zwischen dem oberen, linken und dem unteren, rechten Eckpunkt. Die Koordinaten des oberen, linken Eckpunkts sind durch die `Quad`-Struktur des Quadrates bekannt. Die drei nun verbleibenden Eckpunkte sind nicht berechenbar und müssen an angrenzenden Quadraten ausgelesen werden. Man erhält zwar leicht durch Rechnung die Länge einer Kante des Quadrats ((Kantenlänge des Ausgangsrechtecks) / $2^{LeveldesQuadrats}$) und der linke, obere Eckpunkt ist ohnehin durch die Struktur bekannt, allerdings können mit diesen Informationen die Eckunkte nicht berechnet werden, da Rundungsfehler auftreten und es so zu Artefakten zwischen zwei angrenzenden Quadraten kommen kann, welche in Form weißer Linien auftreten. Da die Berechnung nicht möglich ist, wird, wie bereits angesprochen, auf die bekannten Koordinaten der benachbarten Quadrate zurückgegriffen. Das Verfahren stelle ich nun im folgenden vor.

Von jedem Quadrat sind, aufgrund der verwendeten `Quad`-Struktur, die Koordinaten des linken, oberen Eckpunktes bekannt. Um nun beispielsweise die Koordinaten des rechten, oberen Eckpunktes zu bestimmen, muss durch den Hierarchiebaum auf das rechts vom zu triangulierenden Quadrat liegende Quadrat zugegriffen werden und von diesem dann der linke, obere Eckpunkt ausgelesen werden. Dieser Punkt ist dann der rechte, obere Eckpunkt des zu triangulierenden Quadrats. Analog verhält es zu den beiden anderen, verbleibenden Eckpunkten. Der untere, linke Eckpunkt des zu rendernden Quadrats ist der linke, obere Eckpunkt des unterhalb liegenden Quadrats. Der noch ausstehen-

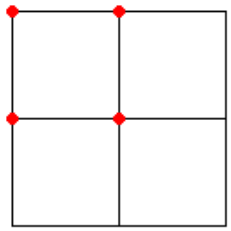
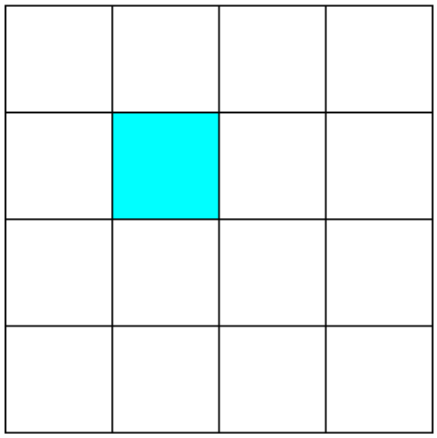


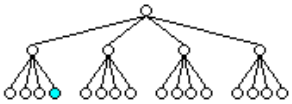
Abbildung 10: Bestimmung der Koordinaten I

de untere, rechte Eckpunkt kann mit Hilfe des rechts unterhalb liegenden Quadrats ermittelt werden. Abbildung 10 soll dies veranschaulichen, für den Fall von gleich großen, nebeneinander liegenden Quadraten. Die rot markierten Stellen sind jeweils die linken, oberen Eckpunkte der vier Quadrate und wie man der Abbildung entnehmen kann, bilden die vier roten Punkte genau die Eckpunkte des linken, oberen Quadrats.

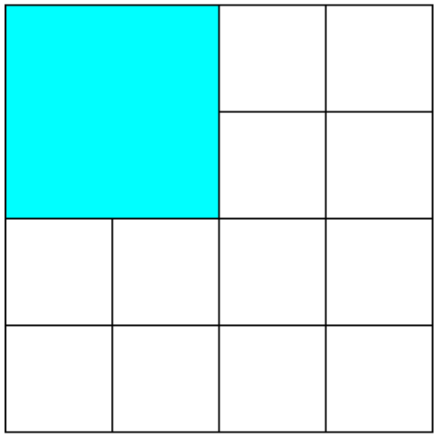
Hierbei sind jedoch zwei Dinge zu berücksichtigen: zum einen gibt es für die südlichsten und östlichsten Quadrate im Gelände keine unterhalb, bzw. rechts angrenzenden Quadrate und zum anderen haben die angrenzenden Quadrate nicht zwingend den gleichen Hierarchielevel, d.h. können unterschiedlich groß sein. Für den ersten Fall ist es notwendig, dass die Höhendaten, welche auf der unteren und rechten Kante des Ausgangsquadrats liegen, seperat gespeichert werden, z.B. in je einem Array. Die ungünstige Lage und evtl. Größenunterschiede benachbarter Quadrate können mit einem verallgemeinerten Algorithmus zum Zugriff auf benachbarte Quadrate neutralisiert werden.

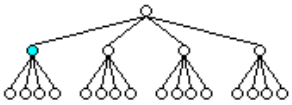
In Abbildung 11 und 12 wird dieser Algorithmus beschrieben. In diesem Beispiel soll der rechte, obere Eckpunkt des blauen Quadrates in der obersten Zeichnung ermittelt werden. Über jeder Beschreibung ist die entsprechende Hierarchiestruktur eingezeichnet. In jeder Zeichnung ist das blau markierte Feld jeweils das, auf dem operiert wird.



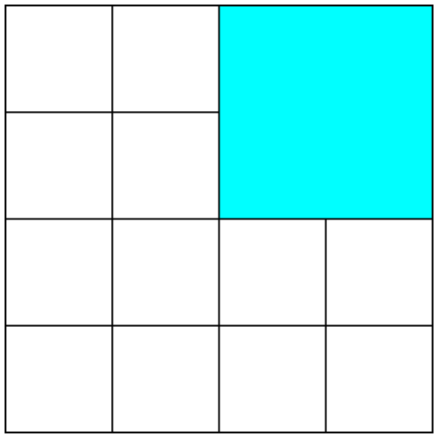


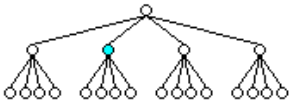
Das markierte Quadrat hat zwar einen direkten rechten Nachbarn in der gleichen Hierarchieebene, allerdings sind die beiden benachbarten Quadrate Kinder unterschiedlicher Quadrate der nächsthöheren Stufe!





Geht man in der Hierarchie um ein Level höher, so erreicht man ein Quadrat, welches einen direkten rechten Nachbarn hat, der zugleich den gleichen Elternknoten hat, wie das aktuelle Quadrat





Im dritten Schritt wird nun auf das rechts zu dem zuletzt betrachteten Quadrat liegende Quadrat zugegriffen. Man sieht, dass das Quadrat bereits den richtigen x- aber noch den falschen z-Wert hat.

Abbildung 11: Bestimmung der Koordinaten II.1

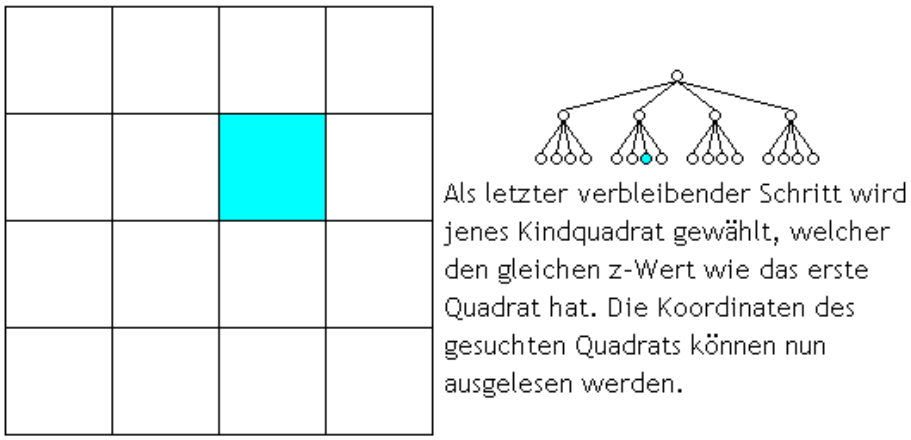


Abbildung 12: Bestimmung der Koordinaten II.2

Alle anderen Eckpunkte lassen sich völlig analog auslesen. Es wird schnell klar, dass dieses Verfahren bei einer ungünstig gelegenen Position in der Hierarchie zeitaufwändig wird und Leistung für eine akkurate und fehlerfreie Darstellung des Geländes geopfert werden muss. Durch den in einem späteren Kapitel besprochenen *View-Frustum-Culling*-Test sind jedoch keine Leistungseinbußen mehr messbar.

2.4.3 Cracks und T-Junctions

Bei einer Verfeinerung eines Geländes, bei der benachbarte Flächen eine unterschiedliche Größe haben, treten in der Regel zwei Arten von Fehlern auf: *Cracks* und *T-Junctions*. Auch nach unserer ersten Verfeinerung mit anschließender Triangulierung treten diese beiden Fehler auf. Abbildung 13 zeigt einen Ausschnitt des Geländes aus Abbildung 9 aus einem anderen Blickwinkel.

Cracks entstehen dadurch, dass ein Quadrat an mehrere Quadrate grenzt, welche in der Hierarchie ein oder mehrere Level unter dem Quadrat liegen. Die Quadrate der tieferen Hierarchielevel können dabei Eckpunkte auf der Kante des Quadrats der höheren Hierarchieebene haben. Wenn nun die Höhe dieser Eckpunkte von der interpolierten Höhe an den Kantenstellen des Quadrats der höheren Hierarchieebene abweicht, entstehen 'Lücken' im Terrain. Ein solcher *Crack* ist in Abbildungs 13 rot eingefärbt und deutlich erkennbar. Der zweite auftretende Fehler sind die so genannten *T-Junctions*. Das sind Eckpunkte von Quadraten einer tieferen Hierarchieebene, welche mit dem angrenzenden Quadrat keinen gemeinsamen Eckpunkt teilt. In Abbildung 13 werden drei solcher Stellen blau umrahmt. Es stellt sich nun die Frage, wie man diese Fehler beseitigt.[LUEBKE 02]

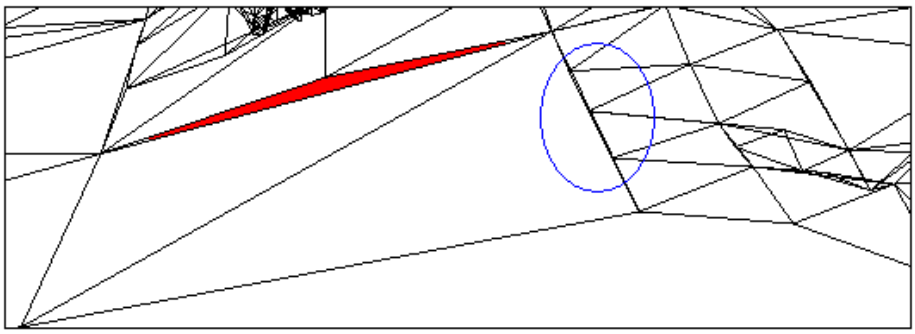


Abbildung 13: Beispiel für Crack (rot) und T-Junctions (blau)

Eine Lösung, welche auch im Rahmen dieser Arbeit verwendet wurde, ist die folgende: man verfeinert das Gelände so lange nach, bis sich zwei angrenzende Quadrate höchstens um eine Hierachiestufe ('power of 2'-Regel) unterscheiden. Siehe dazu das nächste Unterkapitel.

2.4.4 Nachverfeinern

Das Nachverfeinern ist simpel: Es wird die komplette Hierarchiestruktur durchlaufen und bei jedem Blatt der Hierarchie, d.h. bei jedem Quadrat, welches selbst keine Verfeinerung besitzt, wird überprüft, ob die vier benachbarten Quadrate der gleichen Hierarchiestufe mehr als eine Verfeinerung besitzen. Ist dies der Fall, so wird das zu untersuchende Quadrat verfeinert und die vier neu entstandenen Quadrate durchlaufen den gleichen Test. Das Untersuchen der vier benachbarten Quadrate auf ihre Tiefe erfolgt nach dem gleichen Prinzip wie bei der Bestimmung der vier Eckpunkte: die Hierarchiestruktur wird durchlaufen, um die benachbarten Quadrate zu erreichen und, anstatt die drei Koordinaten auszulesen wird nun die Anzahl der weiteren Verfeinerungen ermittelt.

Bei der Nachverfeinerung jedoch kann ein Problem auftreten: wird ein Quadrat mehrfach verfeinert, um die 'power of 2'-Regel zu erfüllen, kann es passieren, dass dadurch an dieses Quadrat angrenzende Quadrate nachverfeinert werden müssen, die eigentlich nicht hätten nachverfeinert werden müssen. Aus diesem Grund muss der Vorgang der Nachverfeinerung mehrfach wiederholt werden, bis keine weitere Verfeinerung durchgeführt werden muss. In dem für diese Arbeit geschriebenen Programm hat sich gezeigt, dass je nach Fehlerschwellwert zwischen Null und Sechs Mal nachverfeinert werden muss, wobei außer bei der ersten Nachverfeinerung kaum Verfeinerungen hinzukommen.

Ist das Gelände nun hinreichend oft nachverfeinert worden, so können die Quadrate trianguliert werden. Die in 2.3.2 beschriebene Grid-Struktur beinhaltete vier bool'sche Variablen (`top`, `left`, `bottom`, `right`), welche nun Verwendung finden. Diese vier Variablen geben an, ob die entsprechende Kante an

Abbildung 14: Triangulierung

einem Qudrat einer tieferen Hierarchieebene liegt oder nicht. Ist die Variable true, d.h. grenzt sie an ein Quadrat einer tieferen Hierarchieebene, so werden für diese Seite des Quadrats zwei Dreiecke statt einem verwendet, wobei das erste Dreieck einen Eckpunkt, den Mittelpunkt auf der Kante, sowie den Mittelpunkt im Quadrat als Eckpunkt besitzt und das zweite Dreieck den anderen Eckpunkt und die beiden angesprochenen Mittelpunkte als Eckpunkte hat. Die resultierende Triangulierung entspricht der *Restricted Quadtree Triangulation* aus [PAJAROLA 07, Seite 4]. Abbildung 14 zeigt zwei Quadrate, wobei bei dem rechten der `right`-Wert true ist.

Abbildung 15 zeigt das gleiche Gelände wie Abbildung 9, nachdem es nachverfeinert und neu trianguliert wurde. Die Nachverfeinerung geschieht ebenso wie die Verfeinerung in der Initialisierungsphase des Programms, nachdem die Terraindaten geladen worden sind. Die Triangulierung geschieht jeweils während des Renderns.

Die folgende Tabelle stellt die Anzahl der benötigten Quadrate und den benötigten Speicherplatz für verschiedene Fehlerschwellwerte dar. Verwendet wurde für alle Messungen der Terraindatensatz 'data_arc_harz', welcher bei allen Beispielanwendungen dieser Arbeit verwendet wurde. Die Spalte `Einsparung` gibt an, wie viel Speicherplatz (in %) durch Reduktion des Datensatzes gespart wurde. Die Angaben in `Speicherplatzbedarf` sind in Byte. Der benötigte Speicherplatz des Ausgangsdatensatzes betrug 4MB. Die Messungen wurden auf einer 32bit-Architektur durchgeführt.

Fehlerschwellwert	Anzahl Quadrate	Speicherplatzbedarf	Einsparung
200	5	96	99,90%
150	17	336	99,89%
100	545	10.896	99,63%
75	1.281	25.616	99,26%
50	4.781	95.616	97,51%
40	8.153	163.056	95,83%
30	16.773	335.456	91,53%
20	49.129	982.576	75,38%

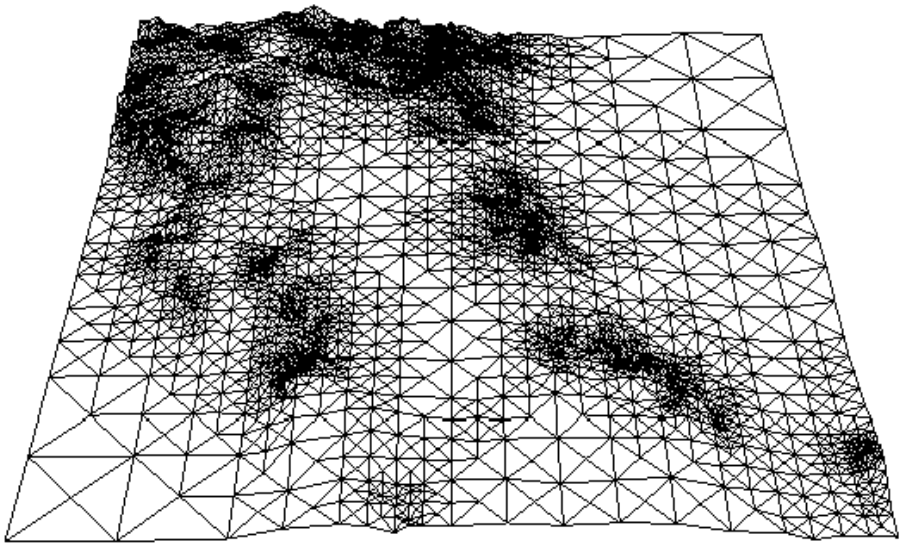


Abbildung 15: Gelände nach dem Nachverfeinern

3 Rendern des Terrains

3.1 Normalen

Normalen spielen in der Computergrafik eine wichtige Rolle. Sie sind beispielsweise zur korrekten Beleuchtung des Terrains notwendig, aber sie sind auch für viele weitere Techniken, wie etwa *Backface Culling*[3], erforderlich. Da mein Algorithmus auf Quadraten arbeitet, werde ich auch während des Renderns des Geländes für jedes Quadrat eine Normale berechnen. Man könnte sie alternativ auch in der Datenstruktur speichern, was die Berechnungen während des Renderns sparen, aber 12 Byte Speicherplatz pro Quadrat kosten würde. Noch einmal zum Vergleich: ohne Speichern der Normalen werden 16 Byte für alle Informationen benötigt. Bevor ein Quadrat gerendert wird, werden die vier Eckpunkte eines Quadrats bestimmt und wenn diese bekannt sind, kann für das Quadrat die Normale bestimmt werden.

Wir benötigen hierfür folgende drei Eckpunkte eines Quadrats: oben,links (OL), oben,rechts (OR) und unten,links (UL). Im ersten Schritt werden die beiden Vektoren $\vec{a}$ und $\vec{b}$ berechnet, wobei $\vec{a}$ von OL nach OR geht und $\vec{b}$ von UL nach OR:

[3]Mit Normalen wird eine der zwei Seiten einer Fläche als Vorderseite definiert. Backface Culling unterbindet das Zeichnen der nicht sichtbaren Seite, wodurch ein Leistungsgewinn erzielt wird.

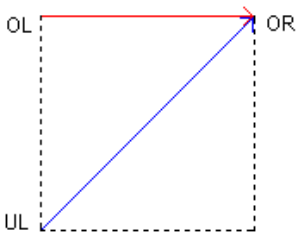


Abbildung 16: Gelände nach dem Nachverfeinern

$$\vec{a} = \begin{pmatrix} OR.x - OL.x \\ OR.y - OL.y \\ OR.z - OL.z \end{pmatrix} \text{ und } \vec{b} = \begin{pmatrix} OR.x - UL.x \\ OR.y - UL.y \\ OR.z - UL.z \end{pmatrix}$$

Abbildung 16 zeigt das Quadrat (gestrichelt), mit den drei relevanten Punkten und den beiden Richtungsvektoren $\vec{a}$ (rot) und $\vec{b}$ (blau). Die beiden Vektoren $\vec{a}$ und $\vec{b}$ spannen nun eine Ebene auf, welche parallel zu dem Quadrat liegt. Um die Normale zu erhalten, berechnen wir in einem zweiten Schritt das Kreuzprodukt $\vec{c} = \vec{a} \times \vec{b}$ der beiden Vektoren. Der durch das Kreuzprodukt erhaltene Vektor $\vec{c}$ hat die Eigenschaft, senkrecht zu der Fläche zu stehen, die durch die beiden Vektoren aufgespannt wird. Damit ist der Vektor $\vec{c}$ orthogonal zu unserem Quadrat. Als finalen Schritt muss dieser Vektor normiert werden. Dazu wird zuerst die Norm $|\vec{c}|$ von $\vec{c}$ berechnet. Ist dieser Wert ungleich 0 wird jede Koordinate von $\vec{c}$ durch diese Norm geteilt und wir erhalten als Ergebnis die Normale zu dem aktuellen Quadrat. Dieser Vorgang muss für jedes Quadrat durchgeführt werden.

3.2 Texturierung

Um das Gelände optisch anspruchsvoller zu gestalten und die monochromen Farben zu verlassen, werden im wesentlich zwei Möglichkeiten genutzt: *height-dependent Coloring* oder aber, was meist der Fall ist, Texturierung. Sowohl die höhenabhängige Farbgebung, als auch zwei Möglichkeiten zur Texturierung werden im folgenden vorgestellt.

3.2.1 Height-dependent Coloring

Bei *height-dependent Coloring* wird die Farbe eines Vertex von der Höhe, d.h. der y-Koordinate, seiner Eckpunkte abhängig gemacht. So können beispielsweise Gebiete bis zu einer bestimmten Höhe grün, stellvertretend für üppige Gras- und Weidelandschaften, und darüber Grautöne verwendet werden, um hohe Gebirge mit felsigem Gestein darzustellen. Siehe dazu Abbildung 17, unten. Hier werden die beiden Werte `minY` und `maxY`, siehe Kapitel 2.3.1, zu Hilfe genommen. Die Helligkeit der Grün-, bzw. Grautöne ist abhängig von der Differenz aus den y-Werten der Knoten und dem `minY`-Wert. Eine Differenz von 0 würde

der Farbe schwarz entsprechen, eine Differenz von maxY - minY weiß. Eine rein monochrome Färbung nach diesem Prinzip zeigt Abbildung 17, oben.

Anzumerken sei, dass *height-dependent Coloring* in der Praxis fast ausschließlich zur Analyse der Geländestruktur geeignet ist und nicht zur tatsächlichen Kolorierung von Gelände in grafischen Anwendung verwendet wird.

3.2.2 Texturierung mit einer Textur

Eine Texturierung, bei der lediglich eine Textur für das gesamte Gelände verwendet wird, bedarf zweier Dinge: einer Textur und einer Zuweisung von Texturkoordinaten zu den Eckpunkten der Dreiecke. Für die Texturierung unserer Beispielanwedung wird folgende Textur (Abbildung 18) verwendet, in einer Auflösung von 512x512 Pixeln. Damit die Textur korrekt auf unser Gelände abgebildet werden kann, müssen jedem Eckpunkt der Quadrate seine Texturkoordinaten zugewiesen werden, welche sich jedoch leicht berechnen lassen. Der linke, obere Eckpunkt des Geländes hat die Texturkoordinaten (0, 0) und der rechte, untere Eckpunkt die Koordinaten (1, 1). Somit wird die Textur genau einmal über das gesamte Gelände 'aufgespannt'. Um nun jedem Eckpunkt der Quadrate seine Texturkoordinaten zuweisen zu können, müssen lediglich die x- und z-Koordinaten des Eckpunktes durch die x- bzw. z-Koordinaten des rechten, unteren Eckpunktes des Ausgangsrechtecks dividiert werden. Dadurch erhält man die korrekten Texturkoordinaten und es ergibt sich ein Ergebnis, wie es in Abbildung 19 zu sehen ist. Das Gelände entspricht dem aus Abbildung 15 mit der in Abbildung 18 gezeigten Textur.

3.2.3 Tiling

Anstatt für das gesamte Gelände eine einzige Textur zu verwenden, können auch mehrere, kleine Texturen verwendet werden und wie kleine Kacheln auf das Gelände abgebildet werden. Der Vorteil dieses Verfahrens ist, dass wenn eine bestimmte Texturart, beispielsweise Gras, in einem großen Bereich des Geländes vorkommt, eine kleine Grastextur ausreicht und diese periodisch wiederholt werden kann. Dadurch ergibt sich ein geringerer Speicherplatzbedarf, da die für das Tiling verwendeten Texturen in der Regel eine wesentlich kleinere Auflösung haben, im Vergleich zu Geländetexturen wie in 2.3.2. Die Texturkoordinaten berechnen sich analog zu 3.2.2, wobei hier zu jeder Texturkoordinate noch die Anzahl der Kacheln in x- und z-Richtung hinzu multipliziert werden muss. Welche Textur in welcher Kachel verwendet werden soll, lässt sich in einem Array speichern.

3.3 View Frustum Culling

In der Computergrafik ist die Leistungsfähigkeit einer grafischen Anwendung, häufig in Frames pro Sekunde, kurz *fps*, gemessen, ein wichtiger Faktor. Ein modernes Computerspiel mit einer ansprechenden Grafik etwa würde sich nicht

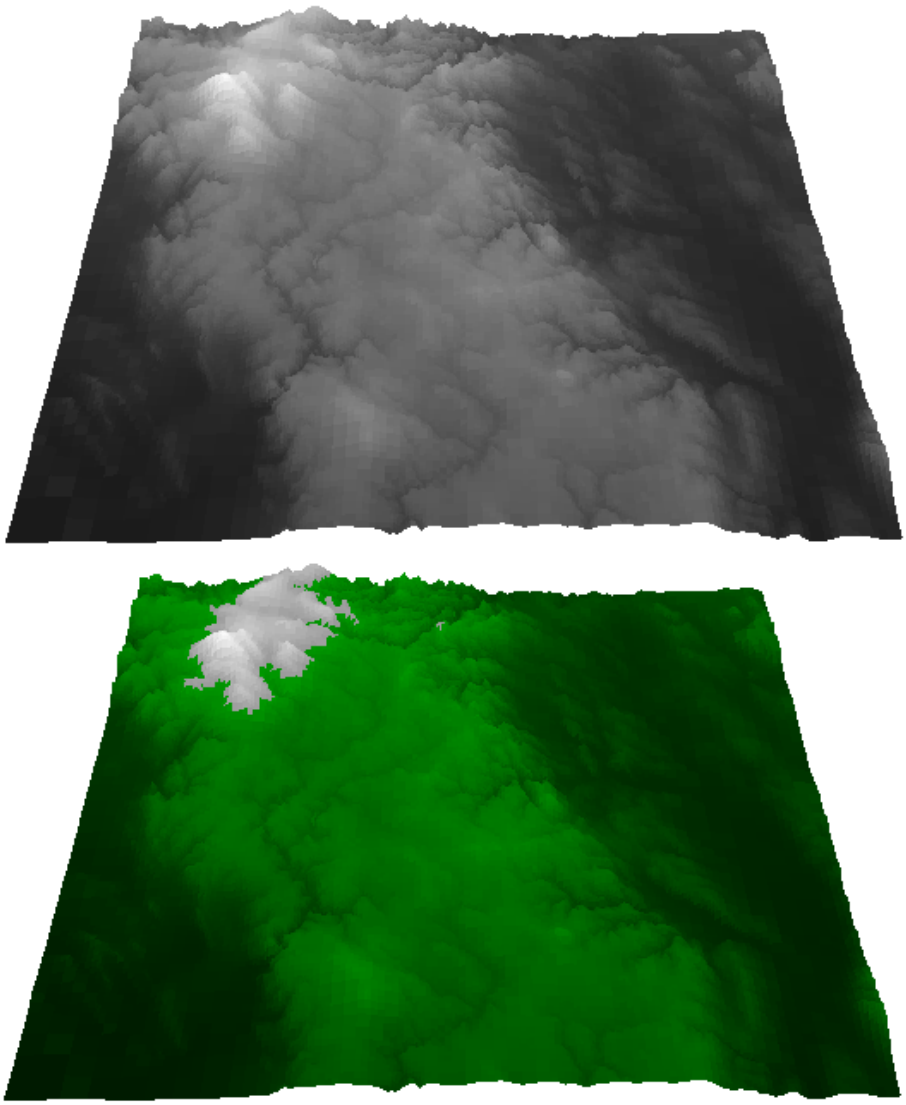


Abbildung 17: height-dependent Coloring

Abbildung 18: Geländetextur

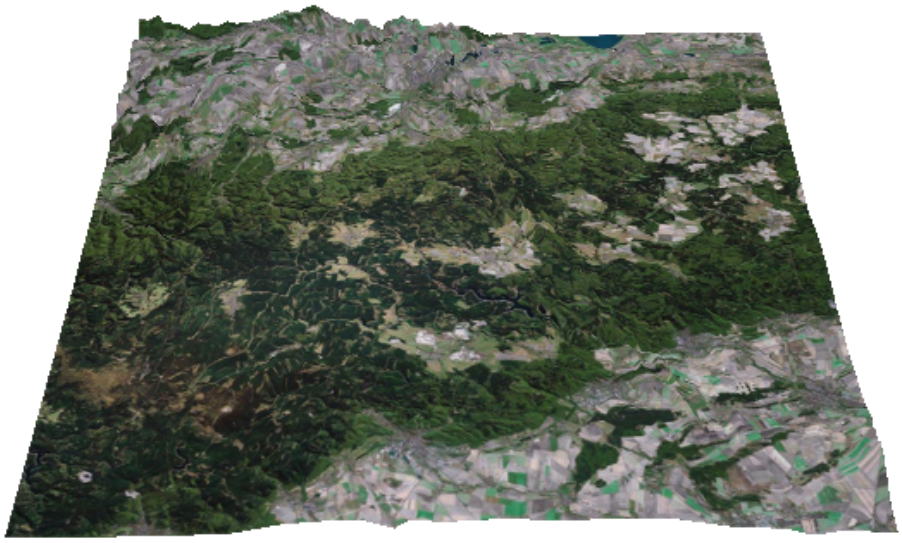


Abbildung 19: texturiertes Gelände

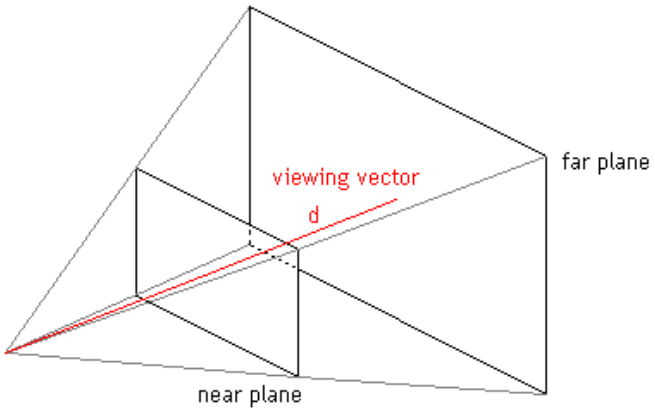


Abbildung 20: 3D-Darstellung des Sichtkegels

verkaufen, wenn es 'ruckeln' würde, d.h. der Spielablauf nicht flüssig wäre. Da der Ausgangsdatensatz in Abhängigkeit eines Fehlerschwellwertes reduziert wurde, hat sich bereits der benötige Speicherplatzbedarf für das Terrain beträchtlich reduziert. Das Rendern des gesamten Terrains ist jedoch, aufgrund der vielen Berechnungen während der Triangulierung, siehe 2.4.4, sehr aufwändig. Doch in der Praxis ist es so, dass der Betrachter lediglich einen kleinen Teil des vollständigen Geländes sieht. Wie viel der Betrachter sieht, ist abhängig von seiner Position im Raum. Ein Pilot in einem Cockpit hat dank seiner Flughöhe einen wesentlich größeren Sichtbereich, als beispielsweise ein Wanderer im Wald. Doch sie haben, wie alle Betrachter, eines gemeinsam: sie sehen nur Objekte in einem gewissen Sichtfeld vor ihnen. Der hier beschriebene *View Frustum Culling*-Test überprüft, ob ein Quadrat im Sichtfeld des Betrachters liegt und falls ja, wird dieses gerendert. So müssen beispielsweise die aufwändigen Berechnungen zur Bestimmung der Eckpunkte bei der Triangulierung für Flächen hinter dem Betrachter nicht durchgeführt werden. In OpenGL-Anwendungen ist der Sichtkegel ein Stumpf einer Pyramide mit rechteckigem Grund, siehe Abbildung 20.

3.3.1 View Frustum Culling Test im Überblick

Alle Objekte, die zwischen der *near plane* und der *far plane* liegen (siehe Abbildung 20), liegen im möglichen Sichtbereich, d.h. alle Objekte, die vor der near plane oder hinter der far plane sind, werden nicht dargestellt. Dies geschieht bereits ohne *View Frustum Culling*, da die near und far plane bereits in OpenGL in der Funktion `gluPerspective` angegeben werden müssen. Diese Funktion, sowie die Funktion `gluLookAt` besitzen alle für den *View Frustum Culling* Test notwendigen Parameter.

Das Vorgehen ist wie folgt: es werden für die sechs Seiten des Pyramiden-

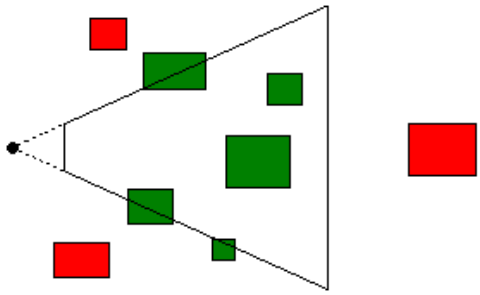


Abbildung 21: 2D-Projektion des Sichtkegels

stumpfes (die near und far plane, sowie die vier Seiten), jeweils die Normalen berechnet und ein Sattelpunkt bestimmt. Mit diesen Informationen wird für jede Fläche eine Ebene im dreidimensionalen Raum definiert. Diese Berechnungen müssen bei jeder Aktualisierung der Betrachterposition oder des Blickwinkels durchgeführt werden, z.B. wenn sich der Betrachter bewegt, neigt oder dreht. Der eigentliche Test sieht nun so aus, dass der Abstand jeder der vier Eckpunkte jedes Quadrats, welches gezeichnet werden soll, zu jeder der sechs Flächen berechnet wird. Der tatsächliche Abstand zu der Fläche spielt keine Rolle. Für uns ist lediglich das Vorzeichen des Abstandes interessant. Ist das Vorzeichen des Abstandes eines Punktes zu einer Fläche positiv, so liegt er 'vor' der Fläche, d.h. in jener Richtung, in welche die Normale zeigt. Der Test für ein Quadrat ist demnach positiv, wenn mindestens einer der vier Eckpunkte eines Quadrats den Test besteht, das heißt, dass der Abstand des Eckpunktes zu allen sechs Flächen ein positives Vorzeichen hat. Damit ist sichergestellt, dass zumindest ein Teil des Quadrats innerhalb des Sichtkegels liegt und das Quadrat gezeichnet werden muss. In Abbildung 21 wird der Sichkegel schwarz umrandet, die grünen Flächen würden den Test bestehen, die roten nicht.

3.3.2 Berechnungen

Alle für die Berechnungen relevanten Parameter sind in den beiden Funktionen `gluLookAt` und `gluPerspective` enthalten:

```
gluPerspective(fovy, aspect, zNear, zFar);
gluLookAt(eyex, eyey, eyez, centerx, centery, centerz, upx, upy, upz);
```

Mit den drei Koordinaten `eye*` und `center*` der Funktion `gluLookAt` kann der Vektor `d` berechnet werden, welcher durch die Mitten von near und far plane geht und damit unsere 'Sichtrichtung' vorgibt. Er ist in Abbildung 20 rot markiert. Nun werden wir die Längen der Kanten der near und far plane bestimmen:

```
Hnear = 2 * tan(fovy / 2) * zNear
Wnear = Hnear * aspect
```

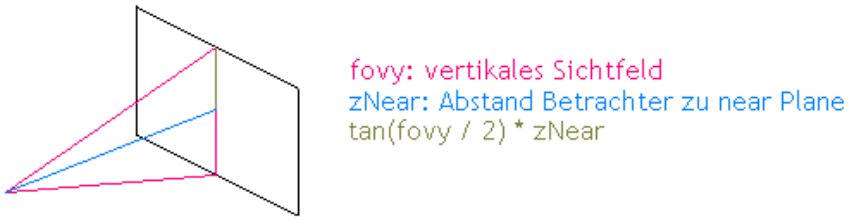


Abbildung 22: Berechnung der near und far plane

```
Hfar = 2 * tan(fovy / 2) * zFar
Wfar = Hfar * aspect
```

`Hnear` und `Wnear` geben die Höhe und Breite der near plane an und entsprechend `Hfar` und `Wfar` die der far plane. Die verwendeten Parameter sind aus den beiden Funktionen oben ersichtlich. Berechnet wird die Höhe einer der beiden Flächen, indem man den Tangens des halben, vertikelen Sichtwinkels (fovy / 2) mit dem Abstand der Fläche zum Betrachter multipliziert, was in Abbildung 22 dargestellt ist. Der resultierende Wert ist die halbe Höhe der Fläche, in Abbildung 22 bräunlich markiert. Um die Breite einer der beiden Flächen zu bestimmen, muss lediglich die Höhe mit dem `aspect`-Wert multipliziert werden, welcher das Verhältnis von Höhe zu Breite der Flächen angibt.

Mit diesen vier Werten können anschließend die vier Eckpunkte der near und far plane berechnet werden, womit alle Eckpunkte des Pyramidenstumpfes berechnet sind. Im folgenden stelle ich die Formel vor, mit der die Position der linken, oberen Ecke der far plane (`ftl`: far top left) berechnet werden kann. Die Berechnung der verbleibenden sieben Eckpunkte erfolgt analog.

```
ftl = eye + d * farDist + (up * Hfar/2) - (right * Wfar/2)
```

`ftl` ist der zu berechnende Punkt und `right` ist der Vektor, den man durch Kreuzprodukt des `up`- und des `d`-Vektors erhält und damit, anschaulich formuliert, vom Betracher aus 'nach rechts zeigt'. Nun, da alle Eckpunkte bekannt sind, können völlig analog zu 3.1 die Normalen der Flächen berechnet werden. Den zuvor angesprochenen Sattelpunkt D erhält man durch skalare Multiplikation der Normale einer der sechs Flächen mit einer der vier möglichen Eckpunkte. Mit der Normale und dem Sattelpunkt besitzt man nun genügend Informationen, um den Abstand eines Punktes zu den Flächen zu berechnen.

Die Berechnung des Abstandes eines Punktes r zu einer Fläche mit der Normalen n und dem Sattelpunkt D geschieht durch $n*r+D$, wobei * für skalare Multiplikation steht. Das Vorzeichen entscheidet nun, wie zuvor beschrieben, ob der Punkt in Normalenrichtung liegt oder nicht.

4 Fraktale Terraingenerierung

In dieser Arbeit wurde bisher auf einem Datensatz gearbeitet, welcher mit Hilfe von Satellitendaten erstellt wurde. Auch wenn jedes Gebiet der Erde als Datensatz geringer Auflösung frei zur Verfügung steht, so ist es auch erwünscht fiktives Gelände zu erstellen, z.B. für Computerspiele. In den folgenden Unterkapiteln stelle ich eine Möglichkeit vor, wie man sich die geometrischen Eigenschaften von Fraktalen zur Erstellung von Gelände nützlich machen kann. In dem konkreten Beispiel geht es um die Generierung einer Küstenlinie. Die Erstellung anderer geometrischer Gebilde wie Berge und Täler funktioniert analog.

4.1 Theoretische Grundlagen

Es existieren unterschiedliche Arten von Fraktalen mit unterschiedlichen Eigenschaften. Lediglich die hohe Skaleninvarianz und Selbstähnlichkeit ist bei allen Fraktalen gleich. Zur Erstellung der angesprochenen Küstenlinie wird auf die nach seinem Entdecker benannte Koch-Kurve zurückgegriffen, welche eine einfache iterative Möglichkeit bietet, um komplexe, fraktale Muster zu erzeugen. Andere Arten von Fraktalen können ebenso verwendet werden, erzeugen jedoch abweichende geometrische Gebilde.

Bei der Koch-Kurve wird jede vorhandene Strecke durch einen Streckenzug aus vier identisch langen Strecken ersetzt. Jede der vier Teilstrecken hat dabei eine Länge von einem Drittel der ursprünglichen Länge. Die Konstruktion erfolgt nach folgendem Schema:

```
zeichne erste Teilstrecke
drehe um 60 Grad in math. pos. Richtung
zeichne zweite Teilstrecke
drehe um 120 Grad in math. neg. Richtung
zeichne dritte Teilstrecke
drehe um 60 Grad in math. pos. Richtung
zeichne vierte Teilstrecke
```

Abbildung 23 zeigt die Koch-Kurve nach dem ersten Interationsschritt. Die grün markierten Flächen geben die drei vorkommenden Winkel von je 60°an. Es gilt zu beachten, dass sich die Gesamtlänge einer Strecke nach einem Iterationsschritt um einen Faktor von 1/3 erhöht. Daraus folgt, dass die Gesamtlänge nicht konvergieren kann und für beliebig viele Iterationsschritte gegen unendlich geht. Bereits nach 10 Iterationsschritten hat sich die Gesamtlänge um mehr als das 17fache erhöht! Abbildung 24 zeigt die ersten fünf Iterationsschritte einer gegebenen Strecke.

Um nun eine geschlossene Fläche zu erhalten, wählen wir zur Initialisierung nicht nur eine Gerade, sondern eine beliebige Anzahl an Strecken, welche eine Fläche begrenzen. Die Form des Gebildes ist abhängig von der Wahl der Ausgangsform. Abbildung 25 zeigt drei Beispiele, wobei links die Ausgangsform zu

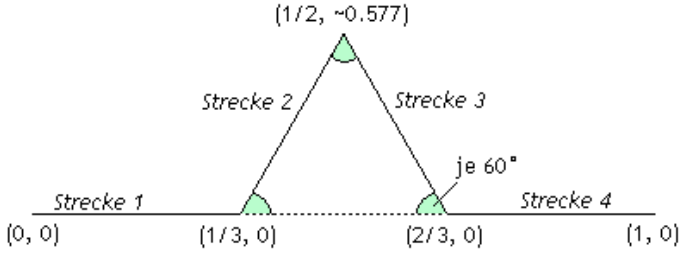


Abbildung 23: Koch-Kurve nach einer Iteration

sehen ist und rechts wird das dazugehörige Gebilde nach fünf Iterationsschritten dargestellt. Das oberste Gebilde ist auch als Kochsche Schneeflocke bekannt.

4.2 Implementierung

Um die erzeugten fraktalen Gelände unmittelbar verwenden und auch an Dritte weitergeben zu können, wird das Gelände im ARC/INFO ASCII Grid Format gespeichert, d.h. wir müssen ein zweidimensionales Feld der Größe $m \times n$ mit Höhendaten erzeugen. Die Informationen aus dem Header ergeben sich unmittelbar aus der Größe des Arrays.

In einem ersten Schritt wird ein zweidimensionales, fraktales Gebilde erzeugt, welches wie unsere spätere Insel aus der Vogelperspektive aussehen soll. In dem konkreten Beispiel habe ich mich für das mittlere Gebilde aus Abbildung 25 entschieden. Gespeichert werden bei der Erstellung des fraktalen Gebildes die einzelnen Linien, wobei für jede Linie die 2D-Koordinaten des Start- und des Endpunktes, sowie der Winkel der Geraden benötigt werden. Die resultierende Struktur einer Linie sieht wie folgt aus:

```
struct line
{
double startX, startY;
double endX, endY;
double angle;
};
```

In der Initialisierungsphase muss nun manuell die Ausgangsform erzeugt werden, in unserem Fall das Quadrat. Im folgenden wird angenommen, dass `Strecke 1` die gegebene Strecke ist, auf welcher der Iterationsschritt durchgeführt werden soll. Die Namensgebung der vier Strecken kann in Abbildung 23 nachgeschlagen werden.

Zuerst werden mehrfach benötigte Werte berechnet: `lenX` und `lenY` geben jeweils ein Drittel des Abstandes des Start- und Endpunktes in x- und y-Richtung

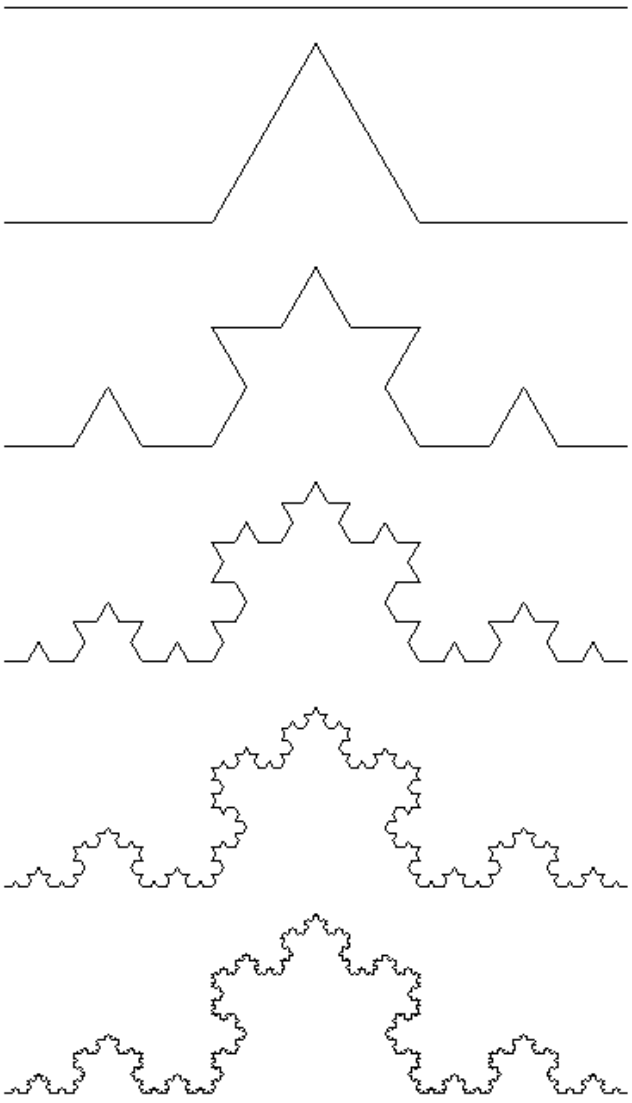


Abbildung 24: Koch-Kurve nach den ersten fünf Iterationen

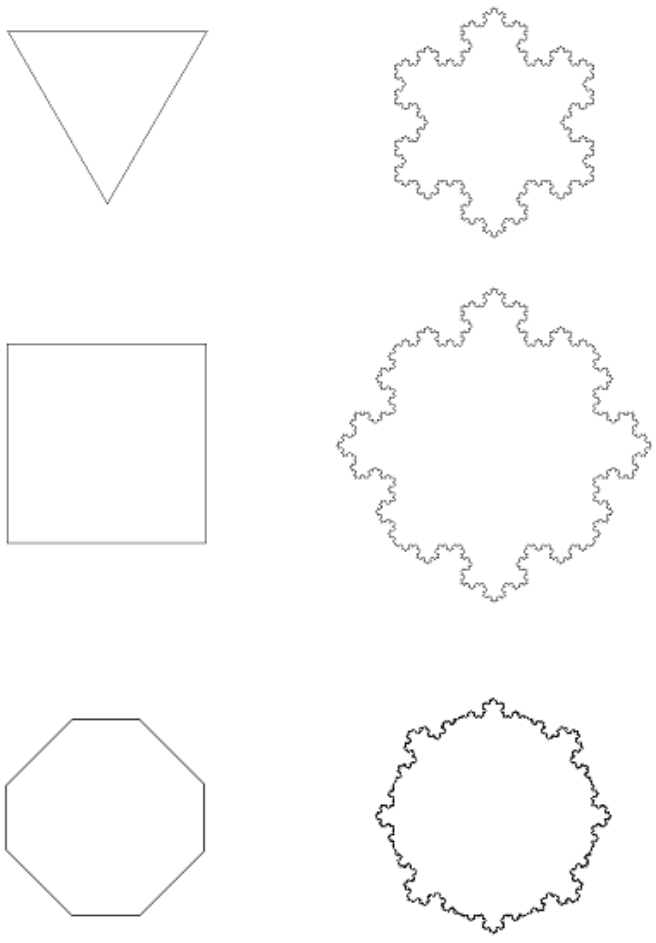


Abbildung 25: links: Ausgangsform, rechts: Koch-Kurve nach 5 Iterationen

an. Der Wert `length`, der sich mit dem Satz des Pythagoras berechnen lässt, beschreibt die Länge der Strecke vom Start- zum Endpunkt.

```
double lenX   = (strecke1.endX-strecke1.startX)/3;
double lenY   = (strecke1.endY-strecke1.startY)/3;
double length = sqrt(pow(lenX, 2) + pow(lenY, 2));
```

Nachdem diese Werte berechnet wurden, werden als nächstes die Eckpunkte der vier Teilstrecken berechnet. Strecke 4 verläuft parallel zur Ausgangsstrecke und hat daher den gleichen Winkel. Ihr Startpunkt beginnt nach 2/3 der Länge der Ausgangsstrecke und endet im Endpunkt von Strecke 1.

```
strecke4.angle  = strecke[i].angle;
strecke4.startX = 2*lenX+strecke[i].startX;
strecke4.startY = 2*lenY+strecke[i].startY;
strecke4.endX   = strecke[i].endX;
strecke4.endY   = strecke[i].endY;
```

Die Berechnung der beiden Punkte für die zweite Strecke ist etwas komplexer. Der Winkel dieser Strecke ist um 60°höher als der Winkel von Strecke 1 und der Startpunkt von Strecke 2 liegt bei einem Drittel der Länge von Strecke 1. Um den Endpunkt zu bestimmen, muss die Ausrichtung der zweiten Strecke in x- und y-Richtung bestimmt werden, mit Hilfe des Kosinus und Sinus. Die Position des Endpunktes in x-Richtung ist des Startpunktes addiert mit dem Produkt aus dem Kosinus des Winkels und der Länge der Strecke. Die Rechnung für die y-Richtung erfolgt analog, jedoch durch Verwenden des Sinus anstatt des Kosinus.

```
strecke2.angle  = strecke[i].angle + 60;
strecke2.startX = lenX+strecke[i].startX;
strecke2.startY = lenY+strecke[i].startY;
strecke2.endX   = strecke[NumOfLines+(i*3)+1].startX
  + length * cos(strecke[NumOfLines+(i*3)+1].angle*PI/180);
strecke2.endY   = strecke[NumOfLines+(i*3)+1].startY
   + length * sin(strecke[NumOfLines+(i*3)+1].angle*PI/180);
```

Die nun verbleibenden zwei Teilstrecken 1 und 3 erhält man aus den beiden bereits berechneten Strecken. Der Startpunkt von Strecke 3 ist der Endpunkt von Strecke 2 und der Endpunkt liegt im Startpunkt von Strecke 4, d.h. Strecke 3 'verbindet' die Strecken 2 und 4. Bei der ersten Strecke muss lediglich der Endpunkt auf ein Drittel seines ursprünglichen Wertes gesetzt werden. Damit die fraktalen Gebilde korrekt berechnet werden und es zu keinen fehlerhaften Darstellungen kommt, müssen alle Strecken in einer Richtung (z.B. im Uhrzeigersinn) durchlaufen werden. Würden die Streckenenden nicht einheitlich durchlaufen werden, könnte es passieren, dass hinzukommende Teilstrecken aufgrund eines falschen Winkels nicht nach außen gerichtet sind. In Abbildung 26, links ist ein mögliches Beispiel dargestellt, was passiert, wenn nicht alle Kanten einheitlich durchlaufen werden. Ein weiteres Problem kann auftreten, wenn das

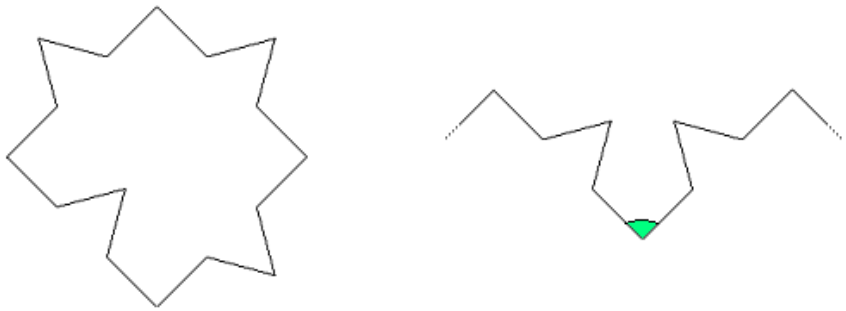


Abbildung 26: links: fehlerhafte Kante
rechts: Ausschnitt eines konkaven Gebilde

Ausgangsgebilde keinen konvexen Linienzug hat. In Abbildung 26, rechts ist ein Ausschnitt eines konkaven Linienzugs dargestellt. Um zu vermeiden, dass sich zwei Strecken schneiden, muss der Winkel zwischen jeder Strecke mindestens 60°betragen. Ist der Winkel kleiner, kann es passieren, dass sich Strecken nach einer späteren Iteration schneiden. In Abbildung 26 ist der kritische Winkel grün markiert.

Nun wurde ein zweidimensionales, fraktales Gebilde erzeugt, von dem wir Informationen über seine Kanten haben, welche in einem Array der Struktur `line` gespeichert werden. Damit wir jedoch das fraktale Gebilde in der Datei abspeichern können, müssen wir wissen, welche Punkte der Karte innerhalb des fraktalen Musters liegen und welche außerhalb. Da wir mit einem bestimmten Fehlerschwellwert verfeinern, müssen die Punkte innerhalb des Gebildes einen höheren Wert haben, als die Punkte außerhalb des Musters. Anschaulich hat die Umgebung um die Insel die Höhe von 0m über dem Meeresspiegel und die Landmasse der Insel einen höheren Wert, da sie aus dem Meer ragt. Die Wahl der Höhenwerte ist abhängig von der Verwendung des Gebildes und wird hier nicht weiter untersucht. Betrachtet wird im folgenden eine effiziente Methode, um zu überprüfen, welche der Punkte innerhalb des fraktalen Musters liegen und welche außerhalb.

Die Frage, ob ein Punkt in unserem Gebilde liegt, wird in zwei Schritte aufgeteilt: während der Initialisierungsphase werden alle Punkte innerhalb der Ausgangsform ermittelt und ihr Höhenwert erhöht. Da in der Regel einfache Objekte mit konvexen Linienzügen, wie Rauten oder Rechtecke, gewählt werden, stellt dies kein Problem dar. Damit reduziert sich der Test während den Iterationen auf das Problem, dass überprüft werden muss, ob sich Punkte in den neu hinzugekommenen Dreiecken befinden und gegebenenfalls müssen diese Höhenwerte erhöht werden. Der im folgenden beschriebene Test muss für jedes Dreieck jedes Iterationsschrittes angewendet werden. Zu Beginn ermitteln wir die drei Eckpunkte A, B und C des Dreiecks. Diese Punkte erhalten wir

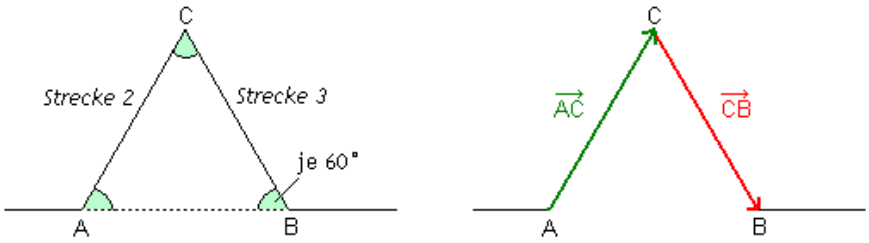


Abbildung 27: Bennenung innerhalb eines Dreiecks

unmittelbar durch Auslesen der Strecken 2 und 3. Der Startwert von Strecke 2 ist Punkt A, der Endwert von Strecke 2 bzw. der Startwert von Strecke 3 ist Punkt C und Punkt B ist der Endwert von Strecke 3 (siehe Abbildung 27, links). Um zu überprüfen, ob sich ein Punkt innerhalb des Dreiecks befindet, wird getestet, ob sich der Punkt als Linearkombination der beiden Vektoren $\vec{AC}$ und $\vec{CB}$ darstellen lassen kann. Diese beiden Vektoren, in Abbildung 27 rechts rot und grün eingezeichnet, sind linear unabhängig. Zur einfacheren Notation gilt im folgenden: $\vec{a} = \vec{AC}$ und $\vec{b} = \vec{CB}$. Die Linearkombination lässt sich nun wie folgt aufstellen: $P = r * \vec{a} + s * \vec{b}$, wobei $r, s \in \mathbb{R}$. Ist diese Gleichung lösbar und die beiden Koeffizienten r und s beide positiv und kleiner oder gleich 1, so liegen die Punkte innerhalb des Dreiecks. Um die Gleichung zu lösen, wird das aus der Linearkombination resultierende Gleichungssystem betrachtet:

$$\begin{pmatrix} P_x \\ P_y \end{pmatrix} = r * \begin{pmatrix} a_x \\ a_y \end{pmatrix} + s * \begin{pmatrix} b_x \\ b_y \end{pmatrix}$$

Stellt man das Gleichungssystem um, so erhält man für den Koeffizienten $s = \frac{P_y - \frac{a_y}{a_x} * P_x}{b_y - \frac{a_y}{a_x} * b_x}$ und durch Einsetzung in das Gleichungssystem erhält man umgehend den Koeffizienten $r = \frac{P_x - b_x * s}{a_x}$. Sind nun beide Koeffizienten größer oder gleich 0 und kleiner oder gleich 1, so liegen diese Punkte innerhalb des Dreiecks und der zuvor bereits abgedeckten Fläche. Der Höhenwert dieser Punkte kann nun erhöht werden und in der Datei abgespeichert werden.

In der folgenden Abbildung 28 ist im unteren Teil ein mögliches Resultat fraktaler Terrainmodellierung zu sehen. Als Ausgangsform wurde hier ein Quadrat gewählt und vier mal iteriert. Zur Veranschaulichung der Höhenunterschiede wurde auf *Height-dependent Coloring* zurückgegriffen, wobei die Insel grün und das umliegende Gebiet blau gefärbt wurde. Im oberen Teil der gleichen Abbildung ist ein Ausschnitt einer Küste einer solchen Insel mit den Verfeinerungen zu sehen. Deutlich zu erkennen ist die hohe Auflösung der Verfeinerung an der Küste und die grobere Vermaschung im Inneren der Insel. Zur Verdeutlichung der Unterschiede der Auflösung der Verfeinerungen wurde bei der Vermaschung auf die 'power of 2'-Regel verzichtet.

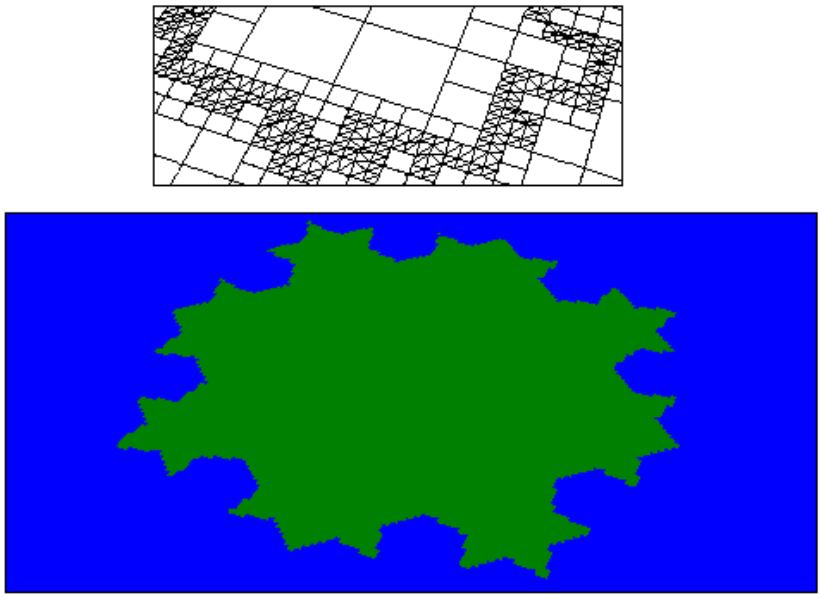


Abbildung 28: Insel mit fraktaler Küstenlinie

5 Geländemodifikationen

In alten wie auch aktuellen Grafikanwendungen, etwa Videospielen, ist das Problem bekannt, dass es nahezu kein dynamisch veränderbares Gelände gibt. Krater von Bomben in Flugsimulatoren werden als zweidimensionale Textur auf das Terrain abgebildet. Ein weiteres Beispiel sind Reifenspuren.

In diesem Kapitel widme ich mich der Fragestellung, wie man ein gegebenes Gelände so verändern kann, dass dreidimensinale Geländemodifikationen möglich werden. Als konkretes Fallbeispiel wird hier auf einen Krater eingegangen und mit seiner Hilfe der Algorithmus verdeutlicht. Andere Geländemodifikationen funktionieren nach dem gleichen Prinzip. Es sind jedoch nicht alle Geländeveränderungen sinnvoll. Während große Meteoritenkrater deutlich sichtbare Veränderungen am Gelände bedeuten, hätten Reifen- oder Kettenspuren nur minimalen Einfluss auf das Gelände, da die Tiefe eines Kraters erheblich größer ist, als die einer Reifenspur. Desweiteren wären für Reifenspuren sehr feine Verfeinerungen notwendig, was ein starkes Nachverfeinern des Geländes bedeuten würde, aufgrund der 'power of 2'-Regel. Im folgenden wird angenommen, dass ein Gelände vorliegt, z.B. eines, welches nach den bisherigen Kapiteln erstellt und verfeinert wurde.

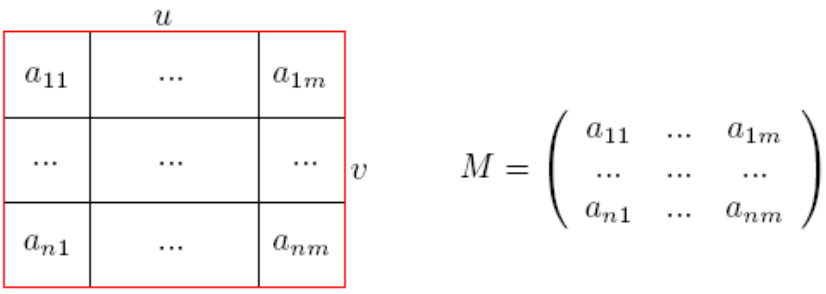


Abbildung 29: begrenzende Fläche (rot) mit Modifikationsmatrix M

5.1 Modifikationsmatrix

Bei der Modifikation eines bereits existierenden Geländes wird die Position der Modifikation bestimmt oder festgelegt und anhand dieser Position werden die Höhenwerte der Quadrate in einem Bereich um diese Position verändert. Die Werte, mit denen die Höhenwerte manipuliert werden, werden in einer Matrix gespeichert, der Modifikationsmatrix. Es wird ein rechteckiges Feld mit den Kantenlängen u und v angenommen, welches den Krater begrenzt. Bereiche, die außerhalb dieses rechteckigen Feldes liegen, müssen nicht modifiziert werden.

Der Bereich innerhalb der rechteckigen Fläche wird nun mit den Werten einer $m \times n$-Matrix M verändert. Dabei wird das rechteckige Feld in $m * n$ Teilfelder zerlegt und jedes Feld wird mit dem entsprechenden Eintrag der Matrix

$$M = \begin{pmatrix} a_{11} & \dots & a_{1m} \\ \dots & \dots & \dots \\ a_{n1} & \dots & a_{nm} \end{pmatrix}$$

modifziert. Die Größe der Teilflächen beträgt $u/m * v/n$. Sind die Matrixeinträge negativ, so würde sich das Gelände an den entsprechenden Stellen senken, bei positiven Werten erhöhen. Abbildung 29 zeigt zur Veranschaulichung ein rechteckiges Feld mit den Kantenlängen u und v, die Matrix M und die Felder, welche den Matrixeinträgen entsprechen.

Die Werte, mit denen die Modifikationsmatrix gefüllt wird, ist abhängig von der Art der Modifikation. Es stellt sich nun die Frage, wie die Matrixeinträge für unseren Krater berechnet werden können. Als Grundlage für die Form des Kraters habe ich eine Standardnormalverteilung mit der Formel

$$\varphi_{0;1}(x) = \frac{1}{\sqrt{2\pi}} \cdot e^{-\frac{1}{2}x^2}$$

gewählt. Diese hat eine vereinfachte Form eines Kraters, wenn die y-Werte mit -1 multipliziert werden. Die Funktion φ weist jedem x-Wert eine Höhe, bzw. Tiefe zu. Um nun die Einträge der zweidimensionalen Matrix M zu berechnen,

wird angenommen, dass x in der Matrixmitte $a_{0.5n,0.5m}$ seinen Ursprung hat und der Abstand eines Matrixeintrages zur Matrixmitte dem Wert von x entspricht, d.h. es folgt:

$$a_{ij} = \varphi(\sqrt{(i - 0.5n)^2 + (j - 0.5m)^2})$$

Die Matrixeinträge können nun optional noch mit einem konstanten Wert k multipliziert werden, um die Maximaltiefe k des Kraters festzulegen.

5.2 Datenstruktur

Um die für unseren Krater relevanten Daten zu speichern, wird die folgende Struktur `Caldera` verwendet. Stukturen für andere Geländemodifikationen haben einen analogen Aufbau.

```
struct Caldera
{
Vertex corner[4];
int range;
int height[9][9];
};
```

Das Array `corner` beinhaltet die Raumkoordinaten der vier Eckpunkten der begrenzenden Rechtecksfläche (in Abbildung 29 rot markiert), `range` bezeichnet den Abstand eines Eckknotens zur Flächenmitte und das zweidimensionale `height`-Array ist unsere Modifikationsmatrix mit $9 * 9$ Einträgen. Es hat sich gezeigt, dass auch für große Kratertiefen k eine 9×9-Matrix ausreicht und die äußersten Einträge 0 sind.

Die Koordinaten der vier Eckpunkte und der Abstand eines Eckpunktes zum Mittelpunkt der Fläche werden für die Auswahl der zu modifizierenden Quadrate benötigt und diese Quadrate werden dann im finalen Schritt mit den Werten des height-Arrays modifziert.

5.3 Geländemodifikation

Ist die Modifikationsmatrix erstellt und sind die relevanten Daten berechnet, so kann die eigentliche Modifikation des Geländes vorgenommen werden. Diese setzt sich aus zwei Schritten zusammen: zuerst müssen die zu modifizierenden Quadrate ermittelt werden und wenn diese bestimmt sind, müssen die Werte entsprechend der Modifikationsmatrix manipuliert werden. Gegebenenfalls ist vor der Manipulation der Höhenwerte ein Nachverfeinern der Quadrate notwendig.

Der erste Schritt besteht nun aus der Auswahl der Quadrate. Dazu wird jedes Quadrat, welches ein Blattknoten in der *Quadtree*-Hierarchie ist, auf seine Lage zur zu modifzierenden Fläche hin untersucht. Dabei gibt es drei Möglichkeiten,

wie das Quadrat räumlich liegen kann: vollständig innerhalb der zu modifizierenden Fläche, vollständig innerhalb dieser Fläche oder durch überschneidung teilweise innerhalb der Fläche. Zur überprüfung der Lage geht man nach folgendem Schema vor: zuerst wird überprüft, ob sich das Quadrat innerhalb der Fläche befindet. Dazu überprüft man jeweils die Koordinaten des oberen, linken und des unteren, rechten Eckpunktes des Quadrates und der zu begrenzenden Fläche. Sind die x- und z-Koordinaten des oberen, linken Eckpunktes größer als die des linken, oberen Eckpunktes der zu begrenzenden Fläche und sind die x- und z-Koordinaten des rechten, unteren Eckpunktes des Quadrats kleiner als die entsprechenden Koordinaten der Fläche, so liegt das Quadrat vollständig in der zu begrenzenden Fläche. Die beiden anderen Fälle, dass das Quadrat vollständig außerhalb liegt oder sich mit der Fläche überschneidet kann mit einem Test überprüft werden. Dazu wird der Abstand jedes Eckpunktes des Quadrats mit jedem Eckpunkt der begrenzenden Fläche bestimmt. Der minimale Abstand, der hier ermittelt wird, wird mit dem `range`-Wert (siehe oben) verglichen. Ist der minimale Abstand kleiner oder gleich des `range`-Wertes, so liegt das Quadrat innerhalb der Reichweite der zu modifzierenden Fläche und muss modifziert werden. Ist der Abstand größer als der `range`-Wert, muss das Quadrat nicht weiter berücksichtigt werden.

Nachdem die zu modifizierenden Quadrate ermittelt sind, wird im zweiten Schritt überprüft, ob sie nachverfeinert werden müssen, damit die Approximation des Kraters hinreichend genau wird. Dazu wird zuerst die Kantenlänge a des entsprechenden Quadrats ermittelt und anschließend wird die Kantenlänge l einer Teilfläche der zu verfeinernden Fläche ermittelt (siehe dazu die rot umrandete Fläche im Abbildung 29). Wie bereits in 5.1 gezeigt, gilt $l = u/m$. In unserer Struktur aus 5.2 haben wir eine 9×9-Matrix, d.h. $m = 9$. u Ist die Kantenlänge der begrenzenden Fläche und kann entweder durch Anwendung des Satzes des Pythagoras aus dem `range`-Wert berechnet werden, oder man bestimmt alternativ den Abstand zweier Eckpunkt dieser Fläche mit Hilfe des `corner`-Arrays der gleichen Struktur. Sind sowohl l, als auch a bekannt, so werden die beiden Kantenlängen mit einander verglichen und gilt $a > l$, so muss das Quadrat verfeinert werden, damit die Matrix-Einträge 1:1 auf die Quadrate übertragen werden können.

Um nun die Höhenwerte der, gegebenenfalls verfeinerten, Quadrate manipulieren zu können, muss nun bestimmt werden, welcher Matrixeintrag zu dem entsprechenden Quadrat gehört. Um die Koordinaten des Matrixeintrags zu bestimmen, subtrahiert man die x- und z-Koordinaten des Quadrats mit den Koordinaten des linken, oberen Eckpunktes der begrenzenden Fläche. Diese sind, zur Erinnerung, im `corner`-Array gespeichert. Man erhält dadurch den Abstand des Quadrats zum linken oberen Eckpunkt in x- und z-Richtung. Um nun die Koordinaten des Matrixeintrags zu erhalten, müssen der Abstand in x- und z-Richtung durch l, die Kantenlänge einer Teilfläche der begrenzenden Fläche, dividiert werden. Als Resultat erhält man die Koordinaten des Matrixeintrags a_{ij} und dieser Wert wird nun zum Höhenwert des Quadrats addiert, wodurch

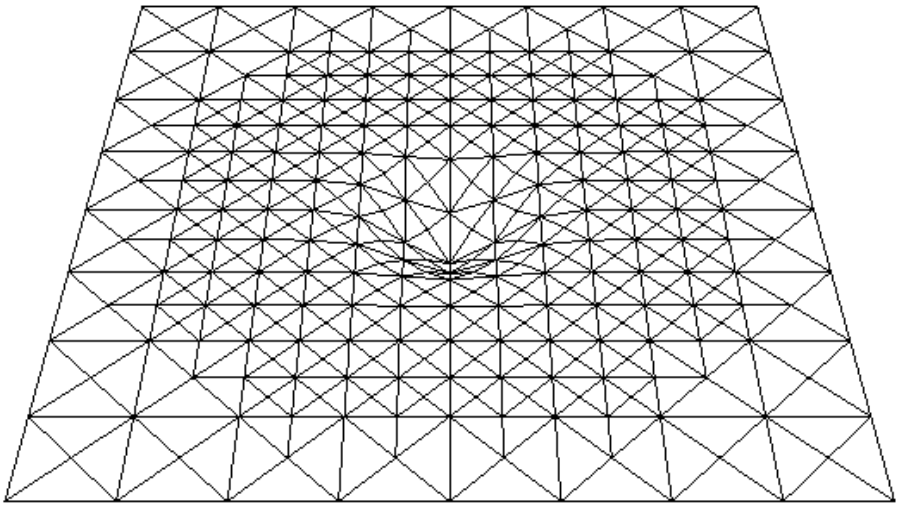


Abbildung 30: Anwendungsbeispiel

sich eine Hebung oder Senkung des Geländes ergibt.

Als finalen Schritt wird noch einmal die Funktion zur Nachverfeinerung aufgerufen, da während der Geländemodifikation Verfeinerungen geschehen sein könnten und es muss für eine korrekte Triangulierung sichergestellt werden, dass die 'power of 2'-Regel, siehe 2.4.3, erfüllt wird.

5.4 Anwendungsbeispiel

Als konkretes Anwedungsbeispiel wird ein Krater der Maximal-Tiefe 1.000 mit dem obigen Verfahren erzeugt. Als Mittelpunkt der Modifikation wird das Zentrum einer Fläche gewählt und die Kantenlänge der begrenzenden Fläche entspricht der Hälfte der Gesamtfläche des ebenen Geländes. Die Modifikationsmatrix sieht wie folgt aus:

$$M = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 6 & 11 & 6 & 1 & 0 & 0 \\ 0 & 1 & 18 & 82 & 135 & 82 & 18 & 1 & 0 \\ 0 & 6 & 82 & 367 & 606 & 367 & 82 & 6 & 0 \\ 0 & 11 & 135 & 606 & 1000 & 606 & 135 & 11 & 0 \\ 0 & 6 & 82 & 367 & 606 & 367 & 82 & 6 & 0 \\ 0 & 1 & 18 & 82 & 135 & 82 & 18 & 1 & 0 \\ 0 & 0 & 1 & 6 & 11 & 6 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Die resultierende Verfeinerung des Geländes ist in Abbildung 30 dargestellt.

6 Zusammenfassung und Ausblick

In dieser Arbeit wurden eingangs Methoden zur Generierung eines Terrains vorgestellt. Hierzu wurden zuerst geeignete Basisalgorithmen gewählt und anschließend ein Verfeinerungskriterium zur Reduzierung des Ausgangsdatensatzes präsentiert. Nachdem die theoretischen Fragen damit abgedeckt waren, wurden effiziente Datenstrukturen entwickelt, um das Terrain und die Hierarchie zu speichern. Als letzter Punkt stand noch die Verfeinerung mit anschließender Triangulierung an. In diesem Zusammenhang wurden auch mögliche Probleme der Triangulierung und Lösungen hierfür genannt.

Nachdem das Gelände hinreichend gut approximiert und trianguliert wurde, folgte das Rendern des Terrains. Dabei wurde auf die Berechnung der Normalen eingegangen, welche für eine korrekte Beleuchtung notwendig sind. Außerdem wurden verschiedene Arten der Texturierung vorgestellt und mit dem *View Frustum Culling* eine Methode gezeigt, um die Rechenzeit zu minimieren.

In den letzten beiden Aufgabenstellungen wurde die Frage behandelt, wie man alternativ Gelände modellieren kann, falls kein Ausgangsgelände vorliegt. Dazu wurde in die fraktale Terrainmodellierung eingeführt. Die Geländemodifikationen bilden eine Möglichkeit, um während der Laufzeit lokale Änderungen am Terrain vornehmen zu können. Das vielfach genannte Beispiel hierfür sind Krater in Videospielen.

Es bleiben jedoch noch drei wesentliche Aufgabenstellungen offen. Dazu gehören neben einer GPU-Implementierung dieser Arbeit auch eine Möglichkeit, um alle Dreiecke des Geländes mit einem einzigen *Triangle Strip* zu zeichnen. Beide Aufgaben würden die Leistung der Anwendung maßgeblich steigern. Desweiteren werden bereits existierende Arbeiten zum Thema fraktaler Terrainmodellierung präsentiert. Diese sollen unter anderem als Ausblick auf Erweiterungen des in dieser Arbeit vorgestellten Modellierungsverfahrens bieten.

GPU-Implementierung Alle Rechenoperationen und Arbeitsschritte, die in dieser Arbeit behandelt wurden, finden zu Lasten der CPU statt. Eine Verbesserung der Leistung erhält man dadurch, dass man möglichst viele Operationen auf die GPU verlagert. Dadurch erhält man nicht nur eine geringere Auslastung der CPU, sondern auch geringere Rechenzeiten.

[HOPPE 05] stellt eine GPU-basierte Implementierung von *Geometry Clipmaps* aus [HOPPE 04] vor. Beim Ansatz der *Geometry Clipmaps* werden eingebettete *Regular Grids* verwendet, welche eine nach außen abnehmende Auflösung besitzen, wobei alle Grids gleicher Größe als ein Level aufgefasst werden. Bewegt sich der Betrachter, so werden die Level (teils) neu berechnet. Dadurch erhält man *view-dependent refinement*, da stets der direkt um den Betrachter liegende Bereich die maximale Auflösung besitzt. In der GPU-Implementierung von [HOPPE 05] wurde die (x,z)-Geometrie der Level durch eine kleine Men-

ge an konstanten *Vertex* und *Index Buffern* vordefiniert. Die Höhenwerte jedes Levels wurden in je einer 2D-Textur mit einem Kanal gespeichert und die Normalen wurden in einer 4-Kanal-Textur gespeichert. Die Daten befinden sich alle im Videospeicher und die meisten Operationen werden nun auf der GPU durchgeführt. Die folgende Tabelle aus [HOPPE 05] zeigt drei der Operationen, welche auf die GPU ausgelagert wurden: Upsampling[4], Synthesis[5] und die Berechnung der Normalen. Diese Operationen müssen nicht mehr von der CPU durchgeführt werden, wodurch diese entlastet wird und die Operationen werden außerdem schneller durchgeführt.

	CPU-basiert	GPU-basiert
Upsampling	3ms	1ms
Synthesis	3ms	≈0ms
Berechnung der Normalen	11ms	0,6ms

[HOPPE 04] zeigt mit der GPU-Implementierung der *Geometry Clipmaps* den Leistungsgewinn, der durch die Auslagerung der Arbeitsschritte und Daten auf die GPU erziehlt werden kann. Eine GPU-Implementierung des in dieser Arbeit vorgestellten Verfahrens scheint daher eine sinnvolle Erweiterung zu sein.

Generalized RQT Triangle Strip Einen weiteren Leistungsgewinn erhält man durch eine geschickte Triangulierung des Geländes. Nach dem in 2.4. vorgestellten Verfahren, welches im wesentlichem der *Restriced Quadtree Triangulation* aus [PAJAROLA 07, Seite 4] entspricht, wird jedes Quadrat einzeln trianguliert. Der angesprochene Leistungsgewinn kann erreicht werden, indem alle Dreiecke des Geländes mit einem einzigen *Triangle Strip* gerendert werden. Die Frage hierbei ist, in welcher Reihenfolge die Dreiecke durchlaufen werden müssen, sodass kein Dreieck zweimal besucht wird. Es wurde bewiesen, dass ein Gelände, welches die 'power of 2'-Regel erfüllt, mit einem einzigen *Triangle Strip* gerendert werden kann.

Fraktale Terrainmodellierung Wie ich bereits in Kapitel 4 schrieb, habe ich lediglich die Generierung einer Küstenlinie mit Fraktalen betrachtet. Im folgenden werde ich kurz auf zwei Projekte eingehen, welche Fraktale genutzt haben, um komplexe Terrains zu generieren, welche in Simulationen und Computerspielen eingesetzt werden können.

[DistFracEx3D], das erste Projekt, stammt von einem Studenten aus Fulda. Nach Angaben des Autors war es sein Ziel 'komplexe Sachverhalte aus der Mathematik und Informatik mit ihrer praktischen Anwendung zu verknüpfen'. Der Benutzer des Programms wählt zu Beginn eine Mandelbrot- oder Juliamenge und das Programm berechnet mittels verteilter Berechnung die fraktale Landschaft. Um die in seiner Anwendung erzeugte Landschaft realistischer zu

[4] Um realistischeres Gelände zu erzeugen, wird das Gelände mit einem Rauschen versehen
[5] Zuweisung eines Höhenwertes zu einer (x,z)-Koordinate

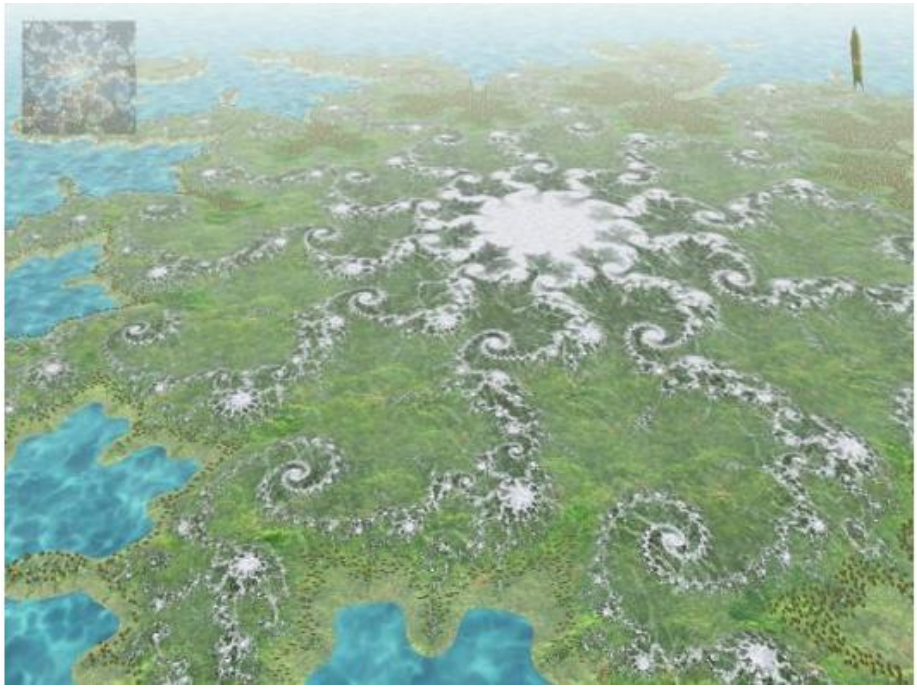


Abbildung 31: Quelle: [DistFracEx3D]

gestalten, wendet er verschiedene Normalisierungs- und Glättungsfilter auf die Landschaft an. Zuletzt wird die Landschaft u.a. noch um Nebel, Wasser und eine fraktale Wolkendecke ergänzt. Abbildung 31 zeigt eine Gelände, welches mit diesem Projekt erstellt wurde.

Das zweite Projekt ist *Terrabrush* von [cyberFunks]. *Terrabrush* ist ein Landschaftsgenerator, mit dem man optisch ansprechendes Gelände für grafische Anwendungen erzeugen kann. [BLUM] bietet eine ausführliche Einleitung in das Programm. *Terrabrush* nutzt fraktale Algorithmen, um die Gelände zu erzeugen. Dem Benutzer stehen dabei eine Reihe an fraktalen Algorithmen zur Verfügung, aber die genaue Verwendung und Art der fraktalen Algorithmen ist nicht dokumentiert. Abbildung 32 zeigt ein Gelände, welches mit *Terrabrush* erzeugt wurde.

Diese beiden Projekte zeigen, dass Kapitel 4 noch ausgebaut werden kann, um nicht nur die Küstenlinie, sondern das vollständige Terrain mit Fraktalen zu modellieren.

Abbildung 32: Quelle: [BLUM]

Literatur

[LUEBKE 02] LUEBKE, D. Level of Detail for 3D Graphics, 2002, pp. 185-215

[HOPPE 04] HOPPE, H., LOSASSO, F. Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids, 2004

[HOPPE 05] HOPPE, H., ASIRVATHAM, A. Terrain Rendering Using GPU-Based Geometry Clipmaps, 2005

[HOPPE 98] HOPPE, H. Smooth View-Dependent Level-of-Detail Control and its Application to Terrain Rendering, 1998

[PAJAROLA 07] PAJAROLA, R., GOBBETTI, E. Survey on Semi-Regular Multiresolution Models for Interactive Terrain Rendering, In *The Visual Computer 23*, 2007, pp. 583–605

[KILGARD 03] MARK, R., GLANVILLE, R., AKELEY, K., KILGARD, M. Cg: A system for programming graphics hardware in a C-like language, In *Proceedings of SIGGRAPH*, 2003

[LINDSTROM 01] LINDSTROM, P., PASCUCCI, V. Visualization of Large Terrains Made Easy, In *Proceedings of IEEE Visualization*, 2001

[MANDELBROT 91] MANDELBROT, B., Die fraktale Geometrie der Natur, 1991

[DistFracEx3D] http://graphenreiter.de/distfracex3d.html, gesichtet am 03.07.2008 um 21:04 Uhr

[cyberFunks] http://www.cyberfunks.de/page/terrabrush/index.htm, gesichtet am 03.07.2008 21:26 Uhr

[BLUM] http://www.matthias-blum.de/Terragen/tuts/BrushTut/tbkey.html, gesichtet am 03.07.2008 21:26 Uhr