Georg-August-Universität
Göttingen
Zentrum für Angewandte
Informatik

ISSN 1612-6793
Nummer ZAI-BSC-2015-10

Bachelorarbeit

im Studiengang "Informatik und Deutsche Philologie (Zwei-Fächer-Bachelor)"

Modelling and rendering landscapes with the language XL: Employing multi-scale graph rewriting rules for level-of-detail representations

Robert Lodahl

am Lehrstuhl für

Computergrafik und Ökologische Informatik

Bachelor- und Masterarbeiten
des Zentrums für Angewandte Informatik
an der Georg-August-Universität Göttingen

16. April 2015

Georg-August-Universität Göttingen
Zentrum für Angewandte Informatik

Goldschmidtstr. 7
37077 Göttingen
Germany

Tel. +49 (5 51) 39-1 72 010
Fax +49 (5 51) 39-1 46 93
Email office@informatik.uni-goettingen.de
WWW www.informatik.uni-goettingen.de

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Göttingen, den 16. April 2015

Bachelor-Arbeit

Modelling and rendering landscapes with the language XL: Employing multi-scale graph rewriting rules for level-of-detail representations

Robert Lodahl

16. April 2015

Institut für Informatik

Betreut durch:
Prof. Dr. Kurth
Prof. Dr. Sax

Zusammenfassung

Die Darstellung von Landschaften, unterstützt durch die Möglichkeiten von Detaillierungsgrad-Technologien, welche ein schnelles Rendering auf derzeitiger Hardware erlauben, soll in dieser Arbeit für die Open-Source Modellierungsplattform GroIMP erreicht werden. Dabei soll die Landschaft nicht als Mesh betrachtet werden, sondern als veränderbarer und multiskalarer Graph unter der Programmier- und Graphen-Sprache XL. Jeder genauere Detaillierungsgrad entspricht dabei einer Skalierung des Graphen. Es soll dazu beigetragen werden einen generellen Ansatz für die Erzeugung, Auswertung und Darstellung multiskalarer Graphen zu schaffen, um Ökosysteme vollständig von einer globalen Ansicht bis hin zu einer metabolischen Größe zusammenhängend modellieren zu können. Weiterhin sollen durch die Ergebnisse dieser Arbeit die Einsatzmöglichkeiten der Plattform GroIMP erweitert werden.

Inhaltsverzeichnis

1 Einführung **4**
1.1 Motivation 4
1.2 Zielsetzung 5
1.3 Struktur der Arbeit 6

2 Grundlagen **8**
2.1 Terminologie 8
2.1.1 Landschaftsmodellierung 8
2.1.2 Eckpunkte, Kanten und Flächen 8
2.1.3 Dreiecke 9
2.1.4 Polygon Mesh 9
2.1.5 Normalenvektor 10
2.2 Dateiformate 10
2.2.1 GeoTIFF 10
2.2.2 DTED 11

3 Rendering **12**
3.1 Preprocess- und Echtzeit-Rendering 12
3.2 Rendering Pipeline 13

4 Höhenkarten und Detaillierungsgrade **17**
4.1 Höhenkarten 17
4.2 Detaillierungsgrad 19
4.2.1 Diskretes Level-of-Detail 20
4.2.2 Kontinuierliches Level-of-Detail 23
4.2.3 Sichtabhängiges Level-of-Detail 25
4.3 Hierarchische Detaillierungsgrad Strukturen 29
4.3.1 Binärbaum 30
4.3.2 Quadtree 31
4.4 Zusammenfassung 32

5 Die Graphensprache XL **34**
5.1 Anwender-Syntax 36
5.2 Architektur 38
5.3 GroIMP 41

6 Umsetzung **43**
6.1 Ziele und Voraussetzungen . . . 43
6.2 Konzept . . . 44
6.2.1 Import der Rasterformate . . . 44
6.2.2 Erstellung des Meshs . . . 46
6.2.3 Funktion der HLOD-Struktur . . . 47
6.3 Implementation . . . 51
6.3.1 LodQuadTree . . . 52
6.3.2 LodQuadNode . . . 56
6.4 Evaluation . . . 64
6.4.1 Möglichkeiten . . . 64
6.4.2 Leistungsbewertung . . . 65
6.5 Anwendung . . . 65
6.5.1 Installation . . . 66
6.5.2 Exemplarischer Anwendungsfall . . . 67
6.6 Zusammenfassung . . . 69

7 Zusammenfassung und Ausblick **71**

Abkürzungsverzeichnis **74**

Formelverzeichnis **75**

Abbildungsverzeichnis **76**

Programmcodeverzeichnis **77**

A Inhalte der DVD **78**

Literaturverzeichnis **79**

Kapitel 1

Einführung

Interaktive Anwendungen, die zur Visualisierung bestimmter Inhalte eine virtuelle Umgebung erschaffen, sind heute nicht mehr nur ein Standard für Anwendungen innerhalb des industriellen Sektors, wo sie zur Simulation neuer Technologien eingesetzt werden. Stattdessen finden Anwendungen, die virtuelle Umgebungen einsetzen, heute Einsatz in fast allen Software-Bereichen. Ein Beispiel hierfür sind Lernsoftwares oder Computerspiele. Der Grund für den steigenden Einsatz dieser Umgebungen liegt darin, dass der Benutzer der Anwendung nicht mehr länger eine passive Rolle einnimmt, wie zum Beispiel bei der Betrachtung eines Filmes oder eines Bildes. Stattdessen kontrolliert er das, was angezeigt wird. Durch die virtuelle Umgebung kann somit die Anschaulichkeit des in der Umgebung behandelten Themas stark erhöht werden.

1.1 Motivation

Gerade bei der Darstellung realistischer virtueller Umgebungen ist das Fachgebiet der Computergrafik seit seiner Entstehung von einem Konflikt geprägt [LRC+03]. Auf der einen Seite sollen extrem realistische, hoch detaillierte, virtuelle Umgebungen generiert und angezeigt werden. Auf der anderen Seite soll diese Anzeige in Echtzeit, mit hohen Bildwiederholungsraten, flüssig und mit möglichst niedrigen Hardware-Anforderungen geschehen. Mit dem Konzept der Detaillierungsgrade, bekannter unter dem Namen *level-of-detail*, existieren Techniken, die diesen Konflikt zumindest scheinbar lösen [CR11]. Durch Ausnutzung der Aufnahmefähigkeit des Nutzers werden nur die auffälligsten Objekte einer Umgebung im vollen Detail angezeigt. Alle anderen Elemente werden reduziert dargestellt, um somit die flüssige Darstellung in Echtzeit zu gewährleisten. Immer komplexer und größer werdende Modelle zeigen, dass die Techniken der Detaillierungsgrade auch heutzutage, trotz der stark erhöhten Hardwareleistungen noch immer notwendig sind.

Insbesondere bei der Anzeige von Landschaften, die leicht mehrere Quadratkilometer umfassen können, sind diese Techniken unabdingbar.

Auch aus theoretischer Sicht ist die Betrachtung der Funktionsweisen einiger Detaillierungsgrade interessant: gerade Techniken zur Darstellung von Landschaften mit mehreren Detaillierungsgraden lassen sich leicht in ein Problem der Graphentheorie transformieren. In diesem Problem entsprechen einzelne Knoten eines Graphen einzelnen Detaillierungsgraden eines Landschaftsmodells. Während über den Graphen traversiert wird, muss nun entschieden werden, welche der Knoten aktiviert werden. Ist ein Knoten aktiviert, wird für die Fläche, die er im Landschaftsmodell repräsentiert, der von ihm beschriebene Detaillierungsgrad angezeigt. Die Knoten des Graphens werden dabei in einer hierarchischen Form angeordnet, sodass die Wurzel des Graphen die am wenigsten detaillierteste Fassung des Modells darstellt.

Mit der Graphensprache XL und der implementierenden Plattform GroIMP existiert eine vorgefertigtes und leistungsfähiges Werkzeug zum Verwalten, Traversieren und Umschreiben von Graphen. Zudem besitzt die Plattform GroIMP noch keine eingebaute Lösung zum Anzeigen von Landschaften. Es bietet sich daher an, die Fähigkeiten von XL zu nutzen, um eine Landschaft mit mehreren möglichen Detaillierungsgraden im Sinne eines multi-skalaren Graphen zu verwalten und GroIMP somit, mit seinen eigenen Mitteln, um die Fähigkeit zu erweitern Landschaften anzuzeigen.

1.2 Zielsetzung

Ziel der Bachelorarbeit ist es, die Anwendungsmöglichkeiten der Open-Source Modellierungssoftware GroIMP zu erweitern. Zurzeit ist diese in der Lage, Vegetation unter realistischen Wachstumsbedingungen zu simulieren. Dies geschieht über den Einsatz von Wachstumsgraph-Grammatiken (RGG), welche mittels der Graphensprache XL unterstützt werden. Die Anwendung soll in der Form erweitert werden, dass sie neben der Vegetationsmodellierung auch Landschaften anzeigen kann. Bisher können nur sehr kleine Abschnitte eingefügt werden, die statisch sind und unabhängig von der Simulation existieren. Die Implementation dieser Arbeit soll dahingegen in der Lage sein auch größere Landschaften anzuzeigen, um die Erzeugung einer virtuellen Realität zu komplettieren. Damit dies möglich ist, muss die Landschaft mittels eines Detaillierungsgrad-Modells dargestellt werden. Hierbei sollen die Vorteile der Sprache XL genutzt werden, damit die Detaillierungsgrade jederzeit über einen Graph beschreibbar bleiben. Zudem sorgt diese Technik dafür, dass Landschaften sowohl von Nutzern als auch von Entwicklern exakt wie

der restliche Teil des Programms genutzt werden können. Dies ist vor allem im Hinblick darauf interessant, dass das Detaillierungsgrad-Modell der Landschaften mit einem der Vegetation verbunden werden könnte, sodass die realistische Simulation beliebig großer Gebiete ermöglicht wird [OK12]. Damit auch reale Orte mittels GroIMP beschrieben werden können, soll die Erweiterung der Anwendung zudem in der Lage sein, Landschaften, die in Form von Höhenkarten in den verbreiteten Formaten GeoTIFF und DTED vorliegen, zu laden und darzustellen.

1.3 Struktur der Arbeit

Die vorliegende Arbeit ist wie folgt strukturiert: Im 2. Kapitel werden grundlegende Begriffe erklärt, die für das Verständnis der restlichen Arbeit relevant sind. Dazu gehören etwa Grundbegriffe der Computergrafik oder Begriffe, für die in der Fachliteratur keine feste Definition existiert. Auch sollen die beiden Format GeoTIFF und DTED präsentiert werden, deren Import durch die Anwendung gewährleistet werden soll.

Im 3. Kapitel wird der Prozess und die Bedeutung des Renderings in der Computergrafik erläutert. Besonders kritisch ist dieser Vorgang bei der Darstellung virtueller Umgebungen, da er hier zur Laufzeit eines Programms ausgeführt werden muss. Es wird erläutert, welche Methoden die Daten einer virtuellen Umgebung durchlaufen um zu einem anzeigbaren Bild umgewandelt zu werden.

Das 4. Kapitel erläutert, wie genau die Struktur von Höhenkarten im allgemeinen aussieht und wieso sich diese Struktur besonders für Landschaften anbietet. Weiterhin werden verschiedene Techniken zur Darstellung von Modellen mit verschiedenen Detaillierungsgraden vorgestellt, welche anschließend auf ihre Brauchbarkeit, für die effiziente Darstellung von Landschaften, bewertet werden.

Im 5. Kapitel wird kurz die Graphensprache XL vorgestellt und welche besonderen Möglichkeiten sie zum Arbeiten mit Graphen bietet. Auch die Anwendung GroIMP, die XL einsetzt, soll vorgestellt werden. Es wird geklärt werden, welche Voraussetzungen zur Realisierung der Zielsetzung herrschen.

Nach diesen Kapiteln der theoretischen Vorbereitung und Analyse wird sich das 6. Kapitel mit der Umsetzung befassen. Dazu wird das Konzept hinter der Implementation beschrieben werden. Die Implementation selbst wird daraufhin ausführlich dokumentiert werden. Dabei wird auch die Behandlung von Fehlern beschrieben werden, die nur im praktischen, nicht im theoretischen Bereich entstehen können. Daraufhin soll eine Beurteilung über

die Möglichkeiten der Erweiterung erfolgen und ein Leistungsbewertung desselben erstellt werden. Abschließend wird die Installation der Anwendungserweiterung erklärt werden und ein Einsatz am Beispiel der Göttinger Umgebung dargestellt werden.

Zuletzt wird das 7. Kapitel die wesentlichen Aspekte der Arbeit und die erreichten Ziele noch einmal zusammenfassen. Ebenfalls soll ein Ausblick gegeben werden, inwiefern sich die Implementation optimieren und weiterentwickeln lässt.

Kapitel 2

Grundlagen

Dieser Abschnitt soll einige Grundlagen klären, die für das Verständnis der restlichen Arbeit unumgänglich sind. Dazu gehört einerseits die Klärung von Begrifflichkeiten, andererseits die Präsentation der in dieser Arbeit verwendeten Dateiformate.

2.1 Terminologie

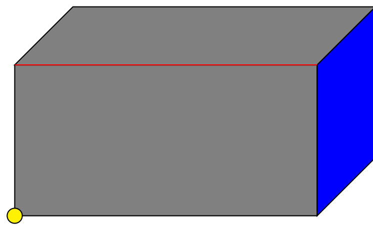
Im folgenden werden Begriffe erläutert werden, die zum größten Teil aus der Computergrafik stammen. Ihre Benennung ist, vor allem innerhalb des deutschen Sprachraums, nicht immer eindeutig und soll daher hier, für den Rest der Arbeit, gesetzt werden.

2.1.1 Landschaftsmodellierung

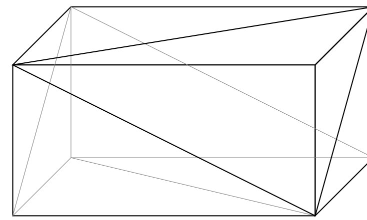
Unter dem Begriff der Landschaftsmodellierung werden in dieser Arbeit Techniken verstanden, die nötig sind, um eine Landschaft, die über eine Höhenkarte in einer Datei beschrieben wird, auszulesen, zu verarbeiten und dem Nutzer in Form einer geometrischen Repräsentation anzuzeigen. Es ist somit allein die Anzeige der Landschaft an sich gemeint und nicht die Modellierung darauf vorhandener Vegetation, Lebewesen oder Gewässer.

2.1.2 Eckpunkte, Kanten und Flächen

Die Eckpunkte eines Polyeders sind eine nichtleere Menge von Punkten des dreidimensionalen Raums. Die sind vermittels eines *geometrischen Graphs* miteinander verbunden,



(a) Ein simples Mesh



(b) Mesh im Drahtgittermodell

Abbildung 2.1: Ein simples Mesh. In (a) sind Fläche, Eckpunkt und Kante markiert, (b) zeigt die Dreiecke

wobei in diesem eine Menge von Verbindungsstrecken zwischen Punkten des Graphen definiert ist. Diese Verbindungsstrecken lassen sich als geometrische Kanten interpretieren. Ein Punkt ist genau dann Eckpunkt eines Polyeders, wenn er Endpunkt einer Kante ist, diese Kante Teil eines zyklischen Graphen (Polygon) ist und der Punkt mindestens zwei Polygone miteinander verbindet [ESK96]. In Abbildung 2.1a ist, aus einer geometrischen Sichtweise heraus, einer der Eckpunkte mittels eines gelben Punkts gekennzeichnet. Eine Kante ist rot markiert und eine Fläche, bzw. ein Polygon blau.

2.1.3 Dreiecke

Heutige Grafikkarten und Grafikbeschleuniger sind auf eine spezielle Art von Geometrieinformationen eingestellt. Polyeder, die in dieser Art vorliegen, besitzen nur dreieckige Flächen (durch eine effiziente Anordnung dieser Flächen ist eine weitere Beschleunigung möglich). Kurz werden diese Flächen auch *Dreiecke* oder *Triangles* genannt. Wird die Geometrie wie erwartet geliefert, ist eine deutliche Erhöhung der Darstellungsgeschwindigkeit möglich. Dabei ist die Form der sichtbaren Fläche irrelevant, nur die interne Darstellung ist von Relevanz. In Abbildung 2.1b ist durch das Drahtgittermodell des dargestellten Polyeders ersichtlich, dass die rechteckigen Grundflächen aus Abbildung 2.1a intern aus zwei dreieckigen Flächen gebildet werden. Weiterhin lässt sich ein grobes Leistungskriterium durch die Anzahl der Dreiecke ableiten: je weniger Dreiecke ein Objekt besitzt, desto schneller kann es dargestellt werden.

2.1.4 Polygon Mesh

Ein Polygon Mesh, kurz Mesh, ist die üblichste Darstellungsform eines Polyeders innerhalb der Computergrafik [BB03]. Das Mesh wird dabei über eine Menge von zusammenhängenden Vertices, Edges und Faces beschrieben. Es wird von einem Triangle-Mesh

gesprochen, wenn das Mesh in der Idealform vorliegt: es besitzt nur dreieckige Flächen. Für die Speicherung von Meshs existieren viele Dateiformate, die allerdings alle auf diesem Grundsatz fußen und nur die beschreibende Sprache unterschiedlich definieren, Metainformationen hinzufügen oder eine (sicherheitsrelevante) Serialisierung hinzuziehen. Für die Darstellung des Modells in Echtzeit-Anwendungen empfehlen sich binäre Dateiformate, da diese wesentlich schneller geladen werden können. Durch die simple Definition mittels Eckpunkten und Kanten kann ein Mesh allerdings auch zur Laufzeit eines Programms erzeugt werden. Es ist möglich ein Mesh im Sinne eines ungerichteten Graphen zu interpretieren: die Vertices bilden in diesem die Knoten und die Edges die Kanten zwischen den Knoten. Für ein Mesh ist auch die Bezeichnung Modell oder Objekt einer Szene, wenn es zur Laufzeit innerhalb einer Anwendung angezeigt wird, üblich.

2.1.5 Normalenvektor

Ein Normalenvektor beschreibt, bei Beschränkung der Definition auf den dreidimensionalen Raum, einen Vektor, der orthogonal zur einer Geraden, Kurve, Ebene oder Fläche steht. Innerhalb der Computergrafik wird jedem Vertex ein Normalenvektor zugeordnet, um die Orientierung der abhängigen Flächen zu einer gegebenen Lichtquelle beschreiben zu können. Mittels dieser Vektoren kann während des Renderings die Beleuchtungsrechnung ausgeführt werden.

2.2 Dateiformate

Wie in der Einführung der Arbeit bereits erwähnt, sollen die Dateiformate GeoTIFF und DTED von dem erstellten Programm unterstützt werden. Zum besseren Verständnis der Funktionsweise beider Formate sollen diese in den beiden folgenden Abschnitten präsentiert werden.

2.2.1 GeoTIFF

Das GeoTIFF-Format ist eine spezialisierte Form des TIFF-Formats [RM00]. In diesem werden innerhalb des Bildes geographische Daten als Rasterdaten mit hoher Abbildungsgenauigkeit gespeichert. Das TIFF-Format wurde als Grundlage gewählt, da dieses ein sehr stabiles Format über einen längeren Zeitraum ist [Lan12] und es eine verlustfreie Speicherung ermöglicht. Das GeoTIFF-Format besitzt eine Reihe von Metatags innerhalb

der Bilddatei, die Angaben zu Geoinformationen (bspw. Koordinaten, Auflösung des Bildes in Bogensekunden und Gesamtgröße des Bildes in Grad) machen. Die Konzeption des Formats geht auf Intergraph zurück, wird inzwischen allerdings von zahlreichen Institutionen im Bereich der Geoinformationsysteme eingesetzt, wie zum Beispiel der United States Geological Survey. Die Darstellung eines GeoTIFF-Bildes ist mit jedem TIFF-Anzeigeprogramm möglich.

2.2.2 DTED

Das DTED-Format (Digital Terrain Elevation Data) ist ein Format zur Speicherung von Rasterdaten über Geländeinformationen [U.S89]. Die Dateien werden dabei in Ordner je nach ihrem Längengrad einsortiert. Innerhalb eines Ordners existiert für jeden Breitengrad eine Datei. Jede Datei beschreibt somit ein Gebiet von 1x1 Grad. Den Rasterzellen in diesen Dateien ist jeweils ein spezifischer Höhenwert zugeordnet. Wie bei GeoTIFF existiert eine Reihe zusätzlicher Metadaten. Die wichtigste gibt die DTED-Stufe an. Über diese wird die Fläche eine Rasterzelle innerhalb des Bildes beschrieben. Es gibt dabei drei verschiedene Stufen, die wie folgt definiert sind:

- DTED-0 $\widehat{=}$ Zellengröße von 30 Bogensekunden, ca. 1 km
- DTED-1 $\widehat{=}$ Zellengröße von 3 Bogensekunden, ca. 100 m
- DTED-2 $\widehat{=}$ Zellengröße von 1 Bogensekunde, ca. 30 m

Das DTED-Format wurde von der National Geospatial-Intelligence Agency (NIMA) für militärische Zwecke konzipiert. Dies ist auch der Grund, weshalb nur DTED-Dateien der Stufe 0 öffentlich verfügbar sind.

Kapitel 3

Rendering

Nachdem die begrifflichen Grundlagen geklärt sind, soll dieses Kapitel den Prozess des Renderings erläutern. Für diesen Begriff existiert im Deutschen keine adäquate Übersetzung. Grob kann Rendering als der Prozess beschrieben werden, in dem eine n-dimensionale Szene in ein Bild umgewandelt werden soll [AMHH10]. Eine Szene beschreibt dabei eine virtuelle Umgebung, in der eine beliebige Menge an Modellen dargestellt wird. Zudem beinhaltet eine Szene eine sogenannte Kamera. Diese besitzt eine Position, eine Richtung und einen Blickwinkel [BB03]. Wie eine echte Kamera wird sie genutzt, um einen Ausschnitt der Szene zu einem Bild umzuwandeln. Ein weiteres obligatorisches Element innerhalb einer Szene ist eine Lichtquelle, da sonst das erzeugte Bild schwarz wäre. Es gibt verschiedene Lichttypen. Die bekanntesten sind dabei das Umgebungslicht, das verhindert, dass irgendein Objekt der Szene unter einen bestimmten Helligkeitswert fällt, und das Punktlicht, das Licht wie eine Lampe an einem festgelegten Ursprung erzeugt und gleichmäßig in alle Richtungen abstrahlt.

3.1 Preprocess- und Echtzeit-Rendering

Der Renderingprozess nutzt eine Kamera und erzeugt aus ihrer Orientierung und ihrem Blickfeld ein Bild, welches auf einem Computerbildschirm angezeigt werden kann. Die Anzeige gehört allerdings nicht mehr zum Prozess des Renderings. Es lassen sich zwei Kategorien von Rendering unterscheiden: das Preprocess-Rendering und das Echtzeit-Rendering. Das Preprocess-Rendering rendert dabei eine Szene, bevor sie angezeigt werden soll. Diese Technik findet vor allem bei Filmen oder Bildern statt. Dies sind Medien in denen der Nutzer durch seine Handlungen die Szene nicht beeinflussen kann. Der Ablauf der erstellten Bilder ist daher immer derselbe. Durch das bereits erfolgte Rendering muss das Wiedergabegerät eines Films diesen Prozess nicht mehr ausführen. Es muss nur noch das

Bild zur Anzeige gesendet werden. Als Resultat ist der Leistungsanspruch solcher Medien an die Hardware deutlich geringer. In Medien, in denen der Anwender die Bewegung der Kamera bestimmt und somit auch die Reihenfolge der dargestellten Bilder, kann das Rendering nicht im Voraus erfolgen. Stattdessen muss das Rendering zur Laufzeit stattfinden. In diesem Fall wird von Echtzeit-Rendering gesprochen. Sämtliche interaktiven Grafikanwendungen können hierfür als Beispiel genannt werden, wie etwa Computerspiele oder Simulationen. Die Qualität des Renderingprozesses muss, in diesem Fall, vernachlässigt werden, damit das Bild sofortig darstellbar ist. Bei Computerspielen etwa wird nicht nur ein einzelnes Bild verlangt, sondern eine dauerhafte Folge von Bildern (auch *Frames* genannt). Damit die Abläufe innerhalb des Spiels als flüssig wahrgenommen werden, müssen in einer Sekunde mehr Bilder angezeigt werden, als das Auge in einer Sekunde aufnehmen kann. Häufig angepeilte Zielwerte sind hierbei 30 oder 60 Bilder pro Sekunde (FPS). Der FPS-Wert sollte zudem nicht zwischen verschiedenen Werten variieren, da sonst einige Abläufe flüssiger wahrgenommen werden als andere.

3.2 Rendering Pipeline

Die einzelnen Schritte, die während des Renderings stattfinden, sind bei Preprocess- und Echtzeit-Rendering theoretisch gleich. Dabei kann beim Preprocess-Rendering mehr Wert auf Qualität gelegt werden, während Echtzeit-Rendering die selben Arbeitsschritte mit Fokus auf Geschwindigkeit ausführen würde. Das Rendering durchläuft beim Erstellen eines Bildes aus einer Szene die sogenannte *Rendering-Pipeline*. Diese bestimmt den Ablauf der benötigten Schritte zur Erzeugung des Bildes. Je nach Wahl des Renderers kann eine andere Pipeline zum Einsatz kommen.

Der Grundablauf jeder Pipeline ist allerdings identisch. Die eingehenden Daten sind geometrische Informationen über die Modelle [BB03]. Das bedeutet die Menge ihrer Vertices, Kanten und Flächen. Ebenfalls gehören dazu die Texturen, die auf den jeweiligen Modellen liegen. Texturen beschreiben das Aussehen des Modells (beispielsweise durch Bilder) [CR11], während das Modell die Form bestimmt. Die Pipeline selbst ist in mehrere Abschnitte unterteilt, die die Daten durchlaufen [AMHH10]. Dabei können die Daten erst in den nächsten Abschnitt aufrücken, wenn der aktuelle beendet ist. Es ist allerdings möglich, in jedem Abschnitt gleichzeitig unterschiedliche Daten zu behandeln. Eine Pipeline mit n Abschnitten kann den Renderingprozess dadurch theoretisch um das n-fache beschleunigen. Praktisch ist dies allerdings nicht der Fall: Haben die Abschnitte eine unterschiedliche Laufzeit, bremst der langsamste die restlichen aus. Es entsteht dadurch ein Flaschenhals im Datenstrom der Pipeline.

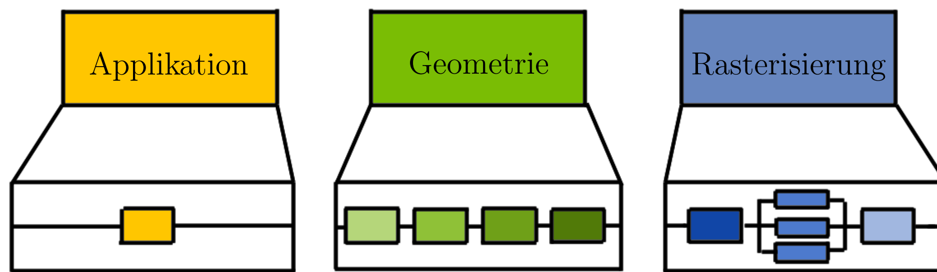


Abbildung 3.1: Hauptabschnitte einer Render-Pipeline: Applikation, Geometrie und Rasterisierung. Die Pipeline verwendet parallelisierte Arbeitsschritte. Entnommen aus [Feu10] nach [AMHH10].

Einen grundlegenden Aufbau einer Rendering-Pipeline zeigt Abbildung 3.1. In dieser wird die Pipeline in die drei Abschnitte Applikation, Geometrie und Rasterisierung aufgeteilt [AMHH10]. Die Optimierung innerhalb dieser Abschnitte ist wegen des zuvor angesprochenen Flaschenhalseffekts enorm wichtig.

Der erste Abschnitt der Rendering-Pipeline, *Applikation*, wird noch vollständig auf der CPU ausgeführt. Ein Entwickler einer Grafik-Anwendung hat in diesem Bereich daher noch die vollständige Kontrolle darüber, wie der Bereich implementiert werden soll. Einerseits sollten hier logische Operationen ausgeführt werden, die Auswirkungen auf die Szene haben [AMHH10]. Dies ist grundsätzlich die Ausführung von Transformationen innerhalb der Szene: Rotation, Translation und Skalierung von Objekten. Auf einer höheren Ebene könnte damit die Ausführung einer Physiksimulation gemeint sein. Andererseits sollten Optimierungen für die nächsten Abschnitte der Rendering-Pipeline ausgeführt werden. Das Ziel ist es dabei, die Menge der zu rendernden Objekte zu minimieren. Ein oft gewählter Ansatz ist das sogenannte *Culling*. In diesem Prozess werden Elemente bestimmt, die aus der momentanen Kameraansicht heraus nicht sichtbar sind. Nur die sichtbaren Objekte werden an den nächsten Abschnitt weitergeleitet. Es wird hier unterschieden zwischen *Frustum Culling*, bei dem Elemente außerhalb des Sichtbereichs ausgefiltert werden, *Backface Culling*, bei dem die, aus der momentanen Perspektive, nicht-sichtbaren Rückseiten von Objekten innerhalb der Szene entfernt werden, und *Occulsion Culling*, bei dem Objekte entfernt werden, die durch Objekte im Vordergrund verdeckt werden [LRC+03]. Unabhängig davon ist der einzig notwendige Arbeitsschritt innerhalb des Applikationsabschnittes die Umrechnung jeglicher weiterzuleitender Geometrie in primitive Geometrie: in Kanten, Vertices und Dreiecke.

Die primitive Geometrie wird an den zweiten Abschnitt, *Geometrie*, weitergeleitet. In diesem Abschnitt werden die meisten Arbeitsschritte vollzogen, die sich auf einzelne Polygone oder Vertices beziehen. Zudem werden die Operationen dieses Abschnitts fast vollständig

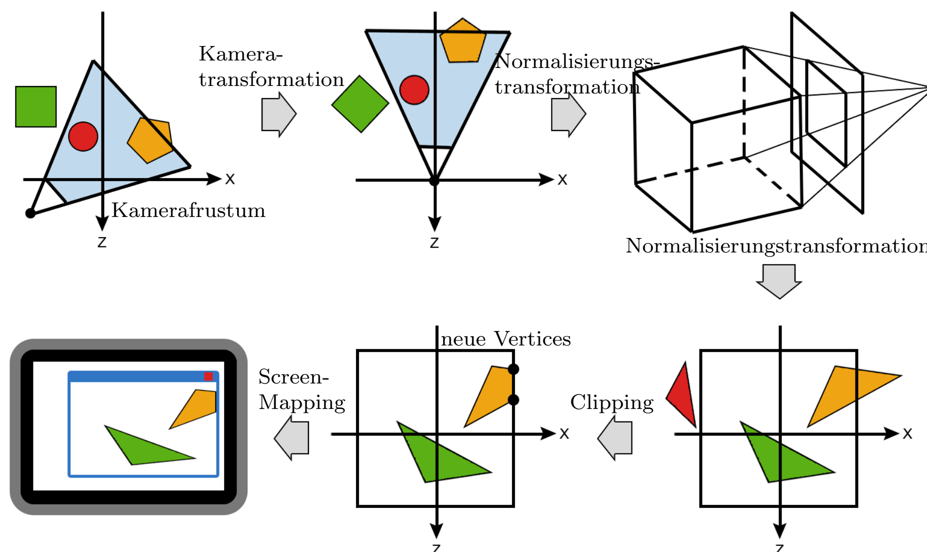


Abbildung 3.2: Ablauf des Geometrie Abschnitts. Kamera-Transformation, Projektion, Screen Mapping, Scan Conversion. Entnommen aus [Feu10] nach [AMHH10]

auf der GPU ausgeführt, was die Handlungsfreiheit der Entwickler, aufgrund der mit nur wenigen Befehlen ausgestatteten Programmierschnittstelle (API) der GPU, begrenzt [AMHH10]. Der Abschnitt selbst wird in mehrere Phasen unterteilt: Kameratransformation, Vertex-Schattierung, Projektion, Clipping und Screen Mapping. Die Arbeitsweise und Reihenfolge dieser Arbeitsschritte ist in Abbildung 3.2 dargestellt.

Die erste Phase, die *Kameratransformation*, besteht aus zwei Schritten. Im ersten Schritt werden die Modellkoordinaten in sogenannte Weltkoordinaten überführt. Dies ist nötig, da jedes Objekt in der Szene über ein eigenes Koordinatensystem verfügt. Dies kann beliebig gedreht, skaliert und verschoben sein, die Geometrie des Modells wird über das eigene Koordinatensystem ausgerichtet. Der Vorteil hiervon ist, dass einem Modell mehrere Koordinatensysteme zugewiesen werden, wodurch eine Geometrie nur einmal geladen werden muss, aber mehrmals angezeigt werden kann. Der erste Schritt der Kameratransformation löst nun diese Koordinatensysteme auf und überführt die Modelle in ein einziges (kartesisches) Koordinatensystem. Im zweiten Schritt wird die Kamera innerhalb dieses Koordinatensystems auf den Ursprung gesetzt, mit einem Blickwinkel in Richtung der negativen z-Achse, sodass die positive x-Achse rechts von der Kamera liegt und die y-Achse orthogonal zur Kamera ist. Dies erlaubt den späteren Phasen eine schnellere Durchführung ihrer Operationen. Alle anderen Elemente der Szene werden abhängig von der Bewegung der Kamera ebenfalls transformiert. Abbildung 3.2 zeigt den Ablauf der Kameratransformation mittels der ersten zwei Bilder. Innerhalb der zweiten Phase, der *Vertex-Schattierung*, werden Materialien und Lichtquellen auf die Modelle innerhalb der Szene angewendet.

Dafür werden die in der Einleitung angesprochen Normalvektoren benötigt: Durch diese kann berechnet werden, in welchem Winkel eine Lichtquelle zu einem Modell steht. Wird dies mit der Entfernung zwischen Modell und Lichtquelle, sowie anderen Objekten der Szene kombiniert, ist es möglich, jedem Teil des behandelten Modells ein Aussehen zuzuweisen. Zusammengefasst spezifiziert die erste Phase die Form und die zweite Phase das Aussehen der Szenenobjekte.

Die dritte Phase transformiert die Szene aus Sicht der Kamera. Diese ist meistens perspektivisch und besteht damit aus einem Sichtkegel. Der Sichtkegel wird durch zwei Werte in der Nähe und Ferne (Near Clip Plane und Far Clip Plane) begrenzt [AMHH10]. Durch die *Normalisierungstransformation* wird das durch diesen begrenzten Kegel beschriebene Volumen in einen Kubus transformiert, der im Ursprung zentriert ist und sich in alle Dimensionen mit dem Wert 1 ausdehnt. Die Inhalt des Volumens (die Objekte der Szene) wird bei dessen Transformation entsprechend verzerrt. Durch diesen Prozess ist anschließend eine Parallelprojektion auf eine rechteckige Fläche möglich. Dieser Prozess wird durch die beiden rechten Bilder in Abbildung 3.2 dargestellt.

In der vierten Phase, dem *Clipping*, werden Objekte entfernt, die außerhalb des Kubus liegen, da diese nicht angezeigt werden. Liegt ein Objekt nur teilweise im Kubus, wird der Teil außerhalb entfernt und neue Vertices am Rand des Kubus für dieses Objekt zur Verfügung gestellt. Abbildung 3.2 zeigt diesen Prozess im fünften Bild. Die letzte Phase ist das sogenannte *Screen Mapping*. In diesem werden die momentan noch dreidimensionalen Koordinaten (durch die Nutzung von OpenGL und Direct3D verwenden selbst 2D-Anwendungen heute 3D-Koordinaten) in zweidimensionale Bildschirmkoordinaten umgewandelt.

Im letzten Abschnitt, der *Rasterisierung*, werden die nun als zweidimensionale Geometrie vorliegenden Modelle in Pixel umgerechnet [AMHH10]. Ist einem Modell eine Textur zugewiesen, wird diese während der Umwandlung zur Bestimmung des Pixel-Farbwerts genutzt. Kommen für einen Pixel mehrere Objekte in Frage, wird das verwendet, welches den niedrigsten z-Koordinatenwert besitzt. Am Ende des Abschnittes entsteht ein fertiges, anzeigbares Bild. Diese beispielhafte Rendering-Pipeline ist für Echtzeit-Rendering optimiert. Vor allem in anderen Gebieten, wie dem Pre-Process-Rendering, können die Pipelines deutlich von der hier vorgestellten abweichen. Für das Echtzeit-Rendering ist eine wichtige Zusatzinformation allerdings, dass für jedes zu rendernde Mesh ein sogenannter *Draw-Call* ausgeführt werden muss, der das Modell der GPU zur Anzeige übergibt [AMHH10]. Da jeder Call zusätzliche Laufzeit kostet, ist das Rendering weniger großer Meshs billiger als das Rendering vieler kleiner Meshs.

Kapitel 4

Höhenkarten und Detaillierungsgrade

Dieses Kapitel beschäftigt sich mit der Beschreibung von Höhenkarten und Detaillierungsgraden. Die beiden Themen sind innerhalb eines Kapitel zusammengefasst, um sie im Verbund betrachten zu können. Es ist das Ziel, die verschiedenen Techniken der Detaillierungsgrade nicht nur zu beschreiben, sondern sie auch im Hinblick ihrer Qualität zur Darstellung von Höhenkarten bewerten zu können.

4.1 Höhenkarten

Eine Höhenkarte ist ein Rasterbild, welches genutzt wird um Daten über eine Fläche zu speichern. Das Rasterbild wird dabei üblicherweise als Graustufen-Bild gespeichert [CR11]. Die Höhenkarte beschreibt dabei ein uniformiertes Raster, das heißt jeder Pixel der Datei beschreibt eine genau gleichgroße Fläche. Die Größe der Fläche wird dabei über eine Auflösung angegeben. Ist zum Beispiel eine Auflösung von 1 m gegeben, beschreibt jeder Pixel eine Fläche von 1 m^2. Bei Höhenkarten wird mittels des Pixels die Höhe der von ihm beschriebenen Fläche als Datenwert gespeichert. Dies geschieht über den Farbwert des Pixels. Für Graustufen ergeben sich dadurch 256 mögliche Höhenwerte (bei 8-Bit Farbtiefe). Standardmäßig beschreibt schwarz (0) dabei den geringsten und weiß (255) den höchsten Wert. Die meisten expliziten Formate für Höhenkarten besitzen zusätzlich Metadaten, die Auskunft über die Auflösung geben, welcher Höhe die Steigerung des Höhenwerts um eins entspricht und welches Gebiet durch die Höhenkarte abgebildet wird. Höhenkarten werden vor allem im Bereich von Geoinformationssystemen verwendet, finden allerdings, aufgrund ihrer guten Kompressionsrate für große Gebiete, auch Anwendung in Filmen oder Computerspielen [LRC+03]. In Abbildung 4.1 ist einmal

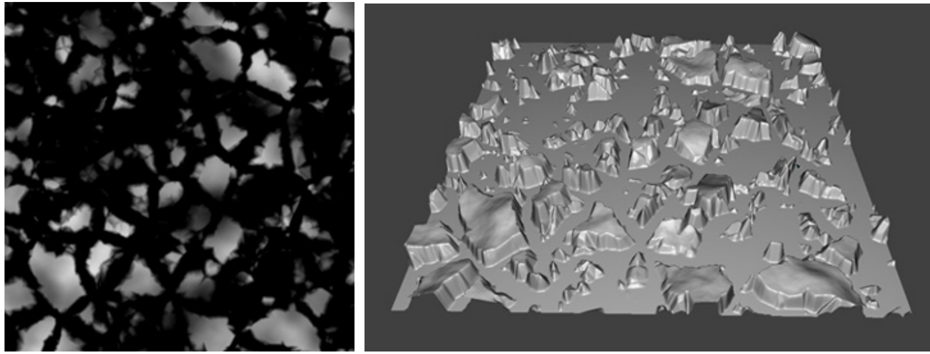


Abbildung 4.1: Links: Höhenkarte in Form eines Graustufen-Rasterbildes. Rechts: Höhenkarte übertragen auf ein uniformiertes Gitter-Mesh. Entnommen aus [KS13].

das Rasterbild einer Höhenkarte zu sehen, sowie die Interpretation als Landschafts-Mesh in einer Anwendung.

Zwar bieten sich zweidimensionale Höhenkarte aufgrund ihrer Einfachheit an, Abbildung 4.2 zeigt jedoch auch einen Nachteil in ihrer Benutzung. Da nur maximale Höhenwerte gespeichert werden, können verschiedene landschaftliche Phänomene nicht dargestellt werden, beispielsweise Überhänge, Höhlen oder Klippen. Hierzu sind andere Formate oder zusätzliche Metadaten nötig [CR11].

Kann jedoch auf solche Details verzichtet werden, ist die Benutzung von Höhenkarten gegenüber gewöhnlichen Meshs und anderen Formaten von Vorteil, da diese wesentlich weniger Speicherplatz benötigen. Während der Laufzeit eines Programms, kann das Programm aus einer Höhenkarte ein Mesh erstellen, welches genau die von der Höhenkarte beschriebene Form aufweist. Diese Interpretation einer Höhenkarte wird *Displacement-Mapping* genannt [CR11, LRC+03]. Es wird ein uniformiertes Mesh-Gitter erzeugt, das genau die Ausmaße der Höhenkarte hat. Das heißt, für jeden Pixel existiert ein Vertex. Um das Gitter zu erzeugen werden die Vertices mittels Kanten mit ihren Nachbarn ver-

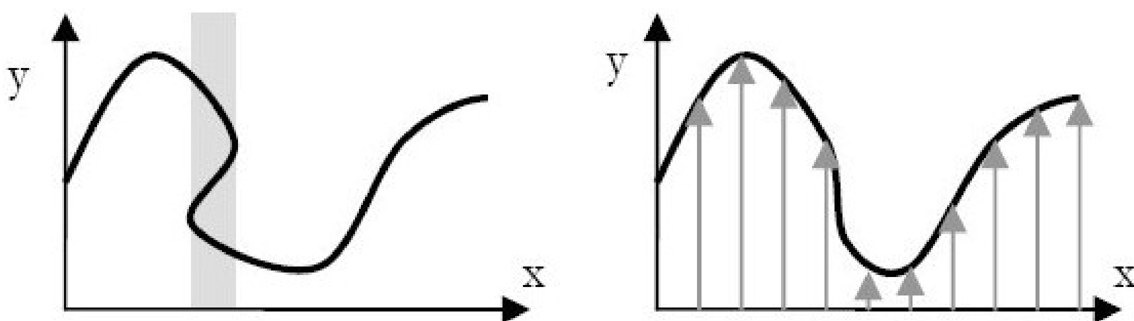
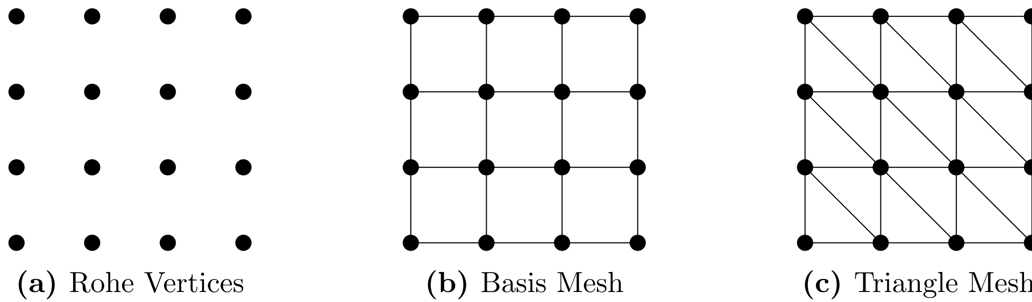


Abbildung 4.2: Limitierte Darstellung durch Höhenkarten (2D äquivalent). Links: (x,y) = (fx[s], fy[s]). Rechts: (x,y) = (x, f[x]). Entnommen aus [Joh04].



(a) Rohe Vertices **(b)** Basis Mesh **(c)** Triangle Mesh

Abbildung 4.3: Erstellung eines initialen uniformierten Mesh-Gitters

bunden, sodass ein einfaches Gitter entsteht. Jede entstandene quadratische Zelle wird in zwei Dreiecke geteilt, indem eine weitere Kante zwischen zwei schräg gegenüber liegenden Vertices erstellt wird. Damit ist die Erstellung eines uniformierten Mesh-Gitters, welches gleichzeitig ein Triangle-Mesh ist beendet. Erklärend sei hier auf Abbildung 4.3 verwiesen. Zur Anwendung der Höhenkarten werden die Koordinaten des Vertex danach über die Gleichung 4.1 verändert:

$$\vec{v} = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} = \begin{pmatrix} p_x * m_{res} \\ p_y * m_{res} \\ p_{Grau} * m_{Höhe} \end{pmatrix}, \tag{4.1}$$

mit Vektor $\vec{v}$, Pixel p und den Metadaten der Höhenkarte m. Für das manipulierte Vertex muss somit lediglich der Wert der Höhe verändert werden (hier: z-Koordinate). Der neue Höhenwert entspricht dabei dem Farbwert des Pixel in Abhängigkeit zum Höhenreferenzwert der Metadaten ($m_{Höhe}$). Soll die Größe der Höhenkarte ebenfalls übernommen werden, müssen die beiden anderen Koordinaten mit der Auflösung der Höhenkarte (m_{res}) multipliziert werden. Das Anwenden einer Höhenkarte entspricht damit der Manipulation, der Verschiebung der Vertices, eines bestehenden Meshs.

4.2 Detaillierungsgrad

Bereits 1976 befasste sich James Clark mit der effizienten Darstellung von verschiedenen Objekten in einer Szene [Cla76]. Hierbei griff er bestehende Algorithmen auf, die sich vor allem damit beschäftigten, sichtbare Flächen von den Szeneobjekten zu finden. Clark zeigte die Vorteile davon, nicht nur die Sichtbarkeit der Objekte zu bestimmten, sondern sie auch abhängig von dieser in verschiedenen Detaillierungsgraden anzuzeigen. Zur Verwaltung der Szene entwarf er eine hierarchische Struktur: den sogenannten Szenengraph. Dieser Graph setzt verschiedene Objekte einer Szene in Zusammenhang, Kinderknoten

übernehmen Translation, Rotation und Skalierung des Elternknotens und ihr Koordinatensystem verhält sich relativ zu dem des Elternteils.

Heutzutage gehören die Verwendung eines Szenengraphen und verschiedener Detaillierungsgrade zum Standardrepertoire in der Computergrafik. Zwar wurden die hierfür notwendigen Methoden inzwischen für viele Teilbereiche verfeinert, die Grundidee hinter der Verwendung von Detaillierungsgraden (englisch: *level-of-detail*, kurz: *LOD*) hat sich jedoch nicht geändert: je unwichtiger (je weniger sichtbar) ein Objekt für eine Szene ist, desto simpler soll die Variante sein in der es angezeigt wird. Ein allgemeiner Level-of-Detail-Algorithmus kann über die drei Schritte Erstellung, Auswahl, Austausch definiert werden [AMHH10]. In der Phase der Erstellung werden aus dem originalen Modell mehrere Varianten mit unterschiedlichem Detaillierungsgrad erstellt. Diese Erstellung kann je nach Verfahren per Hand oder durch einen Algorithmus erfolgen. In der Phase der Auswahl wird zur Laufzeit eines Programms entschieden, welcher Detaillierungsgrad einem Objekt zugewiesen wird. Die Kriterien hierfür sind abhängig vom gewählten Mechanismus [LRC+03], eine Möglichkeit wäre zum Beispiel die Entfernung zwischen Kamera und Objekt auszuwerten. In der letzten Phase, dem Austausch, wird die aktuelle Version des Objekts mit der neuen ausgetauscht. Das Problem in dieser Phase liegt darin, dass der Nutzer möglichst wenig vom Austausch der beiden Varianten mitbekommen soll. Es wird von *Popping* gesprochen, wenn dieses Ziel nicht oder nur ungenügend erreicht wurde und die Objekte merkbar in die Szene *springen* [LRC+03].

Im Folgenden sollen verschiedene Arten des Level-of-Detail vorgestellt werden. Für jede der vorgestellten Arten soll zusätzlich entschieden werden, ob sie eine effiziente Verwaltung und Modellierung einer Landschaft erlaubt, die nur als Höhenkarte vorliegt.

4.2.1 Diskretes Level-of-Detail

Der von James Clark beschriebene Entwurf wird noch heute in vielen Grafikanwendungen ohne Modifikationen verwendet. Methoden, die auf diesem Ansatz fußen, werden dabei unter dem Bereich des diskreten LOD (DLOD) zusammengefasst [LRC+03]. Die Phase der Erstellung der verschiedene Detaillierungsgrade wird hierbei von den anderen Phasen abgekoppelt. Bereits vor der Laufzeit werden statt eines Modells, verschieden viele Varianten erzeugt, jeweils mit unterschiedlich hohen Detaillierungsgraden [Cla76]. Der Fokus liegt hierbei vor allem auf der Reduktion der Dreiecke eines Modells, da diese den größten Leistungsgewinn mit sich bringen. In Abbildung 4.4 ist beispielsweise der *Stanford-Hase* zu sehen. Das linke Modell entspricht dabei der originalen Auflösung, die anderen drei Modelle entsprechen reduzierten LOD-Varianten.

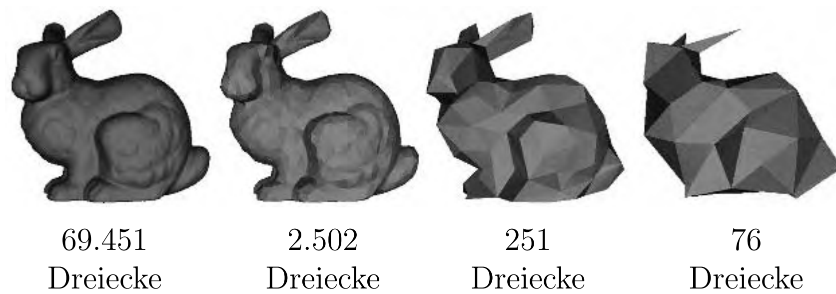


Abbildung 4.4: Verschiedene Detaillierungsgrade eines Modells. Die Details nehmen von rechts nach links ab. Entnommen aus [LRC+03]

Zur Laufzeit wird, anhand eines fixen Kriteriums, die zu verwendende Variante ausgewählt. Für gewöhnlich ist dies der euklidische Abstand des Modells zur Kamera: es werden beliebige, aber feste Grenzen definiert, ab welcher Distanz welche LOD-Variante des Modells angezeigt wird. Der Austausch der Modelle erfolgt dabei sofort nach der Auswahl eines neuen Modells. Durch die gröbere Auflösung der weiter entfernten Modelle kommt es zu einer deutlichen Geschwindigkeitssteigerung des Renderingvorgangs, ohne die sichtbare Qualität des Objekts zu reduzieren. Abbildung 4.5 zeigt die Modelle aus Abbildung 4.4 in verschiedenen Tiefen. Der sichtbare Unterschied ist bei diesen minimal, obwohl das vierte Modell gerade einmal 0,11 % der Dreiecke des Originals besitzt.

Die Verwendung von diskreten LOD hat deutliche Vorteile. Durch die Entkopplung der Erstellung der vereinfachten Modelle von der Anzeige und dem Austausch, wird das System sehr einfach nutzbar. Zusätzlich kann der Algorithmus zum Vereinfachen des Modells beliebig lange benötigen, da die von ihm erzeugten Daten persistent sind. Somit kann ein qualitativ optimales Ergebnis angestrebt werden, statt eines Laufzeit-optimalen Ergebnisses. Weiterhin kann der Renderingprozess deutlich beschleunigt werden, da keine Erstellung der vereinfachten Modelle zur Laufzeit nötig ist, sondern lediglich eine zu nut-

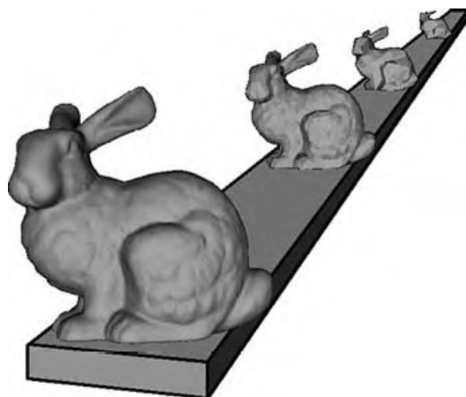


Abbildung 4.5: Detaillierungsgrade in Abhängigkeit zur Entfernung. Die zuvor offensichtlichen Unterschiede der Meshs sind hier kaum sichtbar. Entnommen aus [LRC+03]

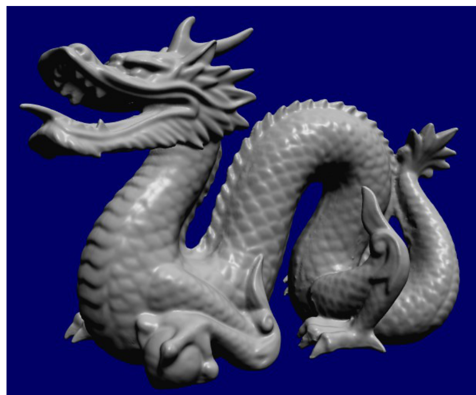


Abbildung 4.6: Modell des Stanford-Drachen mit 1.132.830 Dreiecken. Entnommen aus [Sta14]

zende Variante ausgewählt werden muss. Ein Nachteil des DLOD-Systems liegt allerdings im bereits vorgestellten Popping. Durch die stufenweisen Übergänge werden die Wechsel der Detaillierungsgrade auch dem Nutzer erkenntlich. Dieses Phänomen lässt sich durch die Technik des *Blending*s dämpfen. Dabei findet ein Austausch nicht sofort statt, sondern zwei LOD-Varianten werden zeitgleich als ein Objekt angezeigt [AMHH10]. Die aktuelle Variante wird dabei eingeblendet, die alte Variante im gleichen Maße ausgeblendet. Während dieser Überblendung geht allerdings der angestrebte Vorteil der Detaillierungsgrade verloren, da statt eines Objekts zwei gerendert werden müssen.

Diese Kosten sind im Sinne eines optimalen Erlebnisses für den Nutzer den Aufwand jedoch wert: gerade bei wenig detaillierten LOD-Varianten, sind die Kosten der Anzeige zweier Objekte immer noch niedriger als die des originalen Objekts [AMHH10]. Im Bezug auf Höhenkarten ergeben sich für das DLOD-System jedoch Probleme. Einerseits werden fertige Modelle in verschiedenen Detaillierungsgraden erwartet. Die Landschaft liegt allerdings als Höhenkarte vor. Sie müsste daher direkt nach dem Laden in ein Modell umgewandelt werden und mehrere Verfeinerungen müssten berechnet werden. Dies würde die erste Anzeige deutlich verzögern und den Arbeitsspeicher durch die Ablage gleich mehrere Meshs stark belasten. Durch die vorzeitige Berechnung der LOD-Varianten wäre es zudem nicht möglich, die Landschaft gleichzeitig an verschiedenen Positionen in verschiedenen Detaillierungsgraden anzuzeigen. Dies ist für das effiziente Rendering von Landschaften notwendig, da entfernte Teile der Landschaft nur in geringem Detail dargestellt werden sollten, nahe Teile der gleichen Landschaft allerdings in hohem Detail. Das Konzept der diskreten LOD wird beim Versuch Landschaften darzustellen, damit durch deren schiere Größe unbrauchbar gemacht. DLOD-Systeme eignen sich somit nur für komplexe Modelle die eine bestimmte Gesamtgröße nicht übersteigen, wie etwa Abbildung 4.6.

4.2.2 Kontinuierliches Level-of-Detail

Das kontinuierliche Level of Detail (CLOD) erweitert den Ansatz des diskreten LOD um die Nachteile des DLODs zu reduzieren [AMHH10, LRC+03]. Insbesondere wird das Problem des Poppings fokussiert und der höhere Speicherplatzverbrauch durch die verschieden detaillierten Varianten eines Modelles. Statt vor der Laufzeit mehrere Versionen eines Mehs zu erstellen, wird über das CLOD-System eine Datenstruktur zur Verfügung gestellt, die für jedes Modell ein kontinuierliches Spektrum an Detaillierungsgraden bereithält. Während der Laufzeit wird, bei jeder Kameraänderung, aus dieser Datenstruktur die momentan benötigte LOD-Variante des Modells entnommen. Durch die hierdurch gegebene dauerhafte Versorgung mit voneinander nur sehr geringfügig unterschiedlichen LOD-Varianten eines Modells, wird das Auftreten des Poppings effektiv verhindert. Zudem ist die Detailmenge stets optimal in Abhängigkeit zur Position des Benutzers: bewegt dieser sich geringfügig zum oder vom Objekt, werden dessen Details geringfügig erhöht oder reduziert – beim DLOD wäre keine Änderung eingetreten. Dies ist auch für das Rendering von Vorteil, da nie mehr Dreiecke eines Objekts dargestellt werden, als unbedingt notwendig.

Durch Hoppe wurde 1996 die Datenstruktur der *Progressive Meshes* entworfen [Hop96]. Die Idee dieser Datenstruktur ist, sich an ein gegebenes Mesh durch eine vereinfachte Repräsentation anzunähern. Die Vereinfachung des Meshs wird im Rendering als die benötigte LOD-Variante verwendet. Der Progressive Mesh Algorithmus benötigt dabei lediglich zwei Basisoperationen.

- Edge Collapse: Kantenzusammenführung
- Vertex Split: Vertexaufspaltung

Beim Prozess der Kantenzusammenführung wird das Mesh vereinfacht, indem zwei Vertices, die direkt durch eine Kante verbunden sind durch ein einzelnes Vertex ersetzt werden. Die verbindende Kante fällt dabei weg, alle anderen Kanten die an die beiden Vertices ansetzten werden mit dem neuen Vertex verbunden. Dabei wird verhindert, dass das neue Vertex redundante Kanten besitzt, das heißt mehrere Kanten, die dieses Vertex mit *einem* anderen Vertex verbinden. Dies sorgt dafür, dass die beiden Dreiecke, die durch die zusammengefassten Vertices aufgespannt werden, ebenfalls entfernt werden.

Durch eine vom Algorithmus gewählte Folge von Kantenzusammenführungen *ecol* kann somit ein initiales Mesh $\hat{M} = M^n$ effektiv zu einer vereinfachten Version des Meshs, M^0, transformiert werden:

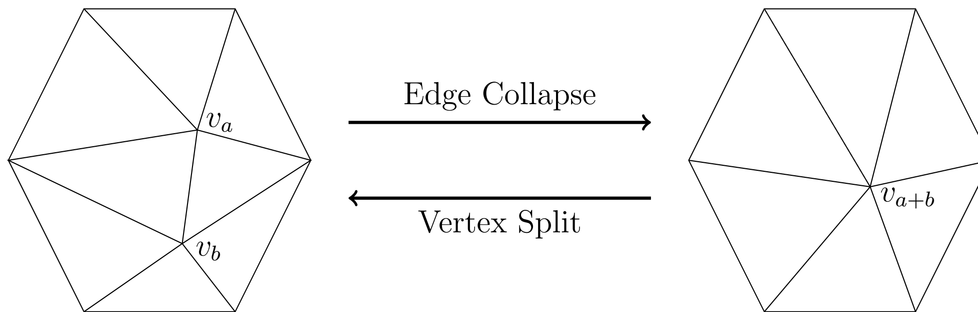


Abbildung 4.7: Eine Kantenzusammenführung und die Vertexaufspaltung als dessen Inversion

$(\hat{M} = M^n) \xrightarrow{ecol_{n-1}} ... \xrightarrow{ecol_1} M^1 \xrightarrow{ecol_0} M^0.$

Die Struktur speichert dabei die jeweils ausgeführten Operationen. Muss der Detaillierungsgrad des Meshs später erhöht werden, werden die zuletzt ausgeführt Operationen mittels dieses Speichers rückgängig gemacht. Dies ist der Vorgang der Vertexaufspaltung *vsplit*. Er stellt die Inverse der Kantenzusammenführung dar. Das bedeutet, dass aus einem Vertex wieder zwei verbundene Vertices entstehen, die dem Mesh zwei neue Dreiecken hinzufügen. Die Vereinfachung, im Sinne obiger Formel, ist somit aufhebbar:

$M^0 \xrightarrow{vsplit_0} M^1 \xrightarrow{vsplit_1} ... \xrightarrow{vsplit_{n-1}} (M^n = \hat{M})$

wobei jede Ausführung von $vsplit_i$ mit $i \in [0, n]$ der Inversen der vorherigen Ausführung von $ecol_{n-i}$ entspricht. Die Zusammenfassung dieser Operationen ist noch einmal bildlich in Abbildung 4.7 dargestellt. Dabei ist vor allem zu beachten, dass die Position des neuen Vertex v_{a+b} entweder eine neu berechnete sein kann (*Edge Collapse*) oder der Position einer der beiden vorherigen Vertices entspricht (*Half Edge Collapse*). In beiden Fällen könnte es zur Entstehung von *missgebildeten* Dreiecken kommen. Damit sind Dreiecke gemeint, die verdreht sind oder dessen Kanten sich überschneiden. Die Auswahl welche Vertices vereinfacht werden muss deswegen gesichert erfolgen [AMHH10]. Der Übergänge zwischen zwei Detaillierungsgraden können in diesem CLOD-System noch weiter reduziert werden, wenn statt eines sofortigen Wechsels der Detaillierungsgrade verschiedene Zwischenschritte angezeigt werden, in welchen der Abstand der beiden Vertices zueinander gesenkt wird. Dieser Prozess wird *Geomorphing* genannt und ist in Abbildung 4.8 dargestellt. [Hop96].

Das kontinuierliche LOD hat allerdings auch diverse Nachteile im Gegensatz zum klassischen DLOD. Erst einmal benötigt der Algorithmus zum Berechnen und Entnehmen der momentanen Detailstufe mehr Rechenzeit, als die einfache Zuweisung eines Modells. Weiterhin muss für jedes Objekt in der Szene, welches durch das CLOD-Systems verwaltet wird, eine der Datenstrukturen angelegt werden, die die Detaillierungsgrade berechnen.

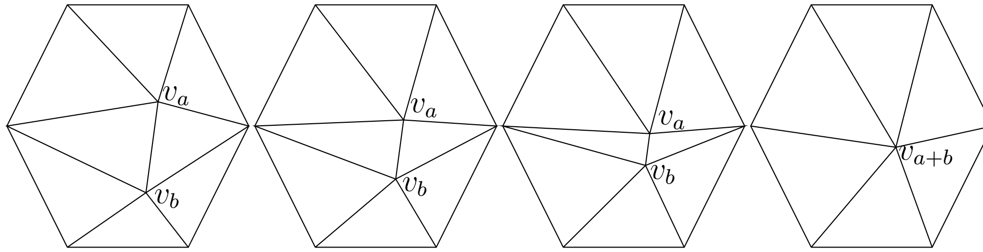


Abbildung 4.8: Schematischer Ablauf des Geomorphings in einem CLOD-System (*Edge Collapse* von links nach rechts)

Es wird dadurch mehr Arbeitsspeicher benötigt. Das Hauptproblem ist allerdings die Berechnung der Detaillierungsgrade. Da der dafür notwendige Algorithmus zur Laufzeit arbeitet, muss er die Qualität der Ergebnisse vernachlässigen, um Geschwindigkeit zu erhalten. Zudem besitzen fast alle Modelle wichtige Bestandteile, die ihre Grundstruktur definieren. Heutige Algorithmen sind allerdings noch nicht in der Lage diese wichtigen Bestandteile von nebensächlichen Details zu unterscheiden: sie verfeinern das gesamte Modelle durchschnittlich im selben Grad [AMHH10]. Beim Anfordern eines niedrigen Detaillierungsgrads könnte dadurch eine Variante zurückgegeben werden, die nicht mehr als das Original erkennbar ist.

Trotzdem ist ein CLOD-System potentiell für das Anzeigen von Landschaften geeignet. Da das Mesh zur Laufzeit vereinfacht und verfeinert wird, muss das Original nur einmal aus der Höhenkarte gewonnen werden. Es wird weiterhin nur eine Struktur benötigt, da die Landschaft nur ein einziges Mesh ist. Auch die Verfeinerungen können bedenkenlos angewandt werden, da die Mesh Struktur der Landschaft einheitlich ist: jedes Dreieck ist gleich wichtig für die Grundstruktur des Meshs. Allerdings verfeinern auch CLOD-Systeme das Mesh überall im gleichen Maß. Die Hautproblematik der Darstellung von Landschaften mithilfe von DLOD-Systemen bleibt somit bestehen.

4.2.3 Sichtabhängiges Level-of-Detail

Eine Erweiterung der CLOD-Systeme besteht mit dem sichtabhängigen Level-of-Detail (VLOD). Hierbei werden sichtabhängige Kriterien verwendet, um zur Laufzeit für ein Objekt den besten Detaillierungsgrad zu bestimmen. Die Techniken zur Vereinfachung des Meshs bleiben dieselben, wie die beim kontinuierlichen LOD. Die neuen Kriterien bestimmen lediglich, an welchem Ort Vertices in welcher Menge zusammengeführt werden. Die Vereinfachungen zwischen zwei Varianten des Meshs sind damit nicht mehr gleichverteilt, sondern abhängig von Distanz, Richtung und Blickwinkel der Szenenkamera [LRC+03]. Dies sorgt dafür, dass nahe Bereiche des Meshs sehr detailliert dargestellt werden, während

weit entfernte oder verdeckte Bereiche stark vereinfacht werden. Die Anzahl der Dreiecke eines Meshs ist somit, zu jeder Zeit, sowohl für die Laufzeit als auch für die Betrachtung optimal.

Im Allgemeinen reichen die folgenden drei Kriterien aus um ein sichtabhängiges LOD-System mit qualitativ hochwertigen Ergebnissen zu erzeugen:

- Sichtbereich
- Flächenausrichtung
- Geometrischer Bildfehler

Diese Kriterien wurden auch in Hoppes Ansatz 1997 verwendet um das vom ihm geschaffene CLOD-System in ein VLOD-System umzugestalten [Hop97a]. Das erste Kriterium besagt, dass sämtliche Teile eines Meshs, die außerhalb des Sichtbereichs der Kamera liegen in ihrem Detail so stark wie möglich vereinfacht werden. Dreiecke innerhalb des Sichtbereichs werden hingegen im vollen Detail dargestellt. Die beiden obigen Bilder in Abbildung 4.9 zeigen einmal das Aussehen eines Meshs aus der Perspektive der Kamera und einmal das gesamte Mesh überblickend. Aus der Kameraperspektive ist das Mesh in vollem Detail zu sehen, die zweite Perspektive zeigt allerdings deutlich die Vereinfachungen außerhalb des Sichtbereichs. Dieses Kriterium überschneidet sich mit dem Prozess des *Cullings* der bei heutigen Renderen üblich ist. Trotzdem ist die Anwendung des Kriteriums sinnvoll, da diese Datenstruktur effizienter mit den geometrischen Daten des Meshs arbeiten kann und somit im Endeffekt die Arbeit des Renderers reduziert.

Das zweite Kriterium vereinfacht alle Dreiecken, deren Flächen entgegengesetzt zur Blickrichtung der Kamera liegen. Dies passiert unter Nutzung des Normalenvektors. Abbildung 4.9 zeigt wiederum (mittige Bilderreihe), dass zwar eine deutliche Vereinfachung des Meshs stattfindet, aus der Kameraperspektive allerdings keine Reduktion des Details bemerkbar ist. Das dritte Kriterium berechnet für jede mögliche Vereinfachung einen Fehlerwert. Diese gibt an, um wie viele Pixel sich die Vereinfachung auf dem entstehenden Bild von einer nicht ausgeführten Vereinfachung unterscheiden würde. Ist der Fehler unterhalb eines beliebigen Grenzwertes, wird die Vereinfachung ausgeführt. Abbildung 4.9 zeigt, in der unteren Bilderreihe, die Anwendung dieses Kriteriums, mit dem Grenzwert von 0,5 Pixeln. Die Reduktion der Dreiecksanzahl durch dieses Kriterium ist am größten, allerdings wurden auch innerhalb des Kamerasichtbereichs Vereinfachungen durchgeführt. Die Veränderung des Modells als Bild liegt dabei jedoch unterhalb von 0,5 Pixeln. Die Vereinfachungen bleiben dadurch für den Benutzer (außer bei sehr genauer Betrachtung) verborgen.

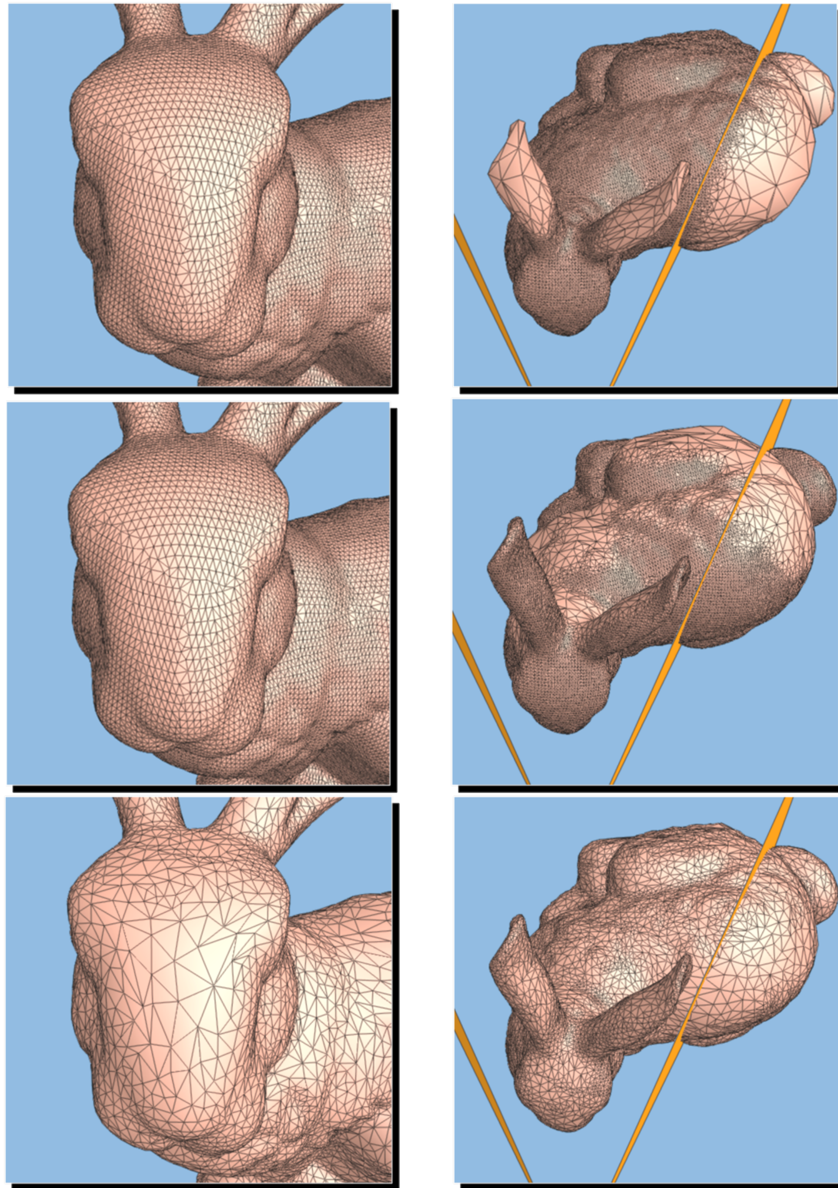


Abbildung 4.9: Vereinfachungen eines Modells mithilfe des Sichtbereichs (oben), der Flächenausrichtung (mittig) und des geometrischen Bildfehlers (unten). Links ist die Kamerasicht zu sehen, die unverändert bleibt. Die Szenensicht auf der rechten Seite zeigt die starke Reduktion von Dreiecken, durch jede der Techniken. Entnommen aus [Hop97b]

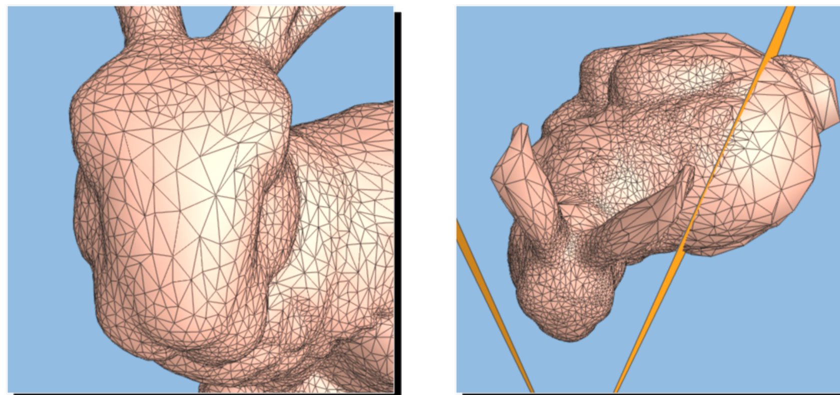


Abbildung 4.10: Kombination der Kriterien Sichtbereich, Flächenausrichtung und Geometrischer Bildfehler. Die originale Anzahl der Dreiecke wurde von 69.473 auf 10.528 reduziert. Entnommen aus [Hop97b]

Abbildung 4.10 zeigt die Kombination der drei Kriterien bei der Anzeige des Objekts. Trotz der Nähe zur Kamera kann das Mesh enorm vereinfacht werden: die Anzahl der Dreiecke wurde von 69.743 auf 10.528 reduziert. Dies wäre in einem CLOD-System unmöglich gewesen. Zudem bleibt das Modell, aus der Nutzerperspektive der Kamera, nahezu unverändert, gegenüber dem Original-Mesh.

Mithilfe dieses Ansatzes sind die Probleme des kontinuierlichen Level-of-Detail für die Anzeige von Landschaften gelöst. Es ist nun möglich für ein Mesh einen Detaillierungsgrad zu erzeugen, der nahe Abschnitte des Meshs detailliert darstellt, während das restliche Mesh gröber dargestellt wird. In der Tat erweiterte Hoppe 1998 seinen Ansatz ein weiteres Mal , um explizit die Darstellung von Landschaften zu verbessern [Hop98]. Die Implementation von Hoppe hat allerdings den Nachteil, dass sie für größere Landschaften anfangs eine allgemeinste Vereinfachung berechnen muss und die dabei ausgeführte Reihenfolge von Kantenzusammenführungen abspeichern muss. Dies erlaubt zwar eine schnelle und konsistente Ausführung des Systems, benötigt allerdings während der gesamten Laufzeit viel Speicher. Ebenfalls muss eine größere Menge an Daten zwischen CPU und GPU versendet werden. Diese Probleme wurden von Hoppe noch nicht vollständig bedacht, da GPUs erst ab dem Jahr 2000 größere Verbreitung fanden [OOS97]. Heutzutage entsteht der hauptsächliche Flaschenhals tatsächlich nicht mehr durch die Menge der Dreiecken (wobei die Reduktion dieser immer noch eins der Hauptziele ist), sondern durch die Auslastung der CPU und dem Datentransfer zwischen GPU und CPU [CR11].

Dieser Umschwung bedingt andere Strukturen. Strukturen die hierarchisch aufgebaut sind und Mengen von Dreiecken verwalten. Die Menge der Dreiecke ist dabei eine Teilmenge der Dreiecke des originalen Meshs, die Strukturen selbst werden *Tiles* genannt. Während der Berechnung des sichtabhängigen LODs wird nun nur auf den Dreiecken gearbeitet,

die durch die Tiles zur Verfügung gestellt werden. Dies reduziert die Menge an Daten die von CPU und GPU verarbeitet werden.

Die Tiles werden dabei während der Laufzeit berechnet, allerdings vor der eigentlichen Anzeige des Meshs. Durch ihren hierarchischen Aufbau ergibt sich die Möglichkeit die Tiles mittels eines Graphen zu verwalten. Die Wurzel ist dabei ein Tile, welches die Größe der gesamten Landschaft umfasst, allerdings nur eine kleine Teilmenge an Dreiecken verwaltet: die Wurzel stellt somit die gröbste darstellbare Fassung der Landschaft dar. Die Kinder der Wurzel umfassen eine identische Anzahl an Dreiecken, decken allerdings eine kleinere Fläche ab. Dadurch wird durch die Nutzung jedes Kindes das mögliche Detail der Landschaft erhöht. Der Ansatz stellt somit einen Hybrid aus DLOD- und VLOD-Systemen dar. Einerseits werden über fest definierte Tiles fixe Detaillierungsgrade bereitgestellt, andererseits werden durch die Anzeigekriterien der VLOD-Systeme die Tiles dynamisch miteinander kombiniert [CR11]. Diese LOD-Systeme sind unter dem Namen hierarchisches LOD (HLOD) bekannt.

4.3 Hierarchische Detaillierungsgrad Strukturen

Im vorherigen Kapitel wurde gezeigt, wie die Techniken der VLOD-Systeme eine optimale Anzeige von Strukturen erlauben, bei der die Verwendung klassischer DLOD-Systeme ungenügend ist. Dazu zählt auch die Darstellung von Landschaften. Weiterhin steht mit den HLOD-Systemen ein Hybrid beider Systeme zur Verfügung, der die Anzeigekriterien der VLOD-Systeme verwendet, sie allerdings primär auf Tiles anstatt auf Dreiecke anwendet. Damit ist ein optimales System zur Verwaltung großer Meshs und Datenmengen geschaffen, dass auch in der Lage ist, Höhenkarten zu verwalten.

Tatsächlich bieten sich Höhenkarten für diese HLOD-Systeme besonders an. Bei diesen muss nur eine Menge an Vertices verwaltet werden. Die Vertices müssen nicht mit einem Mesh verknüpft sein, sondern nur im Sinne eines behandelt werden. Die Höhenkarte kann daher zu einer Menge von Vertices umgerechnet werden, die daraufhin in Tiles unterteilt werden. Wird zur Laufzeit ein Detaillierungsgrad von der HLOD-Struktur angefordert, werden aus den verwendeten Tiles die Vertices entnommen und daraus, ebenfalls zur Laufzeit, ein Mesh erstellt, welches exakt dem angeforderten Detaillierungsgrad entspricht. Beispielsweise müssten dem Wurzel-Tile nur die vier Eckpunkte der Höhenkarte als Vertices mitgeteilt werden. Daraus wäre es möglich zur Laufzeit zwei Dreiecke zu bilden, die eine grobe Variante der Landschaft repräsentieren. In den folgenden Abschnitten sollen einige der möglichen Strukturen für HLOD-Systeme präsentiert werden.

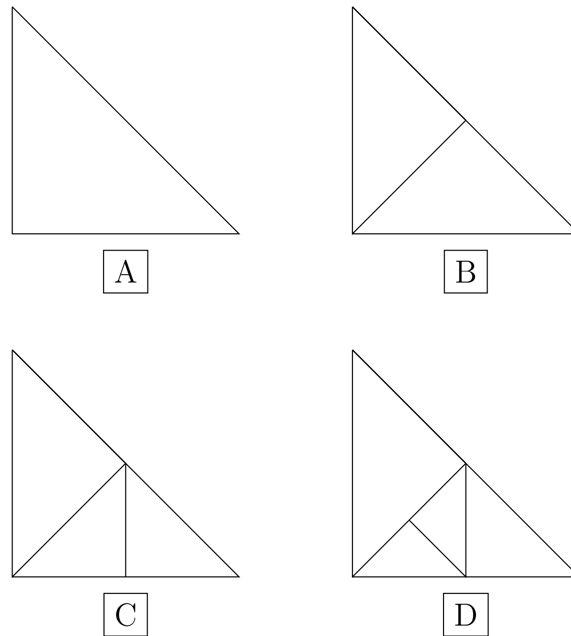


Abbildung 4.11: Binärbaum als HLOD Erzeuger. A: Niedrigster Detaillierungsgrad. B: Detaillierungsgrad um eins erhöht. C: Detaillierungsgrad eines Unterknotens erhöht. D: Detaillierungsgrad rekurisv erhöht.

4.3.1 Binärbaum

Die Nutzung ist eines Binärbaums ist die simpelste Möglichkeit eine Struktur für hierarchische Detaillierungsgrade aufzubauen. Die Wurzel des Baums ist hierbei ein gleichschenkliges Dreieck mit einem rechten Winkel [LRC+03]. Es bildet die gröbste Vereinfachung des verwalteten Meshs, da es genau durch drei Vertices definiert wird. Soll der Detaillierungsgrad erhöht werden, wird das Dreieck in zwei neue Dreiecke unterteilt. Die Trennung wird dabei zwischen Eckpunkt des rechten Winkels und dem Mittelpunkt der Hypotenuse des Dreiecks vollzogen. Der Detaillierungsgrad hat sich durch diese Teilung faktisch verdoppelt. Höhere lokale Detaillierungsgrade werden durch die rekursive Wiederholung des Vorgangs an einem der bereits unterteilten Dreiecke erzeugt. In Abbildung 4.11 wird gezeigt, wie eine Binärbaum stufenweise in seinem Detail erhöht wird. Dabei wird seine Wurzel im Zustand A angezeigt. In B wird diese auf die nächste Detailstufe erhöht. C und D führen diesen Prozess rekursiv fort, wobei jeweils ein vorheriger Elternknoten unterteilt wird, der andere verweilt im niedrigen Detaillierungsgrad. Hieran zeigt sich die Fähigkeit der hierarchischen Detaillierungsgrade, jeweils nur spezifische Stellen einer Struktur in ihrem Detail zu erhöhen.

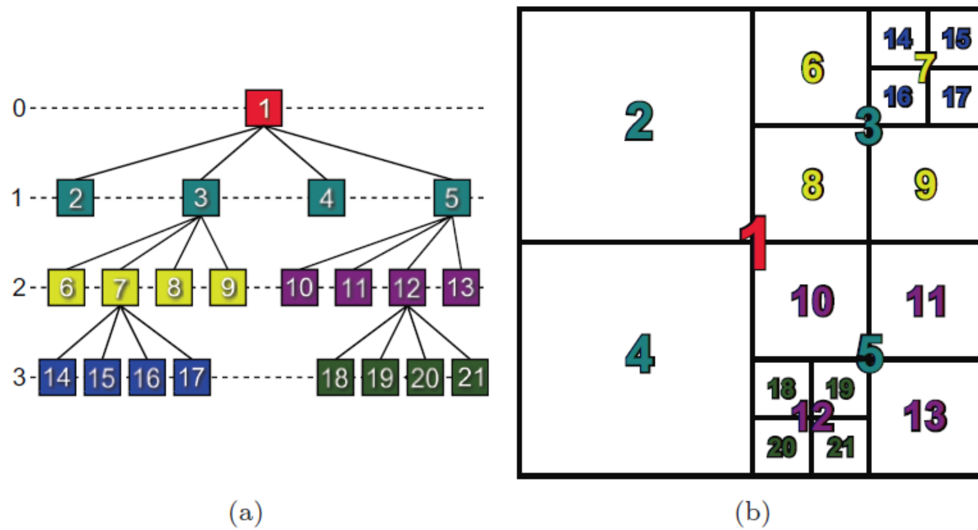


Abbildung 4.12: Schematischer Aufbau eines Quadtrees mit vier Detaillierungsgraden. Entnommen aus [CR11]

4.3.2 Quadtree

Ein Quadtree ist ein Baum, in welchem jeder innere Knoten genau vier Kinder hat [LRC+03]. Wie auch beim Binärbaum steht jeder Knoten für ein Tile. Die Wurzel repräsentiert ebenfalls die gröbste Fassung des Meshs durch mindestens vier Vertices und zwei Dreiecke. Die Wurzel des Quadtrees ist stets eine quadratische Fläche im Gegensatz zu der dreieckigen Grundfläche von Binärbaumen. Soll der Detaillierungsgrad des, durch den Quadtree verwalteten Objekts, erhöht werden, können von der Wurzel ausgehend rekursiv ein oder mehrere Kinder des jeweiligen Knotens aktiviert werden. Diese sorgen dadurch für eine detailliertere Darstellung der betroffenen Stelle. In Abbildung 4.12 ist ein schematischer Aufbau eines Quadtrees zu sehen. Dabei gilt, wie zuvor auch beim Binärbaum, dass entweder alle Kinder eines Knotens aktiv sind (wie zum Beispiel bei Knoten 3) oder keines der Kinder (wie zum Beispiel Knoten 2). Es ist nicht möglich, dass von einem Knoten nur ein, zwei oder drei Kinder aktiv sind: dies entspräche nicht der formalen Definition eines Quadtrees.

Quadtrees eignen sich insbesondere für zweidimensionale Strukturen. Daher sind sie auch optimal für die Verwaltung einer Höhenkarte, denn diese ordnen jedem Höhenwert zweidimensionale Koordinaten zu. Eine Höhenkarte kann daher über einen Quadtree leicht verwaltet werden, wenn stets eine Rasterzelle einem Blatt des Baumes entspricht. Vier nebeneinander liegende Blätter werden jeweils einem Vaterknoten zugeordnet, der somit einer höheren Skalierungsstufe des Baums, bzw. einer Vereinfachung des Meshs entspricht.

Vier nebeneinander liegende Elternknoten werden wiederum über einen weiteren Knoten gruppiert. Dies wird wiederholt, bis eine Wurzel erreicht ist.

4.4 Zusammenfassung

In diesem Kapitel wurde beschrieben, dass eine Höhenkarte ein zweidimensionales Rasterbild ist, welches für ein fest gelegtes Gebiet Höhenwerte liefert. Dabei ist ein Höhenwert immer einer speziellen Koordinate des Gebiets zugewiesen. Durch Umwandlung der Höhenwerte in Vertices lässt sich zur Laufzeit aus der Höhenkarte ein Mesh in Form des beschriebenen Gebiets erzeugen.

Weiterhin wurden verschiedene Level-of-Detail-Systeme vorgestellt. Diese basieren alle auf dem einfachen Prinzip: Erstellung, Auswahl und Austausch. Beim klassischen DLOD werden für jedes Mesh verschieden detaillierte Varianten vor der Laufzeit erstellt und zur Laufzeit lediglich gegeneinander ausgetauscht. DLOD-Systeme sind dadurch äußerst schnell, eignen sich allerdings nicht für Landschaften, da bei diesen mehrere Detaillierungsgrade gleichzeitig angezeigt werden müssen. Dieses Problem haben auch noch CLOD-Systeme, die eine Struktur bereitstellen, welche für ein Mesh zur Laufzeit beliebig viele Detaillierungsgrade erzeugen kann und somit die Probleme des Poppings, die beim DLOD existieren, behebt. Erst VLOD-Systeme erlauben es, abhängig von der Kameraperspektive einem Mesh verschiedene Detaillierungsgrade gleichzeitig zuzuweisen. Sie entfernen dabei Vertices, die außerhalb des Sichtbereichs des Anwenders liegen, oder deren Entfernung nur einen geringen Unterschied im Bild erzeugt. HLOD-Systeme nutzen die Techniken der VLOD-Systeme und verbessern diese für Landschaften durch die Verkleinerung der Problemmenge. Dies passiert, indem sie eine Vorauswahl an Vertices des Meshs treffen, die in sogenannten Tiles gruppiert werden. Die VLOD-Techniken entscheiden in HLOD-Systemen nur noch darüber, welche Tiles verwendet werden, um ein Mesh darzustellen. [CR11].

Zwei verbreitete Strukturen für HLOD-Systeme sind Binärbäume und Quadtrees. Die Funktionsweise beider ist sehr ähnlich. Bei beiden entsprechen die Knoten den Tiles in unterschiedlichen Detaillierungsgraden. Daher ist bei beiden die Wurzel die gröbste Variante des Meshs, ihre Kinder entsprechen immer detaillierteren Verfeinerungen. Dabei zerteilen die Kinder den Elternknoten immer in genau gleichgroße Untergruppen. Somit ist es möglich, bestimmten Flächen innerhalb des verwalteten Meshs einen höheren Detaillierungsgrad zuzuweisen als anderen. Der Unterschied beider Bäume ist, das Binärbäume eine dreieckige Grundfläche und stets zwei Kinder haben. Quadtrees haben eine quadrati-

sche Grundfläche und immer vier Kinder. Quadtrees eignen sich daher für die Verwaltung von Landschaften besser, wenn diese, wie im Falle einer Höhenkarte, bereits nativ als gerastertes, quadratisches Objekt vorliegt.

Kapitel 5

Die Graphensprache XL

Der eigentliche Name der Sprache XL ist *eXtended L-system language*. Dieser Name beschreibt bereits den Grundgedanken der Sprache: Es wird eine Erweiterung der klassischen Lindenmayer-Systeme (L-System) gegeben [Kni08]. In diesen Systemen wird ein Objekt über eine Zeichenketten modelliert. Alle möglichen Zeichen, die in der Zeichenkette vorkommen können, werden in einem Alphabet erfasst. Weiterhin verfügt ein L-System über eine Menge von Produktionsregeln [PHHM97]. Diese bestehen aus einer linken und einer rechten Seite, beide Seiten bestehen aus einem beliebigen Wort des Alphabets. Wird eine Produktionsregel angewendet, werden in der aktuellen Zeichenkette alle Wörter, die der linken Seite der Regel entsprechen, durch das Wort der rechten Seite ersetzt. Die Ersetzung aller Treffer findet dabei parallel und nicht sukzessiv statt. Soll mittels eines L-Systems nun beispielsweise die Entwicklung einer Pflanze beschrieben werden, ist ein Höhenwachstum problemlos beschreibbar. Ein stark vereinfachtes Beispiel: Sei mittels b ein Stil beschrieben, mit c eine Knospe und mit a eine Blüte. Eine Blüte entwickelt sich zu einem Stil mit Knospe und eine Knospe zu einer Blüte. Dann wird dies über das L-System der Gleichung 5.1 beschrieben:

$$\begin{aligned}
&\text{Alphabet} = \{a, b, c\} \\
&\text{Produktionen} = \{c \rightarrow a, a \rightarrow bc\} \\
&\text{Startwort} = \{bc\} \\
&\text{Dann gilt: } bc \rightarrow ba \rightarrow bbc \rightarrow bba \rightarrow bbbc \rightarrow ...
\end{aligned} \tag{5.1}$$

Mithilfe dieses Beispiels werden allerdings auch die Mängel eines L-Systems offenbart. Beispielsweise könnte gefordert sein, das eine Blüte zu einem Stil wird und erst in der nächsten Produktion der oberste Stil um eine Knospe ergänzt wird. Dies kann das obige System nicht leisten, die Regel $b \rightarrow bc$ würde *jedes* b ersetzen. Eine Möglichkeit ist es ein neues Zeichen d einzuführen, welches nicht zum beschriebenen Objekt gehört, son-

dern lediglich das oberste b markiert: $a \to bd, d \to c$. Eine andere Möglichkeit ist durch die Erweiterung der *kontextsensitiven* L-Systeme gegeben. Hier kann in einer Produktionsregel angegeben werden, in welchem Umfeld ein Wort vorkommen muss, damit die Regel aktiviert wird. Ein anderer Mangel der L-Systeme ergibt sich durch die Struktur dieser: sie werden über eine eindimensionale Zeichenkette dargestellt [PHHM97]. Verzweigungen, die in der Pflanzenmodellierung unbedingt erforderlich sind, müssen daher über Sonderzeichen emuliert werden, meistens werden die Zeichen [und] dafür verwendet. Es gibt eine Reihe weiterer Mängel, unter anderem: Produktionsregeln greifen immer und nicht nur zu bedingten Wahrscheinlichkeiten. Wechselwirkungen zwischen einzelnen Verzweigungen oder sogar verschiedenen Objekten können nicht beschrieben werden. Jedes weitere Sonderzeichen, dass nur existiert damit das Objekt beschrieben werden kann, erhöht Komplexität und Unübersichtlichkeit des Systems.

Unter Beachtung dieser Probleme wurde die Sprache XL entworfen, mit einem besonderen Augenmerk auf die Möglichkeit Pflanzen modellieren zu können [Kni08]. Dazu implementiert die Sprache das theoretische Konzept der relationalen Wachstumsgrammatiken (RGG). In diesem Konzept wird ein Objekt mittels eines Graphen repräsentiert. Die einzelnen Knoten des Graphen beschreiben Bestandteile des Objekts (wie die Zeichen eines L-Systems). Unter einander sind die Knoten mit Kanten verbunden, die deren Beziehung untereinander beschreiben. Beispielsweise können hier die Beziehungen der L-Systeme verwendet werden: Nachfolger und Verzweigung. Neben dem Graphen existiert eine Menge von Regeln, die angeben, wie der Graph manipuliert wird. Diese Regeln bestehen ebenfalls aus einer linken und rechten Seite, beide Seiten sind Graphen. Wird im Hauptgraph bei der Anwendung einer Regel eine linke Seite als Teilgraph aufgefunden, wird diese durch die rechte Seite ersetzt. Die Sprache XL selbst verbindet die RGG mit Java und definiert eine Syntax zur Anwendung der RGG. Dabei kann die Graph-Struktur der RGG verborgen werden, sodass die Benutzung der Sprache XL sehr der Nutzung klassischer L-Systeme ähnelt.

Im folgenden soll die grundlegende Syntax, die ein Anwender zum Manipulieren eines Objekts benutzen kann, vorgestellt werden. Zusätzlich sollen Teile der verborgenen Graph-Struktur erläutert werden, die für die Anzeige von Landschaften hilfreich sind. Zuletzt soll erläutert werden, in welchem Zusammenhang die Plattform GroIMP mit der Sprache XL steht.

5.1 Anwender-Syntax

Um die Funktionalitäten von XL unter Java nutzen zu können, empfiehlt es sich eine Funktion mit eckigen statt geschweiften Klammern geöffnet werden [Kni08]. Weiterhin muss die verwendete Klasse von *RGG* erben. Ergänzend sei hierzu Code-Ausschnitt 5.1 empfohlen.

```
1  import de.gogra.rgg.*;
2
3  public class Test extends RGG {
4      protected void init() [
5          Axiom ==> Startobjekt;
6      ]
7
8      public void run() [
9          /* Definitionsbereich der Regeln */
10     ]
11 }
```

Code-Ausschnitt 5.1: Grundgerüst des XL-Codes

Die Abbildung zeigt auch, dass in einer zweiten Funktion, der sogenannten *Init*-Funktion, das Objekt *Axiom* zu einem Startobjekt umgewandelt wird. Das Axiom existiert in jeder RGG der Sprache XL von Anfang an. In der Init-Funktion wird dieses durch ein beliebiges Objekt ersetzt. Die Funktion wird dabei vor der Ausführung aller anderen Funktionen ausgelöst und dient somit der Festlegung eines Startwertes.

Regeln von L-Systemen lassen sich mittels des Operators ==> simulieren [Kni08]. Bei diesem muss lediglich ein Objekt auf der linken und eines auf der rechten Seite der Regel stehen. Verzweigungen lassen sich mithilfe von [,] angeben. Außerdem können die *Turtle*-Befehle zum Zeichnen genutzt werden: RU, RH, RL für Drehungen und F für das Zeichnen einer Linie bestimmter Länge. Code-Ausschnitt 5.2 zeigt den Code einer Regel, der diese Zeichenfunktionen nutzt. Dabei erlaubt XL auch die Definition von Modulen. Module sind Objekte, die bereits bestehende XL-Objekte erweitern und direkt in Regeln verwendet werden können. In diesem Fall beschreibt *A* eine grüne Kugel der Größe 0,1. Die Visualisierung der mehrfachen Ausführung der abgebildeten Regel ist in Abbildung 5.1 zu sehen.

```
1 module A(float len) extends Sphere(0.1) {
2     {setShader(GREEN)}
3 }
4
5 /* ... */
6
7 public void run() {
8     A(x) ==> F(x) [RU(30) RH(90) A(x*0.8)] [RU(-30) RH(90) A(x*0.8)];
9 }
```

Code-Ausschnitt 5.2: Einfache Regel in der Sprache XL

Das Ausführen der Regeln kann dabei an Bedingungen geknüpft werden [KBHK08], zum Beispiel `A(x) (x > 0.5) ==> rechte Seite`. Die Bedingung kann dabei beliebiger Java-Code sein, solange dieser sich als boolscher Wahrheitswert auswerten lässt. Auf der rechten Seite der Regel können weiterhin Java-Kontrollfluss-Operatoren wie *if* oder *for* verwendet werden. Beliebiger Java-Code kann jederzeit mittels der geschweiften Klammern { und } ausgeführt werden. Neben der einfachen Regel zur Simulation eines L-Systems existieren allerdings noch zwei weitere.

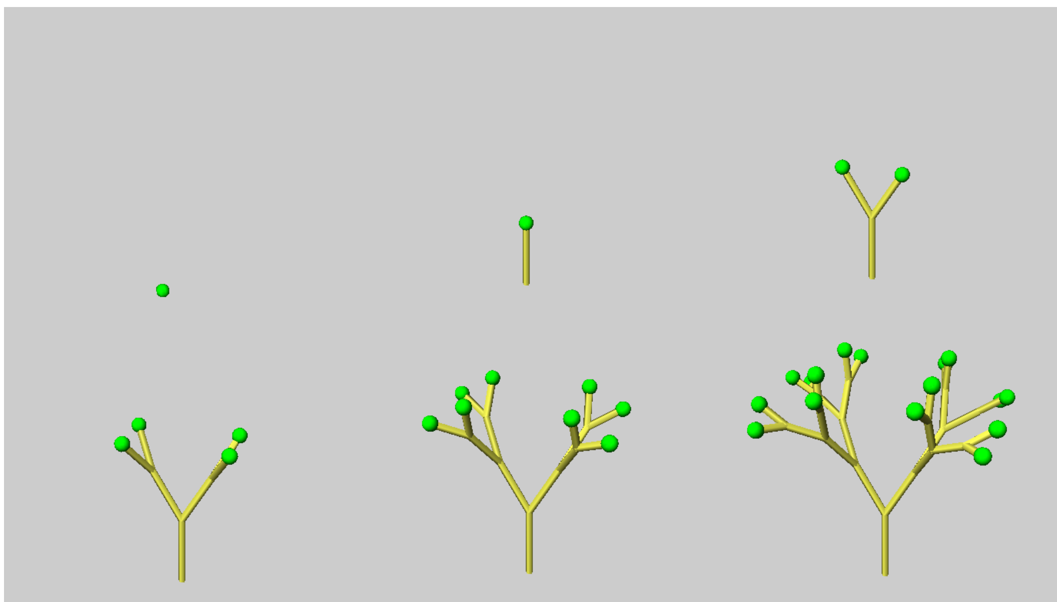


Abbildung 5.1: Fünffache Regelanwendung der Regel aus Code-Ausschnitt 5.2

Die erste Regel ist eine Verallgemeinerung des vorherigen Regelkonzepts und wird mit ==>> dargestellt [KBHK08]. Diese Regel arbeitet explizit auf dem Graphen und simuliert kein L-System mehr. Wie in Code-Ausschnitt 5.3 ersichtlich, können Objekten der

linken Seite Variablen zugewiesen werden (hier x, y, und a), werden diese auf der rechten Seite genannt, wird genau das gespeicherte Objekt durch die rechte Seite eingefügt. Wird lediglich der Objekttyp genannt (A oder X) wird eine neue Instanz des Objekts erstellt, die ersetzen Instanzen gehen verloren. In der Abbildung ist auch die Nutzung einer Bedingung zu sehen (a. enegrgy > 1) und die einer Kontrollflussanweisung. Die standardmäßige Beziehung zwischen zwei Objekten ist *Nachfolger*. Diese wird automatisch erzeugt, werden die Objekte in der Regel nur durch ein Leerzeichen getrennt. Werden sie durch eckige Klammer getrennt, wird die Beziehung *Verzweigung* erstellt. Selbsterstelle Kanten können ebenfalls eingesetzt werden, unter Nutzung von −Kantenname−>. Dies ist in Code-Ausschnitt 5.3 bei −e0−> der Fall.

```
1 x:X [a:A] y:X, (a.energy > 1) ==>>
2     x for(int i = 2; i <= a.energy; i++) ([A]) y -e0-> a;
```

Code-Ausschnitt 5.3: Komplexe Regel in der Sprache XL

Die zweite Regel unterscheidet sich von den vorherigen dadurch, dass sie nicht die Struktur des Graphen ändert, sondern lediglich für jeden Treffer, den Code der rechten Seite ausführt [Kni08]. Diese Regel wird über ::> erzeugt. Darüber können, ohne unnötigen Aufwand, Parameter des Objekts geändert werden oder Informationen angezeigt werden. Eine beispielhafte Regel findet sich unter Code-Ausschnitt 5.4.

```
1 x:Cylinder ::> {println(x);}
```

Code-Ausschnitt 5.4: Code-Regel in der Sprache XL

Diese grundlegende Benutzung der Regeln von XL reicht zur Darstellung von Landschaften aus. Komplexere Simulationen brauchen unter Umständen zusätzliche Funktionen von XL. Diese werden verständlich und umfangreich in [Kni08] erläutert.

5.2 Architektur

Die Sprache XL besitzt, wenn sie innerhalb des Programms GroIMP eingesetzt wird, drei wichtige Klassen, die für die Verarbeitung und Anzeige von Höhenkarten als Landschaften von hauptsächlichen Interesse sind. Diese sind: *Node*, *MeshNode* und *Camera*.

Eine Node repräsentiert einen Knoten eines Graphen [Uni15]. Jede Node ist standardmäßig mit einer Klasse *GraphManager* verbunden, die alle Elemente eines Graphen verwaltet. Durch diese besitzt jede Node eine, innerhalb des GraphManagers, einzigartige

Id über die sie identifiziert werden kann. Zusätzlich besitzt jede Node noch einen optionalen, arbiträren Namen. Der GraphManager speichert diese Namen in einer Map und erlaubt dadurch einen schnellen Zugriff auf die benannten Nodes. Die Node Klasse ist dabei eine Erweiterung der *Edge* Klasse, sodass sie nicht nur sich selbst darstellt, sondern auch die eingehende Kante des Knotens. Dies reduziert die Instanziierungen zusätzlicher Klassen und reduziert somit den Speicherbedarf. Die Node Klasse selbst repräsentiert keinen Bestandteil eines beschriebenen Objekts, sondern dient viel mehr als Grundlage für die spezialisierten Nodes aller Objekte. Deswegen ist sie einerseits in der Lage, beliebige Nutzerdaten zu speichern, andererseits stellt sie viele allgemeine Graphenfunktionen bereit. Hier sollen kurz die Funktionen vorgestellt werden, die für den Rest der Arbeit von Interesse sind:

- **isRoot()** – gibt an ob dieser Knoten die Wurzel des Graphen ist. Es kann eine Aussage hierüber gemacht werden, da der XL-Graph stets azyklisch ist.
- **getBranchNode(x)** – gibt die x-te Verzweigung des Knotens zurück. Die Liste der Verzweigungen ist dabei null-basiert.
- **getAxisParent()** – gibt, falls vorhanden, den Vater des Knotens zurück. Der Vater eines Knotens existiert, falls dieser Knoten von einem anderen die Verzweigung ist.
- **getBranchLength()** – gibt die Anzahl der Verzweigungen zurück.
- **getIndex()** – gibt den momentanen Indexwert eines Knotens zurück. Der Index eines Knotens ist dabei die Position des Knotens innerhalb der aktuellen Verzweigung.
- **getSuccessor()** – gibt den Nachfolger eines Knotens zurück. Der Nachfolger ist dabei standardmäßig der Knoten der selben Verzweigung mit einem Index genau um 1 höher als der Index des momentanen Knotens.
- **getPredecessor()** – gibt den Vorgänger eines Knotens zurück. Funktioniert äquivalent zu obiger Funktion.
- **setName()** – setzt den Namen des Knotens.
- **getId()** – gibt die einzigartige Id des Knotens zurück.
- **appendBranchNode(node)** – Fügt den Knoten node dem aktuellen Knoten als Verzweigung an letzter Stelle hinzu.
- **removeFromChain()** – Entfernt den Knoten aus dem Graphen und löscht seine Beziehungen zu anderen Knoten.

$$\begin{pmatrix} Blick_x & Oben_x & Seite_x & 0 \\ Blick_y & Oben_y & Seite_y & 0 \\ Blick_z & Oben_z & Seite_z & 0 \\ -Pos_x & -Pos_y & -Seite_z & 1 \end{pmatrix}$$

Abbildung 5.2: Sicht-Transformationsmatrix einer Kamera. *Blick* entspricht der Blickrichtung, *Oben* dem Höhenvektor und *Seite* dem Seitenvektor.

Diese Funktionen erlauben es, durch die initiale Nutzung nur einer Node-Klasse einen beliebig komplexen Graphen aufzubauen. Für die Anzeige eines Polygon Meshs können sie allerdings nicht genutzt werden. Hier muss auf die Klasse *MeshNode* zurückgegriffen werden. Diese erlaubt es, ein solches Mesh innerhalb des Graphen zu repräsentieren. Dazu erbt die Klasse einerseits die Funktionen der Node-Klasse, um die selben Graph-Funktionalitäten anbieten zu können, ergänzt diese allerdings noch um die Funktion *draw* zum Anzeigen einer referenzierten Klasse des Typs *Polygons*. Hier kann unter anderem auf die *PolygonMesh*-Klasse referenziert werden, der eine Menge an Vertices und Normalenvektoren zugewiesen werden, sowie eine Liste von Indizes der Verticesmenge, wobei je drei aufeinander folgende Indizes ein Dreieck des Meshs definieren [Uni15]. Die PolygonMesh-Klasse übernimmt die Erzeugung des eigentlichen Meshs.

Die letzte wichtige Klasse ist die *Camera*-Klasse. Diese definiert eine Kamera für die Szene [Uni15]. Für diese Arbeit ist dabei nicht die Funktionsweise der Kamera wichtig, denn diese entspricht genau der Herangehensweise aus Kapitel 3, sondern die Eigenschaften der Kamera: Position, Blickrichtung, Höhenvektor, Seitenvektor und das Blickfeld. Das Blickfeld beschreibt den Öffnungswinkel der pyramidenförmigen Kamerasicht auf die Szene. Die Blickrichtung gibt mittels eines Vektors an, in welche Richtung die Kamera ausgerichtet ist. Der Höhenvektor legt die senkrechte Kameraachse fest. Grob gesprochen wird festgelegt wo *oben* liegt. Der Seitenvektor funktioniert äquivalent und beschreibt die horizontale Kameraachse. Mittels dieser drei Vektoren kann das Sicht-Koordinatensystem der Kamera in Abhängigkeit zum kartesischen Koordinatensystem der Szene beschrieben werden. Weiterhin wird über die *Position*, die Position der Kamera innerhalb des kartesischen Koordinatensystems beschrieben. Die Camera-Klasse speichert alle vier Vektoren in einer 4x4-Matrix, wie sie in Abbildung 5.2 dargestellt ist. Diese Matrix entspricht der Transformationsmatrix, die benötigt wird, wenn ein Objekt mit Weltkoordinaten in den Kameraraum transformiert werden soll. Zu beachten ist dabei, dass die Positionskoordinaten negiert gespeichert werden. Weiterhin besitzt die Camera-Klasse noch die Eigenschaften *minZ* und *maxZ*. Diese geben an, welchen minimalen, bzw. maximalen Abstand ein Objekt von der Kamera haben darf, damit es gerendert wird. Die beiden Werte entsprechen damit der Near und Far Clipping Plane.

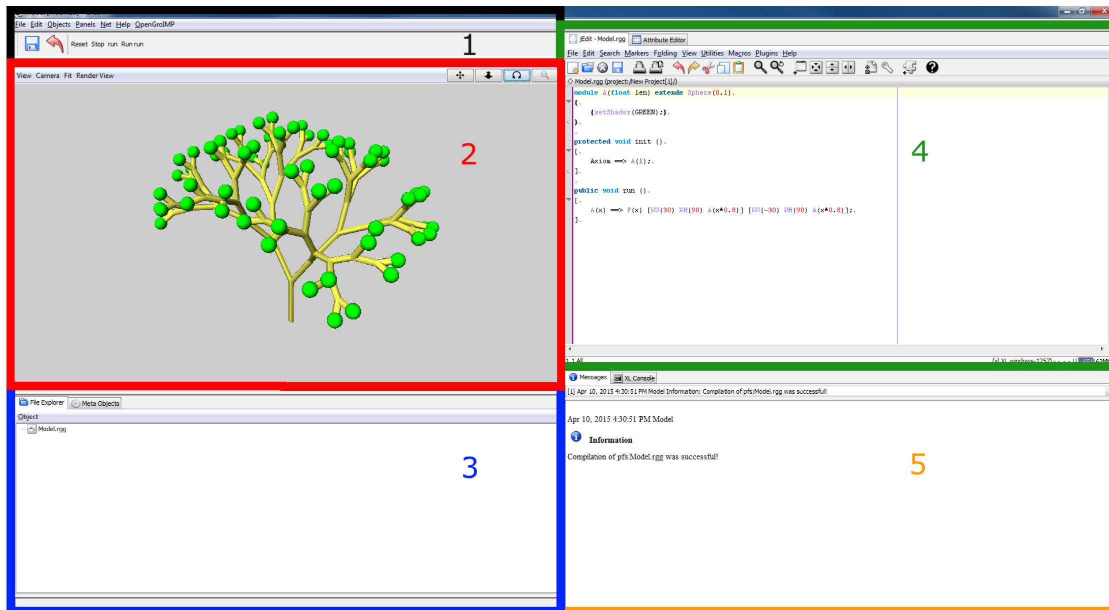


Abbildung 5.3: Oberfläche der Plattform GroIMP. 1: Menüleiste und Methodenanwendung. 2: Szenenansicht. 3: Projektdateien. 4: jEdit XL-Code-Editor. 5: Programmmeldungen und XL-Konsole.

5.3 GroIMP

Die Open-Source Modellierungsplattform GroIMP (Growth-grammar related Interactive Modelling Platform) ist eine Java-Anwendung zur Modellierung realistischer Wachstumsprozesse [Hen13]. GroIMP ist dabei hauptsächlich eine Visualisierungs- und Bearbeitungsplattform für die Sprache XL. In ihr können XL-Projekte verwaltet werden und ihr Code editiert werden. Weiterhin können Regeln fortlaufend oder schrittweise angewendet werden. Das momentane Aussehen des beschriebenen Objekts wird mittels eines bereits implementierten Renderers dargestellt.

Abbildung 5.3 zeigt, wie die verschiedenen Bereiche des Programms (Code-Editor, Rendering und Projektansicht) in einer Oberfläche zusammengeführt werden [Hen13]. Die rot umrahmte Szenenansicht erlaubt dabei nicht nur die Betrachtung des initialen Zustands, sondern auch eine die direkte Manipulation des Objekts, die Wahl verschiedener Ansichten (Drahtgitter, OpenGL, POV-Ray) und die Manipulation der Kamera (bspw.: ein anderer Sichtbereich, Bewegen durch die Szene). Über die schwarz umrahmte Menüleiste können zudem *public void*-Funktionen ausgeführt werden, die im Code in XL formuliert sind. Dadurch können Regeln direkt auf das gerenderte Objekt angewendet und sichtbar gemacht werden.

GroIMP hat aus Entwicklersicht einige Vorteile. Es erlaubt die direkte Beschäftigung mit

der Implementation des gewünschten XL-Codes. Allgemeine Voraussetzungen hingegen, wie die Verwaltung des XL-Graphens, die Kompilation des Codes oder das Rendering und Anzeigen des beschriebenen Objekts, übernimmt GroIMP. Ein Anwender der Plattform muss sich somit nur noch mit seiner eigenen Problemstellung beschäftigen und nicht mit der internen Funktionsweise von Prozessen der Computergrafik.

Kapitel 6

Umsetzung

Nachdem sich die vorherigen Kapitel mit der Klärung theoretischer Vorkenntnisse und praktischer Voraussetzungen beschäftigt haben, wird sich dieses Kapitel mit der Umsetzung des Ziels der Arbeit beschäftigen. Dafür sollen eingangs die zu erreichenden Ziele noch einmal wiederholt werden und mit den bestehenden Voraussetzungen abgeglichen werden. In einem zweiten Schritt wird ein theoretisches Konzept gegeben, wie die Ziele erreicht werden können. Danach soll der Programmcode, der zur Erreichung der Ziele erstellt wurde, im Sinne einer Dokumentation erläutert werden. Ebenfalls soll eine Leistungsbewertung der Implementation erstellt werden. Abschließend werden die Installationshinweise genannt und der Code wird an einem Beispiel demonstriert.

6.1 Ziele und Voraussetzungen

Das Ziel der Arbeit ist die Implementierung einer Erweiterung für die Plattform GroIMP, die es ermöglicht reale geografische Landschaften darzustellen. Dabei soll es möglich sein, die verbreiteten Formate GeoTIFF und DTED mittels der Erweiterung zu importieren. Nachdem Öffnen sollen die Daten in ein anzeigbares Mesh konvertiert werden. Damit eine Echtzeit-nahe Darstellung möglich ist, soll eine LOD-Struktur für die geladene Landschaft erstellt werden, die dafür sorgen soll, dass die Landschaft nur im direkten Sichtbereich des Anwenders hoch detailliert ist.

Mit GroIMP als vorausgesetzte Zielplattform stehen zusätzliche Hilfen zur Erfüllung des Arbeitsziel zur Verfügung. Es ist nicht nötig einen eigenen Renderer zur Darstellung der Landschaft zu entwerfen, stattdessen kann auf den bereits implementierten Renderer von GroIMP zurückgegriffen werden. Da GroIMP ebenfalls die Graphenstruktur von XL nutzt, können die Knotenklassen *Node* und *MeshNode* als Grundlage für die Kno-

ten eines HLOD-Ansatzes verwendet werden. Dies erlaubt auch die Nutzung der Zusatzfunktionen von XLs Graph-Manager. Durch die Anforderung das GeoTIFF- und DTED-Dateien eingelesen werden sollen, ergibt sich die Notwendigkeit einer programminternen Darstellungsmöglichkeit von Höhenkarten, die für jeden Import die gleiche Struktur aufweist.

6.2 Konzept

Das Konzept welches für die Implementierung erstellt wurde ist in drei Hauptkategorien untergliedert. Diese leiten sich direkt aus den eingangs genannten Zielen ab. Die erste Kategorie ist *Import der Rasterformate* und beschäftigt sich mit dem Problem, die GeoTIFF- und DTED-Dateien zu laden und in eine einfache Darstellungsform umzuwandeln, die im restlichen Programm verwendet werden kann. Die zweite Kategorie, *Erstellung des Meshs*, dient dem Zweck mittels gegebener Höhendaten ein Mesh zu erstellen, welches wie jedes andere XL-Modul vom Anwender in GroIMP verwendet werden kann. Die letzte Kategorie ist *Aufbau der HLOD-Struktur* und dient der Lösung des Hauptproblems der Arbeit: Die Darstellung einer Landschaft, ohne einerseits den Hardware-Speicher übermäßig zu belasten und andererseits die Darstellung großer Landschaften mit keiner oder nur kurzer Wartezeit zu ermöglichen. Dazu soll eine passende HLOD-Struktur entworfen werden.

6.2.1 Import der Rasterformate

Der Import der GeoTIFF- und DTED-Dateien stellt eine Schwierigkeit dar. Diese ist darin begründet, dass die meisten Dateien dieser Formate serialisiert sind, sie können daher nicht direkt ausgelesen werden. Möglich wäre es, GeoTIFF-Dateien als normale Bilder zu laden. Die gesamten Metadateien gingen dabei jedoch verloren. Bei DTED steht dieses Verfahren nicht zur Verfügung, da dieses Format in keiner Standard-Javaklasse als Bild geladen werden kann. Zum Laden der beiden Formate müssten somit neue Import-Routinen erstellt werden. Dies würde darüber erfolgen, die Spezifikation mittels Java genau nachzubilden und dadurch die Dateien in das Programm laden zu können. Diese Problemlösungsstrategie ist allerdings ungeeignet, aus folgenden Gründen: Das Erstellen der Import-Routinen könnte mehrere Wochen in Anspruch nehmen und wäre danach immer noch sehr fehleranfällig. Weiterhin wäre das Laden aller GeoTIFF-Dateien nicht garantiert, da es hier auch Sub-Spezifikationen (*Dialekte*) gibt, die von den erstellten Import-Routinen nicht korrekt geladen werden könnten. Damit gewährleistet werden könnte, dass

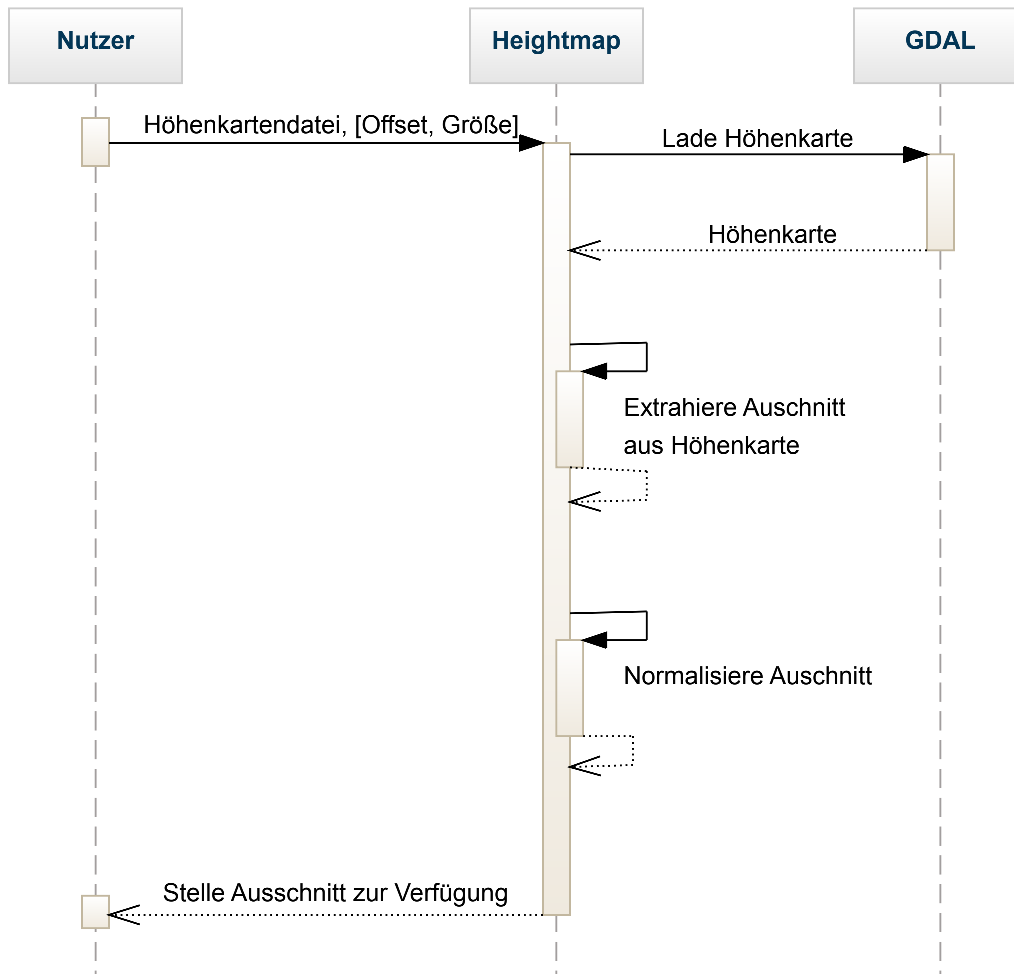


Abbildung 6.1: Ablauf des Import-Schrittes

alle Dateien aller Spezifikationen geladen werden könnten, wäre eine deutliche Mehrarbeit nötig, die als Einzelperson praktisch nicht bewältigbar ist.

Aus diesem Grund wurde für die Arbeit entschieden, eine Klassenbibliothek aus dritter Hand zu verwenden. Eine Java-Klassenbibliothek liegt als *jar*-Datei vor. In dieser Datei befinden sich bereits kompilierte Java-Klassen. Wird eine Bibliothek in ein Projekt eingebunden, können die bereitgestellten Klassen auf dem selben Weg verwendet, wie die der Standard-Java-API. Bei der verwendeten Bibliothek handelt es dabei um die Geospatial Data Abstraction Library (GDAL). Das Ziel der Bibliothek ist es, Entwicklern zu erlauben, beliebige raster- oder vektorbasierte Geodaten verwenden zu können, ohne dass für diese Unterschiede in der Benutzung der Bibliothek entstehen [Ope15]. Dies geschieht dadurch, dass GDAL den Import der einzelnen Formate bereits implementiert hat und diese in ein, stets gleich aufgebautes, Datenmodell umwandelt. Der Entwickler kann somit mit dem abstrakten Datenmodell arbeiten und muss nicht mehr beachten, welches Format geladen wurde.

Durch die Verwendung dieser Bibliothek vereinfacht sich der Arbeitsschritt des Imports zu einigen wenigen Befehlen. GDAL erlaubt dabei nicht nur den Import von GeoTIFF und DTED, sondern ebenfalls von 137 weiteren Formaten. Formal gesehen handelt es sich bei GDAL um ein Open-Source Projekt unter der MIT Lizenz, eine Verwendung ist daher bedenkenlos möglich. GDAL ist ursprünglich eine Plattform-unabhängige C++-Bibliothek, durch die Verwendung von SWIG wurde allerdings eine Java-Schnittstelle bereitgestellt.

Der erwünschte Ablauf dieses Schritt ist in Abbildung 6.1 beschrieben. Der Nutzer soll den Dateipfad zu einer Höhenkarte angeben. Optional kann er noch ein *Offset* und eine *Größe* bestimmen. Die Größe gibt an, wie groß der Ausschnitt sein soll, der aus der Höhenkarte entnommen werden soll. Es muss sich dabei um ein Vielfaches von 2 + 1 handeln. Das Offset bestimmt, von welchen Koordinaten an der Ausschnitt erstellt wird. Sind die Werte nicht gegeben, wird das Offset als $(0, 0)$ behandelt und als Größe, das größte Vielfache von 2 (+ 1) gewählt, welches kleiner als Breite und Länge der Höhenkarte ist. Die Höhenkarte wird danach mittels GDAL geladen. Von der geladenen Höhenkarte wird ein Ausschnitt gemäß der obigen Spezifikationen erstellt, der daraufhin normalisiert wird. Dabei werden die Höhenwerte standardmäßig auf Werte zwischen 0 und 1 gesetzt. Der normalisierte Ausschnitt wird abschließend dem restlichen Programm zur Verfügung gestellt.

6.2.2 Erstellung des Meshs

Durch die Nutzung der MeshNode-Klasse der Sprache XL, ist die Erstellung eines anzeigbaren Meshs aus einer Höhenkarte leicht möglich. Zu Beginn wird der Ausschnitt der Höhenkarte des vorherigen Schritts übernommen und in eine Menge von Vertices umgewandelt, die Vertices werden daraufhin mit Kanten verbunden, sodass ein Mesh entsteht. Der Ablauf der Umwandlung ist dabei genau der in Abschnitt 4.1 (Seite 17) beschriebene. Der Klasse PolygonMesh können Vertices und Indizes übergeben werden. Die Anzeige des Meshs wird von dieser Klasse übernommen, wenn es in einer MeshNode referenziert wird. Die Indexliste wird dabei unter der Ausnutzung einer HLOD-Struktur erstellt, sodass das Mesh mit der minimalen Menge an Vertices erstellt wird.

Damit die Belichtung des Meshs korrekt ist, müssen außerdem Normalenvektoren berechnet werden. Im Falle von GroIMP und XL werden dabei Normalenvektoren für jeden Vektor des Meshs gefordert, nicht für jede Fläche des Meshs. Um diese zu erhalten, müssen in einem ersten Schritt alle Vertices eine Liste an Flächen zugeordnet bekommen, in welchen das jeweilige Vertex als Eckpunkt beteiligt ist. Dann werden für die (dreieckigen) Flächen die Normalenvektoren gemäß Gleichung 6.1 berechnet [Ope13]:

$$\vec{u} = \vec{b} - \vec{a}$$
$$\vec{v} = \vec{c} - \vec{a} \quad (6.1)$$
$$\vec{n} = \vec{u} \times \vec{v}$$

mit den gegeben Vertices der Fläche $\vec{a}$, $\vec{b}$ und $\vec{c}$, den Kanten der erzeugten Fläche $\vec{u}$ und $\vec{v}$ und dem daraus berechenbaren Normalenvektor der Fläche $\vec{n}$. Im Anschluss daran werden die Vertex-Normalenvektoren als Durchschnitt aller Flächen-Normalenvektoren, an denen das jeweilige Vertex beteiligt ist, berechnet. Mathematisch mit Gleichung 6.2 ausgedrückt:

$$\vec{n} = \frac{\sum_{i=1}^{m} n_i}{|\sum_{i=1}^{m} n_i|}, \quad (6.2)$$

mit $\vec{n}$ als Vertex-Normalenvektor und $\vec{n_i}$ den Normalenvektoren der Flächen, die das Vertex nutzen. Der Vorteil dieser Berechnungsmethode liegt darin, dass sie einerseits sehr präzise Resultate liefert (bspw. sind Ergebnisse der *Forward Difference*-Method qualitativ deutlich minderwertiger) und andererseits sehr schnell ausgeführt werden kann [JS02]. Weiterhin kann sie für jede beliebige Mesh-Struktur eingesetzt werden, ohne dass die Qualität stark beeinflusst wird.

6.2.3 Funktion der HLOD-Struktur

Als HLOD-Struktur wurde innerhalb dieser Arbeit ein QuadTree als Grundlage gewählt. Die Wahl fiel auf diese Struktur, da sie ein schnelles Traversieren ermöglicht, einfach zu benutzen und leicht zu visualisieren ist. Zudem sind QuadTrees sehr populär, weswegen genügend Literatur zu ihrer Erstellung und Optimierung existiert. Außerdem bieten sie sich wegen ihrer quadratischen Grundfläche für eine quadratische Höhenkarte besonders an.

Die Grundidee liegt hierbei darin, dass jeder Knoten des Baums ein Tile der HLOD-Struktur beschreibt. Die Wurzel steht dabei für das gröbste Tile des Meshs. Besitzt eine Knoten vier Kinder, sind dies Unterteilungen des groben Tiles in vier feinere Varianten. Der maximale Detaillierungsgrad eines Tiles ist genau dann erreicht, wenn die Vertices die es umfasst in der Höhenkarte exakt nebeneinander liegen. In diesem Fall kann keine

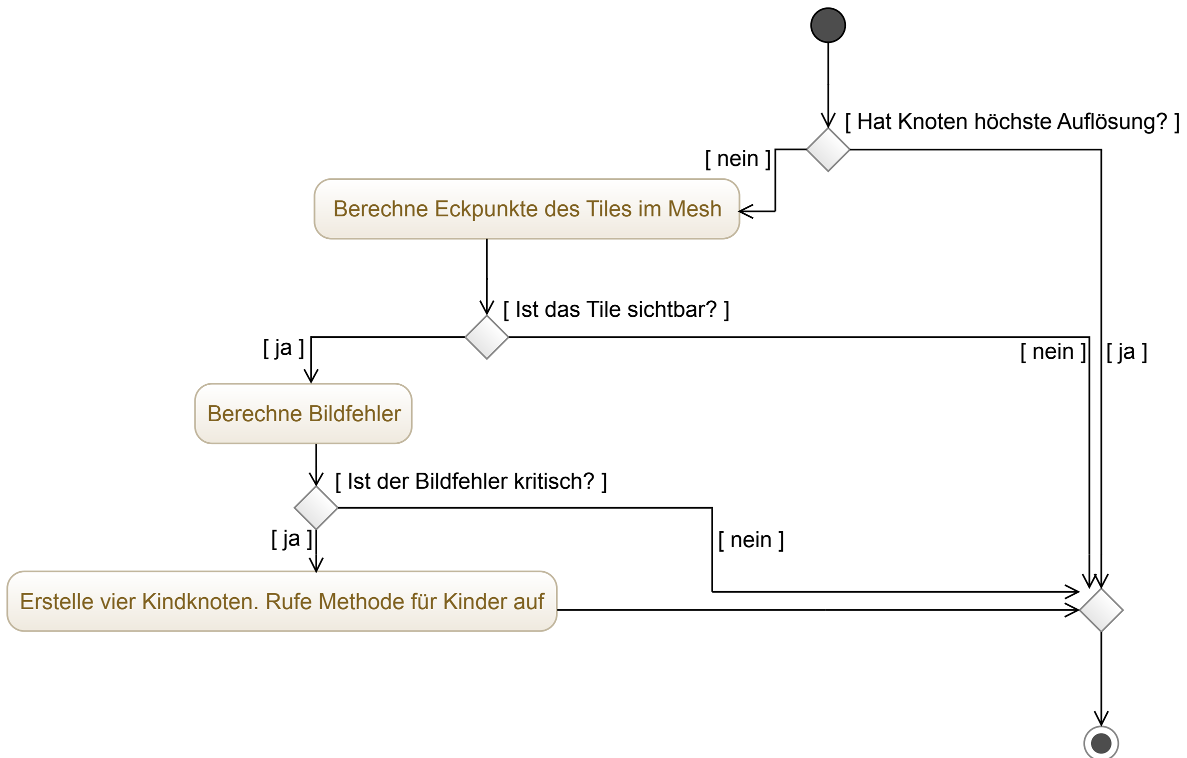


Abbildung 6.2: Aufbau und Aktualisierung des Quadtrees

weitere Verfeinerung mehr möglich sein, es sei denn, es würden prozedurale Verfeinerung erzeugt werden – durch diese wäre allerdings keine Beschreibung eines realen Orts mehr gegeben.

Die Struktur dient der Erfüllung zweier Aufgaben. Erstens sollte sie in der Lage sein, bei Änderungen der Kameraperspektive, den verwalteten Quadtree insofern zu optimieren, als dass neue Knoten hinzugefügt werden, wo sie benötigt werden, und dass alte Knoten an Positionen entfernt werden, an denen sie nicht mehr benötigt werden. Diese Aktualisierung ist nötig um den Speicher möglichst wenig zu belasten und um die Problemmenge (Menge der zu verarbeitenden Vertices) für nachfolgende Prozesse zu minimieren. Zudem sollte die Struktur auf Anforderung des Mesh-Erstellungs-Prozesses eine Indexliste erstellen. Ein Index in dieser Liste steht für einen Array mit diesem Index in der Liste der Vertices. Über drei nacheinander folgende Indizes wird somit ein Dreieck konstruiert. Das Ziel ist es somit, auf Anforderung eines anderen Arbeitsschrittes, eine, im Detaillierungsgrad für die momentane Szenenansicht passende, Vereinfachung des Meshs zu erzeugen.

Abbildung 6.2 beschreibt den konzeptuellen Vorgang der Aktualisierung, bzw. Erstellung des Quadtrees. Die Methode ist darauf ausgelegt, an einer Wurzel aufgerufen zu werden und breitet sich daraufhin rekursiv durch den Baum aus. In einem ersten Schritt muss getestet werden, ob der Knoten bereits den maximal möglichen Detaillierungsgrad re-

präsentiert. In diesem Fall muss der Verfeinerungsprozess abgebrochen werden, da keine weitere Erhöhung des Detaillierungsgrades möglich ist. Anderenfalls werden die vier Eckpunkte des Knoten berechnet, die dieser in einer Repräsentation des Baums als LOD-Mesh hätte. Liegt die Fläche des Tiles, die durch diese Punkte beschrieben wird, außerhalb des Sichtbereichs des Anwenders, ist es unnötig, den Knoten weiter zu verfeinern. Zur Leistungssteigerung wird in diesem Fall der Prozess ebenfalls abgebrochen. Zur Ausführung dieses Tests ist es nötig, den linken, unteren Eckpunkt und den rechten, oberen Eckpunkt in den normalisierten Sichtkubus der Kamera zu transformieren. Liegen die Koordinaten beider Punkte mit ihrer x- oder y-Komponente *entweder* über 1 *oder* unter -1, ist garantiert, dass die Fläche im Bild nicht sichtbar sein wird. Für den Fall, dass das Tile sichtbar ist, wird der Bildfehler eines Vertex im Tile berechnet. Abhängig von der Wahl des Vertexs kann die Qualität der Mesh-Verfeinerung variieren. Der Bildfehler wird unabhängig vom Punkt mit Gleichung 6.3 berechnet [CR11].

$$\omega = \frac{\epsilon x}{2d \tan \frac{\theta}{2}}, \tag{6.3}$$

mit ω als errechneter Bildfehler, d der euklidischen Distanz von der Kamera zum gewählten Vertex, θ dem Blickwinkel der Kamera, x der Breite der Anzeigefläche und ϵ dem Höhenfehler des Vertexs. Im Falle von GroIMP können x und θ aus der Kameraklasse extrahiert werden. Lediglich der Höhenfehler ϵ muss berechnet werden. Dieser entspricht der Höhendifferenz eines Vertex und der Position des Vertex, in einem Mesh, in der das Vertex nicht verwendet wird [Sev02]. Mathematisch,

$$\epsilon = |v_z - f(v_x, v_y)|, \tag{6.4}$$

mit ϵ dem Höhenfehler, v_z der Höhe des Vertexs, v_x und v_y der Position des Vertexs und f einer Funktion, die die Höhe eines Meshs *ohne* $\vec{v}$ an einer gegebenen Position berechnet.

Liegt der Bildfehler über einer arbiträr gesetzten Schwelle, werden dem Knoten vier Kindesknoten hinzugefügt und die Methode wird ebenfalls für jedes Kind aufgerufen. Anderenfalls wird die Methode beendet. Das Bildfehler-Kriterium unterscheidet sich dabei stark vom einfachen Distanz-Kriterium: verringert sich der Abstand des Vertexs zur Kamera, steigt nicht zwangsläufig der Bildfehler. Stattdessen ist die Distanz nur eine Komponente, die den Bildfehler erhöht oder senkt, da ein Objekt, welches näher zur Kamera liegt mehr Platz auf der Anzeige einnimmt, als ein weit entferntes Objekt. Da der Fehler in Pixeln gemessen wird, sorgt so schon eine kleine Änderung an einem nahen Objekt für eine große

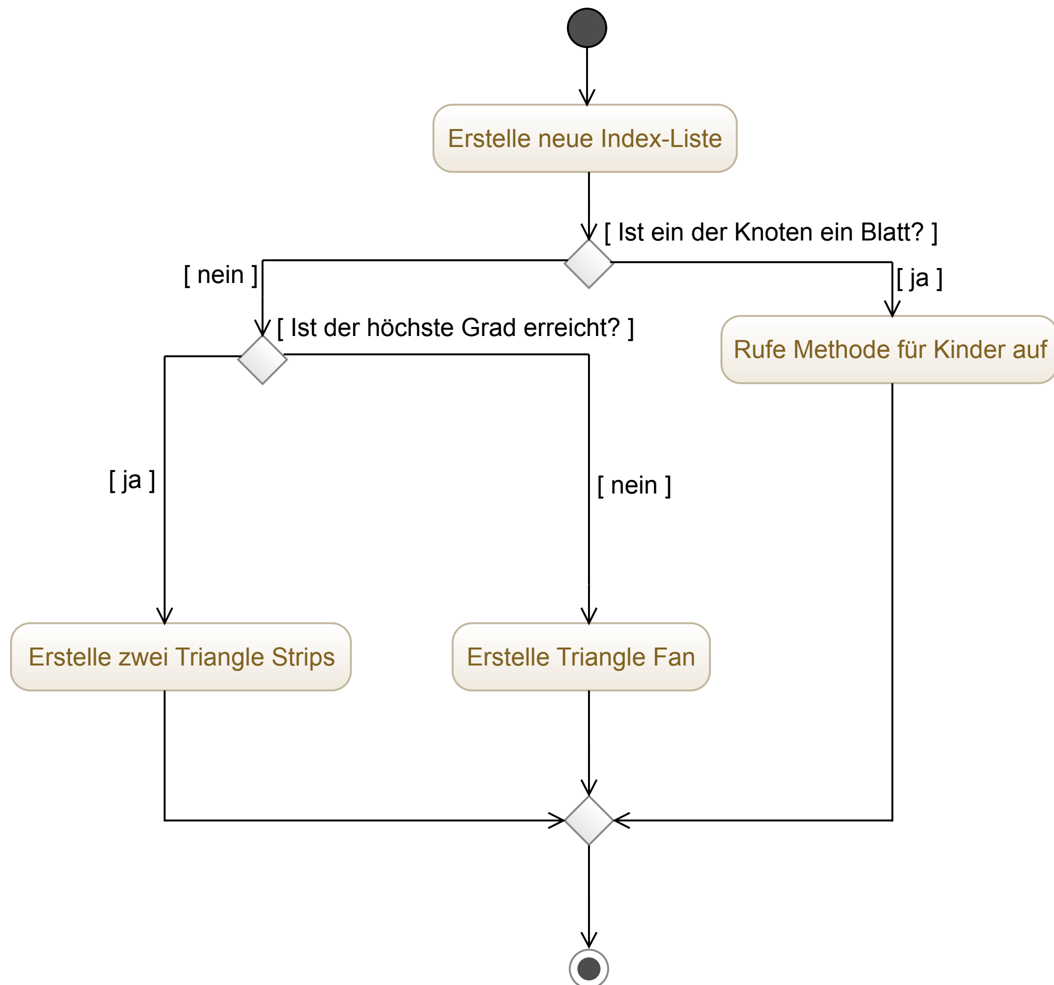


Abbildung 6.3: Index-Erstellung für das LOD-Mesh

Veränderung des Bildfehlers. Neben dieser Komponente existiert mit dem Höhenfehler allerdings noch eine zweite, welche unabhängig von der Entfernung fungiert. Hier ist die Struktur des Objekts relevant: eine flache Ebene mit nur kleinen Höhenvariationen wird nur geringe Höhenfehler ergeben, eine zerklüftete Landschaften sehr hohe. Zusammen wird ein Kriterium gebildet, welches sowohl Struktur als auch Entfernung des Objekts beachtet. Die Formel in Gleichung 6.3 ist dabei formal nur für die Mitte des Sichtkegels korrekt, zu den Rändern wird das Ergebnis unpräziser. Für die Praxis reicht die Qualität allerdings aus, vor allem da der Berechnungsvorgang sehr schnell erfolgen kann [CR11].

Abbildung 6.3 beschreibt den Vorgang der Erstellung der Index-Liste für die Mesh-Erstellung. Zu Beginn wird eine neue Index-Liste erstellt, die nur das Teil-Mesh des aktuellen Knotens (bzw. Tiles) beschreibt. Falls es sich um einen inneren Knoten handelt, soll der Knoten keine eigenen geometrischen Informationen erzeugen, sondern nur die selbe Methode für seine vier Kindesknoten aufrufen und nach deren Beendigung die von ihnen erhaltenen vier Teillisten zusammenfügen. Ist der Knoten allerdings ein Blatt,

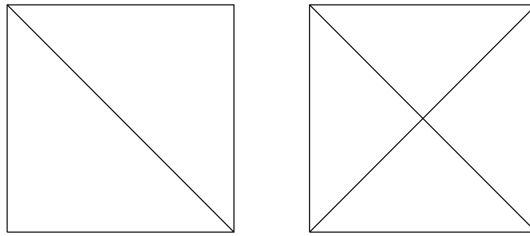


Abbildung 6.4: Triangle-Strip und Triangle-Fan. Ein Strip entsteht durch Nebeneinanderlegen der Dreiecke. Bei einem Fan besitzen jeweils vier Dreiecke ein zentrales Vertex, welches den Mittelpunkt bildet.

sollen geometrische Informationen erzeugt werden. Hier muss allerdings noch einmal unterschieden werden: ist der Knoten auf dem höchst-möglichen Detaillierungsgrad, soll in Form der Höhenkarte ein *Triangle Strip* erstellt werden. Beschreibt der Knoten hingegen eine gröbere Variante des Meshs, soll ein *Triangle Fan* mit den vorliegenden Vertices erzeugt werden. Beide Formen sind in Abbildung 6.4 ersichtlich. Der Triangle-Fan wird verwendet, weil es mit dieser Struktur in der Praxis leichter ist, Verfeinerungen fehlerfrei durchzuführen. In allen Fällen wird am Ende der Methode die erstellte Index-Liste an den aufrufenden Prozess zurückgegeben. Deutlich erkennbar agiert diese Methode rekursiv. Zum Erstellen des kompletten LOD-Meshs ist es daher nur notwendig, die Methode an der Wurzel des QuadTrees aufzurufen. Am Ende liegt somit eine einzelne Index-Liste vor, aus der ein Mesh erstellt werden kann. Dies hat mehrere Vorteile: Die Renderingzeit verkürzt sich, da nur ein Draw-Call für ein einzelnes Mesh und nicht für viele kleine erfolgen muss. Weiterhin werden Grafikfehler verhindert, denn bei mehreren kleinen Meshs könnten Rundungsfehler der Positionsangaben (im Rendering) dazu führen, dass für den Anwender *Risse* in der Landschaft sichtbar wären.

6.3 Implementation

Zwar beschreibt das vorherige Kapitel bereits die theoretische Funktionsweise, jedoch ist auch eine Beschreibung der praktischen Funktionsweise, der Implementation der Erweiterung, notwendig. Dies liegt daran, dass in der Praxis einerseits Details und Fehlerquellen existieren, die in einem Konzept nicht von Belang sind. Andererseits sollen die folgenden Abschnitte als erläuternde Dokumentation der Funktionalität und Methoden der Erweiterung dienen. Im Grunde wurden zwei neue Java-Klassen erstellt. Die erste ist die *LodQuadNode*, die die Node-Klasse erweitert und die einzelnen Nodes innerhalb des QuadTrees repräsentiert. Sie wurde zur Ausführung der rekursiven Methoden des Konzepts erstellt. Die zweite Klasse ist die *LodQuadTree*-Klasse. Diese erweitert

die MeshNode-Klasse. Somit kann sie visualisiert werden und ohne unübliche Benutzung in XL verwendet werden. Sie bildet dadurch einerseits die Schnittstelle zum Anwender und andererseits verwaltet sie das fertiggestellte Mesh. Sie übernimmt den Import der Höhenkarte mit GDAL, die Erstellung und Anzeige des Meshs und den Aufruf der rekursiven Funktionen der LodQuadNode-Klassen. Um die Übersichtlichkeit dieses Kapitels zu gewährleisten, wird das Einbinden von Code-Abschnitten möglichst minimal gehalten. Vielmehr soll die Funktionsweise der einzelnen Funktionen erläutert werden, insbesondere in Situationen, in denen diese vom Konzept abweichen. Eine *technische* Dokumentation, die auch Zusatzfunktionen erläutert, ist mit dem Java-Doc auf dem dieser Arbeit beiliegenden Datenträger gegeben.

6.3.1 LodQuadTree

Die Klasse LodQuadTree dient, wie bereits erwähnt, der Visualisierung der Landschaft in XL. Sie ist damit einerseits die Schnittstelle zum Benutzer, der diese Klasse zum Laden und Manipulieren von Landschaften verwenden kann, andererseits repräsentiert sie die Landschaft, sowohl als Mesh, wie auch als QuadTree, innerhalb des eigentlichen XL-Szenegraphen. Die Knoten des verwalteten QuadTrees sind somit nicht direkt selber Teil des XL-Graphens, sondern werden über die LodQuadTree-Klasse in diesem abstrahiert.

```
1  public LodQuadTree(String dataPath, float xStretch, float yStretch,
2                     float zStretch, float zMax, float zMin, int maxSize,
3                     int offsetX, int offsetY) {...}
4
5  public LodQuadTree(String dataPath, float xStretch,
6                     float yStretch, float zStretch) {...}
7
8  public LodQuadTree(String dataPath, int maxSize,
9                     int offsetX, int offsetY) {...}
10
11 public LodQuadTree(String dataPath, float zMax, float zMin) {...}
12
13 public LodQuadTree(String dataPath) {...}
```

Code-Ausschnitt 6.1: Konstruktoren der LodQuadTree-Klasse

Der Konstruktor einer LodQuadTree-Klasse ist in Code-Ausschnitt 6.1 ersichtlich. Dieser benötigt als Argumente den Pfad zur Höhenkarte (dataPath), Skalierungsfaktoren des zu erzeugenden Meshs in x-, y- und z-Richtung (x-, y- und zStretch), die maximale Höhe und die minimale Höhe der Höhenkarte (bspw. 0 und 255 bei einer normalen 8-Bit-Höhenkarte; mit zMin und zMax), sowie die Größe und Startposition des anzuzeigenden Ausschnitts der Höhenkarte (maxSize, offsetX und offsetY). Da es möglich, das nicht immer alle Optionen dieses Konstruktors benötigt werden, stehen zusätzliche Konstruktoren mit kleineren Argumentlisten als Hilfsfunktionen bereit. Die in diesen Konstruktoren nicht angegebenen Werte werden durch maschinell berechnete Standardwerte ersetzt. Diese sind für

- Die Skalierungsfaktoren immer 1
- Die Startposition befindet sich immer an $(0, 0)$
- Die maximale Größe ist das größte Vielfache von 2, welches kleiner als Länge und Breite der Höhenkarte ist
- Die maximale Höhe ist der höchste vorkommende Höhenwert im Ausschnitt
- Die minimale Höhe ist der kleinste vorkommende Höhenwert im Ausschnitt

Innerhalb des Konstruktors wird die Höhenkarte geladen. Dies geschieht, wie im Konzept beschrieben, über die Nutzung von GDAL. In Code-Ausschnitt 6.2 ist zu sehen, dass dafür nicht viel Code notwendig ist. In einem ersten Schritt müssen mit *AllRegister* die GDAL-Module geladen werden. Dann wird eine abstrakte Datenstruktur zur Verwaltung der Höhenkarte erzeugt, *Dataset*, in welche das Rasterbild geladen wird. Der Zugriff erfolgt dabei lesend. Aus dem Rasterbild wird daraufhin ein Band extrahiert.

```
1 public LodQuadTree(...) {
2     gdal.AllRegister();
3     Dataset dataset = gdal.Open(dataPath, gdalconstConstants.GA_ReadOnly);
4     Band band = dataset.GetRasterBand(1);
5
6     double[] rasterImg = new double[terrainSize*terrainSize];
7     band.ReadRaster(offsetX, offsetY, terrainSize, terrainSize, rasterImg);
8 }
```

Code-Ausschnitt 6.2: Laden der Höhenkarte mit GDAL

Ein Band entspricht einer Farbe innerhalb eines Bildes. Ein Graustufen-Bild hat somit genau ein Band, ein RGBA-Bild vier. Aus diesem Band werden in einem letzten Schritt die Pixel-Informationen ausgelesen. Das Bild wird dabei im Sinne eines zweidimensionalen

Arrays behandelt. Die beiden offset-Werte geben somit an, in welcher Zelle das Auslesen des Ausschnitts gestartet werden soll, der Wert terrainSize (der maxSize entspricht), gibt an, wie weit in x- und y-Richtung des 2D-Arrays Daten ausgelesen werden sollen. Die ausgelesenen Daten werden in einem eindimensionalen Array rasterImg abgelegt, da Java keine zweidimensionalen Arrays anbietet, sondern nur Arrays von Arrays, die langsamer gelesen werden können [Par12]. Mit zwei gegebenen Koordinaten x und y kann ein eindimensionaler Array wie ein zweidimensionaler Array behandelt werden, wenn die Breite des zweidimensionalen Arrays bekannt ist (hier: terrainSize). Die Positionsumrechnung zwischen beiden Arraytypen zeigt Gleichung 6.5:

$$
\begin{aligned}
w &= x + y * Breite \\
x &= w \quad \text{mod } Breite \\
y &= w \quad \text{div } Breite
\end{aligned}
\tag{6.5}
$$

mit x, y als Indizes des 2D-Arrays, w dem Index des 1D-Arrays, *Breite* der Breite des 2D-Arrays, sowie div dem *ganzzahligen* Divisionsoperator.

Im weiteren Verlauf des Konstruktors werden diese Werte mittels der zMin und zMax Werte in den Bereich von 0 bis 1 normalisiert. Aus den normalisierten Werten werden nun Vertices erstellt, die in einer von XL bereitgestellten Klasse, der *FloatList*, abgespeichert werden. Die Umwandlung ist dabei in Code-Ausschnitt 6.3 zu sehen. Jedes Vertex benötigt dabei genau drei Gleitkommazahlen um beschrieben zu werden, je eines für x-, y- und z-Komponente des Vertexs. Als x- und y-Wert werden dabei die 2D-Array-Indizes des momentanen Pixels verwendet, multipliziert mit den eingestellten Skalierungsfaktoren. Für die Höhe wird der normalisierte Farbwert des Pixels verwendet, ebenfalls um den gewählten Faktor skaliert.

```
1 float[] rawList = new float[rasterImg.length*3];
2 for (int i = 0; i < rasterImg.length*3; i += 3) {
3     rawList[i] = (i/3 % terrainSize) * xStretch;
4     rawList[i+1] = (i/3 / terrainSize) * yStretch;
5     rawList[i+2] = rasterImg[i/3] * zStretch;
6 }
7 FloatList vertexData = new FloatList(rawList);
```

Code-Ausschnitt 6.3: Berechnung der Vertices

Der Konstruktor endet damit, eine LodQuadNode zu erstellen und als Wurzel des Baums abzuspeichern.

Normalenberechnung

Die Normalenvektoren werden in der Funktion *refreshNormals* berechnet. Dieser wird eine zuvor erstellte Index-Liste übergeben. Zu Beginn der Funktion wird ein Gleitkommazahlen-Array *normalCalc* mit derselben Größe wie die Vertex-Liste erstellt, dieser dient der Speicherung der Normalenvektoren für jedes Vertex. Daraufhin werden die Normalenvektoren aller Flächen berechnet, die durch die gegebene Index-Liste beschrieben sind. Die Berechnung geschieht dabei gemäß des, im Konzept vorgestellten, Rechenvorgangs. Für jedes Vertex, was an der Fläche beteiligt ist, wird der errechnete Vektor, mit dem Index des jeweiligen Vertexs, in normalCalc abgespeichert. Bereits vorhandene Werte werden dabei nicht überschrieben, sondern mit den neuen Werten zusammengerechnet. Zusätzlich wird in einer HashMap für jedes beteiligte Vertex ein Wert abgespeichert, an wie vielen Flächen dieses Vertex bereits beteiligt ist.

Nachdem die Normalenvektoren aller Flächen berechnet wurden und in dem Array normalCalc für die verwendeten Vertices aufsummiert wurden, durchläuft eine zweite Schleife die zuvor angelegte HashMap. Sie liest dabei für jedes Vertex die Anzahl seiner Verwendungen aus, welche verwendet werden um den, mit dem Vertex verbundenen, Normalenvektor durch den ausgelesenen Wert zu dividieren. Damit ist der Berechnungsprozess der Vertex-Normalenvektoren abgeschlossen, sodass die Werte zur Darstellung eines Meshs verwendet werden können.

Die entstehenden Normalenvektoren würden die Landschaft jedoch falsch belichten. Daher werden zwei zusätzliche Schritte ausgeführt, die in der konzeptionellen Berechnung nicht vorhanden waren. Als erstes wird jeder berechnete Vertex-Normalenvektor normalisiert, das heißt er wird in einen Vektor gleicher Richtung mit Länge 1 umgewandelt. Weiterhin werden alle Werte in *calcNormals*, die exakt 0 entsprechen auf einen leicht erhöhten Wert gesetzt. Dadurch herrscht in der gesamten Szene eine Grundhelligkeit. Für einige Anwendungen könnte es zudem notwendig sein, auf von GroIMP berechnete Normalenvektoren zurückzugreifen. Über den Befehl *calcNormals* kann ein Boolean-Wert an die LodQuadTree-Klasse übergeben werden. Ist dieser *true* wird ab dem nächsten Rendering GroIMP die Berechnung der Normalenvektoren ausführen.

Aktualisierung des Baums

Die *update*-Funktion des LodQuadTrees dient dem Zweck, für die momentane Kameraperspektive ein korrektes LOD-Mesh zu erzeugen. Dazu wird GroIMPs Workbench direkt angesprochen. Diese Klasse repräsentiert die Benutzeroberfläche von GroIMP. Aus dieser Klasse kann auf die Renderersicht zugegriffen werden, in welcher die Kamera-Klasse erreichbar ist. Von dieser werden Position und Blickwinkel der Kamera entnommen.

Im Anschluss daran werden die drei rekursiven Funktionen der Wurzel *checkChildLod*, *repair* und *getIndexData* aufgerufen. Da die LodQuadNode-Klasse selbst kaum Daten speichert, werden den Funktionen die meisten der vom LodQuadTree gespeicherten Daten übergeben, üblicherweise gehört dazu ein Startindex innerhalb der Liste der Vertices und die Breite der 2D-Höhenkarte. Die Funktionen der LodQuadNode werden genauer in einem späteren Abschnitt vorgestellt. Relevant ist jedoch, dass die *getIndexData*-Funktion eine Index-Liste zurückgibt, die angibt, aus welchen Dreiecken das momentane Mesh besteht. Damit für dieses auch korrekte Normalenvektoren existieren, wird die Funktion *refreshNormals* an dieser Stelle mit der Index-Liste als Argument aufgerufen.

Erst nach Abschluss dieser Funktionen, wird eine neue Klasse vom Typ PolygonMesh erstellt, welcher die Vertex-Liste, die Normalenvektoren-Liste und die Index-Liste zugewiesen werden. Die Klasse erzeugt hieraus von alleine ein darstellbares Mesh. Durch den Befehl *setPolygons*, der von der MeshNode-Klasse geerbt wurde, wird das PolygonMesh dem LodQuadTree als Mesh hinzugefügt. Dadurch wird die Landschaft automatisch innerhalb des Renderers von GroIMP anzeigt, solange die LodQuadTree-Instanz Teil des XL-Graphen ist.

6.3.2 LodQuadNode

Die LodQuadNode-Klasse repräsentiert einen Knoten innerhalb des Quadtrees und damit ein Tile innerhalb eines DLOD-Systems. Um die Vorteile der Sprache XL nutzen zu können, erweitert die LodQuadNode die Node-Klasse, sodass sie automatisch einem GraphManager zugeordnet wird und die Funktionen der Node-Klasse verwenden kann. Die Klasse ist von einem Konflikt geprägt: Einerseits sollen ihre Funktionen zur Laufzeit auf allen Knoten des Baums hintereinander ausgeführt werden und die Ausführung des Programms soll dabei nicht verzögert werden. Andererseits soll der Speicher des ausführenden Computers nicht zu sehr belastet werden. Durch die hohe Menge an Instanziierungen der Klasse, ist dabei jedes eingesparte Byte relevant. Da die erstellte Erweiterung das Mesh nur auf Wunsch und nicht zu jeder Kamerabewegung aktualisiert, wurde sich in

diesem Ansatz bewusst für eine speicherplatzeffiziente Methode entschieden, bei der es, abhängig von der Größe der dargestellten Landschaft und der Kameraposition, durchaus zu Verzögerungen in der Darstellung kommen kann. In dieser Variante sind auch die theoretisch interessanteren Graph-Traversierungen ersichtlich. Für eine LodQuadNode müssen in diesem Fall nur zwei Werte abgespeichert werden. Erstens der Integer quadPow, der die momentane Tiefe des Knotens im Graphen angibt. Dabei entspricht quadPow genau der Breite des beschriebenen Tiles. Ist beispielsweise eine Höhenkarte mit den Maßen 513x513 geladen wurden, ist der quadPow-Wert der Wurzel 512. Die Werte seiner Kinder wären entsprechend 256. Zweitens wird ein Byte extraVertices gespeichert, welches im Abschnitt der Fehlerverwendung verwendet wird. Die LodQuadNode speichert keine Vertices die mit dem, von ihr beschriebenen, Tile verbunden sind. Diese werden zur Laufzeit über einen gegebenen Startindex und den quadPow-Wert berechnet.

Wäre eine Laufzeit-effizienten Lösung angestrebt worden, wären sämtliche Prozesse in der Node identisch geblieben. Es wäre lediglich mehr Speicher durch die Vergabe eines Namens der Node verbraucht worden, die in der Map des GraphManagers referenziert worden wäre. Müsste in diesem Fall eine Node gefunden werden, wäre es unnötig, den lange laufenden Prozess der Traversierung zu starten. Stattdessen könnte die gesuchte LodQuadNode einfach anhand ihres Namens aus der Map des GraphManagers entnommen werden.

Besitzt eine LodQuadNode Kinder, sind dies immer vier. Der Index der Kindknoten, der mit *getIndex* in Erfahrung gebracht werden kann, entspricht dabei ihrer späterer Position als Tile innerhalb des Elternknotens. Dabei ist 0 das linke, untere Tile; 1 das rechte, untere Tile; 2 das rechte, obere Tile und 3 das linke, obere Tile.

Baumerstellung

Um in der Lage zu sein, extrem große Höhenkarten zu erstellen, wurde für die Implementation entschieden, dass der QuadTree, bei jeder Änderung der Kameraperspektive neu aufgebaut werden soll. Dabei sollen nur Knoten Kinder besitzen, deren Detaillierungsgrad, bei der momentanen Kamerasicht, nicht ausreichend ist. Diese Herangehensweise spart enorm viel Speicher gegenüber dem Aufbau eines kompletten Baums nach dem Laden einer Höhenkarte, bei welchem in der Laufzeit nur über die Verwendung der jeweiligen Knoten, nicht über ihre Existenz, entschieden werden würde.

Die Funktion *checkChildLod* führt dabei die notwendigen Veränderungen des Graphen durch. Ihr müssen die folgenden Werte übergeben werden: vertexData, die aus der Höhenkarte erstellten Vertices; startIdx, der Index des Vertexs, das in der Mitte des aufgerufenen

Knotens liegt, in der Vertex-Liste; width, die Breite der Landschaft; cam, die Kamera der Szene. Zu Beginn der Funktion wird dabei die Variable quadPow überprüft. Ist diese gleich 1, wird die Funktion abgebrochen, da der höchste darstellbare Detaillierungsgrad erreicht ist. Im Anschluss daran wird der untere linke und der obere rechte Eckpunkt des beschriebenen Tiles geladen. Die beiden Punkte ergeben sich aus dem Startindex, sowie der Hälfte der vertikalen und der horizontalen Breite des Knotens. Danach werden die x-, y- und z-Komponenten der beiden Vertices, die mit diesem Index abrufbar sind, in einen Array geladen. Diese Koordinaten werden durch die Kamera in den Einheitskubus transformiert. Die resultierenden zweidimensionalen Vektoren, können, wie im Abschnitt des Konzepts beschrieben, genutzt werden, um zu testen, ob das Tile für den Anwender sichtbar ist. Der Code, der diesen Test ausführt, ist in Code-Ausschnitt 6.4 dargestellt.

```
1 double[] corners = new double[6];
2 int dlIdx = startIdx - quadPow/2 - (quadPow/2)*width;
3 int urIdx = startIdx + quadPow/2 + (quadPow/2)*width;
4 cornes[0] = vertexData.get(dlIdx*3);
5 /* Äquivalent für corners 1-5 */
6
7 Vector2f dl = new Vector2f(), ur = new Vector2f();
8 cam.projectWorld(corners[0], corners[1], corners[2], dl);
9 cam.projectWolrd(corners[3], corners[4], corners[5], ur);
```

Code-Ausschnitt 6.4: Sichtbarkeitstest eines Tiles

Nur im Falle eines sichtbaren Tiles wird ein Vertex gewählt, welches möglichst nah an der Kamera liegt. Dies wird über Code-Ausschnitt 6.5 erreicht. Den Werten x und y werden dabei die Koordinaten der (nicht invertierten) Kamera zugewiesen, wenn diese im positiven Bereich liegen. Ansonsten werden sie auf 1 gesetzt. Aus vier verfügbaren Vertices, wird daraus das am nächsten zur Kamera liegende berechnet. Die zur Wahl stehenden Vertices entsprechen dabei den Mittelpunkten, der Tiles, die von den Kindsknoten des aktuellen Knotens verwendet werden würden. Der Test des Codes überprüft, getrennt für die x- und y-Koordinate, ob die Position der Kamera höher oder niedriger als der Mittelpunkt des Tiles ist. Ist sie höher, wird der Mittelpunkt eines linken (für x) oder oberen (für y) Kindknotens vergeben, sonst der eines rechten oder unteren. Durch die Kombination beider Indizes ergibt sich genau eines der vier möglichen Vertices, dieses wird im Code in den Vektor posVec abgespeichert.

```
1  int x = camPos.x < 0f ? (int) Math.round(-1f*camPos.x) : 1;
2  int y = camPos.y < 0f ? (int) Math.round(-1f*camPos.y) : 1;
3  int quartPow = quadPow/4;
4  int xPoint, yPoint, estX, estY;
5
6  if (x < posVec.x) { xPoint = -quartPow; estX = -quadPow/2;}
7  else { xPoint = quartPow; estX = quadPow/2;}
8  if (y < posVec.y) { yPoint = -quartPow*width; estY = -(quadPow/2)*width;}
9  else { yPoint = quartPow*width; estY = (quadPow/2)*width;}
10
11 posVec = new Vector3d(
12     vertexData.get((startIdx+xPoint+yPoint)*3),
13     vertexData.get((startIdx+xPoint+yPoint)*3+1),
14     vertexData.get((startIdx+xPoint+yPoint)*3+2)
15 );
```

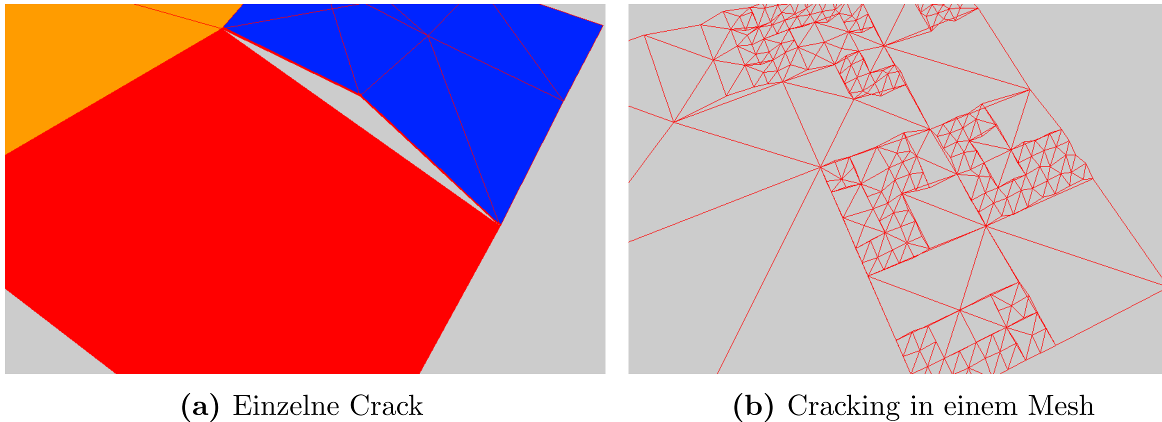
Code-Ausschnitt 6.5: Berechnung des nächsten Punkts an der Kamera

Mit diesem Vektor kann, im weiteren Ablauf der Funktion, der Höhenfehler des Tiles an der Position des Vertices berechnet werden. Als Mittelpunkt eines potentiellen Kindknotens des Tiles, ist dieser kein Bestandteil, des Tiles selber, liegt jedoch genau in der Mitte einer Kante, die zwischen einem Eckpunkt und dem Mittelpunkt des Tiles verläuft. Damit zeigt sich ein weiterer Vorteil in der Verwendung von Triangle-Fans. Die geschätzte Höhe an der Position des Vertexs, ist damit der Durchschnitt der Höhen des entsprechenden Eckpunkts und des Mittelspunkts des momentanen Tiles.

Ist der Höhenfehler berechnet, kann, exakt der mathematischen Formel im Konzept entsprechend, der Bildfehler des Tiles an der Position des gewählten Vertexs bestimmt werden. Ist dieser größer als der maximal erlaubte Fehler, werden, falls nicht bereits vorhanden, neue Kindknoten für den aktuellen Knoten erstellt und die Funktion wird für jeden der Knoten ausgeführt. Ist der Bildfehler kleiner als der maximal erlaubte Bildfehler, werden eventuell bestehende Kindknoten aus dem Graphen entfernt. Für all diese Aktionen werden dabei die Funktionen der Node-Klasse verwendet.

Fehlerbehebung

Ein Baum, der mit der Methodik des obigen Abschnitts verfeinert wurde, entspräche formal dem notwendigen Baum zur Erzeugung eines korrekten LOD-Meshs. In der prak-



(a) Einzelne Crack **(b)** Cracking in einem Mesh

Abbildung 6.5: Cracking innerhalb eines Meshs. In (a) ist die Crack zwischen dem blauen und roten Bereich, durch den grauen Zwischenbereich, deutlich zu erkennen.

tischen Anwendung ist dies allerdings nicht der Fall, da es zu grafischen Fehler kommen kann, wenn zwei Knoten unterschiedlich hohen Detaillierungsgrads direkt nebeneinander liegen. Diese Fehler sind unter dem Begriff *Cracks* bekannt und unter Abbildung 6.5 exemplarisch dargestellt. Sie entstehen dadurch, dass die Vertices der höher detaillierten Tiles nicht zwangsläufig auf der Kante des weniger detaillierten Tiles verlaufen. Die maximale Größe der Crack entspricht somit exakt dem Höhenfehler des zusätzlichen Vertexs an dieser Kante.

Um diese auffälligen Fehler beheben zu können, muss in einem Graphen eine Grundvoraussetzung geschaffen werden: für keine zwei *geometrische* Nachbar-Knoten eines Baums, darf gelten, dass der Unterschied ihres Detaillierungsgrades größer als 1 ist. Der Baum in Abbildung 6.6a wird durch Anwendung dieser Regel in den Baum in Abbildung 6.6b umgewandelt. Ist dieser Zustand geschaffen, muss ein spezielles Prozedere durchgeführt werden, um die Cracks zu entfernen. Jeder Nachbar eines Knotens, der einen höheren Detaillierungsgrad als der Knoten hat, muss an der Kante zum Knoten das höher detaillierte Vertex des Knotens mitbenutzen. In Abbildung 6.7 ist dieser Vorgang der Reparatur abgebildet. Durch das Einfügen des neuen Vertices entsteht eine zusätzliche Kante. Dadurch wird ein einzelnes Dreieck in zwei neue, kleinere ersetzt. Es kommt zu einer Erhöhung des Detaillierungsgrads: das Tile stellt einen Übergang zwischen zwei verschieden detaillierten Stufen her. Zur Speicherung der zusätzlich aktiven Vertices in einem Knoten wird das Byte extraVertices verwendet. Dabei werden die einzelnen Bits des Bytes als Boolean-Werte für die Nutzung eines Zusatz-Vertex interpretiert. Das erste Bit steht für das Vertex in Richtung des unteren Nachbarn, das zweite für das Vertex in Richtung des rechten Nachbarn, das dritte für das Vertex zum oberen Nachbarn und das vierte Bit steht für das Vertex zum linken Nachbarn. Die restlichen vier Bits des Bytes bleiben unbenutzt.

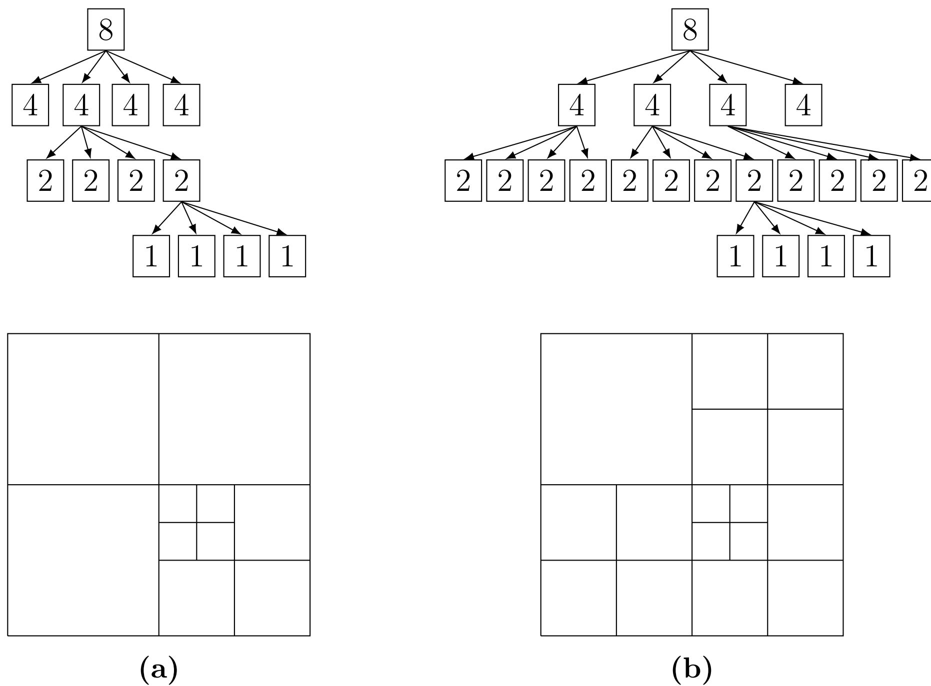


Abbildung 6.6: QuadTree in Graphenform und als Drahtgittermodell. In (a) ist ein einfacher QuadTree dargestellt. In (b) ist der selbe QuadTree zu sehen mit einem maximal erlaubten LOD-Unterschied von 1.

Die Einhaltung obigen Regel und die Reparatur des QuadTrees wird durch die Funktion *repair* gewährleistet, die durch den Baum traversiert und für jeden Knoten überprüft, ob dieser nur Blätter als Kinder besitzt. Ist dies der Fall, ist ein minimaler Detaillierungsgrad erreicht und eine Reparatur kann über die Funktion *enableExtraVerts* gestartet werden. Diese Funktion ruft die Suchfunktion *solveUp* für jeden gesuchten Nachbarn in ihrem Vaterknoten auf. Als Argumente erhält diese Funktion den Nachbarn der gesucht ist. Dabei steht 0 für den unteren, 1 für den rechten, 2 für den oberen und 3 für den linken Nachbarn. Außerdem wird eine Liste übergeben, in der der momentane Index (der durch *getIndex* zurückgegebene) des aktuellen Knotens steht. Die Suchfunktion entscheidet, durch Nutzung des obersten Listenelements und des gesuchten Nachbarn, ob die Suche abgeschlossen

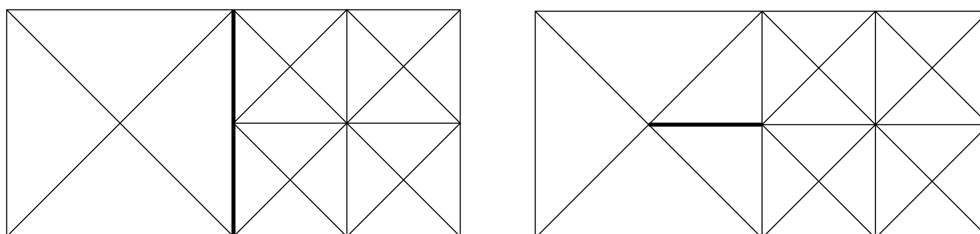


Abbildung 6.7: Crack-Reparatur. Im linken Mesh entsteht ein Crack der fett markierten Linie. Im rechten Bild wurde durch Einfügen eines Zusatzvertex in der Mitte der Kante wird der Fehler behoben. Stattdessen entsteht eine neue Kante.

werden kann oder nicht. Wird beispielsweise der rechte Nachbar gesucht und das oberste Listenelement ist 3, kann die Suche gelöst werden, da der gesuchte Knoten mit Index 2, ebenfalls ein Kind des aktuellen Elternknotens ist. Falls es nicht möglich ist, die Suche auf der aktuellen Ebene zu lösen, wird der Index des momentanen Knotens der Liste hinzugefügt und die Suchfunktion wird für den Elternknoten aufgerufen. Dieser Prozess wird solange fortgesetzt, bis die Suche gelöst werden kann, oder der aktuelle Knoten keinen Elternknoten mehr hat. Im letzten Fall wird *null* zurückgegeben. Für jeden besuchten Elternknoten wird zusätzlich die Funktion *enableExtraVerts* aufgerufen.

Anderenfalls wird die Lösungsfunktion *solveDown* ausgeführt, übergeben wird die Liste der durchlaufenden Indizes und der gesuchte Nachbar. Die Funktion entfernt aus der Liste den obersten Eintrag und wählt einen Kindknoten der momentanen Wurzel aus, der zum gesuchten Nachbarn führt. Wird beispielsweise der rechte Nachbar gesucht und das oberste Element war 2 (ein rechts unten liegendes Tile), wird der Kindknoten mit dem Index 3 ausgewählt. Ist die Liste nun leer, wird der ausgewählte Knoten zurückgegeben, ansonsten wird die Lösungsfunktion mit der verkürzten Liste am ausgewählten Knoten ausgeführt.

Hat ein Knoten in der Lösungsfunktion keine Kinder, werden diese erstellt. Dadurch wird automatisch die Regel eingehalten, dass im gesamten Graphen zwischen zwei Nachbarn der Detaillierungsgrad maximal um 1 voneinander abweichen kann. Falls neue Kinder für den Knoten erstellt wurden, wird für diesen Knoten ebenfalls die *enableExtraVerts*-Funktion aufgerufen.

Sobald die vier Nachbarn gefunden worden sind, wird die *enableExtraVerts*-Funktion fortgesetzt. Dort wird für jeden Nachbarn, der nicht null ist, der extraVertices-Wert ergänzt. Der Wert des unteren Nachbarn wird um 4 erhöht, der des linken um 8, der des oberen um 1 und der des rechten um 2. Damit sind alle Voraussetzungen geschaffen, dass das Mesh bei der nächsten Index-Listen-Erstellung keine Cracks mehr enthält.

Indexerstellung

Die Erstellung der Index-Liste zur Erzeugung eines Meshs, wird über die Funktion *getIndexData* ausgeführt. Dieser Funktion werden die beiden Argumente *width* und *startIdx* mitgeteilt. Der Wert width entspricht dem Wert terrainSize des LodQuadTrees. startIdx beschreibt hingegen den Index des linken, unteren Vertex des, über den Knoten, verwalteten Tiles. Der Ablauf der Funktion selbst ist simpel. Zu Beginn wird eine *IntList* erstellt, zur Speicherung der Indizes. Falls der Knoten selber Kinder hat, ruft er für alle seine

Kinder diese Funktion ebenfalls auf und speichert die von ihnen zurückgegeben IntLists in seiner eigenen ab.

Falls der Knoten jedoch ein Blatt ist, wird in der IntList entweder ein Triangle-Fan abgespeichert, wenn quadPow > 1 ist, oder ein Triangle-Strip, wenn quadPow = 1 ist. Der Vorgang der Erstellung läuft dabei äquivalent zu dem des Konzepts ab und braucht daher nicht genauer betrachtet werden. Für den Triangle-Fan existiert allerdings eine im Konzept nicht beachtete Besonderheit, die durch die Reparatur entsteht. In der Fehlerbehebung wurde das extraVertices-Byte nur mit Werten besetzt. Erst in dieser Funktion, werden die Werte auch um eine geometrische Repräsentation ergänzt. Code-Ausschnitt 6.6 zeigt, wie ein gesetztes, zusätzliches Vertex angewendet wird. Standardmäßig besteht der zu erstellende Triangle-Fan aus vier Dreiecken. Mithilfe einer Schleife, werden die benötigten Vertices durchlaufen und die vier Dreiecken des Fans erstellt.

```
1  byte eVerts = 1;
2  for (int i = 0; i < 4; i++) {
3      int next = (i+1)%4;
4      if ((extraVertices & eVerts) == eVerts) {
5          indices.add(    v[i]    );
6          indices.add(    v[i+5]);
7          indices.add(    v[4]    );
8          indices.add(    v[i+5]);
9          indices.add(    v[next] );
10         indices.add(    v[4]    );
11     } else {
12         indices.add(    v[i]    );
13         indices.add(    v[next] );
14         indices.add(    v[4]    );
15     }
16     eVerts *= 2;
17 }
```

Code-Ausschnitt 6.6: Erstellung der Index-Liste

Als erstes wird das untere, danach das rechte, dann das obere und zuletzt das linke Dreieck erstellt. Bei jedem Dreieck, welches erstellt werden soll, wird zuerst der extraVertices-Wert durch ein logisches Und mit dem eVerts-Wert verbunden. Dieser entspricht in den Schleifendurchgängen 1, 2, 4 und 8. Ist der logische Verbund gleich eVerts, ist das Zusatzvertex für das momentan zu erstellende Dreiecke aktiviert. In diesem Fall werden statt einem

Dreieck, zwei Dreiecken erstellt, die sich das mittlere Vertex des Tiles (v[4]) und das Zusatzvertex (v[i+5]) teilen, unterschiedlich sind somit nur die Eckpunkte von denen sie aufgespannt werden (einmal v[i] und v[next]). Ist das Zusatzvertex nicht aktiv, wird ein einzelnes Dreieck in der Index-Liste abgespeichert. Die Werte im Array v sind dabei die Indizes der verwendeten Vertices des momentanen Knotens. Sie werden zu Beginn der Funktion durch Verwendung von startIdx, width und quadPow berechnet. Dabei ist v[0] bspw. stets startIdx (der untere, linke Eckpunkt) und v[2] (der obere, rechte Eckpunkt) entsprechend v[0] + quadPow + quadPow*width. Diese Berechnungen leiten sich aus den Umrechnungen zwischen 1D- und 2D-Array ab.

6.4 Evaluation

Dieses Kapitel soll sich kurz damit befassen, welche Möglichkeiten die erstellte Erweiterung bietet und inwiefern diese sich mit den formulierten Zielen der Arbeit überdecken. Weiterhin soll eine kurze Leistungsbewertung der Erweiterung im Hinblick auf Ausführungszeit und Speicherplatzverbrauch gegeben werden.

6.4.1 Möglichkeiten

Die erstellte Erweiterung erlaubt die Anzeige von Höhenkarten, die als GeoTIFF- oder DTED-Dateien vorliegen. Dabei wird, abhängig von der Kameraperspektive, ein passender Detaillierungsgrad aus einer für die Erweiterung erstellten HLOD-Struktur entnommen. Die formalen Ziele der Arbeit wurden somit erreicht. Durch die Einbindung von GDAL wurden zudem im Import die möglichen Formate für Höhenkarten stark erhöht. Die Anzeige der Höhenkarten wird auf ein Vielfaches von 2 plus 1 begrenzt, dieses Einschränkung ist jedoch keinesfalls unüblich, wenn die Verwaltung der Landschaft über eine Baumstruktur geschieht [Ait13]. Das erstellte Mesh wird nicht texturiert, denn hier müssten Annahmen über die Umgebung erfolgen, die die Erweiterung zurzeit noch nicht leisten kann. Stattdessen können normale Shader in GroIMP auf das Mesh angewendet werden, um eine korrekte Färbung der Landschaft zu erreichen. Eine zurzeit bestehende Einschränkung ist, dass der Detaillierungsgrad des Meshs nicht aktualisiert wird, wenn sich die Kamera bewegt, sondern dies manuell angefordert werden muss. Für schwächere Computer, die GroIMP oft schon mit normalen Simulationen belastet, ist diese manuelle Korrekter jedoch besser geeignet. Unabhängig davon wurde die Erweiterung mit einer Fülle an Funktionen ausgestattet, die dem Anwender Individualisierung und Komfort

gewährleisten sollen, z.B. der einstellbare maximal erlaubte Bildfehler oder der Wechsel zwischen von der Erweiterung oder von GroIMP berechneten Normalenvektoren.

6.4.2 Leistungsbewertung

Die Leistungsbewertung der Erweiterung muss unverschuldet kleiner als gedacht ausfallen. Der Grund hierfür ist, dass die aktuelle Version von GroIMP einen *Memory Leak* besitzt. Das bedeutet, das Daten, die nicht mehr verwendet werden, aus irgendeinem Grund nicht von Javas Garbage Collector entfernt werden. Dadurch erhöht sich mit zunehmender Laufzeit der Speicherplatzverbrauch eines Programms und Abläufe werden langsamer, wodurch eine korrekte Messung der Leistung verhindert wird. Deswegen wird sich dieses Kapitel auf eine kurze Einschätzung der Leistung begrenzen.

Vom Speicherplatzverbrauch her ist die Erweiterung sehr effizient. Nur die Vertexdaten werden dauerhaft gespeichert. Es findet keine Duplizierung von Variablen oder Listen statt, die meisten Daten werden zur Laufzeit über Funktionsaufrufe übergeben. Da die LodQuadNode-Klasse nur 1 Byte und 1 Integer speichert, ist die Anzahl der vorhandenen Klassen dieses Typs fast irrelevant. Diese Speicherplatz-Effizienz wird lediglich von der Node-Klasse GroIMPs unterminiert, die einige Daten speichert, die für einen QuadTree zur Darstellung von LOD-Meshes unnötig sind. Eine Alternative wäre möglich, wenn eine abstrakte Grundklasse der Node existierte, die es erlauben würde, je nach Anwendungsfall zu entscheiden, welche Elemente wie implementiert werden sollen.

Durch die bewusste Entscheidung für eine theoretische Korrektheit und einen geringen Speicherplatzverbrauch, ist die Laufzeit der Erweiterung nicht optimal. Je nach Größe der Landschaft, Blick der Kamera auf diese und eingestelltem maximal erlaubten Bildfehler, kann die Darstellung des Meshs unterschiedlich lange dauern. Die längsten Laufzeiten werden hierbei von den Traversierungen durch den Graph, bei der Reparatur des Meshs, erreicht. Dieser Abschnitt könnte über die Nutzung der Map des GraphManagers deutlich beschleunigt werden. Im jetzigen Zustand dauert eine Anzeige durchschnittlich 2 bis 30 Sekunden. Hohe Wartezeiten entstehen vor allem bei einer Panormasicht der Landschaft, da in dieser kaum Elemente der Landschaft mittels Culling entfernt werden können.

6.5 Anwendung

Dieser Abschnitt soll Anwendern der Erweiterung erläutern, welche Voraussetzungen zur Installation der Erweiterung erfüllt sein müssen und wie der Installationsvorgang

selbst abläuft. Damit auch ein Verständnis der Benutzung der Erweiterung innerhalb von GroIMP gegeben werden kann, soll in einem zweiten Schritt eine exemplarische Benutzung der Erweiterung in Einzelschritten aus Anwendersicht erfolgen. Diese Demonstration soll auch die Einsatzbereitschaft der Erweiterung darstellen.

6.5.1 Installation

Damit eine Installation der Erweiterung erfolgreich ausgeführt werden kann, müssen folgende Vorbedingungen erfüllt sein:

- Java in Version 7 installiert
- GroIMP installiert

Für einen fehlerfreien Ablauf des Programms sollten dabei GroIMP und Java entweder beide als 32-Bit oder beide als 64-Bit Version vorliegen. Vor der Installation der Erweiterung sollten die GDAL-Dateien auf dem System gespeichert werden. Zur Erzeugung dieser gibt es viele Herangehensweisen. Die einfachste Möglichkeit ist, die bereits kompilierten GDAL-Binärdateien unter `http://download.gisinternals.com/sdk.html` herunterzuladen. Es sollte die Version 1800, bzw. die Compiler-Version MSVC 2013 gewählt werden. Die Plattform sollte der von GroIMP und Java entsprechen. Aus dem ZIP-Archiv sollten die DLL-Dateien im Pfad release-1800-PLATTFORM\bin\ und im Pfad release-1800-PLATTFORM\bin\gdal\java\ in einen Ordner des System kopiert werden, der in der Systemvariable PATH hinterlegt ist. Falls von dieser kein Ordner bekannt ist, können die Dateien auch im Java-Installationspfad in den Ordner bin abgelegt werden. Beispielsweise: C:\Program Files (x86)\Java\bin\. Vorhandene Dateien müssen nicht ersetzt werden. Danach kann die Installation der Erweiterung beginnen: Es muss lediglich der Ordner Landscape vom Datenträger kopiert werden und im GroIMP-Installationsverzeichnis in den Ordner plugins eingefügt werden (z.B.: C:\Program Files (x86)\GroIMP\plugins\). Danach ist die Erweiterung beim nächsten Start von GroIMP verfügbar. Dies bedeutet, dass die Klassen LodQuadTree und LodQuadNode, wie jede andere XL-Klasse ohne weitere Vorbereitung verwendet werden können. Aus Anwendersicht ist allerdings die Verwendung der Klasse LodQuadTree ausreichend. Das nächste Kapitel wird die Verwendung der Klassen mit installierter Erweiterung demonstrieren.

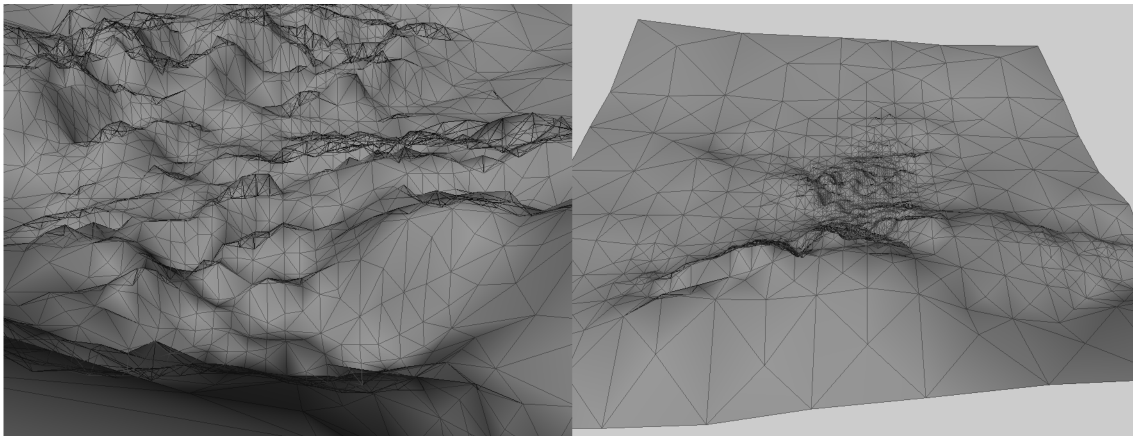


Abbildung 6.8: Detaillierungsgrad im Sichtbereich und im gesamten Mesh

6.5.2 Exemplarischer Anwendungsfall

Das Ziel dieses Abschnitts ist es mithilfe der in der Implementation erstellten Erweiterung eine Landschaft aus Anwendersicht zu erstellen. Als Höhenkarte wird dabei eine GeoTIFF-Datei von Göttingen verwendet, die von der Seite `http://earthexplorer.usgs.gov/` entnommen wurde. Die verwendete Höhenkarte befindet sich allerdings auch auf dem, der Arbeit beiliegenden, Datenträger. Das Beispiel kann auf jedem Computer nachgestellt werden, auf dem die Anweisungen des vorherigen Abschnitts korrekt ausgeführt wurden.

Zuerst muss GroIMP gestartet werden. Über File New RGG Project wird in diesem Fall ein neues Projekt erstellt. Der Name des Projekts bleibt Model, die Namensvergabe ist für das Funktionieren der Erweiterung allerdings irrelevant.

Im jEdit-Editor muss nun eine neue Variable des Typs LodQuadTree erstellt werden. In diesem Beispiel wird als Bezeichner landscape gewählt. Für diese Demonstration werden die von GroIMP automatisch erstellen Code-Elemente zum späteren Größenvergleich beibehalten werden. In einer realen Anwendung müsste allerdings nur die init-Funktion als leerer Funktionskörper beibehalten werden. In der Init-Funktion muss nun mittels der geschweiften Klammern ein neuer Java-Code-Block erstellt werden. In diesem werden die Anweisungen eingesetzt, die zur Erstellung der Landschaft nötig sind. Die genauen Befehle sind in Code-Ausschnitt 6.7 ersichtlich. Der erste Befehl erstellt einen neuen LodQuadTree, der die Landschaft verwaltet. Als Basis für das Mesh wird die Höhenkarte am Pfad C:\ASTGTM2_N51E009_dem.tif verwendet. Sie wird dabei mit dem Wert 255 normiert, dies entspricht dem maximalen Wert einer Graustufe bei 8-Bit-Farbtiefe. Das entstehende Mesh wird sehr klein gehalten: die x- und y-Streckung werden mit 1 angegeben, sodass ein 1x1-Feld im Koordinatensystem von GroIMP der Fläche eines Höhenwertes in-

nerhalb der Höhenkarte entspricht. Der z-Wert wird mit $30f$ dargestellt. Es wird nur

```
1  LodQuadTree landscape;
2
3  module A [...]
4
5  protected void init() [
6      {
7          landscape = new LodQuadTree("C:\ASTGTM2_N51E009_dem.tif", 1f, 1f, 30f,
8                                      256, 0, 257, 100, 100);
9          landscape.setMaxError(5f);
10         landscape.update();
11     }
12     Axiom ==> landscape A(1);
13 ]
```

Code-Ausschnitt 6.7: Laden einer Höhenkarte aus Anwendersicht

ein 256x256 großer Ausschnitt der Höhenkarte dargestellt. Der Größenwert ist entspricht somit 257. Diese Größe sollte für Simulationen innerhalb von GroIMP allerdings ausreichen, wird der Größenvergleich mit der automatisch erstellten Regel des Modells in Abbildung 6.9 betrachtet. Der Ausschnitt wird beginnend von der hundertsten Position, aus x- und y-Sicht, der Höhenkarte entnommen. In einer zweiten Anweisung wird der maximal erlaubte Bildfehler auf $5f$ gesetzt. Danach wird, durch Aufruf der update-Funktion, eine initiale Erstellung des Meshs angefordert. Diese drei Funktionen können auch abgekürzt in XL-Schreibweise verwendet werden, wie Code-Ausschnitt 6.8 zeigt. Dabei wird die Neuerstellung um ein weiteres Argument ergänzt (*5f*), welches dem maximal erlaubtem Bildfehler entspricht.

```
1  LodQuadTree landscape;
2
3  module A [...]
4
5  protected void init() [
6      Axiom ==> A(1) landscape:LodQuadTree("C:\ASTGTM2_N51E009_dem.tif", 1f,
7                                          1f, 30f, 256, 0, 257, 100, 100, 5f);
8  ]
```

Code-Ausschnitt 6.8: Laden einer Höhenkarte. Kurzschreibweise.

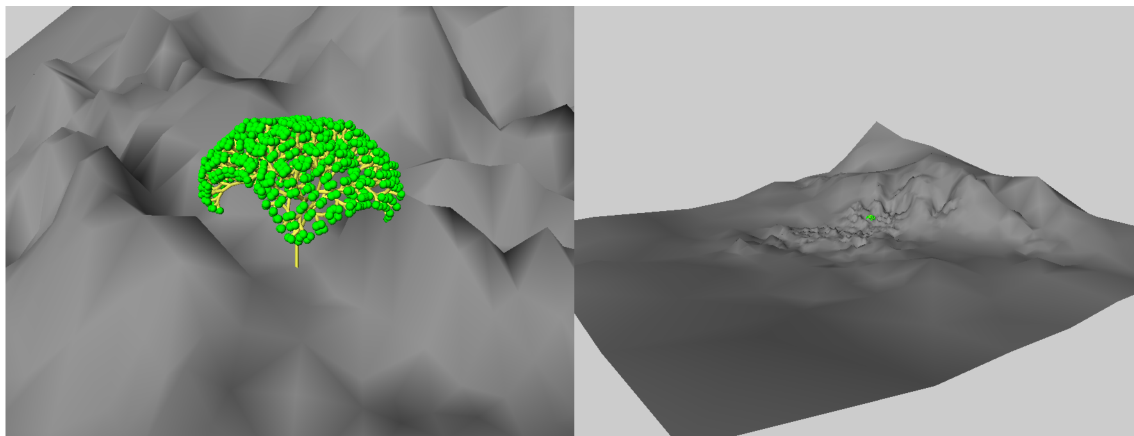


Abbildung 6.9: Größenvergleich der geladenen Landschaft mit 10 Iterationen der Standardregel

Beim nächsten Speichern des Codes sollte nun automatisch die Landschaft geladen werden. Je nach Umfang der Höhenkarte und Position der Kamera läuft dieser Prozess unterschiedlich schnell ab. Soll das Projekt um die Möglichkeit erweitert werden, die LOD der Landschaft auch während der laufenden Simulation zu aktualisieren, muss lediglich eine kleine Funktion ergänzt werden, die in Code-Ausschnitt 6.9 zu sehen ist. Dabei wird eine funktionale Regel verwendet um für alle geladenen Landschaften die *update*-Funktion aufzurufen. Dies sorgt dafür, dass der QuadTree eine neue Version des Meshs erzeugt, mit einem zur Kameraperspektive passenden LOD.

```
1 protected void updateLod() [
2     l:LodQuadTree ::> l.update();
3 ]
```

Code-Ausschnitt 6.9: LOD-Aktualisierung der Landschaft

6.6 Zusammenfassung

In diesem Kapitel wurde die Umsetzung des Arbeitsziels beschrieben. Der theoretischen Vorarbeit wurde damit auch aus angewandter Sichtweise ein Sinn verliehen. Es wurde das Ziel der Arbeit wiederholt, für die Plattform GroIMP eine LOD-basierte Landschaftsanzeige als Erweiterung zu erstellen. Daraufhin wurden innerhalb eines Konzept die Hauptprobleme ausgemacht, die zum Erreichen des Ziels gelöst werden müssen: Die Erstellung eines anzeigbaren Meshs, der Import der Rasterformate GeoTIFF und DTED und der Aufbau der HLOD-Struktur, sodass für jede Betrachtung der Szene eine passende LOD-Variante erzeugt werden kann. Die Erstellung des Meshs wird dabei über die XL-Klassen

MeshNode und PolygonMesh gelöst. Der Import der Rasterformate kann durch die Open-Source Bibliothek GDAL gewährleistet werden, die sogar den Import viele anderer Formate erlaubt. Die HLOD-Struktur wurde als QuadTree konzipiert, der abhängig von der Kameraposition, die Anzahl der Knoten in ihm anpasst. Jeder Knoten repräsentiert dabei ein Tile mit einem bestimmten Detaillierungsgrad, wobei die Wurzel die gröbste Tile-Version des Meshs ist und jedes Kind eines Knotens eine Verfeinerung des Elternknotens darstellt. Ob eine Verfeinerung benötigt wird, wird über das Kriterium des Bildfehlers gemessen. Weiterhin ist die Struktur in der Lage, die Knoten zu durchlaufen und aus den von ihnen repräsentierten Detaillierungsgraden eine Index-Liste zu erstellen, die zur Erstellung der Flächen des Meshs notwendig ist. Der Abschnitt der Implementation zeigte im Anschluss daran, wie die beiden Klassen LodQuadTree und LodQuadNode in Java mit der Unterstützung durch die Sprache XL und der Plattform GroIMP implementiert wurden. Für die einzelnen Funktionen der Klasse wurden, im Sinne einer Dokumentation, Erläuterungen geliefert, die auch auf die einzelnen Funktionsabschnitte eingingen, welche die, im Abschnitt der Konzeption erläuterten, Probleme lösen. Zusätzlich wurde das Problem der Cracks erklärt, sowie eine Möglichkeit präsentiert, diese zu entfernen. Im Bereich der Evaluation wurde gezeigt, dass das Arbeitsziel durch die Implementation erfüllt wurde, es bleibt jedoch weiterhin Raum für Verbesserungen. So ist zwar zurzeit der Speicherverbrauch sehr niedrig, die Aktualisierung des Meshs kann allerdings zu einer kurzen Wartezeit führen, die bei der Umwandlung zu einer dauerhaften Aktualisierung des Meshs entfernt werden sollte. Im letzten Abschnitt wurde gezeigt, wie eine Anwendung der Erweiterung möglich ist: es wurde die Installation auf einem vorinstallierten GroIMP demonstriert und ein Beispiel der Anwendung anhand von Geodaten im Raum Göttingen gegeben.

Kapitel 7

Zusammenfassung und Ausblick

Dieses Kapitel soll den Abschluss der vorliegenden Arbeit bilden. Die Arbeit wurde mit dem Ziel gestartet, die Plattform GroIMP um die Möglichkeit zu erweitern, reale Landschaften, die in Rasterbildern kodiert sind, effizient darstellen zu können. Daraus ergab sich die Notwendigkeit einer Struktur, die in der Lage ist ein Objekt zu verwalten und Vereinfachungen dieses Objekts zu erzeugen, die auf heutigen Computern darstellbar sind. Mit den Techniken des Level-of-Detail sind solche Strukturen gegeben. Die Darstellung von Landschaften setzt an diese besondere Anforderungen, da nicht nur die Anzeige eines bestimmten Detaillierungsgrades gefordert wird, sondern die parallele Anzeige vieler verschiedener, abhängig davon, wie zentral einzelne Teile der Landschaft für das, durch die Kamera der Szene beobachtbare, Bild sind. Klassische LOD-Systeme, wie DLOD oder CLOD, können dies nicht leisten. Erst die VLOD-Systeme beachten die Position des Anwenders, bei der Erstellung von Verfeinerungen. In diesem Systemen wird allerdings immer noch von der Nutzung eines Polygon Meshs ausgegangen, dass im höchsten Detaillierungsgrad vorliegt und vereinfacht werden muss. Rasterbild-Landschaften sind hingegen nur eine sortierte Menge von Höhenwerten eines bestimmten Gebiets. Ihre Verwaltung muss dadurch auf anderem Wege erfolgen. Hier bieten sich HLOD-Systeme an, die ein Objekt vermittels einer hierarchischen Struktur beschreiben. Diese Struktur kann als Graph interpretiert werden. Dabei stellt die Wurzel eine gröbste Verallgemeinerung des gesamten Objekts dar, jedes Blatt entspricht einem Ausschnitt aus der originalen Geometrie des Objekts. Soll ein grobes Objekt einen höheren Detaillierungsgrad benutzen, müssen lediglich die Informationen der Kindsknoten, statt der Wurzel, verwendet werden. Da statt jedes Kindes wiederum dessen Kinder verwendet werden können, können einzelne Positionen des Objekts mit einem höheren Detaillierungsgrad dargestellt werden, als andere.

Zudem wurde in der Arbeit die grundlegende Funktionsweise eines Renderers erläutert. Obwohl GroIMP diesen bereits selbst implementiert hat, ist es wichtig ein Verständnis

über den grundlegenden Ablauf einer Rendering-Pipeline zu haben. Anderenfalls würden wichtige Informationen, wie Optimierungen am Code durchzuführen sind, fehlen. So ist zum Beispiel die Darstellung aller Objekte als Dreiecke wichtig um ein schnelles Rendering zu ermöglichen. Auch das mittels Culling die Anzahl der anzuzeigenden Objekte möglichst vor dem Rendering zu reduzieren ist, wird verständlich.

Abschließend wurde in der Arbeit ein theoretisches Konzept zur Implementation der Erweiterung gegeben. In diesem wurde bestimmt, dass die Höhenkarten mit der Bibliothek GDAL geladen werden sollten, um so die Verwendung beliebiger Rasterformate zu erlauben. Die geladenen Höhenkarten sollten durch einen QuadTree verwaltet werden, da dieser sehr verbreitet ist und die leichte Verwaltung quadratischer Strukturen erlaubt. Es wurden Methoden geplant, wie für einen, durch die Knoten des Baums dargestellten Detaillierungsgrad, Index-Listen erstellt werden können, die die Dreiecke definieren, die das LOD-Mesh erzeugen. Ebenfalls wurde im Konzept definiert, dass das Kriteriums des Bildfehlers einer Position innerhalb der Szene dazu verwendet werden soll, darüber zu entscheiden ob bei der Aktualisierung des Baums ein Knoten um Kindsknoten ergänzt wird, oder ob das durch ihm dargestellte Detail für die momentane Betrachtung der Szene ausreicht. Zuletzt wurde die Implementation selbst erklärt und demonstriert. Es zeigte sich, dass für die meisten Abläufe, die konzeptionelle Beschreibung direkt implementiert werden konnte. Nur die Reparatur von sogenannten Cracks, Lücken die zwischen zwei verschiedenen Detaillierungsgraden auftreten, musste in der Implementation neu eingeführt werden. Dafür ist das Traversieren durch den Graphen notwendig, sodass, mithilfe zusätzlicher Vertices, Übergänge zwischen zwei verschiedenen Detaillierungsgraden erstellt werden können. Eine abschließende Evaluation zeigte, dass die programmierte Erweiterung die in der Einführung gestellten Ziele erreicht hat. Da der Fokus in der Implementation jedoch sehr auf ein Speicherplatz-effizientes Modell gelegt wurde, kommt es bei der Ausführung der Erweiterung zu Wartezeiten, bis die geladene Landschaft angezeigt wird.

Damit ist ein Punkt gefunden, der in zukünftigen Versionen der Erweiterung optimiert werden kann. Wird der GraphManager zum Auffinden von benachbarten Knoten gefunden, kann das Traversieren über den Graphen minimiert werden, sodass selbst die dauerhafte Anpassung an die Kamera, bei jeder Veränderungen dieser, möglich ist. Aber auch andere Ziele könnten in der weiteren Entwicklung von Interesse sein. So könnte die LOD-Darstellung von Landschaften mit der LOD-Darstellung anderer Bestandteile von XL und GroIMP verbunden werden, sodass beliebig große und beliebig detaillierte Systeme in XL simuliert werden könnten. Aber auch auf der technischen Seite wären Verbesserungen möglich. So könnte beispielsweise Geomorphing eingeführt werden, um Popping zu

verhindern oder Nebel am Ende des Sichtbereichs zur Erhöhung des Realismus der Darstellung eingefügt werden. Auch die Verwendung von TINs statt uniformierten Vertex-Rastern könnte angestrebt werden. So könnten Gebiete, die nur wenig Varianz in der Höhe besitzen (etwa Gewässer), dauerhaft als eine Menge weniger Vertices zusammengefasst werden. Deutlich wird anhand dieser Liste, dass die Möglichkeiten der Optimierung und Erweiterung groß sind. Dies liegt einerseits am Alter des Forschungsfeldes, vor allem jedoch an der noch immer anhaltenden Relevanz des Themas. Die Ausbaumöglichkeiten und die Relevanz des Themas zeigen die Notwendigkeit einer Erweiterung zur Darstellung von Landschaften. Die Verwendungsdauer der implementierten Erweiterung kann daher als lange eingeschätzt werden.

Abkürzungsverzeichnis

LOD	Level-of-Detail, dt.: Detaillierungsgrad
DLOD	Diskretes Level-of-Detail
CLOD	Kontinuierliches Level-of-Detail
VLOD	Sichtabhängiges Level-of-Detail
HLOD	Hierarchisches Level-of-Detail
FPS	Frames Per Second, dt.: Bilder pro Sekunde
XL	eXtended L-system language
GroIMP	Growth-grammar related Interactive Modelling Platform
RGG	Relational Growth-Grammer, dt.: Relationale Wachstumsgrammatik
L-System	Lindenmayer-System
API	Application Programming Interface, dt.: Programmierschnittstelle
GDAL	Geospatial Data Abstraction Library
GPU	Graphical Processing Unit
SWIG	Simplified Wrapper and Interface Generator
MSVC	Microsoft Visual C++
MIT	Massachusetts Institute of Technology

Formelverzeichnis

4.1 Vertex Manipulation mittels einer Hohenkarte 19
5.1 Anwendung eines L-Systems 34
6.1 Normalenvektor-Berechnung 47
6.2 Vertex-Normalenvektor-Berechnung 47
6.3 Berechnung des Bildfehlers 49
6.4 Berechnung des Vertex-Fehlers 49
6.5 Umrechnung der Indizes von 1D- und 2D-Arrays 54

Abbildungsverzeichnis

2.1 Ein simples Mesh . . . 9

3.1 Hauptabschnitte einer Render-Pipeline . . . 14
3.2 Geometrie Abschnitt einer Render-Pipeline . . . 15

4.1 Höhenkarten-Repräsentationen . . . 18
4.2 Limitierungen einer Höhenkarte . . . 18
4.3 Uniformiertes Mesh-Gitter . . . 19
4.4 Detaillierungsgrade eines Modells . . . 21
4.5 LOD-Modelle und Entfernung . . . 21
4.6 Modell des Stanford-Drachen . . . 22
4.7 Edge Collapse und Vertexsplit . . . 24
4.8 Ablauf des Geomorphings . . . 25
4.9 Modellvereinfachungen im VLOD-System I . . . 27
4.10 Modellvereinfachungen im VLOD-System II . . . 28
4.11 Binärbaum als HLOD Struktur . . . 30
4.12 Schematischer Aufbau eines Quadtrees . . . 31

5.1 Fünffache Regelanwendung der Regel aus Code-Ausschnitt 5.2 . . . 37
5.2 Sicht-Transformationsmatrix einer Kamera . . . 40
5.3 Oberfläche der Plattform GroIMP . . . 41

6.1 Ablauf des Import-Schrittes . . . 45
6.2 Aufbau und Aktualisierung des Quadtrees . . . 48
6.3 Index-Erstellung für das LOD-Mesh . . . 50
6.4 Triangle-Strip und Triangle-Fan . . . 51
6.5 Cracking in einem Mesh . . . 60
6.6 QuadTree in Graphenform . . . 61
6.7 Reparatur von Cracks . . . 61
6.8 Detaillierungsgrad des Beispielmeshs . . . 67
6.9 Größenvergleich dargestellter Landschaften . . . 69

Programmcodeverzeichnis

5.1 Grundgerüst des XL-Codes . . . 36
5.2 Einfache Regel in der Sprache XL . . . 37
5.3 Komplexe Regel in der Sprache XL . . . 38
5.4 Code-Regel in der Sprache XL . . . 38
6.1 Konstruktoren der LodQuadTree-Klasse . . . 52
6.2 Laden der Höhenkarte mit GDAL . . . 53
6.3 Berechnung der Vertices . . . 54
6.4 Sichtbarkeitstest eines Tiles . . . 58
6.5 Berechnung des nächsten Punkts an der Kamera . . . 59
6.6 Erstellung der Index-Liste . . . 63
6.7 Laden einer Höhenkarte aus Anwendersicht . . . 68
6.8 Laden einer Höhenkarte. Kurzschreibweise. . . . 68
6.9 LOD-Aktualisierung der Landschaft . . . 69

Anhang A

Inhalte der DVD

Auf der beiliegenden DVD befindet sich einerseits die Bachelorarbeit als PDF, sowie die verwendeten Dateien der Erweiterung. Ebenfalls ist eine Dokumentation des Source-Codes vorhanden. Die DVD ist in folgende Verzeichnisse untergliedert:

- **Bachelorarbeit**
  Dieser Ordner beinhaltet die Bachelorarbeit als PDF-Dokument in zwei Versionen: eine mit markierten Verlinkungen und eine ohne diese.
- **Beispieldaten**
  Das Verzeichnis enthält die Höhenkarte, die in Abschnitt 6.5.2 verwendet wurde.
- **Dokumentation**
  Hier befindet sich die Java-Doc-Dokumentation des Quellcodes. Im Gegensatz zu der Dokumentation der Ausarbeitung, ist diese technisch gehalten.
- **Erweiterung**
  In diesem Ordner befinden sich die Dateien der Erweiterung als kompiliertes GroIMP-Plugin. Zusätzlich ist der Source-Code hier gespeichert.

Literaturverzeichnis

[Ait13] Aitchison, Alastair: *Importing DEM Terrain Heightmaps for Unity using GDAL*. 2013. – `https://alastaira.wordpress.com/2013/11/12/importing-dem-terrain-heightmaps-for-unity-using-gdal/` – zuletzt aufgerufen am 15.4.2015

[AMHH10] Akenine-Möller, Tomas ; Haines, Eric ; Hoffman, Naty: *Real-time rendering*. 3. ed. Natick, Mass. : Peters, 2010

[BB03] Bender, Michael ; Brill, Manfred: *Computergrafik: Ein anwendungsorientiertes Lehrbuch*. München, Wien : Hanser, 2003

[Cla76] Clark, James H.: Hierarchical Geometric Models for Visible Surface Algorithms. In: *Communications of the ACM* 19 (1976), Nr. 10, S. 547–554

[CR11] Cozzi, Patrick ; Ring, Kevin: *3D engine design for virtual globes*. Boca Raton, FL : CRC Press, 2011

[ESK96] Encarnacao, Jose ; Strasser, Wolfgang ; Klein, Reinhard: *Graphische Datenverarbeitung*. Oldenbourg Verlag, 1996

[Feu10] Feuerstein, Florian: *Entwicklung eines datenbankbasierten "Level of Detail" Systems zur Beschleunigung von Echtzeitszenen mit zahlreichen Geometriemodellen*. Gießen-Freiberg : Fachhochschule Gießen-Freiberg, 2010

[Hen13] Henke, Michael: *GroIMP v1.4.2 Introduction and Overview*. 2013. – `www.uni-forst.gwdg.de/~wkurth/ssc13/GroIMPIntroduction.pdf` – zuletzt aufgerufen am 15.4.2015

[Hop96] Hoppe, Hugues: Progressive Meshes. In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA : ACM, 1996 (SIGGRAPH '96), S. 99–108

[Hop97a] Hoppe, Hugues: View-dependent Refinement of Progressive Meshes. In: *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA : ACM Press/Addison-Wesley Publishing Co, 1997 (SIGGRAPH '97), S. 189–198

[Hop97b] Hoppe, Hugues: *View-Dependent Refinement of Progressive Meshes*. 1997. – Präsentation zu View-Dependent Meshes

[Hop98] Hoppe, Hugues: Smooth View-dependent Level-of-detail Control and Its Application to Terrain Rendering. In: *Proceedings of the Conference on Visualization '98*. Los Alamitos, CA, USA : IEEE Computer Society Press, 1998 (VIS '98), S. 35–42

[Joh04] Johanson, Claes: *Real-time water rendering*. Lund : Lund University, 2004

[JS02] Jirka, Tomas ; Skala, Vaclav: *Gradient Vector Estimation and Vertex Normal Computation*. Bd. DCSE/TR-2002-08. Pilsen : University of West Bohemia, 2002

[KBHK08] Kniemeyer, Ole ; Barczik, Günter ; Hemmerling, Reinhard ; Kurth, Winfried: Relational Growth Grammars - A Parallel Graph Transformation Approach with Applications in Biology and Architecture. In: Schürr, A. (Hrsg.) ; Nagl M. (Hrsg.) ; Zündorf A. (Hrsg.): *Lecture Notes in Computer Science (LNCS)* Bd. 5088. Berlin-Heidelberg : Springer-Verlag, 2008, S. 152–167

[Kni08] Kniemeyer, Ole: *Design and implementation of a graph grammar based language for functional-structural plant modelling*. Cottbus : Technical University of Cottbus, 2008

[KS13] Kushner, Andrey ; Sedovich, Vyacheslav: *Procedural Content Generation for Real-Time 3D Applications Part 1: Oil Rush*. 2013. – `http://unigine.com/articles/130605-procedural-content-generation/` – zuletzt aufgerufen am 15.4.2015

[Lan12] Landesverwaltungs Nordrhein-Westfalen: *Leitfaden für langzeitstabile Datenformate in der elektronischen Aktenführung*. Landesverwaltung Nordrhein-Westfalen, 2012

[LRC+03] Luebke, David ; Reddy, Martin ; Cohen, Jonathan D. ; Varshney, Amitabh ; Watson, Benjamin ; Huebner, Robert: *Level of detail for 3D gra-*

phics. 1st ed. Boston, MA : Morgan Kaufmann Publishers, 2003 (The Morgan Kaufmann series in computer graphics and geometric modeling)

[OK12] ONG, Yongzhi ; KURTH, Winfried: A Graph Model and Grammar for Multiscale Modelling using XL. In: *IEEE International Conference on Bioinformatics and Biomedicine Workshops* (2012)

[OOS97] OZAKI, Toru ; OTSUKA, Tatsushi ; SHIMIZU, Seiya: 3D-CG System with Video Texturing for Personal Computers. In: *FUJITSU Sci. Tech* 33 (1997), S. 170–176

[Ope13] OPENGL: *Calculating a Surface Normal*. 2013. – https://www.opengl.org/wiki/Calculating_a_Surface_Normal – zuletzt aufgerufen am 15.4.2015

[Ope15] OPEN SOURCE GEOSPATIAL FOUNDATION: *GDAL - Geospatial Data Abstraction Library*. 2015. – http://www.gdal.org/ – zuletzt aufgerufen am 15.4.2015

[Par12] PARTICLE IN CELL CONSULTING LLC: *Code Optimization: Speed up your code by rearranging data access*. 2012. – https://www.particleincell.com/2012/memory-code-optimization/ – zuletzt aufgerufen am 15.4.2015

[PHHM97] PRUSINKIEWICZ, Przemyslaw ; HANAN, Jim ; HAMMEL, Mark ; MECH, Radomir: L-systems: from the Theory to Visual Models of Plants. In: *Plants to Ecosystems. Advances in Computational Life Sciences*. 1997, S. 1–27

[RM00] RITTER, Nils ; MIKE, Ruth: *GeoTIFF Format Specification GeoTIFF Revision 1.0*. 2000. – http://www.remotesensing.org/geotiff/spec/geotiffhome.html – zuletzt aufgerufen am 15.4.2015

[Sev02] SEVALDRUD, Thomas: *Efficient Visualization of Triangulations: Part I*. 18.10.2002. – http://folk.uio.no/inftt/Div/visualization1.pdf – zuletzt aufgerufen am 15.4.2015

[Sta14] STANFORD, University of: *The Stanford 3D Scanning Repository*. 2014. – https://graphics.stanford.edu/data/3Dscanrep/ – zuletzt aufgerufen am 15.4.2015

[Uni15] UNIVERSITÄT GÖTTINGEN: *GroIMP API Dokumentation*. 2015. – http://wwwuser.gwdg.de/~groimp/api/overview-summary.html – zuletzt aufgerufen am 15.4.2015

[U.S89] U.S. Department of Defense: *Military specification digital terrain elevation data (DTED).* Fairfax, VA : U.S. Dept. of Defence, Defence Mapping Agency, 1989