UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Erstellung eines dreidimensionalen Geländemodells des Mittelrheintals

Studienarbeit

im Studiengang Computervisualistik

vorgelegt von
Jakob Bärz

Betreuer: Prof. Dr.-Ing. Stefan Müller
(Institut für Computervisualistik, AG Computergraphik)

Koblenz, im September 2006

Erklärung

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

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

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Motivation

Am 11. Februar 2000 erfolgte der Start der Shuttle Radar Topography Mission (SRTM) [22], — ein Gemeinschaftsprojekt der National Aeronautics and Space Administration (NASA, USA) [2], der National Imagery and Mapping Agency (NIMA, USA) [13], des Deutschen Zentrums für Luft- und Raumfahrt e.V. (DLR, Deutschland) [10] und der Agenzia Spaziale Italiana (ASI, Italien) [15] — mit dem Ziel einer Höhenkartierung der Erdoberfläche. In elf Tagen und nach 181 Umkreisungen der Erde konnten 80% der Landmasse — Lebensraum von 95% der Erdbevölkerung — zwischen 60 ° nördlicher Breite und 58 ° südlicher Breite mittels aktivem Radarsystem und Interferometrie vermessen werden. Die Kosten von zirka 230 Millionen US-Dollar spiegeln eindrucksvoll das wissenschaftliche und wirtschaftliche Interesse an globalen Digitalen Geländemodellen, zu deren Erstellung die gewonnenen Geobasisdaten dienen, wieder.

Während Geographische Informationssysteme (GIS) [25], welche für die Erfassung, Verwaltung, Analyse und Präsentation solcher Geobasisdaten zuständig sind, bis Anfang der 1990er Jahre hauptsächlich von Behörden und Firmen genutzt wurden, bescherten ihnen Angebot und Nachfrage des offenen Marktes in den letzten Jahren einen bemerkenswerten Wachstum. Begünstigt durch den explosionsartigen Leistungsanstieg moderner Hardwaresysteme und die ständig zunehmende Verfügbarkeit raumbezogener Basisdaten, verbreiterte sich das Spektrum der Anwendungsmöglichkeiten maßgeblich. Wissenschaftliche Applikationen für Geologen und Geophysiker, zur Erdbebenforschung und Vulkanbeobachtung konnten ebenso bereichert werden wie Systeme im zivilen Bereich zur Planung im Bau- und Telekommunikationswesen. Das Militär profitierte von topographischen Karten zur Unterstützung von Missionsplanungen sowie von Modellen zur realistischen Gestaltung von Simulationen und für den Einsatz in Waffenleitsystemen.

Die enorme Bandbreite der Verwendungszwecke und hohe Aktualität Geographischer Informationssysteme legen die Auseinandersetzung mit einem Thema aus dem Bereich der Geoinformatik, welche eine Verknüpfung zwischen der Informatik und den Geowissenschaften herstellt und die wissenschaftliche Grundlage von GIS bildet, nahe. Des Weiteren war eine Studienarbeit zur Erstellung eines virtuellen Landschaftsmodells der Mittelrhein-Region in der Arbeitsgruppe Computergraphik an der Universität Koblenz-Landau bislang ein Desiderat und bedarf somit keiner ausführlichen Rechtfertigung.

1.2 Ziel

Als Ziel der vorliegenden Studienarbeit wurde die Erstellung und Präsentation eines dreidimensionalen und texturierten Geländemodells des Mittelrheintals definiert. Den Ausgangspunkt und die Grundlage bildete die Beschaffung und die Aufbereitung von digitalen Geobasisinformationen in Form eines Digitalen Geländemodells (DGM) und Digitaler Orthophotographien (DOP). Anspruch an die Höhendaten war eine höchstmögliche Aktualität und lückenlose Abdeckung sowie maximale Höhengenauigkeit und eine geringe Gitterweite. Bei den Luftaufnahmen zur Texturie-

rung des Modells lag das Augenmerk auf optimaler Farbtiefe und Bodenauflösung. Zentraler Schwerpunkt war die Auswahl und Implementierung eines für die spezifischen Voraussetzungen der gegebenen geotopographischen Datenbasis angepassten Rendering-Algorithmus' zur Visualisierung der modellierten Landschaft.

Die Lauffähigkeit unter *WindowsXP* war ebenso Grundvoraussetzung wie die Entwicklung mit der Programmiersprache *C++* unter Verwendung der *Open Graphics Library* (OpenGL). Das System zur Ausgabe der dreidimensionalen Ansicht musste ein Modul zur Verwaltung und Darstellung der geospezifischen Orthophotografien als Texturen sowie eine Komponente zur interaktiven Navigation durch das Modell für den Benutzer bereitstellen. Der Umfang der Daten machte aus Gründen der Performanz und des Speicherbedarfs eine effiziente Strukturierung für den Einsatz eines *Level of Detail*-Verfahrens unumgänglich.

Wünschenswert war eine Darstellung in Echtzeit und die Vereinfachung der Anpassung an veränderte Anforderungen und der Erweiterung um zusätzliche Bausteine.

1.3 Aufbau

In Kapitel 2 wird die modellierte Landschaft geographisch abgesteckt, die spezifischen Eigenschaften der zugrunde liegenden Geodaten vorgestellt, das eingesetzte Rendering-Verfahren erläutert und weitere Grundlagen näher erklärt.

Konzeptuelle Vorüberlegungen zur Realisierung des gewählten Algorithmus und zur Strukturierung der Daten sowie Ausführungen zur Aufbereitung des Digitalen Geländemodells und der Digitalen Orthophotographien sind in Kapitel 3 offen gelegt.

Einzelheiten zur Implementierung schließen sich im folgenden Kapitel 4 an. Hier sind Informationen zur Entwicklungsumgebung und zur Klassenstruktur des Programms wiedergegeben. Abgerundet wird der Abschnitt durch Detailbetrachtungen wichtiger Klassen und deren essentiellen Routinen.

Die Vorstellung der erreichten Ziele und eine Beurteilung der Ergebnisse, insbesondere in Hinblick auf Performanz und Bildqualität, erfolgt in Kapitel 5.

Das Abschlußkapitel 6 gibt schließlich einen Ausblick auf denkbare Optimierungen und zeigt Erweiterungsmöglichkeiten für das System auf.

2 Grundlagen

2.1 Geographie

Der in dieser Studienarbeit modellierte Landschaftsraum ist der rund 50 Stromkilometer umfassende zentrale Abschnitt des Mittelrheingebietes, in dem der Fluss das Rheinische Schiefergebirge in einem Engtal durchbricht. Er schließt Bereiche der Mittelgebirge Hunsrück, Taunus, Westerwald und Eifel ein. Geographisch bildet die Loreley bei Sankt Goarshausen den südlichen und Koblenz den nördlichen Abschluss der Teilstrecke.

In Abgrenzung zu dem "Mittelrheintal" genannten Teil des Rheins zwischen Bingen und Bonn (528.–655. Rheinkilometer) und der als "Welterbe Oberes Mittelrheintal" [21] bezeichneten Kulturlandschaft zwischen Bingen und Koblenz (Kilometer 528–600), wird hier der Begriff "Zentrales Mittelrheintal" mit den genannten Grenzen vorgeschlagen.

"Zentrales Mittelrheintal" zwischen der Loreley und Koblenz:

- Streckenabschnitt: 550.–600. Rheinstrom-Kilometer
- Gesamtfläche: $416 km^2$ (104 Blatt der Deutschen Grundkarte á 2 x $2km$)

Ebene Ausdehnung[1]:

- Süd-Nord: $34km$ ($50\,°6'56'' - 50\,°25'4''$ nördliche Breite)
- West-Ost: $20km$ ($7\,°28'45'' - 7\,°46'6''$ östliche Länge)

Vertikale Ausdehnung[2]:

- Minimale Höhe über Normalnull: $55,4m$
- Maximale Höhe über Normalnull: $533,7m$

Abbildung 1 illustriert das abgedeckte Areal und gibt die Blattnamen der Deutschen Grundkarte wieder.

2.2 Datenbasis

2.2.1 Digitales Geländemodell

In der Literatur finden sich unterschiedliche Definitionen der Begriffe *Digitales Geländemodell (DGM)*[3] und *Digitales Höhenmodell (DHM)*[4]. Zur Schaffung von Konsistenz und Vermeidung von Unklarheiten wird im Rahmen dieser Arbeit durchweg der Begriff *Digitales Geländemodell* mit der nachstehenden Definition verwendet. Dies geschieht in Übereinstimmung mit der Bezeichnung der vom Landesamt für Vermessung und Geobasisinformation Rheinland-Pfalz (LVermGeo)[11] zur

[1] World Geodetic System 1984, Geographische Koordinaten (Greenwich)
[2] Deutsches Haupthöhennetz 1985 (DHHN85)[23], Höhen in Metern über Normalnull
[3] engl.: Digital Terrain Model (DTM)
[4] engl.: Digital Elevation Model (DEM)

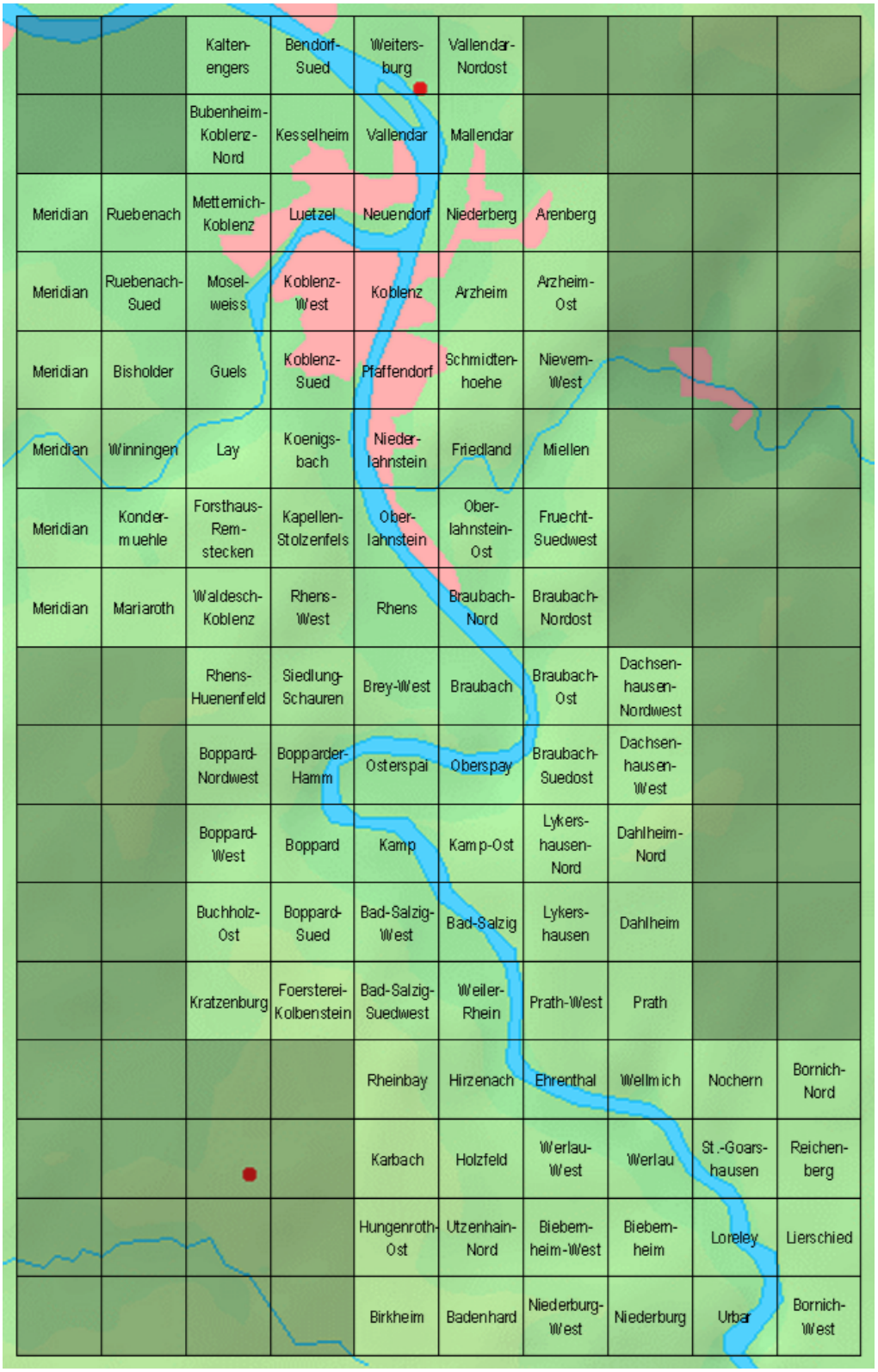

Abbildung 1: Karte “Zentrales Mittelrheintal”

Verfügung gestellten Komponente der ATKIS-Konzeption[5] [14][6].

Ein DGM ist eine numerische Annäherung an die räumliche Form der Erdoberfläche. Als Erdoberfläche wird hier die Grenzschicht zwischen der Erdkruste bzw. den Oberflächengewässern auf der einen und der Atmosphäre auf der anderen Seite bezeichnet. Implizit schließt dies die Berücksichtigung von Vegetation und Bauwerken aus.

Basis des Modells bildet eine endliche Menge dreidimensionaler Positionsangaben von so genannten Stützpunkten. Die Raumpunkte (XYZ-Tripel) des hier eingesetzten DGM lagen in einer regelmäßigen gitterförmigen Anordnung vor. Da jeder Lagekoordinate nur ein Höhenwert zugeordnet ist, wird die Datendimension auch 2,5D genannt.

Wie in der Landesvermessung der alten Bundesländer in Deutschland noch heute üblich, bildet das Deutsche Hauptdreiecksnetz 1990 (DHDN90)[9] die geodätische Grundlage für den vermessungstechnischen Raumbezug. Das geodätische Datum basiert auf dem Ellipsoid von Bessel (1841) mit dem Trigonometrischen Punkt Rauenberg als Fundamentalpunkt (fälschlicherweise auch Potsdam Datum genannt). Als horizontales Kartenbezugssystem wurde das Gauß-Krüger-Koordinatensystem[27] mit 3° breiten Meridiansteifen eingesetzt und die Lage der Punkte durch ihren Koordinatenrechts[6]- (X-Wert) und Koordinatenhochwert[7] (Y-Wert) festgehalten. Die Höhenwerte (Z-Werte) wurden als Funktion der Lage, gemessen über der Bezugsfläche Normal-Null (NN), angegeben.

Für die Modellierung stand ein Digitales Geländemodell mit einer Gitterweite von $10m$ (ein Lagestützpunkt pro 10 Meter) und einer Höhengenauigkeit von etwa $50cm$ zur Verfügung. Die ebene Ausdehnung umfasst das gesamte Begrenzungsrechteck des modellierten Gebietes. Bei einer Gesamtfläche von 20 x $34km$ entspricht dies 2001 x 3401 = 6.805.401 Höhenwerten. Die Lieferung erfolgte als Textdatei im ASCII-Code mit einer Gesamtgröße von etwa 260 Megabyte. Jede Zeile lag im Format *Hochwert, Rechtswert, Höhe* vor.

Ausschnitt aus der DGM-Datei:		
3392000.000	5588000.000	65.348
3392010.000	5588000.000	65.430
3392020.000	5588000.000	65.551
3392030.000	5588000.000	65.645
3392040.000	5588000.000	65.742
3392050.000	5588000.000	65.754
3392060.000	5588000.000	65.753

[5] Amtliches Topographisch-Kartographisches Informationssystem der Arbeitsgemeinschaft der Vermessungsverwaltungen der Länder der Bundesrepublik Deutschland (AdV)

[6] Rechtwinkliger Abstand vom Hauptmeridian in Metern

[7] Entfernung des Punktes zum Äquator in Metern

Abbildung 2: Digitale Orthophotografie “Deutsches Eck”

2.2.2 Digitale Orthophotographien

Eine Digitale Orthophotographie (DOP) ist eine maßstabsgetreue und entzerrte Senkrechtaufnahme der Erdoberfläche. Da DOP aus Luftbildfotografien gewonnen werden, sind in diesem Kontext sämtliche zum Zeitpunkt der Aufnahme auf der Erdoberfläche situierten statischen und dynamischen, natürlichen und künstlichen Objekte — wie Vegetation, Bauwerke und Fahrzeuge — mit eingeschlossen.

Die Originalbilder werden eingescannt, d.h. räumlich in Pixel diskretisiert. Die einzelnen Bildpunkte werden anschließend einer Quantisierung in den *True Color* RGB-Farbraum mit 8 Bit pro Farbkanal unterzogen. Die Elimination der durch die fotografische Zentralprojektion und die Höhenunterschiede aufgrund der Unebenheiten des Reliefs naturgemäß auftretenden Verzerrungen erfolgt durch Verschiebung der einzelnen Bildelemente.

Im Rahmen dieser Arbeit lagen farbige DOP mit einer Bodenauflösung von einem Pixel pro 25cm in der Natur vor. Vertrieben werden die Fotografien im Blattschnitt der Deutschen Grundkarte, d.h. in 2 x 2km bzw. 8000 x 8000 Pixel umfassenden Dateien. Die entsprechenden Blattnamen der modellierten Landschaft sind der Abbildung 1 zu entnehmen. Die Abgabe der DOP erfolgt im TIFF[8] mit jeweils einem TFW[9] für die Georeferenzierung und einer Text-Datei mit Meta-Daten. Zur Reduktion der Datenmenge unterliegen die Orthophotodateien einer Wavelet-Komprimierung mit Faktor vier. Die entpackte Gesamtgröße der 104 Bilder mit einer Auflösung von jeweils 8000 x 8000 Bildpunkten und einer Farbtiefe von 24 Bit beträgt etwa 19 Gigabyte. Es folgt ein Beispiel des Metadatei-Formats und ein Ausschnitt einer Digitalen Orthophotografie ist in Abbildung 2 wiedergegeben.

Beispiel einer Meta-Datei:	
0341005556_dop5__c_025	Bezeichnung des Orthobildes
42_04	Bildflugnummer
06.09.04	Bildflugdatum
30cm	Kammerkonstante
1:13000	Bildmassstab
18um	Scanaufloesung
1:5000	Zielmassstab
25cm	Bodenaufloesung
tfw-File	Georeferenzierung
8000	Bildausdehnung rechts
8000	Bildausdehnung links
24	Bittiefe
Bornich-Nord	DGK5 Blattname
3410000.00	Rechtswert links unten
5556000.00	Hochwert links unten
3412000.00	Rechtswert rechts oben
5558000.00	Hochwert rechts oben
Copyright	Copyrightvermerk
Landesamt fuer Vermessung und	
Geobasisinformation Rheinland-Pfalz	
Ferdinand-Sauerbruch-Strasse 15	
56073 Koblenz	
Tel. 0261 492 0	
Fax. 0261 492 492	
Internet www.lvermgeo.rlp.de	

2.3 Rendering Algorithmus

2.3.1 Vorüberlegung

Der bereits ausgeführte Umfang der Daten macht den naiven Ansatz, das gesamte Modell im unoptimierten Originalzustand mit allen Texturen in feinster Auflösung gleichzeitig zu rendern, aus zwei Gründen unmöglich. Auf der einen Seite ist selbst die modernste Hardware nicht in der Lage, eine solche Ausgabe in Echtzeit zu

[8] Tagged Image File Format
[9] TIFF World file

ermöglichen und auf der anderen Seite reichen weder Arbeitsspeicher noch Grafikkartenspeicher aus, um rund 20 Gigabyte Texturen und 3D-Daten aufzunehmen. Zur Bewältigung dieses Problems wurden verschiedene Konzepte, die im Folgenden näher erläutert werden, kombiniert.

Wenn ein Ausschnitt der Landschaft in mehreren Kilometern Entfernung zu dem Betrachter liegt, besteht die Möglichkeit, Daten zu reduzieren, ohne sichtbare Verluste in der Darstellungsqualität hinnehmen zu müssen. Bei einem Gebirge, das beispielsweise aus 80.000 Dreiecken besteht, aber durch seine Distanz zur Kamera mit nur wenigen Pixeln auf dem Viewport vertreten ist, fehlt die Notwendigkeit der Wiedergabe in der gesamten Detailvielfalt — insbesondere unter Berücksichtigung der Tatsache, dass diese Anzahl an Dreiecken mit insgesamt 192 MB Texturdaten belegt wären.

Zur Verdeutlichung, dass es sich hierbei um ein ganz natürliches Phänomen handelt, sei die Analogie zum Auflösungsvermögen des menschlichen Auges erwähnt. Während wir unter idealen Bedingungen die Grashalme zu unseren Füßen noch mühelos erkennen können, nehmen wir in wenigen Kilometern Entfernung nur noch große Bäume wahr.

2.3.2 Chunked Level of Detail

Diese Erscheinung machen sich die so genannten *Level of Detail*-Verfahren (LoD-Verfahren) zu Nutzen. Der Name leitet sich von Staffelung der Geometrie — und gegebenenfalls der Texturen — in die verschiedenen Detailstufen [10] ab. Allen LoD-Verfahren gemeinsam ist der Ansatz, in Abhängigkeit von der Größe eines Objektes auf dem Bildschirm [11], die sich in der Regel aus seiner Distanz zum Augpunkt und seinem Winkel zu der optischen Achse der Kamera berechnen lässt, eine angemessene Qualitätsstufe für seine Ausgabe auszuwählen. Algorithmen zum Management von *Level of Detail* sind ein altbekanntes und dennoch hochaktuelles Thema der Computergrafik. Die vergangenen Jahre haben zahlreiche Studien hervorgebracht, insbesondere auch im Bereich des Rendering von Landschaften.

Aus der Vielzahl unterschiedlicher Algorithmen fiel die Wahl für diese Arbeit zu Gunsten des von Ulrich Thatcher 2001 vorgestellten *Chunked LoD* [26] aus. Eine Begründung liegt in der einfachen Handhabung von *Out of Core*-Dateien, d.h. des Nachladens der Datensätze von der Festplatte zur Laufzeit des Programms. Des Weiteren lassen sich die unterschiedlichen Detailstufen der Texturen nahtlos in die Geometrie-*Level of Detail*. Da es sich hier um ein Verfahren handelt, das auf großen Aggregationen von Dreiecken statt auf einzelnen Eckpunkten operiert, wird der effiziente Einsatz von Vertex- bzw. Textur-Koordinaten-Arrays zur Übergabe en bloc an die Grafikkarte und zur Verringerung der Anzahl der OpenGL-Funktionsaufrufe ermöglicht.

Der Nachteil der relativ rechenintensiven Aufbereitung der Daten ist zu vernachlässigen, da diese einmalig im Vorverarbeitungsschritt abgehandelt werden

[10] engl.: Level of Detail
[11] engl.: Screenspace

kann. Die Beschränkung auf statische Daten ist für diese Arbeit irrelevant.

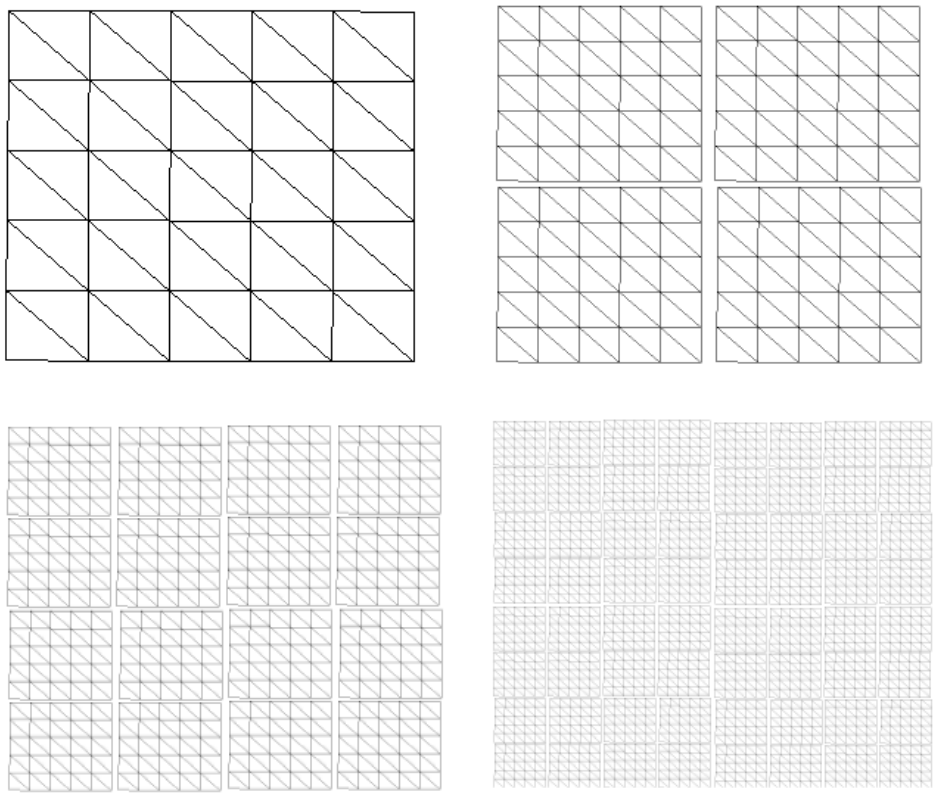

Abbildung 3: Die ersten vier Ebenen eines Quadtrees

2.3.3 Quadtree

Herzstück der Datenorganisation des *Chunked LoD*-Verfahrens ist eine Baum-Struktur, in der jeder innere Knoten genau vier Kindknoten aufweist. Die Anzahl der Kinder zeichnet verantwortlich für den Namen *Quadtree*. Die Knoten speichern statische und voneinander unabhängige Teilblöcke — die so genannten *Chunks* — der Landschaft. Jede Hierarchie-Ebene beinhaltet das vollständige Terrain, jeweils in unterschiedlicher Detailstufe, abhängig von der Tiefe im Baum. Da alle Knoten in dieser Implementierung über die gleiche Anzahl an Dreiecken verfügen, vervierfacht sich der Detailgrad pro Ebene.

Der Wurzelknoten verkörpert das gesamte Modell durch einen einzigen quadratischen *Chunk* mit sehr niedrigem Detailgrad. Die vier Kindknoten auf der Ebene darunter repräsentieren ihren Vaterknoten durch vier gleich große, ebenfalls quadratische Teilblöcke und infolgedessen in größerer Detailvielfalt. Auf der dritten Ebene liegen also bereits 16 *Chunks*, von denen wiederum jeweils vier einen Block

von Ebene zwei in besserer Qualität verkörpern. Dies wird so oft iteriert bis eine Stufe erreicht ist, die das Terrain in der vollen Auflösung wiedergibt. Abbildung 3 stellt die obersten vier Ebenen eines Quadtree in einer schematischen Illustration dar.

2.3.4 Generierung der Detailstufen

Eine Stärke des *Chunked LoD* besteht in der effizienten Berechnung der einzelnen Detailgrade in einem Vorverarbeitungsschritt, der sich demzufolge nicht negativ auf die Rendering-Performanz auswirkt. Im Allgemeinen wird jeder Stufe eine erlaubte maximale Abweichung von den Originaldaten zugewiesen. Abhängig von dieser zugelassenen Fehlertoleranz, kann für jede Ebene ein optimales Dreiecksmodell des *Chunks* bestimmt werden. Die Staffelung der Detailvielfalt ergibt sich automatisch durch Verdopplung der Abweichung auf der Vaterebene.

Für diese Implementierung des Algorithmus' wurde jedoch eine andere Vorgehensweise angewendet. Ein wichtiger Grund liegt in der Tatsache, dass Thatcher im Originalpaper selbst für die feinste Detailstufe noch eine kleine Fehlertoleranz vorgesehen hat, was der Grundidee, ein möglichst naturgetreues Modell unter Verwendung aller zur Verfügung stehenden Daten zu schaffen, entgegenstand. Ein praktischer Vorteil liegt in der einfacheren Implementierung des Vorverarbeitungsschrittes und darin, dass irreguläre Dreiecksnetze die Texturierung erheblich komplizierter gestaltet hätten.

Um die Vorteile eines äquidistanten Höhengitters für eine regelmäßige Triangulierung zu erhalten, erfolgte auf der untersten Baum-Ebene eine Gliederung aller vorliegenden Höhenwerte in Quadrate gleicher Kantenlänge. Zur Reduktion der Datenmenge wurde auf der nächst gröberen, darüber liegenden Vorfahrenstufe jeder zweite Höhenwert einfach ausgelassen. Dadurch besteht die Möglichkeit, für jeden *Chunk*, unabhängig von seiner Detailstufe und seiner Position, exakt dieselbe Anordnung des Dreiecksnetzes und der Texturkoordinaten zu verwenden.

2.3.5 Selektion der Detailstufen

Zur Darstellung des Terrains muss der Quadtree zur Laufzeit ausgewertet werden. Der einfachste Weg, dies zu realisieren, ist die Traversierung des Baumes, beginnend mit dem Wurzelknoten. Bei jedem Knoten wird anhand einer Fehlermetrik abgewogen, ob sich die vorliegende Detailstufe des *Chunks* bereits zur Ausgabe eignet. Sollte dies zutreffen, kann der ganze Teilblock zum Rendern an die Grafikkarte übergeben werden. Andernfalls müssen rekursiv alle vier Kindknoten diesem Test unterzogen werden. Die Rekursion terminiert, vorausgesetzt es konnte vorher keine befriedigende Vereinfachung gefunden werden, spätestens mit Erreichen der vollen Qualität der Originaldaten auf der untersten Baumebene.

Die Entscheidung über die Eignung zur Darstellung eines Terrainblocks wird auf Basis eines statischen, vorab definierten Schwellwertes getroffen. Im Zuge der Traversierung des Quadtrees erfolgen eine Projektion des maximalen geometrischen

Fehlers des jeweils getesteten *Chunks* in das Bildschirmkoordinatensystem des Ausgabefensters und ein Vergleich mit dem Schwellwert. So besteht die Garantie, dass der resultierende Fehler auf dem Monitor eine maximale Anzahl an Bildpunkten nicht übersteigt.

Die Bestimmung des maximalen Pixelfehlers auf dem Bildschirm kann mit einer konservativen — vom Winkel zwischen der Kamera-Sichtachse und der Terrain-Oberfläche unabhängigen — Formel erledigt werden. Folgende Parameter fließen in die Berechnung ein:

- maximaler Pixelfehler auf dem Bildschirm: ρ
- maximaler geometrischer Fehler des *Chunks*: δ
- minimale Distanz zwischen Augpunkt und Bounding Box des *Chunks*: D
- perspektivischer Skalierungsfaktor: K
- Höhe des Ausgabefensters: $viewportheight$
- vertikaler Öffnungswinkel der Kamera: $verticalfov$

In einem ersten Schritt wird der perspektivische Skalierungsfaktor — unter Zuhilfenahme des vertikalen Öffnungswinkels der Kamera — als Verhältnis von der Höhe des Ausgabefensters zu der Höhe der *far clipping plane* bestimmt:

$$K = \frac{viewportheight}{2 * tan(\frac{verticalfov}{2})}$$

Neben der Distanz des Betrachters zum *Chunk* und dem vorab berechneten maximalen geometrischen Fehlers des *Chunks*, dient dieser Faktor der Kalkulation der gesuchten Abweichung des gerenderten Modells gegenüber den Originaldaten im Pixelraum des Darstellungsfensters:

$$\rho = \frac{\delta}{D} * K$$

Diese Metrik ist eine Approximation, da sich der Pixelfehler auf die Mitte des Ausgabefensters bezieht. Von dem Standpunkt der Effizienz betrachtet, handelt es sich jedoch um eine übliche und ausreichende Abschätzung. Die Wahl eines von der Sichtachse unabhängigen Verfahrens hat zwei Gründe. Erstens verringert sich zwar der geometrisch bedingte sichtbare Pixelfehler bei senkrechter Betrachtung der Landschaft, jedoch ist die Selektion eines niedrigen Detaillevels in Hinblick auf die damit verbundene gröbere Texturauflösung in dieser Situation kontraproduktiv. Zweitens bleibt das Terrain bei Rotation der Kamera an einem Punkt stabil.

2.3.6 Texturen

In den Grundzügen gestaltet sich die Texturierung beim *Chunked LoD* verhältnismäßig einfach. In dem Vorverarbeitungsschritt wird bei der Generierung der Detailstufen jedem statischen Terrainblock eine statische Textur mit adäquater

Auflösung zugeordnet. Da die Erstellung der verschiedenen Detailgrade bei den Texturen analog zu dem für die Geometrie beschriebenen Verfahren abläuft, wird von einer ausführlichen Darstellung abgesehen. Auf der Blattebene des Quadtrees liegen die Daten der Digitalen Orthophotografien in der Originalauflösung als Texturen vor. Bei jedem Übergang auf die nächste gröbere Vorfahrenstufe, wurden Breite und Höhe der Textur durch Auslassung jedes zweiten Bildpunktes schlichtweg halbiert.

2.3.7 Skirts

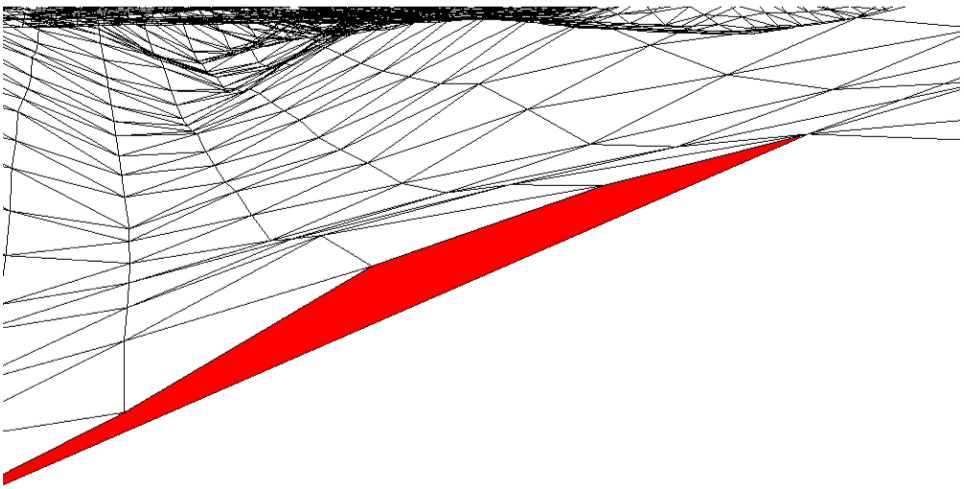

Abbildung 4: “Crack”

Dieser Algorithmus führt aufgrund der unterschiedlichen Abstände der Landschaftsblöcke zu dem Betrachter dazu, dass benachbarte *Chunks* mit unterschiedlichen Detailstufen ausgegeben werden. Durch die (bei der Reduktion auf gröbere Repräsentationen) weggelassenen Eckpunkte, stimmen die Berührungskanten nicht mehr überein und es kommt zu Aussparungen im Dreiecksnetz. Obwohl diese Risse (*Cracks*) durch den von der Fehlermetrik garantierten Schwellwert nur wenige Pixel hoch sind, müssen solche sichtbaren Mängel aus der Darstellung entfernt werden. In Abbildung 4 wurde ein solcher *Crack* in einer Nahaufnahme im *Wireframe*-Modus festgehalten und rot eingefärbt.

Um dieses Problem zu beseitigen, schlägt Thatcher einen einfachen aber wirkungsvollen Ansatz vor. Die Außenkanten der Terrainblöcke werden mit einem vertikalen, nach unten hängenden *Skirt* umgeben. Während die obere Kante des *Skirts* die Ränder des *Chunks* identifiziert, hat die untere keinen festen Abschluss. Notwendige Anforderung ist, dass sie weiter hinab ragen muss, als die gröbste Vereinfachung dieser Flanke und zugleich möglichst kurz sein sollte, um die Füllrate zu erhalten. Die Bestimmung der exakten Höhe des *Skirts* wird demnach in Abhängigkeit von

dem geometrischen Fehler der Detailstufe des Blocks gegenüber dem Originalmodell getroffen.

Die *Skirts* werden zusammen mit den *Chunks* abgespeichert und bestehen aus simplen und statischen Dreiecksstreifen. Da sie nicht in die benachbarten Blöcke hineinragen, erfolgt die Texturierung einfach durch Zuweisung der existierenden Textur des *Chunks*.

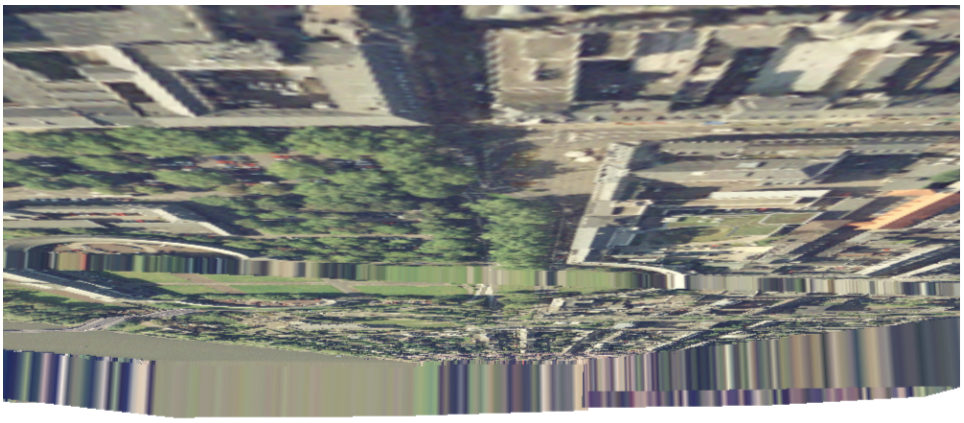

Abbildung 5: “Skirts”

Abbildung 5 zeigt das Modell zu Demonstrationszwecken mit ausgeschaltetem *polygon backface culling* von unten. So konnten die sonst nicht sichtbaren Skirts visualisiert werden.

2.4 Indexed Triangle List

Die Open Graphics Library[1] wird oft als low-level Programmierschnittstelle bezeichnet, da zur Darstellung jeglicher geometrischer Objekte eine Tesselierung in vordefinierte Basisprimitiven vorausgesetzt wird[20]. Das Rendering erfolgt unter Spezifikation des gewünschten Primitiventyps und einer Sequenz von Eckpunkten mit den dazugehörigen Daten. Der *Chunked LoD*-Algorithmus bedient sich der Dreiecke als *low-level* Primitiven zur Zusammensetzung der *Chunks*. OpenGL interpretiert demzufolge eine Gruppe von je drei Punkten aus der Sequenz als Eckpunkte eines individuellen Dreiecks. Die Definition und das Rendering der Primitiven kann auf unterschiedliche Weise erledigt werden.

Mit dem *Immediate Mode* stellt OpenGL ein flexibles Interface zum Zeichnen der Geometrie zur Verfügung. Die Eckpunkte und ihre assoziierten Daten werden von *glBegin(*mode*)* und *glEnd()* eingerahmt, wobei *mode* den Primitiventyp festlegt. Für die Übergabe jedes einzelnen Eckpunktes und jedes einzelnen Eckpunkt-Attributes — wie die Farbe, die Texturkoordinaten oder die Normale — an die

Grafikkarte wird ein separater Befehl benötigt. Der diesem Modus durch die Vielzahl der Funktionsaufrufe inhärente Overhead dient als Ausschlusskriterium des so genannten *glBegin() / glEnd() Paradigmas* für die Ausgabe größerer Aggregationen von Dreiecken.

Eine effizientere Methode stellt die Verwendung von Datenarrays dar. Anstatt jeden Vertex, jede Farbe oder jede Texturkoordinate einzeln per Aufruf einer OpenGL-Funktion zu übergeben, ist die Vordefinition separater Arrays erlaubt. Diese getrennten Vertex-, Farb- und Texturkoordinaten-Arrays dienen der Konstruktion einer ganzen Folge an Primitiven mit einem einzigen Aufruf, z.B. von *glDrawArrays(*mode*,* first*,* count*)*. So kann eine Anzahl von *count* Primitiven vom Typ *mode* auf einen Schlag an die Grafikhardware weitergereicht werden. *First* gibt hierbei den Index des ersten benötigten Vertex im Vertexfeld an.

Die Triangulierung eines regelmäßigen Gitters erzeugt bei n^2 Punkten $2*(n-1)^2$ Dreiecke mit $3 * 2 * (n-1)^2$ Eckpunkten. Jeder Vertex — und jedes dazugehörige Attribut — taucht also fast sechsmal in seinem entsprechenden Array auf. Durch diese Redundanzen wird die Geschwindigkeit des Grafikbusses zur neuen Schwachstelle. Eine Möglichkeit, die Datenmenge der Vertexarrays zu verringern, stellen die so genannten *triangle strips* dar. Hierbei handelt es sich um eine Sequenz von Dreiecken, die sich ihre Eckpunkte teilen. Zur Dopplung von Vertexdaten kommt es bei diesem Verfahren lediglich durch degenerierte Dreiecke (Primitiven, die nicht dargestellt werden, weil ihre Punkte auf einer Linie liegen), die zum Sprung an den Anfang einer neuen Zeile des Eckpunkt-Gitters benötigt werden.

OpenGL bietet jedoch ein weiteres Kommando zum Zeichnen von Primitiven aus einem Datenarray heraus, das sich in diesem Kontext noch besser eignet. Anders als bei den bisher beschriebenen Vorgehensweisen definiert *glDrawElements(*mode*,* count*,* type*,* indices*)* die Abfolge, in der die Dreiecke zu konstruieren sind, über eine eigenständige Tabelle mit Indizes (*indices*) vom Datentyp (*type*), die auf die Positionen der benötigten Eckpunkte im Vertex-Array verweisen. Dadurch können Redundanzen in den Datenarrays völlig vermieden werden. Die Index-Liste gibt dieser Alternative den Namen *Indexed Triangle Lists* (Indexierte Dreieckslisten).

Der nächste Optimierungsschritt wäre nun der Einsatz von *Vertex Buffer Objects*, bei denen die Vertex-Arrays direkt auf der Grafikkarte abgelegt werden können und nicht mehr bei jedem Rendervorgang vom Hauptspeicher aus übertragen werden müssen. Diese Erweiterung ist jedoch nur auf aktueller Hardware lauffähig und kam in dieser Arbeit nicht zum Einsatz.

2.5 Bounding Volume

Ein Bounding Volume ist ein sehr einfacher geometrischer Körper der eine komplexere Form, hier das Dreiecksnetz des Landschaftsblocks, komplett einschließt. Den Zwecken dieser Implementation am besten angemessen ist ein Quader, dessen Kanten an den Achsen des Weltkoordinatensystems ausgerichtet sind, was die Bezeichnung *Axis-Aligned Bounding Box* rechtfertigt. Für den *Chunked LoD*-Algorithmus erfüllen die Bounding Volumes der einzelnen Terrainblöcke zwei wichtige Aufgaben.

Erstens erleichtern sie die Distanzberechnung zwischen *Chunk* und Augpunkt des Betrachters für die Fehlermetrik und zweitens ermöglichen sie das Culling der Terrainblöcke am View Frustum.

2.6 View Frustum Culling

Eine effiziente Methode zur Reduktion der zu rendernden Terrainblöcke und somit zur Optimierung der Framerate ist das *View Frustum Culling*[19][3]. Das Ziel des Cullings ist, die Objekte, die für den Betrachter ohnehin unsichtbar sind, da sie sich außerhalb des Sichtfeldes befinden, herauszufiltern und aus dem Rendervorgang auszuschließen. So werden nur alle potentiell sichtbaren Objekte an die Grafikkarte übergeben. Zur Berechnung des Sichtfeldes werden die mit *gluLookAt* und *gluPerspective* definierten Parameter der virtuellen Kamera zu Hilfe genommen. Durch die perspektivische Zentralprojektion hat das sichtbare Volumen die Form eines abgeschnittenen Pyramidenstumpfes. Basis der Pyramide bildet die *far clipping plane*, während der Augpunkt der Kamera die Spitze festlegt. Die Pyramide wird durch die *near clipping plane* stumpfartig abgeschnitten.

Die Datenstruktur eines Quadtrees eignet sich optimal für den Einsatz von View Frustum Culling. Während der Traversierung des Baumes zur Selektion der Detailstufen wird bei jedem *Chunk* vorab geprüft, ob er sich überhaupt innerhalb des Sichtvolumens befindet. Dieser Test kann unter Zuhilfenahme der für die Distanzberechnung bereits vorliegenden Bounding Box des *Chunks* extrem effizient gestaltet werden. Liegt der komplette umfassende Quader außerhalb des Pyramidenstumpfes, kann der Knoten und alle seine Kinder mit einem Mal verworfen werden. Dadurch wird nicht nur der Grafikhardware durch Verminderung der zu rendernden Geometrie Arbeit abgenommen, sondern sogar die CPU entlastet, da die Traversierung des Baumes und die LoD-Berechnungen verkürzt werden.

2.7 Out of Core

Der Begriff *Out-of-Core* bezieht sich auf Algorithmen, die mit Datensätzen arbeiten, welche zu groß für den Arbeitsspeicher sind. Charakteristisch für diese Methoden sind die Auslagerung von Informationen auf dem Massenspeicher und das Nachladen bei Bedarf. Eine effiziente Struktur der Daten ist von großer Bedeutung, um die Anzahl der notwendigen Zugriffe auf den langsamen Hintergrundspeicher (in diesem Fall die Festplatte) möglichst gering zu halten. Geographische Informationssysteme sind geradezu ein Paradebeispiel für *Out-of-Core*-Verfahren.

Dadurch, dass beim *Chunked LoD* alle benötigten Textur- und Geometriedaten für jeden Terrainblock bereits in sich geschlossen vorliegen, ist ein elementares *Paging* der *Chunks* unproblematisch realisierbar. Im Idealfall übernimmt ein separater Thread das Nachladen der Blöcke und die Freigabe des Speichers.

3 Konzept

3.1 Auflösung der Chunks

Die Auswahl einer optimalen Auflösung der einzelnen *Chunks* hat sich als zentrales Problem und Angelpunkt der Implementierung des Rendering-Algorithmus' erwiesen. Daher seien die Überlegungen zu dieser Fragestellung den weiteren Ausführungen voran gestellt.

3.1.1 Randbedingungen

Die *Chunk*-Auflösung (*chunk resolution*), d.h. die ebene Ausdehnung der Terrainblöcke in X- und Y-Richtung, gemessen in der Anzahl an Höhenwerten, ist auf jeder Stufe des Quadtrees gleich. Sie ist maßgeblich für die Effizienz des Renderings und bestimmt selbstredend die Auflösungen der dazu gehörigen Texturen (*texture resolution*). Daraus ergeben sich folgende Vorgaben:

- ⋆ OpenGL erwartet Texturen mit *power-of-two* Dimensionen. Um rechenaufwändige und gegebenenfalls verlustbehaftete Skalierungen der Texturen zu vermeiden, empfiehlt sich also die Verwendung von Texturen mit einer Seitenlänge, die sich durch eine Zweierpotenz ausdrücken lässt.

$$texture\ resolution \in \{2^2, 4^2, 8^2, 16^2, 32^2, 64^2, 128^2, 256^2, 512^2, 1024^2, \ldots\}$$

- ⋆ Das Verhältnis zwischen der Gitterweite des Digitalen Geländemodells (100 Höhenwerte pro Kilometer) und der Bodenauflösung der Digitalen Orthophotographien (4000 Bildpunkte pro Kilometer) beträgt $1 : 40$. Eine Textur zu finden, die eine Zweierpotenz als Seitenlänge aufweist und deren Abdeckung gleichzeitig mit der Abdeckung eines *Chunks* exakt übereinstimmt, ist demzufolge nicht möglich. Es sollte eine Auflösung anvisiert werden, die so nah wie möglich an eine *power-of-two* Auflösung herankommt. Dadurch kann der Mehraufwand an überflüssigen Daten (*Overhead*) reduziert werden, da die verbleibende unabgedeckte Bildfläche mit ungenutzten schwarzen Pixeln aufgestockt wird. Diese Bedingung erfüllen die in der nachstehenden Liste aufgeführten Werte am Besten:

Höhenwerte	Bildpunkte	Textur	Welt (m)	Overhead
1^2	40^2	64^2	10^2	$0,6\%$
3^2	120^2	128^2	20^2	$0,121\%$
6^2	240^2	256^2	60^2	$0,121\%$
12^2	480^2	512^2	120^2	$0,121\%$
25^2	1000^2	1024^2	250^2	$0,046\%$
51^2	2040^2	2048^2	510^2	$0,0078\%$
102^2	4080^2	4096^2	1020^2	$0,0078\%$

- ⋆ Weiterhin bildet die aufwendige Form der Abdeckung des Areals durch die vorliegenden DOP eine zusätzliche Einschränkung. Um unnötigen Overhead

an Daten und Rechenaufwand aufgrund der Gebiete des DGM ohne Texturen zu vermeiden, sollte für die Seitenlänge der *Chunks* eine Zahl gewählt werden, die sich durch Division der Seitenlänge des Blattschnitts der Deutschen Grundkarte von 200 Höhenwerten mit einer Zweierpotenz ergibt: *Chunk*-Seitenlänge $= 200/n^2$. Dadurch kann sichergestellt werden, dass auf allen Ebenen unterhalb der Ebene, auf der die *Chunk*-Seitenlänge gleich der Blattschnitt-Seitenlänge ist, keinerlei unabgedeckte Bereiche in den *Chunks* vorliegen. Terrainblöcke ohne Textur können so komplett vom Rendering ausgeschlossen werden. Die Einhaltung dieser Einschränkung kann nur von den folgenden Varianten verwirklicht werden:

Höhenwerte	Bildpunkte	Textur	Welt (m)
25^2	1000^2	1024^2	250^2
50^2	2000^2	2048^2	500^2
100^2	4000^2	4096^2	1000^2
200^2	8000^2	8192^2	2000^2

⋆ Da die Erfahrung gezeigt hat, dass mindestens acht bis fünfzehn Texturen pro Frame benötigt werden, ist eine Seitenlänge von 2048 mit einer Größe von $2048 * 2048 * 3$ Byte $= 12$ Megabyte schon zu groß für den Speicher moderater Grafikkarten.

⋆ Eine *chunk resolution* von 16 x 16 Höhenwerten resultiert bereits in 16.250 Texturen allein auf der untersten Ebene des Baumes. Eine möglichst hohe Auflösung ist demnach wünschenswert.

Alle diese Einschränkungen führten letztlich zur Wahl einer *chunk resolution* von 25 x 25 Höhenwerten, die sich als guter Kompromiss erweisen konnte.

3.1.2 Konsequenzen

Die Auswahl der *Chunk*-Auflösung zog die nachstehenden, tabellarisch zusammengefassten Auswirkungen mit sich:

Chunk-Auflösung:	25 x 25 Höhenwerte
Texturauflösung:	1024 x 1024 Pixel
Texturgröße:	3 Megabyte
Gesamtanzahl der Texturen	
faktisch vorliegend:	8.889
theoretisch benötigt:	$8.874\frac{2}{3}$
Gesamtgröße der Texturen	
faktisch vorliegend:	26.667 Megabyte
theoretisch benötigt:	$\sim$ 25.390 Megabyte
Gesamtverschnitt:	$\sim$ 1.276 Megabyte
durch fehlende DOP Abdeckung:	$\sim$ 41 Megabyte
durch Auffüllung zu 2^n:	$\sim$ 1.235 Megabyte

Für die *Chunk*-Ausdehnung in Höhenwerten, die Anzahl der Texturen, den Abdeckungsgrad *in* % und den daraus resultierenden Overhead *in Megabyte* in Abhängigkeit von der Ebene im Quadtree ergaben sich diese Werte:

Baumebene	0	1	2	3	4	5	6	7	8
Ausdehnung	6400^2	3200^2	1600^2	800^2	400^2	200^2	100^2	50^2	25^2
Texturen	1	2	5	11	30	104	416	1664	6656
Abdeckung	10, 2	20, 3	32, 5	59, 1	86, 7	100	100	100	100
Overhead	2, 7	4, 78	10, 125	13, 5	12	0	0	0	0

Der Teilbereich des DGM, der durch die DOP abgedeckt wird, hat keine quadratische Form, sondern ist durch mehrere Rechtecke zu beschreiben. Auf den höheren Ebenen des Quadtrees kommt es daher zu *Chunks*, die Flächen ohne Abdeckung aufweisen oder komplett außerhalb des abgedeckten Areals liegen. Die tatsächliche Anzahl an Dreiecken und Texturen übersteigt folglich die Zahl der theoretisch benötigten. Die folgenden Abbildungen sollen diese Problematik der Abdeckung veranschaulichen.

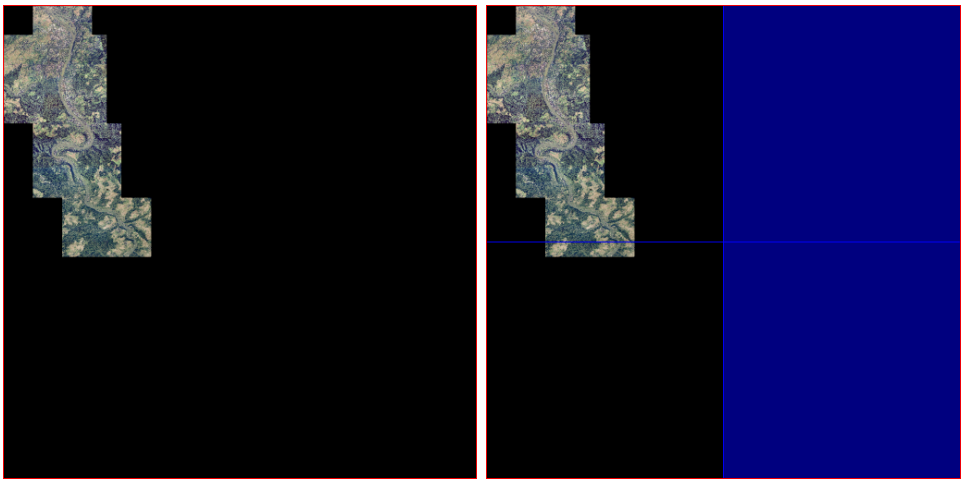

Abbildung 6: Texturabdeckung auf Ebene 0 und 1

Die Grafiken auf Abbildung 6 zeigen den Wurzelknoten auf Ebene 0 (links) und die vier Kindknoten der ersten Ebene (rechts) des Quadtrees. Der *Chunk* auf Level 0 (roter Rahmen) deckt das gesamte Terrain ab. Die schwarzen Flächen sind in der Ausgabe nicht sichtbar, weil den dort befindlichen Eckpunkten schwarze Texel zugeordnet werden. Sie erzeugen lediglich unnötige Texturdaten und überflüssige Dreiecke. Die rechte Illustration zeigt die vier Teilblöcke auf der nächsten Ebene (blauer Rahmen). Für die beiden Knoten, die den blauen Bereich repräsentieren, müssen weder *Chunks* erstellt werden, noch müssen sie in jeweils vier Kindknoten unterteilt werden, da sie komplett außerhalb der Abdeckungsgrenzen liegen.

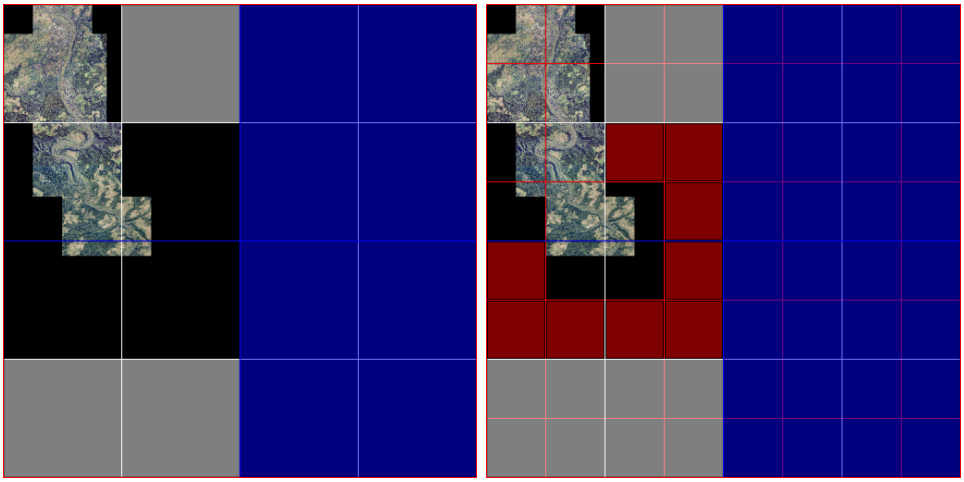

Abbildung 7: Texturabdeckung auf Ebene 2 und 3

Abbildung 7 gibt die nächsten beiden Ebenen wieder. Auf Ebene 2 (links) liegen acht Quadtree-Knoten (weiße Rahmen), von denen wiederum drei (aus den weißen Bereichen) komplett aussortiert werden können. Diese fünf Knoten haben nun 20 Kinder auf Ebene 3, die mit roten Rahmen markiert (im rechten Bild) sind. Die neun Blöcke im roten Bereich können unbeachtet bleiben. Die überflüssigen schwarzen Flächen nehmen von Stufe zu Stufe ab.

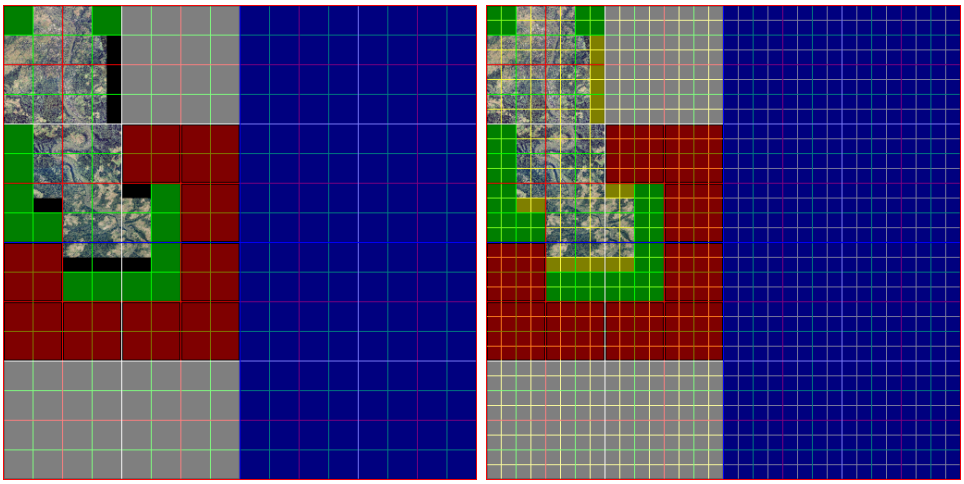

Abbildung 8: Texturabdeckung auf Ebene 4 und 5

Auf Abbildung 8 sind Ebene 4 (grün) und 5 (gelb) zu sehen. Die 44 Blöcke der vierten Ebene können auf 30 reduziert werden. Nach der Ausmusterung weiterer 16 Knoten auf Ebene 5 fallen die Grenzen der verbleibenden 104 *Chunks* mit den

Grenzen der DOP im Originalformat zusammen und so ist sie — und alle darunter liegenden Ebenen — frei von überflüssigen Terrainblöcken.

3.2 Aufbau der Chunks

3.2.1 Dreiecksnetz

Ein *Chunk* setzt sich aus $25 * 25$ Vertices, für jeden Höhenwert einen, zusammen. Dazu kommen eine Zeile am südlichen Rand und eine Spalte am östlichen Rand für den Anschluß an die Nachbarblöcke. Weitere vier Reihen, an jeder Kante eine, dienen der Konstruktion der *Skirts*. In der Summe ergeben sich also für jeden *Chunk* $28 * 28$ Eckpunkte, bestehend aus je drei *short integer* - Werten für die drei Koordinaten. Außerdem werden für jeden Vertex zwei Texturkoordinaten $(2 * 28 * 28)$ vom Datentyp *float* benötigt.

Die Triangulierung einer quadratischen Oberfläche erzeugt in der Regel ein *Mesh* (Dreiecksnetz) mit etwa doppelt so vielen Dreiecken wie die Anzahl der zugrunde liegenden Eckpunkte. Hier kommen $2 * 27 * 27$ Dreiecke mit $3 * 2 * 27 * 27$ Vertices für die Eckpunkte zum Einsatz.

p1	p2	p3	...	p25	p26	p27	p28
p29	**p30**	**p31**	**...**	**p53**	**p54**	*p55*	p56
p57	**p58**	**p59**	**...**	**p81**	**p82**	*p83*	p84
...	**...**	**...**	**...**	**...**	**...**	...	...
p672	**p673**	**p674**	**...**	**p697**	**p698**	*p699*	p700
p701	**p702**	**p703**	**...**	**p725**	**p726**	*p727*	p728
p729	*p730*	*p731*	...	*p744*	*p745*	*p755*	p756
p756	p757	p758	...	p782	p783	p783	p784

Diese Tabelle gibt schematisch die Anordnung der Punkte eines *Chunks* wieder. **p30** symbolisiert, dass dem Punkt mit der willkürlich gewählten laufenden Nummer 30 (die lediglich der Adressierung in den folgenden Ausführungen dient) ein regulärer Stützpunkt des DGM mit 3D-Koordinaten und Textur-Koordinaten zugeordnet ist. Ein Punkt, der in der Form *p727* gesetzt ist, hat die Anpassung an den angrenzenden Block als Existenzgrundlage. Der assoziierte Vertex und das dazugehörige Texel identifizieren die Daten des Schwesterpunktes im Nachbar-*Chunk*. p1 schließlich steht für einen Punkt des umrahmenden *Skirts*. Vertex- und Textur-Daten des *Skirts* sind abgesehen vom Z-Wert exakt die gleichen, wie die des angrenzenden Punktes im Inneren des *Chunks*.

3.2.2 Datenarrays

In dieser Implementierung kamen *Indexed Triangle Lists* zum Zeichnen des Dreiecksnetzes der *Chunks* zum Einsatz. Für diese Methode werden mehrere Tabellen benötigt. Eine Tabelle (*vertex array*) enthält alle 3D-Koordinaten der Eckpunkte in einfacher und redundanzfreier Aneinanderreihung der Form

$$vertex\ array = [p1_x, p1_y, p1_z, p2_x, \ldots, p784_z]$$

Dabei steht $p1_x$ für die X-Koordinate des ersten Vertex. Eine weitere Tabelle (*texture coord array*) speichert analog die Texturkoordinaten ($p1_u$ bedeutet: U-Koordinate des zum ersten Vertex passenden Texels):

$$texture\ coord\ array = [p1_u, p1_v, p2_u, \ldots, p784_v]$$

Damit OpenGL weiß, in welcher Reihenfolge die Eckpunkte auszugeben sind, enthält die dritte Liste (*index array*) für jedes Dreieck die Indizes seiner drei Eckpunkte in der Eckpunktliste:

$$index\ array = [\Delta 1_{v1}, \Delta 1_{v2}, \Delta 1_{v3}, \Delta 2_{v1}, \ldots, \Delta 729_{v3}]$$

$\Delta 1_{v1}$ verweist dabei auf den ersten Vertex des ersten Dreiecks.

Da alle *Chunks*, projiziert auf die durch die X- und Y-Achse aufgespannte Ebene, exakt gleich aufgebaut sind, können für alle *Chunks* auch die gleiche Anordnung der Dreiecke und die gleichen Texturkoordinaten eingesetzt werden. Der *index array* und der *texture coord array* werden bei der Initialisierung des Rendering-Algorithmus' konstruiert und einmalig abgespeichert. Lediglich die Arrays mit den Eckpunkten unterscheiden sich in den 3D-Koordinaten und müssen daher für jeden *Chunk* individuell zusammengesetzt und abgelegt werden.

3.3 Aufbereitung der Daten

3.3.1 Höhendaten

Da die durchschnittliche Höhengenauigkeit des Digitalen Geländemodells laut des Landesamtes für Vermessung und Geobasisinformation Rheinland-Pfalz zwischen $0,5m$ und $5m$ — im Flachland und leicht bewegten Gelände $\pm 0,5m$, im stärker bewegten Gelände $\pm 1m$, im dicht bewaldeten Gelände $\pm 5m$ — beträgt, wurden die in Millimetern angegeben Höhenwerte in einem ersten Schritt auf Zentimeter gerundet. Die Höhenangaben lagen nun in einem Intervall zwischen 5.542 und $53.371cm$, was die speichereffiziente Verwendung des Datentyps *unsigned short integer*[12] mit einem Wertebereich von 0 bis 65.535 ermöglichte.

Zur Verbesserung der Performanz beim Einlesen von der Festplatte erfolgte eine zeilenweise aneinandergereihte Abspeicherung der Binärwerte im Rohdatenformat. Der Speicherbedarf konnte von den rund 260 Megabyte der gelieferten ASCII-Datei auf $2001 * 3401 * 2$ Byte $\approx$ 13 Megabyte verringert werden.

Diese Form der Repräsentation von Höhendaten entspricht dem gebräuchlichen Konzept der so genannten *Heightmap*. Mit dem Begriff Heightmap wird die Abspeicherung äquidistanter Höhenwerte in einem zweidimensionalen Array bezeichnet. Als Veranschaulichung dient die Interpretation der Daten als digitales Graustufenbild (siehe Abbildung 9), das die Landschaft senkrecht von oben abbildet, wobei die Grauwerte die Höhenabstufungen identifizieren.

Vorteile dieses Modells der Datenverwaltung sind der geringe Speicherbedarf, die leichte Implementierung und die intuitive Erfassbarkeit für den Menschen. Der

[12] 2 Byte

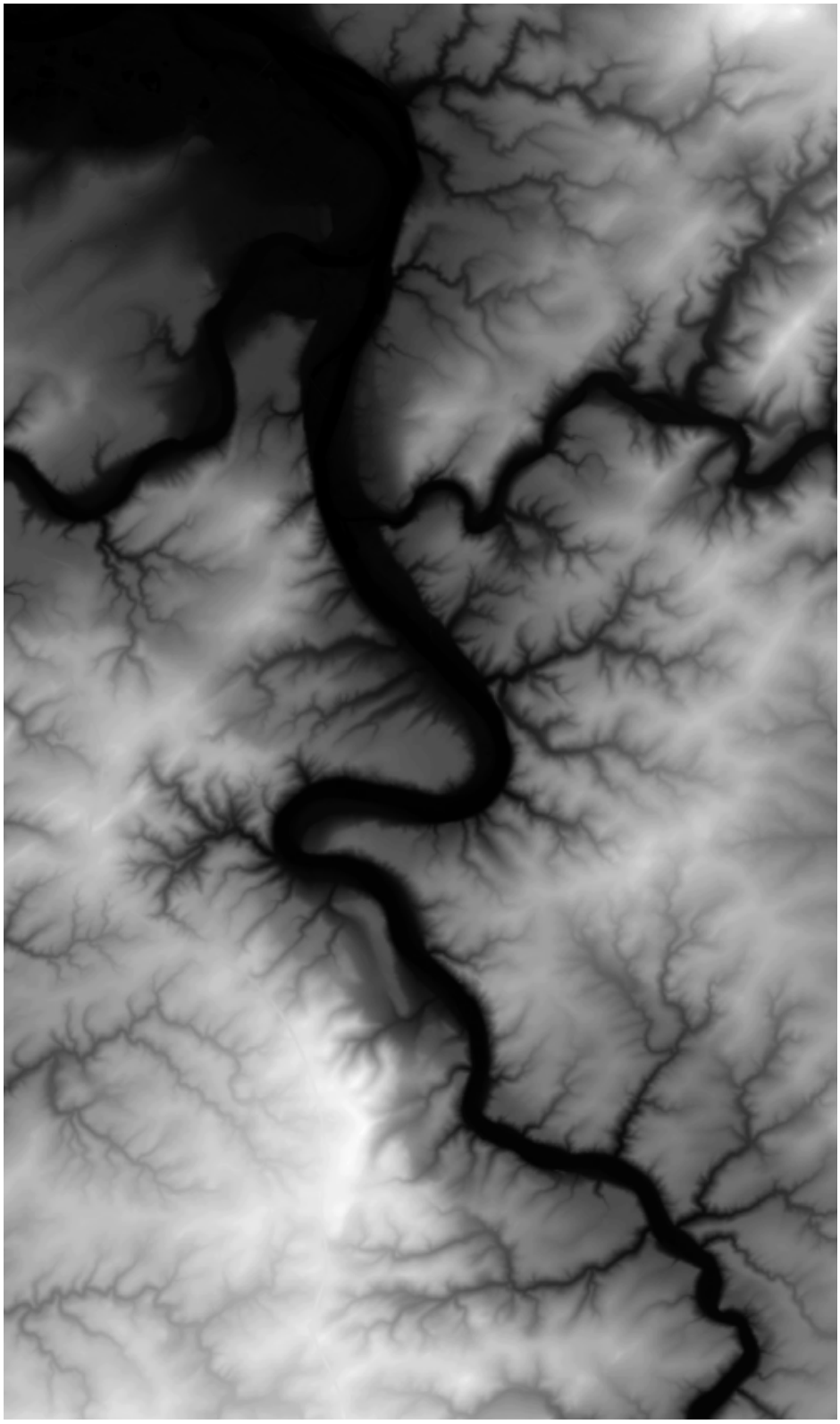

Abbildung 9: Heightmap als Graustufenbild

Nachteil, dass nur ein Höhenwert pro Lagekoordinate vermerkt werden darf, kann auf Grund der Beschaffenheit des vorliegenden Datenbestandes außer Acht gelassen werden, da keine Überhänge und Höhlen berücksichtigt wurden.

3.3.2 Texturdaten

Die Auflösung der *Chunks* bildet die Grundlage für die Zergliederung der Digitalen Orthophotographien in handhabbare Texturen. Die Festlegung auf 25 x 25 Höhenwerten entspricht 1000 x 1000 Bildpunkten pro *Chunk*. Zur Vermeidung von Nähten zwischen benachbarten Texturen dient eine Überlappung von einem Pixel Breite an der südlichen und östlichen Kante und so fallen 1001 x 1001 Pixel Nutzdaten an. Zur Erzeugung von OpenGL-kompatiblen Texturen mit einer Seitenlänge von 2^n werden die genannten beiden Kanten um eine 23 Pixel breite schwarze Spalte bzw. Zeile erweitert. Die effektive Texturauflösung beträgt schließlich 1024 x 1024.

Zur Aufbereitung der Texturen mussten zunächst die 104 mit dem Wavelett-Verfahren komprimierten Digitalen Orthophotographien entpackt werden. Dann konnten aus den 8.000 x 8.000 Pixel bzw. 2.000 x 2.000 Meter (Blattschnitt der Deutschen Grundkarte) abdeckenden Rohformatbildern die gewünschten Texturen in einer 1000 x 1000 Bildpunkte feinen Auflösung gewonnen werden. Diese 6.656 Texturen repräsentieren das gesamte Areal in der maximalen Auflösung auf der untersten Ebene des Quadtrees.

Zur Generierung der nächst gröberen Detailstufe wurden durch Auslassen jedes zweiten Pixels 500 x 500 Punkte große Bilder gewonnen. Je vier davon ergaben eine der insgesamt 1664 (= 6.656/4) wiederum 1000^2 Bildpunkte umfassenden Texturen für die zweitniedrigste Ebene. Diese Rekursion brach mit Erreichen des Wurzelknotens, in dem das gesamte Terrain durch eine Textur wiedergegeben ist, ab.

Anschließend musste jeder Textur für die Überlappung an den Kanten eine Zeile und eine Spalte der benachbarten Textur angehängt werden. Abgeschlossen war die Erstellung der Texturen durch das Auffüllen mit schwarzen Füllpixeln zu 1024 x 1024 Bildpunkte abmessenden Texturen. Wie die aufbereitete Heightmap wurden auch die Texturen als einfache Folge von Binärdaten im RGB-Format abgespeichert.

4 Implementierung

Die Umsetzung des Programms erfolgte auf dem Betriebssystem *Microsoft Windows XP* unter Verwendung der Programmiersprache *C++*. Als integrierte Entwicklungsumgebung (IDE[13]) kam *Microsoft Visual Studio .NET* zum Einsatz. Die Wahl der Programmierschnittstelle (API[14]) zur Grafikhardware fiel zu Gunsten der *Open Graphics Library*(OpenGL) aus. Darauf aufsetzend wurde auf Grafikfunktionen höherer Ebene, angeboten von der *OpenGL Utility Library* (GLU) und dem *OpenGL Utility Toolkit* (GLUT), zurückgegriffen. Dem Systementwurf liegen objektorientierte Gesichtspunkte zugrunde.

4.1 Systemübersicht

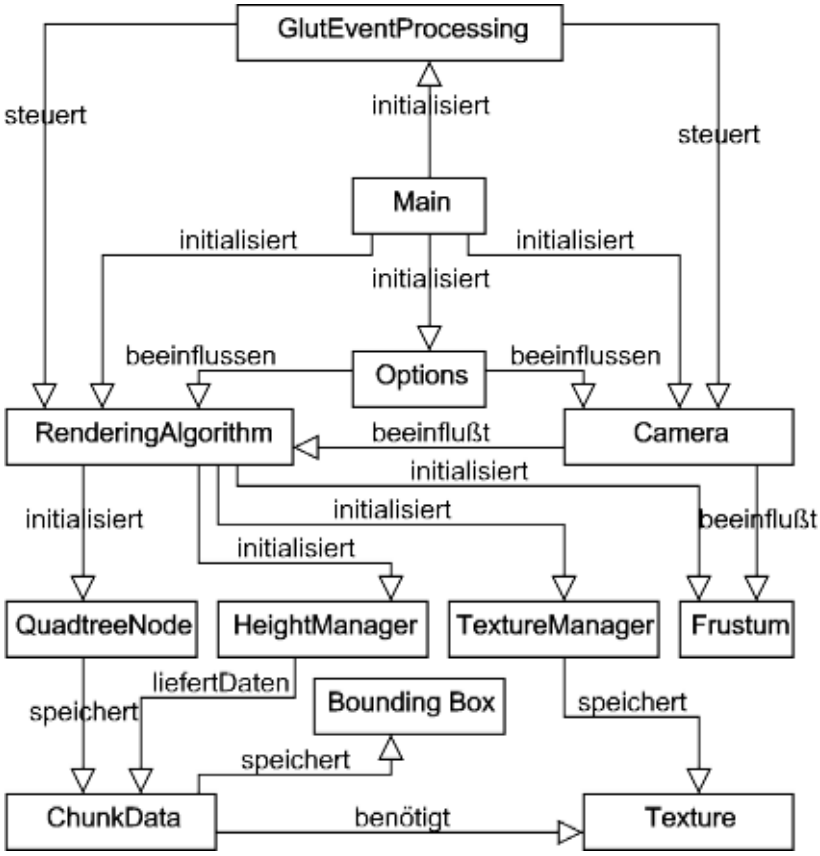

Abbildung 10: Klassenübersicht

Abbildung 10 bietet mit Hilfe eines Klassendiagramms einen Einblick in die Struktur des Programms.

[13]engl.: Integrated Development Environment
[14]engl.: Application Programming Interface

4.2 Klassen

Main

Bei Main handelt es sich um keine Klasse, sondern um eine Funktion, genauer um die von C++ geforderte Hauptfunktion des ganzen Programms. Die Funktion Main wird beim Start des Programms aufgerufen und initialisiert GLUT, OpenGL, den Rendering-Algorithmus und die Camera. Außerdem werden die *GLUT Callback*-Funktionen registriert und die *GLUT Event Processing*-Schleife betreten. Die Parameter für alle diese Aufgaben entnimmt die Hauptfunktion der Klasse Options.

Im Zuge der Initialisierung von GLUT wird ein *double buffered* RGB-Anzeigemodus mit Tiefenpuffer festgelegt. Nach Definition der initialen Fenstergröße und -position erfolgt das Öffnen des Hauptausgabefensters. Initialisieren von OpenGL meint die Auswahl der unterschiedlichen verwendeten Funktionalitäten wie *depth buffering*, *2d texturing*, *smooth shading* und *polygon backface culling* sowie die Spezifikation der dafür benötigten Parameter.

GlutEventProcessing

Mit GlutEventProcessing ist auch keine Klasse gemeint, sondern die Endlosschleife von GLUT zur Ereignisverwaltung. Während dieser Schleife arbeitet das Programm alle Events ab, für die vorab Callback Methoden registriert wurden. Im Einzelnen sind dies Funktionen zur Behandlung von Mausbewegungen (*motion function*), der Betätigung von Maus- (*mouse function*) und Keyboardtasten (*keyboard function*) für die Steuerung der Kamera sowie die Anzeige- (*display function*) bzw. Leerlaufroutine (*idle function*).

Options

Die Klasse Options hilft dabei, globale Variablen zu vermeiden, indem sie sämtliche Startwerte — wie Kameraposition und Sichtrichtung — und Konstanten — wie Fenstername oder Parameter für die Perspektive — verwaltet. Um systemweiten Zugriff zu ermöglichen und sicherzustellen, dass nur ein Objekt dieser Klasse erzeugt wird, wurde auf das Entwurfsmuster des so genannten *Einzelstücks*[15] [16] zurückgegriffen.

Camera

Die Kamera stellt dem Benutzer eine Möglichkeit zur interaktiven Navigation durch das Modell zur Verfügung. Da sich die Funktionalität der Kamerasteuerung auf das Nötigste wie Bewegung und Drehung mit Hilfe von Kugelkoordinaten beschränkt, wird von einer ausführlichen Beschreibung abgesehen. Die Klasse vereinfacht außerdem die Beschreibung eines zentralperspektivischen Betrachtungssichtfeldes durch Konstruktion der OpenGL-Projektionsmatrix via *gluPerspective* sowie

[15] Singleton

die Definition des Kamerakoordinatensystems mittels *gluLookAt*.

RenderingAlgorithm

Diese Klasse enthält die Implementierung der für das Rendering benötigten Funktionen nach Muster des *Chunked Level of Detail*-Algorithmus. Sie übernimmt die Initialisierung und Speicherung des *HeightManagers*, *TextureManagers*, *Frustums* und des *Quadtrees* sowie des *Index-Arrays* und *Texturkoordinaten-Arrays*. Zur Laufzeit erledigt die Klasse die Auswertung und die Ausgabe des Quadtrees.

QuadtreeNode

Die Datenstruktur des Quadtrees wurde durch die Implementierung einer Klasse *QuadtreeNode* zur Verwaltung der Knoten des Baumes realisiert. Jeder Knoten stellt unter der Verwendung eines Klassen-Templates die Speicherung beliebiger Knoten-Daten (*NodeData*) zur Verfügung. Des Weiteren können vier Pointer auf die vier Kindknoten (*ChildNode*) abgelegt werden. Der Zugriff auf die Daten und die Zeiger erfolgt bequem über Accessoren.

ChunkData

Alle für die Evaluierung und Darstellung eines *Chunks* notwendigen Daten werden in einer separaten Instanz der Klasse *ChunkData* abgelegt. Diese Klasse ersetzt den durch das Klassen-Template definierten "Platzhalter" der *QuadtreeNodes* für die Knoten-Daten. Erstellt und mit Daten gefüllt werden die Instanzen im Zuge der Generierung des Quadtrees.

HeightManager

Zur Verwaltung der Höhenwerte des Digitalen Geländemodells, die nach ihrer Aufbereitung als Heightmap vorliegen, wurde eine Klasse namens *HeightManager* implementiert. Diese Klasse dient lediglich der Unterstützung bei der Konstruktion der *ChunkData*. Daher wird der *HeightManager* zu Beginn des Vorverarbeitungsschritts des Rendering-Algorithmus' instanziert und anschließend wieder freigegeben.

TextureManager

Die Verwaltung der Texturen übernimmt die Klasse *TextureManager*[7], die auf einer Warteschlange mit zwei Enden *deque*[16][17] basiert. Diese Warteschlange enthält Zeiger auf Objekte vom Typ *Texture*.

Texture

Die Klasse Texture speichert für jede der aktuell geladenen Texturen den Dateinamen, die von OpenGL zugewiesene ID und die Bilddaten.

[16] Double-Ended QUEue

Bounding Box

Diese Klasse enthält die Größe und Position der umgebenden Quader der Terrainblöcke. Zu jedem Objekt vom Typ *ChunkData* gehört eine Instanz von *Bounding Box*.

Frustum

Die Ablage der Informationen, die zur Beschreibung des Sichtvolumens von Nöten sind, erfolgt schließlich in der Klasse *Frustum*.

4.3 Height Manager

4.3.1 Initialisierung

Bei der Initialisierung liest der Konstruktor die Heightmap und eine Textdatei mit Metadaten des Modells ein. Die Höhendaten werden gemäß ihrer linearen Abfolge von Binärdaten in einem eindimensionalen Array abgelegt. Die Metadatei enthält konstante Informationen, wie die *Chunk*-Auflösung, die Textur-Abdeckung und die Skalierungsfaktoren.

4.3.2 Methoden

Die Klasse stellt im Wesentlichen drei Hilfsfunktionen zur Verfügung. Eine zweidimensionale Clipping-Methode dient der Ermittlung, ob ein *Chunk* innerhalb der Textur-Abdeckung liegt und liefert die abgeschnittenen Grenzen für die Bounding Box zurück. Das Auslesen der Höhenwerte erfolgt über einen erweiterten Accessor der sicherstellt, dass keine Werte, die sich außerhalb der Höhenwert-Abdeckung befinden, abgefragt werden und ein Mapping der *unsigned shorts* in den von OpenGL unterstützten Datentyp *signed short* vornimmt.

Die dritte Funktion ermittelt den maximalen geometrischen Fehler eines Terrainblocks in Abhängigkeit von seiner Detailstufe, genauer gesagt vom Abstand *delta* zweier Höhenwerte. Der Algorithmus läuft in zwei verschachtelten Schleifen über alle Höhenwerte der Originaldaten. Für jeden Höhenwert wird seine Abweichung zu dem linear interpolierten Punkt der getesteten Detailstufe bestimmt. Wenn die Schleifen terminieren steht fest, wie groß der maximale Wert der Abweichungen ist.

Abbildung 11 versucht die Interpolation exemplarisch auf der zweiten Vereinfachungsstufe zu verdeutlichen. Die schwarzen Punkte repräsentieren die Originaldaten auf Blattebene des Quadtrees und die roten Punkte das nach der zweiten Reduktion verbleibende Raster. Zunächst wird der tatsächliche Höhenwert *realHeight* an der Position (x, y) ausgelesen. Nun wird der nächste Rasterpunkt (rx, ry) links oberhalb von (x, y) bestimmt:

$$rx = \lfloor (x/delta) \rfloor * delta$$
$$ry = \lfloor (y/delta) \rfloor * delta$$

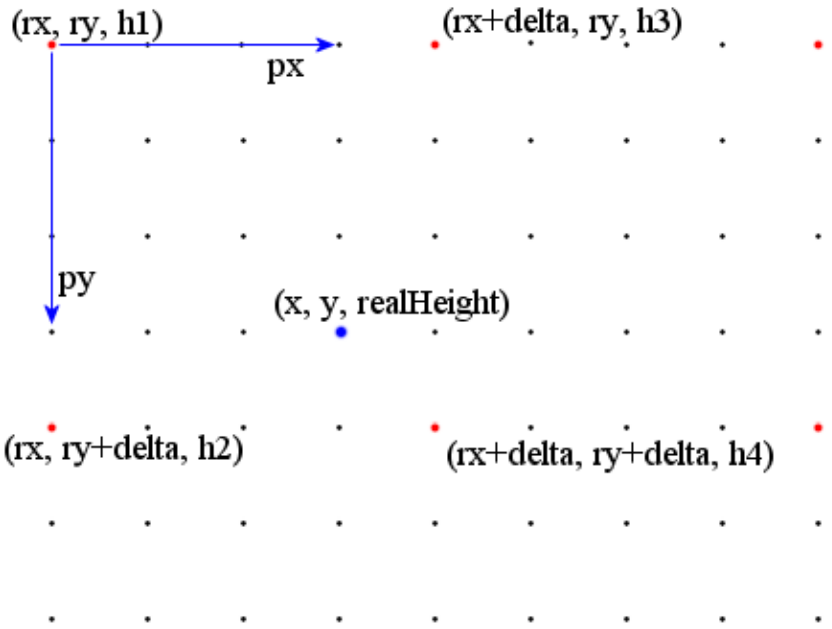

Abbildung 11: Ermittlung des maximalen geometrischen Fehlers

Davon können die drei weiteren Punkte an den verbleibenden Rasterecken abgeleitet werden. Für die Gewichtung der vier Höhenwerte ($h1$ - $h4$) an diesen umgebenden Rasterpunkten, werden bei der linearen Interpolation die Parameter px und py benötigt:

$$px = (x - rx)/delta$$

$$py = (y - ry)/delta$$

In diesem Beispiel ergäbe sich für die interpolierte Höhe ($interpHeight$) und die für Abweichung ($error$):

$$interpHeight = h4 + (h3 - h4) * (1 - px) + (h2 - h4) * (1 - py)$$

$$error = |realHeight - interpHeight|$$

Der allgemeine Ablauf für die Bestimmung des maximalen geometrischen Fehlers eines *Chunks* sei durch den folgenden Pseudo-Code skizziert:

```
maximaler Fehler = 0;

Schleife über alle y-Werte des Chunks
{
   Schleife über alle x-Werte des Chunks
   {
       realHeight = Höhenwert an (x, y);

       rx = ( x / delta ) * delta;
       ry = ( y / delta ) * delta;

       px = ( x - rx ) / delta;
       py = ( y - ry ) / delta;

       h1 = Höhenwert an (rx,       ry      );
       h2 = Höhenwert an (rx,       ry+delta);
       h3 = Höhenwert an (rx+delta, ry      );
       h4 = Höhenwert an (rx+delta, ry+delta);

       Falls py + px < 1
       {
           interpHeight = h1 +
               (h2 - h1) * py +
               (h3 - h1) * px;
       }
       sonst
       {
           interpHeight = h4 +
               (h2 - h4) * (1.0 - py) +
               (h3 - h4) * (1.0 - py);
       }

       Fehler = | realHeight - lodHeight |;

       Falls Fehler > maximaler Fehler:

           maximaler Fehler = Fehler
    }
}

gib maximalen Fehler zurück
```

4.4 Texture Manager

4.4.1 Initialisierung

Um den Rechenaufwand während des Renderings zu reduzieren, wird die *deque* des *TextureManagers* bereits bei der Initialisierung mit leeren Objekten vom Typ *Texture* als Platzhalter gefüllt und für jedes *Texture*-Objekt die folgenden Schritte durchgeführt:

1. Hauptspeicher für die Bilddaten reserviert und mit Nullen gefüllt
2. Eine *OpenGL-Textur*-ID generiert und gespeichert
3. Die Dimension der *OpenGL-Textur* festgelegt
4. Das Textur-Environment bestimmt
5. Die Textur-Verkleinerungs- und die Textur-Vergrößerungs-Funktion gesetzt
6. Das *Wrapping* ausgeschaltet
7. Eine leere schwarze *OpenGL-Textur* erstellt

4.4.2 Methoden

Zur Laufzeit beschränkt sich die Funktionalität des *TextureManager* auf das Nachladen neuer Texturen und das Freigeben nicht mehr benötigter Daten. Im Vorbereitungsschritt des Renderings werden unter Angabe des Dateinamens die Texturen angefordert. Der Manager prüft vorab, ob die Textur bereits geladen wurde und gibt in diesem Fall die entsprechende *OpenGL-Textur*-ID zurück. Andernfalls wird das letzte *Texture*-Objekt aus der *deque* angewählt und dem Objekt der Dateiname der erwünschten Textur zugewiesen. Nun werden die Bilddaten aus der Datei in den reservierten Speicher des Objektes eingelesen.

Anschließend wird die *OpenGL-Textur*, die durch ihre ID dem *Texture*-Objekt zugeordnet ist, gebunden und die ihre Texel mittels der effizienten Funktion *glTexSubImage2D* mit den neuen Daten überschrieben. Letztlich wird der Zeiger auf das Textur-Objekt vom Ende der *deque* entfernt und am Anfang wieder eingefügt. So bleiben kürzlich verwendete Texturen möglichst lange in der aktiven Warteschlange.

Außerdem stellt der *TextureManager* eine Hilfsfunktion zur Ermittlung von Textur-Dateinamen bereit.

4.5 Rendering Algorithmus

4.5.1 Initialisierung

Im Initialisierungsschritt des Algorithmus wird der *index array* durch Ausführung des folgenden Pseudo-Codes erstellt:

```
count = 0;
for( int y = 0; y < (chunk resolution+2); y++ )
{
   for( int x = 0; x < (chunk resolution+2); x++ )
   {
        // Indizes der Vertices des (x * y * 2)-ten Dreiecks
       index array[count++] = (x+0) + (y+0)*(chunk resolution+3);
       index array[count++] = (x+1) + (y+0)*(chunk resolution+3);
       index array[count++] = (x+0) + (y+1)*(chunk resolution+3);

        // Indizes der Vertices des (x * y * 2 + 1)-ten Dreiecks
       index array[count++] = (x+1) + (y+0)*(chunk resolution+3);
       index array[count++] = (x+1) + (y+1)*(chunk resolution+3);
       index array[count++] = (x+0) + (y+1)*(chunk resolution+3);
   }
}
```

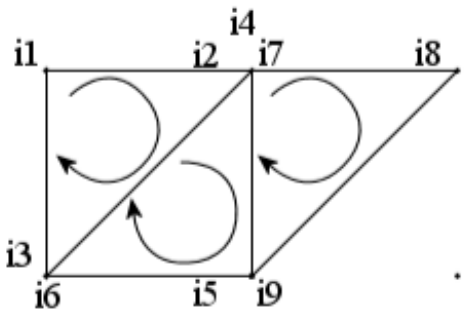

Abbildung 12: Triangulierung

Daraus ergibt sich das in Abbildung 12 wiedergegebene Triangulierungs-Schema. $i1$ steht an dem Punkt, dessen Index im Vertex-Array den ersten Eintrag im Index-Array bildet, dessen Aufbau sich wie folgt gestaltet:

$$index\ array = [i_{p1}, i_{p2}, i_{p29}, i_{p2}, i_{p30}, i_{p29}, i_{p2}, i_{p3}, i_{p30}, \ldots, i_{p756}, i_{p784}, i_{p783}]$$

Dabei meint i_{p1} den Index des zum ersten Punkt gehörenden Vertex im Vertex-Array, bzw. gleichbedeutend den Index des zum ersten Punkt gehörenden Texels im Texturkoordinaten-Array.

Da auch der Texturkoordinaten-Array für alle *Chunks* identisch ist, wird er bei der Initialisierung des Rendering-Algorithmus erstellt und wie der Index-Array als

Member-Variable abgespeichert. Bei der Konstruktion dieses Arrays wird in zwei verschachtelten Schleifen über alle Punkte des *Chunks* gelaufen und die entsprechenden U- und V-Koordinaten der Texel aneinandergereiht. Abgeschlossen wird der Vorgang durch Einbeziehung der zu den *Skirts* gehörenden Texel:

```
count = 0;

// Texturkoordinaten des Skirts der nördlichen Kante
texture coord array[count++] = 0;
texture coord array[count++] = 0;

for( int x = 0; x < (chunk resolution+1); x++ )
{
   texture coord array[count++] = x * scale;
   texture coord array[count++] = 0;
}

texture coord array[count++] = chunk resolution * scale;
texture coord array[count++] = 0;

for( int y = 0; y < (chunk resolution+1); y++ )
{
   // Texturkoordinaten des y. Punktes des westlichen Skirts
   texture coord array[count++] = 0;
   texture coord array[count++] = y * scale;

   for( int x = 0; x < (chunk resolution+1); x++ )
   {
        // Texturkoordinaten für die Vertices des Chunks
       texture coord array[count++] = x * scale;
       texture coord array[count++] = y * scale;
   }

   // Texturkoordinaten des y. Punktes des östlichen Skirts
   texture coord array[count++] = chunk resolution * scale;
   texture coord array[count++] = y * scale;
}

// Texturkoordinaten des Skirts der südlichen Kante
texture coord array[count++] = 0;
texture coord array[count++] = chunk resolution * scale;

for( int x = 0; x < (chunk resolution+1); x++ )
{
   texture coord array[count++] = x * scale;
   texture coord array[count++] = chunk resolution * scale;
}

texture coord array[count++] = chunk resolution * scale;
texture coord array[count++] = chunk resolution * scale;
```

Der Skalierungsfaktor *scale* bildet dabei die Werte von 0 bis *chunk resolution* in den Wertebereich der Texturkoordinaten von 0 bis 1001 ab.

4.5.2 Generierung des Quadtrees

Die Generierung des Quadtrees wird rekursiv im Vorverarbeitungsschritt unter Verwendung des Pseudo-Codes durchgeführt:

```
Erstelle Quadtree ( X, Y, Size )
{
    Erstelle einen neuen QuadtreeNode

    Generiere ChunkData ( X, Y, Size )

    QuadtreeNode.NodeData = ChunkData

    Size = Size / 2

    Falls die unterste Baumebene noch nicht erreicht ist:
    {
        // Generiere Nord-Westlichen Unterbaum
        QuadtreeNode.ChildNode1 =
            Erstelle Quadtree ( X,      Y,      Size )
        // Generiere Nord-Östlichen Unterbaum
        QuadtreeNode.ChildNode2 =
            Erstelle Quadtree ( X+Size, Y,      Size )
        // Generiere Süd-Westlichen Unterbaum
        QuadtreeNode.ChildNode3 =
            Erstelle Quadtree ( X,      Y+Size, Size )
        // Generiere Süd-Östlichen Unterbaum
        QuadtreeNode.ChildNode4 =
            Erstelle Quadtree ( X+Size, Y+Size, Size )
    }

    Gebe Zeiger auf QuadtreeNode zurück
}
```

In der Instanz der Klasse *RenderingAlgorithm* muss dann lediglich der Zeiger auf den Wurzelknoten abgelegt werden, der unter Anwendung der skizzierten Methode mit den Parametern $x = 0$, $y = 0$ und $Size = terrain\ resolution$ erzeugt wird.

4.5.3 Generierung der Chunks

Jede Instanz der Klasse *ChunkData* enthält die folgenden Informationen über den entsprechenden *Chunk*:

- ⋆ Das Array mit den 3D-Koordinaten der Vertices: *vertex array*
- ⋆ Den maximalen geometrischen Fehler: *error*
- ⋆ Die umgebende Axis-Aligned Bounding Box: *bounding box*
- ⋆ Den Namen der Datei der dazugehörigen Textur: *texture name*
- ⋆ Eine (variable) ID der zugeordneten Textur: *texture id*
- ⋆ Eine (variable) Statusflag für das Rendering: *node state*

Die Generierung der *Chunk*-Daten verläuft nach folgendem Muster:

1. Zunächst wird geprüft, ob der *Chunk* innerhalb des texturierten Bereiches liegt. Ansonsten wird seine Statusflag auf `OUT_OF_TERRAIN_BOUNDS` gesetzt. Die Klasse *HeightManager* stellt die Methode für diese Prüfung bereit.
2. Der maximale geometrische Fehler wird ebenfalls vom *HeightManager* berechnet.
3. Der Dateiname der dazugehörigen Textur wird aus der einzigartigen Kombination des (x,y)-Tupels der nordwestlichen Ecke des *Chunks* und seiner Stufe im Quadtree durch eine Methode der Klasse *TextureManager* geliefert.
4. Das Vertex-Array wird individuell und analog zum Texturkoordinaten-Array erstellt.
5. Die Bounding Box wird initialisiert.
6. Die Textur-ID und die Statusflag werden zur Laufzeit gesetzt.

4.5.4 Vorbereitung des Quadtrees

Aufgrund der interaktiven Kamerasteuerung muss der Quadtree in jedem Frame traversiert werden, um zu ermitteln, welche der *Chunks* zur Darstellung gelangen. Als erstes wird anhand der Statusflag geprüft, ob der unterhalb des aktuell untersuchten Knoten befindliche Teilbaum überhaupt innerhalb des texturierten Bereiches liegt und gegebenenfalls aussortiert werden kann. Die nächste Hürde bildet das Frustum Culling. Befindet sich der Teilbaum außerhalb des Sichtfeldes, kann er ebenfalls vom Rendering ausgeschlossen werden.

Nach dieser Dezimierung steht fest, dass der von diesem Knoten repräsentierte Teilbereich des Modells potentiell sichtbar ist. Jetzt wird unter Zuhilfenahme der bereits dargelegten Fehlermetrik geprüft, ob sich die Detailstufe des im Knoten abgespeicherten *Chunks* bereits zur Ausgabe eignet. Trifft dies zu, stellt der *TextureManager* anhand des Dateinamens der assoziierten Textur sicher, dass die Textur aktuell verfügbar ist und liefert die entsprechende *OpenGL-Textur*-ID zurück. Die Statusflag wird schließlich auf `RENDER` gesetzt. Eignet sich die Detailstufe jedoch nicht zur Ausgabe, nimmt die Flag den Status `TRAVERSE` an und die vier Kindknoten werden rekursiv derselben Prozedur unterzogen.

4.5.5 Rendering des Quadtrees

In einer zweiten Traversion des Quadtrees zur Laufzeit werden alle *Chunks* mit dem Status `RENDER` ausgegeben. Dazu muss zunächst die zum Terrainblock gehörende Textur mittels ihrer *OpenGL-Textur*-ID gebunden werden. Dann werden die Lage und das Datenformat des Arrays mit den Vertexkoordinaten des *Chunks* und des einheitlichen Arrays mit den Texturkoordinaten definiert. Durch Aufruf des Kommandos *glDrawElements* erfolgt unter Angabe des Primitiven-Typs "Dreieck" und der Position des Index-Arrays die Ausgabe des Meshes, belegt mit der entsprechenden Textur.

4.6 Bounding Volume

4.6.1 Initialisierung

Bei der Generierung des Quadtrees wird für jeden Knoten die umgebende Axis-Aligned Bounding Box des Terrainblocks bestimmt und abgespeichert. Die zur X- und Y-Achse parallelen Kanten der begrenzenden Box sind durch die Ausmaße des *Chunks* direkt vorgegeben. Die Ausdehnung in Z-Richtung macht eine Suche nach dem minimalen und dem maximalen Höhenwert innerhalb des Blocks notwendig. Abgespeichert wird der Quader in dieser Implementierung durch zwei Vektoren, seinen Mittelpunkt $\vec{center}$ und seine Ausdehnung $\vec{extent}$ in alle drei Raumrichtungen.

4.6.2 Distanzberechnung

Die *Bounding Box*-Klasse stellt eine Funktion bereit, die den Abstand des Augpunktes $\vec{eyePoint}$ zum Quader zurück liefert. Diese Berechnung erfolgt in zwei Schritten. Zunächst wird die Distanz $\vec{distance}$ zwischen dem Augpunkt und dem Mittelpunkt der Box bestimmt:

$$\vec{distance}_x = |\vec{eyePoint}_x - \vec{center}_x|$$
$$\vec{distance}_y = |\vec{eyePoint}_y - \vec{center}_y|$$
$$\vec{distance}_z = |\vec{eyePoint}_z - \vec{center}_z|$$

Die Subskriptionen, z.b. $\vec{distance}_x$, verweisen hier auf die einzelnen Komponenten der Vektoren. In einem zweiten Schritt wird diese Distanz um die Ausdehnung der Box verkürzt, negative Komponenten werden auf 0 gesetzt.

$$\vec{distance}_x = \begin{cases} \vec{distance}_x - \vec{extent}_x & \text{if } \vec{distance}_x > \vec{extent}_x \\ 0 & \text{else} \end{cases}$$

$$\vec{distance}_y = \begin{cases} \vec{distance}_y - \vec{extent}_y & \text{if } \vec{distance}_y > \vec{extent}_y \\ 0 & \text{else} \end{cases}$$

$$\vec{distance}_z = \begin{cases} \vec{distance}_z - \vec{extent}_z & \text{if } \vec{distance}_z > \vec{extent}_z \\ 0 & \text{else} \end{cases}$$

Die Länge des Vektors $\vec{distance}$ entspricht der Entfernung zwischen dem Augpunkt und der Bounding Box:

$$|\vec{distance}| = \sqrt{\vec{distance}^2}$$

4.7 View Frustum Culling

4.7.1 Initialisierung

Grundvoraussetzung für das Culling des Modells am Sichtfeld der Kamera ist eine adäquate mathematische Beschreibung des View Frustums. In dieser Implementierung des Frustum Cullings wurde auf die Hessesche Normalform der sechs Ebenen, die mit den Seiten des Pyramidenstumpfes zusammenfallen, zur Speicherung zurückgegriffen. Dazu werden für jede Ebene ihre normierte Normale $\vec{n}_0$ und ihre Distanz d zum Ursprung des Koordinatensystems benötigt, welche die folgende Gleichung für jeden beliebigen Punkt $\vec{p}$ auf der Fläche erfüllen:

$$\vec{p} * \vec{n}_0 - d = 0$$

So kann das gesamte Frustum mit einem sechs mal vier Werte umfassenden Array dieser Form erfasst werden:

$$Frustum[6][4] = \begin{pmatrix} \vec{n}_{right,x} & \vec{n}_{right,y} & \vec{n}_{right,z} & d_{right} \\ \vec{n}_{left,x} & \vec{n}_{left,y} & \vec{n}_{left,z} & d_{left} \\ \vec{n}_{bottom,x} & \vec{n}_{bottom,y} & \vec{n}_{bottom,z} & d_{bottom} \\ \vec{n}_{top,x} & \vec{n}_{top,y} & \vec{n}_{top,z} & d_{top} \\ \vec{n}_{near,x} & \vec{n}_{near,y} & \vec{n}_{near,z} & d_{near} \\ \vec{n}_{far,x} & \vec{n}_{far,y} & \vec{n}_{far,z} & d_{far} \end{pmatrix}$$

Da dem Benutzer die Möglichkeit gegeben ist, die Kamera frei durch die Welt zu steuern, muss das Frustum für jeden Frame neu berechnet werden. Um diesen Aufwand zur Laufzeit möglichst gering zu halten, werden folgende statischen Parameter bei der Instanzierung des Frustums im Vorverarbeitungsschritt abgespeichert:

- ⋆ Distanz der *near clipping plane* zum Augpunkt: $nearDistance$
- ⋆ Distanz der *far clipping plane* zum Augpunkt: $farDistance$
- ⋆ halbe Höhe der *far clipping plane*: $farHeight2$
- ⋆ halbe Breite der *far clipping plane*: $farWidth2$

Wobei sich $farHeight2$ und $farWidth2$ wie folgt aus dem vertikalen Öffnungswinkel der Kamera $vfov$ und dem Seitenverhältnis der Breite zur Höhe der Pyramidenbasis $aspectRatio$ berechnen lassen:

$$farHeight2 = tan(vfov/2) * farDistance$$

$$farWidth2 = farHeight2 * aspectRatio$$

4.7.2 Berechnung des View Frustums

Mit Hilfe dieser Konstanten und der Kameravariablen erfolgt die Ermittlung der Ebenen des Frustums. Die dynamischen Parameter sind neben dem Augpunkt des Betrachters die drei Kameraachsen zur Beschreibung der Orientierung im virtuellen Raum. Während die Position der Kamera ($eye\vec{P}oint$) und die Z-Achse ($cam\vec{e}raZ$) direkt aus der Kamerainstanz ausgelesen werden können, erfolgt die Berechnung der X-Achse ($cam\vec{e}raX$) mit dem Kreuzprodukt aus dem mit *gluLookAt* definierten up-Vektor und der Z-Achse und die Y-Achse ($cam\vec{e}raY$) ergibt sich analog aus dem Kreuzprodukt von Z- und X-Achse.

Die *far* und die *near clipping plane* lassen sich auf triviale Weise in die Hessesche Normalform überführen. Die Normale der far plane identifiziert die negative Z-Achse der Kamera und ihr Abstand zum Ursprung entspricht *farDistance*. Bei der near plane verhält sich dies analog mit der positiven Z-Achse und mit *nearDistance*.

Die rechte und linke bzw. obere und untere Ebene erfordern mehr Berechnungsaufwand. Zunächst findet die Lokalisierung des Mittelpunktes der far plane ($\vec{fc}$) statt:

$$\vec{fc} = eye\vec{P}oint + cam\vec{e}raZ * farDistance$$

Exemplarisch sei nun die Berechnung der oberen clipping plane wiedergegeben. Dazu wird ein Hilfsvektor ($a\vec{u}x$), der auf die Mitte der oberen Kante der far clipping plane zeigt, bestimmt:

$$a\vec{u}x = \vec{fc} + cam\vec{e}raY * farHeight2$$

Das Kreuzprodukt der X-Achse des Kamerakoordinatensystems mit dem Richtungsvektor zwischen Kameraposition und dem Hilfsvektor liefert die Normale der oberen clipping plane, die nach der Normierung gespeichert werden kann:

$$\vec{n_{top}} = cam\vec{e}raX \ \mathbf{x} \ (a\vec{u}x - eye\vec{P}oint)$$

$$\vec{n}_{top,0} = \frac{\vec{n_{top}}}{|\vec{n_{top}}|}$$

Der Abstand zum Ursprung d kann unter Zuhilfenahme des Skalarproduktes aus Hilfsvektor und Normalen ermittelt werden:

$$d = -a\vec{u}x * \vec{n_{top,0}}$$

Die Berechnung der verbleibenden Ebenen wird analog durchgeführt.

4.7.3 Culling der Bounding Box

Die Entscheidung, ob eine Bounding Box komplett außerhalb des Frustums liegt und somit direkt aussortiert werden kann, erfolgt durch einen extrem schnellen und dennoch gut nachvollziehbaren Algorithmus. In einer Schleife über alle Ebenen des Sichtvolumens wird ihre Lage zu der Bounding Box überprüft. Sollte die Box potentiell sichtbar sein, müssen im *worst case* sechs Tests durchgeführt werden. Liegt die Box jedoch auf der dem Betrachter abgewandten Seite einer der Ebenen, kann

die Schleife sofort abgebrochen und der entsprechende Knoten des Quadtrees mit seinen Kindern vom Render-Vorgang ausgeschlossen werden.

Zur Bestimmung auf welcher Seite der aktuell überprüften clipping plane i sich die Bounding Box befindet, wird der Mittelpunkt der Box $\vec{center}$ in die Ebenengleichung eingesetzt:

$$\vec{center} * \vec{n_{i,0}} - d = distance$$

Dies liefert nicht nur die Distanz des Boxmittelpunktes zur Ebene, sondern auch in welchem Halbraum er sich befindet. Ist $distance > 0$ liegt er in dem Halbraum, der dem Betrachter zugewandt ist, d.h. die Box kann nicht verworfen werden und muss an den verbleibenden Ebenen getestet werden. Ist $distance < 0$ muss anhand der Ausdehnung der Box in Richtung der Ebene ($extentTowardPlane$) überprüft werden, ob nicht nur der Mittelpunkt, sondern die gesamte Box außerhalb des Sichtvolumens liegt. Dazu wird die Ausdehnung der Box ($\vec{extent}$) auf die Normale der Ebene ($\vec{n}$) projiziert. Die einzelnen Komponenten der Vektoren werden durch Subskription (z.b. $\vec{extent_x}$) bezeichnet:

$$extentTowardPlane = |\vec{extent_x} * \vec{n}_x| + |\vec{extent_y} * \vec{n}_y| + |\vec{extent_z} * \vec{n}_z|$$

Ist nun der Betrag des Abstandes zwischen der Box und der Ebene größer als die Ausdehnung der Box in Richtung der Ebene $|distance| > extentTowardPlane$, so ist die Box definitiv unsichtbar und kann endgültig verworfen werden.

5 Ergebnisse

5.1 Resultat

Das Ziel, die Erstellung und Präsentation eines dreidimensionalen und texturierten Geländemodells des Mittelrheintals, konnte umgesetzt werden. Als Basis konnte ein Digitales Geländemodell mit ausgezeichneter Auflösung (Gitterweite 10 m), großer Höhengenauigkeit (durchschnittlich 50 cm) und hoher Aktualität (Erfassung zwischen 1995 und 2005) für die gesamte gewünschte Landschaft des "Zentralen Mittelrheintals" beschafft werden. Von sehr guter Qualität waren auch die zur Verfügung gestellten Digitalen Orthophotographien, die der Texturierung zugrunde gelegt werden konnten. Die Bodenauflösung (pro Bildpunkt 25 cm in der Natur), die Farbtiefe (24 Bit) und die Aktualität (Bildflüge aus dem September 2004) übertrafen alle Erwartungen.

Nach gelungener Einarbeitung konnten die Daten erfolgreich für ihre anvisierte Verwendung aufbereitet und angepasst werden. Neben der effizienten Datenstruktur wurden auch die anderen technischen Anforderungen an das System, wie Lauffähigkeit auf *Windows XP* und Berücksichtigung objekt-orientierter Gesichtspunkte, umgesetzt. Ein für die spezifischen Anforderungen geeigneter Rendering-Algorithmus konnte in Form des *Chunked Level of Detail* ausgewählt und unter Einbezug von *Paging*-Konzepten implementiert werden. Die Programmierung mit *C++* unter Einsatz von *OpenGL* ermöglichte eine interaktive *Frame Rate* bei annehmbarer Qualität.

Die Benutzungsschnittstelle erlaubt eine einfache Navigation durch das Modell mittels Maus und Tastatur. Abstriche mussten bei der Erweiterbarkeit um zusätzliche Komponenten gemacht werden. Möglichkeiten zur Anpassung und Weiterentwicklung sind durch die modulare Programmstruktur zumindest rudimentär gegeben.

5.2 Beurteilung

Im Zuge der Justierung des erlaubten maximalen *screen space errors* zur Herstellung einer guten Balance zwischen Performanz und Bildqualität, erwies sich die extreme Kluft zwischen Geometrie- und Texturauflösung als Schwachstelle des Rendering-Algorithmus'. Zwischen der Skalierung der Textur- und der Geometrie-*Level of Detail* ließ sich kein überzeugendes Gleichgewicht erzeugen. Sogar bei Tolerierung eines deutlich sichtbaren geometrischen Fehlers, nahmen die Detailstufen der *Chunks* in Abhängigkeit von der Distanz zum Betrachter zu langsam ab, um eine angemessene Reduktion der Texturauflösung bei weiter entfernten Terrainblöcken zu erzielen.

Die Kopplung der Geometrie- mit den Textur-*Level of Detail* durch Zuweisung statischer Texturen an die *Chunks*, erlaubte keine optimale Darstellungsqualität bei maximaler *Frame Rate*. Selbst ein hoher Pixelfehler von 8 resultierte bereits in bis zu 35 auszugebenden Terrainblöcken pro Frame. Dies entspricht 115 Megabyte Texturdaten, die nur eine Grafikkarte mit mindestens 128 Megabyte Grafikspeicher handhaben kann. Obwohl der Umfang der Texturdaten den Grafikspeicher bereits

als Flaschenhals des Systems identifizierte, wurden Zeitmessungen zur Beurteilung der Performanz durchgeführt.

5.3 Testsysteme

Die Testreihen für die Auswertung von Geschwindigkeit und Qualität der Ausgabe wurden an zwei Hardwaresystemen unterschiedlicher Leistungsfähigkeit durchgeführt:

	System A	System B
Prozessor	*AMD Athlon XP 1800+*	*AMD Athlon 64 3200+*
Hauptspeicher	768 MB	1 GB
Grafikkarte	*GeForce2 MX*	*GeForce 6600 GT*
Grafikspeicher	32 MB	128 MB
Festplatte	*DiamondMax 16* 80 GB	*WD Caviar WD600AB* 60GB

5.4 Performanz

Die Messungen auf System A ergaben folgende Bildwiederholfrequenzen (*Frame Rates*) in Abhängigkeit von der Anzahl der ausgegebenen Terrainblöcke:

Chunks	0-9	10-17	18	20	24	28	32
Frame Rate	*max*	*max*/2	$12-21$	$6-9$	$3,5-4$	$2,9-3,0$	$2,3-2,5$

Bis zu neun *Chunks* können mit ihren Texturen ausgegeben werden ohne die *Frame Rate* unter die Bildwiederholfrequenz des Monitors herabzusenken. Der Einbruch der Performanz bei mehr als zehn Terrainblöcken findet seine Begründung in der Größe des Grafikkartenspeichers. Die Texturen müssen sich den Speicher der Grafikhardware mit dem *screen frame buffer* des Rendering-Fensters teilen. Für den *color buffer* des Fensters werden bereits Breite x Höhe x Farbtiefe x Anzahl der Puffer (*double buffer*) $\approx 1,75$ Megabyte ($640 * 480 * 3$ Byte $*2$) benötigt und nochmals 0,6 MB für den *z-buffer*). Es verbleiben unter 30 MB für die Texturen, doch zehn Texturen benötigen bereits über 30 MB Speicher. Bis zu 17 *Chunks* werden mit einer konstanten Rate in Höhe der halben Bildwiederholfrequenz des Monitors ausgegeben. Danach fällt die Performanz stetig ab.

Auf System B konnte der Rendering-Algorithmus eine Ausgabe mit der folgenden Anzahl an Bildern pro Sekunde ermöglichen:

Chunks	0-41	42-51	52	54	56	60	64
Frame Rate	*max*	*max*/2	$14-16$	$8-9$	$6-7$	$3,5-3,8$	$2,6-2,8$

Die Analyse der Messwerte führt zu analogen Ergebnissen. Der Grafikkartenspeicher von 128 MB kann bis zu 41 Texturen aufnehmen ($41 * 3$ MB $= 123$ MB), danach halbiert sich die *Frame Rate*. Auf diesem Testsystem nahm die Performanz bei mehr als 52 Texturen konstant ab.

Um die Relation zwischen dem tolerierten *screen space errors* und der maximalen Anzahl der auszugebenden *Chunks* aufzuzeigen und gleichzeitig einen Anhaltspunkt für die Größe des benötigten Grafikkartenspeichers zu geben, sei die nachstehende Tabelle wiedergegeben:

Pixelfehler	Chunks	Texturdaten	Grafikspeicher
2	150+	450+	512
3	100+	300+	512
4	72	216	256
5	55	165	256
6	48	144	256
7	44	132	256
8	35	115	128
16	24	72	128
32	19	57	64
64	18	54	64
128	17	51	64

Bei diesen Messungen wurde das Nachladen der Texturen zur Laufzeit nicht berücksichtigt. Immer wenn eine neue Textur von der Festplatte eingelesen werden muss, kommt es zu einem kurzen Stocken in der Ausgabe. Auf Testsystem A betrug die Ladezeit pro Textur etwa $122ms$ und $134ms$ auf System B. Da aufgrund der Quadtree-Struktur in aller Regel gleich vier Texturen auf einmal angefordert werden, fällt die *Frame Rate* kurzfristig auf etwa vier Bilder pro Sekunde ab. Bei einer hohen Bewegungsgeschwindigkeit über das Terrain geht die Echtzeitausgabe verloren und es kommt zu einer andauernden Sequenz von Einzelbildern.

5.5 Bildqualität

Die Qualität der gerenderten Landschaft hängt im Wesentlichen von der Einstellung des erlaubten Pixelfehlers ab. Um einen Vergleich der Auswirkungen unterschiedlich gewählter Schwellwerte zu ermöglichen, wurden *Screenshots* von drei verschiedenen Szenen mit abweichenden topographischen Merkmalen aufgenommen.

Zur Vermittlung eines möglichst realistischen Eindrucks der Leistungsfähigkeit des Systems, erfolgte die Erstellung der Bilder auf dem langsameren Hardwaresystem A und wenn nicht anders angegeben mit der gleichen Konfiguration wie bei den Performanz-Messungen.

Abbildung 13: Deutsches Eck

Die erste Szene zeigt das Deutsche Eck aus nordwestlicher Richtung in der Vogelperspektive. Beim oberen Screenshot beträgt der erlaubte Fehler 5 Pixel, beim unteren Bild hat der *screen space error* einen Wert von 128. Insbesondere das Reiterstandbild Kaiser Wilhelms I. und die Festung Ehrenbreitstein offenbaren die sichtbare geometrische Abweichung der groben Detailstufen gegenüber den Originaldaten. Das linke Moselufer verdeutlicht die niedrigere Texturauflösung im unteren Screenshot.

Abbildung 14: Loreley

Die Aufnahme der Loreley vom westlichen Rheinufer aus gesehen wurde mit einem Kameraöffnungswinkel von 90° gemacht, um den Vergleich mit einer Originalfotografie[8] (oben) zu ermöglichen. Besonders der herausragende Schieferfelsen weist im untersten Bild (mit Pixelfehler 128) eine hohe Differenz gegenüber seiner Repräsentation im mittleren Screenshot (Pixelfehler 3) auf. Die gröbere Texturauflösung ist am gesamten gegenüberliegenden Ufer auszumachen.

Abbildung 15: Universitätsgelände

Die letzte Szene zeigt eine Luftaufnahme von dem Gelände des Campus Metternich der Universität Koblenz-Landau aus dem Norden betrachtet. Aufgrund der ebenen Beschaffenheit der hier vorliegenden Topographie sind die Abweichungen im Dreiecksnetz kaum sichtbar. Umso deutlicher wird dadurch jedoch der Unterschied zwischen den verschiedenen Texturauflösungen im oberen Bild mit Pixelfehler 3 gegenüber dem unteren Screenshot (wieder mit erlaubtem Fehler von 128).

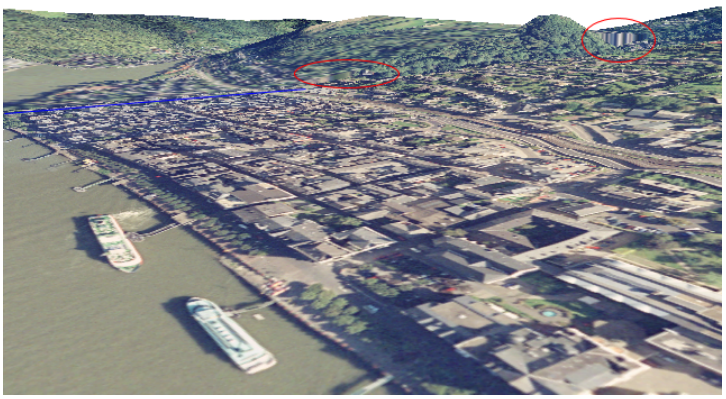

Abbildung 16: Artefakte statischer Bilder

Statische Bilder bei ruhender Kameraposition und -ausrichtung weisen unter Tolerierung extrem hoher Pixelfehler in der Hauptsache zwei Arten an optischen Artefakten auf. Zum einen wird in gebirgigen Landschaftsabschnitten durch hohe Unterschiede in den Detailstufen benachbarter Terrainblöcke die umrahmende Skirt des weiter entfernten *Chunks* mit dem gröberen *Level of Detail* wahrnehmbar. Aus demselben Grund kann es in Situationen der beschriebenen Art zu sichtbaren Übergängen zwischen zwei differierenden Texturauflösungen kommen. Auf Abbildung 16 sind durch einen *screen space error* von 128 sowohl die Skirt (hervorgehoben durch rote Ellipsen), als auch der Auflösungsunterschied an der Grenze (entlang der blauen Linie) zweier *Chunks* visualisiert worden.

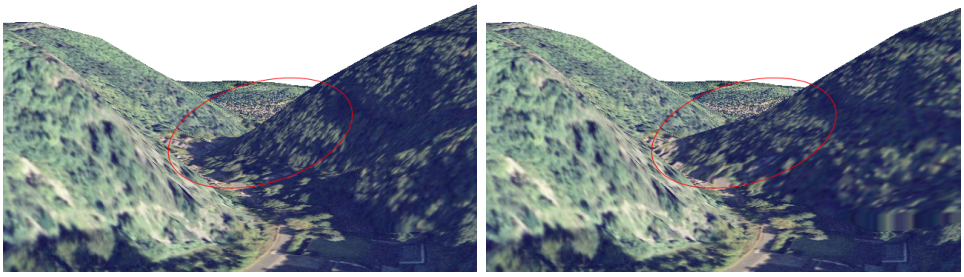

Abbildung 17: Artefakte dynamischer Bilder

Bei dynamischen Bildern durch den schnellen Flug über das Modell, kommt es zu weiteren optischen Störungen. Insbesondere hohe Schwellwerte bei der Fehler-

toleranz sind wieder dafür verantwortlich, dass durch den Wechsel zwischen zwei Detailstufen eines *Chunks* erkennbare Veränderungen im Dreiecksnetz auftreten. In Gebirgslandschaften beispielsweise verwandelt sich in Extremfällen die Silhouette des Reliefs. Eine solche Situation konnte mit den Screenshots auf Abbildung 17 festgehalten werden. Zwischen diesen Aufnahmen liegt nur ein Frame, daher fällt die unterschiedliche Kontur des Berges an der rot umkreisten Stelle direkt ins Auge.

6 Ausblick

Bei der Implementierung des Systems und der Aufbereitung der Daten wurden aus Gründen der besseren Anschaulichkeit oder der Vereinfachung unvollkommene Techniken eingesetzt. Die Beseitigung dieser Schwachstellen kann zum Teil mit überschaubarem Aufwand zu deutlichen Verbesserungen der Qualität oder der Performanz des Renderings beitragen. Ein Beispiel für eine solche Unvollkommenheit soll nicht unerwähnt bleiben.

Die Konstruktion der *Level of Detail* für die Texturen wurde mit Hilfe eines *Pixel Resize*-Verfahrens erledigt. Wie bereits beschrieben, erfolgte die Halbierung der Texturauflösung durch das Entfernen jedes zweiten Bildpunktes. Wenn an dieser Stelle ein *Resampling* per *Weighted Average* zum Einsatz kommen würde, hätte dies einen enormen Anstieg der Bildqualität besonders auf allen gröberen Detailstufen auf den höherenen Ebenen des Quadtrees zur Folge.

Neben diesen konzeptuellen Verbesserungen und funktionalen Aufwertungen — wie z.B. die Ausweitung der Kamerasteuerung — gibt es jedoch zahlreiche grundsätzlichere Möglichkeiten zur Weiterentwicklung des Programms. Einige dieser Methoden sollen in diesem Abschnitt beleuchtet werden.

6.1 Multithreading

Die wichtigste und für eine Echtzeit-Fähigkeit unentbehrliche Weiterentwicklung des Systems ist die Realisierung von softwareseitigem *Multithreading* für das *Paging*. Bei einer Datenbasis von rund 13 Megabyte für die *Heightmap* ist eine Auslagerung der gesamten *Chunks* nicht obligatorisch. Das Nachladen und Freigeben der Texturen sollte jedoch von einem separaten Thread übernommen werden.

Realisieren ließe sich dies im Vorbereitungsschritt des Quadtrees bei der Selektion der für das Rendering bestimmten *Chunks*. Vor jeder Traversierung eines Knotens wird geprüft, ob für alle Kindknoten bereits die Texturdaten vorliegen. Ist dies nicht der Fall, muss das Einlesen der benötigten Texturen von der Festplatte durch einen nebenläufigen Thread übernommen werden. Bis die Texturen zur Verfügung stehen, kann mit der im Knoten vorliegenden gröberen Detailstufe für das Rendering vorlieb genommen werden. Des Weiteren müssen Daten, die innerhalb einer bestimmten Zeitspanne nicht mehr benötigt wurden, wieder aus dem Hauptspeicher entfernt werden.

6.2 Texture Compression

Eine effiziente Möglichkeit zur Verbesserung der Performanz des *Pagings* bietet eine Kompression der Texturen. Schon mit herkömmlicher JPEG-Komprimierung[17][12] kann eine Verringerung der Datenmenge ohne auffällige Verluste auf etwa ein Zehntel[18] erreicht werden. Die Ladezeit von etwa 130 ms für eine Textur könnte so

[17] Joint Photographic Experts Group

auf rund 13 ms verkürzt werden. Allerdings müssten dann die Dauer und der Rechenaufwand für die Dekomprimierung mit einkalkuliert werden. Der Vorteil, den Speicherbedarf für alle Texturen von 26 GB auf $2,6$ GB senken zu können, macht die Texturkompression zu einer denkbaren Alternative.

Es sei in diesem Kontext noch auf eine andere Möglichkeit der Reduktion von Texturdaten hingewiesen. Die Verwendung des internen OpenGL-Formates `GL_RGB5`[4] statt des `GL_RGB` bei der Texturgenerierung aktiviert den Einsatz von Texturen mit 16 Bit pro Texel (5 Bit für rot, 6 Bit für grün, 5 Bit für blau) an Stelle der bisher eingesetzten 8 Bit Auflösung pro Farbkomponente. So ließe sich auf Kosten der Bildqualität eine Verringerung des benötigten Grafikkartenspeichers auf $\frac{2}{3}$ erzielen. Die Anzahl der gleichzeitig gerenderten *Chunks* könnte um die Hälfte erhöht und somit der Pixelfehler verringern werden.

6.3 Geomorphing

Abbildung 18: “Vertex-Popping”

Die Wahl eines hohen erlaubten Pixelfehlers kann zu optisch störenden Übergängen bei der Spaltung eines *Chunks* in seine Teilblöcke — oder umgekehrt bei der Fusion von vier Kindknoten zu einem Vaterknoten — führen. Abbildung 18 zeigt den Ursprung solcher *Vertex Popping* genannten Darstellungsfehler. Wiedergegeben ist ein zweidimensionaler vertikaler Querschnitt durch das Modell. Die beiden Punkte links stehen für zwei Eckpunkte der gröberen Detailstufe. Beim Wechsel zur feineren Detailstufe spaltet sich die verbindende Dreieckskante durch Einfügen eines weiteren Eckpunktes in der Mitte. Der gedachte Mittelpunkt der Linie “poppt” nach unten.

Besonders problematisch ist diese Form der Artefakte beim *Chunked LoD*, da sie niemals einzeln, sondern über den gesamten Terrainblock verteilt auftreten. Zur Vermeidung der lästigen *Popping*-Artefakte beim Wechsel zwischen zwei Detailstufen, bietet sich eine Technik namens *Geomorphing* an. Die Idee besteht darin, den abrupten Sprung durch eine Übergangsphase zu ersetzen, in der eine lineare Interpolation zwischen den beiden Repräsentationen stattfindet.

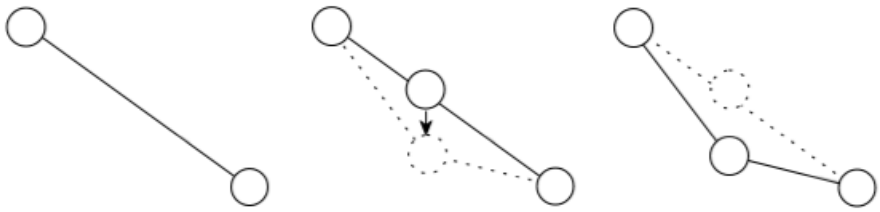

Abbildung 19: “Geomorphing”

In Abbildung 19 wurde dieser Zwischenschritt angedeutet. Die mittlere Grafik soll die Position des Eckpunkt zum Zeitpunkt der Spaltung eines *Chunks* widerspiegeln. Der Wechsel zwischen den Detailgraden ist unbemerkbar, da der Eckpunkt des Kind-*Chunks* exakt auf die Linie gesetzt wurde. Nach Beendigung des Übergangs liegt der Punkt schließlich auf der gewünschten, durch den gestrichelten Kreis angedeuteten Position. Dies kann mit einem so genannten *morph parameter* ermöglicht werden. Diesem Parameter wird bei der Teilung der Wert 1 zugewiesen, der dann langsam bis auf 0 abnimmt. Beim Verschmelzen zweier Terrainblöcke kann die gleiche Methode in umgekehrter Reihenfolge eingesetzt werden.

Diese Technik lässt sich mit wenigen Schritten beim *Chunked LoD* integrieren. Für jeden Eckpunkt eines *Chunks* muss zusätzlich zu seiner eigentlichen Höhe die interpolierte Höhe an seiner horizontalen Position innerhalb der gröberen Detailstufe abgespeichert werden. Der *morph parameter* wird einmal pro Block benötigt und kann in Abhängigkeit von der Entfernung zum Betrachter berechnet werden. So lässt sich für jeden Eckpunkt zu jeder Distanz die passende Höhe bestimmen.

6.4 Vertex Buffer Objects

Eine mächtige Erweiterung zur Verbesserung der Geschindigkeit des Renderingvorgangs stellen die auf aktuellen Grafikkarten verfügbaren *Vertex Buffer Objects* (VBO)[5][24] dar. Sie bieten eine moderne Alternative zu den traditionellen Herangehensweisen (*immediate mode* und *display lists*) von OpenGL für das Rendering von Geometriedaten. Es handelt sich hierbei um gekapselte Datenregionen (*buffer*), die direkt im hochperformanten Grafikkartenspeicher abgelegt werden können und über ihren Bezeichner zugänglich sind. Die Aktivierung dieser Buffer erfolgt über einen Einbindevorgang, der mit dem Muster zum Anwählen der Texturen vergleichbar ist. Der zeitaufwändige Transfer der Daten vom Hauptspeicher zur Grafikhardware in jedem einzelnen Frame kann so eingespart werden.

Vertex Buffer Objects benötigen stets zwei getrennte Puffer. Die *Array Buffer* enthalten die Vertex-Attribute, wie die 3D-Koordinaten der Eckpunkte, ihre Texturkoordinaten oder ihre Farbe. Die Indizes der Elemente werden in dem *Element Array Buffer* abgelegt. Durch diese Teilung ergibt sich eine interessante Verwen-

dungsmöglichkeit im Zusammenhang mit *Level of Detail*-Verfahren. Unter Beibehaltung des gleichen Datensatzes an Eckpunkt-Koordinaten, kann durch Wechsel zwischen verschiedenen Index-Tabellen bequem eine andere Detailstufe gewählt werden.

Eingesetzt werden können die *Array Buffer* beim *Chunked LoD* sowohl für die Vertex-Arrays als auch für den Texturkoordinaten-Array, während der *Element Array Buffer* für den Index-Array benutzt werden kann. Bei dieser Technik ergibt sich für dynamische Daten kein Vorteil, da diese dennoch für jeden Frame übertragen werden müssen. Im Rahmen der dieser Arbeit zugrunde liegenden statischen Heightmap, stellt diese Einschränkung jedoch keinen Nachteil dar.

Literatur

[1] Open graphics library. http://www.opengl.org/.

[2] National Aeronautics and Space Administration. http://www.nasa.gov/.

[3] Ulf Assarsson and Tomas Möller. Optimized view frustum culling algorithms. http://www.ce.chalmers.se/~uffe/vfc_bbox.pdf, 1999.

[4] NVIDIA Corporation. Nvidia opengl extension specifications. http://www.nvidia.com/dev_content/nvopenglspecs/GL_NV_pixel_data_range.txt, 2002.

[5] NVIDIA Corporation. Using vertex buffer objects. http://developer.nvidia.com/object/using_VBOs.html, 2003.

[6] Arbeitsgemeinschaft der Vermessungsverwaltungen der Länder der Bundesrepublik Deutschland. http://www.adv-online.de/.

[7] Borja Fernandez. Implementing a texture manager. http://www.codeguru.com/cpp/g-m/opengl/texturemapping/article.php/c5573/, 2003.

[8] Wikimedia Foundation. Wikipedia. http://de.wikipedia.org/.

[9] Bundesamt für Kartographie und Geodäsie in cooperation with EuroGeographics and EUREF. Information and service system for european coordinate reference systems. http://crs.bkg.bund.de/crs-eu/.

[10] Deutsches Zentrum für Luft-und Raumfahrt. http://www.dlr.de/.

[11] Landesamt für Vermessung und Geobasisinformation Rheinland-Pfalz. http://www.lvermgeo.rlp.de/indexlvermgeo.html.

[12] Joint Photographic Experts Group. http://www.jpeg.org/.

[13] National Imagery and Mapping Agency. http://www.nga.mil/.

[14] Amtliches Topographisch-Kartographisches Informationssystem. http://www.atkis.de/.

[15] Agenzia Spaziale Italiana. http://www.asi.it/.

[16] Danny Kalev. Singleton pattern. http://www.informit.com/guides/content.asp?g=cplusplus&seqNum=148&rl=1, 2006.

[17] Nate Kohl. C/c++ reference. http://www.ce.chalmers.se/~uffe/vfc_bbox.pdf, 2006.

[18] Tom Lane. Jpeg image compression faq. http://www.faqs.org/faqs/jpeg-faq/part1/.

[19] Lighthouse3d. View frustum culling tutorial. http://www.lighthouse3d.com/opengl/viewfrustum/, 2006.

[20] Paul Martz. Drawing primitives in opengl. Sample Chapter is provided courtesy of Addison Wesley Professional, 2006.

[21] Forschung und Kultur Rheinland-Pfalz Ministerium für Wissenschaft, Weiterbildung. Welterbe oberes mittelrheintal. http://www.welterbe-mittelrheintal.de/.

[22] National Geospatial-Intelligence Agency (NGA), the National Aeronautics, and Space Administration (NASA). Shuttle radar topography mission, 2000. `http://www2.jpl.nasa.gov/srtm/`.

[23] Landesvermessungsamtes Nordrhein-Westfalen. Überblick über höhensysteme in deutschland. `http://asp.lverma.nrw.de/HoeTra/HoehensystemeBRD.html`.

[24] Ian Williams of NVIDIA (SPECopc chair) and Evan Hart of ATI. Efficient rendering of geometric data using opengl vbos in specviewperf. `http://www.spec.org/gpc/opc.static/vbo_whitepaper.html`, 2005.

[25] K. Greve R. Stahl. Gis-tutorial. `http://www.giub.uni-bonn.de/gistutor/`, 1998.

[26] Ulrich Thatcher. Chunked lod, 2001. `http://tulrich.com/geekstuff/chunklod.html`.

[27] Stefan A. Voser. The collection of map projections and reference systems. `http://www.geocities.com/mapref/mapref.html`.