UNIVERSITÄT OSNABRÜCK

Institut für Informatik
AG Medieninformatik

Masterarbeit

Interaktive und direkte Visualisierung eines veränderbaren Voxel-Terrains mit OpenCL

Nils Vollmer

Februar 2014

Erstgutachter: Prof. Dr. Oliver Vornberger
Zweitgutachterin: Prof. Dr.-Ing. Elke Pulvermüller

Danksagung

Hiermit möchte ich mich bei allen Personen, die diese Arbeit ermöglicht haben, bedanken: Prof. Dr. Oliver Vornberger sowie Prof. Dr.-Ing. Elke Pulvermüller für die Bereitstellung des Themas und die Begutachtung der Arbeit und Henning Wenke, M. Sc., für die kompetente und neutrale Betreuung. Besonderer Dank gebührt meinen Eltern, ohne deren Unterstützung und Ermutigung das ganze Studium nicht möglich gewesen wäre.

Zusammenfassung

Die vorliegende Arbeit befasst sich mit der interaktiven Visualisierung eines veränderbaren Voxel-Terrains mithilfe moderner Grafikhardware. Diese Veränderungen könnten zum Beispiel von einem Benutzer manuell vorgenommen oder von einem prozeduralen Algorithmus berechnet werden. Für ein möglichst interessantes Terrain werden dreidimensionale Volumendaten als Grundlagen der Berechnungen verwendet. Um Veränderungen schnell darzustellen, arbeiten die Methoden und Algorithmen direkt auf den Volumendaten, ohne diese in eine explizite Dreiecks-Geometrie zu überführen. Die dafür verwendeten Algorithmen basieren auf Raytracing und extrahieren die implizite Oberfläche des Terrains mithilfe des Isosurface Rendering-Ansatzes. Durch die Unabhängigkeit der einzelnen Berechnungen profitieren diese Algorithmen sehr stark von der parallelen Ausrichtung moderner Grafikhardware.

Anhand einer Applikation, die die erwähnten Methoden beispielhaft umsetzt, wird die subjektive visuelle Qualität und Performanz dieser Methoden untersucht. Die Ergebnisse zeigen, dass die angestrebte Visualisierung mithilfe moderner Grafikhardware in Echtzeit möglich ist. Durch die direkte Verwendung der Daten eignen sich die verwendeten Techniken zur effizienten Visualisierung von Veränderungen.

Abstract

The current work is concerned with the interactive visualization of an alterable voxel-terrain by using modern graphics hardware. These alterations could be made manually by a user or they could be calculated by a procedural algorithm. Three-dimensional data will be used in order to create an interesting terrain. To render the alterations as quickly as possible, the algorithms are directly working on the 3D data sets, without converting them into an explicit triangle mesh. The utilised algorithms are based on raytracing, extracting the implicit surface of the terrain by using isosurface rendering. These algorithms benefit from the parallel nature of modern graphics hardware due to their independent calculations.

The subjective visual quality and the performance of these methods will be evaluated based on an exemplary application. The results reveal the possibility of an interactive terrain-rendering by using modern graphics hardware. The utilised methods are suited for the efficient visulization of alterations due to their direct data usage.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Als Forschungsgebiet der Informatik beschäftigt sich die Computergrafik mit der Visualisierung von Daten und modellierten Szenen. Während bis zu den 90er Jahren noch primär technische Anwendungen die Computergrafik dominierten, rückten seitdem immer mehr die nichttechnischen Anwendungen der Unterhaltungsindustrie (Animationsfilm und später Spiele) in den Vordergrund. Spezielle Grafikhardware beschleunigt die notwendigen Berechnung und im kommerziellen Endverbraucher-Bereich hat sich die Verwendung angenäherter Dreiecks-Geometrie für dreidimensionale Szenen durchgesetzt. Seit 2009 existieren Schnittstellen, mit denen diese Grafikhardware auch zur Berechnung allgemeiner, paralleler Algorithmen genutzt werden kann. Diese Arbeit befasst sich mit dieser allgemeinen Verwendung der Grafikhardware um mit Hilfe eines Raytracing-Ansatzes ein dreidimensionales Terrain in Echtzeit zu visualisieren. Das Terrain soll in Form von impliziten Daten, die direkt für die Berechnungen genutzt werden, vorliegen und zur Laufzeit verändert werden können.

1.1 Motivation

Die interaktive Darstellung eines Terrains ist ein Gebiet der Informatik, das von der Computergrafik aber auch von der Geoinformatik erforscht wird. Es werden verschiedene Techniken genutzt, um Datensätze von realen Landschaften statisch zu visualisieren (siehe Abbildung 1.1, links) oder um imaginäre, meist prozedural erstellte Landschaften darzustellen (siehe Abbildung 1.1, rechts). Solche prozeduralen Landschaften sind in den letzten Jahren vor allem im kommerziellen Bereich der Unterhaltungssoftware sehr populär geworden. Derartige Software basiert meist auf prozedural generiertem Terrain und bietet dem Nutzer häufig die Möglichkeit, dieses Terrain zu verändern. Anstatt von einem Benutzer in kleinen Bereichen vorgenommen zu werden, könnten dieser Veränderungen auf einem prozeduralen Algorithmus basieren, der größere Teile des Terrains beeinflusst. So könnte ein Terrain zum Beispiel in einem Spiel zur Laufzeit prozedural umgestaltet werden. In den meisten Fällen basieren derartige Veränderungen auf dreidimensionalen Datensätzen und benötigen deshalb Algorithmen, mit denen diese Daten in Echtzeit visualisiert werden können.

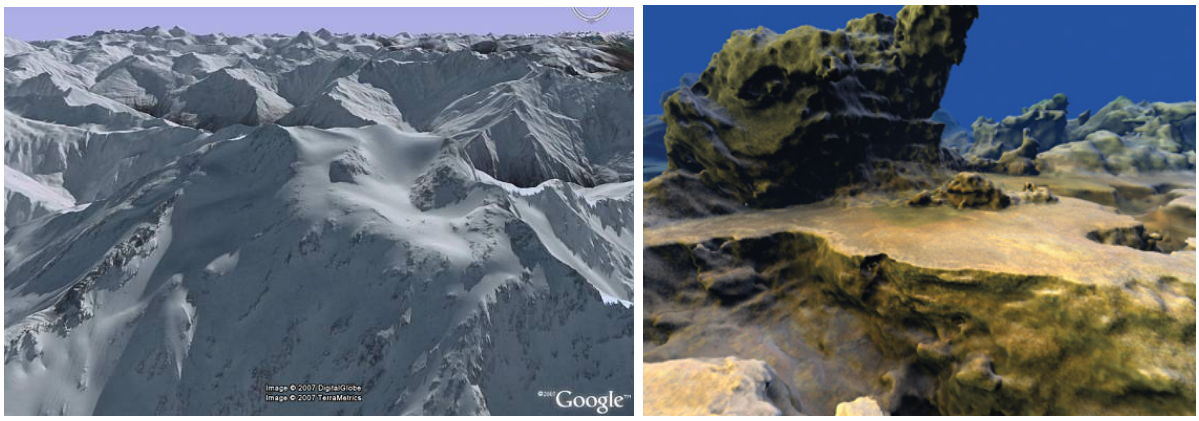

Abbildung 1.1: Terrain-Darstellung durch Google Earth[1] (links) und ein prozedural erstelltes Terrain[2] (rechts)

In der Rastergrafik wird eine darzustellende Szene durch Dreiecke angenähert, die von der Grafikhardware parallel und effizient verarbeitet werden können. Diese Methode ist eine performante und universelle Lösung um beliebige Szenen zu modellieren und visualisieren. Durch die rein lokale Verarbeitung der Dreiecke können globale Effekte, wie zum Beispiel eine realistische Ausbreitung des Lichts, nur mit sehr hohem Aufwand simuliert werden.

Alternativ zur Rastergrafik existieren Ansätze, die direkt auf impliziten, mathematisch definierten Objekten arbeiten. Die wohl bekannteste dieser Methoden ist Raytracing, bei der Strahlen durch die Szene verfolgt und auf etwaige Schnittpunkte mit den Objekten geprüft werden. Die Raytracing-Methode bietet eine intuitive Verarbeitung einer Szene, ist allerdings sehr aufwändig zu berechnen. Durch die unabhängige Verarbeitung der einzelnen Strahlen ist die Methode sehr gut für eine parallele Implementation geeignet. Seit wenigen Jahren lassen sich derartige parallele Algorithmen auf der Grafikhardware effizient ausführen.

Mit diesen Methoden und paralleler Hardware können dreidimensionale Daten in Echtzeit visualisiert werden. Durch die direkte Verarbeitung der impliziten Methoden können manuelle oder prozedural berechnete Veränderungen an den Datensätzen direkt vorgenommen und visualisiert werden.

[1] Abbildung von: `http://www.gearthblog.com/images/images807/terrain.jpg`
[2] Abbildung von: `http://http.developer.nvidia.com/GPUGems3/gpugems3_ch01.html`

1.2 Vorgehensweise und Zielsetzung

Das Ziel dieser Arbeit ist eine Untersuchung der Verwendung impliziter Methoden für die direkte Visualisierung eines veränderbaren Terrains. Dafür werden zunächst die benötigten Datengrundlagen und die daraus resultierenden Algorithmen vorgestellt. Die für die Darstellung der impliziten Terrain-Daten geeigneten Raytracing-Verfahren werden aufgezeigt und die benötigten Algorithmen erläutert. Dabei liegt der Schwerpunkt auf der Besonderheit der dreidimensionalen Datengrundlage.

Im Rahmen dieser Arbeit soll eine Applikation entstehen, die die beschriebenen Verfahren verwendet, um die veränderbaren Daten in Echtzeit zu visualisieren. Die Visualisierung soll direkt, also ohne eine Annäherung durch Dreiecks-Geometrie, erfolgen. Für die geforderte Interaktivität der Darstellung soll die Applikation die Szene mit durchschnittlich über 24 Bildern pro Sekunde (*frames per second*, FPS) wiedergeben können. Um diese Performanz zu erreichen, sollen die vorgestellten Methoden und Algorithmen für den speziellen Anwendungsfall optimiert und auf moderner Grafikhardware parallel ausgeführt werden. Basierend auf den Ergebnissen dieser Applikation soll die Machbarkeit der direkten Visualisierung der impliziten, veränderbaren Daten bewertet werden.

Kapitel 2

Stand der Technik

Während die Visualisierung einer großen, statischen Landschaft oft auf zweidimensionalen Daten basiert, werden für die in dieser Arbeit geforderten Terrain-Eigenschaften dreidimensionale Daten benötigt. Mit diesen Anforderungen sind verschiedene Techniken und Algorithmen entwickelt worden, die in diesem Kapitel vorgestellt und bewertet werden.

2.1 Heightmap

Eine Heightmap kann als einfache Technik zur Speicherung und Darstellung eines realitätsnahen Terrains angesehen werden. Als Ausgangspunkt dient ein planares 3D-Gitter von Polygonen, dessen Auflösung beliebig eingestellt werden kann. Um daraus ein Terrain zu formen, wird jeder einzelne Punkt in der Höhe translatiert. Diese Translation ist oft in einer 2D-Textur, der namensgebenden Heightmap, gespeichert und wird dort nachgeschlagen. Dafür werden die xy-Koordinaten jedes Punktes innerhalb des Meshes in Einheitskoordinaten [0,1] transformiert um anschließend mit diesen Koordinaten die Heightmap zu adressieren. Der resultierende Wert liegt ebenfalls im Einheitsintervall vor und wird mit einem maximalen Höhen-Wert multipliziert. Mit diesem Ergebnis wird der Vertex des Gitters in der Höhe translatiert (für weiterführende Algorithmen siehe [12, 21] oder Kapitel 2 in [18]).

In Pseudocode lässt sich der Algorithmus zum Beispiel wie folgt formulieren:

Algorithmus 1 : Einfacher Heightmap-Algorithmus

```
Data : heightMapImage: 2D-Textur (Bild)
Data : maxHeight: Maximale Höhe des Terrains in Weltkoordinaten
initialisiere 3D Mesh;
foreach Vertex v des 3D Gitters do
    /* berechne Einheitskoordinaten                                  */
    xUnit ← Einheitskoordinate von v.x (Breite);
    yUnit ← Einheitskoordinate von v.y (Tiefe);
    /* lese Einheitskoordinate der Höhe aus dem Bild                 */
    zUnit ← readImage(xUnit, yUnit, heightMapImage);
    /* verschiebe die Höhe des Vertices                              */
    v.z ← zUnit · maxHeight;
end
```

Die Technik besticht durch den geringen Implementationsaufwand und ihre Skalierbarkeit: Wenn die Heightmap nicht als Bild vorliegt, sondern zum Beispiel durch prozedurales Perlin-Noise generiert wird, kann ein beliebig großes Terrain erstellt werden. Durch die Nutzung der Rendering-Pipeline wird eine hohe Performanz erreicht. Abbildung 2.1 zeigt eine beispielhafte Umsetzung der Heightmap-Technik.

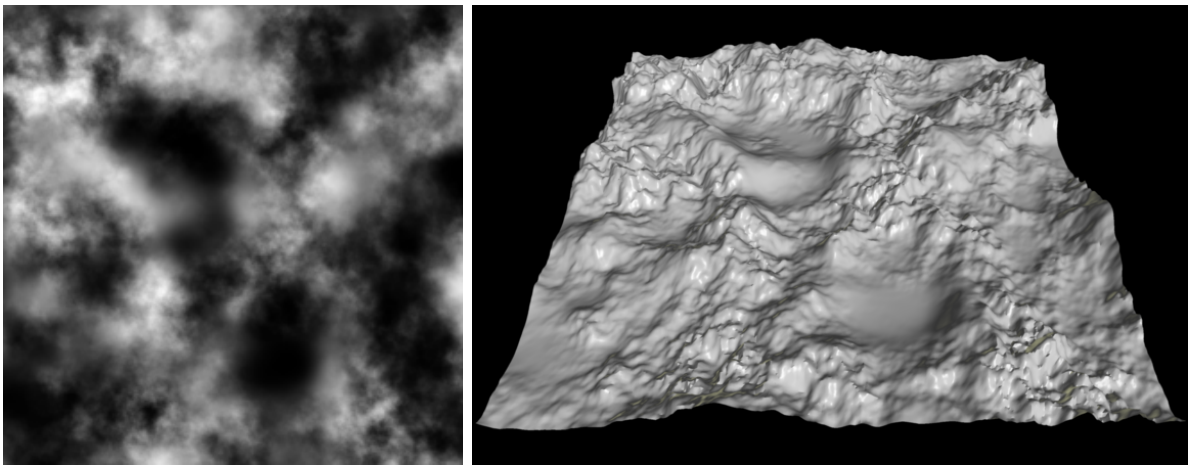

Abbildung 2.1: Eine Heightmap und ein daraus erzeugtes Terrain[3]

Der primäre Nachteil dieser Technik ist die Tatsache, dass in der Höhe nur genau eine Information vorliegt. Somit sind spezielle Terrain-Eigenschaften wie zum Beispiel Höhlen oder Überhänge mit dieser Technik nicht darstellbar. Allerdings werden solche speziellen Eigenschaften in der Industrie oft nicht benötigt und deshalb wird die Heightmap-Technik - oder auf ihr basierende Techniken - in diesem Anwendungsgebiet noch häufig verwendet. Da solche Eigenschaften aber

[3] Abbildungen von: `http://en.wikipedia.org/wiki/Heightmap`

unter anderem die Anforderungen dieser Arbeit sind, kommt diese Technik als Grundlage nicht infrage.

2.2 Voxel-basierende Techniken

Um die speziellen Terrain-Eigenschaften wie Höhlen oder Überhänge visualisieren zu können, bedarf es als Grundlage Daten, als dreidimensionale Rasterdaten vorliegen. Das Rasterdatenmodell bildet Daten durch ein Gitter aus horizontalen Zeilen und vertikalen Spalten als Menge von Zellen ab. Die Größe dieser Zellen ist einheitlich und somit wird die Größe des Datensatzes durch die Auflösung, die in jeder Dimension unterschiedlich sein kann, festgelegt. Abbildung 2.2 zeigt beispielhaft, wie ein solches Gitter aussehen könnte.

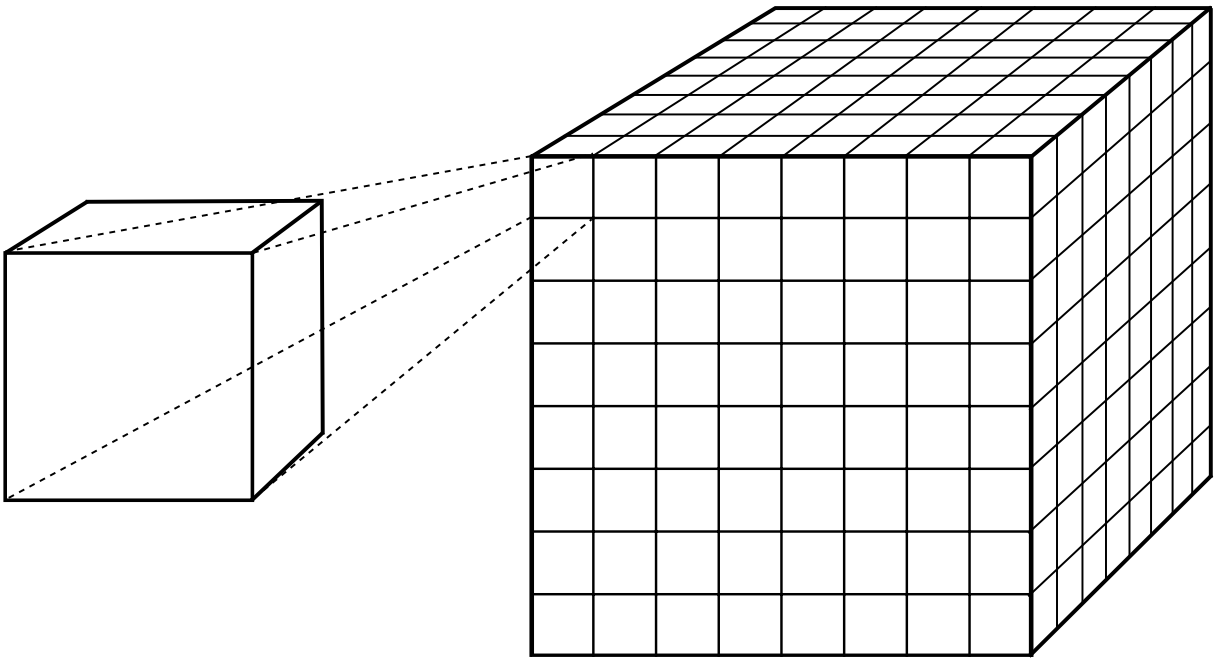

Abbildung 2.2: Schematische Darstellung eines dreidimensionalen Gitters

Den Zellen des Gitters werden unterschiedliche Werte eines Datensatzes zugeordnet. Um Geometrie zu symbolisieren werden oft Dichte-Werte eines Volumens, die im normierten Bereich entweder [-1,1] oder [0,1] vorliegen, genutzt. Diese Form der Rasterdaten wird im Folgenden als Volumendaten bezeichnet. Während die Zellen einer zweidimensionalen Rastergrafik als Pixel bezeichnet werden, wird bei einer dreidimensionalen Rastergrafik von Voxeln gesprochen. Um diese Voxel als Terrain zu visualisieren, muss vorab definiert werden, ab welchem Dichte-Wert das Volumen als Geometrie behandelt wird. Für diesen Schwellenwert bietet sich im Bereich [-1,1] zum Beispiel die Null, also der Vorzeichenwechsel, an.

Um ein in den Volumendaten enthaltenes Terrain zu visualisieren, bieten sich - je nach geforderten visuellen Eindruck - verschiedene Möglichkeiten an. Eine Technik visualisiert zum Beispiel

die Rasterdaten gemäß ihrer Datengrundlage als Würfel. Andere Techniken zielen darauf ab, einen höheren Informationsgehalt aus den Daten zu extrahieren um ein Terrain zu visualisieren, dem die Datengrundlage nicht direkt anzumerken ist. Zwei dieser Techniken werden im Folgenden behandelt.

2.3 Marching Cubes

Der von Lorensen und Cline 1987 entwickelte Marching-Cubes-Algorithmus (siehe [13]) wird verwendet, um die Volumendaten in ein Dreiecks-Mesh zu überführen. Zwischen den Datenpunkten des dreidimensionalen Gitters liegen kleine Würfel, die durch jeweils acht Dichte-Werte repräsentiert werden.

Mit diesen Daten als Grundlage "marschiert"der Algorithmus durch die Würfel und entscheidet für jeden, welche Dreiecke für ihn erzeugt werden müssen, um das durch die acht Werte repräsentierte Volumen als Geometrie darzustellen. Dabei wird jeder Punkt binär betrachtet: Entweder er liegt innerhalb des Volumens (Dichte gleich oder größer als der Schwellenwert) oder außerhalb. Somit ergeben sich $2^8 = 256$ mögliche Fälle. Diese 256 Fälle werden in einer Triangle Lookup Table (TLT) gespeichert, damit die zu erzeugenden Dreiecke schnell anhand einer 8-stelligen Bitfolge über einen einfachen Index-Zugriff auf die TLT herausgefunden werden können. Die 256 unterschiedlichen Fälle lassen sich auf tatsächliche 15 (siehe Abbildung 2.3) unterschiedliche Fälle reduzieren - alle weiteren Fälle können durch Rotation und Spiegelung gebildet werden.

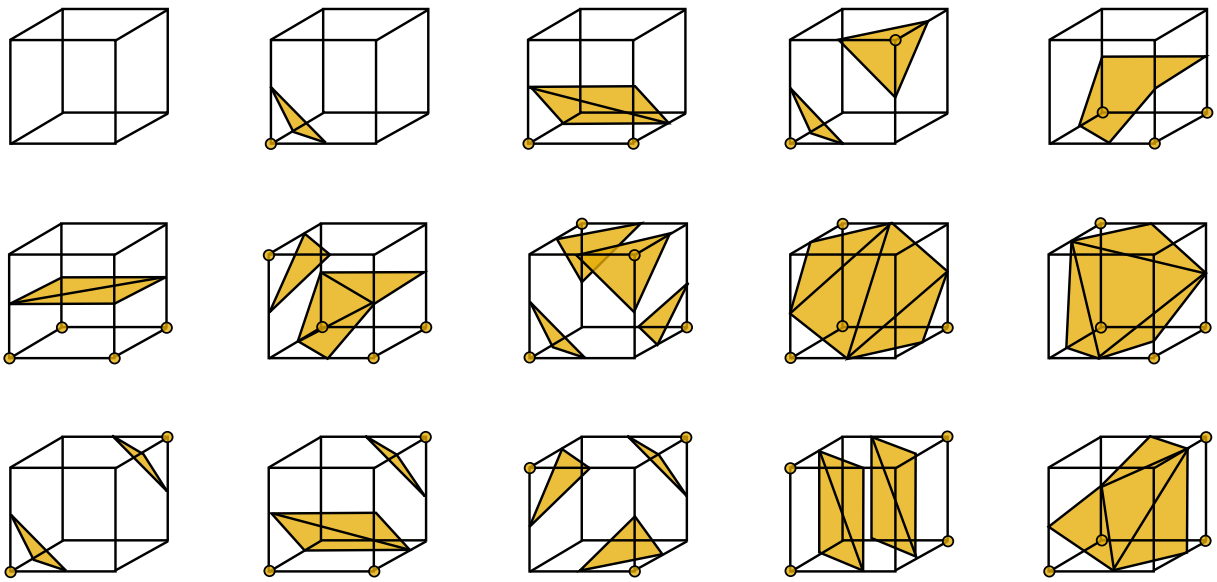

Abbildung 2.3: Die 15 unterschiedlichen Marching-Cubes Fälle

Die Eckpunkte der so erzeugten Dreiecke können im nächsten Schritt durch lineare Interpolation zwischen den Volumen-Werten verschoben werden damit zwischen den Würfeln keine Lücken im Terrain entstehen.

Mithilfe dieser Technik kann aus volumenbasierten Daten ein Terrain erstellt werden, das in der Lage ist, die geforderten Spezialfälle zu visualisieren. Ein weiterer Vorteil des Verfahrens ist die Überführung der Daten in eine Menge von Dreiecken und damit die Verwendung der optimierten Rendering-Pipeline.

Dieser Punkt ist aber auch gleichzeitig ein Nachteil: Wenn die Daten verändert werden, müssen die betroffenen Dreiecke identifiziert und gelöscht werden um anschließend die korrekten neuen Dreiecke für die geänderten Daten zu erzeugen. Des Weiteren wurde diese Technik bereits oft verwendet und ist dementsprechend weit erforscht. Aufgrund dieser Tatsachen grenzt sich diese Arbeit von der Technik ab und es wurde von ihrer Verwendung abgesehen.

2.4 Raytracing

Die grundlegende Raytracing-Technik ist in der Theorie schon mehrere Jahrzehnte alt (siehe Kapitel 3) aber erst seit kurzem ist die Hardware leistungsstark genug um die Technik für Echtzeit-Anwendungen mit komplexeren Szenen zu verwenden. Dabei variiert die verwendete Raytracing-Technik anhand der vorhandenen Daten und der gewünschten visuellen Qualität. So kann zum Beispiel auch eine Dreiecks-Szene mit Raytracing visualisieren werden - die tatsächlichen Stärken der Technik liegen aber in der Darstellung impliziter Geometrie, die nicht durch Dreiecke angenähert sondern mathematisch formuliert wird.

Raytracing gilt seit jeher als intuitive Art der Visualisierung mit der Effekte (wie z. B. Schatten oder Reflexionen) einfach umzusetzen sind. Aufgrund der Entwicklung auf dem Gebiet der Unterhaltungsindustrie wurde die Grafik-Hardware allerdings für Rasterisierung und nicht für Raytracing optimiert. Dies gilt es zu berücksichtigen, um die Performanz der beiden Techniken mit der vorhandenen Hardware zu vergleichen. Die tatsächliche Schwierigkeit bei der Umsetzung einer Raytracing-Technik liegt nicht in der Implementierung der Technik, sondern in der Optimierung für spezielle und komplexe Szenen.

In dem Gebiet der Visualisierung von Volumendaten gibt es zwei primäre Raytracing-Techniken: Volume Rendering und Isosurface Rendering. Erstere dient vor allem dazu, das Volumen als solches darzustellen: Die verschiedenen Dichte-Werte im Volumen können transparent visualisiert werden um zum Beispiel einen CT-Scan zu visualisieren. Beim Isosurface Rendering ist lediglich die Visualisierung einer Oberfläche, die durch einen Schwellenwert (Iso-Wert) angegeben wird, von Interesse. Da die tatsächlichen Volumendaten eines Terrains nicht relevant sind und es genau um eine Oberfläche geht, bietet sich die Verwendung des Isosurface Renderings an. Ferner wird diese Technik für die Visualisierung eines Terrains nicht verbreitet verwendet und stellt einen ausbaufähigen Bereich in der Wissenschaft dar, mit dem sich diese Arbeit auseinandersetzt.

Kapitel 3

Raytracing

Die grundlegende Raytracing-Idee stammt aus den späten 60er Jahren (siehe [3]) und wurde seitdem stetig weiterentwickelt. Seit jeher gilt Raytracing als intuitives aber rechenintensives Verfahren um eine dreidimensionale Szene in ein zweidimensionales Bild zu überführen, das auf einem Monitor angezeigt werden kann (siehe [24]). Aufgrund des hohen Rechenaufwandes hat sich stattdessen die Rastergrafik im Endverbraucher-Markt durchgesetzt. Raytracing wurde dennoch weiter erforscht und wird heutzutage hauptsächlich für die Berechnung fotorealistischer Bilder (z. B. in Animationsfilmen) verwendet. Dieser Abschnitt befasst sich mit den Grundlagen und einigen Erweiterungen des ursprünglichen Raytracing-Ansatzes.

3.1 Grundlagen

Allen Raytracing-Verfahren gemein ist die Verfolgung von Strahlen durch eine dreidimensionale Szene. Dabei muss für jedes Pixel einer impliziten Bildebene entschieden werden, mit welcher Farbe es eingefärbt wird. Die Szene kann aus expliziter Geometrie, wie zum Beispiel aus Dreiecken, oder implizit aus Volumendaten bestehen. Ferner wird zwischen offline und online Raytracing unterschieden. Offline Raytracing beschäftigt sich mit der Erzeugung von möglichst realitätsnahen Bildern, von denen jedes einzelne mehrere Minuten bis Stunden berechnet wird. Online Raytracing stellt keine derart hohen Anforderungen an die visuelle Qualität - stattdessen steht die schnelle Berechnung jedes Bildes und damit die Echtzeitfähigkeit des Verfahrens im Vordergrund. Deshalb werden offline Raytracer in der Praxis meist für computergenerierte Film-Szenen und online Raytracing Verfahren in einigen Computerspielen genutzt. Für diese Arbeit kommt dementsprechend nur die Verwendung eines online Raytracing Verfahrens infrage.

Ein grundsätzlicher Raytracing-Algorithmus lässt sich zum Beispiel wie folgt formulieren:

Algorithmus 2 : Grundlegender Raytracing-Algorithmus

```
foreach Pixel p der Bildebene do
    /* erstelle Strahl von Betrachter durch Pixel-Mitte im Raum          */
    Ray r ← erstelleStrahl(Betrachterstandpunkt, p.Mittelpunkt);
    /* Initialisiere Farbe und minimale Distanz                          */
    minDistanz ← positiv unendlich;
    pixelFarbe ← schwarz;
    /* Pruefe jedes primitive Objekt                                     */
    foreach primitives Objekt o der 3D-Szene do
        /* Errechnet Schnittpunktsdistanz mit dem Objekt - pos. unendlich bei
           keinem Schnittpunkt                                           */
        neueDistanz ← o.SchnittpunktDistanz(r);
        if (neueDistanz < minDistanz) then
            minDistanz ← neueDistanz;
            pixelFarbe ← o.FarbeAnPunkt(r · neueDistanz);
        end
    end
    /* Berechne Beleuchtung                                              */
    beleuchtungsTerm ← berechneBeleuchtung(...);
    /* Faerbe Pixel entsprechend ein                                     */
    p.Farbe ← pixelFarbe · beleuchtungsTerm;
end
```

Dieses Code-Beispiel (Algorithmus 2) skaliert linear mit der Auflösung der Bildebene (Anzahl der Pixel) sowie mit der Anzahl zu behandelnder Objekte. Während sich Letzteres durch die Verwendung von Datenstrukturen (z. B. kD-Tree) verringern lässt, bleibt Ersteres als Einschränkung bestehen. Abbildung 3.1 zeigt einen beispielhaften Aufbau einer Raytracing-Szene. Für jedes Pixel der gerasterten Bildebene wird ein Strahl vom Augpunkt durch den Mittelpunkt des Pixels erstellt. Diese Strahlen werden durch die Szene verfolgt um jeweils den ersten Schnittpunkt mit den Objekten der Szene zu ermitteln. Die gefundenen Schnittpunkte werden anschließend lokal beleuchtet.

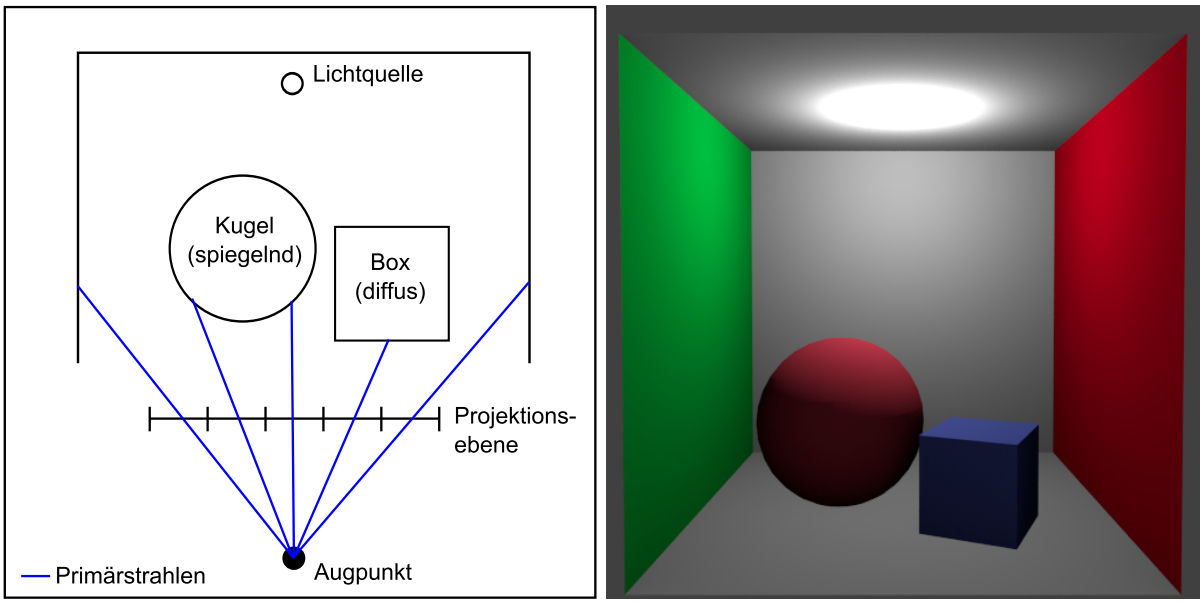

Abbildung 3.1: Schematische Darstellung des Raytracings als Kollisionsberechnung (links) und ein damit berechnetes Bild (rechts)

Es werden jeweils nur die Strahlen dargestellt, die mit Raytracing berechnet werden, also auf etwaige Schnittpunkte mit der Geometrie der Szene geprüft werden. Neben der schematischen Funktionsweise ist in der Abbildung 3.1 ein nach diesem Schema erzeugtes Bild zu sehen, das mit der Animations-Software Blender[4] berechnet wurde.

Das Raytracing-Verfahren orientiert sich nahe an der tatsächlichen Physik und ist deshalb intuitiv verständlich. Weitere Stärken liegen in der guten Erweiterbarkeit des Verfahrens: Über die letzten Jahrzehnte wurden viele Raytracing-Verfahren erdacht und umgesetzt. Um die Wichtigsten dieser Verfahren soll es im Folgenden gehen.

[4] `http://www.blender.org`

3.2 Erweiterungen

3.2.1 Schattenstrahlen

Die trivialste Erweiterung des Raytracing-Verfahrens ist die Verfolgung von Schattenstrahlen (*shadow rays*, siehe z. B. Kapitel 1.2.4 in [19]), die in Abbildung 3.2 dargestellt ist. Nachdem der erste Schnittpunkt eines Strahls mit einem Objekt ermittelt wurde, wird von diesem Schnittpunkt aus ein neuer Strahl erzeugt und zur Lichtquelle verfolgt. Wenn die Verfolgung dieses Strahls in einem neuen Schnittpunkt resultiert, ist der direkte Weg zwischen dem ursprünglichen Schnittpunkt und der Lichtquelle blockiert - er befindet sich also im Schatten des zweiten Schnittpunktes und die Beleuchtung kann entsprechend verändert werden. Wenn die Szene mehrere Lichtquellen enthält muss für jede dieser Lichtquellen ein Schattenstrahl erstellt und verfolgt werden um eine Abstufung des Schattens zu erreichen. Das Ergebnis sind pixelgenaue, harte Schatten. Um die Schattenkanten weicher zu gestalten, kann die Lichtquelle zum Beispiel als Kugel modelliert werden, zu der dann für jeden Punkt mehrere minimal unterschiedliche Schattenstrahlen verfolgt werden. An den Schattenkanten wird nur ein Teil dieser Strahlen die Lichtquelle erreichen und somit können die Schattenkanten weicher dargestellt werden.

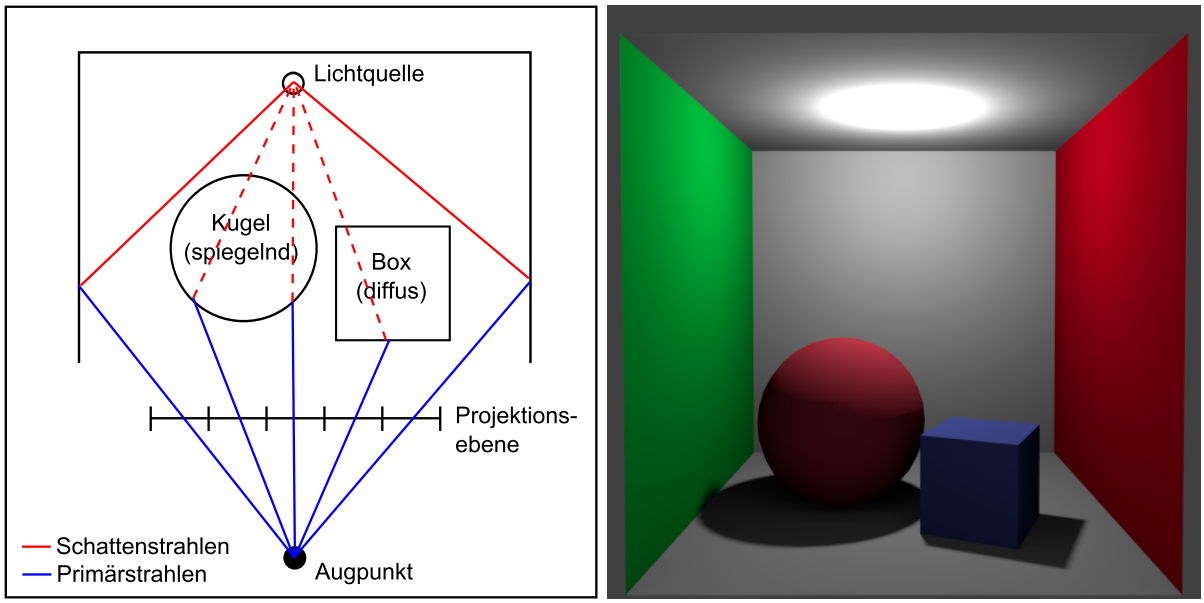

Abbildung 3.2: Schematische Darstellung des Raytracings mit Schattenstrahlen (links) und ein damit berechnetes Bild (rechts)

3.2.2 Rekursives Raytracing

Bei der Erweiterung um Sekundärstrahlen (*secondary rays*, siehe Kapitel 1.2.6 in [19]) werden die eigentlichen Stärken des Verfahrens deutlich. Die Strahlen vom Betrachter-Standpunkt durch die Bildebene werden Primärstrahlen (*primary rays*) genannt. Um die physikalische Verbreitung des Lichts im Raum genauer anzunähern, werden beim rekursiven Raytracing an den Schnittpunkten mit der Geometrie weitere Strahlen erzeugt und verfolgt - diese Strahlen werden Sekundärstrahlen genannt. In der einfachsten Form des rekursiven Raytracings wird ein Strahl bei Kollision mit Geometrie anhand des Einfallswinkels reflektiert und der resultierende Sekundärstrahl weiterverfolgt. Ob und wenn ja zu welchem Anteil der Strahl an einer Oberfläche reflektiert wird, hängt von den Materialeigenschaften (wie z. B. Reflexions- oder Absorptionskoeffizienten) der Oberfläche ab. Lichtdurchlässige Oberflächen (z. B. Glas oder Wasser) können mithilfe eines Brechungskoeffizienten ebenfalls modelliert werden. Wenn die erzeugten Sekundärstrahlen ebenfalls auf Geometrie treffen, kann die Methode rekursiv wiederholt werden - als Abbruchbedingung kann eine maximale Tiefe der Rekursion dienen. Das Ergebnis ist die visuelle Darstellung von Transparenz, Reflexion und Spiegelungen. Die bei den Schattenstrahlen angesprochene Methode, in einigen Fällen mehr als einen Strahl zu erstellen, ist eigentlich Gegenstand des diffusen Raytracings (auch: Stochastisches Raytracing, siehe Kapitel 13 und 14 in [19]). Dieses Verfahren trägt der Tatsache Rechnung, dass punktförmige Lichtquellen nicht existieren und jede tatsächliche Lichtquelle eine gewisse Ausdehnung hat. In gewissen Fällen (z. B. bei einem Schatten- oder Brechungsstrahl) wird nicht einer sondern mehrere Sekundärstrahlen erstellt, verfolgt und ihre resultierenden Farben anschließend gemittelt. Wenn zu wenig Strahlen erzeugt werden oder die Szene zu wenige beziehungsweise zu kleine Lichtquellen enthält, kann es zu Rausch-Artefakten im resultierenden Bild kommen.

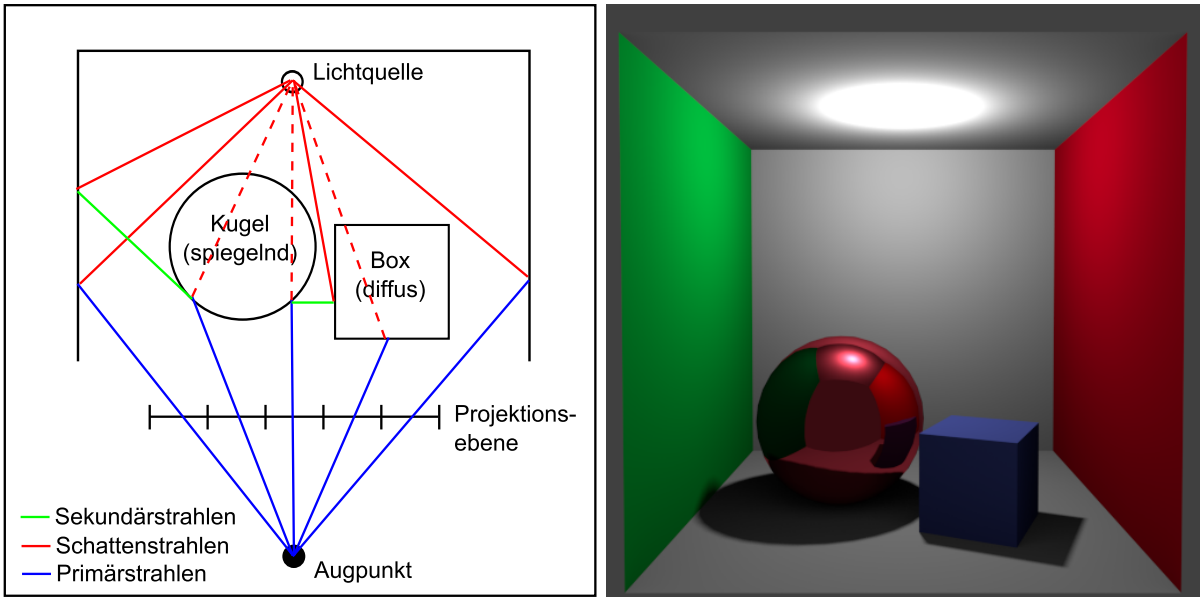

Abbildung 3.3: Schematische Darstellung des rekursiven Raytracings (links) und ein damit berechnetes Bild (rechts)

3.2.3 Path-Tracing

Die bisherigen Verfahren sind nicht in der Lage, die Lösung der *Rendering-Equation* (siehe Kapitel 5 in [19]), also die globale Beleuchtung einer Szene zu gewährleisten. Diese Integralgleichung versucht die Energieerhaltung bei der Ausbreitung von Lichtstrahlen zu beschreiben. Die Schwierigkeit besteht also darin, die tatsächliche Ausbreitung des Lichts zu berechnen beziehungsweise ausreichend genau anzunähern. Um dies zu gewährleisten, werden beim Path Tracing (siehe Kapitel 15.3 in [19]) auch beim Auftreffen auf diffusen Oberflächen mehrere Sekundärstrahlen erzeugt und verfolgt. Somit kann die Ausbreitung des Lichts im Raum approximiert werden wobei auch bei diesem Verfahren Rausch-Artefakte auftreten, wenn zu wenige Strahlen erzeugt werden. Ferner können mit Path Tracing Effekte wie Kaustiken (helle Lichtflecken, die meist durch Bündelung von Licht durch ein lichtdurchlässiges Medium auftreten) erzeugt werden. Als einziges der genannten Verfahren ist Path Tracing in der Lage indirekte Beleuchtung von Oberflächen, die von keiner Lichtquelle direkt beeinflusst werden, zu simulieren.

Zusätzlich zu den bisher genannten Verfahren, die allesamt den Augpunkt als Ursprung haben, existieren auch Verfahren, die Strahlen von der Lichtquelle aus erzeugen. Da diese Verfahren allerdings sehr viele unnötige Strahlen erzeugen, werden sie in der Praxis meist nur als Erweiterung beziehungsweise Kombination mit anderen Verfahren verwendet.

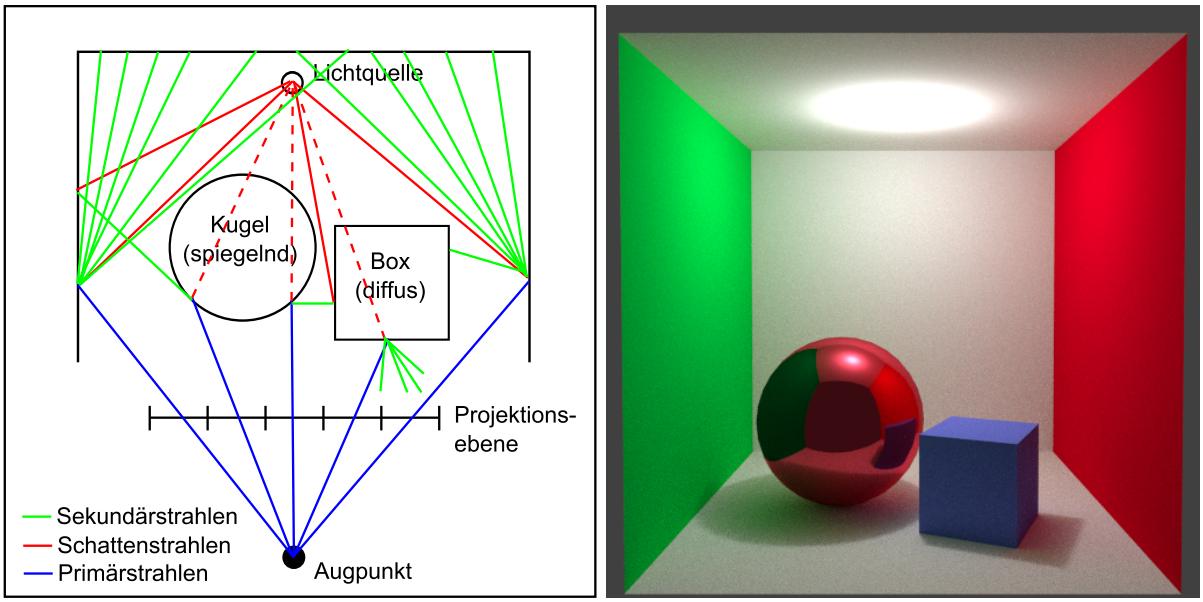

Abbildung 3.4: Schematische Darstellung des Path Tracings (links) und ein damit berechnetes Bild (rechts)

3.3 Volumendaten

Das Raytracing-Verfahren und die Erweiterungen werden zu Demonstrationszwecken oft lediglich auf simpler, impliziter Geometrie, wie zum Beispiel Kugeln oder Ebenen, angewandt um die Komplexität der Berechnungen gering zu halten. Da für ein qualitativ hochwertiges Ergebnis mehrere Millionen Strahlen verfolgt werden müssen, ist dies beim online Raytracing nötig. Neben diesen konstruierten Szenen, die die Vorteile der Verfahren optimal widerspiegeln, existieren auch Daten aus der realen Welt, die mithilfe von Raytracing visualisiert werden können ohne sie vorher in Geometrie (Dreiecke) umzurechnen. Ein klassisches Beispiel für solche realen Daten sind die Ergebnisse einer Computertomographie (CT), die meist in Form von Volumendaten vorliegen. Aufgrund der Datenmenge solcher CT-Scans (oft $256^3 \approx 16,7$ Millionen Werte) und der Tatsache, dass es sich nicht um Geometrie- sondern Volumendaten handelt, müssen hierfür spezielle Raytracing-Techniken angewandt werden.

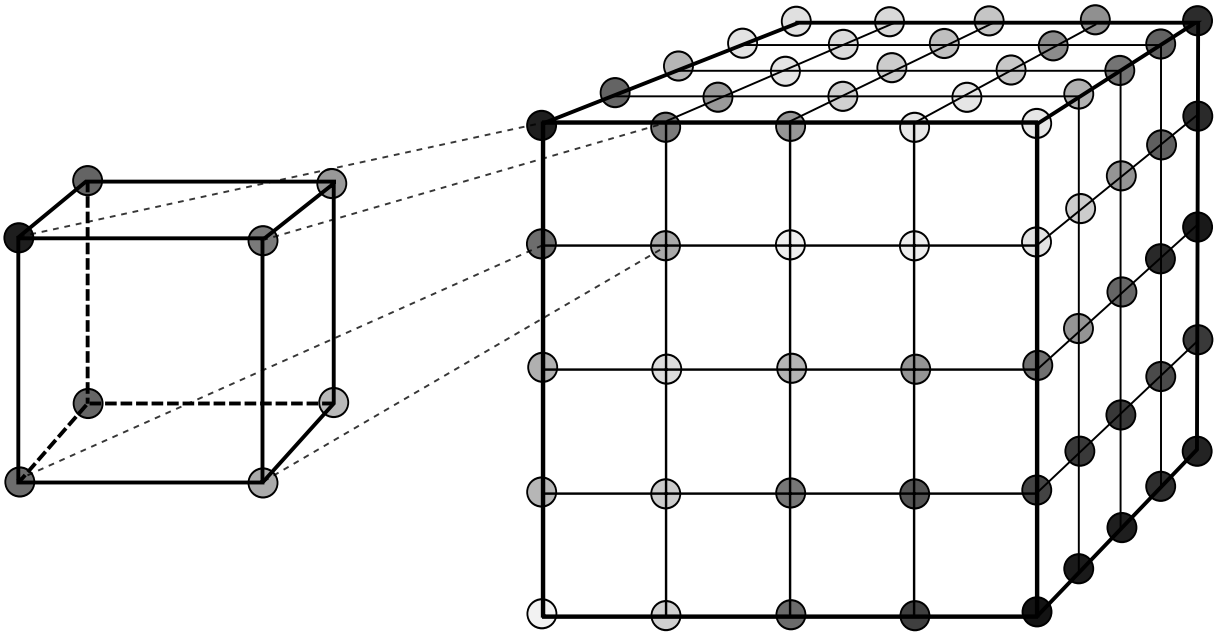

Abbildung 3.5: Schematische Darstellung eines dreidimensionalen Voxel-Gitters. Dunkle Kreise stehen für hohe und helle für niedrige Dichte.

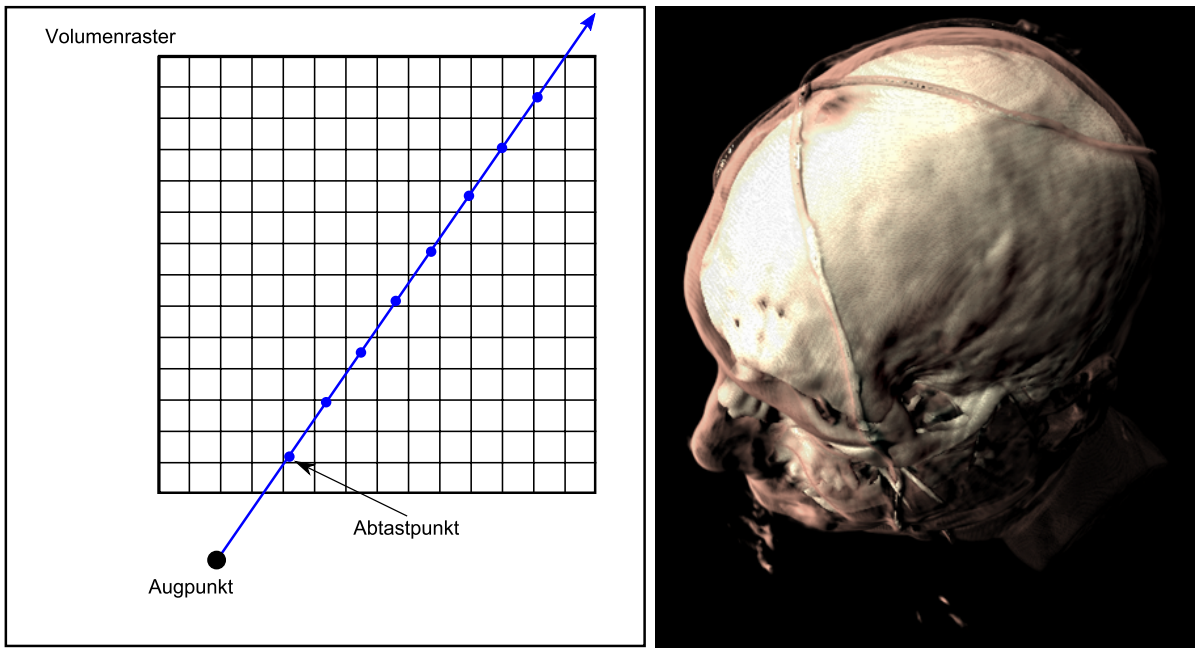

Abbildung 3.6: 2D-Schema des Volume Rendering Verfahrens (links) und ein damit berechnetes Bild[5] (rechts)

Um die Eigenschaften eines solchen CT-Scans zu visualisieren, bieten sich Spezialfälle von Raytracing an. Beim Volume Rendering werden ebenfalls Primärstrahlen erzeugt, die innerhalb des Volumens kontinuierlich abgetastet werden. Da die Daten in einem diskretem Gitter vorliegen, müssen die tatsächlichen Dichte-Werte an den Abtastpunkten des Strahls aus den umliegenden Datenpunkten interpoliert werden. Anhand der Dichte erfolgt meist eine Klassifizierung des Materials (z. B. Blutgefäß, Haut oder Knochen) um anschließend unter anderem die Farbe oder etwaige Transparenz behandeln zu können. Die Punkte werden dann einzeln lokal beleuchtet und im letzten Schritt gemischt. Diese Mischung beginnt am letzten und endet im ersten Punkt, damit etwaige Details, auf die der Strahl als erstes trifft, nicht von dahinter liegenden überlagert werden. So kann zum Beispiel die Haut teilweise transparent dargestellt werden um durch sie hindurch auf das Skelett zu blicken.

[5] Abbildung: `http://graphicsrunner.blogspot.de/2009/01/volume-rendering-102-transfer-functions.html`

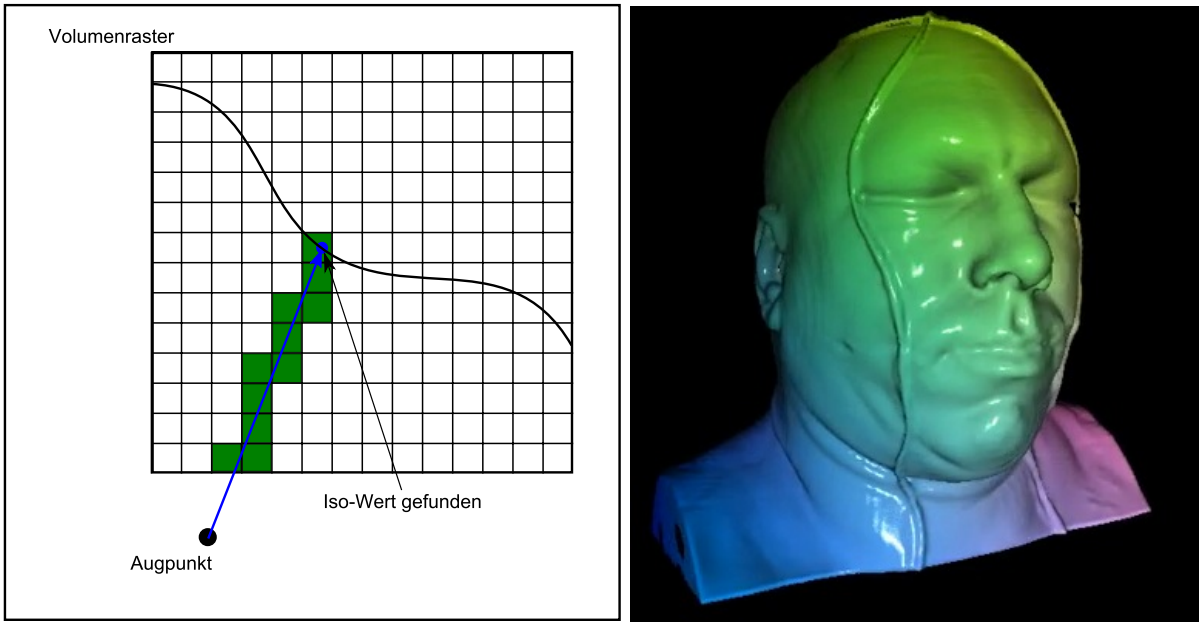

Abbildung 3.7: 2D-Schema des Isosurface Rendering Verfahrens (links) und ein damit berechnetes Bild[6] (rechts)

Um aus den Volumendaten Oberflächen zu visualisieren, bietet sich die Verwendung von Isosurface Rendering an. Anstatt das Volumen als solches zu visualisieren geht es bei diesem Verfahren darum, aus den Volumendaten Oberflächen zu extrahieren. Der Schwellenwert (oder auch: Iso-Wert) der Dichte, ab dem innerhalb des Volumens eine Oberfläche erkannt wird, kann dabei variiert werden. Grundsätzlich muss jeder Strahl von Anfang bis Ende durch das Volumen verfolgt und regelmäßig daraufhin überprüft werden, ob er eine Oberfläche berührt. Anstatt den Strahl regelmäßig abzutasten werden die Voxel, durch die der Strahl stößt, überprüft. Sobald eine Oberfläche gefunden wurde, kann diese beleuchtet werden und es können etwaige Sekundärstrahlen ausgesandt werden. Da das Verfahren in der Lage ist, Oberflächen aus Volumendaten zu extrahieren ohne explizite Geometrie zu erzeugen, eignet es sich sehr gut für die direkte Visualisierung eines Terrains.

[6] Abbildung von: `http://media2mult.uos.de/pmwiki/fields/cg-II-09/index.php?n=VolumeRendering.IsosurfaceRayCasting`

Kapitel 4

Theorie

Um aus den Volumendaten ein Isosurface-Terrain zu visualisieren, sind vier Schritte notwendig. Erstens müssen die Volumendaten als Grundlage generiert und gespeichert werden. Im zweiten Schritt muss ein Methode vorhanden sein, mit der für beliebige Strahlen innerhalb eines Voxels entschieden werden kann, ob und wenn ja wo ein Strahl auf die Isosurface trifft. Drittens ist eine Methode vom Nöten, mit der die Volumendaten effizient traversiert werden können. Im vierten und letzten Schritt müssen die gefunden Schnittpunkte mit Hilfe eines Beleuchtungsmodells beleuchtet werden. Dieses Kapitel wird die Theorie dieser vier Schritte behandeln.

4.1 Daten

Wie in Abschnitt 2.2 bereits erwähnt, dienen dreidimensional gerasterte Dichte-Werte als Grundlage für die meisten Voxel-basierten Visualisierungstechniken. Ein solches Raster muss für die Visualisierung eines Terrains mit entsprechenden Daten gefüllt werden. Diese Daten müssten bei niedriger Höhenkoordinate hohe Dichte-Werte, bei hoher Höhenkoordinate niedrige Dichte-Werte und dazwischen einen Übergang zwischen diesen Werten gewährleisten.

Aufgrund der hohen Datenmenge und dem speziellen Anwendungsgebiet, ist es schwierig, vorgegebenen Volumendatensätze finden, die ein Terrain symbolisieren. Die vorhandenen Datensätze sind häufig die Ergebnisse von CT-Scans (siehe Abbildungen 3.6 und 3.7 in Abschnitt 3.3). Deshalb bietet es sich an, einen prozeduralen Algorithmus, der speziell für die Anforderungen dieser Arbeit angepasst werden kann, für die Generierung der benötigten Daten zu verwenden.

Ebert et al. [5] nennen die Abstraktion als wichtigstes Merkmal prozeduraler Techniken. Anstatt die Details einer Szene explizit zu speichern, können diese durch abstrakte Funktionen oder Algorithmen abstrahiert und erst bei Bedarf berechnet werden. Anstatt die Daten zufällig zu erzeugen, können diese Funktionen oder Algorithmen vom Entwickler festgelegt und mit einer Kontrolle über Parameter versehen werden. Als weitere Wichtige Eigenschaft liefern diese Techniken bei gleichen Ausgangssituationen immer wieder dieselben Inhalte.

Obwohl Algorithmen der prozeduralen Generierung nicht zufällig arbeiten, basieren sie meist auf pseudo-zufällig generierten Daten. Bei der Erstellung von prozeduralen, zweidimensionalen Texturen werden häufig Noise-Funktionen benutzt und da ein Volumendatensatz einer dreidimensionalen Textur entspricht, bieten sich diese Funktionen auch dort an. Noise-Funktionen sind meist für beliebig viele Dimensionen definiert und liefern für Koordinaten dieser Dimensionen einen normierten ([0,1] oder [-1,1]) Wert. Die wohl bekannteste Noise-Funktion ist die nach ihrem Erfinder benannte Perlin-Noise-Funktion beziehungsweise die Erweiterung simplex-Noise ([17]). Durch Aufsummieren verschiedener Frequenzen können unterschiedliche prozedurale Fraktale (siehe Abbildung 4.1, rechts) erstellt werden.

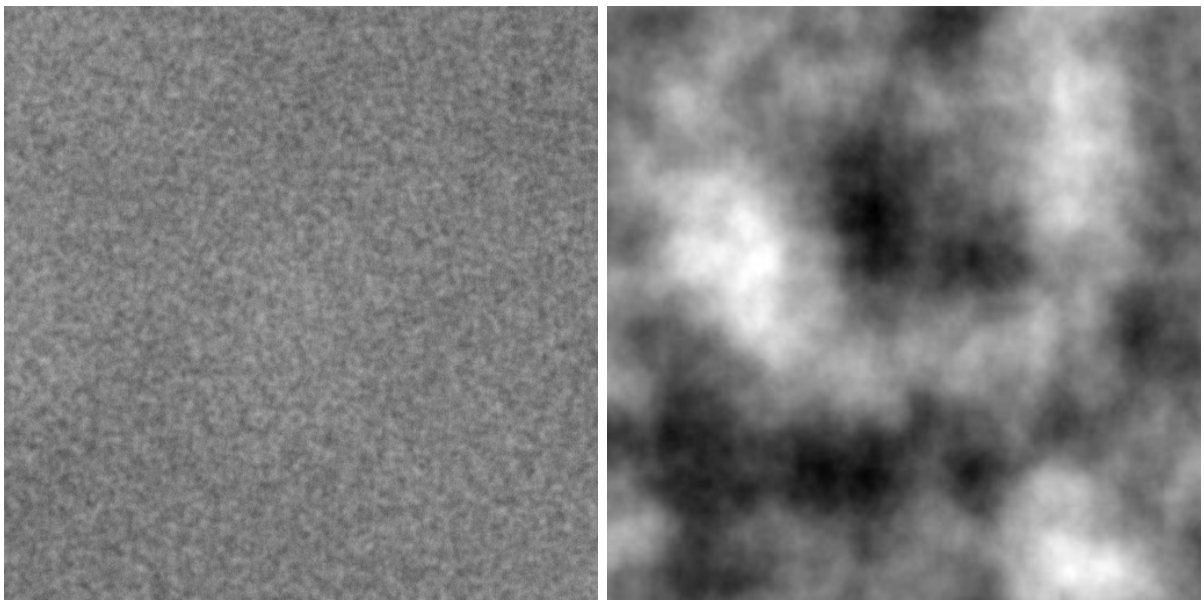

Abbildung 4.1: Perlin-Noise mit hoher Frequenz (links) und unterschiedlichen aufsummierten Frequenzen (rechts)

4.2 Schnittpunktberechnung

Um aus den Volumendaten mittels Isosurface Rendering ein Terrain zu visualisieren, bedarf es einer Methode, um für beliebige Strahlen, die das Volumen durchstoßen, zu entscheiden, ob und wenn ja wo der Strahl auf eine Oberfläche (definiert durch den gesuchten Iso-Wert) trifft. Gesucht wird das kleinste t des Strahls, für das

$$Dichte(Strahl \cdot t) = Iso\text{-}Wert$$

gilt. Deshalb ist es sinnvoll, den Strahl von dem Eintritts- bis maximal zum Austrittspunkt durch das Volumen zu verfolgen (siehe Abschnitt 4.3) und die abgeschrittenen Voxel auf etwaige Oberflächen zu prüfen.

Für diese Prüfung innerhalb eines Voxels wurden bereits mehrere Methoden untersucht, auf die im im Folgenden kurz eingegangen wird. Alle Methoden basieren auf den acht Dichte-Werten eines Voxels und können in approximative und genaue Verfahren unterteilt werden. Die approximativen Verfahren benötigen als Grundlage eine Methode für die Berechnung der Dichte an einem beliebigem Punkt innerhalb eines Voxels, die mittels trilinearer Interpolation (siehe Abbildung 4.2) berechnet werden kann.

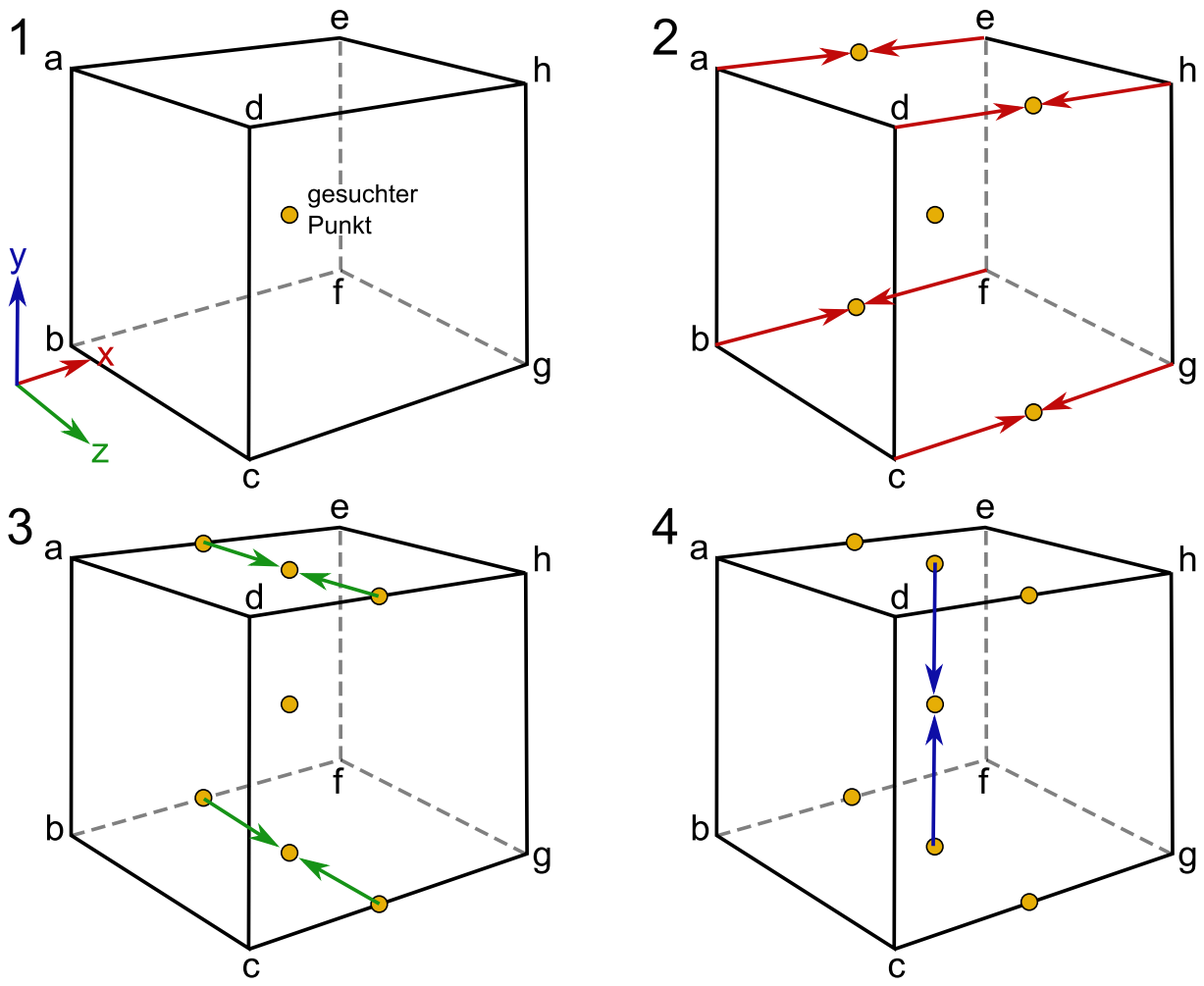

Abbildung 4.2: Schematische Darstellung der trilinearen Interpolation

Das in Abbildung 4.2 veranschaulichte Schema basiert auf den mathematischen Herleitungen von Levoy [11] und lässt sich wie folgt in Pseudocode formulieren.

Algorithmus 3 : Pseudocode der trilinearen Interpolation innerhalb eines Voxels

Data : voxel: Die Datenstruktur, die die acht Dichte-Werte enthält
Data : point: Der Punkt innerhalb des Voxels, für den die Dichte berechnet werden soll

```
/* Errechne Einheitskoordinaten [0,1] der Koordinaten                    */
```
$x \leftarrow \frac{point.x - voxel.min.x}{voxel.max.x - voxel.min.x}$;
$y \leftarrow \frac{point.y - voxel.min.y}{voxel.max.y - voxel.min.y}$;
$z \leftarrow \frac{point.z - voxel.min.z}{voxel.max.z - voxel.min.z}$;

```
/* Interpolation in x                                                    */
```
c00 ← voxel.a · (1-x) + voxel.e · x;
c10 ← voxel.b · (1-x) + voxel.f · x;
c01 ← voxel.c · (1-x) + voxel.g · x;
c11 ← voxel.d · (1-x) + voxel.h · x;

```
/* Interpolation in z                                                    */
```
c0 ← c00 · (1-z) + c10 · z;
c1 ← c01 · (1-z) + c11 · z;

```
/* Interpolation in y                                                    */
```
return (c0 · (1-y) + c1 · y);

Der simpelste approximative Algorithmus [14] geht von einer linearen Funktion innerhalb des Voxels aus und führt eine lineare Interpolation zwischen den Dichte-Werten des Eintritts- und Austrittspunkt des Strahls durch den Voxel, durch.

Algorithmus 4 : Pseudocode der linear interpolierten Schnittpunktberechnung

Data : ray: Der aktuelle Strahl
Data : pIso: Der gesuchte Iso-Wert
Data : tIn, tOut: Die t-Faktoren für den Eintritts- und Austrittspunkt des Strahls
Data : voxel: Die Datenstruktur, die die acht Dichte-Werte enthält

```
/* Errechne Dichte am Eintritts- und Austrittspunkt                      */
```
pIn ← trilineareInterpolation(voxel, ray · tIn);
pOut ← trilineareInterpolation(voxel, ray · tOut);
```
/* Breche ab, wenn der Iso-Wert nicht überschritten wird (kein
   Vorzeichenwechsel)                                                    */
```
if *sign(pIn - pIso) = sign(pOut - pIso)* **then**
  return noHit;
end
return tHit = tIn + (tOut - tIn) · $\frac{pIso - pIn}{pOut - pIn}$;

Darauf aufbauend formulierten Neubauer et al. [2] einen Algorithmus, der diese linear interpolierte Schnittpunktberechnung mehrfach wiederholt. Nach jedem Schritt teilt der gefundene approximierte Schnittpunkt den Strahl in zwei Hälften und das Verfahren wird rekursiv in dem Segment wiederholt, in dem der Iso-Wert enthalten ist. Das Ergebnis ist eine genauere Approximation des Schnittpunktes, die - sofern die Daten des Voxels nur ein einziges Mal aus dem Speicher geladen werden - sich kaum auf die Performanz auswirkt.

Algorithmus 5 : Pseudocode von Neubauers wiederholter linearen Interpolation

Data : ray: Der aktuelle Strahl
Data : pIso: Der gesuchte Iso-Wert
Data : tIn, tOut: Die t-Faktoren für den Eintritts- und Austrittspunkt des Strahls
Data : voxel: Die Datenstruktur, die die acht Dichte-Wert enthält
Data : N: Anzahl an Wiederholungen

```
/* Errechne Dichte am Eintritts- und Austrittspunkt                    */
p0 ← trilineareInterpolation(voxel, ray · tIn);
p1 ← trilineareInterpolation(voxel, ray · tOut);
/* Breche ab, wenn der Iso-Wert nicht überschritten wird (kein
   Vorzeichenwechsel)                                                  */
if sign(p0 - pIso) = sign(p1 - pIso) then
    return noHit;
end

for i..N do
    t ← tIn + (tOut - tIn) · (pIso−p0)/(p1−p0);
    pNew = trilineareInterpolation(voxel, ray · t);
    /* Prüfe die beiden Strahlen-Segmente und interpoliere in einem weiter */
    if sign(pNew - pIso) = sign(p0 - pIso) then
        tIn ← t;
        p0 ← pNew;
    end
    else
        tOut ← t;
        p1 ← pNew;
    end
end

return tHit ← tIn + (tOut - tIn) · (pIso−p0)/(p1−p0);
```

Dieser Prozess könnte dynamisch, zum Beispiel bei einem unterschrittenem Mindestwert der Annäherung an den gesuchten Iso-Wert oder einer erreichten Mindestgenauigkeit, abgebrochen werden. Wenn die Voxel-Auflösung hoch genug ist, reicht eine feste, kleine Anzahl Wiederholungen allerdings aus.

Diese beiden approximativen Verfahren haben den Nachteil, dass sie in speziellen Fällen fehlerhafte Ergebnisse der Schnittpunktberechnung liefern. Dies geschieht zum Beispiel dann, wenn zwischen den Eintritts- und Austrittspunkt genau zwei Oberflächenübergänge stattfinden: Zum Beispiel wenn der Strahl innerhalb eines Voxel in der "Geometrie" (*Dichte* $\geq$ *Iso-Wert*) startet und endet, dazwischen diese Geometrie aber verlässt. Durch die Signum-Abfrage können diese Fälle nicht erkannt werden und beide Algorithmen finden keinen Schnittpunkt. Die folgenden Abbildungen verdeutlichen diese Fehlerfälle.

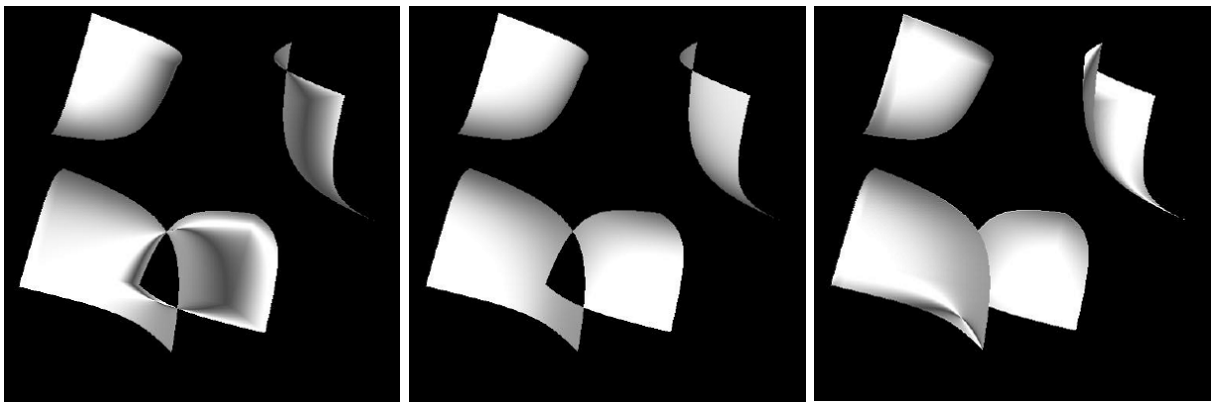

Abbildung 4.3: Durch Interpolation nicht gefundene Schnittpunkte (links und Mitte). Mit Marmitts Verfahren werden alle Schnittpunkte gefunden (rechts)[7].

Es existieren analytische Verfahren, die nicht von diesen Fehlerfällen betroffen sind. Parker, Shirley et al. [16] schlagen die Lösung eines kubischen Polynoms dritten Grades vor, dessen Lösung exakt aber auch aufwändig ist. Darauf aufbauend formulierte Schwarze [22] die analytische Inversion, die Spezialfälle des Polynoms vorab behandelt und erst anschließend das komplette Polynom löst.

Marmitt et al. [14] schlagen einen Algorithmus vor, der auf den folgenden beiden Beobachtungen basiert: Erstens ist nur der erste Schnittpunkt relevant und zweitens liefert die Neubauer-Methode den korrekten Schnittpunkt, wenn in dem betrachteten Intervall genau ein Oberflächenübergang stattfindet. Dazu werden die Oberflächenübergänge (Wurzeln der Gleichung) durch Betrachtung der Extremstellen isoliert. Dafür muss lediglich die quadratische erste Ableitung des Polynoms, also

$$f'(t) = 3At^2 + 2Bt + C$$

gelöst werden. Anschließend wird der Strahl an diesen Extremstellen aufgeteilt und diese Segmente, die garantiert maximal einen Oberflächenübergang enthalten, anhand des gesuchten Iso-Werts in Richtung des Strahls verarbeitet. Innerhalb des ersten Segments, in dem der Oberflächenübergang stattfindet, wird mit Neubauers Methode der korrekte Schnittpunkt berechnet. Dieser Algorithmus liefert die gleichen exakten Ergebnisse ohne dafür auf die Lösung eines kubischen Polynoms angewiesen zu sein. Bei einer direkten Umsetzung des Algorithmus auf der GPU kam es allerdings zu nicht erkannten Schnittpunkten, was in Bildrauschen resultiert.

[7] Abbildungen aus [14]

4.3 Volumen-Traversierung

Wie im Kapitel 3.1 beschrieben, muss für jedes Pixel des zu berechnenden Bildes mindestens ein Strahl erstellt und durch das Volumen verfolgt werden, um die Farbe dieses Pixels zu ermitteln. Dabei besteht das Problem, dass ein Großteil der circa zwei Millionen Voxel innerhalb eines Blockes keinen Oberflächen-Übergang enthalten und somit für die Terrain-Visualisierung leeren Raum darstellen. Die Herausforderung eines Algorithmus für interaktive Anwendungen liegt also in einer schnellen Verarbeitung dieser leeren Blöcke beziehungsweise darin, nur die Blöcke zu berechnen, in denen Oberflächen-Übergänge vorhanden sein können.

Eine mögliche Lösung dieses Problems ist die Verwendung von Raum-partitionierenden Algorithmen. Dabei wird der dreidimensionale Raum in kleiner Teilräume unterteilt und diese Teilräume werden in einer Datenstruktur (meist Baum-Strukturen) gespeichert. Bei der Traversierung des Raumes oder des Volumens wird dann diese Datenstruktur traversiert und innerhalb der Blatt-Knoten nach einem Schnittpunkt gesucht. Für den Spezialfall des Isosurface-Renderings schlagen Wald et al. [23] die Verwendung eines impliziten kD-Baumes vor, bei dem die Unterteilung des Raumes in einem binären Baum gespeichert wird. Um den Iso-Wert zur Laufzeit ändern zu können, werden Bäume für mehrere vordefinierte Iso-Werte vorberechnet. Im selben Jahr veröffentlichten Foley et al. [6] einen Ansatz zur Traversierung eines kD-Baumes auf modernen GPUs, der aber nicht die Performanz einer optimierten CPU-Lösung erreichte. Dieser *kd-restart* Ansatz wurde von Horn et al. [8] um zwei Techniken (*push-down* und *short-stack*) erweitert, um die speziellen Eigenschaften der Hardware auszunutzen. Eine weitere Abwandlung von Foley's Ansatz ist der von Hughes et al. [9] vorgestellte *kd-jump* Ansatz, bei dem Nachbarknoten durch Sprünge schneller erreicht werden. Diese Verfahren gehen von statischen Szenen aus und somit müsste bei einer Veränderung der Volumendaten zusätzlich die Datenstruktur angepasst oder sogar komplett neu erstellt werden.

Alternativ zu der Verwendung von Baum-Strukturen bietet es sich an, die speziellen Eigenschaften der Volumendaten zu nutzen. Amantides und Woo [1] schlagen die Verwendung eines Voxel-Stepping Algorithmus für das Raytracing von simplen geometrischen Objekten vor. Der dreidimensionale Raum wird in ein gleichmäßiges Raster unterteilt, wobei für jede Zelle gespeichert wird, welche Objekte in ihr enthalten sind. Die Strahlen werden dann im Raytracing-Schritt durch dieses Raster verfolgt, um in jeder abgeschrittenen Zelle nur die enthaltenen Objekte auf einen etwaigen Schnittpunkt zu prüfen (Abbildung 4.4 veranschaulicht dieses Vorgehen). Da die Volumendaten bereits in einem gleichmäßigen, dreidimensionalem Gitter angeordnet sind, ist die Portierung dieses Algorithmus (siehe Algorithmus 6) denkbar einfach: Die Zellen entsprechen den Voxeln und anstatt die enthaltenen Objekte zu prüfen, müssen die acht Datenwerte aus dem Speicher geladen werden, um einen etwaigen Oberflächenschnittpunkt zu berechnen.

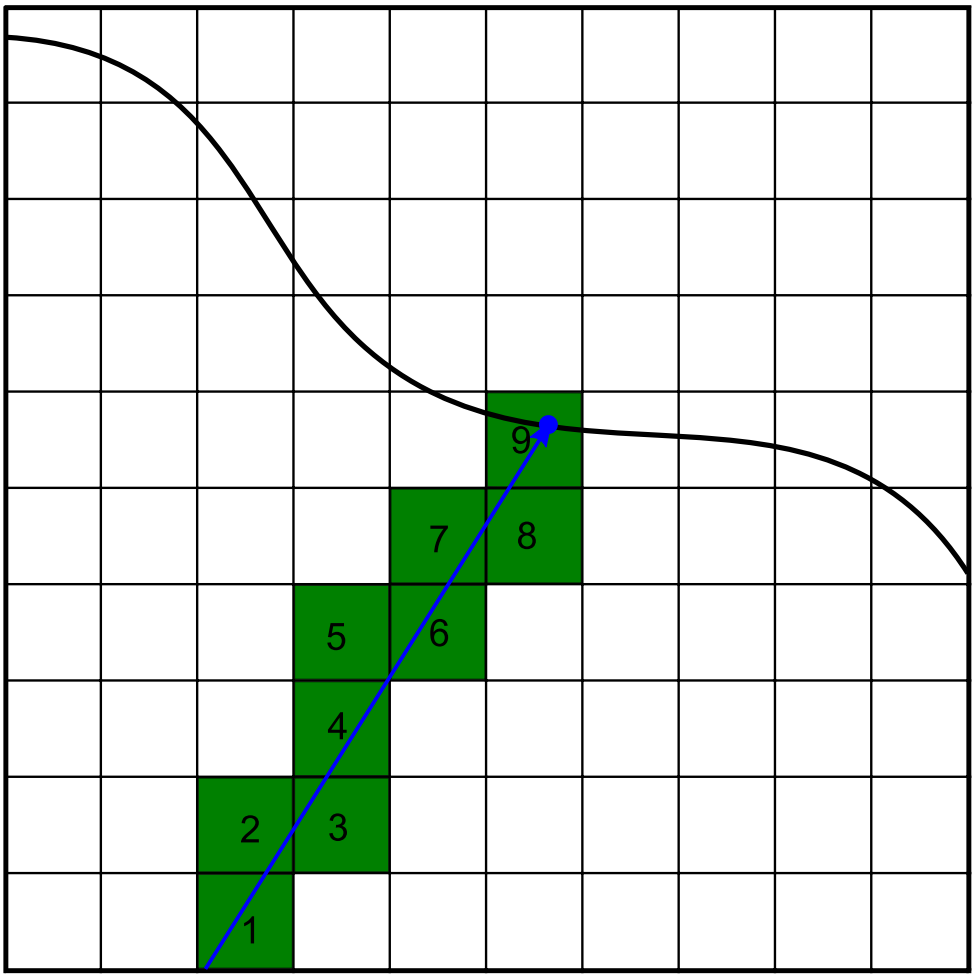

Abbildung 4.4: Zweidimensionales Schema des Voxel-Stepping-Algorithmus

Algorithmus 6 : Pseudocode des dreidimensionalen Voxel-Stepping-Algorithmus

Data : dimensionSize: Größe eines Voxels
Data : ray: Der aktuell verfolgte Strahl
Data : voxel: Der aktuelle Voxel (Start-Voxel)
Data : isovalue: Der gesuchte Dichte-Wert

```
/* initialisiere Variablen                                                    */
X,Y,Z ← Start-IDs des Voxels für jede Achse;
stepX, stepY, stepZ ← -1 oder 1 abhängig von den Vorzeichen des Strahls;
tMaxX, tMaxY, tMaxZ ← t-Wert, an dem der Strahl die entsprechende Voxel-Grenze erreicht;
tDeltaX, tDeltaY, tDeltaZ ← for tDeltaX = dimensionSize/((ray.x < 0) ? -ray.x : ray.x);

/* Schleife für das Abschreiten der Voxel                                     */
while kein Schnittpunkt gefunden und der Strahl sich noch im Volumen befindet do
    /* lese die acht Dichte-Werte aus dem Speicher                            */
    voxel.data ← readDensityValues(data, X, Y, Z);
    /* pruefe auf Schnittpunkt mit Oberfläche (Neubauer)                      */
    tHit ← intersectonFunction(voxel, ray, isovalue);
    if tHit > 0 then
        berechne weitere Werte (z. B. die Oberflächennormale);
        break;
    end
    /* Wenn kein Schnittpunkt gefunden wurde: Schreite zum nächsten Voxel     */
    if tMaxX < tMaxY then
        if tMaxX < tMaxZ then
            X ← X + stepX;
            tMaxX ← tMaxX + tDeltaX;
        end
        else
            Z ← Z + stepZ;
            tMaxZ ← tMaxZ + tDeltaZ;
        end
    end
    else
        if tMaxY < tMaxZ then
            Y ← Y + stepY;
            tMaxY ← tMaxY + tDeltaY;
        end
        else
            Z ← Z + stepZ;
            tMaxZ ← tMaxZ + tDeltaZ;
        end
    end
end
```

4.4 Beleuchtung

Nachdem der erste Schnittpunkt eines Strahls mit der Isosurface gefunden wurde, muss entschieden werden, wie der dazugehörige Pixel eingefärbt wird. Dazu wird mindestens ein Farbwert benötigt, der optional beleuchtet werden kann. Eine speicherintensive Möglichkeit diesen Farbwert zu ermitteln wäre, für jeden Voxel einen zusätzlichen Materialwert zu speichern, dem ein Farbwert zugeordnet wird. Alternativ dazu wird beim Volume Rendering meist eine Zuordnung vom Dichte-Wert auf eine Farbe vorgenommen - beim Isosurface Rendering würde dies allerdings in der gleichen Farbe für jeden Schnittpunkt resultieren. Stattdessen bietet sich bei der Visualisierung eines Terrains eine ähnliche Zuordnung von der normierten Höhe in Weltkoordinaten auf einen Farbwert an.

Ein Vergleich zwischen konstanter und berechneter Beleuchtung zeigt, wie sehr der plastische Eindruck einer 3D-Szene durch Beleuchtung verändert wird:

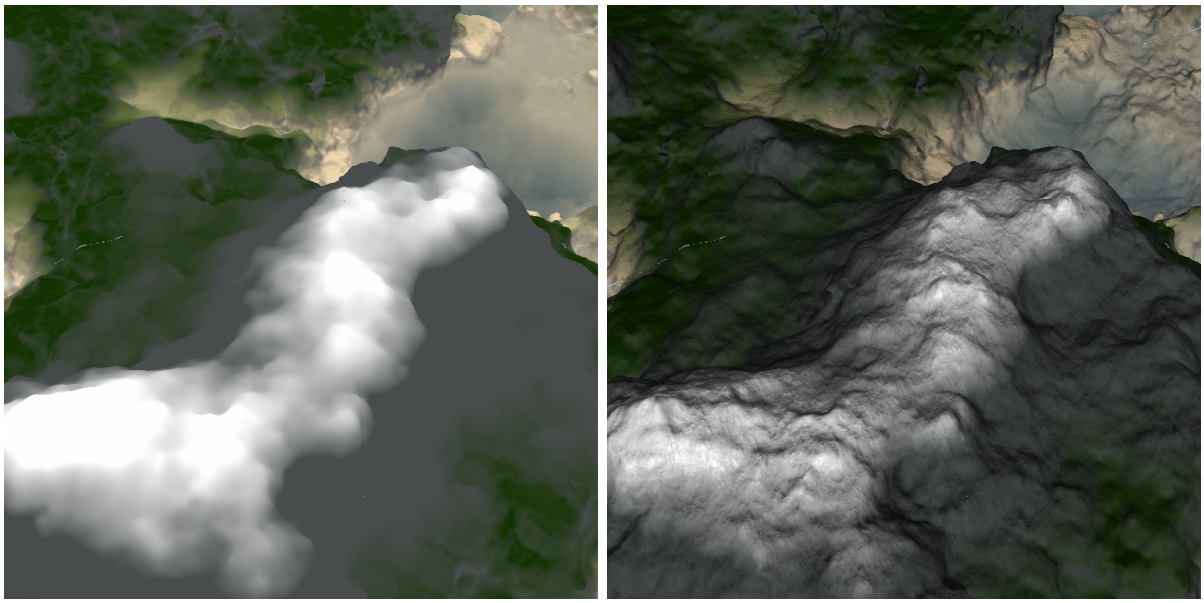

Abbildung 4.5: Terrain-Szene mit konstanter (links) und lokaler Beleuchtung (rechts)

Die Beleuchtungsmodelle der Computergrafik können grundsätzlich in lokale und globale Modelle unterteilt werden. Während die lokalen, empirischen Modelle eingesetzt werden, um Oberflächen von Objekten zu simulieren, setzen sich die globalen Modelle mit der (physikalisch korrekten) Ausbreitung von Licht innerhalb einer Szene auseinander. Die bekanntesten lokalen Beleuchtungsmodelle sind die Modelle von Phong [20] und Blinn [4].

Am Beispiel des Phong-Beleuchtungsmodells lässt sich die Funktionsweise der lokalen Beleuchtung erklären. Das Modell teilt die Beleuchtung in drei Komponenten auf: Den ambienten, diffusen und spekularen (spiegelnden) Term.

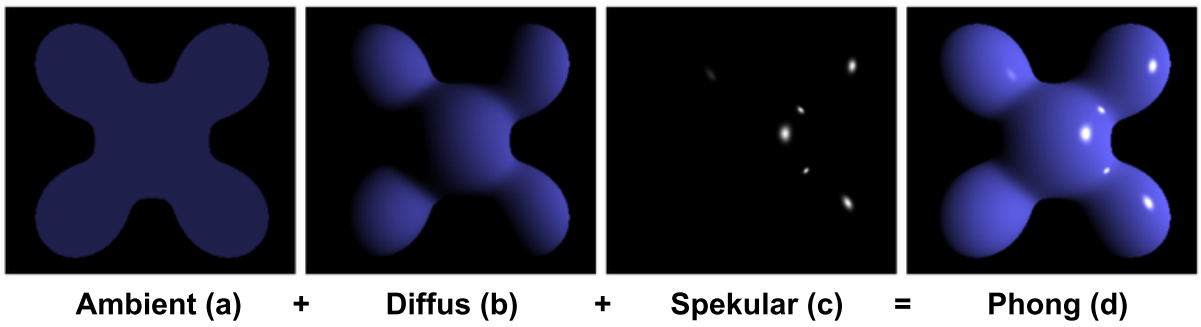

Abbildung 4.6: Komponenten des Phong-Beleuchtungsmodells[8]

Die ambiente Komponente des Modells ergibt sich aus einem für alle Punkte konstantem Beleuchtungswert, der Umgebungslicht annähern soll, und einem empirisch bestimmten Materialfaktor. Durch die Unabhängigkeit von dem Betrachtungswinkel und Einfallwinkel des Lichts führt die ambiente Komponente zu einer konstanten Beleuchtung der Objekte (Abbildung 4.6 a).

Bei der diffusen Komponente wird der Einfallswinkel des Lichts berücksichtigt. Dazu wird der Winkel zwischen dem Lichtstrahl und der Oberflächennormalen berechnet und der diffuse Beleuchtungsterm darauf basierend berechnet. Dadurch können der Lichtquelle abgewandte Oberflächen dunkler und ihr zugewandte Oberflächen heller dargestellt werden. Durch die Verarbeitung der Oberflächennormale wird mit der diffusen Komponente der plastische Eindruck eines Objekts simuliert (Abbildung 4.6 b).

Die spekulare Komponente setzt den Sichtstrahl mit dem Strahl des einfallenden Lichts in Relation. Dazu wird der Lichtstrahl an der Oberfläche reflektiert. Basierend auf dem Winkel zwischen diesem reflektierten Strahl und dem Strahl, der von der Oberfläche zum Betrachter verläuft, wird die spekulare Reflexion berechnet. Durch eine exponentielle Erhöhung des Wertes können spiegelnde Reflexionen auf der Oberfläche eines Objekts simuliert werden (Abbildung 4.6 c).

Die Summe der drei Komponenten resultiert in der Gesamtbeleuchtung eines Punktes (Abbildung 4.6 c). Mit Hilfe von Konstanten kann der Einfluss jeder Komponente abgeschwächt oder verstärkt werden, um verschiedene Materialien zu simulieren. Zum Beispiel könnte Holz mit einem sehr schwachen, und Metall eher mit einem höheren, spekularen Anteil angenähert werden.

[8] Abbildung von: `http://de.wikipedia.org/wiki/Phong-Beleuchtungsmodell`

Kapitel 5

Werkzeuge

Um allgemeine Berechnung auf der GPU auszuführen, wird eine Programmierschnittstelle benötigt, mit der die Daten übertragen und die Algorithmen implementiert werden können. Für die Anzeige eines vorberechneten Bildes wird ebenfalls eine entsprechende Schnittstelle benötigt. Bei der Wahl der verwendeten Werkzeugen stand die Plattformunabhängigkeit der Applikation im Vordergrund und so wurde von der Verwendung von DirectX (nur Microsoft-Windows Systeme) oder NVIDIA CUDA (nur NVIDIA GPUs) abgesehen und stattdessen auf die offenen Schnittstellen OpenGL sowie OpenCL zurückgegriffen.

5.1 OpenGL

Um auf einem PC ein von der Grafikkarte berechnetes Bild auszugeben, muss auf eine Grafik-Schnittstelle zurückgegriffen werden. Die simpelsten dieser Schnittstellen bieten vorgefertigte, rudimentäre Methoden an, um einfache Rastergrafik zu visualisieren. Um aufwändigere Berechnungen und Effekte zu realisieren, ist die Verwendung einer erweiterbaren Schnittstelle Pflicht. Die verbreitetsten dieser Schnittstellen sind Direct3D und OpenGL (**Open G**raphics **L**ibrary), die beide in den 90er Jahren erschienen. Während Direct3D auf das Betriebssystem Microsoft Windows angewiesen ist, kann OpenGL auf jedem System mit entsprechender Hardwareunterstützung genutzt werden.

Während OpenGL anfangs noch mit der festen Grafik-Pipeline arbeitete, wurde der Funktionsumfang stetig erweitert und seit der Version 3.0 ist im *core profile*[9] die Verwendung der festen Pipeline nicht mehr möglich und stattdessen muss die programmierbare Pipeline mit Fragment- und Vertex-Shadern benutzt werden. Der Funktionsumfang des *core profiles* kann über *Extensions* erweitert werden. Da OpenGL für diese Arbeit nur sehr minimalistisch verwendet wird, sind die neueren Funktionalitäten, die allesamt auf eine Verbesserung der Rastergrafik abzielen, nicht relevant.

[9] `http://www.opengl.org/wiki/Core_And_Compatibility_in_Contexts`

5.2 OpenCL

Neben der Entwicklung von 3D-Applikationen wurde die GPU, aufgrund ihrer hohen Anzahl an separaten Kernen, sehr früh bereits für parallele Algorithmen genutzt. Erste Applikationen (siehe [18] z. B. Kapitel 46: *Improved GPU Sorting*) arbeiteten mangels Alternativen sogar mit den Shadern der Grafikschnittstellen. Der Grafikkartenhersteller NVIDIA veröffentlichte 2007 mit CUDA (**C**ompute **U**nified **D**evice **A**rchitecture) eine Schnittstelle für allgemeine Berechnungen (GPGPU) auf NVIDIA GPUs (siehe [10]). Die Firma Apple veröffentlichte im Jahre 2008 die Schnittstelle OpenCL (**Open C**omputing **L**anguage), mit deren Hilfe plattformunabhängige, parallele Programme realisiert werden können (siehe Kapitel 1 in [7]). Kurze Zeit später wurde der erste Entwurf in Zusammenarbeit mit mehreren anderen Firmen[10] ausgearbeitet, zur Standardisierung bei der Khronos Group eingereicht und im Dezember 2008 als OpenCL 1.0 veröffentlicht.[11] Im Juni 2010 wurde mit OpenCL 1.1 die nächste Version veröffentlicht, die bis heute die aktuell unterstützte Version auf NVIDIA-GPUs ist. Im November 2011 folgte Version 1.2, die bereits von ATI-GPUs unterstützt wird. Die Version 2.0 wurde im November 2013 spezifiziert, wird aber bisher von keiner GPU unterstützt.[12]

Dem Ziel der Plattformunabhängigkeit wird OpenCL durch eine spezielle Architektur (siehe Abbildung 5.1) gerecht. In diesem System existiert genau ein Host, der mehrere Geräte (*devices*) verwaltet. Die Rolle eines Device kann zum Beispiel eine GPU, aber auch die CPU einnehmen, während Letztere immer als Host fungiert. Ein Device verfügt wiederum über mehrere, unabhängige Recheneinheiten (*compute unit*, CU), die bei Mehrkernprozessoren die verfügbaren Kerne darstellen. Jede dieser Recheneinheiten ist in ein oder mehrere ausführende Elemente (*processing element*, PE) unterteilt, die auf der untersten Ebene die tatsächlichen Berechnungen ausführen.

Durch den modularisierten Aufbau ist gewährleistet, dass OpenCL-Programme auf unterschiedlichster Hardware lauffähig sind. Für die beste Leistung muss ein Programm allerdings für eine spezifische Hardware optimiert werden. Des Weiteren kann ein Host mehrere Devices verwalten und die zu bearbeitenden Aufgaben auf diese verteilen.

[10] Alle Mitglieder der Khronos Group: `https://www.khronos.org/about`
[11] Revisionen aller Versionen: `https://www.khronos.org/registry/cl/`
[12] `http://www.khronos.org/conformance/adopters/conformant-products/`

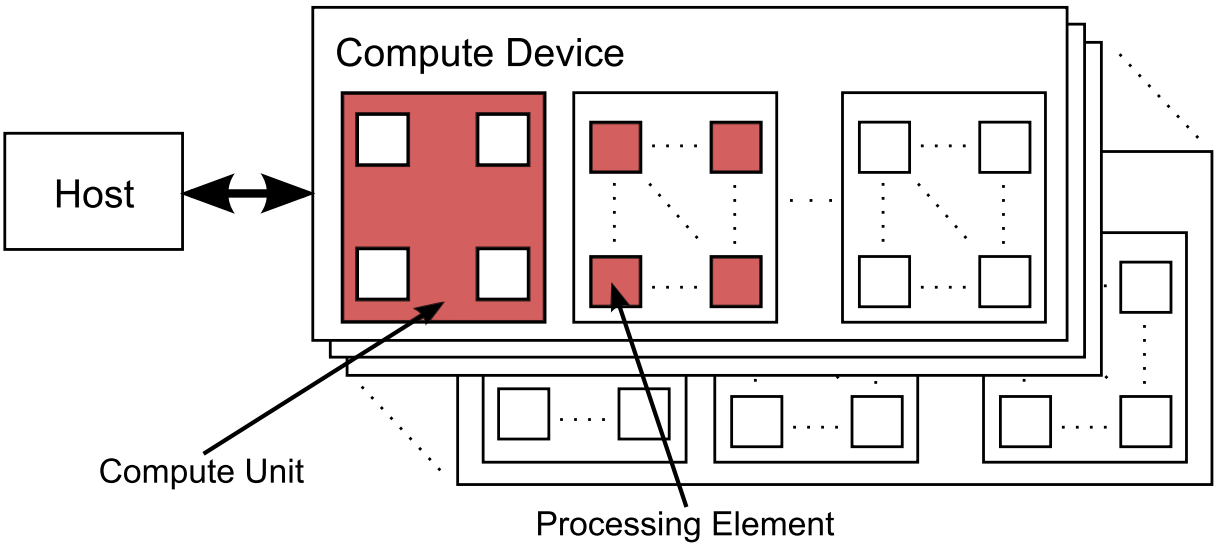

Abbildung 5.1: Schematische Darstellung einer OpenCL-Architektur[13]

Diese Aufgaben, die sogenannten Kernel, werden in OpenCL C programmiert und zur Laufzeit vom OpenCL-Compiler übersetzt um anschließend auf einem Device ausgeführt zu werden. OpenCL C basiert auf ISO C99[14] und verfügt über zusätzliche Funktionen und Datentypen für die parallelen Berechnungen. So erlaubt OpenCL C zum Beispiel die Verwendung von vektoriellen Datentypen sowie Datentypen für Bild-Daten. An einigen Stellen wurde die Sprache, aufgrund ihrer parallelen Auslegung, im Bezug auf C99 eingeschränkt. Deshalb müssen Arrays über eine feste Länge verfügen, Rekursion kann nicht verwendet werden und es existieren keine Zeiger auf Funktionen.

Um die Berechnungen der Kernel durchzuführen, verwendet OpenCL sogenannte Work-Items, die in einem ein- bis dreidimensionalen Gitter angeordnet und adressiert werden. Dabei muss der Nutzer die Dimension sowie die Größe in jeder Dimension beim Start eines Kernels angeben. Des Weiteren werden die Work-Items zu Work-Groups gruppiert, innerhalb derer ein schneller, gemeinsamer Speicher genutzt und die Work-Items synchronisiert werden können. Wenn die einheitliche Größe aller Work-Groups nicht vom Nutzer angegeben wird, wählt OpenCL eine passende (aber nicht unbedingt optimale) Größe aus.

[13] Abbildung von: `http://developer.amd.com/resources/documentation-articles/articles-whitepapers/opencl-and-the-amd-app-sdk-v2-4/`

[14] `http://www.open-std.org/jtc1/sc22/wg14/`

Abbildung 5.2 veranschaulicht diesen Aufbau:

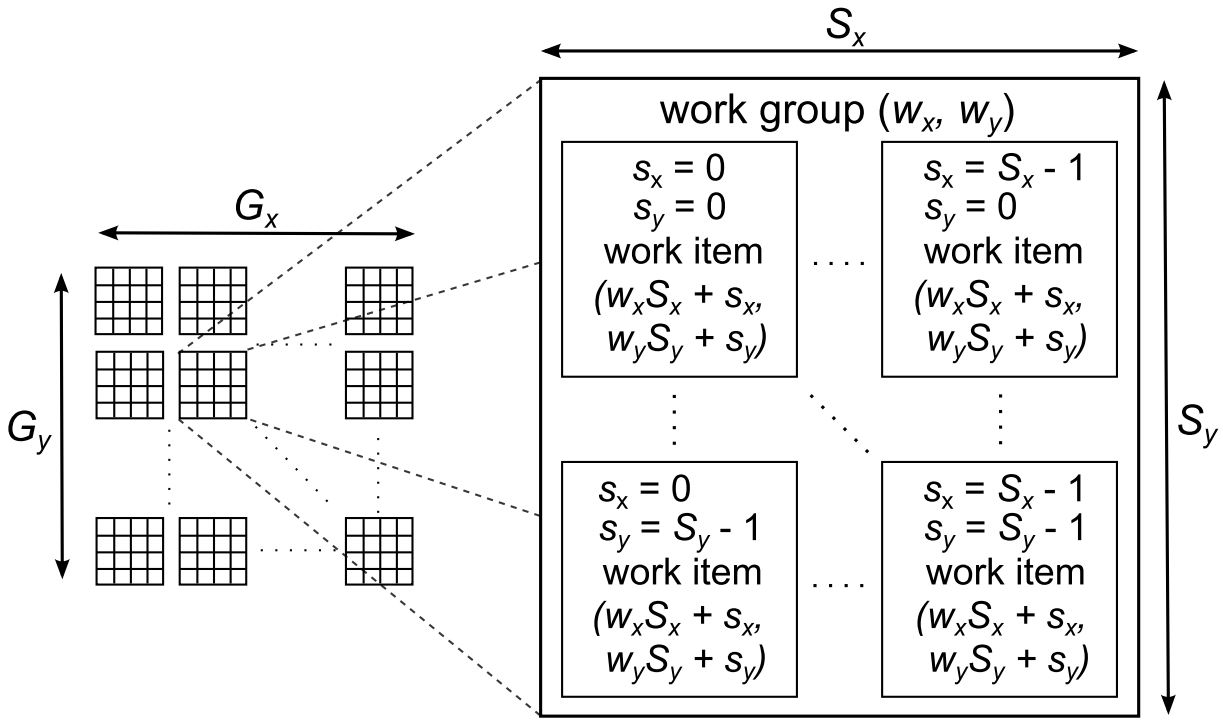

Abbildung 5.2: Work-Items und Work-Groups in OpenCL[15]

Zusätzlich zur modularen Architektur unterstützt OpenCL ein Speichermodell mit mehreren verschiedenen Arten von Speicher, die sich in ihrer Geschwindigkeit und Zugriffsrechten unterscheiden. Aufsteigend nach der Geschwindigkeit aufgelistet sind das:

- **host memory:** Der normale Arbeitsspeicher des Hostprozesses. Nur der Host kann darauf zugreifen.
- **global memory:** Gemeinsamer Speicher der OpenCL-Kernel. Für jede Kernel-Instanz sichtbar.
- **constant memory:** Schnellerer Spezialfall des globalen Speichers, auf den jede Kernel-Instanz nur lesenden Zugriff hat.
- **local memory:** Sehr schneller, aber kleiner Speicher, auf den nur die Kernel-Instanzen einer Work-Group zugreifen können. Insbesondere hat jede Work-Group ihren eigenen lokalen Speicher.
- **private memory:** Der schnellste Speicher, auf den nur eine Kernel-Instanz Zugriff hat.

[15] Abbildung von: `http://developer.amd.com/resources/documentation-articles/articles-whitepapers/opencl-and-the-amd-app-sdk-v2-4/`

Die folgende Abbildung 5.3 veranschaulicht dieses Speichermodell:

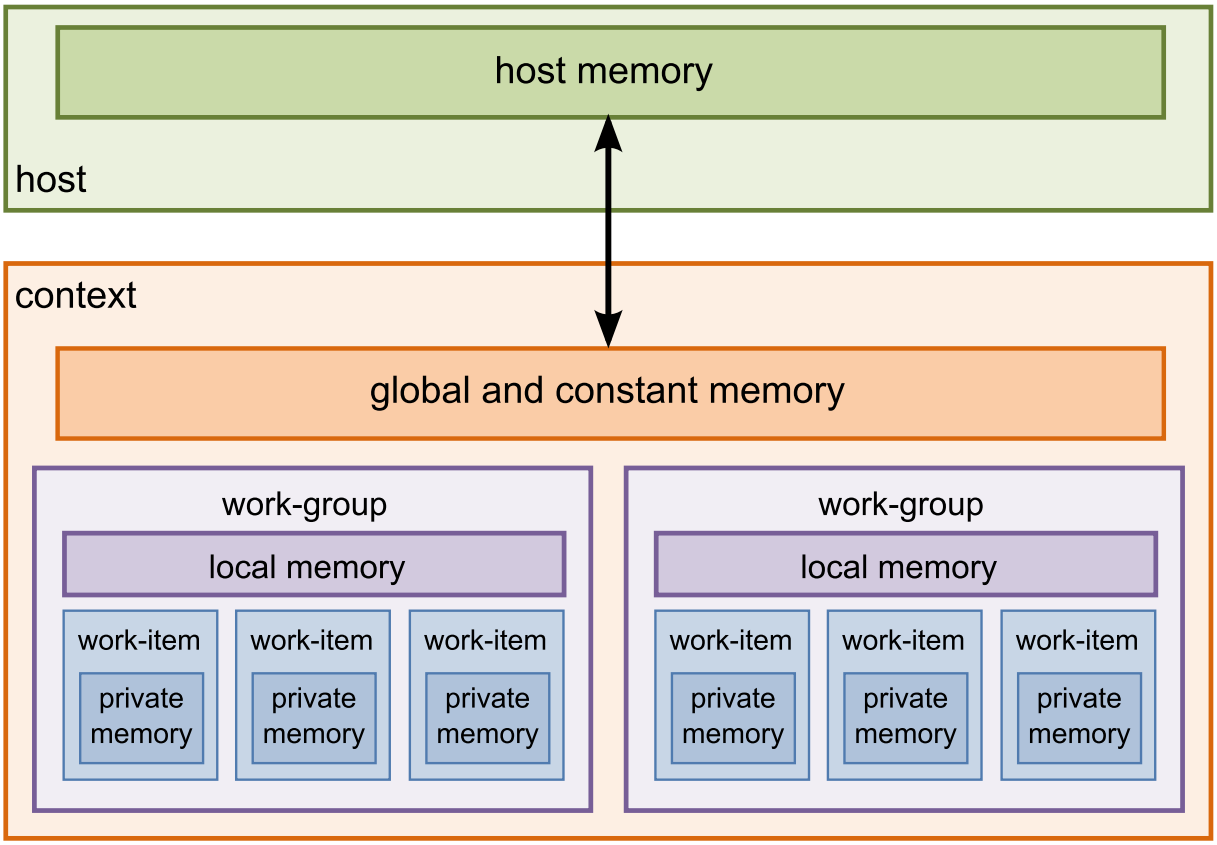

Abbildung 5.3: OpenCL Speichermodell[16]

Allgemein gilt: Je schneller der Speicher ist, desto kleiner beziehungsweise eingeschränkter ist er auch. Der schnelle private Speicher wird automatisch benutzt, wenn Variablen (grundsätzlich privat) deklariert werden. Allerdings gibt es auch für privaten Speicher eine Obergrenze, die nicht überschritten werden sollte, da sonst der private Speicher auf den sehr viel langsameren Arbeitsspeicher ausgelagert wird. Der globale Speicher wird meist genutzt, um globale Daten, die für alle Kernel-Instanzen gleich sind, an Selbige zu übertragen. Dieser Speicher ist durch den Arbeitsspeicher des Devices limitiert (aktuelle, hochwertige GPUs verfügen meist über 2-6 GB) und entsprechend langsam (vgl. Kapitel 5 in [10]). Der besondere lokale Speicher kann einige parallele Anwendungen in der Theorie beschleunigen, indem wiederverwendete Daten nur ein mal aus dem globalen Speicher gelesen und im lokalen Speicher abgelegt werden (*caching*).

[16] Abbildung von: `http://de.wikipedia.org/wiki/OpenCL`

Kapitel 6

Umsetzung

In diesem Kapitel wird die Umsetzung der direkten, veränderbaren Terrain-Visualisierung und die dafür verwendeten Techniken beschrieben. Für die direkte Visualisierung der Daten wird Isosurface Rendering, das mittels OpenCL auf der Grafikkarte berechnet wird, genutzt. Hierfür behandelt der Abschnitt das Speicher-Layout der Volumendaten, die konkreten Algorithmen des Isosurface Renderings, die Umsetzung der interaktiven Veränderbarkeit sowie weitere Techniken, die die visuelle Qualität oder die Performanz der Applikation erhöhen.

6.1 Erstellung der Terrain-Daten

Isosurface Rendering wird genutzt, um Oberflächen-Eigenschaften aus Volumendaten zu extrahieren. Obwohl diese Volumendaten theoretisch unendlich viele Werte enthalten, liegen sie als diskrete, meist gleichmäßige, Punktwolken vor. Ein Volumen wird also durch sehr viele, gleichmäßig in alle drei Dimensionen verteilte, Dichte-Stichproben ([0,1]) repräsentiert. Um die Dichte zwischen diesen Stichproben zu errechnen, bietet sich eine trilineare Interpolation an (siehe Abschnitt 4.2).

Um diese kubischen Volumen als Grundlage mit Daten zu füllen, die einem Terrain nahe kommen, wird der von NVIDIA vorgestellte Algorithmus (Kapitel 1 in [15]) verwendet. Der Algorithmus basiert auf der Verwendung von mehreren kleinen dreidimensionalen Noise-Feldern, aus denen anhand der Weltkoordinate des zu berechnenden Punktes *gesampled* wird. Durch Addition von hohen Amplituden und niedrigen Frequenzen kann ein Terrain-ähnliches Volumen erzeugt werden.

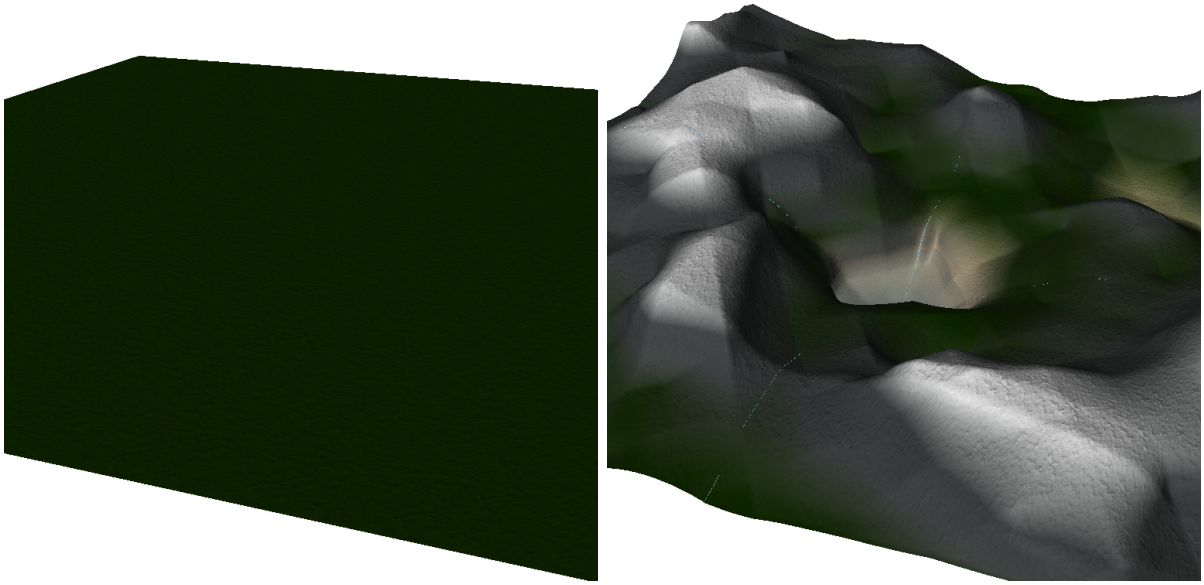

Abbildung 6.1: Links: Flaches Ausgangs-Terrain - Rechts: Terrain mit hohen Amplituden und niedrigen Frequenzen

Als Ausgangspunkt dient die negative Höhe der Weltkoordinate, sodass Werte unter dem Nullpunkt mit einer positiven und Werte darüber mit einer negativen Dichte starten. In einem letzten Normalisierungs-Schritt werden diese negativen Werte auf Null (keine Dichte) übertragen. Mit einem Iso-Wert von 0.5 entsteht ein planares Terrain auf Höhe Null der Weltkoordinaten (Abbildung 6.1, links). Anschließend werden für jeden Volumenpunkt mehrere Stichproben mit verschiedenen Parametern aus den Noise-Feldern hinzu addiert. Stichproben mit hoher Amplitude und niedriger Frequenz resultieren in Bergen und Tälern, die selten auftreten (Abbildung 6.1, rechts). Des Weiteren ist ein Muster auf dem Terrain zu erkennen, das durch die geringe Auflösung der Noise-Felder entsteht.

Durch das Hinzufügen von mehreren Stichproben mit hoher Frequenz und niedriger Amplitude werden weitere Details hinzugefügt und die Musterbildung verringert (Abbildung 6.2, links). Für das finale Terrain werden weitere Stichproben mit hohen sowie niedrigen Frequenzen und Amplituden hinzugefügt, die auf rotierten Weltkoordinaten basieren. Als Resultat sind keine Muster mehr erkennbar und dem Terrain werden Details, wie Höhlen und Überhänge, hinzugefügt (Abbildung 6.2, rechts).

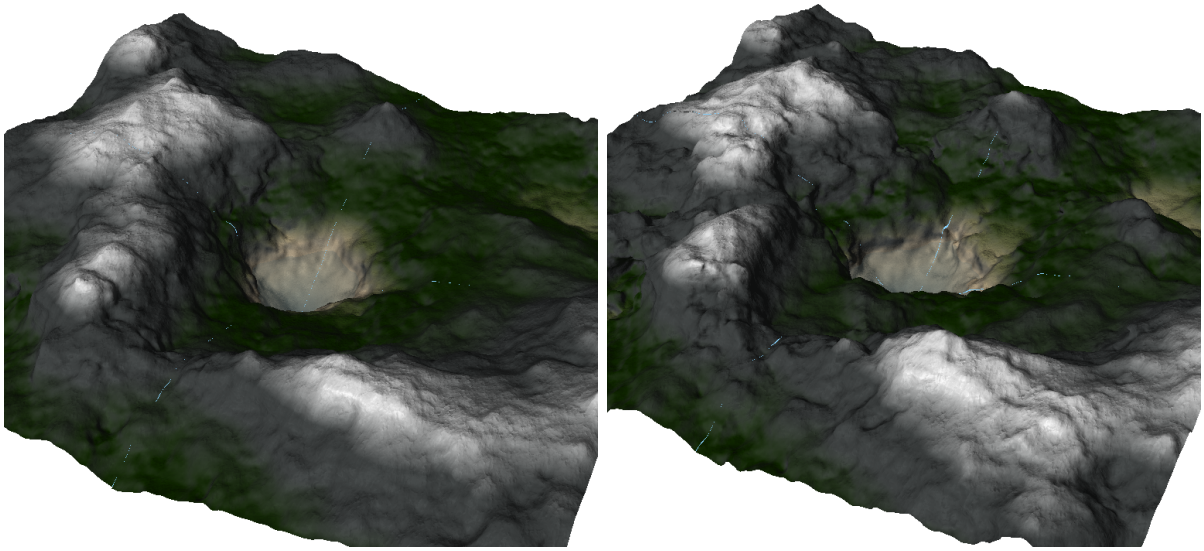

Abbildung 6.2: Verschiedene Amplituden und Frequenzen generieren ein Terrain mit weiteren Details

6.2 Optimierungen

Um die so erstellten Volumendatensätze in Echtzeit visualisieren zu können, bedarf es effizienter Algorithmen für die Volumen-Traversierung und Schnittpunktberechnung. Während die entsprechende Theorie in den Abschnitten 4.2 und 4.3 vorgestellt wurde, werden in diesem Abschnitt die verwendeten Algorithmen beziehungsweise Methoden erläutert.

6.2.1 Aufteilung des Terrains in mehrere Datenblöcke

Um ein großes Terrain mit Volumendaten zu erstellen, kann das Volumen in jeder Dimension immer weiter vergrößern werden. Dies wird aufgrund des kubischen Speicherbedarfs allerdings sehr schnell die Größe aller gängigen Datenstrukturen überschreiten. Um diesen kubischen Speicherbedarf zu vermeiden, wird das Terrain lediglich in der Breite sowie Tiefe aber nicht in der Höhe vergrößert. Diese eingeschränkte Auflösung in der Höhe limitiert zwar die Berge und Täler des Terrains, die subjektiven Einschränkungen sind allerdings relativ gering.

Des Weiteren wird das *teile und herrsche*-Prinzip angewandt: Das Gesamtterrain besteht aus mehreren kubischen Volumen mit jeweils 128^3 Voxeln (also 129^3 Datenpunkten). Von diesen Volumen werden in Breite und Tiefe mehrere nebeneinander erzeugt und einzeln visualisiert.

Abbildung 6.3 zeigt einen Screenshot aus der Vogelperspektive, bei der die Aufteilung in die einzelnen Volumen deutlich wird.

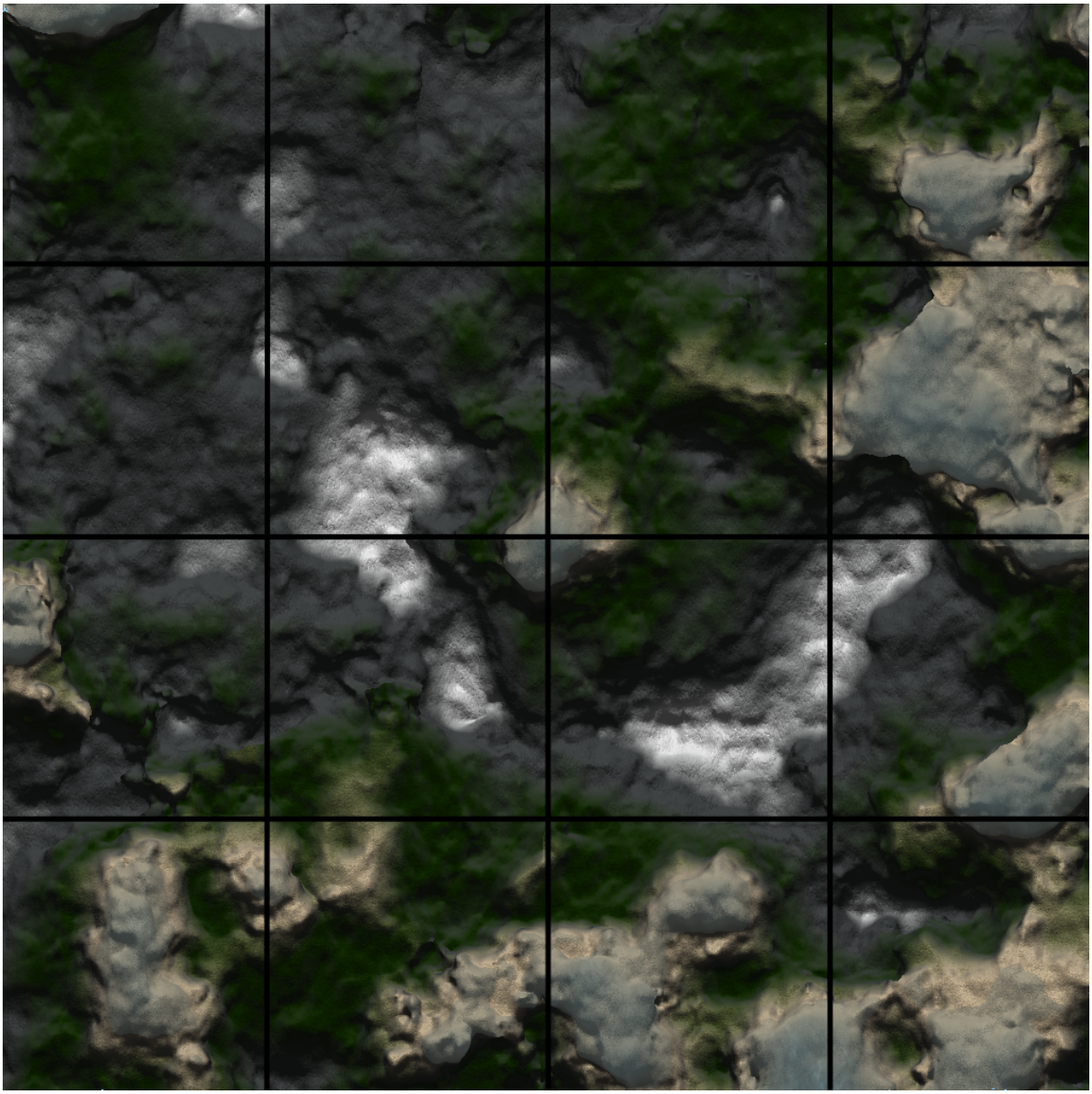

Abbildung 6.3: Aufteilung des Terrains in Datenblöcke

6.2.2 Frustum

Die so erstellten Datenblöcke, mit jeweils 128^3 Voxeln, werden im Rendering-Vorgang von jeweils einem gestarteten OpenCL-Kernel verarbeitet. Um unnötige Berechnungen weitestgehend zu vermeiden, wurde ein *frustum culling* implementiert, um zu gewährleisten, dass nur die Datenblöcke verarbeitet werden, die gerade in dem vom Nutzer gewählten Kamera-Ausschnitt liegen. Abbildung 6.4 veranschaulicht dieses Verfahren.

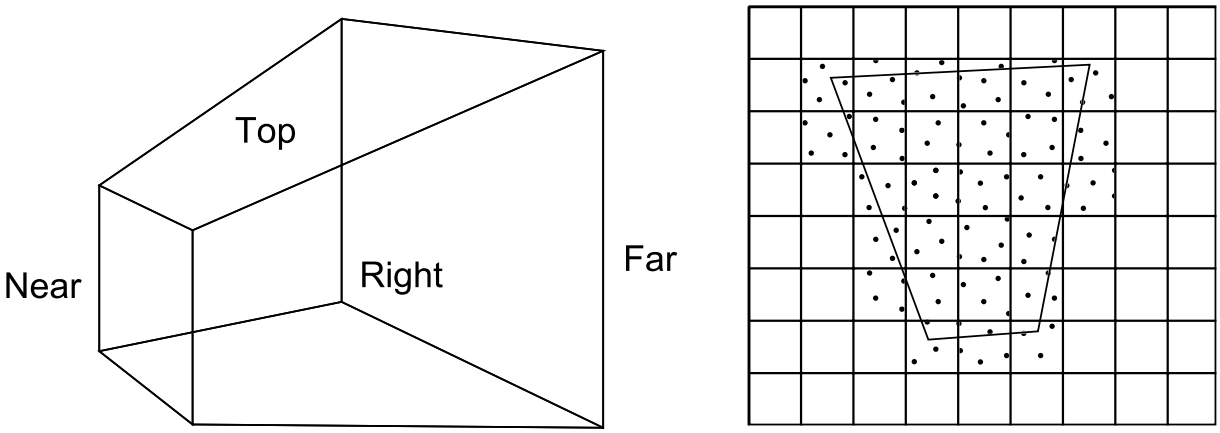

Abbildung 6.4: Links: Ein 3D-Frustum - Rechts: 2D-Schema des *frustum cullings*

Des Weiteren werden die Datenblöcke bei jeder Kamerabewegung oder -drehung aufsteigend nach dem Abstand ihres Mittelpunktes zur aktuellen Kameraposition sortiert und in dieser Reihenfolge verarbeitet. Zusätzlich wird für jedes zu berechnende Pixel des Bildes ein Bit gespeichert, dass angibt, ob diesem Pixel bereits ein Farbwert zugewiesen wurde. Sollte dieses Bit in einer Kernel-Instanz bereits von einem, näher am Betrachter liegendem, vorher verarbeiteten Datenblock gesetzt sein, so führt diese Instanz keine weiteren Berechnungen aus und kann für andere Berechnungen genutzt werden. Durch diese beiden Verfahren ist sichergestellt, dass das Resultat korrekt ist (nahe Schnittpunkte können nicht durch dahinter liegende überschrieben werden) und keine unnötigen Berechnungen durchgeführt werden.

6.2.3 Mögliche Datentypen für die Datenblöcke

Zur Speicherung der kubischen Volumen bietet OpenCL zwei grundsätzliche Möglichkeiten: Die Verwendung von Buffern oder dreidimensionaler Texturen. Buffer sind konsekutive Datenblöcke im Speicher und obwohl die Daten der Texturen letztendlich auch konsekutiv im Speicher liegen, verfügt die Hardware über spezielle Textur-Einheiten, die zum Beispiel automatische Interpolation zwischen den Daten ermöglichen. Bei beispielhaften Implementationen war ein einzelner Lesezugriff auf einen Buffer meist schneller als der gleiche Zugriff auf einer gleich großen Textur. Des Weiteren arbeiten Buffer sehr schnell, wenn mehrere, konsekutiv im Speicher liegende Daten geladen werden müssen. Die acht Datenwerte eines Voxels liegen in einem dreidimensionalen

Buffer aber nicht konsekutiv im Speicher und so muss von diesem Best-Case Fall des Buffers abgesehen und ein genauerer Vergleich der Datenstrukturen angestellt werden.

Um die beiden Speichermöglichkeiten besser vergleichen zu können, wurde ein minimaler Test-Kernel geschrieben, der in einem dreidimensionale Volumen die Voxel sequentiell abschreitet und die Daten (jeweils acht Float-Werte) dieser Voxel aus dem Speicher liest. Dabei handelt es sich um einen zweidimensionalen Kernel, dessen Dimension der Breite und Höhe des Volumens entspricht. Im Kernel wird mit einer Schleife über die dritte Dimension iteriert, sodass jedes Work-Item auf die Daten einer Voxel-Reihe zugreift. Damit der Compiler die Speicherzugriffe beim Optimieren nicht entfernt, wird nach jedem Voxel-Zugriff eine simple Berechnung mit den Daten durchgeführt. Der folgenden Pseudocode veranschaulicht diesen Test-Kernel:

Algorithmus 7 : Pseudocode des Test-Kernels

```
Data : volumeData: Buffer oder Textur)
Data : volumeDimension: Die Dimension in Höhe, Breite und Tiefe
/* work-IDs (Koordinate der Dimension)                                    */
i ← global id(0);
j ← global id(1);
for int k = 0; k < volumeDimension; k++ do
    /* die acht Dichte-Werte des Voxels auslesen                          */
    for int u = 0; u <= 1; u++ do
        for int v = 0; v <= 1; v++ do
            for int w = 0; w <= 1; w++ do
                /* Daten aus Textur oder Buffer lesen - die Reihenfolge (hier
                   i,j,k) wird zum Testen verändert                       */
                dataValue ← readData(volumeData, i+u, j+v, k+w);
                eine Berechnung mit dataValue ausführen;
            end
        end
    end
end
```

Um das unterschiedliche Verhalten der beiden Datenstrukturen zu verdeutlichen, wurde die Art der dreidimensionalen Adressierung variiert. Die folgende Tabelle veranschaulicht die gemessenen Kernel-Laufzeiten in Abhängigkeit der verwendeten Datenstruktur und Adressierungsart.

Tabelle 6.1: Laufzeit (in ms) des Test-Kernels

Adressierung	**Buffer**	**Textur**
i,j,k	112	4,8
i,k,j	101	4,7
j,i,k	102	10,1
j,k,i	2,5	45
k,i,j	91	11,1
k,j,i	2,5	35,8

Der Buffer ist der Textur in genau zwei Fällen überlegen und die anderen vier Fälle weisen sehr viel schlechtere Performanz auf. Die Textur hat auch solche Worst-Case Fälle und zwar genau dort, wo der Buffer besonders schnell ist - allerdings sind diese Fälle schneller als die Worst-Case Fälle des Buffers.

Dieses Verhalten entspricht der Architektur einer GPU, die sehr schnelles *caching* und *filtering* bei Texturen ermöglicht (siehe Kapitel 5 in [7]). Bei dem Buffer geschieht diese Beschleunigung nur dann, wenn die Daten konsekutiv im Speicher liegen und in diesem Fall ist er der Textur überlegen. Im Normalfall wird der Traversierungs-Algorithmus aber nicht genau diese schnelle Zugriffsart auf einen Buffer ermöglichen, sodass die Verwendung von Texturen in einer besseren Performanz resultiert.

Die beispielhafte Implementation eines Caches im local memory resultierte nicht in dem gewünschten Performanz-Gewinn weil das Device den lokalen Speicher vermutlich bereits automatisch zum *caching* verwendet und je mehr lokalen Speicher der Entwickler selbst verwendet, desto weniger hat die GPU dafür zur Verfügung.

6.2.4 Schnittpunktberechnung

Bei der Berechnung des Schnittpunktes innerhalb eines Voxels wurde letztendlich von der Verwendung eines exakten Algorithmus abgesehen und stattdessen Neubauers Ansatz verwendet. Die analytischen Verfahren scheiden aufgrund ihres hohen Aufwands aus und Marmitts Ansatz ist durch das angesprochenen Rauschen ohne Abwandlungen nicht geeignet. Wie bereits erwähnt sind die Fehlerfälle von Neubauers Ansatz nur dann relevant, wenn innerhalb eines Voxels mehrere Oberflächenübergänge enthalten sind. Das Terrain müsste also sehr schnelle Änderungen enthalten oder die Voxel-Auflösung müsste zu gering gewählt werden, damit diese Fehlerfälle auftreten. Beidem wurde gezielt entgegengewirkt, sodass diese Fälle in dem finalem Terrain sehr selten auftreten und somit den subjektiven optischen Eindruck nicht nennenswert verschlechtern.

6.2.5 Traversierungs-Algorithmus

In Abschnitt 4.3 wurde neben der Verwendung von komplexen Datenstrukturen (z. B. Bäumen) ein Voxel-Stepping-Algorithmus vorgestellt, der sich speziell für den Gitter-basierten Aufbau eines Volumen eignet.

Dieser Algorithmus hat ein Worst-Case-Verhalten in dem Fall, dass die Strahlen sehr viele Voxel abschreiten müssen bevor sie in einer Oberfläche terminieren beziehungsweise auch dann, wenn sie ohne Schnittpunkt durch das komplette Volumen verfolgt werden müssen. Das primäre Problem ist also eine schnelle Behandlung des leeren Raumes. Um die langsamen Zugriffe auf den globalen Speicher der Volumendaten zu minimieren, wurde der Algorithmus deshalb um zwei weitere Schritte erweitert. Diese Erweiterungen basieren beide auf den vorberechneten Informationen, in welchen Voxeln bei einem bestimmten gesuchten Iso-Wert überhaupt ein Oberflächenübergang vorhanden sein kann. Ein solcher Übergang kann nur dann stattfinden, wenn:

$$\min(\textit{Dichte-Werte des Voxels}) \leqq \textit{Iso-Wert} \leqq \max(\textit{Dichte-Werte des Voxels})$$

gilt. So kann für jeden Voxel mit einem Bit gespeichert werden, ob es einen Oberflächenübergang enthält, oder nicht. Bevor also die langsamen acht Speicherzugriffe für die Dichte-Werte eines Voxels erfolgen, wird vorab das vorberechnete Bit überprüft und die Dichte-Werte werden nur bei gesetztem Bit ausgelesen. Trotz zusätzlichem Speicherbedarfs sowie einem zusätzlichen Zugriff bei gesetztem Bit resultiert diese Erweiterung in einer deutlichen Beschleunigung durch das schnellere Überspringen des leeren Raumes (siehe Tabelle 6.2).

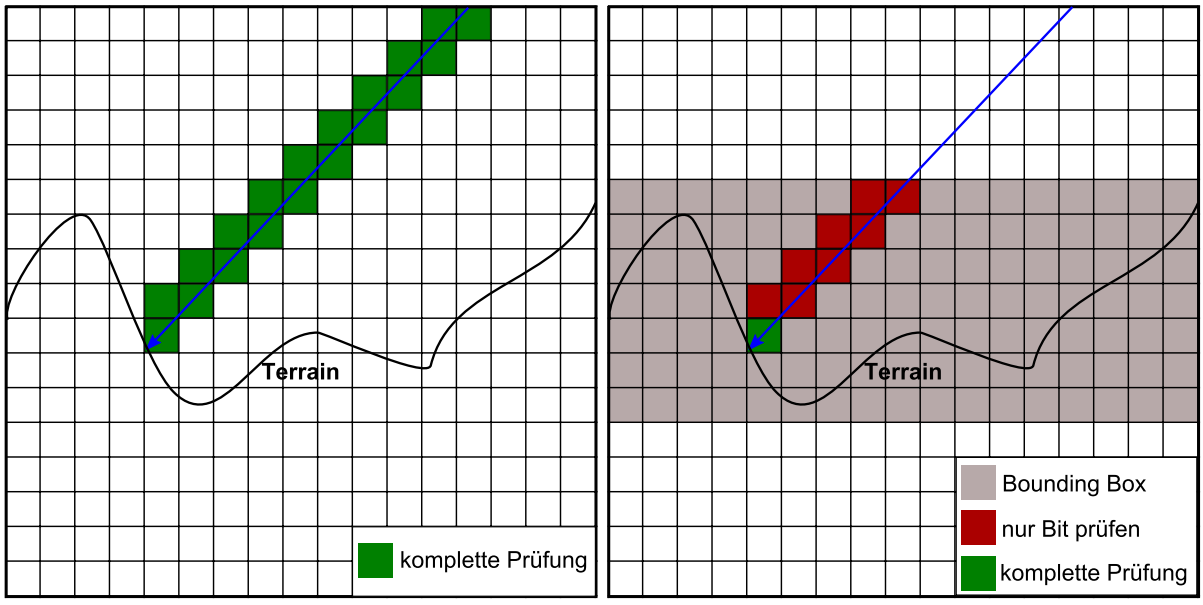

Abbildung 6.5: Beispiele des Voxel-Stepping Algorithmus ohne (links) und mit Erweiterungen (rechts)

Die zweite Erweiterung dient der Verfeinerung der um das Volumen liegenden Bounding Box, die jeder Strahl durchstoßen muss um weiter behandelt zu werden. Anstatt diese über das komplette Volumen zu legen, wird die Höhe der Bounding Box anhand des in der Höhe minimalen und maximalem Voxels mit möglichem Oberflächenübergang angepasst. Die Bounding Box wird also auf den Bereich, in dem Oberflächenübergänge stattfinden können, eingegrenzt, was in den meisten Fällen in einer weiteren Performanz-Verbesserung (siehe Tabelle 6.2) resultiert.

Tabelle 6.2: Performanz (in FPS) des Traversierungs-Algorithmus auf einem Testsytem (NVIDIA GTX 670 und Intel Core i5 2500K). Die prozentualen Veränderungen beziehen sich jeweils auf den Algorithmus ohne Erweiterungen. Abbildung 6.6 zeigt die drei Szenen.

Szene	**ohne Erweiterungen**	**Bounding Box**	**Bit Check**	**Kombination**
Nahaufnahme	94	94 (+0%)	113 (+20%)	113 (+20%)
Vogelperspektive	49	55 (+12%)	55 (+12%)	56 (+14%)
Überblick	16	22 (+37%)	29 (+81%)	30 (+87%)

Der vorgestellte Algorithmus mit den beiden Erweiterungen resultiert in einer interaktiven Darstellung des Terrains bei akzeptablen Kosten durch zusätzlichen Speicherbedarf und zusätzlicher Änderung der Datenstruktur bei Veränderung des Terrains. Die anderen vorgestellten Datenstrukturen erreichen wahrscheinlich eine ähnliche bis leicht bessere Performanz, haben aber den Nachteil des sehr hohen Aufwands bei Änderungen des Terrains. Beispielhafte Implementierung der Ansätze von Horn et al. [8] resultierten in einigen Fällen (viel leerer Raum) in leicht besserer Performanz als der Voxel-Stepping-Algorithmus mit den genannten Erweiterungen aber in anderen Fällen (Kamera nahe am Terrain) auch in schlechterer Performanz. Aufgrund dieser ähnlichen Ergebnisse und der viel aufwändigeren Umsetzung der Veränderbarkeit des Terrains wird von der Verwendung komplexerer Datenstrukturen abgesehen und stattdessen der in diesem Abschnitt vorgestellte Algorithmus verwendet.

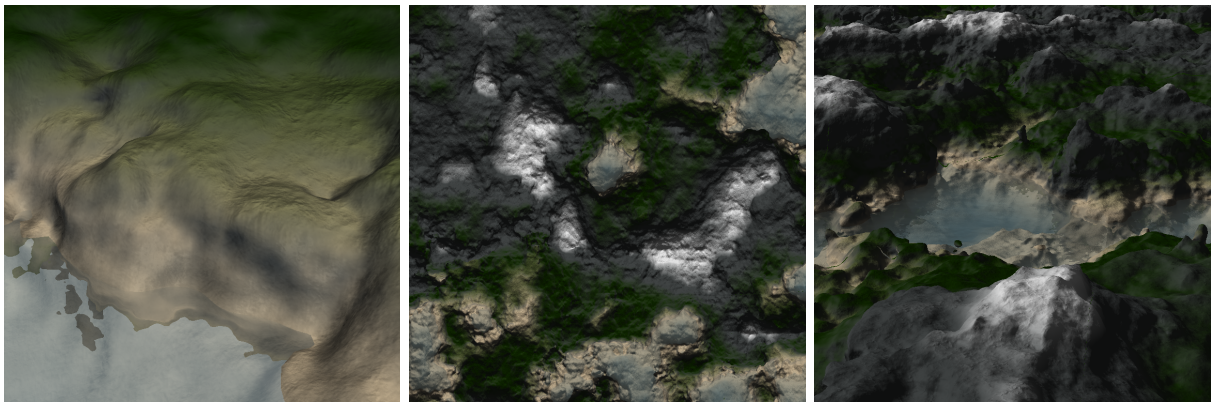

Abbildung 6.6: Ausgewählte Szenen für den Performanz-Test: Ein Nahaufnahme (links), eine Vogelperspektive (Mitte) und ein Blick über das Terrain (rechts).

6.3 Visualisierung

Wie bereits im Abschnitt 3.1 erwähnt, geht es beim Raytracing letztendlich darum, jedem Pixel des Bildes eine Farbe zuzuweisen. Deshalb ist die Grundlage der Visualisierung eines Terrains die Material-Farbe eines Schnittpunktes mit diesem Terrain. Anstatt für jeden Punkt des Volumens eine Materialeigenschaft zu speichern, wird die entsprechende Farbe anhand der Höhe linear interpoliert. Dies geschieht über eine eindimensionale Textur, die aus RGB-Werten für die Visualisierung bestimmter Materialien besteht. Diese simple Interpolation anhand der Höhe wird durch eine weitere Technik (siehe Abschnitt 6.3.3) angepasst.

6.3.1 Interpolation der Oberflächen-Normalen

Damit ein plastischer Eindruck des Terrains entsteht, muss die ermittelte Farbe des Schnittpunktes beleuchtet werden. Eine Grundlage der meisten Beleuchtungsmodelle ist die Oberflächen-Normale. Diese kann bei Volumendaten aus dem lokalen Gradienten gebildet werden:

Algorithmus 8 : Oberflächen-Normale durch lokalen Gradienten bestimmen

Data : x,y,z: Die Einheitskoordinaten des Punktes
Data : volumeData: Das dreidimensionale Volumen
Result : normal: Die berechnete Oberflächen-Normale des Punktes
float3 normal = (0, 0, 0, 0);
float e = 0.01;
/* Berechne Gradienten */
normal.x = readData(volumeData, x-e, y, z);
normal.x -= readData(volumeData, x+e, y, z);

normal.y = readData(volumeData, x, y-e, z);
normal.y -= readData(volumeData, x, y+e, z);

normal.z = readData(volumeData, x, y, z-e);
normal.z -= readData(volumeData, x, y, z+e);

/* Normalisiere Ergebnis */
return normalize(normal);

Alternativ zu diesem Verfahren, das auf sechs Zugriffen auf die Textur basiert, könnte die Normale durch den lokalen Gradienten innerhalb eines Voxels, also basierend auf den, bereits aus der Textur gelesenen, acht Dichte-Werten, umgesetzt werden. Durch die Verwendung dieses lokalen Gradienten innerhalb der einzelnen Voxel kommt es allerdings zu Unstetigkeiten der Normalen an den Übergängen zwischen den Voxeln, sodass diese durch die Beleuchtung sichtbar werden (siehe Abbildung 6.7). In der Abbildung sind die Unstetigkeiten der Normalen an den Voxel-Übergängen deutlich sichtbar. Deshalb wird die in Algorithmus 8 beschriebene (langsamere aber qualitativ bessere) Methode genutzt.

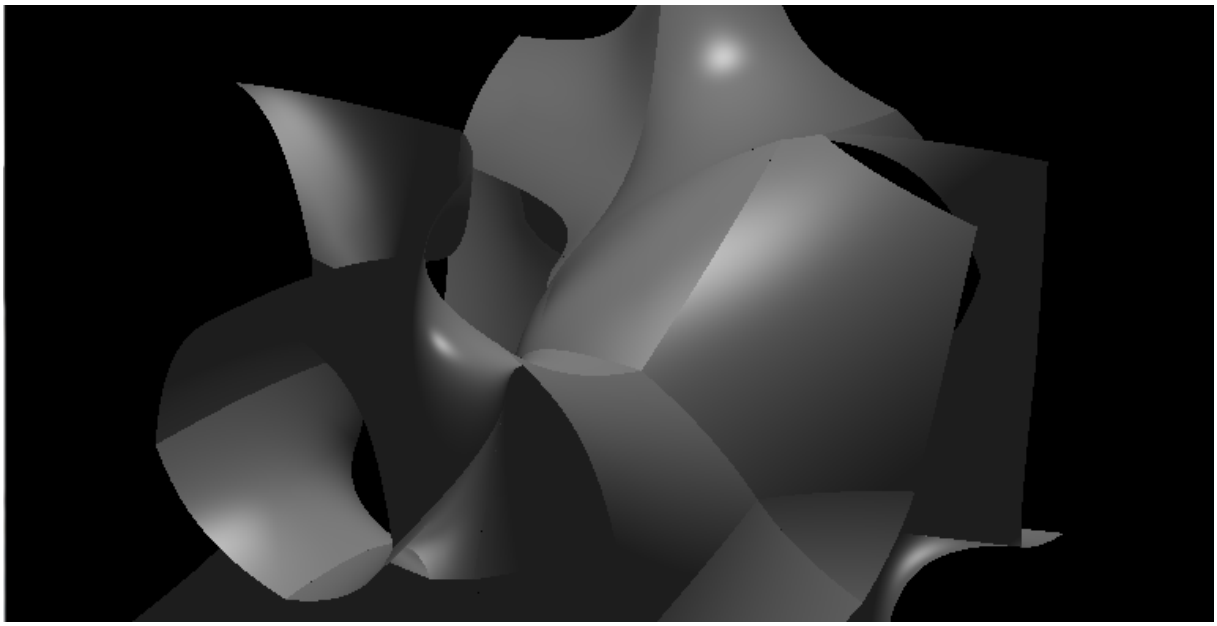

Abbildung 6.7: Visualisierung eines kleinen Volumens mit zufälligen Dichte-Werten und innerhalb der Voxel interpolierten Normalen

6.3.2 Beleuchtung

Mit der Oberflächenfarbe, der Normalen und einer dynamischen Lichtquelle, die eine Sonneneinstrahlung beispielhaft simulieren soll, kann jeder gefundene Schnittpunkt beleuchtet werden. Als Beleuchtungsmodell wird dabei das lokale Phong-Modell verwendet. Lokale Beleuchtungsmodelle sind schnell berechenbar weil sie globale Effekte der Beleuchtung ignorieren. So werden zum Beispiel auch einfache Schatten durch blockierende Objekte zwischen dem zu beleuchtenden Punkt und der Lichtquelle nicht beachtet.

Als Erweiterung der lokalen Beleuchtung ermöglicht der Raytracing-Ansatz die einfache Umsetzung direkter Schatten. Dafür muss lediglich ein Strahl vom gefundenen Schnittpunkt zur aktuellen Position der Lichtquelle erstellt und durch das Volumen verfolgt werden. Wenn dieser Strahl auf eine Oberfläche trifft, wird die Verfolgung abgebrochen und die Beleuchtung des Schnittpunktes abgedunkelt da sich dieser im Schatten befindet.

6.3.3 Veränderung der Normalen durch *triplanar texture mapping*

Eine weit verbreitete Technik zur qualitativen Erhöhung der optischen Qualität einer dreidimensionalen Szene ist das Bump-Mapping. Diese Technik macht es möglich, mithilfe von einfachen Texturen bei Oberflächen mit wenigen Details den visuellen Eindruck höherer Details zu erwecken. Dabei werden die Oberflächen-Normalen anhand der Informationen einer Bump-Textur leicht angepasst und können so mit entsprechender Beleuchtung zu besagtem visuellen Eindruck höherer Details führen. Da diese Illusion des Detailgrads nicht in der Objektgeometrie vorhanden ist, ist der Aufwand der Berechnung verhältnismäßig gering. Allerdings hat das Verfahren den Nachteil, dass die Illusion bei flachen Betrachtungswinkeln aufgrund der nicht vorhandenen Geometrie zusammenbricht und keine Schatten damit berechnet werden können.

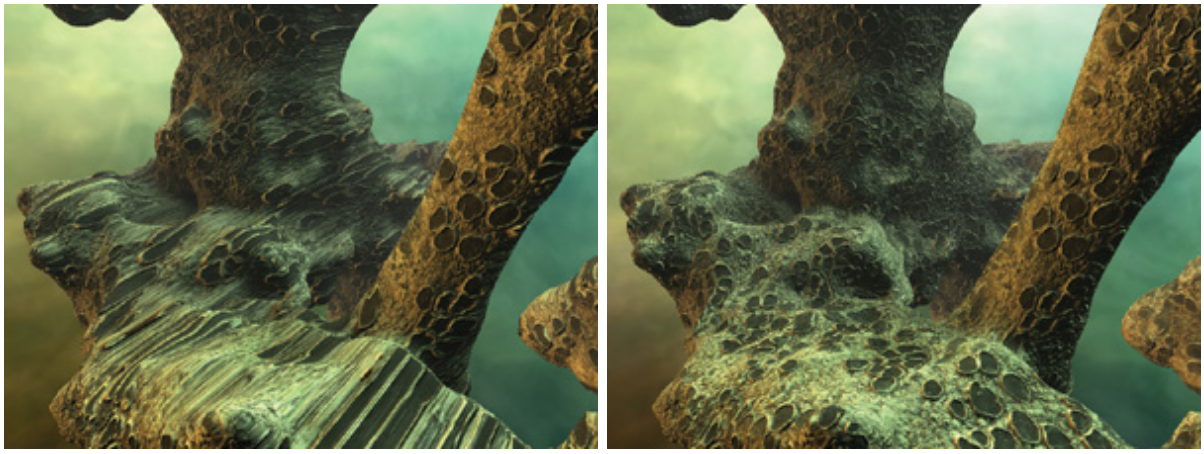

Abbildung 6.8: Die planare Projektion (links) einer Textur führt zu Streckungen der Textur, die beim *triplanar texture mapping* (rechts) nicht auftreten.[17]

Das klassische *bumpmapping*-Verfahren sieht die Verwendung von zweidimensionalen Texturen für zweidimensionale Flächen vor und führt bei höheren Dimensionen zu eine Verzerrung der Textur. Um ein dreidimensionales Terrain mithilfe von wenigen zweidimensionalen Texturen zu versehen, muss die Technik also leicht angepasst werden. Geiss (Kapitel 1 in [15]) schlägt die Verwendung von *triplanar texturing* vor, bei dem nicht nur eine Textur anhand einer Dimension planar projiziert wird. Stattdessen wird für jede Dimension eine Textur und eine Kombination der drei einzelnen Projektionen verwendet. Für diese Kombination wird jede Dimension mit einem Gewicht versehen, sodass die Dimension, die die geringste Verzerrung der Textur verursacht, das höchste Gewicht erhält. Die Gewichte werden anhand der Oberflächen-Normalen ausgerechnet. Algorithmus 9 zeigt die Umsetzung des Verfahrens in der Applikation:

[17] Abbildungen aus [15]

Algorithmus 9 : Triplanar Texturing (Color + Bump)

Data : col1: Farbe für horizontale Oberflächen
Data : col2: Farbe für vertikale Oberflächen
Data : worldPos: Position des Schnittpunktes in Weltkoordinaten
Data : normal: Oberflächen-Normale des Schnittpunktes
Data : texScale: Skalierungs-Faktor für die Texturen
Data : bumpTexture: Array von mehreren 2D Bump-Texturen
Data : textureDepthCoord: Einheitskoordinate, die die Verwendung mehrerer Texturen ermöglicht
Result : finalNormal: Durch *bumpmapping* veränderte Normale
Result : finalColor: Durch *triplanar coloring* veränderte Farbe

```
/* Berechne Gewichte für jeden Dimension anhand der Normalen */
float3 weights = fabs(normal.xyz);
weights = (weights - 0.2) * 7;
weights = fmax(weights, (float3)(0));
weights /= (weights.x + weights.y + weights.z );

/* Berechne Texturkoordinaten für jeden Dimension */
float4 coord1 = (float4)(worldPos.yx * texScale, textureDepthCoord, 0);
float4 coord2 = (float4)(worldPos.xz * texScale, textureDepthCoord, 0);
float4 coord3 = (float4)(worldPos.zy * texScale, textureDepthCoord, 0);

/* Berechne Bump-Werte für jede Dimension */
float2 bumpFetch1 = leseBild(bumpTexture, coord1).xy - 0.5;
float2 bumpFetch2 = leseBild(bumpTexture, coord2).xy - 0.5;
float2 bumpFetch3 = leseBild(bumpTexture, coord3).xy - 0.5;
float3 bump1 = (float3)(0, bumpFetch1.x, bumpFetch1.y);
float3 bump2 = (float3)(bumpFetch2.x, 0, bumpFetch2.y);
float3 bump3 = (float3)(bumpFetch3.x, bumpFetch3.y, 0);

/* Berechne finalen Normale und Farbe */
float3 bumpVec;
bumpVec = bump1.xyz * weights.xxx + bump2.xyz * weights.yyy + bump3.xyz * weights.zzz;
finalNormal = normalize(normal + bumpVec);

/* Berechne Einfärbung für jede Dimension */
finalColor = col1.xyzw * weights.xxxx + col2.xyzw * weights.yyyy + col1.xyzw * weights.zzzz;
```

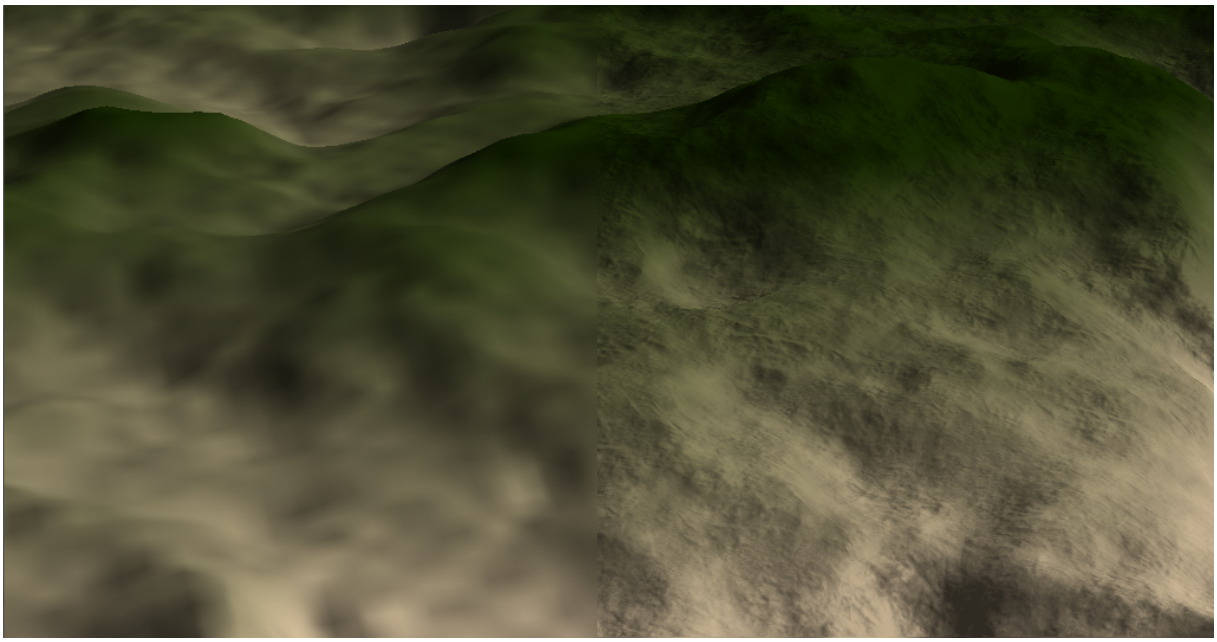

Abbildung 6.9: Verbesserung des subjektiven Detailgrads des Terrains durch Bumpmapping

Das Verfahren eignet sich für jegliche Anwendung von Texturen, wird in der Applikation allerdings nur für Bump-Maps genutzt. Anstatt Texturen für die Einfärbung des Terrains zu verwenden, wird das erwähnte Verfahren der Einfärbung anhand der Höhe mithilfe der Gewichte erweitert. Für die Breite und Tiefe wird die gleiche Farb-Textur verwendet während in der Höhe eine angepasste Textur genutzt wird. Dadurch werden steile Oberflächen durch grauen Felsen und planare Oberflächen eher durch Sand- oder Grasfarben visualisiert.

6.3.4 Impliziter Wasserpegel

Um die visuelle Qualität des Terrains zu erhöhen, wurde ein fester, globaler Wasserpegel implementiert, durch den der Eindruck von Seen entstehen soll. Sobald ein Strahl einen Schnittpunkt unterhalb des definierten Wasserpegels liefert, wird dieser Punkt alternativ visualisiert. Dafür wird der Schnittpunkt mit der Wasserebene ermittelt um einerseits die Distanz der Strecke zu berechnen, die der Strahl durch das Wasser dringt. Anhand dieser Distanz und anhand der Wassertiefe des Schnittpunkts, wird die ermittelte Farbe mit einer blauen Wasserfarbe gemischt. Je tiefer der Punkt unter Wasser liegt und je weiter der Strahl durch das Wasser dringen muss, desto mehr setzt sich die Wasserfarbe gegenüber der Farbe des Schnittpunktes durch.

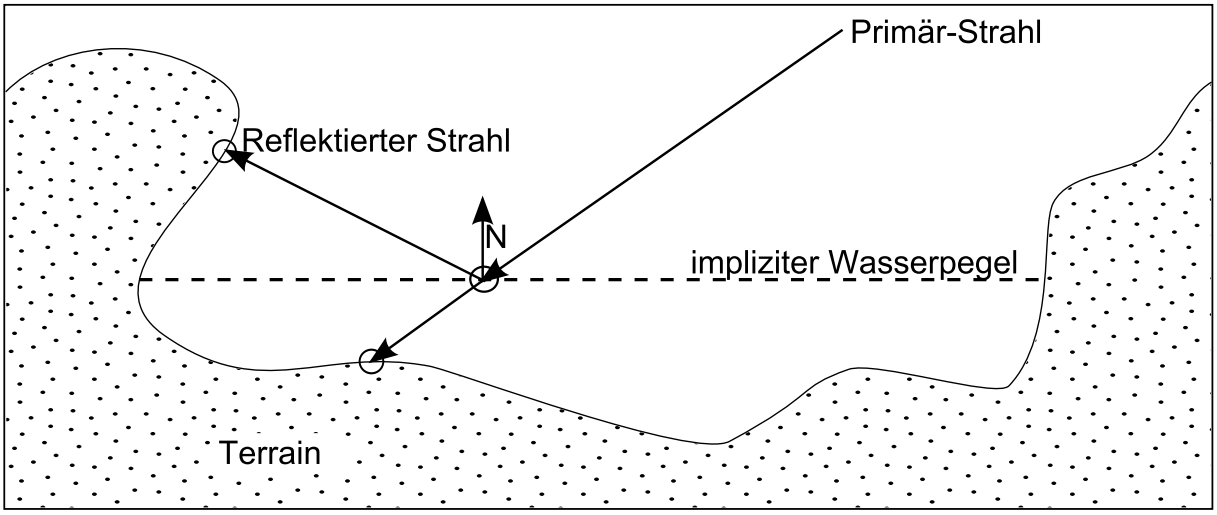

Abbildung 6.10: Schematische Darstellung der Spiegelung an einem implizitem Wasserpegel

Der Schnittpunkt mit der Wasseroberfläche wird zusätzlich benötigt, um von dort aus einen reflektierten Strahl zu erstellen und durch das Volumen zu verfolgen (siehe Abbildung 6.10). Die gefundene Farbe des Schnittpunkts dieses reflektieren Strahls wird wiederum mit der ermittelten Farbe des Schnittpunkts und der Wasserfarbe gemischt und erzeugt so eine, vom menschlichen Betrachter erwartete, Spiegelung auf der Wasseroberfläche. Des Weiteren wird die Oberflächen-Normale der Wasseroberfläche mithilfe einer über die Zeit animierten Normalen-Textur angepasst, um Unebenheiten auf der Wasseroberfläche, die sonst ein perfekter Spiegel wäre, zu simulieren.

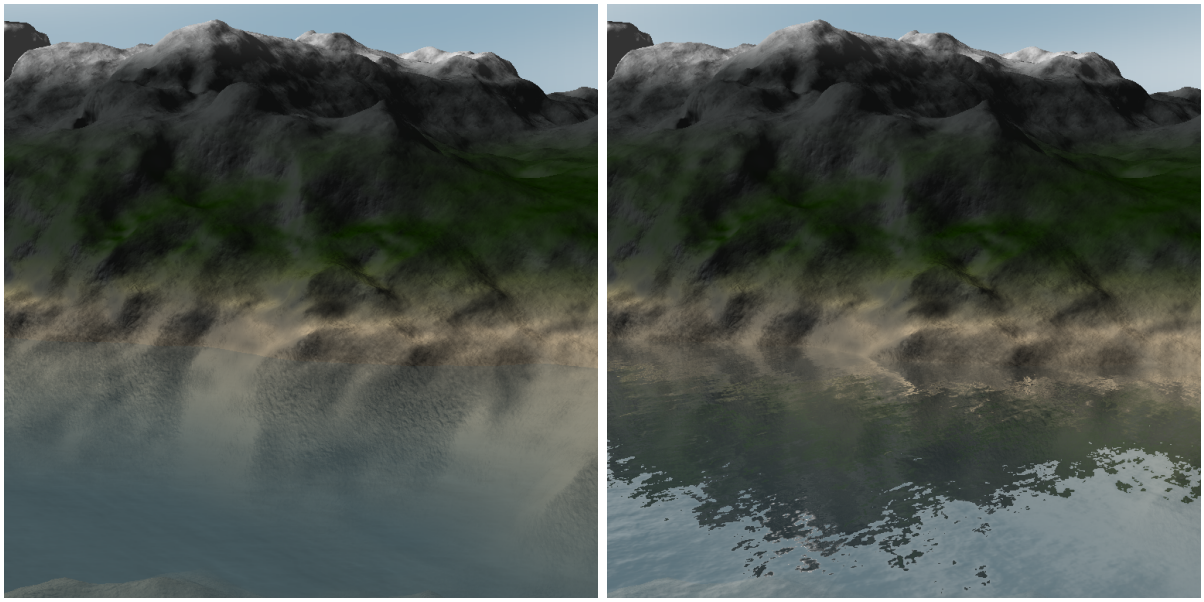

Abbildung 6.11: Darstellung des Wassers ohne (links) und mit Reflexionen (rechts)

6.4 Veränderbarkeit

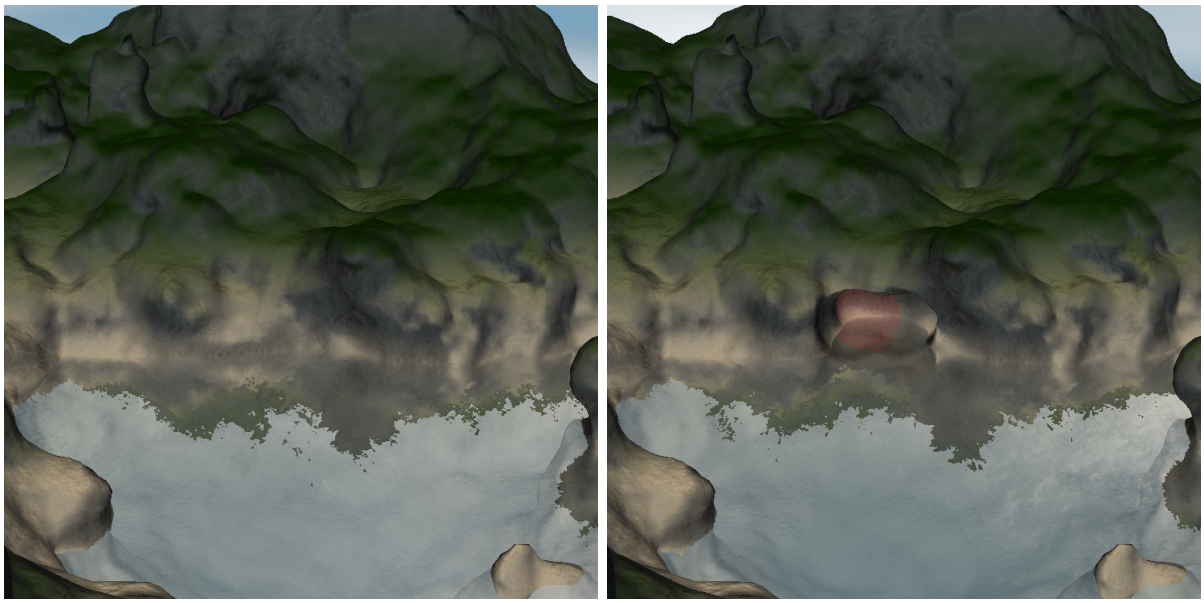

Abbildung 6.12: Veränderungen am Terrain: Ausgangssituation (links) und erste Änderungen (rechts)

Der Verzicht auf komplexe Datenstrukturen (wie z. B. Baum-Strukturen) macht sich in der Einfachheit der Umsetzung der Veränderbarkeit des Terrains bemerkbar. Da die Algorithmen direkt auf den Volumendaten arbeiten, müssen lediglich diese Daten angepasst werden - eine aufwändige Aktualisierung einer Baum-Struktur ist nicht nötig.

Um das Terrain zu verändern, müssen also vor allem die Volumendaten geändert werden. Texturen und Buffer verfügen über Methoden, die das Schreiben beliebiger Datenblöcke erlauben. Die Funktion *clEnqueueWriteImage(...)* ermöglicht das Schreiben von Datenblöcken (definiert durch Höhe, Breite und Tiefe) an eine beliebige Position (definiert durch Indizes der Dimensionen) innerhalb der dreidimensionalen Textur. Analog dazu ermöglicht die Funktion *clEnqueueWriteBufferRect(...)* das Schreiben eines rechteckigen Datenblocks in einen Buffer. So ist für jede Änderung von benachbarten Dichte-Werten genau ein OpenCL-Befehl nötig.

Die Interaktion ist so umgesetzt, dass der Nutzer den zu verändernden Bereich der Terrain-Oberfläche auswählt (siehe Abbildung 6.13) und das Programm ermittelt darauf aufbauend die betroffenen Volumendaten, liest diese aus dem Speicher, verändert sie und schreibt sie anschließend zurück. Darüber hinaus müssen die beiden Erweiterungen zur Beschleunigung der Traversierung des leeren Raumes aktualisiert werden. Die Änderung des vorberechneten Bits erfolgt für den veränderten Bereich nachdem die Dichte-Werte aktualisiert wurden. Die Höhen-Informationen für die angepasste Bounding-Box werden nur bei Bedarf, also bei neuen minimalen oder maximalen Voxeln mit gesetztem Bit, aktualisiert.

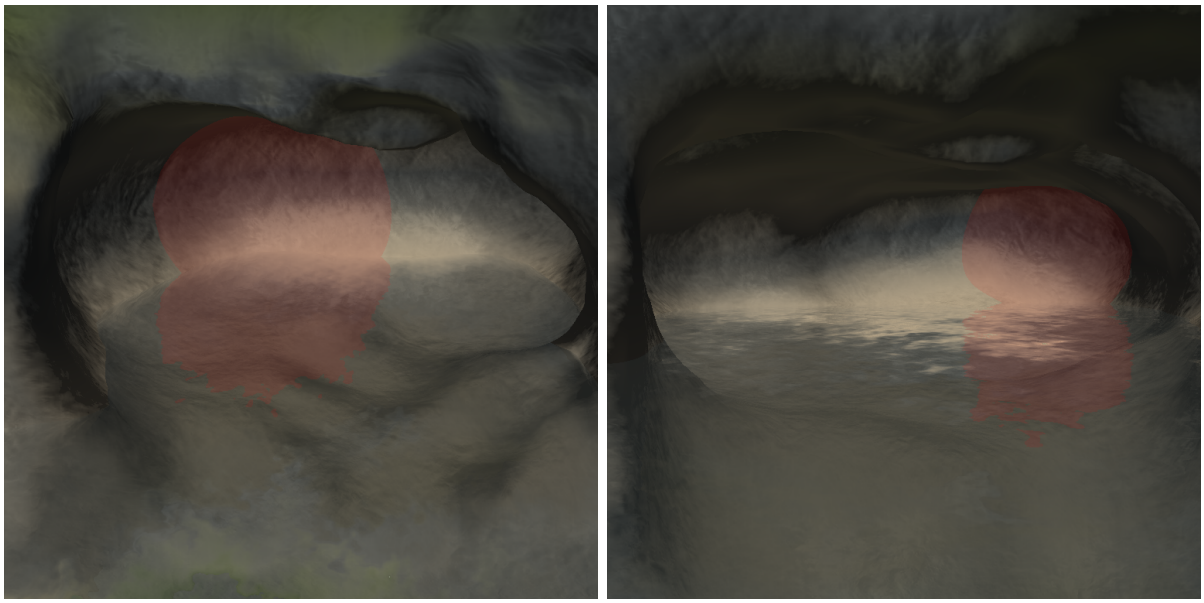

Abbildung 6.13: Veränderungen am Terrain: Weitere Veränderungen

Um die Veränderbarkeit beispielhaft zu demonstrieren, soll in dem Terrain auf Höhe des Wasserpegels eine Höhle ausgegraben werden. Abbildung 6.12 zeigt die Ausgangssituation (links) und erste Änderungen am Terrain (rechts). Für präzisere Änderungen wurde die Kamera in Abbildung 6.13 vor die Höhle bewegt um diese tiefer auszugraben. Abbildung 6.14 zeigt den Bau eines Tunnels (links) und das Ergebnis aus der Totalen (rechts).

Des Weiteren wurde eine Speicherung der vorgenommenen Änderungen am Terrain implementiert. Diese Änderungen werden innerhalb eines Programmablaufs gespeichert und auf einen Tastendruck werden sie binär in eine Datei geschrieben. Dafür wird für jeden Datenblock, in dem Änderungen vorgenommen wurden, eine Datei erstellt, in der die geänderten Daten-Werte mit ihrer ID und dem neuen Dichte-Wert gespeichert werden. Um Speicherplatz zu sparen wird nur für die Datenblöcke, in denen Änderungen vorgenommen wurden, eine Datei angelegt in die nur die Werte gespeichert werden, die von den Änderungen betroffen sind. Bei einem erneuten Programmstart kann beim Erstellen der Daten für jeden Datenblock geprüft werden, ob Änderungen gespeichert wurden (Datei vorhanden) um diese auszulesen und zu übernehmen.

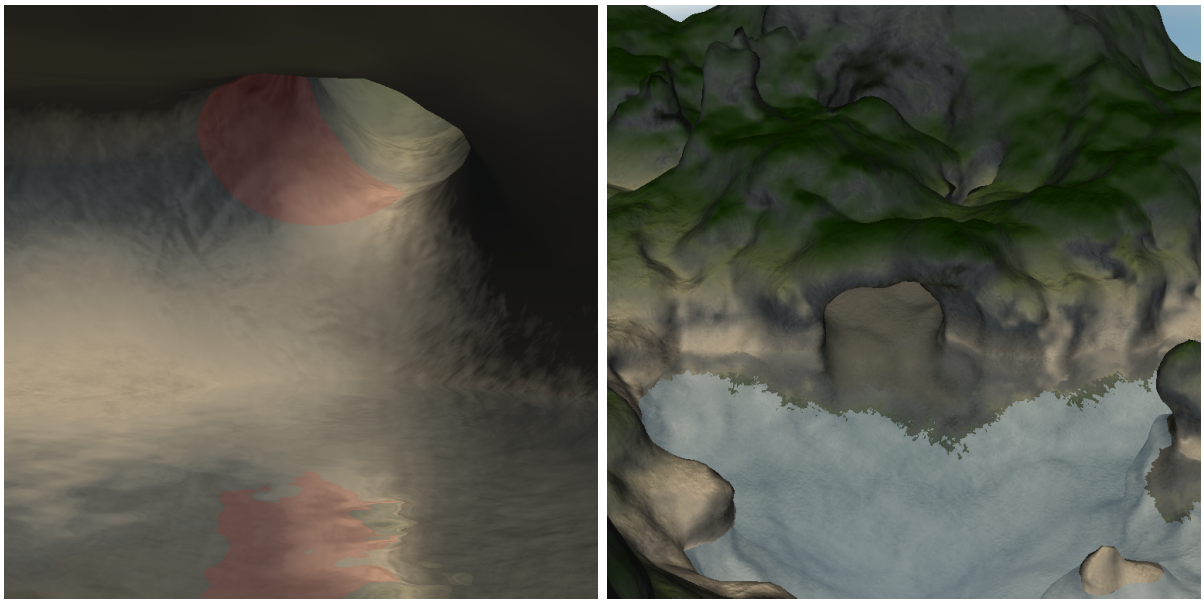

Abbildung 6.14: Veränderungen am Terrain: Ausgrabung eines Tunnels (links) und das Endergebnis (rechts)

6.5 Ergebnisse

Die Verknüpfung der genannten Methoden und Algorithmen führt im Resultat zu der Darstellung eines Terrains, das fast ausschließlich prozedural erstellt und visualisiert wird. Die Algorithmen arbeiten vor allem auf den impliziten Volumendaten und überführen diese nicht in eine Geometrie. Die gewünschte Echtzeitfähigkeit des Programms wird auf aktueller Hardware (NVIDIA GTX 670 und Intel Core i5 2500K), bei einer Bildauflösung von 768^2 Pixeln, in den meisten Fällen erreicht und nur selten fällt die Performanz unter die gewünschten 24 FPS (siehe Abbildung 6.15). Diese Performanz wird auch bei einem großen Terrain ($12^2 = 144$ Datenblöcke mit jeweils 129^3 Dichte-Werten) erreicht. Da das Terrain nicht zur Laufzeit erstellt, sondern vorberechnet wird, ist die maximale Größe durch den Speicher der verwendeten GPU limitiert. Ein Terrain aus 12^2 Datenblöcken benötigt bei dem gewählten Speicheraufbau circa 1,8 GB im Speicher der GPU (die GTX 670 verfügt über 2 GB). Durch die direkte Verwendung der Daten können Veränderungen des Terrains mit niedrigem Implementations- und Berechnungsaufwand vorgenommen werden. Die Verwendung einer optimierten Datenstruktur (z. B. kD-Tree) würde im Mittel zu einer besseren Performanz der Traversierung führen aber dafür würde der Aufwand bei Veränderungen des Terrains stark ansteigen.

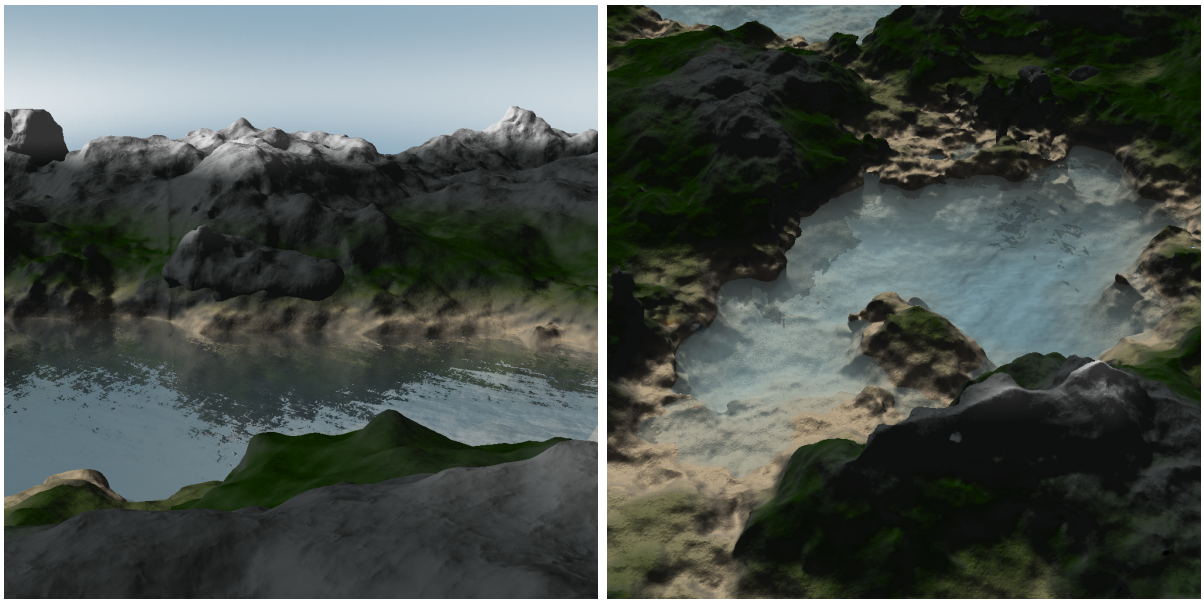

Abbildung 6.15: Visualisierung des Terrains in Echtzeit mit 29 (links) und 34 (rechts) FPS

Ein Vergleich mit der Darstellung der Daten als Würfel (siehe Abschnitt 2.2) erlaubt eine Bewertung des subjektiven visuellen Eindrucks der Terrain-Darstellung. Abbildung 6.16 zeigt den gleichen Bereich eines Terrains als Würfel- und Interpolations-Darstellung. Vor allem im Nahbereich (unten in den Abbildungen) werden die Stärken der interpolierten Visualisierung deutlich aber auch bei dem weit entfernten Berg sind höhere Details in der Einfärbung des Terrains zu erkennen.

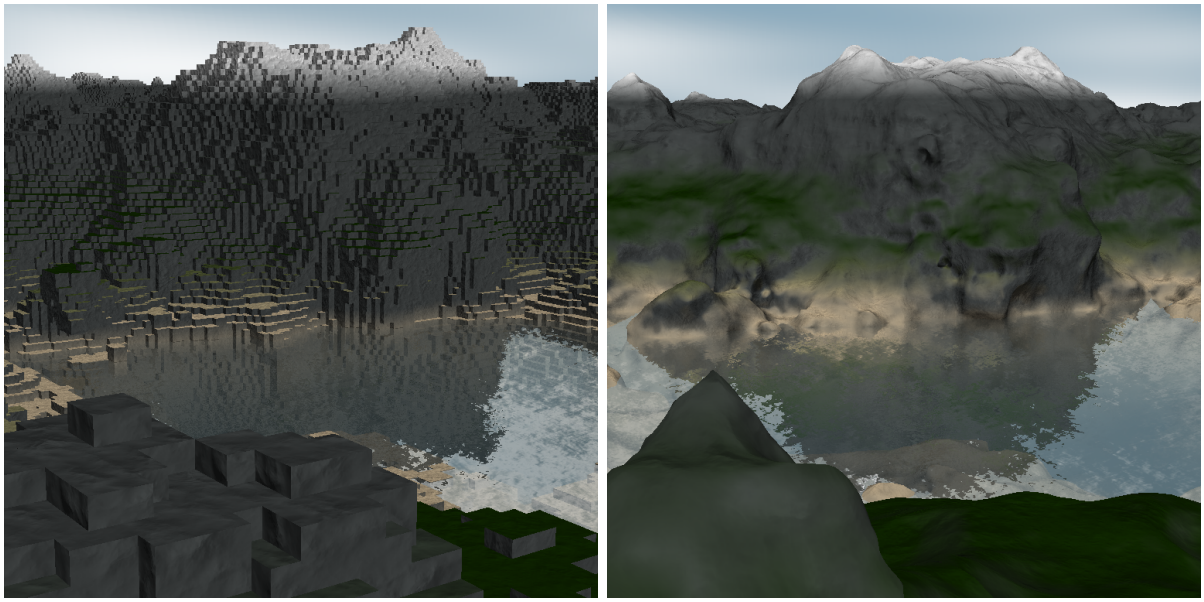

Abbildung 6.16: Darstellung eines Terrains mit Würfeln (links) und interpolierten Oberflächen (rechts)

Die Abbildungen 6.17 und 6.18 zeigen Visualisierungen der Applikation mit den höchsten visuellen Einstellungen.

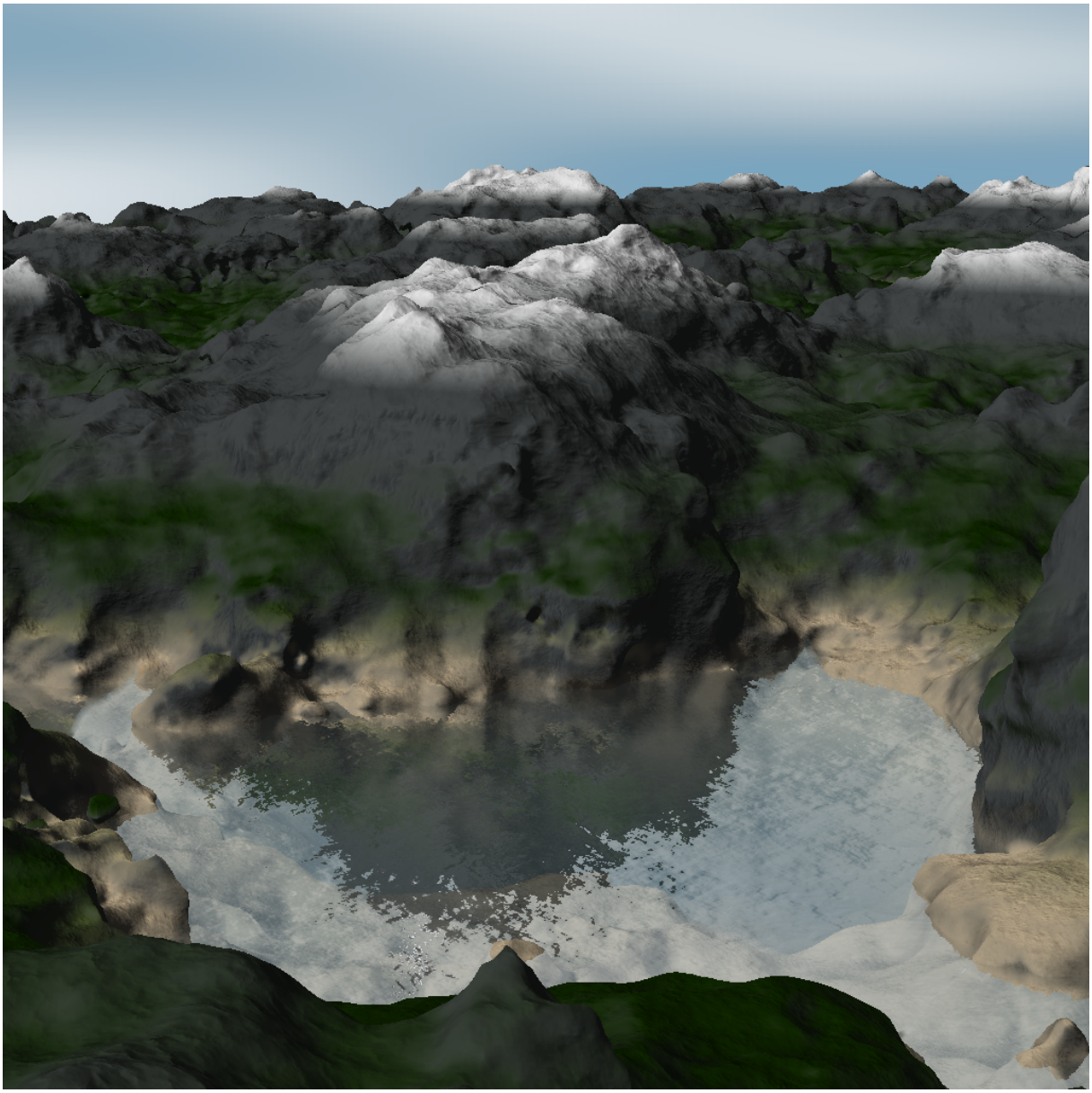

Abbildung 6.17: Visualisierung des Terrains mit allen Effekten

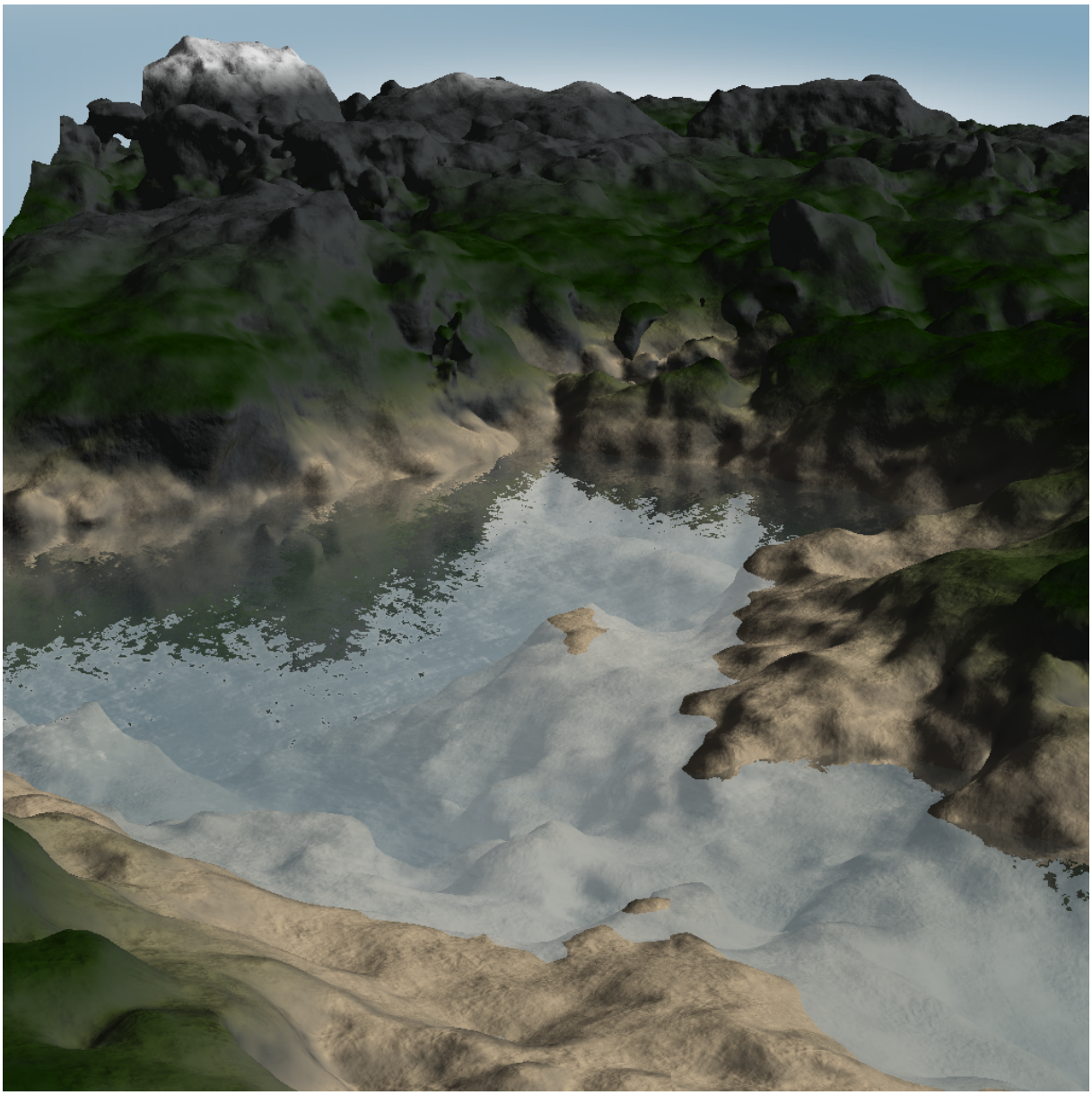

Abbildung 6.18: Prozedural generierte Strukturen

Kapitel 7

Fazit und Ausblick

Es wurde ein Verfahren vorgestellt, mit dem prozedural erzeugte Terrain-Daten interaktiv visualisiert werden können. Aufgrund der geforderten Terrain-Details (Überhänge und Höhlen) handelt es sich bei diesen Daten um dreidimensionale Rasterdaten, beziehungsweise Volumendaten. Die verwenden Algorithmen sind in der Lage, direkt auf den Volumendaten zu arbeiten anstatt diese zuerst in eine angenäherte Dreiecks-Geometrie zu überführen. Durch diese direkte Visualisierung können Oberflächendetails, wie zum Beispiel abgerundete Flächen, für die bei Verwendung einer Dreiecks-Geometrie sehr viele Dreiecke nötig wären, mit wenigen Voxeln dargestellt werden. Der Nutzer kann in Echtzeit durch die Szene navigieren und Änderungen (Entfernen und Hinzufügen) am Terrain vornehmen, die direkt übernommen und bei Bedarf gespeichert werden. Um die visuelle Qualität der Darstellung zu erhöhen, wurden einige prozedurale und empirische Erweiterungen vorgenommen. Da die visuelle Qualität der prozeduralen Komponenten nicht der Schwerpunkt dieser Arbeit ist, wurde selbige nur zu einem gewissen Grad verfeinert.

Das Problem des Speicherbedarfs könnte mit einer dynamischen Speicherverwaltung gelöst werden. Anstatt ein Terrain aus einer festen Anzahl Datenblöcken beim Programmstart zu generieren, müsste die Generierung für benötigte Datenblöcke im Hintergrund ausgeführt werden. Dies würde eine Portierung des Generierungs-Algorithmus auf parallele Hardware (angesprochen z. B. mit OpenCL) voraussetzen, um die Performanz des Programms nicht negativ zu beeinflussen (die sequentielle Generierung eines Datenblocks benötigt auf dem Testsystem 2-3 Sekunden). Da sich der Algorithmus von NVIDIA sehr gut für parallele Berechnungen eignet, wäre eine performante Portierung möglich. Mit so einem Algorithmus könnten die benötigten Datenblöcke zur Laufzeit generiert und nicht mehr benötigte Datenblöcke aus dem Speicher entfernt werden. Dies wäre voraussichtlich eine Lösung des Speicherproblems und würde zusätzlich ein theoretisch unendlich großes, prozedurales Terrain ermöglichen.

Die Veränderungen des Terrains könnten, anstatt manuell von einem Anwender vorgenommen zu werden, auf den Berechnungen eines prozeduralen Algorithmus basieren. Ein solcher Algorithmus könnte zum Beispiel die Erosion eines Terrains durch einen Fluss simulieren. Sofern dieser Algorithmus auf Volumendaten basiert, könnten die berechneten Änderungen in Echtzeit

mit dem vorgestellten Verfahren visualisiert werden. Auf dem Gebiet der Unterhaltungssoftware wäre eine prozedurale Umgestaltung eines Terrains zur Laufzeit möglich.

Als eine alternative Erweiterung könnte die Beleuchtung der Szene verbessert werden. Anstatt auf lokale Beleuchtung mit zusätzlichen direkten Schatten zu setzen, könnte die Szene mit vorberechneten Informationen teilweise global beleuchtet werden. Da diese vorberechnete Beleuchtung sehr aufwändig wäre, erschwert ihre Verwendung die angesprochene Generierung der Datenblöcke zur Laufzeit und würde sich eher für ein vorberechnetes Terrain eignen.

Eine weitere mögliche Erweiterung ist die Integration von physikalischen Berechnungen, die nicht auf die Verwendung einer Physik-*Engine* angewiesen sind. Diese *Engines* basieren meist auf Dreiecks-Geometrie um die physikalischen Berechnungen effizient auszuführen. Um ein implizites Volumen physikalisch zu verarbeiten, müsste also entweder eine Physik-*Engine*, die die Berechnungen auf Volumendaten ausführen kann, genutzt oder die Szene durch eine Dreiecks-Geometrie angenähert werden. Ersteres ist mit bestehenden Physik-*Engines* nicht möglich und Letzteres würde dem impliziten Ansatz widersprechen sowie bei einer zu ungenauen Annäherung zusätzlich zu falschen Ergebnissen führen.

Als Resultat kann festgehalten werden, dass eine direkte Visualisierung eines veränderbaren, impliziten Voxel-Terrains mit moderner Grafikhardware in Echtzeit möglich ist und sich die interpolierte Darstellung für ein Terrain eignet.

Abbildungsverzeichnis

[18] Abbildung von: `http://www.gearthblog.com/images/images807/terrain.jpg`
[19] Abbildung von: `http://http.developer.nvidia.com/GPUGems3/gpugems3_ch01.html`
[20] Abbildungen von: `http://en.wikipedia.org/wiki/Heightmap`
[21] Abbildung: `http://graphicsrunner.blogspot.de/2009/01/volume-rendering-102-transfer-functions.html`
[22] Abbildung von: `http://media2mult.uos.de/pmwiki/fields/cg-II-09/index.php?n=VolumeRendering.IsosurfaceRayCasting`

[23] Abbildungen aus [14]
[24] Abbildung von: `http://de.wikipedia.org/wiki/Phong-Beleuchtungsmodell`
[25] Abbildung von: `http://developer.amd.com/resources/documentation-articles/articles-whitepapers/opencl-and-the-amd-app-sdk-v2-4/`
[26] Abbildung von: `http://developer.amd.com/resources/documentation-articles/articles-whitepapers/opencl-and-the-amd-app-sdk-v2-4/`
[27] Abbildung von: `http://de.wikipedia.org/wiki/OpenCL`
[28] Abbildungen aus [15]

Literaturverzeichnis

[1] Amanatides, John; Woo, Andrew: A Fast Voxel Traversal Algorithm for Ray Tracing. In: *In Eurographics 1987*, 1987

[2] A.Neubauer; L.Mroz; H.Hauser; R.Wegenkittl: Cell-based first-hit ray casting. In: *Proceedings of the symposium on Data Visualisation 2002*, 2002

[3] Appel, Arthur: Some Techniques for Shading Machine Renderings of Solids. In: *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*. New York, NY, USA : ACM, 1968 (AFIPS '68 (Spring))

[4] Blinn, James F.: Models of Light Reflection for Computer Synthesized Pictures. In: *Proceedings of the 4th Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA : ACM, 1977 (SIGGRAPH '77)

[5] Ebert, David S.; Musgrave, F. K.; Peachey, Darwyn; Perlin, Ken; Worley, Steven: *Texturing and Modeling: A Procedural Approach*. 3rd. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2002

[6] Foley, Tim; Sugerman, Jeremy: KD-tree Acceleration Structures for a GPU Raytracer. In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, ACM, 2005 (HWWS '05)

[7] Gaster, Benedict; Howes, Lee; Kaeli, David R.; Mistry, Perhaad; Schaa, Dana: *Heterogeneous Computing with OpenCL*. 1st. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2011. – ISBN 0123877660, 9780123877666

[8] Horn, Daniel R.; Sugerman, Jeremy; Houston, Mike; Hanrahan, Pat: Interactive K-d Tree GPU Raytracing. In: *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, ACM, 2007 (I3D '07)

[9] Hughes, David M.; Lim, Ik S.: Kd-Jump: a Path-Preserving Stackless Traversal for Faster Isosurface Raytracing on GPUs. In: *IEEE Transactions on Visualization and Computer Graphics* 15 (2009), Nr. 6, S. 1555–1562

[10] Kirk, David B.; Hwu, Wen-mei W.: *Programming Massively Parallel Processors: A Hands-on Approach*. 1st. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2010. – ISBN 0123814723, 9780123814722

[11] Levoyl, Marc: Efficient Ray Tracing for Volume. In: *ACM Transactions on Graphics*, 1990

[12] Lindstrom, Peter; Koller, David; Ribarsky, William; Hodges, Larry F.; Faust, Nick; Turner, Gregory A.: Real-time, Continuous Level of Detail Rendering of Height Fields. In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA : ACM, 1996 (SIGGRAPH '96)

[13] Lorensen, William E.; Cline, Harvey E.: Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In: *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA : ACM, 1987 (SIGGRAPH '87)

[14] Marmitt, Gerd; Kleer, Andreas; Wald, Ingo; Friedrich, Heiko: Fast and Accurate Ray-Voxel Intersection Techniques for Iso-Surface Ray Tracing. In: *Proceedings of Vision, Modeling, and Visualization (VMV)*, 2004

[15] Nguyen, Hubert: *Gpu Gems 3*. Addison-Wesley Professional, 2007

[16] Parker, Steven; Shirley, Peter; Livnat, Yarden; Hansen, Charles; Sloan, Peter-Pike: Interactive Ray Tracing for Isosurface Rendering. In: *IEEE Visualization 1998*, 1998

[17] Perlin, Ken: Improving Noise. In: *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA : ACM, 2002 (SIGGRAPH '02)

[18] Pharr, Matt; Fernando, Randima: *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005

[19] Pharr, Matt; Humphreys, Greg: *Physically Based Rendering: From Theory to Implementation*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2004

[20] Phong, Bui T.: Illumination for Computer Generated Pictures. In: *Commun. ACM* 18 (1975), Juni, Nr. 6, S. 311–317

[21] Roettger, Stefan; Heidrich, Wolfgang; Seidel, Hans peter: Real-Time Generation of Continuous Levels of Detail for Height Fields, 1998

[22] Schwarze, Jochen: Cubic and Quartic Roots. In: *Graphics Gems (Andres Glassner)*, 1990

[23] Wald, Ingo; Friedrich, Heiko; Marmitt, Gerd; Seide, Hans peter: Faster Isosurface Ray Tracing using Implicit KD-Trees. In: *IEEE Transactions on Visualization and Computer Graphics* 11 (2005), S. 2005

[24] Whitted, Turner: An Improved Illumination Model for Shaded Display. In: *Commun. ACM* 23 (1980), Nr. 6, S. 343–349

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Osnabrück, Februar 2014