

Hochschule für Technik und Wirtschaft Berlin
Fachbereich 4

Masterarbeit

im Studiengang Internationale Medieninformatik

zur Erlangung des akademischen Grades
Master of Science

Thema: Vorteile und Nachteile von Voxeln gegenüber klassischer
3D-Modellierung

Autor: Kalina Lebecka
Matrikel-Nr. 542370

Datum: 2. April 2018

1. Betreuer: Prof. Dr. Tobias Lenz
2. Betreuer: Prof. Dr. David Strippgen

Inhaltsverzeichnis

1 Einleitung	5
2 Klassische 3D-Modellierung	7
2.1 Polygone	7
2.2 Grafikpipeline in OpenGL	7
3 Datenstruktur von Voxeln	10
3.1 Uniform Grid	10
3.2 Octree	11
3.3 Sparse-Voxel-Octree	12
3.3.1 Pointer-based Implementierung	12
3.3.2 Pointerless Implementierung	13
3.4 Speicherbedarf und Zugriff	15
3.5 Streaming	17
4 Voxel und Polygone im Vergleich	18
4.1 Terrain	18
4.1.1 Modifizierbarkeit	18
4.1.2 Erosion	23
4.1.3 Destructibles	25
4.1.4 Prozedurale Generierung	29
4.2 Level of Detail	30
4.3 Texturieren	41
4.4 Bewegung	52
4.4.1 Euklidische Transformation	52
4.4.2 Animation	54
4.5 Bildsynthese	61
4.5.1 Rasterung	61
4.5.2 BRDF	63
4.5.3 Globale Beleuchtung	64
4.5.4 Voxel-Rendering	69
4.6 Fraktale	80
5 Soft- und Hardware Unterstützung für Voxel	82
5.1 Software	82
5.2 Hardware	83
6 Fazit	86

Zusammenfassung

Die vorliegende Arbeit beschäftigt sich mit Voxeln und ihrem Potenzial, mit der klassischen, dreiecksbasierten Computergrafik zu konkurrieren. Die Motivation ist, herauszufinden, in welchen Szenarien Voxel gut zur Darstellung von 3D-Modellen geeignet sind. Dabei wird eine Anzahl an Eigenschaften und Einsatzgebieten von Voxeln und Dreiecken durch eine umfangreiche Literaturrecherche untersucht. Abhängig vom Anwendungsszenario zeigten sich verschiedene Stärken und Schwächen von Voxeln und Dreiecken. Trotz ihrer optimierten Datenstruktur hat die Verwendung von Voxelmodellen einen großen Speicherverbrauch zur Folge. Darüber hinaus weisen selbst hoch aufgelöste Formen bei näherer Betrachtung eine würfelige Struktur auf. Doch verfügen Voxel auch über praktische Eigenschaften. Sie lassen sich leicht prozedural generieren und gut modifizieren, haben ein implizites Level of Detail, brauchen für die Texturierung nicht parametrisiert werden und lassen sich effizient beleuchten. Hybride Systeme, die sowohl Voxel als auch Dreiecke verwenden, können oft aus beiden dieser Strukturen profitieren.

1 Einleitung

Voxel, auch *volume elements* genannt, sind diskretisierte Punkte, die in einem dreidimensionalen Raum liegen. Meistens werden Voxeln Volumen zugeschrieben, wodurch sie eine würfelige Form annehmen. Als Datenstruktur sind sie hilfreich in vielen wissenschaftlichen Feldern, wie beispielsweise in der Physik. Visualisierungen mithilfe von Voxeln sind vor allem im medizinischen Bereich und Medienbereich üblich. Während sich ihre voluminöse Natur perfekt zur Darstellung von dreidimensionalen, medizinischen Aufnahmen eignet, sind Voxel in Computerspielen und Filmen weniger beliebt. Viel öfter werden Modelle mit Polygonnetzen abgebildet. Dreiecke sind zum Standard geworden, was unter anderem daran erkennbar ist, dass heutige Grafikkarten komplett auf ihre Berechnung ausgelegt wurden.

Es gibt mehrere Alternativen zu Polygonen, wie beispielsweise NURBS oder Solid modeling, doch Voxel stellen vermutlich den stärksten Konkurrenten dar. Sie können unglaublich feine Details abbilden und besitzen dabei Eigenschaften, die sich bei der Bildsynthese als praktisch erweisen. Doch der überwiegende Großteil von Computerspielen greift auf die bewährte und klassische dreiecksbasierte Modellierung zurück. Die Motivation dieser Arbeit ist der Vergleich von Voxeln und Dreiecken, mit dem Ziel, herauszufinden, wie viel Potenzial Voxel als alternative Primitive haben. Dabei stehen interaktive Anwendungen, wie etwa Computerspiele, in denen sich Dreiecke seit Jahrzehnten beweisen, im Fokus.

Welche Technologien Computerspiele genau verwenden, wird oftmals nicht verraten. Eine einfache Aussage, dass ein bestimmtes Computerspiel Voxeltechnologie einsetzt, wäre außerdem äußerst unpräzise. Voxel stellen eine facettenreiche Technologie dar, die verschiedenartig eingesetzt werden kann. In der Spieleindustrie sind vor allem zwei Verwendungsarten von Voxeln oft vertreten.

Die erste Verwendung finden Voxel in der Modellierung der virtuellen Welt. Genau solch eine Gebrauchsart der Voxel steht in dieser Arbeit im Fokus. Ob Landschaft, Gegenstände oder Spielfiguren - alles, was aus Polygonen gebaut werden kann, kann auch in Voxelform existieren. Erwähnenswert ist aber, dass in den meisten Spielen Voxelmodelle vor der Bildsynthese „polygonisiert“ werden. Das heißt, dass Voxel zwar für die Speicherung der Datensätze, aber nicht direkt zur Visualisierung verwendet werden.

Die zweite, relativ neue Art der Verwendung von Voxeln, fokussiert sich auf die Berechnung von globaler Beleuchtung. Die schon seit langem existierenden Raytracing-Algorithmen ermöglichen eine viel realistischere Bildsynthese als die klassische Rasterung, doch sie werden wegen ihres hohen Rechenaufwands nicht in Echtzeit-Szenarien, wie Computerspielen, eingesetzt. Wie es sich herausstellt, ist die Berechnung von globaler Beleuchtung mit Voxeln als Primitiven leichter, als mit Polygonen. Die Technologien, die das ermöglichen, heißen Voxel Cone Tracing und verwenden in der Regel Voxel nur als Beschleunigungsdatenstruktur. Dies soll kein Fokus dieser Arbeit sein. Trotzdem wird Voxel Cone Tracing hier behandelt, denn es kann auch dazu gebraucht werden, Voxel selbst zu rendern.

Im Laufe der Arbeit werden Voxel und Polygone genau aus den beiden erwähnten Perspektiven betrachtet, d. h. wie gut sie sich zur Abbildung bestimmter Strukturen eignen und wie gut sie gerendert werden können. Als erstes wird jedoch kurz dargelegt, wie die klassische 3D-Modellierung funktioniert (Kapitel 2.2) und wie genau Voxel gespeichert werden (Kapitel 3).

Im Kapitel 4 werden mehrere Aspekte und Einsatzgebiete der Primitiven alleinstehend betrachtet. Dabei wird jedes Primitiv getrennt betrachtet und, wenn nötig, am Ende kurz mit der anderen verglichen. Es wird unter anderem untersucht, wie gut sich Voxel und Polygone zur Darstellung von Umgebung eignen (Kapitel 4.1), welchen Platz sie in einem Level-of-Detail-System haben (Kapitel 4.2) und wie sie texturiert (Kapitel 4.3) oder animiert (Kapitel 4.4) werden können. Anschließend wird im Kapitel 4.5 ausführlich analysiert, mit welchen Bildsyntheseverfahren beide Primitive gerendert werden können. Fraktale, als eine nicht verbreitete, doch interessante Form von Computergrafik, werden kurz im Kapitel 4.6 behandelt.

Inwieweit Voxel überhaupt von heutiger Soft- und Hardware unterstützt werden, wird im Kapitel 5 dargelegt. Die Arbeit wird mit dem Fazit im Kapitel 6 beendet.

2 Klassische 3D-Modellierung

Der Begriff „Computergrafik“ wurde erstmals 1960 von Verne L. Hudson, einem Angestellten bei Boeing, verwendet und von seinem Mitarbeiter William Fetter popularisiert (Kasik und Senesac, 2014). In den anfänglichen Jahren der Computergrafik entstanden viele verschiedene Visualisierungs-techniken, wie die Vektor-, Bitmap- und die Polygonografik. Letztere ermög-lichte die Darstellung einer dreidimensionalen Welt und bot einen bisher nicht bekannten Grad an Immersion. Carlson (2003) erwähnt, dass bereits 1980 eine Fakultät und Gruppe von Studenten aus der University of North Carolina Hardware entwarfen, die speziell auf 3D-Grafik ausgelegt war. In demselben Jahrzehnt wurden dann die ersten Grafikkarten produziert und Standards dafür festgelegt. Die Popularisierung der Polygonografik erreichte in den 90er Jahren dann seinen endgültigen Höhepunkt, zuerst in Arcade-, dann in Konsolen- und schließlich in Computerspielen.

2.1 Polygone

Die Definition eines Polygons erfolgt ausschließlich durch die Bestimmung seiner Eckpunkte, auch Vertices genannt. Die Punkte definieren eine Ebene (*face*) und die Kanten (*edges*) des Polygons. Bei mehr als drei Vertices kann es dazu kommen, dass diese nicht alle auf der gleichen Ebene liegen, was zu Problemen bei der Bildsynthese führt. Aus diesem Grund sind Po-lygonnetze, die aus Dreiecken bestehen, die gängigste Methode 3D-Objekte darzustellen. Dementsprechend wird klassische Computergrafik auch dreiecksbasierte Computergrafik genannt.

2.2 Grafikpipeline in OpenGL

Um die Funktionsweise der dreiecksbasierten Computergrafik zu verstehen, ist es sinnvoll, ihre Grafikpipeline zu betrachten. Da eine Grafikpipeline so-wohl software- als auch hardwareabhängig ist und es deshalb keine allgemeine Pipeline gibt, wird hier als Beispiel die Lösung von OpenGL präsentiert (siehe Abbildung 1).

Im ersten Schritt, „Vertex Specification“ genannt, wird eine Liste an Vertices erstellt und festgelegt, wie diese zu interpretieren sind. An dieser Stelle wird beispielsweise bestimmt, ob die Vertices nach ihrer Reihenfolge in der

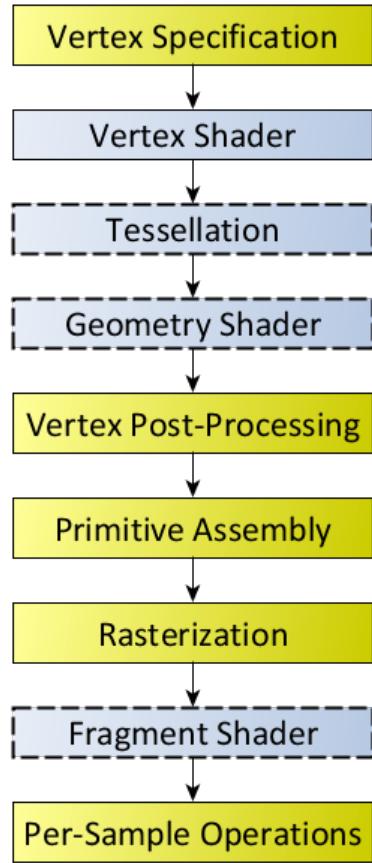


Abbildung 1: Die OpenGL-Pipeline. Programmierbare Shader werden blau gekennzeichnet, optionale Schritte werden mit gestrichelter Linie umrandet. (OpenGL Wiki: Rendering Pipeline Overview, 2017)

Liste oder einer festgelegten Reihenfolge abgearbeitet werden sollen. Auch die Art des Primitivs, d. h. ob es sich um Punkte, Linien, Dreiecke oder Vierecke handelt, wird hier festgelegt. Welche Daten genau in den Vertices gespeichert werden, bestimmt man im sog. *Vertex Array Objects*. Die Daten an sich werden aber in *Vertex Buffer Objects* gelagert.

Die nächsten drei Schritte gehören zum Überbegriff „Vertex Processing“. Im Vertex-Shader können beliebige Transformationen an Vertices ausgeführt werden, wobei der Shader nur Informationen über den einen, zurzeit bearbeiteten Vertex verfügt. Die typische Aufgabe des Vertex-Shaders ist die Berechnung der Weltkoordinaten eines Objektes, die Position der Spielwelt hinsichtlich der Kamera und die graphische Projektion der Szene auf den zweidimensionalen Bildschirm. Die Pro-Vertex-Operationen, wie z. B.

die Berechnung des Lichts, erfolgen auch im Vertex-Shader. Wesentlich für den Vertex-Shader ist, dass er für jeden eingegebenen Vertex nur einen Vertex wieder ausgibt. Die „Tessellation“, der zweite Teil vom Vertex Processing, dient der Zerlegung der Primitiven in kleinere Bestandteile und ist optional. Das gleiche gilt für den Geometry-Shader, der als Eingabe ein Primitiv bekommt und keine oder mehrere Primitive wieder ausgibt.

In der „Vertex Post-Processing“-Phase werden oft Operationen ausgeführt, die als Vorbereitung für weitere Schritte der Pipeline dienen, z. B. das *clipping*. Hier werden die Polygone bzw. die Polygonenteile entfernt, die außerhalb des Kamerawinkels sind und nicht gerendert werden müssen. Darauffolgend werden die Vertices im „Primitive Assembly“-Schritt zu Primitiven zusammengesetzt. Die Primitive, die der Kamera abgewandt sind, werden im Prozess namens *face culling* verworfen.

Im weiteren Schritt, „Rasterization“, werden die Primitive rasterisiert. Die Rasterung eines Primitivs führt zur Erstellung mehrerer Fragmente. Die Größe eines Fragments hängt mit Pixeln zusammen, doch für ein Pixel wird mindestens ein Fragment erstellt.

Der letzte Shader, Fragment-Shader genannt, ist optional. Hier wird vor allem die Farbe des Fragments anhand von Indizien wie Textur, Belichtung, Nebel und Ähnliches berechnet. Findet diese Verarbeitung nicht statt, behalten die Fragmente ihre Tiefe, haben jedoch keine Farbe.

Im letzten Schritt der Pipeline, „Per-Sample Operations“, werden einige, oft optionale Tests und Operationen ausgeführt, die meist über die endgültige Sichtbarkeit und Farbe eines Fragments entscheiden. Dazu gehört beispielsweise der *alpha test*, der der Darstellung von Billboards dient, und der *scissor test*, in dem Fragmente, die sich außerhalb eines rechteckigen Bereiches befinden, entfernt werden. Zu weiteren Beispielen gehören der *depth test*, der die Überlappung mehrerer Fragmente überprüft und der *stencil test*, der einen zusätzlichen Buffer zur Überprüfung spezieller Bedingungen bietet, mit dem sich verschiedenartige Effekte, wie beispielsweise Schatten, erzeugen lassen. Außerdem findet an dieser Stelle *color blending* statt, ein Prozess in dem die Farbe der teiltransparenten Fragmente mit der Farbe der „darunter“ liegenden Fragmente gemischt wird, um die Transparenz einer Fläche vorzutäuschen.

3 Datenstruktur von Voxeln

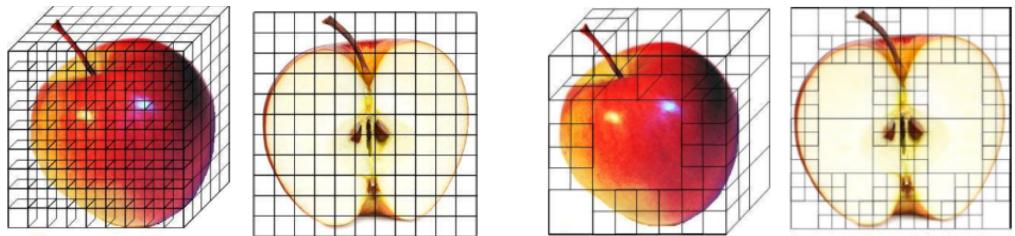
3.1 Uniform Grid

Eine der schlichtesten Datenstruktur zur Darstellung von Voxelmodellen ist ein Uniform Grid, das beispielsweise von Engel et al. (2006) erklärt wird. Im zweidimensionalen Raster von Elementen wird ein Element „Pixel“ genannt. Wird dieser Raum um eine zusätzliche Dimension erweitert, wird der das quadratische Pixel zu einem Würfel, oder anders ausgedrückt, zu einem „Voxel“. Ähnlich wie ihre zweidimensionalen Brüder, stehen Voxel, die in einem Uniform Grid gespeichert werden, dicht aneinander. Der große Unterschied zu Pixeln ist, dass Voxel möglicherweise keine Daten enthalten und somit durchsichtig sein können. Eine Visualisierung eines Uniform Grids ist auf der Abbildung 2a zu sehen.

Die einfachste Implementierung eines Uniform Grids mit binären Voxeln wäre eine zweidimensionale Liste an Bits. Nach Museth (2013) ist ein Uniform Grid als Datenstruktur aus mehreren Gründen vorteilhaft. Es ist simpel und verständlich, aber vor allem lassen sich die meisten volumetrischen Algorithmen und Operationen, wie Diskretisierung oder Interpolation, leicht darauf anwenden. Dies liegt an der einheitlichen Größe der Voxel. Museth (2013) erwähnt aber auch, dass in der volumetrischen Modellierung eine nicht einheitliche Abtastung von Vorteil sein kann. Darüber hinaus verbrauchen Uniform Grids, proportional zur Größe des Models, viel Speicher.

Effizientes Raycasting in einem Uniform Grid wurde bereits 1987 von Amanatides und Woo (1987) vorgestellt. In ihrem Algorithmus nutzen Amanatides und Woo eine Vektorgleichung in Parameterform um den Ray abzubilden. In jedem Schritt speichern sie, für jeweils eine Achse, den Wert t an der Stelle der nächsten Subdivision im Grid. Der kleinste der Werte wird dann ausgewählt, um von dort aus einen weiteren Schritt entlang der Linie zu gehen und den durchquerten Voxel zu speichern.

Havran et al. (2000) untersuchten, wann Uniform Grids in Hinsicht auf Raycasting sinnvoll in der Verwendung sind und stellten dabei heraus, dass die Verteilung der Geometrie im Raum eine signifikante Rolle spielte. Bei einer einheitlichen Verteilung sei die durchschnittliche Traversierung der Rays kurz, bei einer uneinheitlichen Verteilung dagegen seien Uniform



(a) Uniform Grid (links) und sein Querschnitt (rechts).
(b) Octree (links) und sein Querschnitt (rechts).

Abbildung 2: Uniform Grid und Octree visualisiert. (Gebhardt et al., 2009)

Grids ineffizient. Trotzdem ist dies die einzige Voxelstruktur, die mit Hardware von den meisten Grafikkarten unterstützt wird.

3.2 Octree

Wie Hossain et al. (2015) erklären, könnten die von Uniform Grids gegebenen Einschränkungen durch die Verwendung von Octrees umgegangen werden. In einem Octree wird jeder Voxel mit inhomogenen Daten auf acht Oktanten geteilt. Diese werden dann selbst zu Voxeln, die gegebenenfalls weiter geteilt werden müssen. Dieser rekursive Prozess dauert so lange, bis die Daten im aktuellen Voxel homogen sind oder eine gewisse Tiefe erreicht wurde. Die Abbildung 2b stellt die Aufteilung des Raumes gut dar. Der Vorteil von Octrees ist, dass leere oder solide Bereiche eines Models weniger Speicher verbrauchen, weil sie im Gegensatz zu Voxeln an der Isofläche nicht klein unterteilt werden müssen. Die Nachteile von Octrees, die zum Beispiel Goswami (2012) benennt, sind, dass es keine direkte Kontrolle über die Anzahl der Voxel gibt und dass der Baum sehr unausgeglichen (englisch *unbalanced*) werden kann. Je schlechter der Baum balanciert ist, desto länger dauert die Suche nach bestimmten Daten. Im Worst-Case-Szenario werden $\mathcal{O}(n)$ Operationen gebraucht, wobei n für die Anzahl der Knoten im Octree steht. Ist der Baum ausgeglichen, sind $\mathcal{O}(\log n)$ Operationen nötig, um an die gesuchten Daten zu kommen.

3.3 Sparse-Voxel-Octree

Ein Sparse Voxel Octree (SVO) ist eine verbesserte Octree-Datenstruktur, die je nach Implementierung verschiedene Optimierungen beinhaltet. Laine und Karras (2011) nennen ihren Baum „spärlich“, weil dieser keine Teilbäume mit leeren Voxeln enthält, was die häufigste Verbesserung eines SVO ist. Eine weitere Optimierung, die Gebhardt et al. (2009) nennt, wäre, die inneren Voxel eines Models auch zu verwerfen und nur die Oberflächenvoxel zu speichern. Dieses Vorgehen spart zwar zusätzlich viel Platz, ist aber bei von Spielern modifizierbaren Voxelstrukturen nicht sinnvoll.

Von weiterer großer Bedeutung ist die Art und Weise, wie ein Octree implementiert wird. In der Literatur werden zwei Modelle erwähnt: pointer-based und pointerless Implementierungen. Beide Modelle werden folgend ausführlich anhand der Beschreibung von Geier (2014) dargestellt.

3.3.1 Pointer-based Implementierung

Pointer-based oder auch hierarchische Lösungen basieren auf der „Verknüpfung“ der Baumknoten mit Pointern, also Objektreferenzen.

In einem Knoten acht Pointer zu seinen acht Kinder zu speichern ist ein Standardverfahren. Wird von einer Größe von 8 Byte pro Pointer ausgegangen, werden insgesamt 64 Byte Speicherplatz benötigt, um Referenzen zu allen Oktanten eines Voxels zu speichern. Trotz des großen Speicherverbrauchs ist diese Lösung nicht nur leicht zu implementieren, sondern ermöglicht auch eine schnelle Traversierung des Baumes.

In einem weiteren Verfahren, von Geier (2014) *Blockrepräsentation* genannt, werden Kinder zu „Blöcken“ zusammengefasst, damit nur eine Referenz, die des Blockes, im Elternknoten gespeichert werden muss. Somit besetzen die Referenzen pro Knoten nur noch 8 Byte Speicherplatz. Der Nachteil dieser Lösung ist, dass die Kinderknoten nicht mehr einzeln *on-demand* im Speicher alloziert werden können, wie Geier anmerkt. Bei der Teilung des Voxels werden alle Oktanten sofort alloziert, was vor allem im Fall von spärlichen Bäumen, mit vielen leeren Oktanten, nicht optimal ist. Die Visualisierung der Blockrepräsentation ist auf der Abbildung 3a zu sehen.

Eine Art Kompromiss zwischen der Standard- und Blockrepräsentation ist die, ebenfalls von Geier beschriebene, *Geschwister-Kind-Repräsentation*.

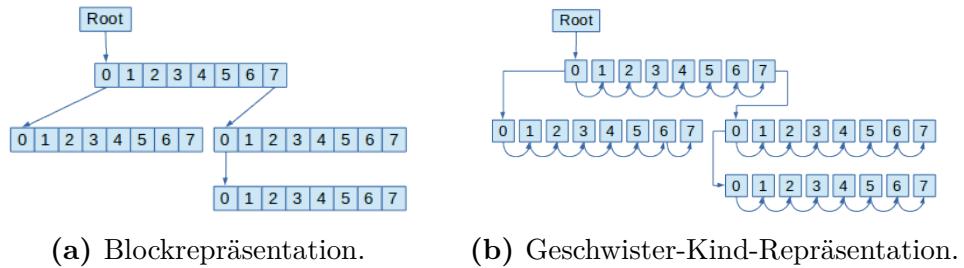


Abbildung 3: Alternative Implementierungen von pointer-based Octrees. (Geier, 2014)

Hier wird in jedem Knoten eine Referenz zu seinem nächsten Geschwister-Teil und zu seinem ersten Kind gespeichert, was der Speicherbelegung von 16 Byte entspricht. Im Gegensatz zu der Blockrepräsentation, können die Knoten in der Geschwister-Kind-Repräsentation wieder on-demand alloziert werden. Der Nachteil dieser Implementierung ist, dass sich der Traversierungsweg zu einem bestimmten Knoten verlängert, wodurch sich die Suche im Baum verlangsamt. Die Geschwister-Kind-Repräsentation wird auf der Abbildung 3b dargestellt.

3.3.2 Pointerless Implementierung

Als Beispiel für eine pointerless Implementierung stellt Geier (2014) sogenannte lineare bzw. *hashed* Octrees vor. Der lineare Quadtree, die zweidimensionale Äquivalente vom Octree, wurde das erste Mal von Gargantini (1982) beschrieben. Gargantinis Implementierung lässt sich problemlos um eine weitere Dimension ergänzen, wodurch sie auch Octrees abbilden kann.

Lineare Octrees besitzen keine explizite Baumstruktur, doch sie werden in sog. Z-Ordnung abgebildet. Dafür verfügt jeder Knoten über einen individuellen Ortungscode (englisch *locational code*), der seine Position im Baum festlegt. Alle Knoten eines Octrees werden in einer Hashtabelle gespeichert, wobei der Zugriff auf einen einzelnen Knoten mit seinem Ortungscode möglich ist.

Wie Geier (2014) vermerkt, werden die Ortungscodes so aufgebaut, dass anhand eines Knotens sein Elternteil oder seine Kinder schnell und leicht abgeleitet werden können. Der Code wird folgendermaßen aufgebaut: jeder Oktant bekommt, basierend auf seiner Position, eine 3-Bit lange Zahl von 0 bis 7 zugewiesen. Ein beispielhaftes Muster der Zuweisung ist auf der

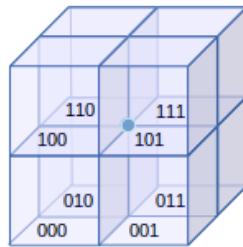


Abbildung 4: Lineares Octree, ein Beispiel einer pointerless Implementierung (Geier, 2014)

Abbildung 4 zu sehen, wobei das Muster natürlich beliebig sein kann, denn nur seine Konsistenz über den gesamten Baum ist wichtig.

Der Aufbau des Codes erfolgt rekursiv durch die Verknüpfung aller Eltern-codes vom Stammknoten bis zum gewünschten Oktanten. Das heißt, dass das Unten-Rechts-Vorne-Kind des 000-Oktantes aus der Abbildung 4 einen Ortungscode 000 001 besitzt.

Um den Oktanten 000 001 rechnerisch von 001 unterscheiden zu können, wird der Stamm des Baumes mit dem Code 1 versehen, sodass die Sequenzen zu jeweils 1 000 001 und 1 001 werden. Geier (2014) vermerkt, dass mit dieser Methode ein 32 Bit langer Code 10 Tiefen und ein 64 Bit langer Code 21 Tiefen eines Octrees beschreiben kann. Die Tiefe des Codes c lässt sich einfach mit der Funktion $\log_2(c)/3$ berechnen.

Obwohl kein direkter Vergleich zwischen linearen und pointer-based Octrees gefunden werden konnte, sind die Vorteile linearer Octrees klar. In einem Octree bis zur Tiefe von 10 werden maximal 32 Bits und bis zur Tiefe von 21 maximal 64 Bits pro Knoten benötigt. Wird ein mit dem üblichen pointer-based Standardverfahren implementierter Octree mit 21 Tiefen zu einem linearen Octree umgewandelt, verringert sich sein Speicherverbrauch um das Achtfache.

Geier (2014) deutet aber auf einen Nachteil von linearen Octrees hin. Im Gegensatz zu pointer-based Lösungen, lassen sich lineare Octrees nicht so leicht verändern, weshalb sie eher für die Darstellung statischer Modelle geeignet seien. Geiers Aussage basiert vermutlich auf dem bekannten Problem von schlechten Hashfunktionen, die langsam in der Ausführung sind und viele Kollisionen verursachen. Andererseits ist die durchschnittliche Ausführungszeit der oben erwähnten Operationen in hashed Octrees konstant, während die Darstellung eines Octrees als ein Baum die durch-

schnittliche Such-Komplexität von $\mathcal{O}(\log n)$ impliziert. Welche Implementierung am geeignetsten ist, hängt letztendlich von vielen Faktoren ab und muss individuell von der Situation abhängig entschieden werden.

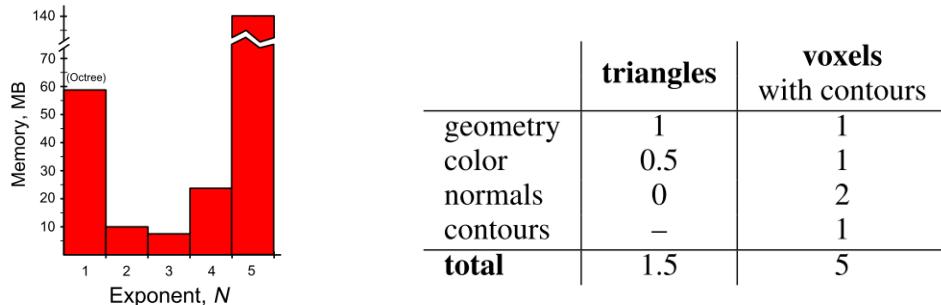
3.4 Speicherbedarf und Zugriff

Wie bereits erwähnt, variiert der Speicherbedarf von Octrees bzw. SVOs stark und hängt von ihrer Implementierung und der Menge der in den Voxeln kodierten Informationen ab. McNeely et al. (1999) untersuchten Optimierungen hierarchischer Strukturen und fanden dabei heraus, dass ein N-Baum, in dem jeder Knoten auf 2^{3N} (mit $N > 1$) Subvolumen geteilt wird, möglicherweise weniger Speicher als ein Octree (in dem $N = 1$) beansprucht. Im Falle der von McNeely getesteten Szene, verbrauchte ein Baum mit $N = 3$ am wenigsten Speicher, wie auf der Abbildung 5a zu sehen ist. Das optimale N ist jedoch stark von dem „Spärlichkeitsgrad“ der Szene abhängig. Crassin (2011) weist zusätzlich darauf hin, dass ein Baum mit niedrigem N meistens speichereffizienter sei, während sich Bäume mit großem N durch eine kürzere Traversierungszeit auszeichneten.

Egal in welcher Ausführung, Voxel haben den schlechten Ruf eines immensen Speicherverbrauchers. Laine und Karras (2011) verglichen in ihrer Studie ein Voxelmodell mit einem Polygonmodell. Letzteres war zwar sehr grob, verfügte aber über einzigartige Texturen und Displacement Mapping. Somit hatten die beiden Modelle einen vergleichbaren Grad an Details.

Ist die Geometrie eines Polygonmodells wirklich grob, verbraucht sein Dreiecksgitter kaum Speicher. Die Normalen müssen nicht gespeichert werden, da sie von der Displacement Map abgeleitet werden können. Laut Laine und Karras (2011) lassen sich Farben mit der DX-Texturkompression mit 4 Bits pro Texel kodieren, während für dreiachsiges Displacement Mapping, implementiert mit der DXT5-Kompression von van Waveren und Castaño (2008), 8 Bits pro Texel gebraucht werden. Dies entspricht 1,5 Bytes pro Texel.

Voxel von Laine und Karras (2011) haben eine spezifische Eigenschaft namens *contours*. Es sind zwei parallele Ebenen, die innerhalb eines Oberflächenvoxels die generelle Ausrichtung der Modelloberfläche beschreiben. Die Contours werden bei der Bildsynthese verwendet, damit das blockige Modell glatter aussieht. Laut Laine und Karras werden durchschnittlich



(a) Speicherverbrauch eines 2^{3N} -Baumes mit Voxeln.
(McNeely et al., 1999)

(b) Speicherverbrauch (pro Voxel) eines groben Polygonnetzes mit Texturen und Displacement Mapping versus Speicherverbrauch eines optisch ähnlichen Voxelmodells. Alle Angaben in Bytes. (Laine und Karras, 2011)

Abbildung 5: Speicherverbrauch von Voxeln und Polygonen.

circa 5 Bytes pro Voxel, das die Farbe, die Normalen und eigene Contours speichert, benötigt. Die Zahl stützt sich auf der Annahme, dass durchschnittlich vier Kinder eines Knotens Blätter sind und dementsprechend nicht weiter geteilt werden.

Eine Zusammenfassung ist auf der tabellarischen Abbildung 5b zu sehen. In dem System von Laine und Karras (2011) haben Voxelmodelle circa einen 3,33 höheren Speicherverbrauch, als visuell entsprechende Polygonmodelle. Die Belastung des Arbeitsspeichers hängt jedoch stark mit der verwendeten Auflösung zusammen, denn sowohl Octrees als auch Dreiecksgitter mit MIP-Maps, lassen sich deutlich simplifizieren. Der Vergleich von Laine und Karras (2011) zeigt eine generelle Tendenz von Voxeln zu einem größeren Speicherverbrauch, doch wie die Autoren selbst feststellen, ist der Vergleich von Voxeln und Polygonen nur eingeschränkt möglich. Taucht beispielsweise auf einer größeren Fläche nur eine Farbe auf, muss diese in einem Voxelmodell für mehrere Voxel gleichzeitig nur einmal in dem Elternknoten gespeichert werden. Solches Minimalisieren von redundanten Daten ist auch in der Geometrie möglich (siehe Kapitel 4.6). Diese Optimierungstechniken sind stark von der Beschaffenheit der Daten abhängig und wurden deshalb von Laine und Karras (2011) in ihrem Vergleich nicht berücksichtigt.

3.5 Streaming

Auch wenn vorsichtig implementierte Voxelstrukturen einen für große Festplatten akzeptablen Speicherverbrauch haben, sind einzelne Szenen oft zu groß, um sie komplett in den Arbeitsspeicher zu laden. Doch um dies nicht tun zu müssen, ist die explizite Implementierung einer Streaming-Technik erforderlich. Ein in der Spieleindustrie bekanntes Beispiel einer Streaming-Technik sind MegaTextures. Sie wurden von id Software entwickelt und werden verwendet, um die Nutzung gigantischer Texturen zu ermöglichen, indem nur ihre nötigen Teile in den Arbeitsspeicher geladen werden. Es gibt auch diverse andere Techniken für das Streaming von Texturen, die auf derselben Idee basieren (Crassin, 2011). Gigantische Texturen werden auf kleinere, gleich große Bereiche aufgeteilt, von denen nur ein Bruchteil tatsächlich in den Arbeitsspeicher geladen und zu einer spärlichen virtuellen Textur zusammengefügt wird. Bei der Bildsynthese im Fragment-Shader werden die UV-Koordinaten eines gesuchten Texels so übersetzt, dass sie auf die richtige Stelle in der virtuellen Textur zeigen. Existiert der gesuchte Bereich nicht, wird er dynamisch in den Arbeitsspeicher geladen.

Octrees haben eine Struktur, die sich ohne weiteren Aufwand leicht streamen lässt. Es werden nur die Knoten eines Octrees alloziert, die für die Bildsynthese benötigt werden. Eine ausführliche Beschreibung von Voxel-Streaming stellen Crassin et al. (2009) in ihrer Publikation zur Verfügung. Ihre Idee ist folgende: Jeder Strahl (bezogen auf Raycasting, siehe Kapitel 4.5.3) wird so lange auf seiner Länge gesampelt, bis der von ihm abgetastete Alpha-Wert Eins entspricht, oder bis er auf einen Knoten trifft, der im Arbeitsspeicher fehlt. Ein fehlender Knoten wird dann aus dem Festspeicher geladen. Im Moment, in dem ein Strahl auf einen Knoten trifft (egal ob schon länger oder erst kurz geladen), wird in dem Knoten ein Timestamp gespeichert. Neue Elemente werden nach dem LRU-Prinzip (*least recently used*) immer an Stellen der Knoten geladen, die am längsten nicht verwendet wurden. Außerdem werden Knoten, die seit langem nicht von Strahlen getroffen wurden, nach einer bestimmten Zeit automatisch verworfen. Die Verbindung von Streaming und Raycasting hat praktische Nebeneffekte. Beispielsweise werden Voxel, die aus der Sicht der Kamera verdeckt oder aufgrund des Kamerafrustums unsichtbar sind, nicht geladen. Diese Funktionalität besteht, ohne dass weitere Tests erforderlich sind.

Da die relativ geringe Speicherzugriffsgeschwindigkeit in modernen Systemen schnell zu Engpässen führen kann, analysieren Crassin et al. (2009) in ihrer Publikation zusätzliche Optimierungen, die das Streaming beschleunigen können. Diese werden hier, in Anbetracht des Umfangs dieser Arbeit, nicht weiter behandelt.

4 Voxel und Polygone im Vergleich

4.1 Terrain

Das Erzeugen von Landschaften ist ein Gebiet der Spieleentwicklung, in dem der Einsatz von Voxeln am populärsten erscheint. Es ist auch das Gebiet, in dem Voxel 1992 vom Entwickler NovaLogic das erste Mal in der Spielegeschichte verwendet wurden. Die Landschaft in Comanche: Maximum Overkill ist aus heutiger Sicht vielleicht nicht mehr so beeindruckend, damals setzte sie sich aber durch den einmaligen Detailgrad von der Menge ab. Die realistische Szenerie wurde in jener Zeit mit einem einfachen Raycasting auf einer Heightmap und mit passenden perspektivischen Transformationen erreicht. Heute werden Voxel in komplizierteren Strukturen gespeichert, die viel mehr zu bieten haben. Die interessantesten Aspekte bei der Darstellung von Landschaften mit Voxeln, sowie die Möglichkeiten der dreiecksbasierten Grafik in diesem Gebiet, werden in den folgenden Unterkapiteln dargelegt.

4.1.1 Modifizierbarkeit

Voxel. Das beste Beispiel dafür, dass aus echten, voluminösen Voxeln mehr Gebrauch gemacht werden kann als aus Heightmaps, ist das Spiel Minecraft von Mojang. In Minecraft werden zwar keine Voxel-Rendering-Techniken verwendet, denn die Landschaft bilden Polygonnetzblöcke, doch diese Blöcke werden intern als Voxel gespeichert. Was Minecraft so beliebt gemacht hat, ist die Möglichkeit des Spielers der wunschgemäßen Modifizierung der voxelartigen Landschaft. Beispielsweise können in Minecraft tiefe Minen gegraben und aus den gewonnenen Ressourcen Berge aufgetürmt werden. Demzufolge kann gesagt werden, dass es generell zwei Boolesche Operatoren gibt, mit denen sich ein Modell modifizieren lässt: Subtraktion

und Addition. Die Umgestaltung eines Voxelmodells mit diesen Operationen wird oft als virtuelle Bildhauerei (englisch *sculpting*) bezeichnet.

Damit die Modifizierung in Echtzeit ausgeführt werden kann, müssen Voxel in einer Datenstruktur gespeichert werden, die schnelles Finden, Entfernen und Hinzufügen ermöglicht. Wie schon im Kapitel 3.3 angedeutet wurde, gibt es keine perfekte Datenstruktur. Dies spiegelt sich auch in der Literatur wider. Çit et al. (2013) untersuchten beispielsweise die Modifizierung von Voxelmodellen in Echtzeit und entschieden sich für lineare Octrees, während McNeely et al. (1999) in einem ähnlichen Anwendungsfall zu hierarchischen Strukturen griffen.

Die Modifizierung einer Voxelstruktur ist in der Theorie unkompliziert. Mit einem addierenden/subtrahierenden Tool selektiert der Benutzer einen Bereich, die betroffenen Voxel werden daraufhin solide oder leer. Die zwei Herausforderungen dabei sind das Selektieren eines bestimmten Bereiches und die Anpassung der Datenstruktur auf eine solche Weise, dass diese weiterhin „spärlich“ genannt werden kann. Die Modifizierung mit Tools, die eine primitive Form haben, wie etwa eine Kugel oder ein Würfel, ist sehr leicht umzusetzen. Deshalb wird im Folgenden ein Vorgehen erklärt, das, unabhängig von der Toolform, immer gleich funktioniert.

Besteht das Selektionstool aus Voxeln, die von der Größe her den Voxeln des Modells entsprechen, kann die Bildhauerei wie im Folgenden beschrieben geschehen: Für jeden Voxel v des Tools wird im Modell der kleinste Knoten k gesucht, der v enthält. Ist k ein Blatt im Baum und seine Daten gleichen nicht denen von v , muss k solange weiter geteilt werden, bis die vom Tool bestimmte Voxelgröße erreicht ist. Gleichen sich die Daten von v und k , ist keine weitere Unterteilung nötig. Wenn k kein Blatt ist, ersetzt v sein entsprechendes Kind. Nachdem die Modifizierung beendet wurde, wird das Elternteil des modifizierten Voxels überprüft. Sind die Daten seiner Kinder nach der Modifizierung homogen, werden diese entfernt, um Speicherplatz zu sparen.

Wird das Voxelmodell durch die Verwendung eines Addition-Tools größer, muss möglicherweise auch sein Hüllkörper (englisch *bounding box*) vergrößert werden. Dies ist besonders bei pointer-based Lösungen leicht umsetzbar, indem ein neuer Wurzelvoxel mit acht Kindern erstellt wird, wobei eine Kinderreferenz auf den alten Wurzelvoxel des Baumes gerichtet wird. Dadurch wird zusätzlicher „Platz“ für neue Voxel erschaffen, aber die Größe

des Modells verdoppelt sich auf jeder der drei Achsen.

Obwohl die Funktionalität der Modifizierung von Voxelmodellen leicht umzusetzen ist, sind Echtzeitveränderungen mit einem Tool nur in einer kleinen Skala möglich. Die Ausführungszeit wächst linear zum betroffenen Bereich und kann deshalb bei großen Bereichen zu lang dauern.

Neben gezielter, manueller Modifizierung des Terrains, gibt es eine weitere Möglichkeit dies zu verändern: durch Zerstörung. Sogenannte *destructibles* sind Spielemente, die durch physische Kräfte zerfallen können. Diese Art der Modifizierung wird im Kapitel 4.1.3 behandelt.

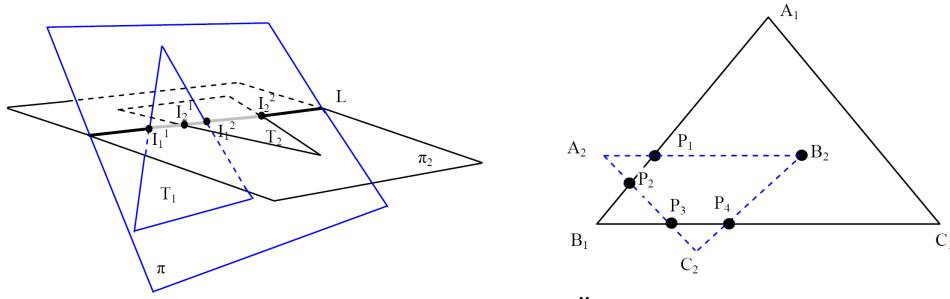
Polygone. Eine Modifizierung des Dreiecksgitters eines Modells ist, ähnlich wie bei Voxelmodellen, mit Booleschen Operatoren möglich. Eine Methode für diese Art der Modifizierung wurde von Qu et al. (2008) vorgestellt und wird im Folgenden am Beispiel von Subtraktion erklärt.

Angenommen wird, dass vom Körper A (Modell) der Körper B (Tool) subtrahiert wird. Der Algorithmus erfolgt in drei Schritten.

Der erste Schritt ist, alle Überschneidungen zwischen den Dreiecken von A und B zu finden. Jedes Dreieck eines Körpers wird auf Schnittpunkte mit allen Dreiecken des anderen Körpers überprüft.¹ Qu et al. (2008) unterscheiden grundsätzlich zwischen zwei Fällen. Der erste Fall bezieht sich auf eine Überschneidung von zwei komplanaren Dreiecken, während der zweite sich mit einer Überschneidung von zwei nicht planaren Dreiecken beschäftigt. Für die Berechnung der Überschneidung zwischen zwei komplanaren Dreiecken stellen Qu et al. eine eigene Methode vor, während sie für nicht planare Dreiecke den Algorithmus von Möller (1997) vorschlagen. Welche Schnittpunkte genauer gesucht werden, wird auf der Abbildung 6 visualisiert.

Der zweite Schritt ist die Retriangulierung. Dabei verwenden Qu et al. (2008) eine inkrementelle Methode, deren Voraussetzung ist, dass das mo-

¹Dieser Prozess ist rechnerisch aufwändig, weshalb sich bei großen Modellen Optimierungstechniken empfehlen, wie z. B. das Legen eines dreidimensionalen Uniform Grids über das Modell und eine Speicherung der Dreiecke in den Knoten, zu denen sie physisch gehören. Dank dieser Datenstruktur kann die Auswahl an Dreiecken, die sich potenziell mit einem ausgewählten Objekt überschneiden, schnell eingegrenzt werden. Durch die Verringerung der Anzahl an nötigen Schnitttests wird das gesamte Verfahren deutlich schneller.



(a) Überschneidung zweier nicht planaren Dreiecke. Schnittpunkte: I_1^1 und I_1^2 .

(b) Überschneidung zweier komplanarer Dreiecke. Schnittpunkte: P_1 , P_2 , P_3 , P_4 und B_2 .

Abbildung 6: Überschneidung zweier Dreiecke. Im Fall (a) werden die Punkte gesucht, an denen die Kanten oder Ecken eines Dreiecks die Fläche des anderen Dreiecks überschneiden. Im Fall (b) zählen zur Schnittpunkten alle Punkte, an denen die Kanten oder Ecken eines Dreiecks die Kanten des anderen Dreiecks überschneiden. Zusätzlich werden die Ecken eines Dreiecks dazu gezählt, wenn sie sich in der Fläche des anderen Dreiecks befinden. Im Fall, dass sich zwei Kanten beider Dreiecke überschneiden, werden nur ihre Ecken als Schnittpunkte anerkannt. (Qu et al., 2008).

delliertes Polygonnetz die Umkreisbedingung² der Delaunay-Triangulierung erfüllt. Die gefundenen Schnittpunkte werden einer nach dem anderen den Dreiecken zugeordnet, auf denen sie liegen. Nach der Zuordnung jedes Punktes erfolgt eine Retriangulierung, bevor ein weiterer Punkt hinzugefügt wird. Die Retriangulierung ist simpel: wird ein Punkt zu einem Dreieck hinzugefügt, werden dessen Ecken mittels neuer Kanten mit dem hinzugefügten Punkt verbunden. So entstehen aus einem Dreieck drei Dreiecke, die möglicherweise keine Umkreisbedingung mehr erfüllen.

Damit das Polygonnetz weiterhin eine Delaunay-Triangulierung genannt werden kann, muss möglicherweise an manchen Dreiecken das sogenannte Lawson-Flip-Verfahren von Lawson (1972) angewandt werden. Dieses ist auf der Abbildung 7 zu sehen. Durch das „Flippen“ der Kante wird die Umkreisbedingung zweier Dreiecke lokal wieder erfüllt, allerdings können diese Veränderungen wiederum die Umkreisbedingungen der benachbarten Dreiecke stören. Dies muss geprüft und die betroffenen Dreiecke müssen

²Die Umkreisbedingung besagt, dass der Umkreis eines Dreiecks keine Punkte anderer Dreiecke enthalten soll. Dies ist eine Bedingung der Delaunay-Triangulierung. Erfüllt ein Polygonnetz diese Bedingung nicht, kann es dazu kommen, dass es sehr schmale und lange Polygone enthält. Dies bringt wiederum viele Nachteile mit sich, wie z. B. eine seltsame Beschattung bei der Rasterung.

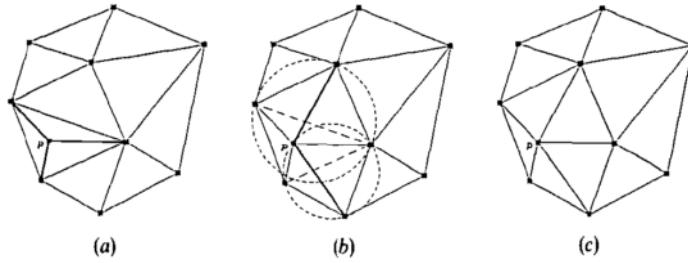


Abbildung 7: Wird ein neues Punkt zum Polygonnetz hinzugefügt, werden die dadurch entstandenen Dreiecke auf die Umkreisbedingung geprüft. Enthält der Umkreis eines Dreiecks A einen Punkt eines anderen Dreiecks B , wird die gemeinsame Kante von A und B gelöscht. Anschließend wird eine neue Kante erstellt, die die Punkte von A und B , die bisher nicht verbunden waren, verbindet. (Tsai, 1993)

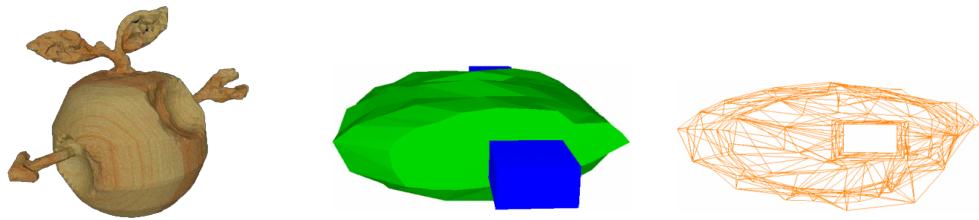
gegebenenfalls auch geflippt werden. Im schlimmsten Fall sind alle Dreiecke im Polygonnetz betroffen, was heißt, dass der Aufwand für das Flippen nach dem Hinzufügen eines Punktes $\mathcal{O}(n)$ beträgt. Nach Su und Scot Drysdale (1997) beläuft sich jedoch die Zeit im Durchschnitt auf $\mathcal{O}(1)$, wenn die Punkte in zufälliger Reihenfolge hinzugefügt werden.

Im letzten Schritt des Algorithmus werden die Dreiecke von A , die auch in B oder auf der Grenze von B liegen, gelöscht. Darauffolgend werden die Dreiecke von B , die auch innerhalb von A liegen, zu A hinzugefügt. Qu et al. (2008) liefern eine detaillierte Beschreibung, wie herausgefunden werden kann, welche Dreiecke betroffen sind.

Nach der Ausführung der oben genannten Schritte ist die Modifizierung des Polygonnetzes abgeschlossen. Leider machten Qu et al. (2008) keine Bemerkungen zur Interaktivität ihrer Methode.

Vergleich. Sowohl Voxel- als auch Polygonmodelle lassen sich mit Booleschen Operationen modifizieren (siehe Abbildung 8). Boolesche Operationen an Polygonnetzen scheinen weniger populär zu sein. Sie sind zwar in gängigen Polygonmodellierungsprogrammen eingebaut, werden aber selten genutzt, weil sie des Öfteren eine unschöne Struktur des Polygonnetzes ergeben. Trotz der Umkreisbedingung in der Delaunay-Triangulierung kann es zu *skinny* (kurze Basis und lange Seiten) Dreiecken kommen.

Die meisten Spiele, die modifizierbares Terrain haben, verwenden dafür Voxeltechnologien. Manche Entwickler, wie Çit et al. (2013), voxelisieren



(a) Ein Voxelmodell eines Apfels, erschaffen nur mit Booleschen Operationen. (Perng et al., 2001)

(b) Ein Polygonmodell vor der Subtraktionsoperation. (Qu et al., 2008)

(c) Ein Polygonmodell nach der Subtraktionsoperation. (Qu et al., 2008)

Abbildung 8: Visualisierungsbeispiele zu Booleschen Operationen an Voxeln und Polygonen.

sogar ihre dreiecksbasierten Modelle, um anschließend Boolesche Operationen darauf auszuführen und die Ergebnisse wieder zu polygonisieren. Dies ist ein Hinweis darauf, dass der Einsatz Boolescher Operationen auf großen Polygonnetzen nicht oder nur schwer interaktiv sein kann.

Voxel sind nicht nur resistenter gegen Fehler, sie sind auch leichter zu handhaben, weil jeder Voxel eine Farbe speichert. Somit ist die Textur gleich in das Modell integriert. Wird das Modell modifiziert, z. B. mit einem Subtraktionstool, erfordert es keine weitere Anpassung mehr, während ein Polygonmodell retexturiert werden muss.

4.1.2 Erosion

Eine weitere Anwendung der Voxelmodelle stellen Beneš und Forsbach (2001) vor. In ihrer Arbeit legen sie dar, dass nicht nur Informationen, wie ob der Voxel solide ist oder welche Farbe er hat, sondern auch geologische Merkmale gespeichert werden können. Somit können sich in einem Voxel Informationen zu Gehalt und Menge von Wasser, Mineralien oder Gas befinden. Dies ermöglicht eine noch realistischere Implementierung von Simulationsmodellen, wie beispielsweise Erosion.

Erosion ist ein Prozess der Abtragung von Gestein, Erde oder anderen gelösten Materialien von einem Ort zum anderen, wobei die Träger beispielsweise Wind, Wasser oder Gravitation sind. In der Literatur wird viel darüber diskutiert, wie Erosion effizient simuliert werden kann. Die für diese Zwecke gängigsten Datenstrukturen sind nach Beneš Heightmaps oder Voxel. Heightmaps sind schnell und simpel, Voxel sind dafür in der Lage

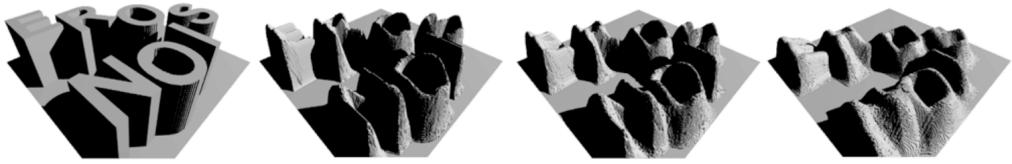


Abbildung 9: Beispiel einer thermischen Erosion. (Beneš und Forsbach, 2001)

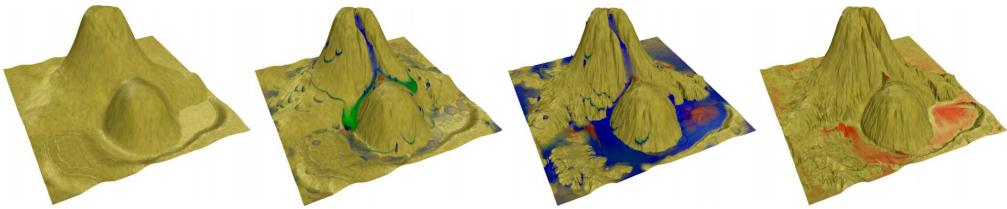


Abbildung 10: Beispiel einer hydraulischen Erosion mit Regen und einer Wasserquelle. (Mei et al., 2007)

Strukturen wie Höhlen oder Klippenvorsprünge abzubilden.

Zwei oft erwähnte Erosionsarten sind die thermische und hydraulische Erosion. Die thermische Erosion entsteht auf Grund von Temperaturänderungen, wie beispielsweise durch ständigen Wechsel von kalten Nächten und sonnigen Tagen, da dies die Struktur des Gesteins brüchig macht. Die kleinen Teilchen, die abbrechen, fallen nach unten. Ein Beispiel dieses Phänomens ist auf der Abbildung 9 zu sehen. Die thermische Erosion wurde das erste Mal algorithmisch von Musgrave et al. (1989) beschrieben. Um die Steigung an einer bestimmten Stelle und in der Zeit t zu berechnen, wird die Höhe a_t^v von Voxel v mit der Höhe a_t^u seines Nachbarvoxels u verglichen. Wenn die Steigung größer als der Talus-Winkel ist, wird ein fester Prozentsatz c_t der Menge an Material von v auf den Nachbar übertragen. Gilt also $a_t^v - a_t^u > T$, dann tritt $a_t^u + c_t(a_t^v - a_t^u - T)$ ein.

Die hydraulische Erosion entsteht, wenn fallendes Wasser, wie Regen oder ein Fluss, Teile von Gestein und Erde aufnimmt und diese zu einer anderen, niedrigeren Stelle transportiert (siehe Abbildung 10). Musgrave et al. (1989) stellten ein Verfahren vor, in dem Wasser in den oberen Voxeln der Landschaft gespeichert wird. Jeder Voxel v speichert seine Höhe a^v , seine Menge an Wasser w^v und die Menge der sich darin enthaltenen Sedimente s^v . In jedem Schritt werden Nachbarvoxel u von Voxel v gesucht, deren $w^u + a^u$ kleiner als $w^v + a^v$ ist. Auf diese Voxel wird dann Wasser und Se-

diment von v übertragen, wobei ein Bruchteil des Sediments auf v bleibt, wodurch seine Höhe steigert. Wie viel und wie schnell es übertragen wird, wird durch Parameter kontrolliert.

Obwohl Musgrave et al. (1989) für die Simulation Heightmaps verwendet haben, kann man die Algorithmen leicht an Voxelstrukturen anpassen. Beide von Musgrave et al. beschriebenen Methoden haben die $\mathcal{O}(n)$ Komplexität, aber hydraulische Erosion ist deutlich langsamer. Weder die hydraulische Erosion von Musgrave et al., noch die von Beneš und Forsbach (2001) ist interaktiv, obwohl bei beiden Lösungen schnelle Heightmaps verwendet wurden. Daher lässt sich feststellen, dass sich Voxel für die Simulation von Erosion eignen, die Simulation muss aber vorab berechnet werden. Echtzeit-Erosion an Voxelmodellen bleibt derzeit schwer zu erreichen. Trotzdem sind Voxel im Fall der Erosionssimulation durch ihre physische Korrektheit eine deutlich bessere Wahl als eine klassische, dreiecksbasierte Modellierung.

4.1.3 Destructibles

Zerstörbare Gegenstände im Spiel werden oft Destructibles genannt. Es gibt zwei wesentliche Eigenschaften, die Destructibles auszeichnen. An erster Stelle müssen sie in der Lage sein zu zerfallen, zu zerplatzen oder auf eine andere Weise kaputtzugehen. Zweitens müssen sich die Reste, die von einem zerstörten Destructible übrig bleiben, entsprechend den auf sie ausgeübten Kräften verhalten. Leider grenzt es an das Unmögliche, alle existierenden Gegenstände realistisch als Destructibles darstellen zu können. Der Grund dafür ist, dass es zu viele verschiedene Materialien verschiedenartiger Zusammensetzungen gibt, die alle über unterschiedliche Eigenschaften verfügen. Deshalb beschränkt sich die Spieleindustrie darauf, nur aus möglichst simpel aufgebauten Gegenständen oder Materialien Destructibles zu erstellen. Dazu gehören beispielsweise Wände, Türen oder Glasscheiben, die sich alle ein ähnliches Zerfalls muster teilen.

Polygone. Ein klassisches, *statisches* Vorgehen ist, das Modell bei jeder Kraftausübung auszutauschen und die eventuell davon abgefallenen Teile zu animieren. Dies hat den Nachteil, dass das Modell manuell verändert werden muss und dass die Destruktion immer gleich aussieht, unabhängig davon, woher die Kraft kommt. Aus diesem Grund werden seit ein-



(a) Destruktion mithilfe eines dreidimensionalen Voronoi-Diagramms als Muster.



(b) Destruktion mithilfe eines prozedural erstellten, spinnwebenähnlichen Musters.

Abbildung 11: Beispiele von Destructibles anhand von zwei verschiedenen Mustern. Der rote Zeiger kennzeichnet die Aufprallstelle. (Müller et al., 2013)

paar Jahren *dynamische*, bzw. prozedurale Verfahren erforscht. Wie Müller et al. (2013) darlegen, basieren diese Verfahren auf der Idee, generische Zerfallsmuster an der Aufprallstelle anzuwenden. In ihren Tests verwenden sie selbst zwei Arten von Mustern: dreidimensionale Voronoi-Diagramme (siehe Abbildung 11a) und prozedural erstellte, spinnwebenähnliche Strukturen (siehe Abbildung 11b). Erstere können für diverse Materialien verwendet werden, während letztere zur Abbildung zersprungener Glasscheiben geeignet sind. In beiden Mustern gibt es eine Art Zentrum, in dem die konvexen Zellen am kleinsten sind.

Wenn ein Objekt mit großer Geschwindigkeit mit einem anderen Gegenstand kollidiert, wird das Muster so darauf platziert, dass sein Zentrum an der Aufprallstelle liegt. Die Kanten des Musters bestimmen, wie der Gegenstand zerkleinert werden soll. Die Suche nach Überschneidungen zwischen dem Objekt und dem Muster, sowie die darauf folgende Retrangulierung, kann, wie es im Kapitel 4.1.1 beschrieben wurde, erfolgen.

Müller et al. (2013) verwendeten in ihren Tests einen Rigid-Body-Simulator von Tonge et al. (2012), um die zerkleinerten Bestandteile eines Destructibles in Bewegung zu setzen. Tonge et al. berechnen auf der GPU

näherungsweise die zwischen Objekten wirkenden Kräfte durch approximatives Lösen von linearen Gleichungssystemen. Das durchaus sehr komplizierte Simulationsverfahren wird allerdings hier nicht weiter behandelt. Die wichtigste Erkenntnis aus der Publikation von Tonge et al. für den Vergleich von voxelbasierter und dreiecksbasierter Computergrafik besteht darin, dass die Simulation der physischen Kräfte auf polygonbasierten Modellen in Echtzeit möglich ist. Dies bestätigten auch Müller et al. (2013), die den Rigid-Body-Simulator zum Testen ihrer Zersetzungstechnik verwendet haben und ebenfalls interaktive Bildsynthesezeiten erzielten.

Voxel. Ob auch voxelbasierte Modelle sich als Destructibles eignen, wurde unter anderem von Domaradzki und Martyn (2016) untersucht. Ihre Idee ähnelt stark anderen in dreiecksbasierter Grafik verwendeten Lösungen, doch die Umsetzung unterscheidet sich etwas aufgrund der spezifischen Struktur eines SVO.

In ihrem System benutzen Domaradzki und Martyn ein explizites Voronoi-Diagramm. Sie erstellen eine Menge an zufällig verteilten Punkten, die jedoch im Zentrum des Diagramms dichter sind. Es werden keine Ebenen zwischen den Voronoi-Punkten berechnet, sondern ihr Abstand zu einzelnen Voxeln wird direkt interpretiert. Angenommen $\{s_i\}_{i \in S}$ definiert die Voronoi-Punkte, während $S = 1, \dots, n$ eine Menge seiner Indizes ist, dann wird folgende Funktion, die die Zugehörigkeit eines Punktes x zu einer Voronoi-Zelle ermittelt, aufgestellt:

$$\gamma(x) = \{k \in S : \|s_k - x\| = \min_{i \in S} \|s_i - x\|\}. \quad (1)$$

Die Funktion $\gamma(x)$ liefert Indizes der Voronoi-Punkte, zu denen x gehört. Ist die zurückgegebene Menge kein Singleton, liegt x auf einer Ebene, einer Geraden oder einem Vertex des Voronoi-Diagramms. Die Funktion wird anschließend an würfelige Voxel angepasst. Ein Voxel wird definiert durch die Menge seiner Vertices $V = \{v_i\}_{i=1, \dots, 8}$ und die Funktion $\mathcal{U}(V)$ liefert die Menge aller Indizes der Voronoi-Punkte, zu denen Vertices aus V gehören.

$$\mathcal{U}(V) = \bigcup_{v \in V} \gamma(v) \quad (2)$$

Liefert $\mathcal{U}(V)$ kein Singleton, gehört V zu der Menge von Voxeln \mathcal{B} , die auf der Grenze zwischen auseinanderfallenden Bestandteilen eines Destructibles liegen. Diese Voxel werden an der Stelle dynamisch weiter unterteilt, wobei, wie üblich, die Unterteilung abbricht, wenn ein Kind homogene Daten enthält (seine \mathcal{U} -Funktion gibt ein Singleton zurück) oder die größtmögliche Auflösung erreicht wurde. Am Ende dieses Prozesses enthält \mathcal{B} nur die Kinder eines Voxels, die immer noch auf einer Grenze mehrerer Voronoi-Zellen liegen. Folglich wird ein *fracture boundary set* (FBS) hergerichtet, indem jedes $V \in \mathcal{B}$ solange kopiert wird, bis die Anzahl der Kopien die Kardinalität von $\mathcal{U}(V)$ erreicht. Jede Kopie wird dabei mit dem Index des entsprechenden Voronoi-Punktes verbunden, sodass alle Kopien eines Voxels als Elemente getrennter Mengen wahrgenommen werden können. Die FBS ist also die Menge aller Voxel-Kopien, die an Kanten des Voronoi-Diagramms liegen. Anhand dieser Menge, die abschließend nach Zugehörigkeiten zu bestimmten Voronoi-Punkten sortiert wird, werden neue SVOs erstellt. Da Domaradzki und Martyn (2016) die Position der Voxel aus FBS mitspeichern, ist die Erstellung neuer Octrees unproblematisch.

Wurde ein Destructible auf mehrere Octrees aufgeteilt, kann die Simulation der darauf wirkenden Kräfte beginnen. Dafür müssen die Kollisionsstellen zwischen den SVOs bekannt sein. Domaradzki und Martyn (2016) überprüfen zuerst die Hüllkörper der Octrees auf gegenseitige Überschneidungen. Sind solche zwischen einem Octreepaar vorhanden, werden diese gleichzeitig traversiert und es wird ein weiterer Schnitttest zwischen jedem Oktantenpaar einer Tiefe durchgeführt. Dies ist ein effizientes Verfahren, denn es werden nur die Voxel auf Überschneidungen geprüft, deren Eltern sich auch überschnitten haben. Außerdem wird nicht der gesamte Octree abgesucht, denn die letzten Überschneidungstests finden im aktuellen LOD statt. Anhand der Überschneidungen wird die Fläche der Kollision berechnet und darauf basierend werden die Kräfte auf die Objekte angewandt.

Das Verfahren von Domaradzki und Martyn (2016) besteht aus zwei Etappen: dem Zerlegen des Octrees und der Simulation der physischen Kräfte. Das Zerlegen kann durchaus in interaktiver Zeit geschehen, doch seine Skalierbarkeit ist schlecht. Während der Stanford Bunny aus der Abbildung 12, der aus acht Octreetiefen besteht, in 12.9 ms zerstört wurde, werden für genau das gleiche Modell, nur aus zehn Tiefen bestehend, 110.8 ms gebraucht. Am längsten dauert der „Insel-Test“, in dem Domaradzki

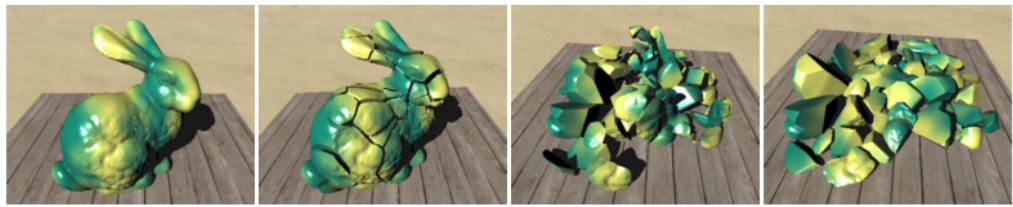


Abbildung 12: Ein Stanford Bunny (acht Octreetiefen), zerlegt in mehrere Bestandteile mithilfe eines Voronoi-Diagramms mit 90 Zellen. (Domaradzki und Martyn, 2016)

und Martyn disjunkte Voxelstrukturen, die sich in einer Voronoi-Zelle befinden, suchen, um sie abschließend zu trennen. Optimierungen in diesem Schritt könnten den Algorithmus deutlich beschleunigen.

Abgesehen von der Zerlegung, werden nach jedem Zeitschritt (englisch *time step*), der bei Domaradzki und Martyn (2016) 1 ms dauert, physische Auswirkungen berechnet. Domaradzki und Martyn legen dar, dass die Simulation der Trümmer des Stanford Bunnys mit acht Octreetiefen 4-9 ms pro Zeitschritt dauerte. Dies würde bedeuten, dass die Hardware mindestens 400 ms pro Sekunde nur mit der Berechnung dieser Simulation beschäftigt ist. Dementsprechend wird geschlussfolgert, dass vor allem kleine Voxelmodelle durchaus als dynamische Destructibles genutzt werden können, doch die damit verbundenen Rechenkosten hoch sind. Dreiecke schneiden in diesem Feld zur Zeit besser ab, doch werden Voxel weiter intensiv erforscht, haben sie eine Chance als alternative Destructibles eingesetzt zu werden.

4.1.4 Prozedurale Generierung

Im Gegensatz zu den meisten in einer Spielwelt vorkommenden Objekten, lässt sich das Terrain gut prozedural modellieren. Eines der möglichen Verfahren wurde von Geiss (2007) vorgestellt. In seiner Methode verwendet Geiss Uniform-Grid-Voxel, an deren Stellen er Werte aus einer Dichtefunktion abfragt, um zu bestimmen, ob die Voxel solide oder leer sind. Die Dichtefunktion besteht bei Geiss aus einer Summe von Abtastungen aus verschiedenen 3D-Texturen mit zufälligem Gradient-Rauschen (beispielsweise Perlin-Noise). Es werden gleich mehrere Rausch-Texturen mit unterschiedlichen Frequenzen und Amplituden verwendet, damit die endgültige

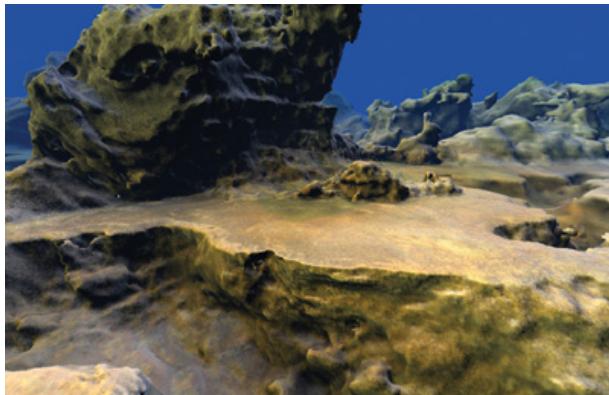


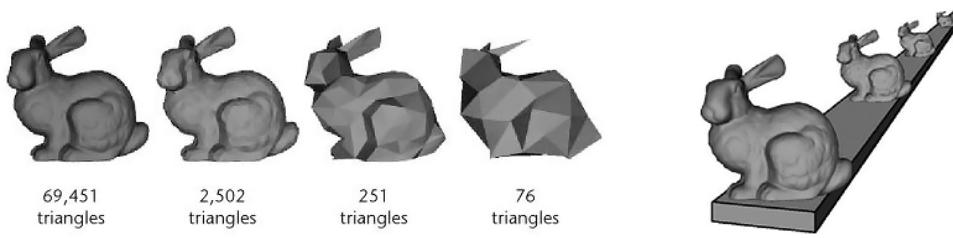
Abbildung 13: Eine mit der Methode von Geiss prozedural generierte Landschaft. Das Terrainvolumen wurde mit Voxeln gebaut und abschließend mit dem Marching-Cubes-Algorithmus polygonisiert. (Geiss, 2007)

Rauschfunktion aus mehreren Oktaven besteht, wodurch sie abwechslungsreicher wird. Geiss verwendet in seinem Beispiel neun Oktaven, wobei jede davon zur weiteren „Zerkniterrung“ des Terrains beiträgt, bis irgendwann Eigenschaften wie Bögen und Hügel entstehen.

Geiss (2007) präsentiert mehrere Methoden zur Einbettung von bestimmten Eigenschaften in die Landschaft. Durch eine feste Bestimmung des Grundniveaus werden alle Voxel unter dem festgelegten Wert solide, wodurch eine flache Fläche entsteht. Surrealistische Formen werden eingebaut, indem, noch vor der Abtastung der Rausch-Texturen, die Welt-Koordinaten mit einer anderen Rauschfunktion verzerrt werden. Mit der gezielten Verzerrung der Weltkoordinaten können auch terrassenähnliche Gebilde gebaut werden, wie auf der Abbildung 13 zu erkennen ist. Etliche andere Charakteristika können integriert werden, nicht nur durch eine Manipulation des Codes, sondern auch mithilfe von selbst erstellten 2D-Texturen, die beispielsweise Daten, wie die Frequenz der Rausch-Texturen, über einen Bereich beeinflussen. Für die Erstellung einer polygonbasierten Landschaft mit den Eigenschaften des Geiss-Terrains, wird genau das gleiche Voxelverfahren verwendet, mit abschließender Extraktion der Volumenisofläche.

4.2 Level of Detail

Wie Goswami (2012) erklärt, ist Level of Detail (LOD) eine Technik, in der die Komplexität einer Objektrepräsentation nach unterschiedlichen Krite-



(a) Dreiecksformen vom Stanford Bunny in vier verschiedenen LOD-Stufen.
(b) Die gleichen Modelle wie in (a), in der gleichen Anordnung, perspektivisch betrachtet.

Abbildung 14: Visualisierung der Grundlagen von LOD. (Luebke, 2001)

rien verringert wird, wie auf der Abbildung 14 am Beispiel eines Polygonmodells zu sehen ist. Ein übliches Kriterium für die Verringerung der Komplexität eines Objektes ist seine Entfernung zur Kamera. Des Weiteren nennt Goswami die Geschwindigkeit, mit der sich die Kamera bewegt, oder die Signifikanz der Topographie als Kriterien. LOD-Modelle besetzen weniger Arbeitsspeicher und, laut Goswami, verringern sie außerdem den Workload in der Grafikpipeline, weil weniger Vertextransformationen ausgeführt werden müssen. Da üblicherweise die Distanz der Kamera zum Modell sein LOD bestimmt, wird die Verringerung der Komplexität in entfernten Teilen des Objektes vom Betrachter gar nicht bemerkt. Es gibt mehrere Bereiche, in denen LOD implementiert werden kann, wie beispielsweise Texturen, jedoch wird in diesem Kapitel hauptsächlich Geometrie behandelt.

Voxel. Die hierarchische Struktur von Voxeln ist nicht nur bei der Kompression der Daten vorteilhaft, sondern auch, weil sie das LOD des Objektes zur Verfügung stellt. Octrees haben von Natur aus einen implizierten Detaillierungsgrad, der sich ideal für die Verringerung der Komplexität eines Objektes eignet. Die Umsetzung eines auf Voxeln basierenden LOD-Systems ist deshalb, im Vergleich zur Vereinfachung von Dreiecksgittern, relativ leicht.

Jabłoński und Martyn (2016) unterscheiden in ihrem LOD-Management-Algorithmus für SVOs zwischen zwei Schritten: der Evaluation, in der bestimmt wird, *wann* das LOD eines Objektes geändert werden muss, und der Transition, die sich darum kümmert, *wie* das LOD geändert wird.

Die Evaluation kann unterschiedlich umgesetzt werden. Eine Möglichkeit ist, laut Jabłoński und Martyn, eine stetige Evaluationsfunktion, die das LOD anhand der Entfernung der Kamera zum Objekt berechnet. Die Autoren geben als Beispiel eine lineare Funktion, in der y das gesuchte LOD des Objektes ist:

$$y = \max(0, \min(N - \frac{disc}{x}, N))$$

N steht dabei für die Gesamtanzahl der LODs des Objektes, $disc$ für die Entfernung der Kamera zum Objekt und x definiert die Distanz bzw. das Offset zwischen einzelnen LOD-Stufen. Ist das Ergebnis kleiner als 0, größer als N oder eine Dezimalzahl, wird es zum nächst erlaubten LOD-Integer mathematisch abgerundet oder trunkiert. Wie auch sonst üblich, ist LOD0 in dieser Formel die Stufe mit den wenigsten Details, während LODN den größten Detailgrad bietet. Jabłoński und Martyn (2016) merken an, dass die genannte Funktion auf der linearen Abhängigkeit zwischen einem LOD des Objektes und seiner Distanz zur Kamera basiere, was in einer perspektivisch betrachteten Szene keine wirklichkeitsgetreuen Ergebnisse liefert. Laut den Autoren ist in solch einem Szenario der Einsatz von Exponential- oder Potenzfunktionen sinnvoller.

Viel genauere Ergebnisse liefert jedoch ein ganz anderes Evaluationsverfahren, das anstelle der Distanz zum Objekt, die Anzahl an Pixeln, die einen einzigen Voxel darstellen, als Parameter verwendet. Mit Hilfe eines Compute-Shaders³ ist es laut Wright et al. (2012) möglich, die Anzahl an Pixeln zu berechnen, die ein Voxel beim Rendern einnehmen wird. Um diese Information nutzen zu können, suchen Jabłoński und Martyn (2016) zuerst einen Knoten im Octree, der sich im aktuellen LOD und in der aktuellen Lage des Modells am nächsten zum Betrachter befindet. Dieser Knoten, der auch *pattern node* genannt wird, wird bei der Evaluation des LODs verwendet. Jabłoński und Martyn entschieden sich für ein bildbasiertes Suchverfahren, bei dem sie in einem Offscreen-Buffer den Voxeltiefenwert z rendern und dabei zusätzlich die IDs der Voxel speichern. In der entstandenen Textur wird dann nach dem Knoten gesucht, der der Kamera am nächsten ist. Der zusätzliche Renderschritt lässt sich nach Jabłoński und Martyn vermeiden, indem die Tiefe der Voxel aus dem vergangenen Frame

³Ein Compute-Shader kann auf der Grafikkarte Berechnungen ausführen, die nicht direkt zu der Grafikpipeline gehören.

verwendet wird. Die Autoren erwähnen auch, dass zusätzliche Berechnungen zur Findung des Pattern-Nodes nötig sind, wenn die Oberflächenvoxel eine unterschiedliche Größe haben.

Die herausgefundene „Füllrate“ eines Voxels, also wie viele Pixel sich in einem Voxel befinden, lässt sich anschließend zur Wahl des nächsten LODs verwenden. Jabłoński und Martyn (2016) nutzen eine Formel, in der sie den Interpolationswert zwischen zwei LODs berücksichtigen. In einem Szenario mit sehr feinen Voxeln ist dies nicht nötig. Ein folgendes Vorgehen wäre ausreichend:

$$action = \begin{cases} \text{increase LOD} & \text{if } fillRate < minRate \\ \text{decrease LOD} & \text{if } fillRate \geq maxRate \\ \text{do nothing} & \text{otherwise} \end{cases} \quad (3)$$

Die Werte hängen natürlich vom Szenario ab und können beliebig angepasst werden. Für eine große Auflösung müssten allerdings kleine Werte gewählt werden, wie beispielsweise $minRate = 1$ und $maxRate = 4$.

Obwohl ein LOD-System in einem Octree impliziert ist, hat es doch gewisse Nachteile. Selbst wenn die Auflösung perfekt angepasst wurde und ein Pixel genau so groß wie ein Voxel ist, kann es dazu kommen, dass die Raumlage der Voxel nicht genau mit dem Gitter der Pixel übereinstimmt. In solchen Fällen würde nur eine Art bilineare Filterung zwischen den Voxeln genaue Farbwerte der Pixel zurückgeben. Dieses Problem wird im Kapitel 4.3 genauer behandelt. Ein weiterer Nachteil des LODs im Octree ist das Downsampling, also die Verringerung des Detailgrades durch Zusammenfassung der Kinderknoten zu einem Elternknoten. Abgesehen von Problemen, die bei der Bildung der Durchschnittsfarbe mehrerer Voxel entstehen (siehe Kapitel 4.3), gehen beim Downsampling bestimmte Daten verloren. Laut Laine und Karras (2011) resultiert die Bildung eines Mittelwertes von Normalen oder BRDFs (Materialeigenschaften, siehe Kapitel 4.5.2) mehrerer Voxel oft in unerwarteten Ergebnissen. Laine und Karras erwähnen jedoch ergänzend, dass die selben Probleme in der klassischen Computergrafik beim Mip Mapping von Normalen und BRDFs entstehen und sich die dafür existierenden Lösungen wahrscheinlich auf Voxel übertragen lassen.

Polygone. Die Erschaffung eines LOD-Systems für dreiecksbasierte Grafik ist ein breites Thema, das ausgiebig erforscht wurde. Zu der Forschung hat unter anderem ein Wissenschaftler namens David Luebke beigetragen, der in seinem Buch „Level of Detail for 3D Graphics“ (Luebke et al., 2003) ausführlich und strukturiert über existierende LOD-Herausforderungen schreibt und Algorithmen vorstellt, die diese Herausforderungen lösen. Luebke et al. konzentrieren sich primär auf dreiecksbasierte Grafik und unterscheiden darin grundsätzlich zwischen drei LOD-Arten. Dazu gehören das diskrete, das kontinuierliche (oder „progressive“, wie Hoppe (1997) diese Art erstmals genannt hat) und das betrachtungsabhängige LOD.

Das diskrete LOD ist eine Methode, in der jedes Modell beim Laden der Applikation im Rahmen einer Aufbereitung stufenweise vereinfacht und in dieser abgeänderten Form gespeichert wird. Dadurch existieren für jedes Modell mehrere Versionen in unterschiedlichen Detaillierungsgraden, die, je nach Evaluierung des LODs, eingesetzt werden. Luebke et al. (2003) nennen diese Methode „traditionell“, denn seitdem sie von Clark (1976) vorgestellt wurde, wird sie bis heute ohne Modifizierungen in den meisten 3D-Programmen eingesetzt. Diskretes LOD ist simpel zu implementieren, darüber hinaus kann das Austauschen des Polygonnetzes beim Laden eines neuen LODs praktisch sein, denn alle Daten, UV-Koordinaten und Texturen inbegriﬀen, können leicht ersetzt werden. Problematisch bei der diskreten Methode sind die sog. *popping* Artefakte, die, wenn das Austauschen von LODs abrupt und bemerkbar ist, entstehen. Dies kann mit Alpha-Blending der zwei LOD-Stufen oder mit komplizierteren Geomorphing-Techniken abgeschwächt werden.

Das kontinuierliche LOD folgt einem anderen Prinzip. Hoppe (1998), der Erfinder von *progressive meshes* (PM), war ein Pionier dieser Technik. Anstatt eine Menge an verschiedenen LODs beim Laden des Programms zu kreieren, baut Hoppe einen PM - eine Datenstruktur, die eher ein kontinuierliches Spektrum an Detail beinhaltet. Ein PM besteht aus einem Polygonnetz M^n , das mit Hilfe einer Sequenz von Simplifikations-Transformationen *ecol* (*edge collapse*), stufenweise zu M^0 vereinfacht wird. Hoppe visualisiert dieses Vorgehen mit folgender Formel:

$$M^n \xrightarrow{\text{ecol}_{n-1}} \dots \xrightarrow{\text{ecol}_1} M^1 \xrightarrow{\text{ecol}_0} M^0 \quad (4)$$

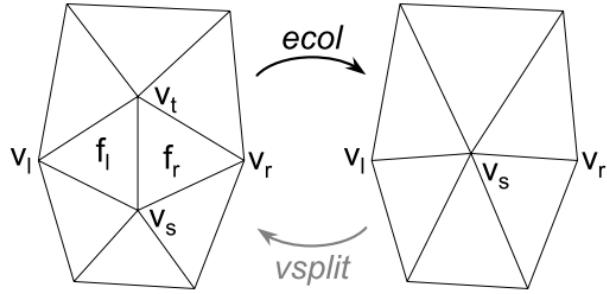


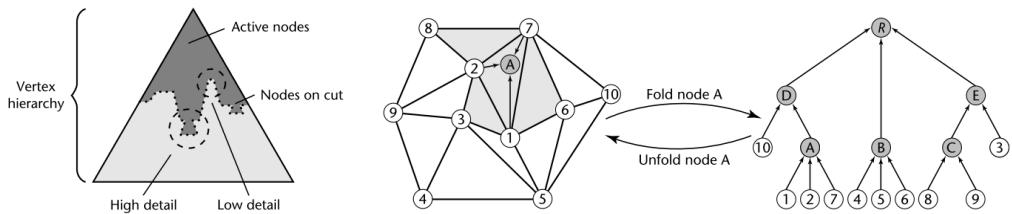
Abbildung 15: *Edge collapse* Transformation entfernt immer einen Vertex und zwei Polygonflächen, während ihre Inverse, *vertex split*, einen Vertex und zwei Polygonflächen hinzufügt. (Hoppe, 1998)

Da *edge collapse* eine Inverse hat, den *vertex split* (in der Formel *vsplit*), kann das grobe Modell wieder mit Details versehen werden:

$$M^0 \xrightarrow{vsplit_0} M^1 \xrightarrow{vsplit_1} \dots \xrightarrow{vsplit_{n-1}} M^n \quad (5)$$

Ein PM wird durch einen Tupel $(M^0, \{vsplit_0, \dots, vsplit_{n-1}\})$ repräsentiert. Das „Einstürzen“ der Kanten und „Teilen“ der Vertices wird auf der Abbildung 15 visualisiert. Die Reihenfolge der Transformationen wird von einem Optimierungsalgorithmus, der den Fehler zwischen M^n und M^{n-1} minimiert, bestimmt. Nach Luebke et al. (2003) ist die Granularität dieser Methode ihr größter Vorteil gegenüber diskretem LOD. Dadurch, dass ein genaues LOD vorhanden ist, werden die Ressourcen besser genutzt, denn es werden nie mehr Polygone als nötig gerendert. Außerdem, wie Hoppe (1997) zeigt, können PMs leicht mit Geomorphing erweitert werden, wodurch ein weicher Übergang zwischen den Transitionen gewährleistet wird. Der Nachteil vom kontinuierlichen gegenüber dem diskreten LOD ist, dass es mehr Verarbeitungs- und Speicherressourcen beansprucht.

Die dritte Vorgehensweise, das betrachtungsabhängige LOD, ist laut Luebke et al. (2003) eine Erweiterung der kontinuierlichen Methode. Hier wird abhängig von der Position und dem Blickwinkel des Betrachters das angemessene LOD dynamisch ausgewählt. Darüber hinaus nennen Luebke et al. das betrachtungsabhängige LOD anisotropisch, denn ein Objekt kann sich über mehrere Detaillierungsgrade spannen. Diese Methode bietet die beste Granularität und ist bei großen Modellen, wie Terrain, im Grunde unersetzlich. Im Vergleich zu ihren Alternativen beansprucht sie aber deutlich



(a) Eine vereinfachte Darstellung der Vertexhierarchie. Der „Schnitt“ durch die Hierarchie bestimmt den Detaillierungsgrad des Modells.

(b) Darstellung eines Polygonnetzes als Vertexhierarchie. Hier ist sichtbar, dass es nicht unbedingt ein Binärbaum sein muss. Mehrere Vertices (hier 1, 2 und 7) können zu einem Vertex (A) „zusammengefaltet“ werden. Dieser Prozess führt zur Auflösung der grau gekennzeichneten Polygone.

Abbildung 16: Schematische Darstellung des Aufbaus und der Funktionsweise einer Vertexhierarchie, die betrachtungsabhängiges LOD ermöglicht. (Luebke et al., 2003)

mehr Verarbeitungs- und Speicherressourcen. Zum Erscheinungszeitpunkt der Publikation von Luebke et al. basierten alle betrachtungsabhängigen LOD-Verfahren auf einem ähnlichen Konzept der Vertexhierarchie. Diese entsteht, indem die Vertices eines Polygonnetzes in der Initialisierungsphase der Applikation rekursiv zusammengefügt werden, zum Beispiel mit dem *edge collapse* Verfahren. In den Blättern der Hierarchie werden ursprüngliche Vertices gespeichert, während die zusammengefügten Vertices durch ihre Elternknoten ausgedrückt werden. Je näher diese zur Wurzel des Baumes sind, desto größer wird das Polygonnetz. Im Lauf der Applikation wird mit jeder Bewegung des Betrachters ein „Schnitt“ in der Vertexhierarchie gemacht, der das aktuelle LOD des Modells bestimmt (siehe Abbildung 16a). Die Dreiecke, die während den Simplifikations-Transformationen in der Initialisierungsphase gelöscht werden, werden bei dieser Gelegenheit gespeichert, um das „Falten“ und „Entfalten“ der Vertices in Echtzeit zu ermöglichen. Dieser Vorgang wird auf der Abbildung 16b dargestellt.

Abgesehen von den aufgezählten Arten von LOD-Techniken erwähnen Luebke et al. (2003) weitere Aspekte der LODs. Der Vereinfachungsalgorithmus kann die Topologie des Modells entweder erhalten, oder modifizieren. Er entscheidet sich für eine Vereinfachungsform basierend auf Genauigkeitsmetriken. Diese sind entweder auf die Erhaltung einer bestimmten Treue oder die Nichtüberschreitung einer bestimmten Anzahl an Polygonen ausgelegt. Folglich gibt es, außer *edge collapse*, andere lokale

Simplifikations-Transformationen, wie *vertex-pair collapse*, *triangle collapse*, *cell collapse*, *vertex removal* oder *polygon merging*. Dazu kommen globale Operatoren, die die Vereinfachung eines Modells, unter anderem mit Hilfe von Voxeln, ermöglichen. Die Reihenfolge der Transformationen wird in einem „Vereinfachungsprozess“ bestimmt, der unterschiedliche Mottos befolgen kann. Zuletzt ist zu erwähnen, dass die verwendete Fehlermetrik, wie z. B. die euklidische Distanz oder die Hausdorff-Metrik, einen großen Einfluss auf das Endergebnis hat.

Allgemein lässt sich sagen, dass die Erschaffung eines LOD-Systems für dreiecksbasierte Szenarios Probleme mit sich bringt, die bei Voxeln und Octrees nicht vorhanden waren. Dafür präsentieren sich Polygone aus der Nähe betrachtet besser, als würfelige Voxel. Ein Hybrid beider Systeme kann potenziell aus den Vorteilen beider Lösungen profitieren.

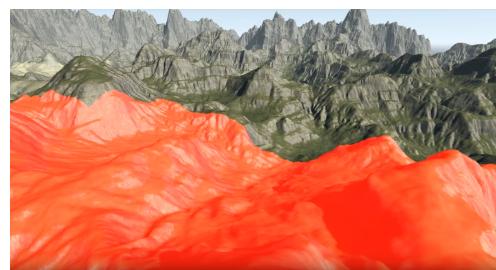
Hybrid. Bei der Erschaffung von Terrain sind Voxel aufgrund ihrer Datenstruktur vorteilhaft. Sie verfügen über ein impliziertes LOD und können, im Gegensatz zu Heightmaps, komplexe Strukturen wie Höhlen darstellen. Eine Voxellandschaft in einem Spielszenario, in dem der Spieler die komplette Kontrolle über die Kamera hat, muss jedoch eine sehr hohe Auflösung haben, damit sie selbst bei großem Zoomfaktor gut aussieht. So eine Datenstruktur würde sehr viel Speicherplatz beanspruchen.

Eine alternative Methode ist, die nächstgelegene Umgebung des Spielers, die im höchsten LOD dargestellt werden muss, in Echtzeit zu polygonisieren. Erst ab einer bestimmten Entfernung der Voxel zur Kamera werden diese direkt gerendert. Dieses hybride System wurde beispielsweise von Mulgrew (2014), dessen Arbeitsergebnisse auf der Abbildung 17 gezeigt werden, umgesetzt.

Die Polygonisierung von Voxeldaten kann mittels verschiedener Methoden realisiert werden. In den meisten Fällen wird die Oberfläche eines Modells mit Marching Cubes, einem älteren und wohlbekannten Algorithmus von Lorensen und Cline (1987), extrahiert. Der Name der Methode fasst ihre Funktionsweise gut zusammen: Ein Raum wird in diskreten Schritten von einem imaginären Würfel durchquert, der an seinen acht Ecken die Werte des Volumens abtastet. In einem typischen Spielszenario kann ein Voxel entweder solide oder leer sein, weshalb jede Ecke des Marching Cubes entweder innerhalb oder außerhalb des Volumens ist. Insgesamt gibt



(a) Darstellung der Spielerperspektive. Die nächstgelegende Umgebung ist polygonisiert, während entfernte Bereiche als Voxel gerendert werden.



(b) Spielerperspektive mit Markierung des polygonisierten Bereiches.



(c) Zoom auf die Grenze zwischen zwei LODs.

Abbildung 17: Ein hybrides LOD-System mit einem SVO, das partiell polygonisiert wird. (Mulgrew, 2014)

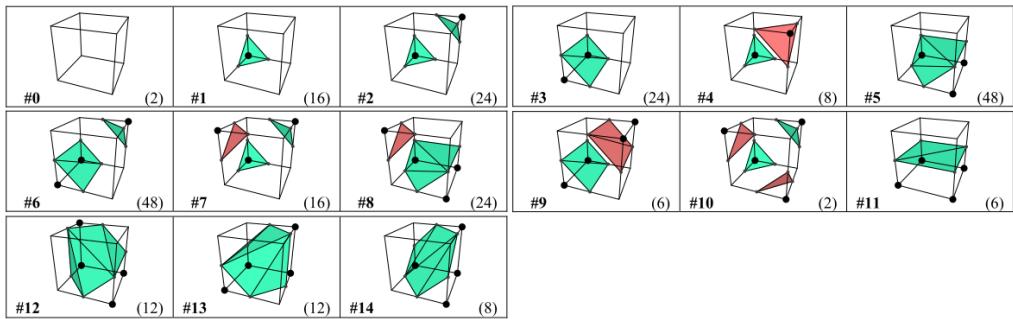
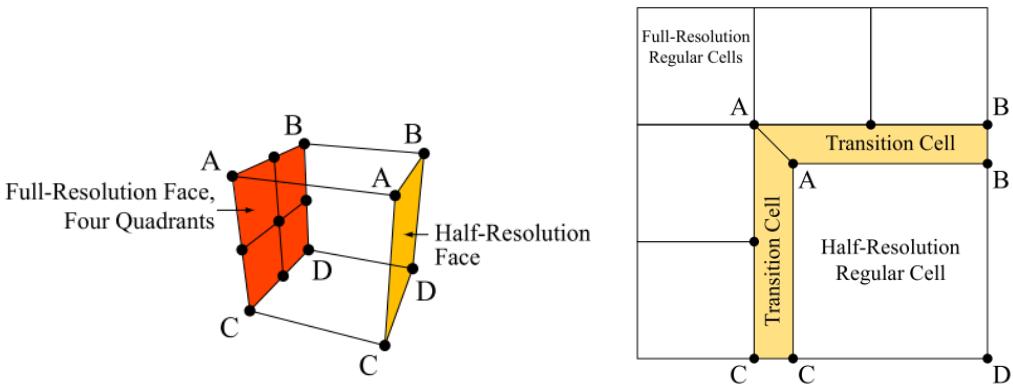


Abbildung 18: Die Äquivalenzklassen des Marching Cubes Algorithmus. Schwarze Punkte markieren Ecken, die sich innerhalb des Volumens befinden. Die Vorderseiten der Polygone sind grün, die Hinterseiten dagegen rot. Die Zahl links unten ist der Index der Klasse, während rechts unten die Anzahl der äquivalenten Kombinationsmöglichkeiten, die die Klasse darstellt, vermerkt wird. (Lengyel, 2010)

es bei acht Ecken mit einem Booleschen Wert $2^8 = 256$ Kombinationsmöglichkeiten. Bei zweien davon, wenn alle Ecken entweder innerhalb oder außerhalb des Volumens sind, hat der Algorithmus nichts Weiteres zu tun. Geometrie in Form von Dreiecken muss nur bei den restlichen 254 Möglichkeiten produziert werden. Aufgrund von gemeinsamen geometrischen Ähnlichkeiten lassen sich jedoch alle 256 Möglichkeiten auf 15 Äquivalenzklassen zusammenfassen. Somit gehören zu einer Klasse eine Kombination und ihre Inverse, sowie auch alle ihr gleichenden Kombinationen, die sich nur um eine Rotation um die x - y - z -Achsen des Würfels unterscheiden. Für jede Äquivalenzklasse wird das Aussehen ihres Polygonnetzes kodiert und in einer Tabelle gespeichert, in der die IDs der Items beispielsweise aus den Binärwerten der Würfecken gewonnen werden können. Die 15 Äquivalenzklassen sind auf der Abbildung 18 zu sehen.

Es gibt allerdings mehrdeutige Fälle, in denen es nicht ersichtlich ist, wie die Vertices, die sich mitten auf den Würfelkanten befinden, verbunden werden sollten. Diese Fälle richtig und schnell zu behandeln ist nicht wirklich trivial, doch gibt es eine Vielzahl von Wissenschaftlern, wie z. B. Lengyel (2010), die sich mit diesem Problem auseinandersetzen und ihren eigenen modifizierten Marching Cubes Algorithmus vorschlagen.

Damit weit entfernte Polygone größer sind, wird mit absteigendem LOD der Marching Cube vergrößert. Das Problem eines polygonisierten Voxelterrains sind Risse, die an den Grenzen von unterschiedlichen LODs entstehen.



(a) Topologisches Layout einer Transitionszelle. Die Seite, die mit kleineren Voxeln (mit hoher Auflösung) benachbart ist, ist rot gekennzeichnet, während die orangefarbene zu großen Voxeln (mit niedriger Auflösung) gerichtet ist.

(b) Die Lage einer Transitionszelle, die einen großen Voxel mit kleinen Voxeln verbindet.

Abbildung 19: Eine Transitionszelle wird zwischen Voxeln unterschiedlicher Auflösung platziert, um ihre Geometrie lückenlos zu verbinden. (Lengyel, 2010)

Die Polygonoberfläche ist schließlich nur eine Annäherung an die eigentliche Volumenoberfläche und wird genauer, je kleiner der Marching Cube ist. Wird dieser vergrößert, passen die Vertices des niedrig aufgelösten Würfels nicht mehr mit den Vertices seines höher aufgelösten Nachbarn überein. *Stitching*, das Füllen der Lücke mit Polygonen, ist eine triviale Lösung, die jedoch laut Lengyel (2010) Schattierungsartefakte hervorruft und sich besser für Heightmaps als für Voxeldaten eignet. Lengyel löst das Problem der Risse mit Hilfe von *transition cells*, deren Aufbau auf der Abbildung 19a zu sehen ist. Wird diese Transitionszelle wie ein Marching Cube bezüglich der Booleschen Werte an ihren Ecken betrachtet, ergeben sich daraus $2^9 = 512$ mögliche Kombinationen, die wiederum 73 Äquivalenzklassen bilden. Lengyel visualisiert das Aussehen all dieser Klassen in seiner Publikation. Die Breite der Transitionszelle sollte frei einstellbar sein, doch wenn sie 0 ergibt, ist das Ergebnis äquivalent zum klassischen *stitching*. Bessere Ergebnisse werden erreicht, wenn die Breite der Transitionszelle einen großen Bruchteil der Breite ihres hochauflösten Voxels ergibt. Der letzte Schritt des Algorithmus ist, die Voxel mit niedriger Auflösung an den LOD-Grenzen runterzuskalieren, um den Platz für Transitionszellen zu schaffen, wie die Abbildung 19b zeigt.

Wirkt das Terrain nach der Polygonisierung kantig, liegt es entweder an der niedrigen Auflösung der Voxel, oder den zu kleinen Marching Cubes. Eine Nachbearbeitung mit einem Algorithmus zur Glättung des Polygonnetzes oder der Vertexnormalen kann die Landschaft zusätzlich abfeilen. Die Polygonisierung mit Transitionzellen nach Lengyel (2010) ist in Echtzeit ausführbar und in dieser Form in Lengyels alter Engine C4 und seiner neuen Tombstone Engine implementiert. Visuell liefert das Hybridsystem Ergebnisse, die unabhängig von der Perspektive eine gute Auflösung haben, gleichzeitig behält das Terrain die Vorteile von Voxeln bei.

4.3 Texturieren

Polygone. Um die Dreiecke mit mehr Realismus auszustatten, werden auf ihnen Texturen angebracht, die zusätzliche Details beinhalten. Dieser Prozess ist das Texture Mapping, das erstmals von Catmull (1974) auf „gewölbten“ Oberflächen angewandt wurde. Eine Textur ist meistens zweidimensional und enthält Graustufen, RGB oder RGBA Daten. Werden diese direkt auf die Oberfläche des Objektes übertragen, handelt es sich um die Urform des Texture Mappings, auch *diffuse mapping* genannt. Mittlerweile sind aber zahlreiche andere Formen, wie beispielsweise *normal mapping*, *bump mapping*, *displacement mapping*, *environment mapping*, *light mapping* oder *occlusion mapping*, entstanden. Mehrere dieser Formen können gleichzeitig an einem Objekt angewandt werden. Solch eine „Multitextur“ ermöglicht die Erschaffung eines fotorealistischen Abbildes eines Low-Poly-Objektes in Echtzeit.

Heckbert (1989) dokumentierte den Verlauf von Texture Mapping detailliert in seiner Arbeit. Darin beschreibt er als ersten Schritt die Parametrisierung der Fläche (englisch *surface parametrization*), in der die Texturfläche auf das Objekt gemappt wird. Dabei wird jeder Vertex eines Modells mit UV-Koordinaten aus dem Texturraum verbunden. Die Art der gewählten Parametrisierung bestimmt das Aussehen der Textur auf der Objektoberfläche.

Lansdale (1991) unterscheidet zwischen der bedingten (englisch *induced*) und der expliziten Parametrisierung. Laut seiner Arbeit wird das bedingte Mapping aus einer n -Point Korrespondenz zwischen dem Textur- und Objektraum und aus dieser Korrespondenz folgenden, geometrischen Abbil-

dung geschlussfolgert. Diese Abbildung kann wiederum affin oder projektiv sein und mit einer Matrix ausgedrückt werden. Explizites Mapping kann dagegen, laut Lansdale (1991), durch eine Zusammensetzung von analytischen Formeln, die in endlicher Zeit berechnet werden können, definiert werden.

„Two-Part Texture Mappings“ ist eine Arbeit von Bier und Sloan (1986), in der eine Umsetzung von expliziter Parametrisierung detailreich beschrieben wurde. Die Idee dabei ist, das Mapping der Texturfläche $[u, v]$ auf die Objektfläche $[x, y, z]$ in zwei Abschnitte zu unterteilen: Abschnitt S und Abschnitt O . S steht für „mapping to intermediate surface“ und bedeutet, dass zuerst die Abbildung der Textur auf eine Zwischenfläche, bzw. $[u, v] \rightarrow [x_s, y_s, z_s]$, gefunden werden muss. O ist eine Abkürzung für „mapping to object“, was die letztendliche Abbildung $[x_s, y_s, z_s] \rightarrow [x, y, z]$ bedeutet. Zur Vereinfachung der Parametrisierung wird demzufolge eine Zwischenfläche verwendet. Diese kann unterschiedliche Formen annehmen, sollte jedoch in einer Parameterdarstellung ausgedrückt werden können. Bier und Sloan (1986) liefern als Beispiel eine Formel zur Parametrisierung eines Zylinders:

$$S : [u, v] \rightarrow \left[\frac{c}{r}u + \theta_0, dv + h_0 \right], \quad \text{wenn } -\pi r < u \leq \pi r. \quad (6)$$

Die Inverse dieser Parametrisierung lautet:

$$S^{-1} : [\theta, h] \rightarrow \left[\frac{r}{c}(\theta - \theta_0), 1/d(h - h_0) \right], \quad \text{wenn } -\pi < \theta < \pi. \quad (7)$$

Der Wert r steht für den Radius des Zylinders, während $[c, d]$ die Skalierung und $[\theta_0, h_0]$ die Position der Textur definieren. Bier und Sloan (1986) beschreiben zusätzlich, wie sich Texturen auf andere geometrische Formen wie Würfel oder Sphären abbilden lassen. Die Visualisierung letzterer ist auf der Abbildung 21a zu sehen.

Nachdem die Zwischenfläche S definiert wurde, kann sie anschließend auf das Zielobjekt mit O -Mapping projiziert werden. Dies kann, wie die Abbildung 20 darstellt, auf vierfache Weise erfolgen.

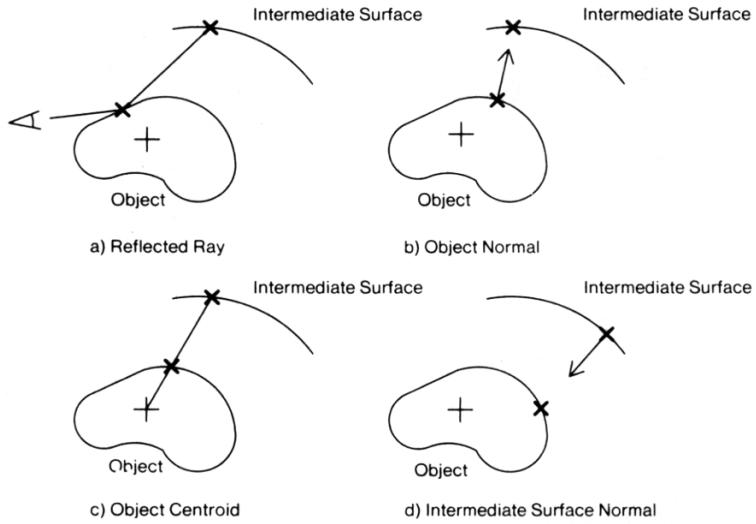
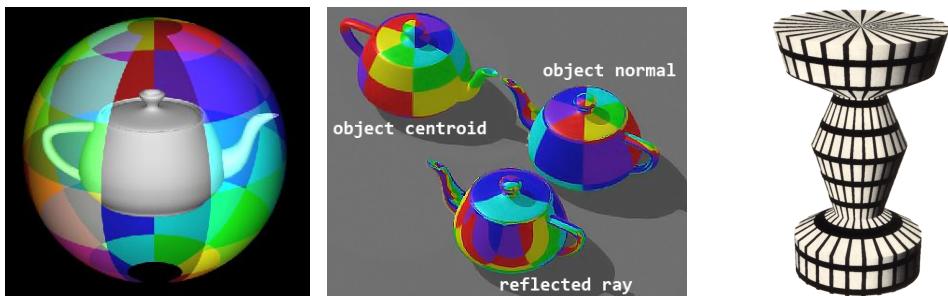


Abbildung 20: (a) *Reflected ray* - hier wird ein Strahl aus der Kamera auf das Objekt geworfen, der anschließend reflektiert wird und auf die Zwischenfläche fällt. Das Mapping erfolgt in der O^{-1} -Richtung.
 (b) *Object normal* - das Mapping findet statt, indem die Normale einer Objektstelle auf die Zwischenfläche fällt. O^{-1} -Richtung.
 (c) *Object centroid* - durch die Objektstelle verläuft eine Gerade, die auch durch das Zentrum des Objektes verläuft. Der Schnittpunkt der Geraden mit der Zwischenfläche wird mit der Objektstelle gemappt. O^{-1} -Richtung.
 (d) *Intermediate surface normal (ISN)* - hier wird die Normale eines Punktes auf der Zwischenfläche verfolgt, bis sie auf das Objekt fällt. O^1 -Richtung.
 (Bier und Sloan, 1986)

Reflected ray, object normal, object centroid und intermediate surface normal sind vier Arten, wie eine Zwischenfläche auf eine Zielfläche abgebildet werden kann. Ein Beispielergebnis dieses Mappings mit einer sphärischen und zylindrischen Zwischenfläche wird auf der Abbildung 21a und 21c präsentiert.

Aufgrund des Mappings einer zweidimensionalen Fläche auf ein dreidimensionales Objekt entstehen meistens Verzerrungen und Diskontinuitäten. Wie stark diese sind, hängt von der Form der Zwischenfläche, der Objektfläche und der Art des O-Mappings ab. Bier und Sloan (1986) deuten an, dass das ISN-O-Mapping am besten mit Zwischenflächen in Form eines Zylinders oder Würfels funktioniere, während *object centroid* Mapping besser an Würfeln oder Sphären ausführbar sei. Was aber letztendlich am besten geeignet ist, hängt von dem gewünschten Ergebnis ab.



- (a) Ein Objekt, umhüllt mit einer sphärischen Zwischenfläche, die bereits aus einer Textur parametrisiert wurde. (Wolfe, 1997)
- (b) Ergebnisse dreier verschiedener O-Mappings, die *normal* als O-Mapping auf der Abbildung 20 erläutert wurden. Ihre Zwischenfläche ist auf (a) zu sehen. (Wolfe, 1997)
- (c) *Intermediate surface* mit zylindrischer Zwischenfläche. (Bier und Sloan, 1986)

Abbildung 21: Visualisierungsbeispiele für die Two-Part-Texture-Mapping-Technik.

Der zweite Aspekt von Texture Mapping ist nach Heckbert (1989) die Projektion aus dem lokalen Raum des Objektes auf den Bildschirmraum (englisch *screen space*). Die Projektion verursacht im Rasterisierungsverfahren ein Problem, das hier im Folgenden erläutert wird. Um in einem dreidimensionalen Raum Attribute (z. B. Farben oder Normalen) eines bestimmten Punktes S in einem Dreieck ABC herauszufinden, wird zwischen den Vertices linear interpoliert. Wird aber ABC mit Hilfe einer perspektivischen Matrix auf den Bildschirmraum projiziert, liefert die lineare Interpolation auf der Stelle S Ergebnisse, die nicht mehr den Ergebnissen aus der vorigen Berechnung gleichen. Dieses Problem ist auf der Abbildung 22, als auch 23 visualisiert. Trotz der visuellen Inkorrekttheit ist diese affine Abbildung schnell, einfach und war laut Heckbert in den 80er Jahren populärer als andere, korrekte Methoden. Heutzutage muss jedoch die perspektivische Korrektheit gesichert sein.

In seinem Lösungsvorschlag geht Low (2002) davon aus, dass die Kamera fest fixiert und ihre Ausrichtung parallel zu der z -Achse ist. Dadurch können z -Werte der Pixel zur Darstellung einer korrekten Perspektive verwendet werden. Low nutzt unter anderem Kehrwerte, um die z -Werte herauszufinden und zwischen den Attributwerten im Bildschirmraum zu interpolieren. Am Beispiel der Abbildung 23 hat Low folgende Formel zur Berechnung des z -Wertes herausgearbeitet:

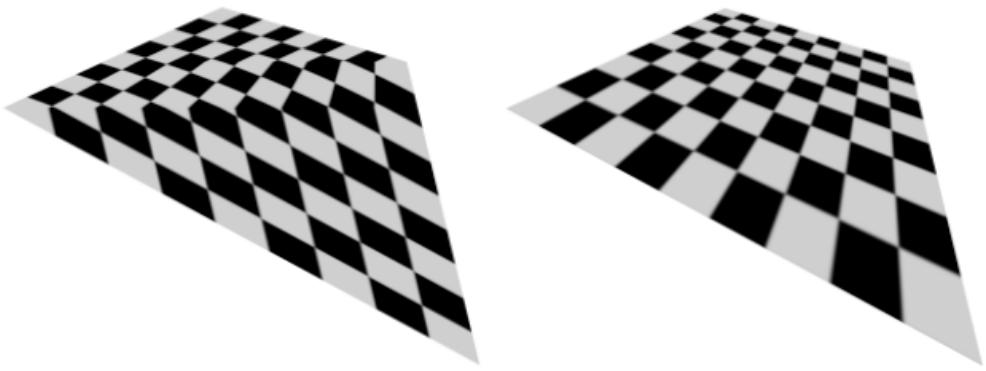


Abbildung 22: Eine lineare Interpolation zwischen den Attributen im Bildschirmraum führt zu einer affinen Abbildung der Textur (links). Eine perspektivisch korrekte Lösung (rechts) erfordert das Berücksichtigen der Projektion. (Lansdale, 1991)

$$Z_t = \frac{1}{\frac{1}{Z_1} + s \left(\frac{1}{Z_2} - \frac{1}{Z_1} \right)} \quad (8)$$

Z_t wird anschließend in folgender Formel zur Berechnung des Attributwertes (in diesem Fall UV-Koordinaten) I_t verwendet:

$$I_t = \left(\frac{I_1}{Z_1} + s \left(\frac{I_2}{Z_2} - \frac{I_1}{Z_1} \right) \right) / \frac{1}{Z_t} \quad (9)$$

An dieser Stelle kann mithilfe von I_t das richtige Texel in der Textur gefunden und ausgegeben werden.

Leider kann es selbst mit einer cleveren Parametrisierung und perspektivisch korrekten Projektion bei der Bildsynthese eines texturierten Objektes zu visuellen Fehlern kommen. Diese entstehen aufgrund von einer Unterabtastung der Textur. Heckbert (1989) erklärt das Phänomen im Folgenden: Die reale Welt, so wie wir sie sehen, ist eine kontinuierliche räumliche Domäne, während eine Textur, in Form eines Arrays, in diskreten Schritten abgebildet wird. Wird die Nächste-Nachbarn-Abtastung durchgeführt, ist das Ergebnis ungenau und es kommt zu Artefakten, die besonders bei animierten Objekten stören. Es gibt mehrere Methoden diese Fehler zu minimieren. Eine bekannte Praktik, die Heckbert (1989) auch beschrieben hat, ist die Speicherung der Texturen in mehreren Auflösungen, wie bei Mip Mapping

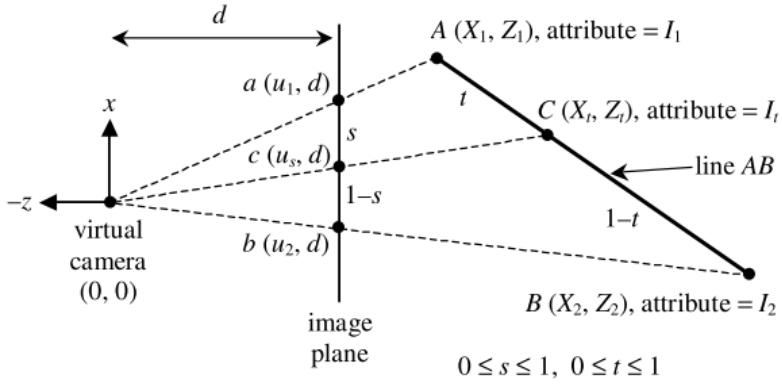


Abbildung 23: Eine lineare Interpolation der Werte im Bildschirmraum liefert keine perspektivische Korrektheit. Linie AB wird auf den Bildschirm projiziert, wodurch eine neue Linie ab entsteht. Der Punkt c befindet sich in der Mitte von ab , obwohl seine Ursprungsabbildung C näher an A als an B liegt. Die Werte s und t werden zur linearen Interpolation verwendet. (Low, 2002)

oder anisotroper Filterung. Somit kann für die Abtastung immer die Textur ausgewählt werden, in der ein Texel die Größe des Pixels hat. Weitere Methoden, wie *full-scene anti-aliasing (FSAA)* oder *multisample anti-aliasing (MSAA)* eliminieren Aliasing-Effekte, indem sie die Szene in einer höherer Auflösung rendern, um sie später runterzuskalieren. Abschließend kann auch ein mathematischer Filter, der während der Abtastung die Farbe der Nachbartexel mit einberechnet, verwendet werden. Heckbert (1989) gibt als Beispiel die Formel des Gauß-Filters:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \quad (10)$$

Der Wert σ steht für die Varianz der Gauß-Verteilung und bestimmt den Glättegrad. Das x ist dabei die horizontale und y die vertikale Distanz vom Mittelpunkt.

Voxel. Die Texturierung von Voxelmodellen ist in der Theorie viel unkomplizierter und selbstverständlicher als die Texturierung von Dreiecken. Crassin (2011) merkt an, dass die Parametrisierung eines Modells in voxelbasierter Form unnötig ist, denn die Position der Oberflächenvoxeln und Texturkoordinaten im dreidimensionalen Raum sind gleich. Crassin bezeich-

net Perlin (1985) und Peachey (1985), anhand ihrer unabhängigen Publikationen im Jahr 1985, als Pionieren der „soliden Texturen“. Sie waren die ersten, die Texturen als dreidimensionales Uniform Grid gespeichert haben. Der Speicherverbrauch dieser Datenstruktur wächst kubisch mit ihrer Auflösung, trotz dessen ist das Uniform Grid laut Crassin eine übliche Art dreidimensionale Texturen zu implementieren. Benson und Davis (2002), sowie auch DeBry et al. (2002), waren laut Crassin im Jahr 2002 die ersten, die dreidimensionale Texturen in einer optimierten Datenstruktur (Octree) gespeichert haben. Wie oft in einem SVO speichern Benson und Davis nur die Oberflächenvoxel, um die Größe des Octrees möglichst gering zu halten. Da die Geometrie von Voxelmodellen in der Regel bereits als SVO gespeichert wird, braucht die Textur nicht mehr vom Modell getrennt zu bleiben. Das „Texturieren“ heißt in diesem Fall nichts anderes, als jeden existierenden Voxel eine Farbe zuzuteilen.

Es existieren mehrere erwähnenswerte Aspekte dieser Texturierung. Benson und Davis (2002) verwenden in ihrer Studie Octree-Texturen auf dreiecksbasierten Modellen, wobei sie bei jedem Messpunkt auf der Oberfläche die Farbe von 3x3x3 benachbarten Voxeln interpolieren, um die Textur zu glätten. Dies sollte bei hochauflösten Voxelmodellen zwar nicht nötig sein, aber bei niedrigauflösten Modellen ist dieses Verfahren, auch wenn schwer umzusetzen, durchaus sinnvoll.

Eine besondere Stärke von Octree-Texturen liegt in ihrem hierarchischen Aufbau, der als eine dreidimensionale Äquivalente zu MIP-Maps gesehen werden kann. Die Voxel der höchsten Auflösung beinhalten eine eigene Farbe, während die Durchschnittsfarbe aller Geschwistervoxel meistens in ihrem Elternteil gespeichert (wie in Crassin et al. (2011)) wird. In Folge einer Filterung wird diese Durchschnittsfarbe bei der Bildsynthese verwendet, wenn das Modell aus einer größeren Entfernung betrachtet wird. Das Problem stellen die Voxel dar, die sich nahe sind, jedoch nicht zur selben Oberfläche gehören, beziehungsweise eine unterschiedliche Ausrichtung haben. Die Verwendung der Durchschnittsfarbe aller benachbarten Voxel kann dann falsche Ergebnisse liefern, wie auf der Abbildung 24 visualisiert wurde. Die von Benson und Davis (2002) vorgeschlagene Lösung des Problems ist vereinfachte Voxelnormalen mitzuspeichern und die generelle Ausrichtung der Oberfläche bei der Berechnung der Durchschnittsfarbe zu berücksichtigen.

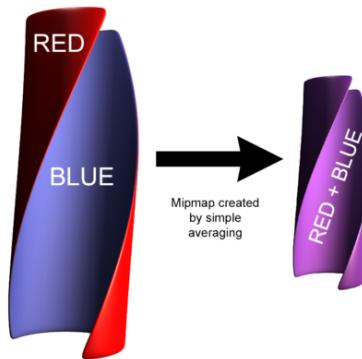


Abbildung 24: Wird eine Durchschnittsfarbe der benachbarten Voxeln bei der Filterung verwendet, kann es dazu kommen, dass eine unsichtbare Fläche die Farbe einer sichtbaren Fläche beeinflusst. Auf der linken Seite der Abbildung befindet sich ein Objekt, das flach ausgelegt aus einer Entfernung je nach Perspektive entweder blau oder rot erscheinen sollte. Rechts davon ist das gleiche Objekt nach einer trivialen Filterung zu sehen. (Benson und Davis, 2002)

Die Funktionsweise von Octrees unterscheidet sich in einer Hinsicht stark vom klassischen Mip Mapping. Eine zweidimensionale MIP-Map besteht aus einer Textur in Originalauflösung und ihrer mehrmals verkleinerten Kopien, die in dem Prozess der Skalierung mit einem bilinearen oder ähnlichen Filter behandelt wurden. Alleine die Bildung der Durchschnittsfarben aller Kinderknoten in einem Octree ist jedoch keine klassische Filterung in engeren Sinne. Die Blätter in einem Octree werden so lange zusammengefasst, bis ein Voxel der Größe eines Pixels entspricht. Wenn der Voxel jedoch leicht verschoben wird und nur noch die Hälfte des Pixels überdeckt, wird keine Filterung auf ihm und seinem Nachbar angewandt. Nur wenn der Pixel eine Durchschnittsfarbe der zwei Voxel bekommt, können flickernde Aliasing-Effekte vermieden werden. Solche können z. B. auf dem Video von Mulgrew (2014) aus der Abbildung 17 in weit entfernen LODs beobachtet werden. Bilineare Filterung von Voxeln ist nicht unbedingt leicht umzusetzen. Mehr dazu im Kapitel 4.5.4.

Eine weitere Problematik ist das Entwerfen von dreidimensionalen Texturen an sich. Um nur die Oberfläche eines Modells zu texturieren, muss eine Software vorhanden sein, die das Kolorieren von Voxeln im dreidimensionalen Raum ermöglicht. Soll ein Modell modifizierbar sein, wie es z. B. bei Terrain sinnvoll ist, muss auch sein Inneres texturiert werden.

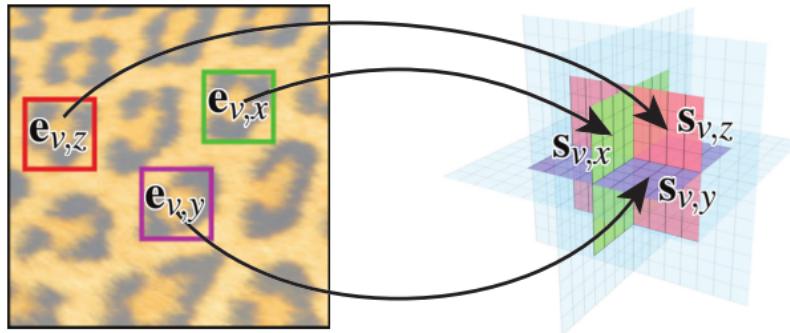


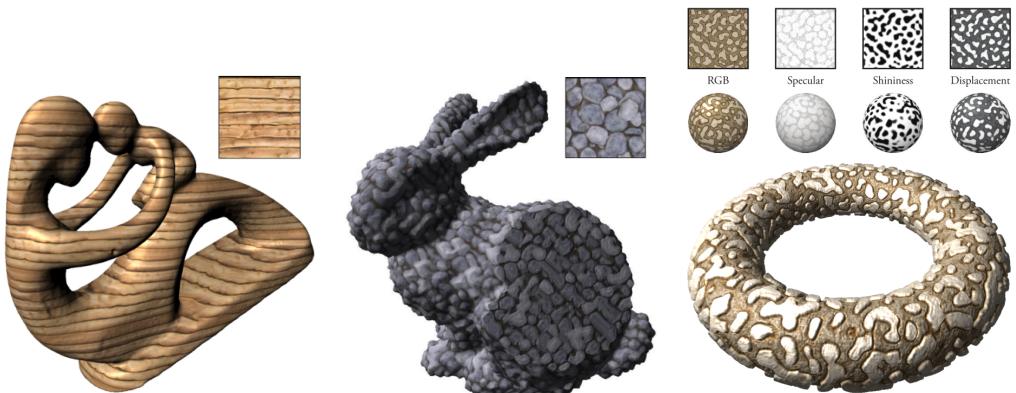
Abbildung 25: Visualisierung der Nachbarschaften eines Voxels im synthetisierten Volumen und das Exemplar. (Kopf et al., 2007)

Eine manuelle Erstellung solch einer Textur ist kompliziert, weshalb oft in diesem Fall zu prozeduralen Lösungen gegriffen wird. Eine interessante Alternative haben Kopf et al. (2007) in ihrer Studie vorgestellt. Sie erforschen wie aus einer zweidimensionalen (Exemplar genannt) eine dreidimensionale (solide) Textur synthetisiert werden kann. Zuerst bilden sie einen Würfel, der durch ein reguläres Grid auf Voxel konstanter Größe geteilt wird. Jeder Voxel speichert am Anfang eine zufällige Farbe aus dem Exemplar. Anschließend wird die Nachbarschaft eines Voxels iterativ verändert, basierend auf der ähnlichsten gefundenen Nachbarschaft des Exemplars. Das Ziel ist die Energiefunktion, die den Unterschied zwischen dem Exemplar und dem dreidimensionalen Volumen bemisst, zu minimisieren. Für ein Exemplar e und ein synthetisiertes Volumen s sieht die Energiefunktion von Kopf et al. folgendermaßen aus:

$$E(a, \{e\}) = \sum_v \sum_{i \in \{x,y,z\}} \|s_{v,i} - e_{v,i}\|^r \quad (11)$$

Dabei ist s_v ein Voxel, dessen zu x-, y- und z-Achse orthogonale, vektorisierte Nachbarschaften jeweils als $s_{v,x}$, $s_{v,y}$ und $s_{v,z}$ ausgedrückt werden. Der Wert $e_{v,i}$ steht für die Nachbarschaft aus dem Exemplar, die am ähnlichsten zu $s_{v,i}$ ist, während r mit dem Wert 0.8 ein Optimierungsfaktor ist. Die Abbildung 25 visualisiert die Beziehung zwischen $s_{v,i}$ und $e_{v,i}$.

Der Algorithmus erfolgt iterativ in zwei Schritten. Zuerst wird die ähnlichste Nachbarschaft $e_{v,i}$ gesucht, um abschließend darauf basierend ein neuer Wert für s_v gewählt. Kopf et al. (2007) beschreiben im Detail beide Phasen, sowie eine weitere Optimierung des Verfahrens durch Einbezie-



(a) Das Modell der Mutter mit einem Kind sieht aus, wie aus Holz geschnitzt.

(b) Ein Stanford Bunny aus granuliertem Stein. Der Schnitt zeigt das texturierte Innere.

(c) Ein Beispiel für verschiedenartige Kanäle einer Textur.

Abbildung 26: Drei Modelle, ausgestattet mit einer dreidimensionalen Textur, die aus einem zweidimensionalen Bild synthetisiert wurde. (Kopf et al., 2007)

hung eines Histogramms, das eine zusätzliche Berücksichtigung der globalen Merkmale bietet. Nach mehreren Iterationen entsteht ein dreidimensionaler Würfel, der im Idealfall in jedem Schnitt seines Volumens ähnlich zu dem Exemplar ist. Abschließend können die Texel ohne weitere Parametrisierung direkt auf Voxel eines Modells übertragen werden. Das Verfahren von Kopf et al. bietet für viele Exemplare visuell erstaunlich gute Ergebnisse, wie auf der Abbildung 26 zu sehen ist. Leider erfordert das Speichern einer hochauflösten Textur im Inneren des Modells viel Speicher.

Das Texturieren eines Voxelmodells durch Zeichnen oder Abbildung auf eine solide Textur bringt gewisse Vorteile mit sich. Die Künstler sparen Zeit, weil kein UV-Mapping durchgeführt werden muss, und es entstehen keine Verzerrungen oder „Säume“ infolge der Parametrisierung. Es gibt jedoch auch Probleme mit dieser Art von Texturierung. Es muss beachtet werden, dass zwei unterschiedlich texturierte aber geometrisch gleiche Modelle nicht unbedingt den selben Aufbau in der Datenstruktur haben. Wird z. B. ein Würfel, der aus einem Voxel besteht, mit einer höher aufgelöster Textur versehen, muss er neu berechnet werden, da seine Daten nicht mehr homogen sind. Dies ist vor allem bei flachen, nicht organischen Strukturen, wie z. B. Gebäuden, oft der Fall. Speicherplatztechnisch ist diese Lösung nicht optimal, weil die Schlichtheit der Geometrie nicht mehr ausgenutzt wird.



(a) Ein Screenshot aus Landmark, einem mit Voxel-Farm-Plugin gemachten Spiel. Der Boden ist ein Beispiel für ein Modell mit geringem Geometrie- und hohem Texturdetail.

(b) Ein Stück Mauer, das nach einer interaktiven Modifizierung automatisch neu texturiert wurde.

Abbildung 27: Bilder von Cepero (2016) präsentieren, warum Speichern einer Farbe pro Voxel problematisch ist und wie dieses Problem mit Polygonisierung und prozeduralem UV-Mapping gelöst werden kann.

Auf der Abbildung 27a ist ein Beispiel zu sehen, in dem Geometrie und Textur einen komplett unterschiedlichen Grad an Detail haben.

Über dieses Problem reflektiert in seinem Blog der Entwickler von Voxel Farm⁴, Cepero (2016). Er erwähnt, dass Technologien wie MegaTextures von id Software zwar die Erstellung eines Voxels pro Texel unterstützen können, dennoch ist dies weiterhin ineffizienter als klassische Polygonisierung und UV-Mapping. Aus diesem Grund setzte Cepero in Voxel Farm letztere Technik um. Leider geht er in seinem Beitrag nicht ins Detail, merkt aber an, dass die Umsetzung langwierig war und viele Tricks erforderte. Die größte Herausforderung war dabei, dass die Spielwelt interaktiv modifizierbar sein soll. Die Ergebnisse seiner Arbeit veranschaulicht er in einem Video, von dem das Einzelbild aus der Abbildung 27b stammt.

Vergleich. Das größte Problem, das das Speichern einer Farbe pro Voxel mit sich bringt, ist der Speicherverbrauch dieser Methode. Modelle, die simple und flache Formen haben, aber detailreich texturiert wurden, können von der Spärlichkeit ihrer Geometrie speicherplatztechnisch nicht profitieren. Vor allem in diesen Fällen scheint dreiecksbasierte Grafik effizienter zu sein. Diese leidet aber an einem langwierigen Parametrisierungsprozess, der

⁴Ein Plugin für Unreal Engine, mit dem Voxelwelten erschafft werden können.

des Öfteren zu Verzerrungen und „Säumen“ führen kann. Auch die Modifizierung von Polygonmodellen ist problematisch, denn nur ihre Oberfläche ist von Texturen bespannt. In diesem Zusammenhang sind solide Voxelmodelle mit texturiertem Inneren praktischer. Die Stärken und Schwächen beider Praktiken sind dementsprechend sehr stark von der Art der Modelle, Texturen und der Anwendung abhängig. Eine Alternative sind hybride Systeme wie in Voxel Farm, in denen modifizierbare Modelle vor der Bildsynthese polygonisiert und automatisch texturiert werden.

4.4 Bewegung

Der Vorgang, eine dreidimensionale Szene in Bewegung zu bringen und sie abschließend in Frames zu rendern, erzeugt eine Illusion von Dynamik. Be- trachtet man Dynamik, kann im Allgemeinen zwischen ihren zwei Arten unterschieden werden: „simple“ Bewegungen, wie euklidische Transformationen, die die Abstände und Winkel erhalten, und die freie Animation. Diese zwei Arten von Dynamik werden in folgenden Unterkapiteln getrennt betrachtet.

4.4.1 Euklidische Transformation

Euklidische Transformationen, wie Translation, Rotation oder Skalierung sind bekanntlich sehr leicht auf Polygonnetzen auszuführen. Es reicht passende Transformationsmatrizen auf allen Vertices anzuwenden, wobei die Reihenfolge der Ausführung zu beachten ist, weil die Rotationsmatrix das Objekt um ihren Ursprung dreht.

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Die Translationsmatrix. t_x bestimmt die Verschiebung entlang der x-Achse, t_y entlang der y-Achse und t_z entlang der z-Achse.

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Die Skalierungsmatrix. s_x , s_y und s_z sind die Skalierungsfaktoren entlang der entsprechenden Achsen.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Die Rotationsmatrix. θ bestimmt den Winkel der Rotation um die x-Achse

Die Rotationsmatrix. θ bestimmt den Winkel der Rotation um die y-Achse.

Die Rotationsmatrix. θ bestimmt den Winkel der Rotation um die z-Achse.

Da die Voxel selbst keine Information über ihre eigene Position speichern, sondern diese von der Postion der Elternknoten ableiten, ist die Translation, Rotation und Skalierung eines gesamten Voxelmodells simpel und erfolgt durch eine einfache Transformation des Wurzelobjektes beziehungsweise des lokalen Raumes. Dabei erfolgt keine Veränderung in der Datenstruktur, was das Verfahren sehr effizient macht. Diese Art der Transformation kann allerdings bewirken, dass die Voxelseiten des Objektes nicht mehr parallel zu den Achsen des globalen Raumes ausgerichtet sind, was besonders bei großen Voxeln auffällt und möglicherweise nicht erwünscht ist.

Saona et al. (1997) untersuchen Octrees und Rotationsalgorithmen, bei denen die Ausrichtung der Voxel parallel zu den Achsen bleibt und deshalb die Datenstruktur verändert werden muss. Die Autoren arbeiten mit zwei Kopien eines Modells: dem Ursprungsmodell und seiner transformierten Nachbildung. Weil Voxel eine beschränkte Auflösung haben, kann eine darauf angewandte Transformation die Ursprungsform des Modells verändern, was mit jeder weiteren darauf folgenden Transformation in zunehmender Verformung resultiert. Deshalb werden neue Transformationen mit den alten kombiniert, auf dem Ursprungsmodell angewandt und abschließend in seiner Nachbildung gespeichert. Dadurch wird eine hohe Fehlerrminimalisierung gesichert. Es gibt keine euklidische Transformation, die sich mit diesem Verfahren nicht umsetzen lassen würde, jedoch ist der Rechenaufwand hoch und das Aussehen des Modells wird verändert. Aus diesem Grund wird meistens doch zu der einfachen Drehung des Wurzelobjektes gegriffen.

4.4.2 Animation

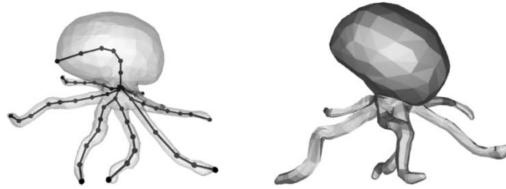
Seit circa 50 Jahren wird daran gearbeitet, die Kunst der Animation zu perfektionieren, um möglichst realistisch natürliche Bewegungen abbilden zu können. Polygone haben sich in diesem Feld dermaßen durchgesetzt, dass in der Wissenschaftswelt kaum noch nach Alternativen gesucht wird. In diesem Unterkapitel wird kurz die populäre Technik der Skelettmierung erläutert und der Stand der Voxelanimationsforschung untersucht.

Skelettmierung. Die populärste Animationstechnik, die seit circa 40 Jahren der Standard in Computerspiel- und Filmindustrie ist, wird Rigging oder einfach Skelettmierung genannt. Sie wurde das erste Mal von Magnenat-Thalmann et al. (1988) vorgestellt. Wie der Name schon verrät, wird in dieser Technik ein „Skelett“ des Modells gebaut (siehe Abbildung 28a). Dies besteht laut Magnenat-Thalmann et al. aus einer Menge an Segmenten, die durch Gelenke (englisch *joints*) verknüpft sind. Zwei Segmente bilden zwischen einander einen Winkel, der „Gelenkwinkel“ genannt wird, und die Segmente selbst können als Äquivalente zu Knochen (englisch *bones*) gesehen werden. Die Knochen bilden eine hierarchische Struktur. Die Position und Rotation jedes Kindelements ist relativ zu seinem Elternteil, mit der Ausnahme des Wurzelementes, das relativ zu den Weltkoordinaten ist. Obwohl der Rig eines Subjektes vom Aufbau her öfters einem reellen Skeletten ähnelt, ist dies keine Regel, denn diese Technik kann genauso gut an Objekten angewandt werden.

Nach der Erstellung des Skelettes beginnt ein Prozess, der *skinning* genannt wird. Jeder Vertex kann mit multiplen Gelenken verbunden werden, wobei jede solche Verbindung entsprechend gewichtet wird. Ein Vertex V , dessen Verbindung zum Gelenk j_1 mit 0.8 und zum Gelenk j_2 mit 0.2 gewichtet wurde, wird zu 80% von j_1 's und zu 20% von j_2 's Bewegung beeinflusst. Die Gewichtungen aller Verbindungen eines Vertices sollen genau 1.0 ergeben.

Die folgende abstrakte Formel dient der Berechnung der neuen Position vom Vertex V nach einer Bewegung des Skelettes.

$$V' = \sum_{j=0}^n W_j M_j V \quad (12)$$



(a) Ein Octopusmodell (rechts) und sein „Skelett“ (links). Die Gelenke werden als Punkte und die Segmente als Linien dargestellt. (Wade und Parent, 2002)



(b) Unrealistisches Verhalten der Hautoberfläche beim verbogenen Gelenk. (Mohr und Gleicher, 2003)

Abbildung 28: Rigging. Bei (a) wird ein hierarchisches Skellet eines Modells visualisiert, bei (b) wird ein Beispiel eines Problems, das durch die *smooth skinning* Methode entstanden ist, gezeigt.

Dabei sind j das jeweilige Gelenk, W_j seine Gewichtung und M_j seine Transformationsmatrix. Das n ist die Anzahl aller Gelenke, mit denen V verbunden ist.

Will man weniger abstrakt werden und die Veränderungen in Weltkoordinaten ausdrücken, muss die Formel noch leicht geändert werden. Mohr und Gleicher (2003) erläutern die neue Formel und erwähnen, dass diese das „traditionelle und interaktive Skinningmodell“ darstellt. Mohr und Gleicher nennen diese Methode *linear blend skinning*, doch in der Literatur wird sie des Öfteren anders genannt. Unter anderem taucht oft der Name *smooth skinning* auf, der ebenfalls in Autodesk Maya verwendet wird.

Generell wird zwischen zwei Stadien des Skeletts unterschieden: es kann sich in der Anfangslage (*bind* oder *rest pose*) oder in der Endlange befinden. Die erste ist eine Pose ohne jegliche Flexion, jede andere Pose entspricht dagegen der Endlage. Sollte das Skelett in der Anfangslage sein, muss die Matrix in der obigen Formel einer Einheitsmatrix gleichen. Um dies zu erreichen, wird für jedes Gelenk j eine inverse Matrix M_j^{-1} seiner Weltmatrix M_j berechnet. Eine Veränderung aus der Anfangslage a in die Endlage e kann man somit nach Mohr und Gleicher (2003) folgendermaßen beschreiben:

$$V_e = \sum_{j=0}^n W_j M_{j,e} M_{j,a}^{-1} V_a. \quad (13)$$

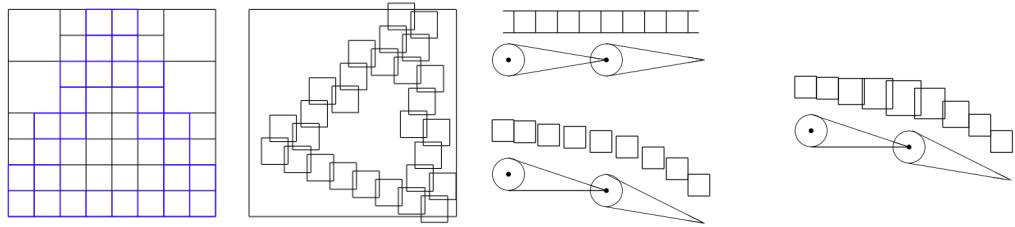
Wenn $a = e$, dann gibt es keine Veränderung, denn:

$$V' = \sum_{j=0}^n W_j I_4 V = I_4 V = V. \quad (14)$$

Smooth skinning ist eine fehlerbehaftete Methode, die in vielen Situationen unrealistische Ergebnisse liefert (siehe Abbildung 28b). Es gibt zahlreiche Optimierungen, mit denen die Ergebnisse deutlich verbessert werden können, z. B. die von Mohr und Gleicher (2003), Zayer et al. (2005) oder Botsch et al. (2006). Eine Mischung aus guter Skinning- und Motion-Capture-Technik kann die Realität nahezu fotorealistisch abbilden. Zusätzlich ist in Computerspielen inverse Kinematik, die ebenfalls ausgiebig erforscht wurde, wegen ihrer immersiver Art von ganz großer Bedeutung. Eine gute Zusammenfassung der Forschungsgeschichte der inversen Kinematik wurde von Aristidou et al. (2017) erstellt.

Autonome Voxelanimation. Eine der am öftesten genannten Schwächen von Voxeln ist ihre Animation. Es gibt keine bekannten schnellen Algorithmen, die Voxelmodelle mit einem „Skelett“ animieren würden. Eine triviale Lösung wäre, für jedes Einzelbild der Animation die modifizierte Datenstruktur zu speichern. Dabei wächst aber der Speicherverbrauch des Modells linear zur Anzahl der Einzelbilder, was bei Modellen, die selbst statisch manchmal bereits ganze Gigabytes besetzen, inakzeptabel ist. Laine und Karras (2011), die die Deformation von Voxelmodellen kurz besprochen haben, weisen jedoch darauf hin, dass der Speicherverbrauch bei dieser Methode sich mit ähnlichen Kompressionstechniken, die in Videokompression eingesetzt werden, deutlich verringern lässt. Trotz alledem beansprucht diese triviale Methode viel Speicher und setzt voraus, dass die Einzelbilder vorher aufgezeichnet sind, was den Einsatz von Techniken wie die inverse Kinematik ausschließt.

Der Hauptgrund warum Voxel schwer zu animieren sind, ist die Beschaffenheit von Octrees. In einem Voxel wird keine Information bezüglich seiner globalen Position gespeichert, nur die relative Position zum Elternteil ist bekannt. Die hierarchische Struktur bewirkt, dass ein Voxel nicht leicht verschoben werden kann. Bautembach (2011) erwähnt, dass Octrees keine „organisatorische“ Hierarchie haben, in der z. B. der „Finger“ zur „Hand“, und die „Hand“ zum „Arm“ gehört. Allgemein ist ein Octree in seiner Natur



(a) In einem Quadtree gespeicherter Dreieck (links, blau) und das gleiche Dreieck nach einer Rotation (rechts).

(b) Eine triviale Implementierung von Skinning resultiert in Lücken zwischen den Voxeln (links unten). Um dies zu verhindern, werden die Voxeln vergrößert (rechts).

Abbildung 29: Animation von Voxeln. Eine euklidische Transformation ist auf (a) zu sehen, während (b) eine nicht euklidische Transformation darstellt. (Bautembach, 2011)

nicht animierbar, denn nur sehr kleine Veränderungen im Modell können zu gewaltigen Veränderungen in der Datenstruktur führen.

Trotzdem wird immer wieder versucht, Voxelmodelle zu animieren. Bautembach (2011) erforscht z. B. wie euklidische und nicht euklidische Transformationen auf Voxel ausgeführt werden können. Er behandelt jeden Voxel wie ein Atom, das unabhängig von seinen Nachbarn transformiert werden kann. Die Transformationen werden alle erst in der Renderingphase berechnet und das Ergebnis wird gespeichert, ohne dass das eigentliche Modell verändert wird.

Das Verfahren von Bautembach am Beispiel einer Rotation sieht folgendermaßen aus: (1) traversiere den Baum bis zu den Blättern und berechne on-the-fly ihre Position, (2) wende eine Rotationsmatrix an um die neue Position des Voxels herauszufinden. An der Stelle werden die Daten des verschobenen Voxels vom Rasterizer im Z-Buffer und der Textur gespeichert. Ein Beispiel solch einer Rotation ist auf der Abbildung 29a zu sehen. Um nicht euklidische Transformationen anzuwenden, greift Bautembach auf die Skinning-Technologie zurück, wobei sein Vorgehen ähnlich wie bei der Rotation ist. Der einzige Unterschied ist, dass bei jedem Voxel der Dehnungsfaktor berechnet und gegebenenfalls darauf angewandt wird (siehe Abbildung 29b).

Animationen dieser Art haben ihre Nachteile, wie Bautembach (2011) selbst zugibt. Nur moderate Veränderungen, die nicht allzu stark vom sta-

tischen Voxelmodell abweichen, sehen gut aus. Extreme Modifikationen können z. B. dazu führen, dass manche Voxel derart vergrößert werden müssen, dass sie sich deutlich vom Modell mit ihren Kanten abheben. Außerdem bewirken extreme Modifikationen, dass die Oktanten eine komplett andere Position als ihr Elternvoxel haben können, was wiederum den Einsatz der LOD-Technik unmöglich macht. Eine weitere Folge davon ist, dass die Bildsynthese solch einer Animation nur noch suboptimal ist, weil die Schnittpunkttest-Optimierungen die darauf basieren, dass die Voxelgröße bekannt ist, nicht mehr eingesetzt werden können. Abschließend rendert Bautembach zwar einen animierten Charakter in 40 FPS, in einem Spiel gibt es jedoch deutlich weniger verfügbare Ressourcen. Die Ausführung einer Abbildungsmatrix auf jedem Voxel ist ohne Zweifel aufwändiger, als auf jedem Vertex eines Polygonnetzes.

Voxelanimation mithilfe klassischer Skelettanimation. Eine andere Methode, *animation through shell deformation*, ist laut ihrer Autoren Pavia und Crassin (2010) dazu fähig, auch große Voxelmodelle zu animieren. Die Grundidee basiert auf einer Publikation von Porumbescu et al. (2005), die das erste Mal *shell maps* benannt und beschrieben haben. Shell Maps sind entwickelt worden, um auf Flächen feine, geometrische Details anzubringen. Im Gegensatz zum Displacement Mapping, womit Eigenschaften wie Überhänge nicht leicht umsetzbar sind, können Shell Maps beliebige Strukturen wiedergeben. Ein Beispiel einer Anwendung ist auf der Abbildung 30 zu sehen. Die Autoren der Technologie erklären sie folgendermaßen: „A shell map is a bijective (one-to-one) mapping between shell space and [three-dimensional] texture space that can be used to generate fine-scale features on surfaces.“ (Porumbescu et al., 2005, p. 626)

Zuerst wird, wie die Abbildung 31a darstellt, aus einem Polygonmodell S die „Versatzfläche“ (englisch *offset surface*) S_o generiert. Die Volumen dazwischen werden Shellraum genannt. Da beide Flächen, S und S_o , die gleiche Struktur haben, kann der Shellraum wie auf der Abbildung 31b auf Prismen aufgeteilt werden. Auch der Texturraum wird ähnlich aufgeteilt, wodurch jedes Prisma aus dem Shellraum mit einem entsprechenden Prisma aus dem Texturraum abgebildet werden kann.

Das Problem dieser Aufteilung ist, dass die Prismen nicht unbedingt konvex werden (siehe Abbildung 32a), was Probleme bei der Parametrisierung

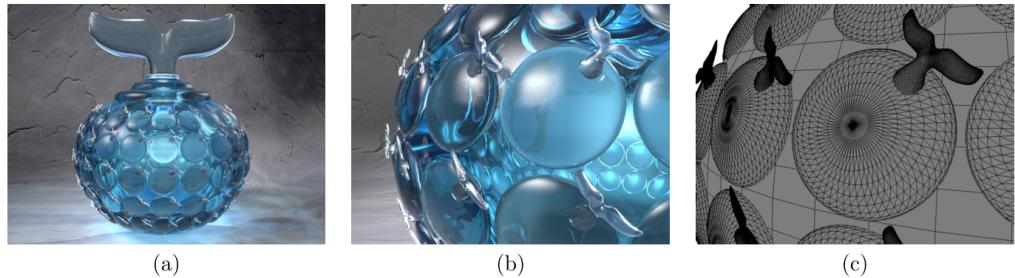
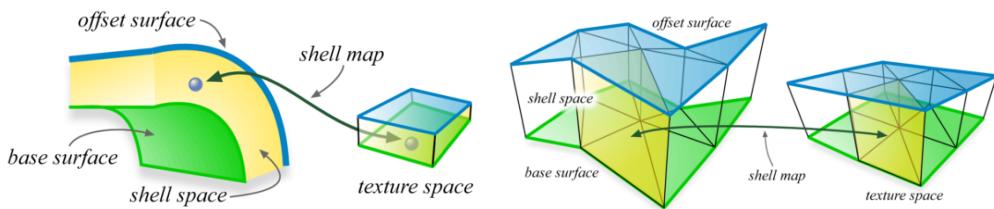


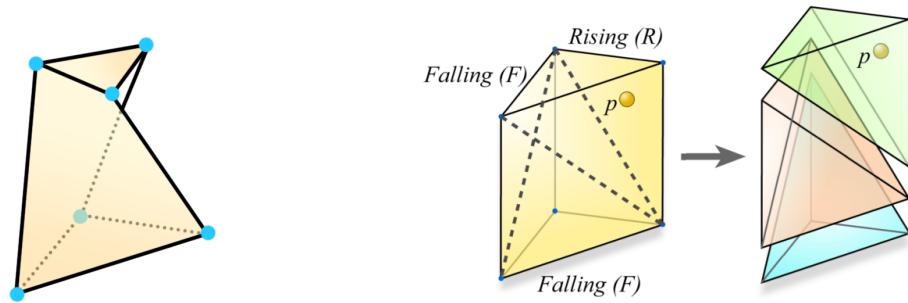
Abbildung 30: Die „Walvase“. Ein Walschwanz wird auf eine kleine Glasscheibe angebracht. Die entstandene Geometrie wird dann zur Texturierung des Gefäßes verwendet. Auf (a) ist die ganze Vase, auf (b) ein Teil davon, und auf (c) deren Polygontnetz zu sehen. (Porumbescu et al., 2005)



(a) Die Versatzfläche (blau) wird basierend auf der Modellfläche (grün) generiert. Der Raum dazwischen (gelb) ist der Shellraum. Eine Shell Map ist eine eins-zu-eins Funktion zwischen dem Shell- und dem Texturraum.

(b) Der Shellraum wird auf Prismen aufgeteilt, die alle jeweils ein entsprechendes Prisma im Texturraum besitzen.

Abbildung 31: Visualisierungen der Entstehung eines Shellraums und seiner Mappierung zum Texturraum. (Porumbescu et al., 2005)



(a) Ein Beispiel eines nicht konvexen Prismas.

(b) Aufteilung eines Prismas auf drei Tetraeder.

Abbildung 32: Shellraum-Prismen und ihre Aufteilung. (Porumbescu et al., 2005)

verursachen kann. Die Autoren gehen deshalb noch weiter: Jedes Prisma wird, wie auf der Abbildung 32b, auf drei Tetraeder aufgespalten, wobei eine bestimme Konsistenz eingehalten werden muss. Es gibt sechs mögliche Aufteilungen eines Prismas und es ist nicht irrelevant, welche gewählt wird. Problematisch sind z. B. zwei Prismen, die sich eine nicht planare Seite teilen. Werden diese Prismen unterschiedlich aufgeteilt, kann eine Lücke im Gitter oder eine Überlappung zwischen mehreren Tetraeder entstehen. Zur Lösung dieses Problems schlugen Porumbescu et al. (2005) einen Algorithmus vor, der auf der Publikation von Erleben und Dohlmann (2004) basiert. Ist die Aufteilung der Prismen beendet, erfolgt mithilfe von baryzentrischen Koordinaten eine Abbildung der Geometrie aus dem Texturraum auf den Shellraum.

Pavia und Crassin (2010) modifizieren das Verfahren, indem ihre Textur Voxel anstatt Polygone enthält. Zuerst bilden sie eine grobe, polygonisierte Abbildung ihres Voxelmodells. Der daraus generierte Shellraum kann dann auf das Voxelmodell abgebildet werden, wodurch das Modell zu einer volumetrischen Textur wird. Bewegt sich das simplifizierte Polygonmodell mithilfe von Skeletanimationstechniken, werden auf seiner Oberfläche Voxel gerendert, die aus der Textur gemappt wurden. Pavia und Crassin (2010) verwenden Raycasting und einen A-Buffer um die richtigen Texturkoordinaten zu finden und zu rendern. In jedem Einzelbild der Animation muss der Shellraum zwar neu berechnet werden, er behält aber seine Struktur. Was die Anzahl und Reihenfolge der Vertices angeht, ist eine erneute Berechnung der Tetraeder nicht nötig. Diese Eigenschaft macht Voxelani-

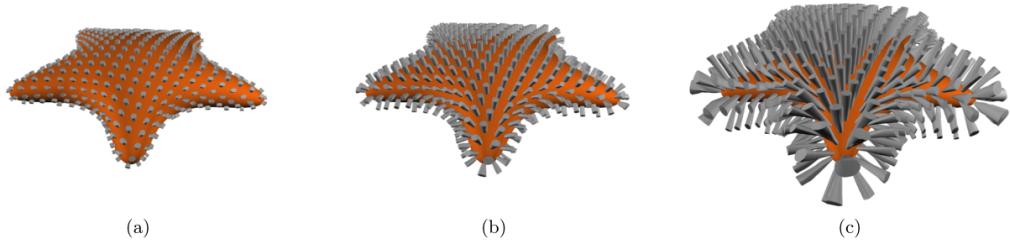


Abbildung 33: Die Verzerrung von Zylindern, die auf die Oberfläche eines Seesterns gemappt wurden. Je dicker der Shellraum, desto stärker die Texturverzerrung. (Porumbescu et al., 2005)

mation durch Shell-Deformation zu einer außergewöhnlich guten Lösung des Voxelanimationsproblems.

Leider hat der Algorithmus von Pavia und Crassin (2010) gewisse Nachteile. Die Kombination der anerkannten Polygonanimationstechniken mit dreidimensionalen Texturen funktioniert zwar gut, doch es handelt sich dabei nicht um eine alleinig auf Voxeln basierte Animation. Visuell ist zwar eine Animation entstanden, technisch gesehen wird das Voxelmodell aber nicht animiert. Aus diesem Grund lässt sich schlussfolgern, dass diese Technik keine allgemeine Lösung der Voxelanimation darstellt, auch wenn sie visuell relativ gute Ergebnisse liefert. Einen visuellen Nachteil gibt es allerdings. Je dicker der Shellraum wird, desto mehr Verzerrungen erscheinen in seinen äußeren Schichten. Dies passiert vor allem bei Wölbungen des Modells, wie auf der Abbildung 33 zu sehen ist.

4.5 Bildsynthese

4.5.1 Rasterung

In der Computergrafik wird generell zwischen zwei Arten von Bildsynthese (im englischen Sprachraum als *rendering* bekannt) unterschieden: der Rasterung, deren Ablauf kurz im Kapitel 2.2 angesprochen wurde, und der Algorithmen zur Simulation der globalen Beleuchtung. Die klassische Grafikpipeline basiert auf dem Prinzip der Rasterung, deren Ablauf lokal genannt werden kann, da jedes Dreieck unabhängig von anderen Dreiecken gerendert wird. Davon profitieren moderne Systeme stark, weil sich das Verfahren leicht parallelisieren lässt und die gleichzeitige Verarbeitung mehrerer Primitiven eine besonders schnelle Bildsynthese erlaubt. Die parallele

Natur der Rasterung spiegelt sich in dem Bau der Prozessoren wider; während CPUs meistens nur über ein paar Kerne verfügen, sind es in GPUs meistens Hunderte oder Tausende (mehr dazu im Kapitel 5).

Die Hauptprobleme der Rasterung sind darauf zurückzuführen, dass ihre Funktionsweise sehr realitätsfern ist. Die getrennte und unabhängige Bearbeitung aller Primitiven führt dazu, dass grundlegende Effekte wie Reflexion, harte Schatten oder Transparenz in der Implementierung umständlich sind. Andere Effekte, wie die indirekte Beleuchtung, weiche Schatten, Kaustik, Refraktion, Streuung und Dispersion des Lichts sind noch schwieriger in der Umsetzung. Manches lässt sich zwar gut vortäuschen, z. B. durch *baking* des Lichts auf eine Textur, doch Carmack (2011) nennt solche Methoden „Hacks“, ohne die die Rasterung keine realistischen Ergebnisse erzielen könnte.

Carmacks Aussage sollte nicht unmittelbar als Kritik an der Rasterung verstanden werden. Es ist allbekannt, dass eine simple Rasterung keine fotorealistischen Szenen abbilden kann. Mit Hacks allerdings kann eine immense Qualität der synthetisierten Bildern erreicht werden. Vor allem zahlreiche Texture-Mapping-Techniken, die mittlerweile nicht nur die Farbe einer Fläche festlegen, sondern auch zusätzlich Geometrie und Licht simulieren können, steigern den Realismusgrad in rasterisierten Szenen stark. Die Verwendung bestimmter Texture Maps hat auch oft zur Folge, dass gute Ergebnisse sich mit weniger Polygonen und Lichtberechnungen erzielen lassen. Sich alle Hacks, die bei der Rasterung verwendet werden können, anzueignen, ist durch ihre Anzahl und die Art der teilweise einzigartigen Techniken sehr schwer. Ein Beispiel dafür sind die im Kapitel 4.4.2 beschriebenen Shell Maps. Die Funktionsweise vieler Algorithmen für die globale Beleuchtung ist in dieser Hinsicht oft viel verständlicher.

Die Rasterung ist umstritten eine gute Bildsynthese-Technik, die sich seit Jahren in der klassischen Grafikpipeline beweist. Parallelisierbarkeit und Effizienz sind ihre größten Vorteile. Ihr Hauptnachteil ist die Absenz eines globalen Beleuchtungsmodells, was zu Qualitätsmängeln führt, die nur mit umständlichen Methoden vermeidbar sind. Außerdem können Voxel nicht effizient mit der Rasterung gerendert werden, was unter anderem von Laine und Karras (2011) bewiesen wurde. Ein beliebtes Vorgehen ist deshalb Voxellmodelle vor dem Rendern zu polygonisieren, z. B. mit dem Marching Cubes Algorithmus (siehe Kapitel 4.2) oder dem Dual Contou-

ring Algorithmus von Ju et al. (2002). Dual Contouring liefert nach McLaren (2015), der beide Methoden verglich, visuell bessere Ergebnisse, da er im Gegensatz zu MC keine scharfen Kanten glättet. Allerdings erfordert Dual Contouring das Vorhandensein der Volumennormalen, die nicht nur viel Speicherplatz verbrauchen sondern auch schwer zu approximieren sind. Egal welche Polygonisierungsmethode gewählt wird, das daraus entstandene Polygonnetz kann letztendlich herkömmlich rasterisiert werden.

4.5.2 BRDF

Wie das Licht beim Kontakt mit einer opaken Fläche reflektiert, wird mathematisch durch eine von Nicodemus (1965) definierte Bidirektionale Reflexionsverteilungsfunktion (BRDF) festgelegt. Diese nimmt als Parameter die Richtung des eingehenden und ausgehenden (reflektierten) Lichtstrahls. Zurück gibt die Funktion das Verhältnis der reflektierten Strahldichte zu der eingehenden Bestrahlungsstärke. Eins der ersten, der mittlerweile vielen bekannten BRDFs, ist das Phong-Beleuchtungsmodell. Phong (1975) definiert die Reflexion des Lichts als eine Zusammensetzung aus Umgebungslicht, diffusem Licht und gespiegeltem Licht. Die erste Komponente, das Umgebungslicht, ist sowohl vom Standpunkt des Betrachters als auch dem Einfallswinkel des Lichts unabhängig. Die zweite Komponente, das diffuse Licht, ist zwar auch vom Betrachtungswinkel unabhängig, denn es reflektiert in allen Richtungen, aber die Strahldichte hängt vom Einfallswinkel des Lichtstrahls ab. Die letzte Komponente, das gespiegelte Licht, wird von beiden diesen Faktoren bestimmt. Das Phong-Modell, sowie seine optimierte Abwandlung, das Blinn-Phong-Modell, gehören zu den älteren Methoden, werden aber laut Montes und Ureña (2012) auch heutzutage oft zur Simulation von Licht in der rasterungsbasierten Bildsynthese verwendet. Beide diese Modelle erfüllen jedoch weder den Energieerhaltungssatz, der besagt, dass von einem Punkt nicht mehr Strahlungsenergie ausgehen kann, als auf den Punkt fällt, noch sind sie symmetrisch. Eine symmetrische BRDF würde die Funktion $brdf(\omega_i, \omega_o) = brdf(\omega_o, \omega_i)$ erfüllen, wobei ω_i für die Richtung des eingehenden und ω_o für die Richtung des ausgehenden Lichtstrahls steht (Montes und Ureña, 2012). Diese fehlenden Eigenschaften bewirken, dass das Phong- und Blinn-Phong-Modell nicht physikalisch plausibel sind.

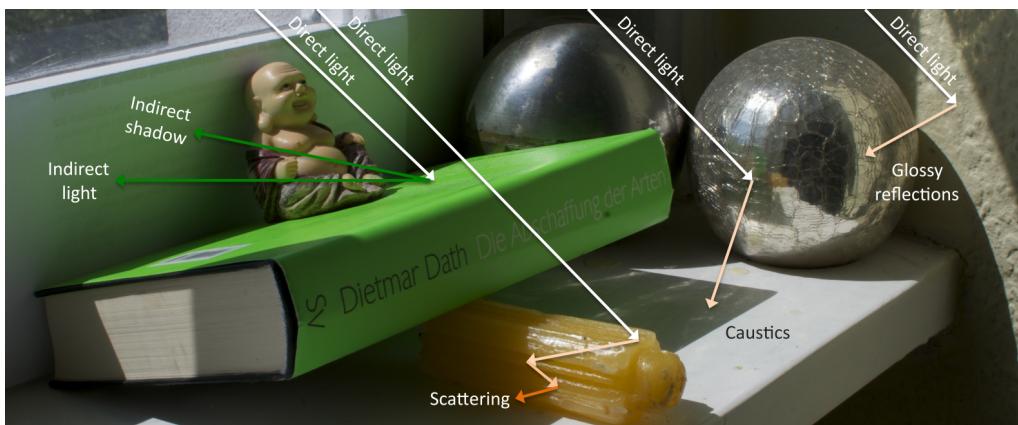


Abbildung 34: Eine Fotografie von einigen in der realen Welt vorkommenden Lichtphänomene. (Ritschel et al., 2012)

In ihrer Arbeit vergleichen Montes und Ureña (2012) über ein Dutzend verschiedener BRDFs. Phong- und Blinn-Phong-Modell klassifizieren die Autoren als empirische BRDFs, d. h. solche, die die Reflexion des Lichts imitieren, ohne die Physikgesetze dahinter zu berücksichtigen. Die Gruppe von theoretischen BRDFs schließt dagegen Funktionen ein, die die Lichtreflexion basierend auf physischen Gesetzen simulieren, wie z. B. das Torrance–Sparrow-, Cook–Torrance-, oder Ashikhmin–Shirley-Modell. Diese Modelle liefern meistens realistischere Ergebnisse als empirische BRDFs, benötigen aber eine längere Rechenzeit. Eine Analyse einiger BRDFs inklusive visuellen Vergleichs wurde von Ngan et al. (2004) durchgeführt.

4.5.3 Globale Beleuchtung

Im realen Leben kann die Reise eines Photons lang und kompliziert werden. Nachdem es eine Lichtquelle verlassen hat, kann es im Kontakt mit Materie reflektieren, brechen, streuen, oder absorbiert werden, wobei sich dabei nicht nur seine Bahn, sondern auch oft seine Wellenlänge ändert. Letztendlich fällt dieses Photon vielleicht auf eine menschliche Pupille und wird als Teil eines größeren Bildes wahrgenommen. Auf der Abbildung 34 sind die erwähnten Effekte, sowie auch die natürliche Entstehung von indirekter Beleuchtung, gut zu erkennen. Indirekte Beleuchtung wird in rasterungsbasierten Methoden nicht berücksichtigt, aber ihre Simulation ist der wichtigste Leitgedanke für die globale Beleuchtung (englisch *global illumination*) und Algorithmen zu ihrer Berechnung.

Die perfekte Simulation der Realität, in der eine Glühbirne innerhalb einer Sekunde circa 10^{20} Photonen ausstrahlen kann, ist mit existierenden Technologien unmöglich. Um den Rechenaufwand im Rahmen zu halten, wird die Anzahl an simulierten Photonen limitiert und ihr Weg meistens „rückwärts“ verfolgt, d. h. vom Auge bis zur Lichtquelle. Dadurch könnten Photonen, die das Auge gar nicht treffen und irrelevant für die Bildsynthese sind, gleich verworfen werden.

Die Ausbreitung von Lichtstrahlen, die von einer Fläche ausgesendet und reflektiert werden, wurde gleichzeitig von Immel et al. (1986) und Kajiya (1986) in einer Integralgleichung beschrieben. Diese Gleichung ist als Rendergleichung bekannt. In einer leicht vereinfachten Form sieht die Rendergleichung folgendermaßen aus:

$$L_o(x, \omega_o) = E_o(x, \omega_o) + \int_{\Omega} L_i(\omega_i) brdf(x, \omega_i, \omega_o) \cos\theta dl. \quad (15)$$

Wie hier erkennbar, ist die ausgehende Strahldichte L_o an der Stelle x eine Summe aus der emittierten Strahldichte E_o und dem reflektierten Licht. Dieses ist ein Integral über jeden von x ausgehenden Vektor in einer Hemisphäre Ω . Das Integral summiert die aus allen Richtungen einfallende Strahldichte, die zuvor mit dem Cosinus ihres Einfallswinkels und der BRDF der Oberfläche multipliziert wurde. Dementsprechend ist die Rendergleichung rekursiv, denn für die auf Punkt x einfallende Strahldichte $L_i(\omega_i)$ muss zuerst die Strahldichte $L_o(\omega_o)$, die von allen auf x reflektierenden Punkten ausgeht, berechnet werden.

Die Algorithmen, die diese Gleichung zu lösen versuchen, simulieren die globale Beleuchtung. Sie liefern im Grunde nur eine numerische Annäherung, denn rekursive Integrale in einer unendlichen Domäne wie einer Hemisphäre verkomplizieren die Gleichung in dem Ausmaß, dass sich diese analytisch nicht lösen lässt. Glücklicherweise verliert das Licht in jeder Rekursionsstufe Strahlungsenergie, wodurch es schwächer und weniger relevant für das Gesamtbild wird. Dies bedeutet, dass sich die Beschränkung der Anzahl von Rekursionsstufen visuell eventuell nicht bemerkbar macht. Dadurch reicht selbst eine Annäherung an das Ergebnis aus, um eine photorealistische Wirkung zu erzielen.

Raytracing ist eine fundamentale und verständliche Methode, die als Basis für viele Algorithmen für die globale Beleuchtung dient.

Raytracing. Das Prinzip von Raytracing ist sehr simpel zu verstehen und umzusetzen. Für jedes Pixel im Bild wird ein Strahl (englisch *ray*) aus der Kamera in die Szene geworfen. Anschließend wird nach Schnittpunkten dieses Strahles (oft auch Primärstrahl genannt) mit der Geometrie gesucht. Ist die Oberfläche mit dem Schnittpunkt x absolut spiegelnd, wird ein Reflexionsstrahl geworfen und x bekommt die Farbe des nächsten Schnittpunktes. Ansonsten wird aus dem Schnittpunkt x ein sog. Schattenstrahl in die Richtung der Lichtquelle gesendet, um die direkte Beleuchtung von x zu revidieren. Dies ist die primitivste Form von Raytracing, die zwar eine einfache Berechnung der direkten Beleuchtung und spiegelnden Reflexionen bietet, aber noch keine globale Beleuchtung simuliert. Höher entwickelte Algorithmen, wie z. B. Path Tracing, berücksichtigen zusätzlich indirekte Beleuchtung, basieren aber auf dem selben Prinzip wie Raytracing. Mittlerweile dient Raytracing deshalb als Oberbegriff für alle ähnlichen Algorithmen.

Path Tracing. Der nächste logische Schritt in der Weiterentwicklung von Raytracing ist die Berücksichtigung der indirekten Beleuchtung. Dies geschieht in Path Tracing durch Verwendung von Sekundärstrahlen, die aus dem Schnittpunkt x in allen Richtungen ausgehen und zur Evaluation des von anderen Oberflächen reflektierten Lichts verwendet werden. Wie genau das einfallende Licht aussieht, hängt jedoch auch von der indirekten Beleuchtung der aussendenden Fläche ab. Um diese herauszufinden, müssen erneut mehrere Sekundärstrahlen gesendet werden. Diese Rekursion geht theoretisch ins Unendliche, praktisch jedoch ist sie nach ein paar Tiefen nicht mehr nötig.

In der Regel werden zur Berechnung der indirekten Beleuchtung eines Punktes nur ein paar Sekundärstrahlen verwendet, um den Rechenaufwand zu reduzieren. Der Grund dafür ist, dass die Rekursion bei jedem Abprall eine abrupte Vervielfachung der Sekundärstrahlen verursacht, wie auf der Abbildung 35 zu sehen ist. Einen Integral durch Durchschnittsbildung von nur wenigen, zufällig ausgesuchten Werten zu approximieren zählt zu den Monte-Carlo-Methoden, weshalb Path Tracing und alle seine höher entwi-

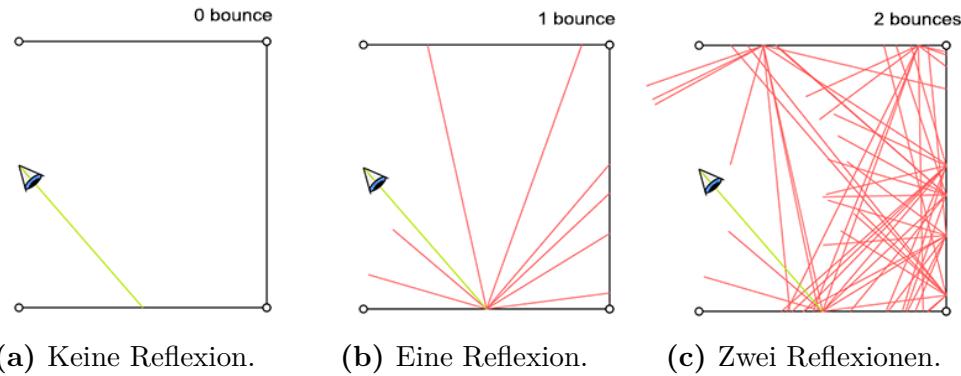
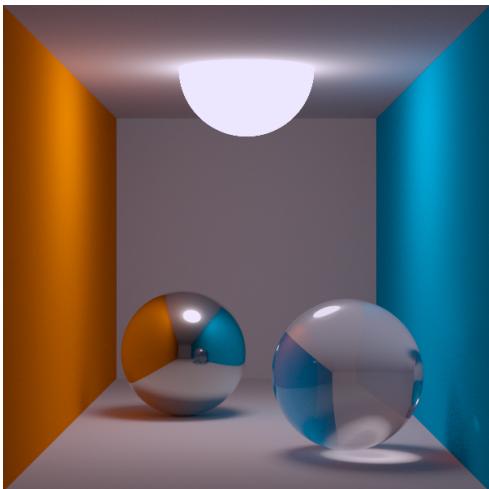


Abbildung 35: Exponentielles Wachstum der Sekundärstrahlen (rot), die keinmal, einmal und zweimal abprallen. Der Primärstrahl ist grün gekennzeichnet. (Scratchpixel 2.0: Rendering an Image of a 3D Scene)

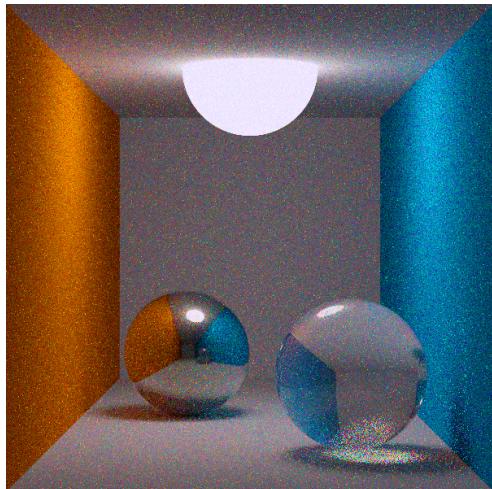
ckelten Formen zu der Familie von Monte-Carlo-Raytracing-Algorithmen gehören.

Die Approximation der wahren Beleuchtung erzeugt Rauschen im Bild, wenn nicht genügend Proben in Form von Sekundärstrahlen verwendet werden. Dieses Phänomen ist gut auf der Abbildung 36 zu sehen. Außerdem kann Path Tracing aufgrund seiner zufälligen Natur bestimmte Effekte nur schlecht oder gar nicht abbilden. Eine indirekte Beleuchtung durch spiegelnde Reflexion ist nur dann realistisch abbildbar, wenn die Sekundärstrahlen, anstatt zufällig, in die gespiegelte Richtung geworfen werden. Deutlich schwieriger ist die Abbildung von Kaustiken, denn ein Punkt x verfügt über keinen Hinweis, aus welcher Richtung er durch Kaustiken beleuchtet wird.

Echtzeitanwendungen. Die Idee von Path Tracing ist simpel und leicht umzusetzen. Doch Path Tracing hat auch Nachteile, wie die Unfähigkeit bestimmte Effekte zu erzeugen, sowie den großen Rechenaufwand, der für rauschfreie Ergebnisse benötigt wird. Existierende Verbesserungen von Path Tracing, wie z. B. bidirektionales Path Tracing und Metropolis Light Transport, adressieren teilweise diese Probleme. Darüber hinaus gibt es weitere, auf Raytracing basierende Algorithmen (z. B. Photon Mapping), die bestimmte Beleuchtungseffekte genauer, aber andere dafür schlechter abbilden. In ihrer Studie untersuchen Meneghel und Netto (2015) mehrere Algorithmen und evaluieren die mit ihnen synthetisierten Bilder mit drei verschiedenen Metriken. Dabei fanden sie heraus, dass Photon Mapping be-



(a) Viele Sekundärstrahlen.



(b) Wenige Sekundärstrahlen.

Abbildung 36: Ein mit einem Path Tracer gerendertes Bild. (Wikipedia: Path Tracing, Author: Thomas Kabir, 2007)

sonders bei spiegelnden Reflexionen effizient ist, während klassisches Path Tracing besser mit diffusen, und bidirektionales Path Tracing besser mit glänzenden Oberflächen funktioniert.

Eins haben jedoch alle Algorithmen für die globale Beleuchtung gemeinsam: sie sind langsam. Eine fotorealistische Synthese eines Bildes kann Minuten oder sogar mehrere Stunden dauern. Von besonderer Bedeutung sind deshalb Beschleunigungsstrukturen, wie das Uniform Grid, der k-d-Baum oder der Octree, die die Szene auf Bereiche aufteilen und dadurch die Anzahl der Schnitttests zwischen den Strahlen und der Geometrie deutlich verringern. Hapala et al. (2011) entwickelten in ihrer Studie einen Algorithmus, der eine geeignete Beschleunigungsstruktur für eine Szene findet. Die Wahl der Beschleunigungsstruktur ist dabei abhängig von den Eigenschaften der Szene. Außerdem zeigen Lehtinen et al. (2012), dass eine gezielte Auswahl von Sekundärstrahlen einen immensen visuellen Einfluss hat. Dieses heuristische Vorgehen, das auch Importance Sampling genannt wird, verkürzt dabei die benötigte Renderdauer. Ferner erwähnt Ritschel et al. (2012), dass sich fast jeder Algorithmus für die globale Beleuchtung mit Caching beschleunigen lässt. Die Idee von Caching ist, dass das Licht nur an bestimmten Probepunkten berechnet wird und die Lichtwerte dazwischen mit Interpolation geschätzt werden. Caching liefert aber des Öfteren realitätsferne Ergebnisse und die erwähnten Algorithmen sind ohne Caching

zu langsam, um in interaktiven Szenarien eingesetzt zu werden.

Eine beliebte Methode die globale Beleuchtung in ein Spiel zu integrieren ist, diese im Voraus zu berechnen – beispielsweise mit den oben erwähnten Methoden – und die Ergebnisse in Form einer Lightmap zu speichern. Dies verbraucht aber sehr viel Speicherplatz, da die daraus resultierenden Texturen einzigartig und nicht wiederverwendbar sind. Darüber hinaus es setzt voraus, dass die Szene statisch bleibt. Alternativ zu Lightmaps, wird zu Algorithmen gegriffen, die nur die indirekte Beleuchtung oder ihr ähnliche Effekte simulieren, während alles andere in der schnellen Rasterungs-Pipeline berechnet wird. Ein Beispiel von solchen Algorithmen sind Umgebungsverdeckungs-Algorithmen (englisch *ambient occlusion*, kurz AO), von denen am meisten die primitive, aber schnelle AO-Approximation SSAO (ss steht für *screen space*) benutzt wird (Ritschel et al., 2012). AO-Algorithmen sind zwar eine sehr primitive Annäherung an die globale Beleuchtung, reichen jedoch in vielen Fällen aus.

Ritschel et al. (2012) vergleichen in ihrer Arbeit 43 verschiedene Algorithmen für die globale Beleuchtung unter Aspekten wie Schnelligkeit, subjektive Bildqualität, Dynamik, GPU-Unterstützung, Skalierbarkeit und Komplexität in der Implementierung. Zufolge ihrer Ergebnisse gibt es mehrere Algorithmen, die nicht nur schnell genug sind, um sie in Echtzeit-Anwendungen verwenden zu können, sondern auch in dynamischen Szenarios gut funktionieren. Beispiele dafür sind der AO-Algorithmus von Kontkanan und Aila (2006), die AO-Implementierung, kombiniert mit einer Rekursion von Sekundärstrahlen, von Ritschel et al. (2009) und der auf Voxeln basierende Algorithmus für die globale Beleuchtung von Thiedemann et al. (2011).

4.5.4 Voxel-Rendering

Umständliche und oft ungenaue Polygonisierung von Voxeldaten lässt sich durch direktes Rendern von Voxeln umgehen. Die einfachste Voxel-Bildsynthese-Pipeline basiert auf der simpelsten Form von Raytracing, dem Raycasting, in dem nur Primärstrahlen geworfen werden. Die Idee des Algorithmus unterscheidet sich nicht von dem bereits beschriebenen Verfahren. Für jedes Pixel des Bildschirms wird ein Strahl aus der Kameraposition in die Szene geworfen, der für die Berechnung der Schnitttests mit der Geo-

metrie verwendet wird. Die Farbe des als erstes getroffenen Voxels wird in dem Ausgangspixel des Strahles gespeichert.

Das Raycasting in Voxel bringt einige Probleme und Herausforderungen mit sich, denn es ist relativ langsam, anfällig für Aliasingartefakte und erfordert eine Vorberechnung der Voxelnormalen, die für alle Beleuchtungsmodelle erforderlich sind. Die folgenden drei Abschnitte beschäftigen sich genau mit diesen Themen. Im letzten Abschnitt mit dem Titel *Voxel Cone Tracing* wird eine neue Beleuchtungsmethode von Crassin et al. (2011) vorgestellt.

Raycasting in den Octree. Die Anzahl an Schnitttests, die in einer Voxelwelt geprüft werden müssen, scheint zwar geringer zu sein als in einer dreiecksbasierten Szene, doch vor allem in Szenarios, in denen es viele getrennte und sich vielleicht sogar überschneidende Octrees gibt, ist gut optimiertes Raycasting besonders wichtig. Über effiziente Methoden wurde viel geforscht und geschrieben, unter anderem von Frisken und Perry (2002), die als Lösung eine neue Art von Hashfunktion für Octrees vorgeschlagen.

Das Ziel ist, Operationen wie die Lokalisierung eines Punktes (Bestimmung in welchem Octreeknoten er sich befindet) und die Nachbarsuche zwischen den Knoten möglichst zu beschleunigen. In der klassischen Ausführung dieser Operationen werden in jeder Octreestufe drei Vergleiche durchgeführt, um anschließend mithilfe von Pointern den Baum zu traversieren. Diese Methode ist ineffizient, denn erstens ist die Traversierung von Bäumen nicht vorhersehbar, was die Pipeline stark verlangsamt (Pritchard, 2001), und zweitens basiert sie oft auf langsamer Rekursion. Frisken und Perry (2002) stellen Octrees vor, die als Bäume implementiert wurden, aber Eigenschaften von linearen Octrees (siehe Kapitel 3.3.2) besitzen. Infolgedessen verfügt jeder Knoten über drei Ortungscodes, je einen pro Dimension. Wie in linearen Octrees kodiert jeder Bit im Code die zu den Geschwistern relative Position des Knotens in der jeweiligen Tiefe, was auch zur Folge hat, dass sich die Nachbarn eines Knotens gut von seinem Ortungscode ableiten lassen und nicht umständlich in einem rekursiven Verfahren gesucht werden müssen. Ein Octree wird mithilfe einer Transformation so positioniert, dass er sich im $[0,1] \times [0,1] \times [0,1]$ Bereich befindet. Folglich kann der Punkt, in dem der Strahl auf den Octree trifft, auch normiert werden. Abschließend wird jede Koordinate des Punktes separat mit

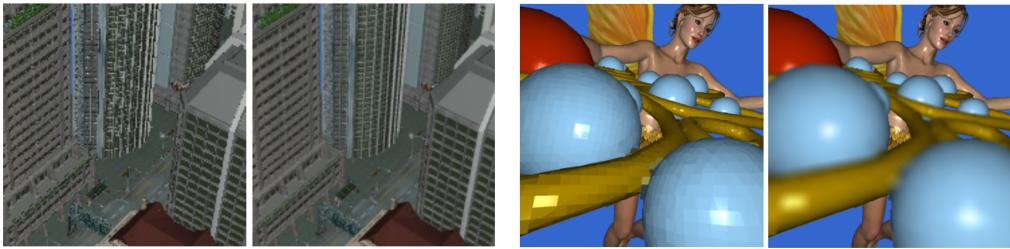
derselben Methode, die bei einzelnen Knoten eingesetzt wurde, in einen Ortungscode übersetzt. Diese Ortungscodes des Punktes werden entweder als „Wegweiser“ im Baum verwendet, um ihn ohne umständliche Vergleiche zu traversieren, oder als Indizes, falls der Octree doch in einer Tabelle gespeichert wird.

Nach der Lokalisierung des ersten Knotens kann der Strahl weiter verfolgt werden, z. B. in Schritten, die proportional zu den kleinsten Voxeln im Baum sind. Es wird solange entlang des Strahls gesucht, bis ein Voxel gefunden wird, der nicht leer ist. Dabei können sowohl weitere Punkte am Strahl lokalisiert, als auch nur Nachbarn des ersten Knotens auf eine Überschneidung mit dem Strahl geprüft werden. Frisken und Perry (2002) berichten, dass das Einsetzen des hier beschriebenen Verfahrens ihre Anwendung bedeutend beschleunigte.

Es existieren zahlreiche andere Methoden für effizientes Raycasting in einen Octree. Einen anderen Weg als Frisken und Perry haben Laine und Karras (2011) eingeschlagen, indem sie die Methode von Amanatides und Woo (1987), die sich auf Uniform Grids konzentrierten, an Octrees angepasst haben. So, wie bei Amanatides und Woo, arbeiten Laine und Karras mit Strahlen in der Parameterform. Anhand der Größe der Voxel und der Steigerung der Geraden berechnen sie den Wert t , an dem der nächstfolgende Voxel auftritt. Laine und Karras passen diese Methode an Octrees an, indem sie die unterschiedliche Größe an traversierten Voxeln berücksichtigen.

Aliasing. Wie bereits im Kapitel 4.3 erwähnt wurde, verursacht das Raycasting in Voxel Aliasing-Artefakte, die zwar nicht stark sind, aber bei der Bewegung der Kamera doch zum Vorschein kommen. Sie sind unter anderem darauf zurückzuführen, dass Octrees zwar MIP-Maps ähneln, aber keine bilineare Filterung einschließen. Befindet sich ein Pixel beispielsweise genau über zwei Voxeln, sollte seine Farbe der Durchschnittsfarbe der beiden gleichen. Bilineare Filterung zwischen den Voxeln wäre zwar möglich, doch rechenintensiv und unfähig Treppeneffekte bei Silhouetten von Objekten zu vermeiden. Viel bessere Ergebnisse liefern zwei andere bekannte Methoden: Supersampling und Post-Filterung.

Supersampling basiert auf der Idee, gleich mehrere Strahlen (*samples*) pro Pixel zu verwenden. Dadurch entsteht ein Bild, dessen Auflösung bei-



(a) Bildsynthese mit einem (links) und vier (rechts) Samples pro Pixel. **(b)** Post-Filterungs-Methode zur Glättung unterabgetasteter Bereiche.

Abbildung 37: Filterung-Techniken, die bei Voxeln anwendbar sind. (Laine und Karras, 2011)

spielsweise das Vierfache der gewünschten Auflösung ist. Abschließend wird dieses Bild verkleinert, wodurch die Farben von vier Pixeln zu einer Farbe gemittelt werden. Diese Methode mit vier Samples wurde unter anderem von Laine und Karras (2011) in ihrem Voxel-Rendering-System getestet und stellte sich, wie erwartet, als rechenaufwändig heraus. Wie die Abbildung 37a zeigt, liefert Supersampling schon ab vier Samples gute Ergebnisse.

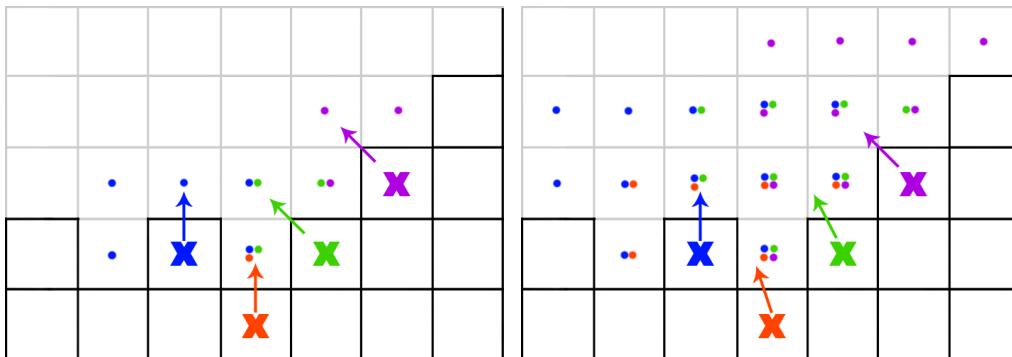
Eine andere Methode, Post-Filterung, basiert auf der Idee, bereits synthetisierte Bilder im Nachhinein, ohne Berücksichtigung jeglicher Geometrie, zu verbessern. Es gibt mehrere Algorithmen, die zu Post-Filterung zählen, wie beispielsweise FXAA (*fast approximate anti-aliasing*), doch obwohl sie Treppeneffekte glätten, dämmen sie auch Texturdetails. Eine andere Art von Post-Filterung beschreiben Laine und Karras (2011) in ihrer Arbeit. Ihr Ziel war es, „unterabgetastete“ Bereiche, d. h. solche, in denen die Voxel ihre maximale Auflösung erreicht haben und größer als Pixel sind, zu glätten. Dies erreichen sie mit einem Filter, dessen Größe dem Voxel im gerenderten Pixel entspricht. Die Ergebnisse, die auf der Abbildung 37b zu sehen sind, zeigen, dass diese Methode eine gute Alternative zur Polygonisierung des unterabgetasteten Bereiches darstellt.

Voxelnormalen. Wie bereits dargelegt, sind zur korrekten Berechnung der Beleuchtung die Oberflächennormalen nötig. Sie können mit Marching Cubes oder aus dem Z-Buffer rekonstruiert werden, jedoch sind diese beiden Methoden nicht sehr präzise. Muniz und Clua (2008) beschreiben ein genaueres Verfahren, bei dem zuerst Oberflächenvoxel gesucht werden und dann jeder Voxel getrennt betrachtet wird. Die nächstgelegene Nachbar-

schaft eines betrachteten Voxels verrät die Ausrichtung seiner Tangentialebene. Ist die Oberfläche durch eine implizite Funktion definiert, kann die Tangentialebene direkt berechnet werden, allerdings ist dies bei Voxelmodellen normalerweise nicht der Fall. Muniz und Clua schlagen deshalb vor, die implizite Funktion durch eine Distanzfunktion in der Form $f(x_p, y_p, z_p) = 1/\sqrt{(x_c - x_p)^2 + (y_c - y_p)^2 + (z_c - z_p)^2}$ zu substituieren. Dabei stehen (x_c, y_c, z_c) für die Koordinaten des betrachteten Voxels und (x_p, y_p, z_p) für die Koordinaten eines innerhalb einer festgelegten Distanz liegenden Nachbarvoxels. Wie leicht zu erkennen ist, gibt die Funktion für weiter entfernte Nachbarvoxel einen kleineren Wert zurück, als für unmittelbare Nachbarvoxel. Dies bewirkt, dass die unmittelbare Nachbarschaft einen größeren Einfluss auf die zu berechnende Tangentialebene hat. Zur Abschätzung der Tangentialebene erstellen Muniz und Clua drei partielle Ableitungen von f , jeweils nach x_p , y_p und z_p ausgerechnet. Ein Kreuzprodukt aus zwei Richtungsvektoren der Tangentialebene ergibt die Normale des betrachteten Voxels. Je größer die Nachbarschaft, die in die Berechnung einbezogen wird, desto resistenter sind die Normalen gegen Ausreißer, wodurch die berechnete Beleuchtung „weicher“ wirkt.

Alternativ zu der komplizierten Berechnung von Muniz und Clua (2008), kann die Normale eines Voxels anhand seiner leeren Nachbarn approximiert werden. Der Vorschlag ist, die relativen Positionen der leeren Nachbarvoxel zu summieren und den daraus entstandenen Vektor abschließend zu normieren. Wie bei Muniz und Clua wird auch diese Methode resistenter gegen Ausreißer, wenn eine größere Nachbarschaft miteinbezogen wird. Dies ist auf der Abbildung 38 erkennbar. Ähnlich wie bei dem Algorithmus von Muniz und Clua kann auch hier der Einfluss eines Nachbarn anhand seiner Distanz zum betrachteten Voxel gewichtet werden.

Auch, wenn nicht bewiesen, dürfte der Algorithmus von Muniz und Clua (2008) und die hier vorgeschlagene Lösung ähnliche Ergebnisse erzielen. Während beim Vorschlag von Muniz und Clua kompliziertere mathematische Formeln verwendet werden, begrenzt sich die Quantität der Berechnungen auf die relativ geringe Anzahl der Oberflächenvoxel. Die hier vorgeschlagene Formel ist simpler, muss aber dafür für alle leeren Voxel innerhalb einer Distanz ausgerechnet werden. Die Geschwindigkeit beider Methoden ist jedoch zweitrangig, denn die Normalenberechnung findet nicht in Echtzeit statt.



(a) 3x3-Filter. Maximal 8 Nachbarvoxel werden in die Berechnung der Oberflächennormalen miteinbezogen.

(b) 5x5-Filter. Maximal 24 Nachbarvoxel werden in die Berechnung der Oberflächennormalen miteinbezogen.

Abbildung 38: Approximation einer Voxelnormalen anhand der relativen Position der leeren Nachbarvoxel. Zwecks Verständlichkeit in zwei Dimensionen veranschaulicht. Die Oberflächenvoxel wurden mit „x“ markiert. Die leeren Nachbarschaftsvoxel, die zur Berechnung der Oberflächennormalen verwendet werden, wurden mit bunten Punkten in der entsprechenden Farbe gekennzeichnet. Werden alle entstandenen Normalen im Gesamtbild betrachtet, zeigt sich der Zusammenhang zwischen ihrer „Weichheit“ und der Größe der miteinbezogenen Nachbarschaft. Die teurere Methode mit dem 5x5-Filter liefert deutlich bessere Ergebnisse, als ihre schnellere Alternative mit dem 3x3-Filter.

Voxel Cone Tracing. Voxel Cone Tracing (VCT) ist der Oberbegriff für mehrere, ähnlich funktionierende Algorithmen zur Berechnung von globaler Beleuchtung. Das erste Mal wurde die bahnbrechende Beleuchtungsmethode von Crassin et al. (2011) vorgestellt. Mittlerweile entstanden mehrere Abwandlungen der Methode von Crassin et al., wie die Voxel Global Illumination (VXGI) von NVIDIA oder das Cascaded Voxel Cone Tracing in The Tomorrow Children. Die Unterschiede zwischen diesen Technologien sind nicht beträchtlich, weshalb im folgenden Abschnitt die Funktionsweise von VCT nur anhand der Arbeit von Crassin et al. untersucht wird.

Obwohl VCT kompliziert und schwer in der Umsetzung ist, ist seine Grundidee simpel. Crassin et al. voxelisieren eine dreiecksbasierte Szene und berechnen für jeden Voxel seine direkte Beleuchtung. Die Voxelstruktur wird dann zur Approximation der indirekten Beleuchtung verwendet. Der größte Unterschied von VCT zum klassischen Raytracing besteht darin, dass, anstatt Sekundärstrahlen in die Polygoneometrie zu senden, das

abprallende Licht aus den nahestehenden Voxeln mit kegelförmigen Cones abgetastet wird. Das Hauptproblem von Raytracing ist, dass indirektes Licht theoretisch aus unendlich vielen Richtungen berücksichtigt werden sollte. Da dies nicht möglich ist, werden in der Praxis nur einzelne Stichproben des einfallenden Lichts genommen, was in einer Echtzeitanwendung starkes Rauschen verursacht. Cones dagegen definieren ein größeres Stück Raum mithilfe eines Raumwinkels, was bedeutet, dass mit nur ein paar Cones die ganze Hemisphäre abgebildet werden kann. Der Nachteil dieses Verfahrens ist, dass Cones nur eine Approximation von mehreren Strahlen sind, was zu Fehlern in der Beleuchtung führen kann.

VCT ist eine Beleuchtungsmethode, die für dreiecksbasierte Grafik entwickelt wurde. Die genauen Schritte des Algorithmus, sowie die Veränderungen, die gemacht werden müssten, um ihn an voxelbasierte Szenen anzupassen, werden in den folgenden Absätzen dargelegt.

Der erste Schritt des Algorithmus ist, die vorhandene Polygonszene zu voxelisieren. Wenn die Szene dynamisch sein soll, muss der Vorgang in jedem Frame wiederholt werden. Alles in VCT, auch die Voxelisierung, wird auf der GPU ausgeführt, um möglichst effiziente Rechenzeiten zu gewährleisten. Da dieser Schritt für voxelbasierte Szenen irrelevant ist, wird es hier nicht weiter behandelt. Die beste Referenzquelle zur Voxelisierung stammt von Crassin und Green (2012).

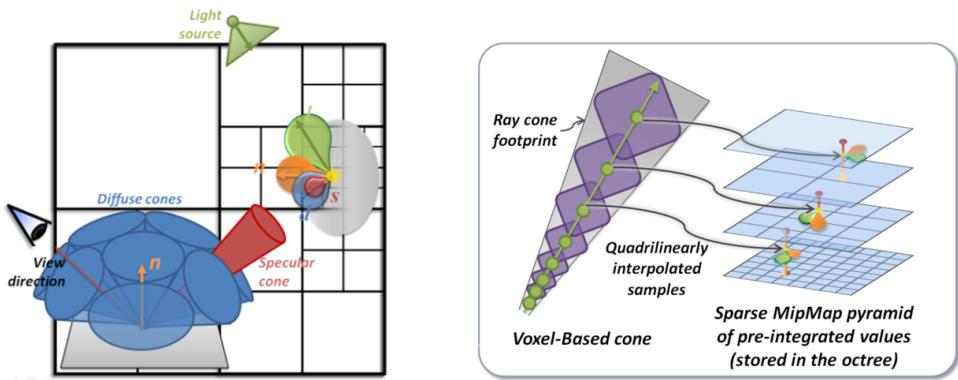
Besteht eine Voxelstruktur, wird als nächstes die gesamte Szene aus der Sicht jeder Lichtquelle rasterisiert, um jedes sichtbare Fragment zu finden. In seinem entsprechenden Voxel mit der höchsten Auflösung wird dann das einfallende Licht gespeichert. Da die klassische Grafikpipeline nicht in einer voxelbasierten Szene genutzt werden kann, muss hier ein direkter Verdeckungstest zwischen Strahlen, die aus der Lichtquelle ausgehen, und der Geometrie durchgeführt werden. Da es jedoch unendlich viele von einer Lichtquelle ausgehende Strahlen gibt, müssen diese auf irgendeine Weise reduziert werden. Hier gibt es zwei Möglichkeiten. Ähnlich wie bei Crassin et al. (2011), die sich durch Rasterung aus Sicht der Lichtquelle auf eine Auflösung begrenzen, könnte man auch bei Voxeln die Anzahl der geworfenen Strahlen festlegen. Hier muss aber beachtet werden, dass, wenn die getroffenen Voxel aus der Sicht des Lichts kleiner als ein Pixel sind, das injizierte Licht auch in all ihren Geschwistern gespeichert werden sollte. Anders ausgedrückt: das betroffene LOD muss beachtet werden. Eine

alternative Methode wäre, einen Verdeckungstest mit jedem existierenden Oberflächenvoxel auszuführen, was vor allem bei Punktlichtern praktische Vorteile hätte. Doch auch wenn sich in diesem Fall die Anzahl der gebrauchten Strahlen, beispielsweise durch eine vorläufige Auswertung des Skalarproduktes zwischen der Voxel- und Lichtnormalen, reduzieren lassen würde, bliebe sie trotzdem sehr hoch. Diese Methode wäre vermutlich zu aufwändig, um sie, wie es für eine komplett dynamische Szene nötig ist, in jedem Frame zu wiederholen. Eine dritte Alternative wäre die Erstellung eines Polygonnetzes aus den Voxeln, das als Hilfstruktur zur Berechnung der direkten Beleuchtung verwendet werden kann. Vor allem für statische Szenen mit dynamischer Beleuchtung scheint dies eine plausible Lösung zu sein.

Als nächstes wird der Octree zu einer MIP-Map, indem die Daten aus der feinsten Auflösung gemittelt und in ihrem entsprechenden Elternteil gespeichert werden. Dabei handelt es sich nicht nur um die Beleuchtungsstärke eines Voxels, sondern auch um seinen Farbwert, inklusive dem Alpha-Wert. Sind beispielsweise zwei Voxel durchsichtig und ihre übrigen sechs Geschwister opak, davon drei blau und direkt beleuchtet und drei gelb und beschattet, wird ihr Elternteil grün und seine Beleuchtungsstärke und sein Alpha-Wert würden beide 0,5 ergeben.

Das Vorhandensein einer MIP-Map ist entscheidend, denn ein weiterer großer Unterschied zwischen Cone- und Raytracing ist, dass Cones in eine MIP-Map und nicht in die feinste Geometrie verfolgt werden. Cone Tracing ist der nächste Schritt im Algorithmus von Crassin et al. (2011). Die Szene wird mit Rasterung bei polygon- und Raycasting bei voxelbasierten Szenarios aus der Sicht der Kamera gerendert. An der Weltkoordinatenstelle jedes Pixels wird eine Hemisphäre aus mehreren Cones, wie auf der Abbildung 39a zu sehen ist, erstellt. Bezuglich der Quantität der Cones gibt Crassin et al. als Richtwert die Zahl Fünf, doch andere Implementierungen, wie die in The Tomorrow Children, nutzen bis zu 16 Cones (McLaren, 2015).

Das Abtasten der Beleuchtungswerte geschieht in einem Prozess namens Ray Marching. Anstatt, wie im klassischen Raycasting, Schnitittests mit dem Octree durchzuführen, werden bei Ray Marching sequentiell immer größere „Schritte“ in die Richtung des mittleren Strahles eines Cones gemacht. Dies wird auf der Abbildung 39b visualisiert. Die Länge der Schritte hängt allein von dem Radius des Cones an der entsprechenden Stelle ab. Der Radius



- (a) Für jedes gerenderte Pixel, wird an der Stelle seiner Weltkoordinaten eine Farb- und Beleuchtungswerte aus einer gefilterten Hemisphäre aus mehreren diffusen und spiegelnden Cone erstellt.
- (b) Cones tasten die Farb- und Beleuchtungswerte aus einer gefilterten MIP-Map ab.

Abbildung 39: Cone Tracing in eine Voxelstruktur. (Crassin et al., 2011)

des Cones hat außerdem einen Einfluss auf die gewählte MIP-Map-Tiefe. Zusammengefasst lässt sich sagen, dass nach jedem Schritt eine Probe aus dem größten überlappenden Knoten entnommen wird, dessen Seitenlänge kleiner ist, als der doppelte Radius an der Probestelle. Um eine weiche Änderung des Lichts zu gewährleisten, sollte die entnommene Probe quadrilinear, d. h. zwischen zwei MIP-Map-Stufen, interpoliert werden. Die manuelle Implementierung einer solchen Operation ist mühsam, doch wenn die Voxelstruktur wie bei Crassin et al. (2011) auf 3D-Texturen basiert, kann die dafür in die Hardware eingebaute quadrilineare Interpolation genutzt werden.

Außer des Beleuchtungswertes wird auch in jedem Schritt der Verdeckungswert α kumuliert. Das Ray Marching wird bis zu einer festgelegten Distanz, oder solange $\alpha < 1$, fortgeführt. Hier zeigt sich der größte Nachteil von Cone Tracing: Da auch die Alpha-Werte von Voxeln im MIP-Map-Verfahren gefiltert werden und es keine richtigen Schnitttests gibt, kann Cone Tracing Verdeckungen nur approximieren. Als Visualisierungshilfe wird im Folgenden ein problematisches Szenario dargelegt: Für einen Punkt P auf einer Oberfläche soll die indirekte Beleuchtung, die aus einer etwas entfernten, dünnen Wand abprallt, berechnet werden. Der Cone Tracer tastet in der Nähe der Wand die MIP-Map-Werte ab. Doch aufgrund der bestehenden Entfernung zwischen P und der Wand wird eine MIP-Map-Tiefe ausgesucht, die alle Werte ihrer Kinder gemittelt enthält.



Abbildung 40: Beleuchtung mit Voxel Cone Tracing. Auf der linken Seite der Abbildung ist nur die direkte und auf der rechten die globale (direkte plus indirekte) Beleuchtung zu sehen. (José Villegas' Implementierung: josevillegas.github.io)

Im Fall einer dünnen Wand bedeutet dies, dass der abgetastete Alpha-Wert kleiner als 1 sein wird. Dadurch ist der gesamte Verdeckungswert $\alpha < 1$ und das Ray Marching wird nicht abgebrochen. Befindet sich hinter der Wand eine weitere Fläche, werden auch ihre Werte abgetastet, was zu falschen Ergebnissen, wie ungewolltes Color Bleeding, führt.

Trotz der Ungenauigkeiten was die Umgebungsverdeckung angeht, ist Cone Tracing, vor allem bei Abwesenheit sehr dünner Modelle, eine gute Approximation. Vorteilhaft dabei ist, dass sich mit dem Verfahren nicht nur indirekte Beleuchtung, sondern auch Ambient Occlusion berechnen lässt. Dafür muss der Verdeckungswert in den ersten Schritten des Ray Marchings ausgewertet werden. Cone Tracing kann auch bei der Berechnung der direkten Beleuchtung verwendet werden. Die Idee ist, aus jedem Voxel einen Cone zur Lichtquelle zu schicken und den Verdeckungswert zu kumulieren, der anschließend aussagt, inwieweit die getestete Stelle beschattet wird. Dieses Verfahren resultiert in einer direkten Beleuchtung mit weichen Schatten.

Der letzte Schritt von VCT ist, die Werte der direkten und indirekten Beleuchtung in einer Struktur zusammenzuaddieren. Um die zweite Reflexion der indirekten Beleuchtung zu berechnen wird diese Struktur erneut zu einer MIP-Map gefiltert und das Cone Tracing beginnt von vorne. Crassin et al. (2011) verwenden in ihrer Arbeit zwei Reflexionen, doch auch eine liefert bereits gute Ergebnisse. Auf der Abbildung 40 ist eine Voxelstruktur zu sehen, die mit VCT beleuchtet wurde.

Crassin et al. (2011) testeten ihren Algorithmus mit einer Sponza-Szene,

die sie mit einem Octree mit neun Tiefen in Voxel unterteilt haben. Sie verwendeten die Geforce GTX 480, eine Grafikkarte aus dem mittleren Leistungsbereich, und renderten ins 1024x768 große Viewport. Diffuses Licht wurde mit drei und spiegelndes Licht mit einem Cone berechnet. Crassin et al. erzielen mit einem bewegten Objekt und einem bewegten Licht eine durchschnittliche Bildfrequenz von 11,4 FPS.

VCT in voxelbasierter Grafik. Wie bereits erwähnt, wurde der Algorithmus von Crassin et al. (2011) für dreiecksbasierte Computergrafik entwickelt. Das Verfahren nutzt die schnelle Rasterungs-Pipeline an mehreren Stellen, wie bei der Injektion des Lichts in die Voxelstruktur oder bei der finalen Bildsynthese. Wie im vorigen Abschnitt erklärt, lassen sich die Funktionalitäten einer Rasterungs-Pipeline mit Raycasting ersetzen, doch es bleibt ungewiss, inwieweit die Performanz des gesamten Algorithmus daran leidet.

Eine weitere Problematik stellt die Voxelstruktur der Szene dar. Bei Crassin et al. (2011) sind die Voxel der feinsten Auflösung relativ grob, denn sie werden nur zur Approximation der indirekten Beleuchtung verwendet. In einem komplett auf Voxeln basierten Szenario müssten die Voxel feiner sein, damit sie dem Betrachter nicht würfelig erscheinen. Dies ist nicht so problematisch bei dem gut skalierbaren Cone Tracing, aber es verlängert die Berechnung des direkten Lichts und die finale Bildsynthese.

Um diese Probleme teilweise zu umgehen, wäre eine Kombination aus dem hybriden Renderingverfahren aus Kapitel 4.2 und Voxel Cone Tracing interessant. Anstatt, wie bei Crassin et al. (2011), eine Polygonszene zu voxelisieren, würde in einem hybriden System die unmittelbare Umgebung des Spielers polygonisiert. Dies ermöglicht die Nutzung der schnellen Rasterungs-Pipeline in dem polygonisierten Bereich und glättet kantige Voxel, während entfernte Voxel beim Raytracing weiter von ihrer MIP-Map-artigen Struktur profitieren können.

Abschließend ist es wichtig zu erwähnen, dass das Verfahren von Crassin et al. (2011) zwar bahnbrechend in der Problematik der interaktiven globalen Beleuchtung ist, es jedoch für die meisten Szenarien und Spiele noch nicht genügend Produktionsqualität besitzt. Die optimierte VCT-Version in The Tomorrow Children, auch Cascaded VCT genannt, erzielt zwischen 20-30 FPS, was für viele als zu wenig empfunden wird. Außerdem ist VCT

von Crassin et al. unter Entwicklern dafür bekannt, dass es schwer zu implementieren ist. Alle Bestandteile des Algorithmus wurden so angepasst, dass sie auf der GPU laufen können. Die Voxelstruktur bei Crassin et al. ist zum Beispiel relativ kompliziert, denn sie ist eine Kombination aus einem linearen Octree und „Bricks“ – 3D-Texturen, die jeden Voxel im Octree approximieren. Nur durch die Einbettung von 3D-Texturen in das System können Crassin et al. die Vorteile der GPU, wie quadrilineare Interpolation, nutzen. Der Implementierungsschwierigkeitsgrad macht es schwer, das Verfahren einfach so auszuprobieren, was einer der Gründe dafür sein könnte, dass nicht viele Spiele mit VCT umgesetzt worden sind. Doch die Ergebnisse von VCT sind beeindruckend und mit weiterer Forschung und immer besser werdender Hardware wird dieses Verfahren womöglich die Zukunft der globalen Beleuchtung sein.

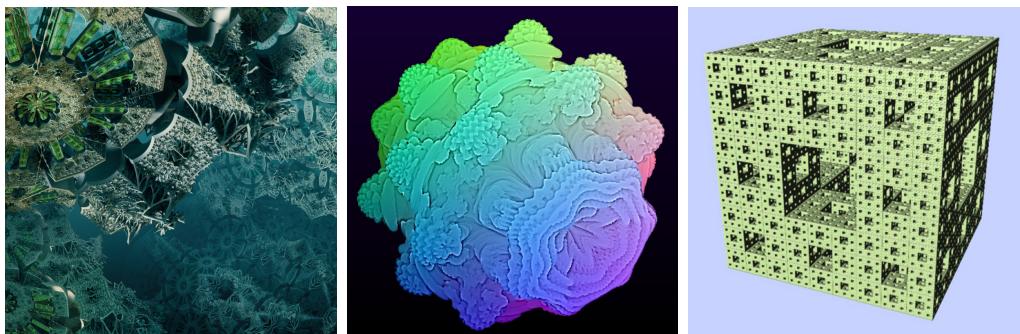
4.6 Fraktale

Gebilde, die unter jeder Auflösung selbstähnlich erscheinen, werden Fraktale genannt. Da sie mathematisch oder algorithmisch definiert werden, ist die Auflösung ihres Inhaltes komplett von dem Zoom, mit dem sie betrachtet werden, unabhängig. Weil Fraktale auch im realen Leben selten zu finden sind, ist ihre Nützlichkeit in Computerspielen gering. Doch sie können atemberaubend aussehen und werden deshalb oft in künstlerischen Visualisierungsvideos eingesetzt (siehe Abbildung 41a).

Wie Wood (2012) darlegt, sind die typischen Fraktal-Rendering-Programme nicht auf Echtzeitbildsynthese ausgelegt. Ein Bild kann manchmal stundenlang synthetisiert werden, wobei keine Speicherung von Volumendaten nötig ist. Wenn es sich laut Wood aber um eine interaktive Darstellung von Fraktalen handelt, sind Voxel als Datenstruktur die erste Wahl. Polygone wären in diesem Fall weniger geeignet, denn es ist leichter das Volumen eines Fraktals zu berechnen, als seine Isofläche zu extrahieren. Viele Fraktale haben auch partikelähnliches oder organisches Aussehen, was beides besser durch Voxel abbildbar ist.

Wood (2012) erwähnt in seiner Arbeit zwei Arten von Fraktalen. Die, deren Generierung *iterated function system* (IFS) genannt wird, gehören zur ersten Art, während *Escape-time*-Fraktale die zweite Art repräsentieren.

IFS-Fraktale bestehen aus einer Vereinigung seiner eigenen Kopien, wobei



(a) Digitale Kunst mit Fraktalen. (von und Julius Horsthuis, julius-horsthuis.com)

(b) Ein mit Raycasting (von und SSAO gerenderter Schwamm, der mit Julius Voxels-Mandelbulb. (Wood, 2012)

(c) Ein Menger-Schwamm, der mit einem spärlichen, zirkulären Graphen definiert werden kann. (Crassin, 2011)

Abbildung 41: Fraktale als alternative Form der Computergrafik.

die Kopien durch eine iterative Funktion skaliert, verschoben oder rotiert werden. Dies heißt gleichzeitig, dass sie, unabhängig vom Betrachtungsabstand, einen immer identischen Aufbau aufweisen. Das wahrscheinlich berühmteste Beispiel eines IFS-Fraktals ist das Sierpinski-Dreieck, in drei Dimensionen auch Sierpinski-Tetraeder genannt. Wood (2012) beurteilt Voxel als weniger geeignet zur Darstellung von IFS-Fraktalen, weil sie keine explizite Funktion mit sich liefern, mit der die Zugehörigkeit eines zufälligen Punktes zum Fraktalvolumen geprüft werden kann.

Die Escape-time-Fraktale, darunter Mandelbrot und seine dreidimensionale Äquivalente, Mandelbulb (siehe Abbildung 41b), haben jedoch genau diese Eigenschaft. Nach Wood (2012) sind es Fraktale, die durch eine iterative Formel oder Rekursionsgleichung in jedem Punkt einer Domäne definiert werden. Was sie zusätzlich auszeichnet, ist die *distance estimation function* (DE), die die kürzeste Distanz aus einem beliebigen Punkt zum Fraktal zurückgibt. Die DE-Funktionen unterscheiden sich je nach Fraktal, woraus sich die Schwierigkeit ergibt, dass die nötige DE-Funktion eines bestimmten Fraktals zuerst gefunden werden muss. Für die bekanntesten Fraktale ist dies aber nicht nötig, da ihre DE-Funktionen bereits beschrieben worden sind. Ein Escape-time-Fraktal ist also tatsächlich praktisch darzustellen, indem bei der Erstellung des Octrees jeder Voxel auf die Entfernung zum Fraktalvolumen mit der DE-Funktion geprüft wird. Ist im Radius des

Voxels kein Volumen, braucht dieser nicht aufgeteilt zu werden. Die Erstellung und Verfeinerung beim Zoom verläuft dementsprechend ohne jegliche Vorberechnungen, auch wenn die DE-Funktion gelegentlich falsch positive und/oder falsch negative Ergebnisse liefert.

Die Escape-time-Fraktale, die mit Voxeln visualisiert werden, haben jedoch den Nachteil einen größeren Speicherverbrauch zu haben. Außerdem wird die Octree-Traversierung mit wachsendem Zoom immer langsamer. Das Traversierungsproblem löst Wood (2012), indem er in Voxeln Referenzen zu ihren Nachbarn speichert, womit er schnelles Ray Marching ohne rekursive Traversierung des Graphen ermöglicht. Außerdem speichert er den Knoten, in dem sich die Kamera befindet, als den „Skip“-Knoten, woraus dann die Strahlen starten. Was das Speicherverbrauch-Management angeht, nutzt Wood ein ähnliches Prinzip wie das von Crassin et al. (2009), indem er Knoten, die lange nicht mehr gerendert wurden, aus dem Arbeitsspeicher wirft.

Das Problem des Speicherverbrauchs betrifft allerdings nicht unbedingt IFS-Fraktale, denn durch ihre starke Selbstähnlichkeit können sie als ein zirkulärer Graph dargestellt werden. Ein Beispiel von solch einem spärlichen IFS-Fraktal wird von Crassin (2011) präsentiert. Seinen Menger-Schwamm, der auf der Abbildung 41c zu sehen ist, implementiert er als einen N^3 -Baum, mit $N = 3$. Da das Teilungsmuster beim Menger-Schwamm immer gleich ist, besteht der Baum aus insgesamt nur 28 Knoten. Seine Auflösung ist dabei fast unendlich, denn seine Tiefe ist nur von der Fließkomma-Genauigkeit beschränkt.

5 Soft- und Hardware Unterstützung für Voxel

5.1 Software

Die zwei größten Computergrafik-APIs, OpenGL und Direct3D, haben einen begrenzten Voxelsupport. OpenGL unterstützt seit der 1.2 Version 3D-Texturen in Form von Uniform Grid. Die in Direct3D implementierten 3D-Texturen sind ähnlich, haben aber den Vorteil, dass sie als *Volume Tiled Resources* verwendet werden können. Tiled Resources ist eine Technik von Direct3D, die Gemeinsamkeiten mit MegaTextures teilt. Sie erlaubt das Allozieren von nur sichtbaren oder nötigen Voxeln, wodurch sich der

Arbeitsspeicherverbrauch deutlich verringert. Trotz des partiellen Voxel-supports in beiden APIs werden Voxel oft manuell implementiert. Eigene Implementierungen können beschleunigt werden, indem parallele Aufgaben auf die GPU ausgelagert werden. Dies ist mithilfe von Compute-Shadern, OpenCL oder CUDA möglich (siehe Kapitel 5.2 für mehr Details).

Außer dem technischen Support ist auch der künstlerische Aspekt von Bedeutung. Voxel eignen sich aufgrund ihrer natürlichen Handhabung perfekt für die 3D-Modellierung. 3D-Computergrafikprogramme stehen dabei vor der Herausforderung, spezielle Tools, mit denen intuitive „Bildhauerei“ in Voxeln möglich ist, zu entwickeln. Leider, genau wie in dem Fall der APIs, findet man auch unter Computergrafikprogrammen selten solche, die Voxel-Modellierung unterstützen. Grund dafür könnten Grafikprogramme wie ZBrush von Pixologic⁵ sein, die es geschafft haben, polygonbasierte digitale Bildhauerei zu perfektionieren. Die meisten existierenden Voxel-Editoren, wie etwa Qubicle von Minddesk⁶, sind nur zur Erstellung von Modellen aus sehr groben Voxeln geeignet. Ein Beispiel eines richtigen, voxelbasierten Bildhauerei-Programms ist 3D-Coat von Pilgway⁷.

5.2 Hardware

Heutige Grafikkarten sind nahezu vollkommen an die klassische Rasterungs-Pipeline angepasst. Sie sind effizient bei Vektor- und Matrixberechnungen, die bei geometrischen Transformationen von Polygonen oft auftreten. Außerdem werden sie gezielt so gebaut, dass sie gut mit parallelen Berechnungen umgehen können. Vertexshader, Fragmentshader und das Zusammensetzen der Vertices zu Primitiven, sind alles von Natur aus parallele Operationen. Das liegt daran, dass die Attribute eines Vertex, Fragmentes oder Primitivs, nicht von anderen Vertices, Fragmenten oder Primitiven beeinflusst werden. Deshalb unterscheiden sich GPUs und CPUs voneinander deutlich in ihrer Bauweise. Während CPUs nur wenige arithmetisch-logische Einheiten und Kerne besitzen, können GPUs Tausende davon haben. Dank dieser Architektur können GPUs in vielen Threads alle Bestandteile eines Bildes simultan berechnen. Demzufolge sind klassische Grafikkarten nicht auf die Berechnung der globalen Beleuchtung ausgelegt, denn

⁵<http://pixologic.com/>

⁶<http://www.minddesk.com/>

⁷<https://3dcoat.com/>

die pathtracingähnlichen Algorithmen haben wegen der Rendergleichung eher eine sequentielle und keine parallele Natur.

Obwohl GPUs sich eine lange Zeit nur der Rasterungs-Pipeline gewidmet haben, entwickelte sich in den letzten Jahren ein neuer Trend, in dem Grafikkarten als allgemeine parallele Prozessoren genutzt werden (Crassin, 2011). Der Trend wird oft *general-purpose computing on graphics processing units*, oder kurz GPGPU, genannt. Die immense parallele Rechenleistung der GPUs kann genutzt werden, indem Programme, die selbst paralleler Natur sind, darauf ausgelagert werden. Die Auslagerung wird durch Benutzung bestimmter APIs ermöglicht. Hier stehen mehrere zur Wahl, beispielsweise OpenCL, CUDA oder die Compute-Shader von OpenGL und DirectX.

Crassin (2011) legt dar, dass Echtzeit-Bildsynthese von Voxeln nur seit der Erscheinung der GPGPU-Technologien möglich ist. In seinem System passt er alle Bestandteile so an, dass diese auf der GPU laufen können. Ein gutes Beispiel dafür ist die Datenstruktur, in der er Voxel speichert. Crassin bevorzugt GPU-3D-Texturen vor Octrees, denn bei Letzteren lässt sich keine hardwarebeschleunigte Texturfilterung anwenden. Eine Interpolation der Texturwerte ist aber nach Crassin entscheidend für die gute Qualität des Bildes. Andererseits sind 3D-Texturen an die Struktur des Uniform Grids gebunden, wodurch sie extrem viel Speicherplatz verbrauchen. Da selbst die modernsten GPUs einen stark begrenzten Speicherplatz haben, ist das direkte Nutzen von 3D-Texturen in großer Auflösung unmöglich. Crassin fusioniert deshalb zwei Systeme, indem er zwar den Raum mithilfe eines Octrees aufteilt, doch jeder Knoten im Octree zu einer kleinen 3D-Textur (*brick*) zeigt. Die Auflösung der Bricks ist unabhängig von der Tiefe der dazugehörigen Knoten und ihre Aufgabe ist, den Inhalt ihres Knoten zu approximieren. Wie alle MIP-Maps, zeichnet sich auch dieses System mit einer gewissen Redundanz aus, doch diese gemischte Datenstruktur hat auch mehrere Vorteile. Durch die Verwendung eines Octrees verbrauchen leere Bereiche kaum Speicherplatz. Außerdem erlauben Octrees, dass nur die sichtbaren und relevanten LODs (Bricks) in den GPU-Speicher geladen werden. Die Einbettung der 3D-Texturen lässt effizientes, über Hardware beschleunigte Filterung zu und ihre Uniform-Grid-Struktur trägt zu schnelllem Raycasting bei.

Anhand dieses Beispiels wird klar, dass moderne GPUs durchaus dazu

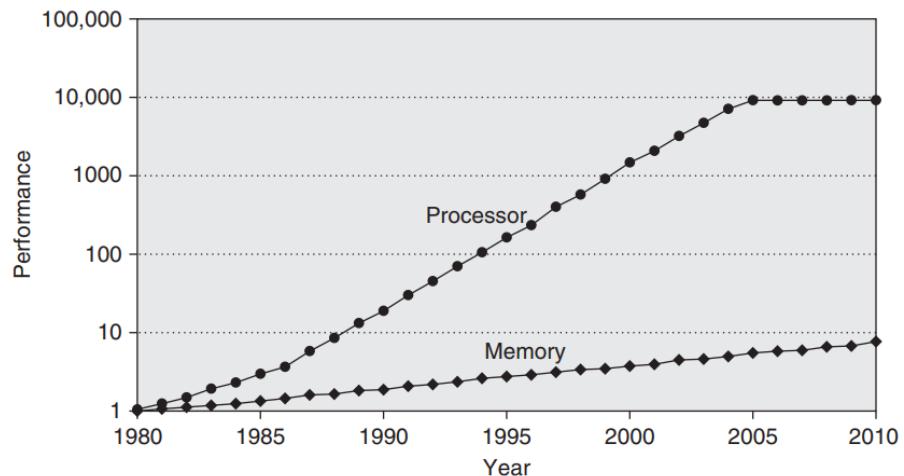


Abbildung 42: Die Kluft der Performanzsteigerung zwischen der Datenverarbeitung (pro Kern) und dem Speicherzugriff, relativ zu den Basiswerten aus dem Jahr 1980. Wegen der stark unterschiedlichen Werte musste die vertikale Achse logarithmisch dargestellt werden. Ab ungefähr 2005 startete der Mehrkernprozessorentrend, weshalb die Performanz einzelner Kerne nicht mehr anstieg. Dadurch nahm nur der Bedarf an einer hohen Datenübertragungsrate weiter zu. (Hennessy und Patterson, 2011)

geeignet sind, mit Voxeln umzugehen, doch sie bieten keine direkte Unterstützung für voxelbasierte Grafik. Vielmehr wird unter Entwicklern versucht, geschickte Tricks anzuwenden, um Voxel nutzen und rendern zu können, da die GPUs sich bei der Beschleunigung voxelbasierter Prozesse nicht beteiligen, sowie sie es im Falle von dreiecksbasierter Grafik tun. Theoretisch könnte angenommen werden, dass, angesichts der ständig steigenden Performanz von Grafikkarten und der Existenz von GPGPU, es in der Zukunft möglich sein wird, interaktive Voxel in voller Auflösung und dynamischer Beleuchtung auf klassischen GPUs zu rendern. Doch dies ist fraglich. Wie Hennessy und Patterson (2011) bei der Betrachtung der Prozessorentwicklung festgestellt haben, wird die Kluft zwischen der Geschwindigkeit der Datenverarbeitung und der Geschwindigkeit des Speicherzugriffs immer größer. Dieser Trend ist deutlich auf der Abbildung 42 zu sehen. Wie auf dem Graphen erkennbar, steigt die Geschwindigkeit des Speicherzugriffs nur langsam, doch diese ist bei Voxeln von großer Bedeutung. Crassin (2011) umgeht das Problem durch ein selbst erstelltes Caching-System, das die Wiederverwendung bereits geladener Daten ermöglicht und somit die Anzahl an Speicherzugriffen reduziert.

Um die maximale Performanz beim Umgang mit Voxeln zu erzielen, müssten diese mit speziell dafür entwickelter Hardware berechnet werden. Carmack (2010) erzählte in einem Interview mit Bob Colayco, wie er Voxel-systeme analysierte, um herauszufinden, wie eine für sie angepasste Hardware aufgebaut werden müsste. Seine Expertise ist, dass ein Voxel-Ray-Tracer in der GPU deutlich spärlicher sei, als die aktuelle Ausstattung, die ein Rasterizer benötigt. Dennoch gab Carmack zu, dass die Entwicklung einer neuen Hardware kostspielig sei und deshalb nicht wirklich attraktiv, denn mit Innovationen sind viele Risiken verbunden. Der Einsatz von Voxeln zur Berechnung der globalen Beleuchtung, wie beispielsweise bei VXGI von NVIDIA, kann jedoch als ein Schritt in die Richtung der voxelbasierten Modelle gesehen werden und weckt die Hoffnung, dass die Industrie doch Interesse an alternativen Computergrafikmethoden hat.

6 Fazit

Das Ziel der vorliegenden Arbeit war es, Voxel und Polygone im Hinblick auf ihre Eignung als grafische Primitive zu untersuchen. Polygone, oder genauer gesagt Dreiecke, sind in den letzten Jahrzehnten zu dem beliebtesten Primitiv geworden, doch ihre Verwendung bringt auch negative Konsequenzen mit sich. Voxel können als ein Primitiv verwendet werden, das potenziell die Probleme der klassischen Computergrafik löst.

Im Laufe der Arbeit wurden die verschiedenen Aspekte und Einsatzgebiete der beiden Primitiven getrennt betrachtet. Zusammenfassend lässt sich sagen, dass Voxel einige Stärken aufweisen, die sie von Dreiecksgittern unterscheiden. Sie lassen sich leicht modifizieren, vereinfachen und in den Arbeitsspeicher streamen. Aufgrund ihrer soliden Eigenschaften eignen sich Voxel gut zur Ausführung bestimmter Operationen, wie die Simulation von Erosion oder prozedurale Generierung des Terrains. Auch erwiesen sich Voxel als eine gute Datenstruktur zur Echtzeitberechnung von globaler Beleuchtung. Auf der anderen Seite hingegen ist die klassische dreiecksbasierte Computergrafik speicherplatzsparender. Es zeigte sich, dass die Simulation von physischen Kräften, wie es im Falle von Destructibles nötig ist, eher mit Polygon- als mit Voxelmodellen in Echtzeit umsetzbar ist. Außerdem lassen sich Dreiecke leicht und schnell animieren, was bei Voxeln besonders

problematisch ist. Der größte Vorteil von Polygonen ist jedoch, dass sie in der Rasterungs-Pipeline schnell und effizient gerendert werden können.

In Bezug auf die untersuchten Einsatzgebiete von Voxeln und Polygonen lässt sich die Schlussfolgerungen ziehen, dass nicht pauschal beantwortet werden kann, welches Primitiv nun besser ist. Voxel können zum Teil nützlicher sein als Dreiecke, doch nur in bestimmten Anwendungsszenarien. Für große Spiele mit vielen riesigen Szenen könnte der Speicherverbrauch von Voxeln ein beträchtliches Problem darstellen. Sind die Szenen dagegen groß, aber ihre Anzahl klein, betrifft das Speicherverbrauchsproblem hauptsächlich den Arbeitsspeicher und kann mit einem Streamingsystem gelöst werden. Darüber hinaus hängen die Vorteile beider Primitiven stark von den Strukturen ab, die sie abbilden sollen. Formen, die sich mit nur ein paar Punkten abbilden lassen, sind effizienter mit Polygonen darzustellen, als mit Voxeln. Die Stärke der Voxel liegt hingegen bei der Abbildung natürlicher und organischer Strukturen. Die genaue Wiedergabe solcher unregelmäßigen und detaillierten Formen erfordert viele Primitive, die, wenn aus der Entfernung betrachtet, nur vereinfacht dargestellt werden sollten. In diesem Fall ist die Tatsache, dass Voxel über ein natürliches und gut funktionierendes LOD-System verfügen, ein großer Vorteil. Des Weiteren bieten Voxel zwar viele Vorteile, doch bringen sie auch neue Herausforderungen mit sich. Ein gutes Beispiel dafür ist die Implementierung eines modifizierbaren Terrains. Dank einer schnellen Datenstruktur, wie dem Octree, ist dies mit Voxeln technisch leicht umsetzbar, doch es bringt auch bedenkliche Folgen mit sich. Abgesehen von dem größeren Speicherverbrauch, den solide Gegenstände aufweisen, muss dazu auch ihr Inneres texturiert werden, was die Arbeitsbelastung der Künstler deutlich erhöht. Außerdem muss das Spiel designtechnisch darauf vorbereitet werden, dass der Spieler so viel Gestaltungsfreiraum besitzt. Wird dies nicht gemacht, könnte es dazu kommen, dass eine Veränderung der Umgebung dem Spieler neue, aber vom Designer nicht vorgesehene Wege eröffnet.

Anhand dieses Beispiels wird sichtbar, dass die Wahl eines geeigneten Primitivs nur dann möglich ist, wenn das Konzept der Anwendung feststeht. Es sollte außerdem nicht unerwähnt bleiben, dass Voxel und Polygone nicht unbedingt als Konkurrenten betrachtet werden müssen. Da ihre Stärken in getrennten Bereichen liegen, kann ihr gezielter Einsatz zu einer gegenseitigen Ergänzung führen. Ein Beispiel dafür wäre ein hybrides System, in

dem naheliegende Modelle polygonisiert und entfernte Modelle als Voxel dargestellt werden. Die maximale Auflösung der Voxelstrukturen kann in solch einem System niedriger sein, ohne dass dies zu einer Verschlechterung der Bildqualität führt. Die vorhandene Voxelstruktur ist nützlich, weil sie sich um das LOD kümmert und für effiziente Modifikationen oder die Berechnung der globalen Beleuchtung genutzt werden kann.

Der Grund, weswegen Dreiecke beliebter als Voxel sind, liegt nicht nur direkt an den Vor- und Nachteilen der beiden Primitiven. Es sollte nicht vergessen werden, dass sich Dreiecke in einer Zeit durchgesetzt haben, in der Grafikkarten noch deutlich weniger Leistung hatten. Dreiecke waren damals der effizienteste Weg, 3D-Modelle abzubilden. Da sie zum Standard geworden sind, dreht sich die Forschung seit Jahrzehnten hauptsächlich um sie. Doch mittlerweile ist die Technik so weit fortgeschritten, dass auch kompliziertere Primitive schnell berechnet werden können. Voxel haben den Vorteil, dass sie gut mit Raytracing funktionieren. Da Rasterung viele Tricks erfordert, werden Raytracing und ähnliche Strahlenverfolgungsmodelle von vielen als der nächste logische Schritt in der Computergrafik gesehen. Doch ob diese Vision neuer Standards in der Computergrafik verwirklicht werden kann, hängt stark von den Grafikkartenherstellern ab. Nur, wenn Voxel auf einer dafür ausgelegten Hardware gerendert werden, können sie ihr komplettes Potenzial entfalten.

Abbildungsverzeichnis

1	OpenGL-Pipeline. (OpenGL Wiki)	8
2	Uniform Grid und Octree visualisiert. (Gebhardt et al., 2009)	11
3	Alternative Implementierungen von pointer-based Octrees. (Geier, 2014)	13
4	Lineares Octree, ein Beispiel einer pointerless Implementie- rung (Geier, 2014)	14
5	Speicherverbrauch von Voxeln und Polygonen. (McNeely et al., 1999) und (Laine und Karras, 2011)	16
6	Überschneidung zweier Dreiecke. (Qu et al., 2008)	21
7	Delaunay-Triangulierung und Umkreisbedingung. (Tsai, 1993)	22
8	Visualisierungsbeispiele zu Booleschen Operationen an Vo- xeln und Polygonen. (Perng et al., 2001)(Qu et al., 2008) . .	23
9	Beispiel einer thermischen Erosion. (Beneš und Forsbach, 2001)	24
10	Beispiel einer hydraulischen Erosion mit Regen und einer Wasserquelle. (Mei et al., 2007)	24
11	Beispiele von Destructibles anhand von zwei verschiedenen Mustern.(Müller et al., 2013)	26
12	Ein Stanford Bunny, zerlegt in mehrere Bestandteile mithilfe eines Voronoi-Diagramms. (Domaradzki und Martyn, 2016) .	29
13	Eine mit der Methode von Geiss prozedural generierte Land- schaft. (Geiss, 2007)	30
14	Visualisierung der Grundlagen von LOD. (Luebke, 2001) . .	31
15	<i>Edge collapse</i> Transformation von Hoppe. (Hoppe, 1998) . .	35
16	Vertexhierarchie und das „Falten“ und „Entfalten“ eines Ver- tex. (Luebke et al., 2003)	36
17	Ein hybrides LOD-System mit einem SVO, das partiell po- lygonisiert wird. (Mulgrew, 2014)	38
18	Die Äquivalenzklassen des MC-Algorithmus. (Lengyel, 2010)	39
19	Transitionszelle von Lengyel. (Lengyel, 2010)	40
20	O-Mapping von Bier und Sloan. (Bier und Sloan, 1986) . .	43
21	Two-Part-Texture-Mapping-Technik von Bier und Sloan. (Wol- fe, 1997)(Bier und Sloan, 1986)	44
22	Projektion von Texturen. (Lansdale, 1991)	45

23	Lineare Interpolation der Werte im Bildschirmraum versus im 3D-Raum. (Low, 2002)	46
24	Color-Bleeding bei Voxeln. (Benson und Davis, 2002)	48
25	Visualisierung der Nachbarschaften eines Voxels im synthetisierten Volumen und das Exemplar. (Kopf et al., 2007) . .	49
26	Synthetisierung von dreidimensionalen Texturen. (Kopf et al., 2007)	50
27	Szenario, in dem das Speichern einer Farbe pro Voxel ungünstig ist. UV-Mapping von Voxeln. (Cepero, 2016)	51
28	Rigging und damit verbundene Probleme. (Wade und Parent, 2002)(Mohr und Gleicher, 2003)	55
29	Animation von Voxeln. (Bautembach, 2011)	57
30	Ein Objekt texturiert mit Shell-Maps. (Porumbescu et al., 2005)	59
31	Visualisierungen der Entstehung eines Shellraums und seiner Mappierung zum Texturraum. (Porumbescu et al., 2005) . .	59
32	Shellraum-Prismen und ihre Aufteilung. (Porumbescu et al., 2005)	60
33	Die Verzerrung von Shell-Maps-Texturen. (Porumbescu et al., 2005)	61
34	Eine Fotografie von einigen in der realen Welt vorkommenden Lichtphänomene. (Ritschel et al., 2012)	64
35	Exponentielles Wachstum der Sekundärstrahlen. (Scratchpixel 2.0)	67
36	Ein mit einem Path Tracer gerendertes Bild. (Wikipedia) . .	68
37	Filterung-Techniken, die bei Voxeln anwendbar sind. (Laine und Karras, 2011)	72
38	Approximation einer Voxelnormalen anhand der relativen Position der leeren Nachbarvoxel	74
39	Cone Tracing in eine Voxelstruktur. (Crassin et al., 2011) . .	77
40	Globale Beleuchtung mit Voxel Cone Tracing. (José Villegas)	78
41	Fraktale als alternative Form der Computergrafik. (Julius Horsthuis)(Wood, 2012)(Crassin, 2011)	81
42	Die Kluft der Performanzsteigerung zwischen der Datenverarbeitung (pro Kern) und dem Speicherzugriff. (Hennessy und Patterson, 2011)	85

Literaturverzeichnis

- J. Amanatides und A. Woo. A Fast Voxel Traversal Algorithm for Ray Tracing. *Eurographics*, 87(3):3–10, 1987.
- A. Aristidou, J. Lasenby, Y. Chrysanthou, und A. Shamir. Inverse Kinematics Techniques in Computer Graphics: A Survey. *Computer Graphics Forum*, 00(00):1–24, 2017.
- D. Bautembach. *Animated Sparse Voxel Octrees*. Masterarbeit, University Of Hamburg, 2011. URL <http://masters.donntu.org/2012/fknt/radchenko/library/asvo.pdf>.
- B. Beneš und R. Forsbach. Layered data representation for visual simulation of terrain erosion. *Proceedings of SIGGRAPH*, Seiten 80–86, 2001.
- D. Benson und J. Davis. Octree textures. *ACM Trans. Graph.*, 21(3):785–790, 2002.
- E. A. Bier und K. R. Sloan. Two-Part Texture Mappings. *IEEE Computer Graphics and Applications*, 6(9):40–53, 1986.
- M. Botsch, M. Pauly, M. Gross, und L. Kobbelt. PriMo: coupled prisms for intuitive surface modeling. *Eurographics Symposium on Geometry Processing*, Seiten 11–20, 2006.
- W. Carlson. A Critical History of Computer Graphics and Animation. Technical report, Ohio State University, 2003. URL <https://web.archive.org/web/20070405172134/http://accad.osu.edu/~wayneC/history/lessons.html>.
- J. Carmack. John Carmack Interview for Firing Squad / Interviewer: Bob Colayco, 2010. URL https://archive.org/stream/johnc-interviews-johnc-interviews_djvu.txt.
- J. Carmack. GPU Race, Intel Graphics, Ray Tracing, Voxels and more / Interviewer: Ryan Shrout, 2011. URL <https://web.archive.org/web/20180402201756/https://www.pcper.com/reviews/Editorial/John-Carmack-Interview-GPU-Race-Intel-Graphics-Ray-Tracing-Voxels-and-more/Transcr>.
- E. E. Catmull. *A Subdivision Algorithm for Computer Display of Curved*

- Surfaces*. Dissertation, The University of Utah, 1974. URL http://www.pixartouchbook.com/storage/catmull_thesis.pdf.
- M. Cepero. *Applying textures to voxels*. Blog, 2016. URL <https://web.archive.org/web/20180301185805/http://procworld.blogspot.de/2016/05/applying-textures-to-voxels.html>.
- G. Çit, K. Ayar, S. Serttaş, und C. Öz. A Real-Time Virtual Sculpting Application With a Haptic Device. *TWMS Journal of Applied and Engineering Mathematics*, 3(2):223–230, 2013.
- J. H. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, 1976.
- C. Crassin. *GigaVoxels: A Voxel-Based Rendering Pipeline For Efficient Exploration Of Large And Detailed Scenes*. Dissertation, Université Grenoble Alpes, 2011. URL http://artis.imag.fr/Publications/2011/Cra11/CCrassinThesis_EN_Web.pdf.
- C. Crassin und S. Green. Octree-Based Sparse Voxelization Using The GPU Hardware Rasterizer. In P. Cozzi und C. Riccio, editors, *OpenGL Insights*, Seiten 304–319. CRC Press, 2012.
- C. Crassin, F. Neyret, S. Lefebvre, und E. Eisemann. GigaVoxels: Ray-Guided Streaming for Efficient and Detailed Voxel Rendering. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, Seiten 15–22, 2009.
- C. Crassin, F. Neyret, M. Sainz, S. Green, und E. Eisemann. Interactive indirect illumination using voxel cone tracing. *Computer Graphics Forum*, 30(7):1921–1930, 2011.
- D. DeBry, J. Gibbs, D. D. Petty, und N. Robins. Painting and rendering textures on unparameterized models. *SIGGRAPH*, Seiten 763–768, 2002.
- J. Domaradzki und T. Martyn. Fracturing Sparse-Voxel-Octree objects using dynamical Voronoi patterns. In *Computer Graphics, Visualization and Computer Vision WSCG*, Seiten 37–46, 2016.
- K. Engel, M. Hadwiger, J. M. Kniss, A. E. Lefohn, C. R. Salama, und D. Weiskopf. *Real-time volume graphics*. Taylor & Francis Ltd, 2006. ISBN 1568812663.

- K. Erleben und H. Dohlmann. The Thin Shell Tetrahedral Mesh. *Proceedings of DSAGM*, Seiten 94–102, 2004.
- S. F. Frisken und R. N. Perry. Simple and Efficient Traversal Methods for Quadtrees and Octrees. *Journal of Graphics Tools*, 7(3):1–11, 2002.
- I. Gargantini. An Effective Way to Represent Quadtrees. *Communications of the ACM*, 25(12):905–910, 1982.
- S. Gebhardt, E. Payzer, L. Salemann, A. Fettinger, E. Rotenberg, und C. Seher. Polygons, Point-Clouds and Voxels, a Comparison of High-Fidelity Terrain Representations. *Fall Simulation Interoperability Workshop*, Seiten 1–9, 2009.
- D. Geier. *Advanced Octrees 2: Node Representations*. Blog, 2014. URL <https://geidav.wordpress.com/2014/08/18/advanced-octrees-2-node-representations/>.
- R. Geiss. Generating Complex Procedural Terrains Using the GPU. In *GPU Gems 3*, Kapitel 1. Addison-Wesley, 2007.
- P. Goswami. *Level-of-Detail and Parallel Solutions in Computer Graphics*. Dissertation, Universität Zürich, 2012. URL <http://www.zora.uzh.ch/id/eprint/62018/1/Goswami.pdf>.
- M. Hapala, O. Karlik, und V. Havran. When It Makes Sense to Use Uniform Grids for Ray Tracing. *Proceedings of WSCG*, Seiten 193–200, 2011.
- V. Havran, J. Prikryl, und W. Purgathofer. Statistical Comparison of Ray-Shooting Efficiency Schemes. Technical report, Vienna University of Technology, 2000. URL https://www.researchgate.net/publication/2638687_Statistical_Comparison_of_Ray-Shooting_Efficiency_Schemes.
- P. S. Heckbert. Fundamentals of Texture Mapping and Image Warping, 1989. ISSN 10974199. URL <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.47.3964&rep=rep1&type=pdf>.
- J. L. Hennessy und D. A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann, 2011. ISBN 9780123838728.

- H. Hoppe. View-dependent refinement of progressive meshes. *Proceedings of SIGGRAPH*, Seiten 189–198, 1997.
- H. Hoppe. Efficient implementation of progressive meshes. *Computers and Graphics (Pergamon)*, 22(1):27–36, 1998.
- M. M. Hossain, T. M. Tucker, T. R. Kurfess, und R. W. Vuduc. A GPU-parallel construction of volumetric tree. *Proceedings of the 5th Workshop on Irregular Applications Architectures and Algorithms*, Seiten 1–4, 2015.
- D. S. Immel, M. F. Cohen, und D. P. Greenberg. A radiosity method for non-diffuse environments. *Computer Graphics*, 20(4):133–142, 1986.
- S. Jabłoński und T. Martyn. Real-Time Rendering of Continuous Levels of Detail for Sparse Voxel Octrees. In *Computer Graphics, Visualization and Computer Vision. Short Papers Proceedings*, Seiten 79–88. 2016.
- T. Ju, F. Losasso, S. Schaefer, und J. Warren. Dual contouring of hermite data. *ACM Trans. Graph.*, 21(3):339–346, 2002.
- J. T. Kajiya. The rendering equation. *Computer Graphics*, 20(4):143–150, 1986.
- D. Kasik und C. J. Senesac. Visualization: Past, Present, and Future at The Boeing Company, 2014. URL http://www.elysuminc.com/gpdis/2014/DX28_Boeing-Kasik-Senesac-Visualization-DX-Open.pdf.
- J. Kontkanen und T. Aila. Ambient Occlusion for Animated Characters. In *Technology*, Seiten 343–348, 2006.
- J. Kopf, C.-W. Fu, D. Cohen-Or, O. Deussen, D. Lischinski, und T.-T. Wong. Solid texture synthesis from 2D exemplars. *ACM Trans. Graph.*, 26(3):2, 2007.
- S. Laine und T. Karras. Efficient sparse voxel octrees. *IEEE Transactions on Visualization and Computer Graphics*, 17(8):1048–1059, 2011.
- R. Lansdale. *Texture Mapping and Resampling For Computer Graphics*. Masterarbeit, University of Toronto, 1991. URL <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.57.4266&rep=rep1&type=pdf>.
- C. L. Lawson. Transforming triangulations. *Discrete Mathematics*, 3(4):365–372, 1972.

- J. Lehtinen, T. Aila, S. Laine, und F. Durand. Reconstructing the indirect light field for global illumination. *ACM Trans. Graph.*, 31(4):1–10, 2012.
- E. Lengyel. *Voxel-based terrain for real-time virtual simulations*. Dissertation, University of California, Davis, 2010. URL <http://transvoxel.org/Lengyel-VoxelTerrain.pdf>.
- W. E. Lorensen und H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *Computer Graphics*, 21(4):163–169, 1987.
- K.-L. Low. Perspective-Correct Interpolation. Technical report, University of North Carolina at Chapel Hill, 2002. URL <http://citeseer.ist.psu.edu/viewdoc/download?doi=10.1.1.3.211&rep=rep1&type=pdf>.
- D. Luebke, M. Reddy, J. D. Cohen, A. Varschney, B. Watson, und R. Hubner. *Level of Detail for 3D Graphics*. Morgan Kaufmann, 2003. ISBN 9780080510118.
- D. P. Luebke. A developer’s survey of polygonal simplification algorithms. *IEEE Computer Graphics and Applications*, 21(3):24–35, 2001.
- N. Magnenat-Thalmann, R. Laperrire, und D. Thalmann. Joint-dependent local deformations for hand animation and object grasping. *Proceedings on Graphics Interface*, Seiten 26–33, 1988.
- J. McLaren. The Technology of The Tomorrow Children. In *Game Developers Conference*, 2015. URL <https://www.gdcvault.com/play/1022428/The-Technology-of-The-Tomorrow>.
- W. A. McNeely, K. D. Puterbaugh, und J. J. Troy. Six degree-of-freedom haptic rendering using voxel sampling. *Proceedings of SIGGRAPH*, Seiten 401–408, 1999.
- X. Mei, P. Decaudin, und B. G. Hu. Fast hydraulic erosion simulation and visualization on GPU. In *Proceedings - Pacific Conference on Computer Graphics and Applications*, Seiten 47–56, 2007.
- G. B. Meneghel und M. L. Netto. A Comparison of Global Illumination Methods Using Perceptual Quality Metrics. In *SIBGRAPI Conference on Graphics, Patterns and Images*, Seiten 33–40, 2015.
- M. Müller, N. Chentanez, und T.-Y. Kim. Real Time Dynamic Fractu-

- re with Volumetric Approximate Convex Decompositions, 2013. URL <https://www.youtube.com/watch?v=eB2iBY-HjYU>.
- T. Möller. A Fast Triangle-Triangle Intersection Test. *Journal of Graphics Tools*, 2(2):25–30, 1997.
- A. Mohr und M. Gleicher. Building efficient, accurate character skins from examples. *ACM Trans. Graph.*, 22(3):562, 2003.
- R. Montes und C. Ureña. An Overview of BRDF Models. Technical report, University of Granada, 2012. URL <http://citeseerk.ist.psu.edu/viewdoc/download?doi=10.1.1.923.6560&rep=rep1&type=pdf>.
- T. Mulgrew. Sparse Voxel Octree and Polygon Hybrid, 2014. URL https://www.youtube.com/watch?v=XkSS_veoSg0.
- M. Müller, N. Chentanez, und T.-Y. Kim. Real time dynamic fracture with volumetric approximate convex decompositions. *ACM Transactions on Graphics*, 32(4):1, 2013.
- C. E. V. Muniz und E. W. G. Clua. Finding Surface Normals From Voxels. Technical report, Universidade Federal Fluminense, 2008. URL https://www.ppmsite.com/sibgrapi2007/finding_surface_normals_from_voxels.pdf.
- K. Museth. VDB: High-Resolution Sparse Volumes with Dynamic Topology. *ACM Trans. Graph.*, 32(3):1–22, 2013.
- F. K. Musgrave, C. E. Kolb, und R. S. Mace. The synthesis and rendering of eroded fractal terrains. *Computer Graphics*, 23(3):41–50, 1989.
- A. Ngan, F. Durand, und W. Matusik. Experimental Validation of Analytical BRDF Models. In *Proceedings of SIGGRAPH 2004 Sketches*, 2004. URL <https://web.archive.org/web/20180402202228/https://geidav.wordpress.com/2014/08/18/advanced-octrees-2-node-representations/>.
- F. E. Nicodemus. Directional Reflectance and Emissivity of an Opaque Surface. *Applied Optics*, 4(2):767–773, 1965.
- D. R. Pavia und C. Crassin. Animating ultra-complex voxel scenes through shell deformation. Technical report, Université Joseph Fourier, 2010. URL <https://hal.inria.fr/hal-00840637/document>.

- D. R. Peachey. Solid texturing of complex surfaces. *Proceedings of SIGGRAPH*, Seiten 279–286, 1985.
- K. Perlin. An image synthesizer. *Proceedings of SIGGRAPH*, Seiten 287–296, 1985.
- K. Perng, W. Wang, M. Flanagan, und M. Ouhyoung. A real-time 3D virtual sculpting tool based on modified marching cubes. *International Conference on Artificial Reality and Tele-Existence*, Seiten 64–72, 2001.
- B. T. Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, 1975.
- S. D. Porumbescu, B. Budge, L. Feng, und K. I. Joy. Shell maps. *ACM Trans. Graph.*, 24(3):626–633, 2005.
- M. Pritchard. Direct access quadtree lookup. In M. DeLoura, editor, *Game Programming Gems 2*, Kapitel 4.5, Seiten 394–401. Charles River Media, 2001.
- H. Qu, M. Pan, B. Wang, Y. Wang, und Z. Wang. Boolean operations on triangulated solids and their applications in 3D geological modelling. In *Advances in Spatio-Temporal Analysis*, Seiten 21–26. Taylor & Francis, 2008.
- T. Ritschel, T. Grosch, und H.-P. Seidel. Approximating dynamic global illumination in image space. In *I3D Proceedings*, Seiten 75–82, 2009.
- T. Ritschel, C. Dachsbacher, T. Grosch, und J. Kautz. The state of the art in interactive global illumination. *Computer Graphics Forum*, 31(1):160–188, 2012.
- C. Saona, I. Navazo, und A. Vinacua. Geometric Transformations in Octrees using Shears. Technical Report December, Universitat Politècnica de Catalunya, 1997. URL <https://upcommons.upc.edu/bitstream/handle/2117/96515/R97-62.pdf>.
- P. Su und R. L. Scot Drysdale. A comparison of sequential Delaunay triangulation algorithms. *Computational Geometry*, 7(5-6):361–385, 1997.
- S. Thiedemann, N. Henrich, T. Grosch, und S. Müller. Voxel-based Global Illumination. In *I3D '11 Proceedings*, Seiten 103–110, 2011.
- R. Tonge, F. Benevolenski, und A. Voroshilov. Mass splitting for jitter-free

- parallel rigid body simulation. *ACM Transactions on Graphics*, 31(4):1–8, 2012.
- V. J. Tsai. Delaunay triangulations in TIN creation: An overview and a linear-time algorithm. *International Journal of Geographical Information Systems*, 7(6):501–524, 1993.
- J. van Waveren und I. Castaño. Real-Time Normal Map DXT Compression, 2008. URL <http://www.nvidia.com/object/real-time-normal-map-dxt-compression.html>.
- L. Wade und R. E. Parent. Automated generation of control skeletons for use in animation. *Visual Computer*, 18(2):97–110, 2002.
- R. Wolfe. Teaching Texture Mapping Visually. *SIGGRAPH Education Slide Set*, 1997.
- E. Wood. *Three Dimensional Fractal Exploration using a Scale-Adaptive Sparse Voxel Octree*. Dissertation, Gonville & Caius College, 2012. URL http://www.errollw.com/assets/eww23_part2_diss.pdf.
- R. S. Wright, N. Haemel, G. Sellers, und B. Lipchak. *OpenGL SuperBible*, volume 33. Addison-Wesley, 2012. ISBN 9781604138795.
- R. Zayer, C. Rössl, Z. Karni, und H. P. Seidel. Harmonic guidance for surface deformation. *Computer Graphics Forum*, 24(3):601–609, 2005.

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Datum:

(Unterschrift)