UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Vergleich von Voxelisierungsstrategien auf der GPU

Bachelorarbeit

zur Erlangung des Grades eines Bachelor of Science (B.Sc.)
im Studiengang Computervisualistik

vorgelegt von
Arend Buchacher

Erstgutachter: Prof. Dr.-Ing. Stefan Müller
(Institut für Computervisualistik, AG Computergraphik)
Zweitgutachter: Gerrit Lochmann, M.Sc.
(Institut für Computervisualistik, AG Computergraphik)

Koblenz, im September 2014

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

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

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

Institut für Computervisualistik
AG Computergraphik
Prof. Dr. Stefan Müller
Postfach 20 16 02
56 016 Koblenz
Tel.: 0261-287-2727
Fax: 0261-287-2735
E-Mail: stefanm@uni-koblenz.de

UNIVERSITÄT KOBLENZ · LANDAU

Fachbereich 4: Informatik

Aufgabenstellung für die Bachelorarbeit
Arend Buchacher (Matr.-Nr. 211 100 073)

Thema: **Vergleich von Voxelisierungsstrategien auf der GPU**

In keinem Bereich der Informatik hat sich die Hardware so rasant entwickelt wie im Bereich der Computergraphik. Aktuelle GPUs unterstützen dabei neben der normalen Graphik-Pipeline auch sehr allgemein zu programmierende Compute-Shader. Die Herausforderung besteht in der Untersuchung, wie herkömmliche Graphik-Verfahren sich mit Hilfe der neuen Möglichkeiten umsetzen lassen und wo die Möglichkeiten und Grenzen der Compute-Shader liegen.

Ziel dieser Arbeit ist es, verschiedene Verfahren zur Voxelisierung einer Szene mit Hilfe der GPU zu recherchieren und zu analysieren. Darauf aufbauend sollen ausgewählte Verfahren implementiert und verglichen werden. Optional ist eine Verwendung der Voxelisierung in Beispielanwendungen, oder eine Verbindung mit dem Linespaceverfahren.

Schwerpunkte dieser Arbeit sind:

1. Einarbeitung in Compute-Shader und Recherche bisheriger Ansätze
2. Konzeption und Implementierung der Verfahren
3. Optionale Erweiterungen
4. Demonstration und Bewertung der Ergebnisse
5. Dokumentation

Koblenz, den 1.4.2014

A. Buchacher

- Arend Buchacher -

S. Müller

- Prof. Dr. Stefan Müller-

Zusammenfassung

Die folgende Arbeit vermittelt einen grundlegenden Überblick über die Funktionsweise und Implementierung von aktuellen Voxelisierungsstrategien auf der GPU. Neben etablierten Voxelisierungsverfahren mithilfe der Rasterisierungspipeline werden neue Möglichkeiten mithilfe von GPGPU-Programmierung untersucht. Auf der Basis der Programmiersprache C++ und der Grafikbibliothek OpenGL wird die Implementierung mehrerer Verfahren erläutert. Die Verfahren werden hinsichtlich der Performanz und der Qualität der Voxelisierung verglichen und im Bezug auf mögliche Anwendungsfälle kritisch bewertet. Weiterhin werden zwei Beispielanwendungen beschrieben, in denen die Verwendung einer voxelisierten Szene eine Erweiterung von bestehenden Echtzeitgrafikverfahren ermöglicht. Zu diesem Zweck werden die Konzepte und die Implementierungen von Transmittance Shadow Mapping und von Reflective Shadow Mapping, das um voxelbasierte Umgebungsverdeckung erweitert wird, erläutert. Abschließend wird die anhaltende Relevanz von Voxelisierung in einem Ausblick auf aktuelle Forschungen und weitere Anwendungen und Erweiterungen der vorgestellten Verfahren aufgezeigt.

Abstract

The following thesis imparts a general view of the mechanics and implementation of latest voxelization strategies using the GPU. In addition to established voxelization procedures using the rasterization pipeline, new possibilities arising from GPGPU programming are examined. On the basis of the programming language C++ and the graphics library OpenGL the implementation of several methods is explained. The methods are compared in terms of performance and quality of the resulting voxelization and are evaluated critically with regards to possible use cases. Furthermore, two exemplary applications are detailed that use a voxelized scene in such a way that the augmentation of established techniques of real time graphics are facilitated. To this end, the concepts and the implementations of Transmittance Shadow Mapping and of Reflective Shadow Mapping utilizing a voxel based ambient occlusion effect is explained. Finally, the prolonging relevance of voxelization is put into prospect, by addressing latest research and further enhancements and applications of the presented methods.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Volumendaten spielen in der Computergrafik seit Langem eine große Rolle. Sie bieten eine Alternative zu der traditionellen geometrischen Repräsentation durch Polygone. Durch die Visualisierung von volumetrischen Scandaten sind sie zu einem fundamentalen Teil in der medizinischen Bildgebung geworden. Doch auch in anderen Bereichen der Computergrafik ist eine diskrete, angenäherte Repräsentation der virtuellen Szene häufig von großem Nutzen. Die Anwendungsgebiete reichen von der Beschleunigung von Ray-Tracing ([Amanatides und Woo, 1987]) über Kollisionserkennung ([Lawlor und Kalée, 2002]) bis zu konstruktiver Festkörpergeometrie in CAD-Anwendungen ([Liao und Fang, 2002]) und weiter.

Um ein Polygonmodell für volumetrische Berechnungen nutzen zu können, ist erst eine entsprechende Konvertierung in eine diskrete, volumetrische Repräsentation nötig. Diese Konvertierung wird als *Voxelisierung* bezeichnet. Die ersten Voxelisierungsverfahren nach [Kaufman und Shimony, 1987] wurden lange als aufwendiger Vorverarbeitungsprozess ausgeführt. Erste Voxelisierungen mithilfe der GPU konnten mit dem Verfahren von [Fang et al., 2000] durchgeführt werden. Seitdem sind zahlreiche effiziente Verfahren erschienen, die die hardwarebeschleunigte Rasterisierungspipeline zu diesem Zweck nutzen.

Seit einiger Zeit wird die Allzweckprogrammierung auf der GPU (*GPGPU*) in rasanter Geschwindigkeit vorangetrieben und zugänglicher. Insbesondere in der Verbindung mit traditionellen Grafikanwendungen besteht großes Potential für die Ausnutzung der Grafikhardware für anfallende Aufgaben. Es gibt Verfahren, die durch diese neuen Möglichkeiten auf die Zweckentfremdung der Rasterisierungspipeline verzichten können.

In dieser Arbeit soll ein Überblick über verschiedene Verfahren zur Voxelisierung einer Szene mithilfe der GPU gegeben werden. Es soll gezeigt werden, wie die Verfahren implementiert werden können und mit welchen Implikationen dabei zu rechnen ist. Es sollen sowohl Voxelisierungsverfahren mithilfe der Rasterisierungspipeline als auch flexible Lösungen mithilfe von GPGPU-Programmierung vorgestellt werden. Die Verfahren sollen analysiert und verglichen werden, um anwendungsbezogene Nutzbarkeit und Unterschiede zwischen den Verfahren herauszustellen. Weiterhin soll der Nutzen einer voxelbasierten Repräsentation der Szene demonstriert werden, indem unterschiedliche Anwendungen der Voxelisierung im Umfeld der Echtzeitgrafik erläutert und implementiert werden. Weiterführend soll ein Einblick in mögliche Erweiterungen, aktuelle Entwicklungen und verwandte Verfahren gegeben werden.

In Abschnitt 2 werden die Grundlagen der Voxelisierung und die für diese Arbeit relevanten OpenGL-Funktionen erläutert. In Abschnitt 3 wird eine Auswahl von Voxelisierungsverfahren erklärt und die Implementierung erläutert. In Abschnitt 4 werden die Verfahren im Bezug auf Perfor-

manz und Qualität der Ergebnisse verglichen. Die Verfahren werden im Bezug auf den Anwendungsfall auf ihre Eignung hin bewertet. In Abschnitt 5 werden zwei implementierte Beispielanwendungen erläutert und die Ergebnisse präsentiert. In Abschnitt 6 werden mögliche Verbesserungen und Erweiterungen der implementierten Verfahren und weiterführende Verfahren erläutert.

2 Grundlagen

In diesem Abschnitt werden die für diese Arbeit nötigen Grundlagen erklärt. Zunächst werden einige grundlegende Begriffe der Voxelisierung von virtuellen Objekten erläutert. Außerdem wird die zur Implementierung der Verfahren notwendige Verwendung einiger Aspekte der Grafikbibliothek OpenGL erläutert.

2.1 Voxelisierung

Die Voxelisierung einer Szene beschreibt eine diskrete Abbildung eines Raumes in eine finite Anzahl von Elementen. Algorithmen, die der Voxelisierung dienen, werden auch *3D-Scan-Conversion* Algorithmen genannt, da die Scan-Konvertierung von Geraden oder Linien der Diskretisierung dient. Der Vergleich zur Rasterisierung, bei der eine Szene auf eine finite Anzahl von Pixeln diskretisiert wird, liegt nahe. Die Voxelisierung kann als Rasterisierung der Szene in ein dreidimensionales Abbild aus einer finiten Anzahl von Volumenelementen verstanden werden. Ein dreidimensionales Volumenelement wird als Voxel bezeichnet, in Anlehnung an die Bezeichnung Pixel für ein zweidimensionales Bildelement (engl. *picture element*).

Die Voxelisierung einer Szene kann auf verschiedene Arten geschehen. Im Allgemeinen lassen sich je nach Voxelisierungsstrategie Unterschiede in den nachfolgenden Eigenschaften und Herangehensweisen feststellen.

2.1.1 Voxelgitter-Strukturen

Das durch die Voxelisierung entstehende Volumen wird im folgenden als Voxelgitter bezeichnet. Sind die Voxel in allen Dimensionen gleich groß, d.h. würfelförmig, so wird das Voxelgitter als uniform oder einheitlich bezeichnet. Oft ist aber auch eine uneinheitliche Form gewünscht, beispielsweise, wenn entlang eines Kamera-Frustums voxelisiert werden soll. Abbilding 1 zeigt schematisch, wie eine derartige Voxelgitterstruktur aussehen könnte.

Auch hierarchische Voxelgitterstrukturen können bestehen, sodass in jeder Hierarchiestufe die Information von mehreren Voxeln der darunterliegenden Stufe zusammengefasst wird. Im diesem Zusammenhang sind

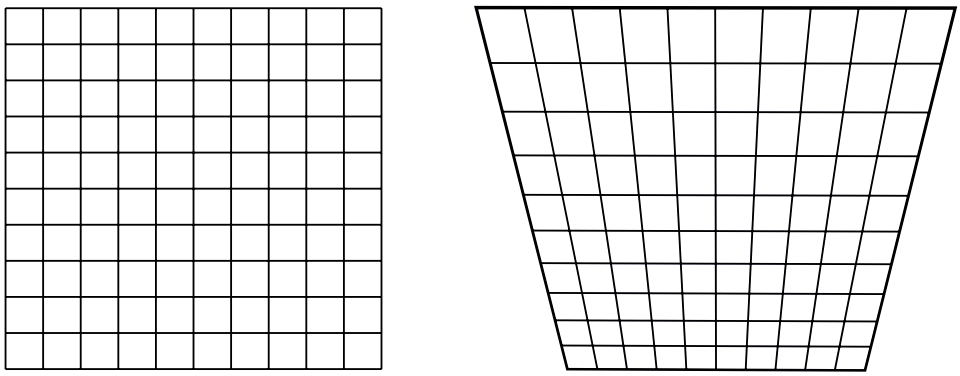


Abbildung 1: Eine einheitliche 2D-Voxelgitterstruktur und eine uneinheitliche 2D-Voxelgitterstruktur der Auflösung 10x10.

insbesondere *Octree*-Strukturen ein häufig angewandtes Konzept. In Abschnitt 3.5 wird der Aufbau einer Voxelgitterhierarchie beschrieben.

Werden in der Voxelgitterstruktur leere Bereiche ausgelassen, so kann von einer sparsamen (engl. *sparse*) Voxelgitterstruktur gesprochen werden. Eine solche Struktur wird häufig in Verbindung mit einer hierarchischen Voxelgitterstruktur verwendet. In Abschnitt 6.2 wird ein Verfahren beschrieben, das eine solche Struktur verwendet.

2.1.2 Surface oder Solid Voxelization

Die Oberflächenvoxelisierung (engl. *surface voxelization* oder *boundary voxelization*) beschreibt die Voxelisierung einer Szene, wobei nur die Oberflächen der Modelle erfasst werden. Voxel, die sich im Inneren eines Modells befinden und keine Oberfläche schneiden, werden als nicht belegt gesetzt. Die Modelle werden als hohl interpretiert. Weiterhin bezeichnet eine konservative Voxelisierung eine Voxelisierung, bei der ein Voxel als belegt gilt, wenn es von einer Oberfläche geschnitten oder auch nur berührt wird. Eine Voxelisierung, bei der der Mittelpunkt des Voxels überlappt sein muss, kann mit der Standardrasterisierung verglichen werden.

Die solide, bzw. wasserdichte Voxelisierung (engl. *solid voxelization*) beschreibt die Voxelisierung einer Szene, wobei auch das Innere der Modelle erfasst wird.

So werden Voxel, die sich im Inneren eines Modells befinden, als belegt gesetzt. Voraussetzung für eine fehlerfreie wasserdichte Voxelisierung ist häufig, dass die Modelle keinerlei Lücken aufweisen, bzw. geschlossen sind.

2.1.3 Binary oder Multi-Valued Voxelization

Eine binäre Voxelisierung (engl. *binary voxelization*) liegt vor, wenn die Voxel keine weitere Information enthalten, außer ob sie belegt oder nicht belegt sind. Im Rahmen dieser Arbeit wurden einige Verfahren implementiert, die zu dieser Art von Voxelisierung führen.

Eine mehrwertige Voxelisierung (engl. *multi-valued voxelization*) liegt vor, wenn weitere Informationen pro Voxel gespeichert werden. Häufig sind dies die Oberflächen-Normale, Referenzen zu geschnittenen Dreiecken oder die Lichtintensität.

2.2 OpenGL

Für die Implementierung der Verfahren in dieser Arbeit wird die offene Grafikbibliothek *OpenGL* verwendet. Die dieser Arbeit zugrunde liegende Spezifikation ist die 2012 veröffentlichte Version 4.3[1]. In dieser Version wurden Compute-Shader eingeführt, mit deren Hilfe sich universelle Berechnungen ausführen lassen. Es wurde die mit dieser Version erschienene Spezifikation[2] der *OpenGL Shading Language* (kurz GLSL) Version 4.30 verwendet.

2.2.1 Texturzugriffe

In dieser Arbeit spielen Texturzugriffe eine wichtige Rolle. Das Speicherformat und die lesenden und schreibenden Zugriffe aus Shader-Programmen heraus besitzen einige Feinheiten, die unbedingt beachtet werden müssen, um die Funktionalität der Verfahren garantieren zu können. Die in Abschnitt 3 beschriebenen Verfahren stützen sich oftmals auf die Möglichkeit des expliziten Zugriffs auf bestimmte Bits von Texturwerten. In OpenGL ist dies nur mit manchen Speicherformaten unkompliziert möglich. Es eignen sich insbesondere ganzzahlige Datentypen ohne Vorzeichen. Diese Formate sollten möglichst sowohl zur Speicherverwaltung als auch als Eingabetyp und Rückgabetyp von Texturzugriffen verwendet werden. Ein Texturelement wird als *Texel* bezeichnet, wobei Texel im Folgenden synonym zu Pixel verwendet wird.

Soll der Speicherplatz einer Textur mit OpenGL allokiert werden, so gibt es seit OpenGL Version 4.2 die Funktion *glTexStorage()*[3], die eine unveränderbare Struktur einer Textur festlegt. Sie legt das interne Speicherformat der Texturdaten mit expliziter Bitanzahl und die Dimensionen des

[1] [Segal und Akeley, 2012], `http://www.opengl.org/registry/doc/glspec43.core.20120806.pdf` (Zugriff: 28.09.2014)

[2] [Kessenich et al., 2012], `http://www.opengl.org/registry/doc/GLSLangSpec.4.30.6.pdf` (Zugriff: 28.09.2014)

[3] OpenGL Spezifikation: 8.19 Immutable-Format Texture Images

Speicher-Arrays fest. So kann als internes Format beispielsweise der konstante Parameter *GL_RGBA8* gewählt werden, um pro Texel einen RGBA-Wert mit jeweils 8 Bit zu speichern. Diese RGBA-Werte werden in einem Shader als normalisierte Gleitkommazahl zwischen 0.0 und 1.0 ausgelesen, doch intern als ganzzahliger Wert zwischen 0 und 255 gespeichert. Das interne Format *GL_R32UI* speichert pro Texel einen einzelnen vorzeichenlosen Integerwert mit 32 Bit. Um diesen Wert in einem Shader auszulesen, muss er zwar im vorzeichenlosen Integer-RGBA-Format angefordert werden, der eigentliche Wert des Texels befindet sich jedoch im Rot-Kanal.

Es gibt verschiedene Arten, aus einem Shader auf eine Textur zuzugreifen. Die üblichste Methode zum Lesen einer Textur ist der Zugriff mithilfe einer uniformen *sampler*-Variable[4]. Ihr Typ sollte so gewählt werden, dass sie den Datentyp der Textur widerspiegelt, aus der sie liest. Eine 2D-Textur des Formats *GL_R32UI* sollte von einem *usampler2D* gelesen werden. Ein Sampler liefert immer einen Vektor mit vier Komponenten zurück, wobei Kanäle, die die zugrundeliegende Textur nicht verwendet, auf 0 gesetzt werden.

Der willkürlich lesende und schreibende Zugriff auf Texturen aus einem beliebigen Shader-Stadium ist seit der in Version 4.2 beinhalteten Erweiterung *Image Load Store* möglich. Nunmehr ist das Binden von Texturen an Framebuffer nicht mehr die einzige Möglichkeit, um aus einem Shader in eine Textur zu schreiben. Auch für Compute-Shader bietet diese Erweiterung eine Möglichkeit um das Schreiben in Texturen zu realisieren. Die Erweiterung ermöglicht es, ein Level einer Textur an eine neue Art von Textur-Schnittstelle, eine *image unit*, zu binden, das die direkte Manipulation der Bilddaten erlaubt. Dies steht im Gegensatz zum Binden von Textur-Objekten an *texture units*, die in GLSL mit den oben genannten *sampler*-Variablen verwendet werden.

In einem Shader kann der Zugriff auf ein an eine *image unit* gebundenes Texturlevel mithilfe einer uniformen *image*-Variable realisiert werden. Dies gestaltet sich ähnlich zu der Verwendung von Samplern, es müssen allerdings noch weitere Qualifier gesetzt werden. Zunächst muss der zum internen Datenformat der Textur korrespondierende Formatqualifier[5] gesetzt werden. Außerdem muss der zur Verwendungsart passende Zugriffsqualifier gesetzt werden, der während des Bindens mit *glBindImageTexture()*[6] festgelegt wurde.

Es existieren außerdem Funktionen, um atomare Operationen auf *image*-Variablen auszuführen[7]. Auf diese Weise kann ein Texel von vielen Shadern bearbeitet werden, ohne dass die parallele Bearbeitung zu Inkohärenz zwischen den Shadern führt. Dies ist möglich, da eine atomare Ope-

[4]GLSL Spezifikation: 8.9.2 Texel Lookup Functions
[5]GLSL Spezifikation: 4.4.6.2 Format Layout Qualifiers
[6]OpenGL Spezifikation: 8.25 Texture Image Loads and Stores
[7]GLSL Spezifikation: 8.12 Image Functions

ration immer erst abgeschlossen sein muss, bevor der nächste Shader eine atomare Operation ausführen darf. Unter anderem sind auch die bitweisen logischen Verknüpfungsoperationen *imageAtomicOr()*, *imageAtomicXor()* und *imageAtomicAnd()* vorhanden. Atomare Operationen sind allerdings nur auf ganzzahlige *image*-Variablen im einkanäligen ganzzahligen Datenformat anwendbar. Das heißt, die zugrundeliegende Textur muss im vorzeichenlosen Fall in dem Format *GL_R32UI* vorliegen und die *image*-Variable den Formatqualifier *r32ui* besitzen. Außerdem muss die gesetzte Zugriffsart das Lesen und Schreiben erlauben.

```
// variable declarations
layout( binding = 0 ) uniform  sampler2D samplerVar1;
layout( binding = 1 ) uniform usampler2D samplerVar2;

layout( rgba8, binding = 0 ) uniform readonly image2D  imageVar1;
layout( r32ui, binding = 1 ) uniform           uimage2D imageVar2;
```

Quellcode 1: Deklaration von sampler- und image-Variablen

Quellcode 1 zeigt die beispielhafte Deklaration von *sampler*- und *image*-Variablen in einem Shaderprogramm. Es werden zwei 2D-*sampler*-Variablen deklariert, wobei die zweite Variable ein vorzeichenfreies Integer-Format besitzt. Es werden außerdem zwei 2D-*image*-Variablen deklariert, wobei die erste Variable ausschließlich lesenden Zugriff erlauben soll und 8-Bit Gleitkommazahlen im RGBA-Format besitzt. Die zweite Variable besitzt ein einkanäliges vorzeichenfreies Integer-Format.

```
// read texels using samplers
 vec4 sample1 = texture( samplerVar1, vec2(0.5,0.5));
uvec4 sample2 = texture( samplerVar2, vec2(0.5,0.5));

// read texels using images
ivec2 res     = imageSize( imageVar1 );
 vec4 texel1  = imageLoad( imageVar1, res / 2 );
      res     = imageSize( imageVar2 );
uvec4 texel2  = imageLoad( imageVar2, res / 2 );
 uint value   = texel2.r;

// optional memory barrier
// memoryBarrier();

 uint before  = imageAtomicOr( imageVar2, res / 2, 256u );
```

Quellcode 2: Zugriff auf *sampler*- und *image*-Variablen

Quellcode 2 zeigt die beispielhafte Verwendung von *sampler*- und *image*-Variablen in einem Shaderprogramm. Es wird bei beiden 2D-*sampler*-Variablen mithilfe von *texture()* der Mittelpunkt der Texturen ausgelesen. Sampler stützen sich dabei auf die in der Textur eingestellte Texturfiltermethode, sodass eventuell zwischen den nächsten Texeln interpoliert wird, wenn die gewählte Texturkoordinate nicht auf dem Mittelpunkt eines Texels liegt.

Um mithilfe von *imageLoad()* ein Texel zu lesen, wird dessen Bildkoordinate explizit angegeben. Daher wird die Auflösung der Texturen über die *image*-Variablen angefordert und halbiert als Bildkoordinate verwendet, um das mittlere Texel anzufordern. Da die zweite *image*-Variable den Formatqualifier *r32ui* besitzt, wird der tatsächliche Texelwert aus dem Rotkanal gelesen. Abschließend wird der Texelwert durch die atomare Operation *imageAtomicOr()* mit dem vorzeichenlosen Integer-Wert 256 verknüpft. Diese Funktion liefert den Texturwert vor Ausführung der Operation zurück. Eine Speicherbarriere[8] kann in Zeile 13 verwendet werden, um zu garantieren, dass der Wert von **value** in jeder Shader-Instanz gleich ist. Es ist sonst möglich, dass eine Shader-Instanz den Texturwert verändert, bevor eine andere Shader-Instanz ihn das erste Mal ausliest. Nachfolgend eine mögliche Beispielsequenz von drei parallel ausgeführten Shader-Instanzen, die ohne eine Speicherbarriere zu diesem Ergebnis führen:
Shader 1 liest den Wert 0 in **value** ein.
Shader 2 liest den Wert 0 in **value** ein.
Shader 1 schreibt den Wert 256 und liest den Wert 0 in **before** ein.
Shader 3 liest den Wert 256 in **value** ein.
Shader 2 schreibt den Wert 256 und liest den Wert 256 in **before** ein.
Shader 3 schreibt den Wert 256 und liest den Wert 256 in **before** ein.
Durch die Speicherbarriere wird garantiert, dass alle Methodenaufrufe der Compute-Shader bis zu dieser Zeile beendet sind, bevor das Programm weiter läuft. Da hier eine Kohärenz der Werte aber nicht nötig ist, kann auf die Speicherbarriere verzichtet werden. Im Hauptprogramm können Speicherbarrieren mithilfe von *glMemoryBarrier()*[9] bei Bedarf verwendet werden.

2.2.2 Compute-Shader

Mit der Spezifikation von OpenGL 4.3 wurde ein neuer Shader-Typ, der Compute-Shader[10], eingeführt. Dieser ist abgekoppelt von der Rasterisierungspipeline zu betrachten und wurde zu *GPGPU*-Zwecken eingeführt.

GPGPU steht für *General Purpose Computation on Graphics Processing Units* (Allzweck-Berechnungen auf Grafikprozessoreinheiten) und erfährt weiterhin wachsendes Interesse, wie von [Owens et al., 2007] untersucht. Auch die stete Weiterentwicklung der für GPGPU ausgelegten Programmierschnittstellen CUDA (kurz für *Compute Unified Device Architecture*) und OpenCL (kurz für *Open Computing Language*) sprechen dafür. Compute-Shader sind als fester Bestandteil von OpenGL aus Grafikanwendungen leichter zugänglich, im Umfang dadurch aber auch eingeschränkter als beispielsweise OpenCL.

[8] GLSL Spezifikation: 8.17 Shader Memory Control Functions
[9] OpenGL Spezifikation: 7.12.2 Shader Memory Access Synchronization
[10] OpenGL Spezifikation: 19 Compute Shaders

Compute-Shader besitzen eine von anderen Shader-Programmen abweichende Semantik. Zwar ist die Syntax vorgegeben durch GLSL, doch ist die Handhabung grundlegend verschieden zu der von anderen Shader-Stadien. Im Gegensatz zu beispielsweise Vertex-Shadern oder Fragment-Shadern ist die Anzahl der gestarteten Shader-Instanzen nicht unbedingt von der Anzahl von Vertices oder erzeugten Fragmenten abhängig. Es gibt keine vordefinierten Eingaben oder Ausgaben, wie etwa ein Vertex-Attribut oder die Pixelfarbe.

Compute-Shader sind zu Gruppen in einem abstrakten dreidimensionalen Raum angeordnet, der gemeinhin als *Compute Space* bezeichnet wird. Die Bedeutung der Raumdimensionen ist im weitesten Sinne von dem Benutzer zu bestimmen. Der Compute Space dient in erster Linie dazu, einzelne Instanzen und Gruppen von Compute-Shadern zu identifizieren. Da auch die Anzahl der gleichzeitig von einem Grafikprozessor bearbeitbaren Shader-Instanzen begrenzt ist, werden gleichgroße Gruppen von Compute-Shadern zu jeweils einer lokalen Arbeitsgruppe (*local work group*) zusammengefasst. Die Größe der lokalen Arbeitsgruppe ist die einzige vom Benutzer festzulegende Eingabe eines Compute-Shaders. Sie bestimmt die kleinste mögliche Einheit von Compute-Shadern, die ausgeführt werden kann. Die Gesamtheit der lokalen Arbeitsgruppen wird als globale Arbeitsgruppe (*global work group*) bezeichnet. Eine lokale Arbeitsgruppe besitzt einen gemeinsamen Speicherbereich und wird gleichzeitig auf genau einem Grafikkartenprozessor bearbeitet.

Eine globale Arbeitsgruppe von Compute-Shadern kann zu jeder Zeit im Programmfluss gestartet werden. Dazu gibt es die OpenGL-Funktion *glDispatchCompute(int num_groups_x, int num_groups_y, int num_groups_z)*, bei der die Parameter jeweils die Anzahl der lokalen Arbeitsgruppen in der betreffenden Dimension bestimmen. Es sei an dieser Stelle darauf hingewiesen, dass das System nicht garantiert, dass die lokalen Arbeitsgruppen sequentiell in jeder Dimension abgearbeitet werden. Eine Abhängigkeit zwischen den lokalen Arbeitsgruppen sollte daher unbedingt vermieden werden. Die maximalen Werte für die Größe der Arbeitsgruppen und des gemeinsamen Speicherplatzes sind von der Hardware abhängig. Die möglichen Anfragen sind der Spezifikation zu entnehmen[11].

Es gibt einige vordefinierte Variablen, die die genaue Identifizierung einer Instanz eines Compute-Shaders ermöglichen.

- *uvec3 gl_WorkGroupSize* enthält die im Shader definierte Größe einer lokalen Arbeitsgruppe.
- *uvec3 gl_NumWorkGroups* enthält die Parameter mit denen *glDispatchCompute()* aufgerufen wurde.

[11] OpenGL Spezifikation: Table 23.62. Implementation Dependent Compute Shader Limits

- *uvec3 gl_WorkGroupID* enthält den Index der Arbeitsgruppe zu der der Compute-Shader gehört.
- *uvec3 gl_LocalInvocationID* enthält den Index der Instanz innerhalb der Arbeitsgruppe.
- *uvec3 gl_GlobalInvocationID* enthält den globalen Index dieser Instanz und berechnet sich aus der Größe einer lokalen Arbeitsgruppe, dem Index der lokalen Arbeitsgruppe und dem Index innerhalb der lokalen Arbeitsgruppe. Er kann als Koordinate des Compute-Shaders im Compute Space verstanden werden.

Die in dieser Arbeit am häufigsten gebrauchte Identifikationsvariable ist dabei *gl_GlobalInvocationID*.

Ein Compute-Shader-Programm wird analog zu den anderen Shader-Typen von OpenGL erstellt. Die Erstellung durch die Funktion *glCreateShader(GLenum shaderType)* erfolgt dann mit dem konstanten Parameter *GL_COMPUTE_SHADER*. Der Quellcode wird wie gewohnt an die Funktion *glShaderSource()* übergeben und mit *glCompileShader()* kompiliert. Ist das Compute-Shader-Programm nun mithilfe von *glUseProgram(GLuint program)* aktiviert worden, kann *glDispatchCompute()* wie oben beschrieben zum Ausführen verwendet werden.

Zugriffe auf Texturen oder Buffer-Objekte müssen während der Ausführung eines Compute-Shaders realisiert werden. Dazu eignen sich die in Abschnitt 2.2.1 beschriebenen Texturzugriffsmethoden mit *imageLoadStore* oder die Verwendung der neuen Buffer-Schnittstelle, die im nachfolgendem Abschnitt 2.2.3 erläutert wird, um etwa den Zugriff auf Vertex-Buffer oder Index-Buffer zu realisieren.

2.2.3 Shader Storage Buffer Object

Ein *Shader Storage Buffer Object*[12] (kurz SSBO) ist ein Buffer-Objekt, das aus einem Shader gelesen oder geschrieben werden kann. Da Compute-Shader keinen vordefinierten Input oder Output besitzen, muss der Zugriff auf Modell-Informationen manuell konfiguriert werden. Für die folgenden Implementierungen wurden sie verwendet, um aus einem Compute-Shader auf Vertex-Informationen zuzugreifen.

Um ein bereits auf der Grafikkarte bestehendes Buffer-Objekt auszulesen, muss es zunächst an einen SSBO-Bindepunkt gebunden werden. Dazu kann die Methode *glBindBufferBase(GLenum target, GLuint index, GLuint buffer)*[13] verwendet werden. Diese Funktion bindet das Buffer-Objekt *buffer* an die durch *target* und *index* spezifizierte Schnittstelle. Gleichzeitig legt

[12] `http://www.opengl.org/registry/specs/ARB/shader_storage_buffer_object.txt` (Zugriff: 26.09.2014)

[13] OpenGL Spezifikation: 6.1.1 Binding Buffer Objects to Indexed Targets

die Methode den zulässigen Speicherbereich für Zugriffe aus einem Shader fest.

Im Folgenden wird beschrieben, wie ein Index-Buffer eines Dreieckmodells in einem Compute-Shader ausgelesen werden kann. Zunächst wird der Index-Buffer an den Index 0 der möglichen SSBO-Bindepunkte gebunden. Ein Aufruf an die Methode könnte wie in Quellcode 3 erfolgen.

```
glBindBufferBase(
    GL_SHADER_STORAGE_BUFFER,    // target
    0,                           // target binding index
    indexBufferHandle);          // buffer handle
```

Quellcode 3: Binden eines Buffer-Objekts an einen Buffer-Bindepunkt

Der Zugriff auf einen als SSBO gebundenen Buffer erfolgt in dem Shader-Programm ähnlich wie auf ein Array. Der Buffer wird im Shader-Programm mit dem gleichen Bindepunktindex als Buffer-Objekt deklariert. Ein zusätzlicher Qualifier bestimmt das erwartete Speicherlayout[14]. Soll beispielsweise der Index-Buffer ausgelesen werden, könnte die passende Variable wie in Quellcode 4 deklariert werden. Die Annahme ist, dass der Buffer aus einem Array von vorzeichenfreien Integerwerten besteht. Im Folgenden wurde stets der Qualifier *std430* verwendet, der insbesondere den Zugriff auf einen aus aufeinanderfolgenden Einzelwerten bestehenden Buffer erleichtert. Eine weiterführende Beschreibung der korrekten Verwendung des Speicherlayout-Qualifiers ist der Spezifikation zu entnehmen.

```
layout(std430, binding = 0) buffer IndexBuffer{ uint indices[]; };
```

Quellcode 4: Deklaration einer Buffer-Variable

Quellcode 5 zeigt beispielhaft, wie die Indizes eines Dreiecks aus einem so deklarierten Index-Buffer gelesen werden könnten. Mithilfe der *gl_GlobalInvocationID*-Variable des Compute-Shaders werden drei aufeinanderfolgende Indizes aus dem Index-Buffer ausgelesen. Diese Indizes könnten mithilfe des entsprechenden Vertex-Buffers verwendet werden, um die Vertex-Informationen auszulesen.

```
uint gid = gl_GlobalInvocationID.x;

uint indexV0 = indices[ gid * 3 + 0 ]; // index of first  vertex
uint indexV1 = indices[ gid * 3 + 1 ]; // index of second vertex
uint indexV2 = indices[ gid * 3 + 2 ]; // index of third  vertex
```

Quellcode 5: Zugriff auf einen Buffer mithilfe der eindeutigen *gl_GlobalInvocationID*

[14] OpenGL Spezifikation: 7.6.2.2 Standard Uniform Block Layout

3 Implementierte Voxelisierungsverfahren

Im Rahmen dieser Arbeit wurde eine Auswahl von bewährten Voxelisierungsverfahren implementiert. Die meisten Verfahren können von der Oberflächenvoxelisierung auf eine solide Voxelisierung angepasst werden. Subjekt dieser Arbeit ist der Prozess der Voxelisierung, im Gegensatz zu der verwendeten Datenstruktur des Voxelgitters. Aus diesem Grund soll in den vorgestellten Verfahren eine binäre Oberflächenvoxelisierung erreicht werden, um eine flexible Grundlage für mögliche Erweiterungen zu schaffen. Weitere verwandte Ansätze und Verfahren werden in Abschnitt 6 in Ausblick gestellt.

Zur Untersützung von OpenGL wurden die offenen Bibliotheken von *GLFW*[15], *GLEW*[16] und *GLM*[17] verwendet. Es wurde die offene Bibliothek *Assimp*[18] zum Importieren von 3D-Modellen aus Dateien verwendet.

Zu beachten ist, dass bei Quellcodeausschnitten zur Anschaulichkeit meist auf die explizite Anwendung von Typumwandlungen verzichtet wurde.

3.1 Fast Scene Voxelization

Das in dieser Arbeit grundlegendste Verfahren zur Voxelisierung mithilfe der GPU ist das von [Eisemann und Décoret, 2006] vorgestellte Verfahren des *Slicemapping*. Das Verfahren nutzt die Rasterisierungspipeline zum Voxelisieren und benötigt bloß einen Vertex-Shader und Fragment-Shader. Das Voxelgitter wird durch eine Textur repräsentiert, die als Framebuffer gebunden ist. Dabei wird eine binäre Oberflächenvoxelisierung erzielt.

3.1.1 Vorgehen

Zunächst wird ein Voxelgitter um die zu voxelisierende Szene definiert. Dazu wird die implizite Definition eines Voxelgitters durch das Rendern einer Szene genutzt. Der Umriss eines Kamerafrustums definiert gemeinsam mit der Auflösung des Viewports und des Framebuffers ein Voxelgitter.

Während des Rasterisierungsprozesses wird für jedes Primitiv eines Modells die passende Position in diesem Gitter gefunden. Jedes produzierte Fragment beschreibt dabei mit seiner Pixel-Position (x, y) und seinem Tiefenwert z eine Zelle des Voxelgitters. Der Tiefenwert z wird bei gewöhnlichem Rendern zum Verwerfen von verdeckten Fragmenten verwendet. Doch obwohl verdeckte Fragmente verworfen werden, hat die Hardware zu diesem Zeitpunkt Zugriff auf diese Information. Die Idee ist, diese

[15] `http://www.glfw.org/` (Zugriff 24.09.2014)
[16] `http://glew.sourceforge.net/` (Zugriff 24.09.2014)
[17] `http://glm.g-truc.net/0.9.5/index.html` (Zugriff 24.09.2014)
[18] `http://assimp.sourceforge.net/` (Zugriff 24.09.2014)

Information statt dessen zu verwenden, um ein Voxelgitter in einer RGBA-Textur zu kodieren.

Zu diesem Zweck wird eine Kamera in der Szene platziert, deren Frustum den zu voxelisierenden Bereich der Szene umschließt. Diese Kamera wird fortan als Voxelisierungskamera bezeichnet. Das Frustum kann sowohl orthographisch als auch perspektivisch sein. Der Voxelisierungskamera wird ein Viewport zugeordnet, dessen Dimensionen (w, h) die Auflösung des Voxelgitters in x- und y- Richtung fest legen. Ein Pixel (x, y) des Viewports entspricht einer Spalte in der Tiefe. Diese Spalte von Zellen soll fortlaufend in den RGBA-Werten kodiert werden. Die Bits der vier einzelnen Farbkanalwerte werden dazu lediglich als ein zusammenhängender Vektor von Bits interpretiert. Ist für jeden Kanal ein 8-Bit-Wert reserviert, ergibt sich folglich eine Gesamttiefe von $4 * 8 = 32$ Zellen. Es wird bei [Eisemann und Décoret, 2006] an dieser Stelle darauf hingewiesen, dass die Verteilung der Zellen in der Tiefe ungleich geschehen kann. Für die folgenden Schritte tritt dieser Fall aber nicht ein, und es werden die Zellen zwischen der vorderen und hinteren Clippingebene verteilt. Die Schnittmenge aller Zellen, die zu einem bestimmten Tiefen-Bit korrespondieren, beschreibt einen Schicht (engl. *slice*) der Textur, welche daher auch als *Slicemap* bezeichnet wird.

Das Rendern der Objekte erfolgt mithilfe einfacher Shader-Programme. Zunächst wird die Textur als *Framebuffer* gebunden und mit der Farbe schwarz gesäubert. Dadurch sind alle Bits auf 0 gesetzt. Es werden die Matritzen für Projektion, Model, View und der Viewport entsprechend der Voxelisierungskamera, der Auflösung des Framebuffers und dem zu voxelisierenden Objekt gesetzt. Durch die Rasterisierung eines Primitiven wird jeweils ein Fragment-Shader pro geschnittener Spalte erzeugt. Aus dem Tiefenwert z des Fragments lässt sich die betroffene Schicht i bestimmen. Im Falle einer orthographischen Projektion ist z linear skaliert. Mithilfe einer Bit-Maske wird dem Tiefenwert ein Wert zugeordnet, der überall eine 0 eingetragen hat, außer einer 1 an der korrespondierenden Bit-Stelle. Dieser Wert wird mithilfe einer bitweisen ODER-Verknüpfung in den Framebuffer geschrieben.

Abbildung 2 zeigt schematisch die Zuordnung eines erzeugten Fragments zu einer korrespondierenden Bitmaske. Es sei darauf hingewiesen, dass in dieser Darstellung die Bit-Reihenfolge (*Endianness*) des Wertes der Bitmaske nicht beachtet wird. Wichtiger ist die Eindeutigkeit der Zuordnung einer Bitmaske zu einer Schicht des Voxelgitters.

Die Bitmasken können für alle möglichen Schichten in einer 1D-Textur bereitgestellt werden. Sie müssen nur einmal berechnet werden, wobei folgende Konvention für einen RGBA-Wert aus 32-Bit verwendet werden kann:

$$\mathrm{b}_i = 2^i. \tag{1}$$

Gleichung 1 zeigt die Berechnung des korrespondierenden Werts einer Bit-

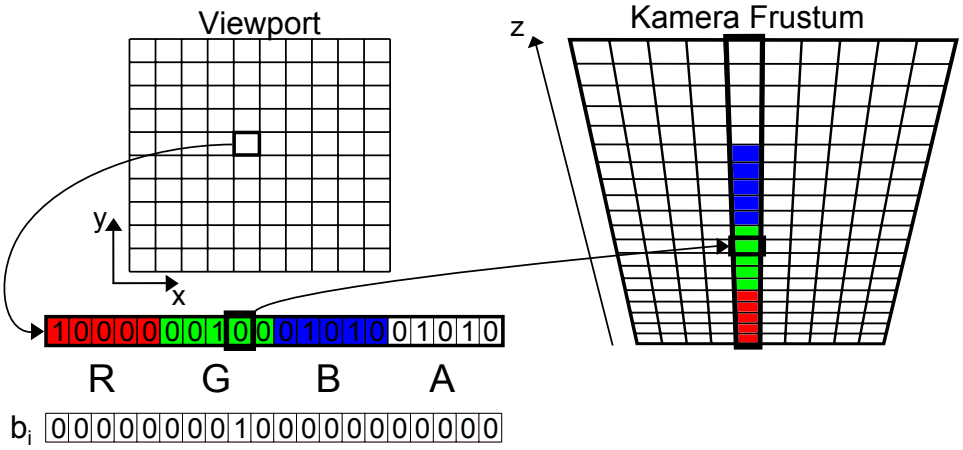


Abbildung 2: Kodierung eines Voxelgitters im Kamerafrustum und Zuordnung einer Schicht zu einer Bitmaske nach [Eisemann und Décoret, 2006].

maske b zur Schicht mit dem Index i. Die Bitmaske der ersten Schicht mit $i = 0$ entspricht also dem Wert 2^0, die der 32. Schicht dem Wert 2^{31}.

3.1.2 Details der Implementierung

Die Schreibmethode des Fragment-Shaders, mit der er in den Framebuffer schreibt, kann verändert werden. Eine Möglichkeit ist die Verwendung der Blending-Funktion, die den bestehenden Farbwert mit dem Ausgabewert des Fragments nach gegebener Methode verrechnet. Es gibt zudem die Funktion *glLogicOp(GLenum opcode)*[19], mit der alternativ eine logische Operation zum Schreiben in den Framebuffer aktiviert werden kann. Der übergebene Parameter setzt die gewünschte logische Operation, die auf den ausgehenden RGBA-Wert und den RGBA-Wert an der betreffenden Stelle des Framebuffers angewendet werden soll. Der initiale Wert ist *GL_COPY*, wodurch der ausgehende Wert in den Framebuffer kopiert wird. Um eine logische ODER-Verknüpfung der Werte vorzunehmen, muss die Konstante *GL_OR* übergeben werden.

Wird die logische Operation mithilfe von *glEnable(GL_COLOR_LOGIC_OP)* aktiviert, so kann gleichzeitig kein Blending aktiv sein. Außerdem muss die Ausgabetextur des Framebuffer-Objekts ein ganzzahliges internes Format besitzen. Die Operation hat keinen Effekt auf Gleitkomma-Texturen.

Eine geeignete Textur muss also mit *glTexImage2D* oder *glTexStorage2D* (Vgl. Abschnitt 2.2.1) erstellt werden. Da die Textur ohne initiale Pixeldaten erstellt werden kann, würde ein Aufruf von *glTexStorage2D* mit dem Wert *GL_RGBA8* für den Parameter *internalFormat* das Texturobjekt hinreichend spezifizieren. Es wird dann eine Textur mit vier Kanälen und jeweils

[19] OpenGL Spezifikation: 17.3.11 Logical Operation

8 Bit pro Kanal erstellt, wobei die Werte als unformatierter Byte-Wert gespeichert werden.

Um die Bit-Maske als Textur bereitzustellen, werden die Masken bereits bei der Erstellung der 1D-Textur in die Texel geladen.

```
GLuint bitMaskHandle;

unsigned char bitMaskData[32][4] = {
{1,0,0,0},  {2, 0,0,0}, {4, 0,0,0}, {  8,0,0,0}, // R: 0..7
{16,0,0,0}, {32,0,0,0}, {64,0,0,0}, {128,0,0,0},
{0, 1,0,0}, {0, 2,0,0}, {0, 4,0,0}, {0,  8,0,0}, // G: 8..15
{0,16,0,0}, {0,32,0,0}, {0,64,0,0}, {0,128,0,0},
{0,0, 1,0}, {0,0, 2,0}, {0,0, 4,0}, {0,0,  8,0}, // B: 16..23
{0,0,16,0}, {0,0,32,0}, {0,0,64,0}, {0,0,128,0},
{0,0,0, 1}, {0,0,0, 2}, {0,0,0, 4}, {0,0,0,  8}, // A: 24..31
{0,0,0,16}, {0,0,0,32}, {0,0,0,64}, {0,0,0,128},
};

glGenTextures(1, &bitMaskHandle);
glBindTexture(GL_TEXTURE_1D, bitMaskHandle);
glTexImage1D( GL_TEXTURE_1D, 0, GL_RGBA, 32, 0, GL_RGBA,
    GL_UNSIGNED_BYTE, &bitMaskData);
```

Quellcode 6: Erstellung der 1D-Textur mit Bit-Masken

Quellcode 6 zeigt die Erstellung einer 1D-Textur, deren Texel zu der Bitmaske einer bestimmten Tiefe im Voxelgitter korrespondieren. Da Fragment-Shader in GLSL immer RGBA-Werte ausgeben, wurden die Bits fortlaufend auf die RGBA-Kanäle verteilt, wodurch der höchste Wert pro Kanal gleich $2^7 = 128$ ist und jeder Kanal ein Viertel der Gesamtanzahl von Schichten abdeckt.

```
#version 330
uniform mat4 uniformModel;
uniform mat4 uniformViewProjection;

layout (location = 0) in vec4 positionAttribute;

out float passDistanceToCam;

void main() {
    vec4 pos = uniformModel * positionAttribute;
         pos = uniformViewProjection * pos;

    // distance to cam in normalized device coordinates
    passDistanceToCam = ( ( pos ).z + 1.0 ) * 0.5;

    gl_Position = pos;
}
```

Quellcode 7: Vertex-Shader zur Voxelisierung in Slicemap

```
#version 330
```

```
2
3  in float passDistanceToCam;
4
5  uniform sampler1D uniformBitMask;
6
7  out vec4 bitValue;
8
9  void main() {
10     // bit mask lookup determines bit value
11     bitValue = texture( uniformBitMask, passDistanceToCam );
12 }
```

Quellcode 8: Fragment-Shader zur Voxelisierung in Slicemap

Quellcode 8 zeigt den Fragment-Shader, der zu einem Tiefenwert die passende Bitmaske ausliest. Der einzige übergebene Wert aus dem Vertex-Shader in den Fragment-Shader ist der interpolierte Tiefenwert des Fragments.

3.1.3 Grenzen

Die Tiefe des Voxelgitters ist beschränkt. Sie hängt davon ab, wie viele Bits für die RGBA-Werte der Textur zur Verfügung gestellt werden. Zum Zeitpunkt der Veröffentlichung dieses Verfahrens waren diese für logische Operationen noch auf 8 Bit pro Kanal begrenzt. Mittlerweile sind auch 32 Bit pro Kanal nutzbar, wodurch sich die Anzahl der Schichten pro Slicemap auf 128 erhöhen lässt. Dennoch kann die Auflösung in $x-$ und y-Richtung die der z-Dimension bei Weitem übersteigen.

Es gibt weiterhin noch die Möglichkeit, mehrere Texturen an den Framebuffer zu binden (*Multiple Render Targets*, kurz MRT). Auf diese Weise kann die Tiefenauflösung noch auf ein Vielfaches gesteigert werden. Abbildung 3 zeigt eine in der Auflösung 512x512 gerenderte Slicemap, die aus 4 als hintereinanderliegend interpretierten Teil-Slicemaps besteht. Auf dem Boden zeigt sich mit zunehmender Distanz die Aneinanderreihung der RGBA-Kanäle der Texturen. Jede der zugrundeliegenden RGBA-Texturen besitzt 8 Bit pro Farb-Kanal. Dadurch ergeben sich insgesamt $4*4*8 = 128$ Schichten und eine Voxelgitterauflösung von 512x512x128. Ein mehrwertiger Datentyp der Farbkanäle, etwa ein ganzzahliger vorzeichenfreier 32-Bit Datentyp würde entsprechend eine Tiefenauflösung von 512 Schichten erzielen.

Geometrien, die parallel zur Voxelisierungsrichtung liegen, werden nicht erfasst. Dadurch kann es zu größeren Löchern im Voxelgitter kommen oder zum Auslassen von kleinen Strukturen, die nicht voxelisiert werden, da kein Fragment für sie erzeugt wurde. Mögliche Lösungen dieses Problems werden in Abschnitt 6.1.1 angesprochen.

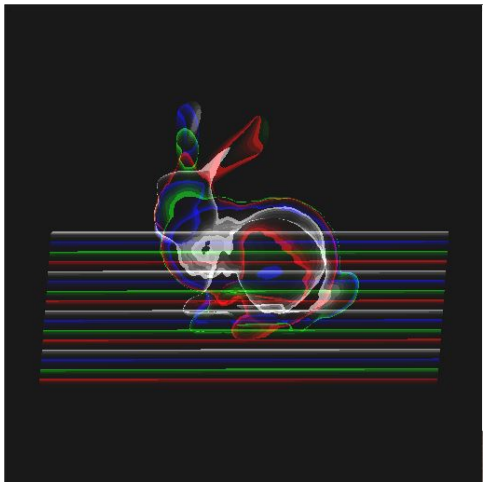


Abbildung 3: Mithilfe von MRT in mehreren Texturen kodierte Slicemap aus einer schrägen Sicht von oben auf eine Szene. Die Farbwerte der Texturen werden als Pixelfarbe verwendet.

3.2 Texturatlanten

Das nachfolgend erläuterte Verfahren wurde vorgestellt von [Thiedemann et al., 2011] und erweitert die in Abschnitt 3.1 erläuterte Strategie. Ein Objekt soll voxelisiert werden, indem eine gleichmäßige Abtastung der Geometrie erfolgt und die Abtastpunkte im Voxelgitter eingetragen werden. Auf diese Weise sollen Probleme mit zur Voxelisierungsrichtung parallelen Geometrien vermieden werden. Auch hier soll die Voxelisierung in einer texturbasierten Slicemap gespeichert werden. Die Abtastung erfolgt mithilfe eines Texturatlas, eine 2D-Repräsentation der Geometrieoberfläche.

3.2.1 Vorgehen

Es sind einige Vorverarbeitungsschritte nötig, bevor ein Texturatlas zum Füllen eines Voxelgitters verwendet werden kann. Texturatlanten haben ihren Ursprung in der Spiele-Entwicklung. Es handelt sich dabei um Texturen, deren verschiedene Bereiche bestimmten Teilen der Geometrie zugeordnet sind. In diesem Sinne können zugeordnete Texel als Abtastpunkte der Geometrie verstanden werden. Üblicherweise werden auf diese Weise Farb- oder Materialeigenschaften in Texturatlanten gespeichert, um sie während dem Rendern des Modells abzurufen. In diesem Verfahren soll in einem Texel die Weltposition eines Punktes der Geometrie-Oberfläche gespeichert werden. Mit dieser Information kann dann das Voxelgitter gefüllt werden.

Abbildung 4 zeigt einen Texturatlas, der für ein Modell generiert wurde. Jedes weiße Pixel des Texturatlas korrespondiert zu einem bestimmten

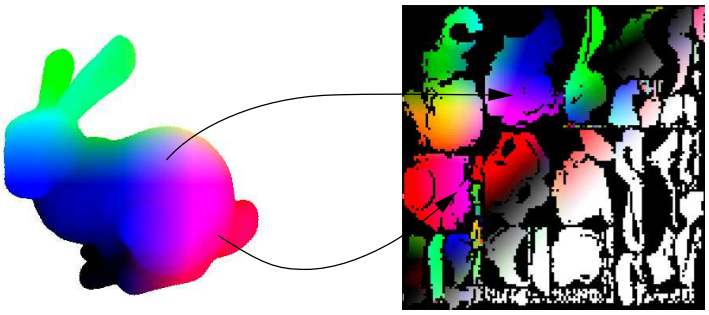
Punkt auf der Oberfläche des Modells. Ein schwarzes Pixel korrespondiert mit keinem Punkt auf der Oberfläche des Modells.

Abbildung 4: Ein Modell und der dafür generierte Texturatlas

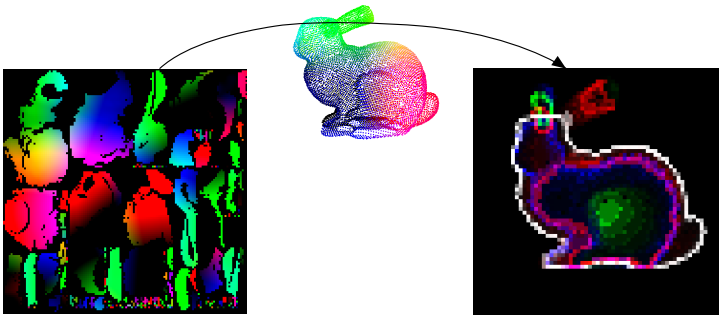
Texturatlanten besitzen häufig Bereiche, die keiner Geometrie zugeordnet sind. Diese Bereiche beinhalten somit keine relevante Information. Es sollen zunächst nur die relevanten, bzw. validen Texel erfasst werden. Für jedes valide Texel, das mit einem Punkt auf der Geometrieoberfläche korrespondiert, soll ein Vertex generiert werden. Jeder dieser Vertices wird bei der Voxelisierung mithilfe der im Texturatlas gespeicherten Weltposition in das Voxelgitter eingetragen. Ein Fragment-Shader füllt analog zu Abschnitt 3.1 das betroffene Voxel des Voxelgitters.

Zu beachten ist, dass das Modell ein eindeutiges *UV-Mapping* besitzen muss, damit jedes genutzte Texel genau einem Punkt der Geometrie zugeordnet ist. Es ist zudem darauf zu achten, dass der Texturatlas eine ausreichend große Auflösung besitzt, da jedes gültige Texel maximal ein Voxel füllt. Eine passende Auflösung ist dabei im Anwendungsfall zu bestimmen, da auch das Verhältnis der Modellgröße zur Voxelgröße beachtet werden muss. Es können sonst Löcher im Voxelgitter entstehen, da die Oberfläche der Geometrie unterabgetastet wird. Eine zu hohe Auflösung führt zu dem mehrmaligen Schreiben des gleichen Voxels. Außerdem besteht ein direkter Zusammenhang zwischen der Performanz und der Anzahl der gerenderten Punkte. Eine ideale Auflösung führt zu einer Eins-zu-Eins Abbildung eines Texels auf eine Voxelposition.

Abbildung 5 zeigt die Schritte von dem Aktualisieren des Texturatlas bis zur voxelisierten Szene. Der Texturatlas wird mit den Weltpositionen der Modelloberfläche beschrieben (Abbildung 5a). Als nächstes erfolgt die eigentliche Voxelisierung in das Voxelgitter. Dann wird für jeden validen Texel des Texturatlas ein Vertex gerendert, der die Weltposition aus dem Texturatlas verwendet (Abbildung 5b). Durch das Punkt-Rendering wird für jeden der korrespondierenden Oberflächenpunkte garantiert ein Fragment generiert.



(a) Weltpositionen in Texturatlas rendern



(b) Valide Texel mit Punkt-Rendering in Voxelgitter rendern

Abbildung 5: Die auszuführenden Schritte der Voxelisierung mithilfe eines Texturatlas nach [Thiedemann et al., 2011]

3.2.2 Details der Implementierung

Um für ein Modell ein eindeutiges UV-Mapping zu erstellen, wurde für komplexere Testobjekte die in der freien 3D-Modelliersoftware Blender enthaltene Funktion *Smart UV Project*[20] verwendet. Sie berechnet ein eindeutiges UV-Mapping des Modelles, während das Flächenverhältnis bewahrt werden kann.

Zunächst wird eine Textur und ein Framebuffer-Objekt in einer festgelegten Auflösung erstellt. Es muss bei der Erstellung der Textur darauf geachtet werden, dass das gewählte Texturformat[21] auch nicht-normalisierte Werte speichern kann. Dies ist nötig, da Weltkoordinaten üblicherweise Werte über 1.0 und unter 0.0 beinhalten können. Die Textur wird als Ausgabeziel an das Framebuffer-Objekt gebunden.

Um die validen Texel zu identifizieren, wird der Texturatlas als Framebuffer gebunden und mit $(0.0, 0.0, 0.0, 0.0)$ gesäubert. Es wird dann der Shader zum Beschreiben des Texturatlas mit dem zugehörigen Modell ein-

[20] `http://wiki.blender.org/index.php/Doc:2.6/Manual/Textures/Mapping/UV/Unwrapping/Smart_UV_Project` (Zugriff: 24.09.2014)

[21] OpenGL Spezifikation: 8.5.2 Encoding of Special Internal Formats

mal aufgerufen. Alle Texel, die mit einem Oberflächenpunkt korrespondieren, wurden mit Weltkoordinaten beschrieben. Dadurch wird für die betroffenen Texel auch die homogene Koordinate im Alpha-Kanal auf 1.0 gesetzt. Mithilfe der Funktion *glGetTexImage()*[22] wird der Inhalt der Textur angefordert. Für jedes Texel, dessen Alpha-Wert gleich 1.0 ist, wird die Texturkoordinate des Mittelpunktes des Texels in einer Liste gespeichert. Die Liste enthält nun alle validen Texel und wird in einem Vertex-Buffer-Objekt auf die Grafikkarte geladen.

Um den Texturatlas zu aktualisieren, beispielsweise wenn das Modell sich bewegt hat, wird der Texturatlas wie oben beschrieben als Framebuffer gebunden und das Modell gerendert. Der Vertex-Shader, der den Texturatlas mit den aktuellen Weltpositionen schreiben soll, erhält als Eingabe die Vertexposition und UV-Koordinate. Nachdem die Weltposition des Vertex berechnet wurde, wird sie dem Fragment-Shader in einer benutzerdefinierten Ausgabe-Variablen übergeben. Es soll ein Fragment genau an der Stelle des Texturatlas erzeugt werden, an der die Weltposition eingetragen werden soll. Es wird dazu die UV-Koordinate des Vertex als Fragment-Position in die vordefinierte Ausgabevariable *gl_Position* eingetragen. Vorher wird die UV-Koordinate auf das Interval von -1.0 bis 1.0 skaliert. Dies hat zur Folge, dass ein Fragment an der Stelle des Texturatlas erzeugt wird, mit dem der gerenderte Oberflächenpunkt korrespondiert. Der Fragment-Shader kann dann die Weltposition ohne weitere Berechnungen als Ausgabewert schreiben. Quellcode 9 und 10 zeigen die entsprechenden Shader-Programme.

```
1  layout (location = 0) in vec4 positionAttribute;
2  layout (location = 1) in vec2 uvCoordAttribute;
3
4  uniform mat4 uniformModel;
5
6  out vec3 passWorldPosition;
7
8  void main() {
9      passWorldPosition = (uniformModel * positionAttribute).xyz;
10     gl_Position = vec4((uvCoordAttribute * 2.0) - 1.0, 0.0, 1.0);
11 }
```

Quellcode 9: Vertex-Shader zum Füllen des Texturatlas

```
1  in vec3 passWorldPosition;
2
3  out vec4 worldPosition;
4
5  void main() {
6      worldPosition = vec4( passWorldPosition, 1.0 );
7  }
```

[22] OpenGL Spezifikation: 8.11 Texture Queries

Quellcode 10: Fragment-Shader zum Füllen des Texturatlas

Um den aktualisierten Texturatlas zum Füllen des Voxelgitters zu verwenden, wird das Vertex-Buffer-Objekt gebunden und gerendert, das während der Texelvalidierung generiert wurde. Während der gleiche Fragment-Shader wie in Quellcode 8 verwendet werden kann, muss der Vertex-Shader (Vgl. Quellcode 7, Zeile 10), wie in Quellcode 11 gezeigt, angepasst werden. Auf diese Weise wird die lokale Vertexposition als Texturkoordinate des Texturatlas verwendet, um die Weltposition auszulesen. Die lokale Vertexposition liegt, wie oben beschrieben, genau in der Mitte eines validen Texels des Texturatlas.

```
1    // read world position from texture atlas
2    vec4 pos = texture( uniformTexture, positionAttribute.xy );
```

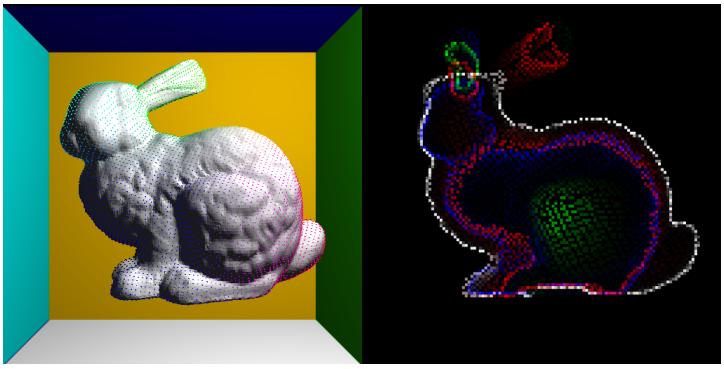
Quellcode 11: Anpassung des Vertex-Shaders zur Voxelisierung in Slicemap

3.2.3 Grenzen

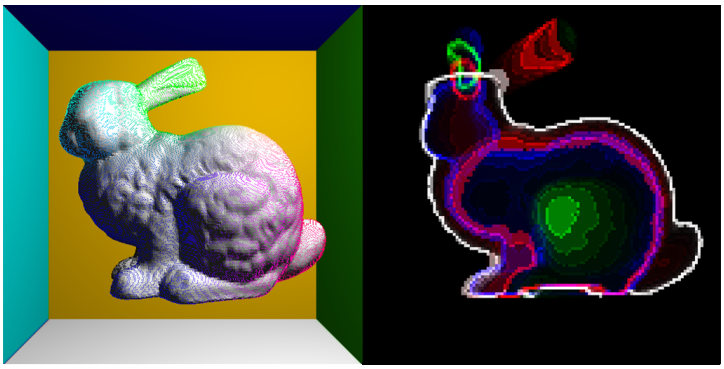
Die Qualität des Verfahrens ist stark von der Wahl der Auflösung des Texturatlas, der Qualität des UV-Mappings und der Auflösung des Voxelgitters abhängig. Oft muss eine geeignete Auflösung manuell gefunden werden, die eine zufriedenstellende Voxelisierung erzeugt. Gleichzeitig muss darauf geachtet werden, dass die mit einer hohen Auflösung des Texturatlas einhergehende steigende Anzahl von Texel-Vertices zusätzliche Rechenzeit verbraucht. Da im Gegensatz zum Verfahren aus Abschnitt 3.1 auch mehrere Fragmente im gleichen Pixel und der gleichen Schicht entstehen können, werden möglicherweise außerdem unnötige Schreibvorgänge vorgenommen. Eine zu niedrige Auflösung des Texturatlas wiederum führt zu einer lückenhaften Voxelisierung. Abbildung 6 zeigt die Auswirkungen verschiedener Auflösungen des Texturatlas auf die Voxelisierung. Die obere Darstellung zeigt, dass eine zu niedrige Auflösung des Texturatlas zu Lücken im Voxelgitter führen kann. Die untere Darstellung zeigt, dass eine hohe Auflösung die Voxelisierung im Vergleich kaum verbessert.

3.3 Voxelisieren mit Compute-Shadern

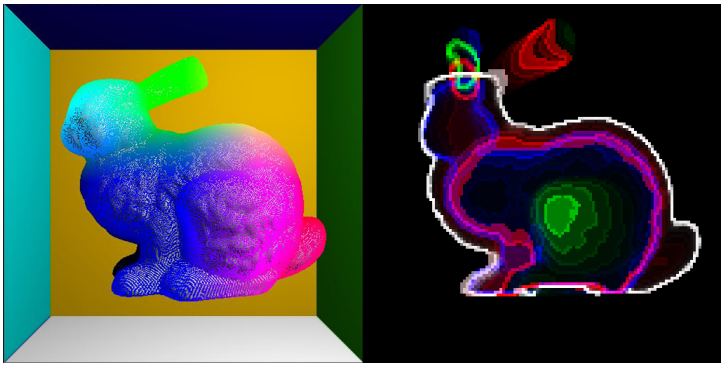
Dieses Verfahren kann in der Funktionsweise als selbstständiger Rasterisierer verstanden werden. Es zeichnet sich dadurch aus, dass es auf die Verwendung der Rasterisierungspipeline vollständig verzichtet. Alle Berechnungen werden durch GPGPU-Programmierung realisiert. Die hier vorgestellte Implementierung orientiert sich an dem Verfahren nach [Schwarz, 2012]. In genannter Veröffentlichung werden Verfahren vorgestellt, um mit Direct3D 11 konservative Oberflächenvoxelisierung auszuführen. Es wird



(a) 128x128 Pixel, 8684 generierte Vertices



(b) 384x384 Pixel, 78938 generierte Vertices



(c) 1024x1024 Pixel, 561570 generierte Vertices

Abbildung 6: Mit verschiedenen Auflösungen des Texturatlas ausgeführte Voxelisierung. Links werden die Vertices auf den korrespondierenden Oberflächenpunkten des Modells dargestellt.

eine erneuerte Lösung für das in Abschnitt 3.1 beschriebene Verfahren mithilfe der aktuellen Spezifikation von Direct3D vorgestellt. Weiterhin wird eine Lösung vorgestellt, um eine konservative Voxelisierung mithilfe der Direct3D-eigenen GPGPU-Programmierschnittstelle DirectCompute zu rea-

lisieren. Im Zuge dieser Arbeit wurde es für OpenGL auf Compute-Shader übertragen und implementiert.

3.3.1 Motivation

In der regulären Rasterisierungspipeline wird ein Pixel nur als gefüllt angesehen, wenn dessen Mittelpunkt überdeckt ist. Es werden folglich keine Fragmente produziert, wenn ein Pixel zwar von einem Dreieck geschnitten wird, der Mittelpunkt aber frei ist. Als Konsequenz werden bei rasterisierungsbasierten Voxelisierungsverfahren keine Voxel als belegt gesetzt. Es kommt in dem Voxelgitter zu Löchern durch Dreiecke, die parallel zur Projektionsrichtung liegen. Außerdem könnten kleine Strukturen unbeachtet bleiben, wenn sie keinen Pixelmittelpunkt verdecken. In Abschnitt 6.1.1 werden alternative Möglichkeiten genannt, um diese Fehler bei der Rasterisierung zu vermeiden.

Laut [Schwarz, 2012] wurde in Direct3D das Schreiben der Pixelwerte mithilfe des ODER-Operators bereits außer Kraft gesetzt. Der ursprüngliche Verwendungszweck dieser Operatoren war die mittlerweile veralteten Methode des *Color Indexing*.Auch in OpenGL ist diese Methode bereits zur Entfernung markiert[23]. In OpenGL 4.3 ist mit atomaren Funktionen aber eine aktuellere Option erschienen. Mit *Image Load Store* (Vergleich Abschnitt 2.2.1) besteht die Möglichkeit aus einem Shader Programm in eine beliebige Texturposition zu schreiben. Die atomare Operation *imageAtomicOr()* ermöglicht die bitweise logische ODER-Verknüpfung eines Texturwerts mit einem Eingabewert. Auch in Direct3D existiert eine äquivalente Funktion mit den gleichen Restriktionen bezüglich des 32-Bit Texturformats.

Es ist prinzipiell möglich, den in Abschnitt 3.1 aufgeführten Fragment-Shader für die Verwendung der neuen Funktionen anzupassen. Dann wird nicht mehr über den regulären Output des Fragment-Shaders in die Textur geschrieben, sondern über einen expliziten Aufruf an *imageAtomicOr()* mit den entsprechenden Parametern. Danach wird das Fragment mithilfe des *discard*-Kommandos verworfen. Allerdings ist dies aufgrund der geringeren Optimierung der zufälligen Texturzugriffe im Gegensatz zu der hardwarebeschleunigten Rasterisierungspipeline potentiell langsamer.

Andererseits ist man nicht mehr an die mit der Texturdarstellung einhergehenden Restriktionen gebunden.

3.3.2 Vorgehen

Ein Dreieck sei definiert durch die drei Vertices $\mathbf{v}_i$ mit $i \in 0, 1, 2$. Alle Dreiecke werden parallel verarbeitet, sodass ein Thread pro Dreieck aktiv ist. Der Compute-Shader berechnet eine Bounding Box um das Dreieck und

[23] `http://www.opengl.org/wiki/History_of_OpenGL#Deprecation_Model` (Zugriff: 29.09.2014)

schneidet dieses zunächst an dem Voxelgitter. Ist das resultierende Rechteck leer, so befindet es sich außerhalb des Voxelgitters und der Compute-Shader endet. Andernfalls wird über alle von dem Rechteck geschnittenen Voxel iteriert und überprüft, ob diese von dem Dreieck geschnitten werden. Es muss ein Überlappungstest ausgeführt werden und das Voxel bei Überlappung mithilfe einer atomaren ODER-Operation im Voxelgitter gesetzt werden.

Die Geometrieinformation für ein Dreieck wird dem Compute-Shader durch den Zugriff auf den Vertex-Buffer und Index-Buffer des zu verarbeitenden Objekts geliefert. Der Compute-Shader wendet dann selbst alle Transformationen an, um das Dreieck in das Voxelgitterkoordinatensystem zu überführen.

Es folgt die Berechnung der Bounding Box und die Auswahl der potentiell geschnittenen Voxel, die im Folgenden auch Überlappungskandidaten genannt werden. Es werden alle Voxel ausgewählt, die das Rechteck mindestens teilweise schneiden oder berühren, was einer konservativen Herangehensweise entspricht. Es wird dann für jedes dieser Voxel getestet, ob das Dreieck die Ausmaße des Voxels mindestens teilweise überlappt. Der ausgewählte Überlappungstest führt einer konservativen Oberflächenvoxelisierung (Vgl. Abschnitt 2.1.2).

3.3.3 Überlappungstest

Der von [Schwarz, 2012] angewendete Überlappungstest wurde weitestgehend von [Schwarz und Seidel, 2010] übernommen. Er basiert auf der von [Akenine-Möller, 2001] vorgestellten Routine, die dem Überlappungstest zwischen Dreieck und Quader dient. Der Test beruht auf dem als *Separating Axes Theorem* (abgekürzt *SAT*) bekannten Trennungssatz zwischen konvexen Objekten.

Dieser besagt, dass sich zwei Objekte nicht überlappen, wenn eine Achse existiert, für die sich die Intervalle der Projektion der zwei Objekte auf diese Achse nicht überschneiden. Eine Beschreibung zu verschiedenen Anwendungsfällen findet sich in der Veröffentlichung von [Eberly, 2001].

Ziel ist es, mithilfe von möglichst wenigen Achsenprojektionen alle Möglichkeiten abzudecken, um eine Überlappungsfreiheit ausschließen zu können. Die Routine besteht aus folgenden Teil-Tests. Zunächst erfolgt ein Test, ob sich die Bounding Boxes des Dreiecks und des Quaders überschneiden. Es folgt ein Test, ob die Ebene, die das Dreieck aufspannt, den Quader schneidet. Als letztes wird getestet, ob sich das Dreieck und der Quader in allen drei 2D-Projektionen der Koordinatenachsen überlagern. Sobald einer der Tests fehlschlägt, d.h. keine Überlappung festgestellt wird, wird der Test frühzeitig beendet. Im Unterschied zu dem in [Akenine-Möller, 2001] beschriebenen Ablauf wurden einige Vereinfachungen vorgenommen und sollen nachfolgend näher beschrieben werden.

Überlappungstest minimale umgebende Rechtecke

Der erste Überlappungstest wäre nach [Akenine-Möller, 2001] eigentlich ein Überlappungstest der Bounding Boxes. Dieser Test kann allerdings übersprungen werden, da die Wahl der Kandidatenvoxel die Überschneidung der Bounding Boxes voraussetzt.

Überlappungstest Ebene zu Voxel

Folglich ist der erste ausgeführte Überlappungstest der Test von Ebene zu Voxel. Für ein betrachtetes Voxel V mit den Voxel-Indizes (x, y, z) ist die Bounding Box im Voxelraum definiert durch die Koordinaten der Voxelecken $\mathbf{x} = [x, y, z]$ und $\mathbf{x'} = [x + 1, y + 1, z + 1]$. Es wird die Normale des Dreiecks $\mathbf{n}$ berechnet und die beiden diagonalen Voxelecken $\mathbf{c_1}$ und $\mathbf{c_2}$ werden bestimmt, die mit $\mathbf{n}$ am meisten übereinstimmen. Eine solche Ecke wird auch als *critical corner* bezeichnet.

$$c_{1,x} = \begin{cases} 1, & n_x > 0, \\ 0, & n_x \leq 0, \end{cases} \qquad c_{1,y} = \begin{cases} 1, & n_y > 0, \\ 0, & n_y \leq 0, \end{cases} \qquad c_{1,z} = \begin{cases} 1, & n_z > 0, \\ 0, & n_z \leq 0, \end{cases}$$

Dabei werden die Ecken relativ zu $\mathbf{x}$ definiert, womit sich für $\mathbf{c_2}$ ergibt:

$$c_{2,x} = 1 - c_{1,x}, \qquad c_{2,y} = 1 - c_{1,y}, \qquad c_{2,z} = 1 - c_{1,z}, .$$

Es wird dann $\mathbf{c_1}$ und $\mathbf{c_2}$ in die Ebenengleichung eingesetzt und die Vorzeichen der Ergebnisse werden verglichen. Es gilt Gleichung 2, wobei $d_k = \mathbf{n} * (\mathbf{c_k} - \mathbf{v_0})$.

$$(\mathbf{n} * (\mathbf{x} + \mathbf{c_1} - \mathbf{v_0}))(\mathbf{n} * (\mathbf{x} + \mathbf{c_2} - \mathbf{v_0})) = (\mathbf{n} * \mathbf{x} + d_1)(\mathbf{n} * \mathbf{x} + d_2) \leq 0 \quad (2)$$

Wenn beide Vorzeichen unterschiedlich sind, befinden sich die Ecken in unterschiedlichen Halbräumen der Ebene. Das bedeutet, dass die Ebene das Voxel schneidet. Sind die Vorzeichen gleich, liegen die Ecken im gleichen Halbraum und es besteht keine Überlappung. Ist eines der beiden Ergebnisse gleich null, so wird die Ebene berührt.

Überlappungstest 2D-Projektionen der Hauptachsen

Der letzte Überlappungstest nach jeweiliger Projektion in Richtung der Hauptachsen bedient sich der von [Pineda, 1988] vorgestellten Kantenfunktionen von Dreiecken. Diese beschreiben die Kanten eines Dreiecks mit der einfachen 2D-Geradengleichung

$$e_i(\mathbf{p}) = \mathbf{m}_i * (\mathbf{p} - \mathbf{w}_i), \quad (3)$$

wobei $i \in \{0, 1, 2\}$ den Index der Kante von $\mathbf{w}_i$ zu $\mathbf{w}_{(i+1) \bmod 3}$ bestimmt. $\mathbf{w}_i$ ist die Menge der 2D-Dreiecksvertices. $\mathbf{m}_i$ bezeichnet die Kantennormale, die in das Innere des Dreiecks zeigt. Für $\mathbf{m}_i$ gilt:

$$\mathbf{m}_i = [w_{(i+1) \bmod 3,y} - w_{i,y}, w_{i,x} - w_{(i+1) \bmod 3,x}],$$

wenn das Dreieck im Uhrzeigersinn definiert ist und

$$\mathbf{m}_i = [w_{i,y} - w_{(i+1) \bmod 3,y}, w_{(i+1) \bmod 3,x} - w_{i,x}],$$

wenn das Dreieck gegen den Uhrzeigersinn definiert ist. Nach einer ähnlichen Herangehensweise wie zur Bestimmung der Voxelecken $\mathbf{c}_k$ wird nun zu jeder Kantennormalen $\mathbf{m}_i$ die Ecke $\mathbf{f_i}$ des minimalen umgebenden Rechtecks gesucht, zu dem die Kantennormale zeigt. Für die jeweilige Ecke wird die Kantenfunktion ausgewertet. Nur wenn sich alle ausgewählten Ecken bezüglich ihrer Kantenfunktion im positiven Halbraum befinden, liegt eine Überschneidung vor.

3.3.4 Details der Implementierung

Der Zugriff auf ein Dreieck wurde mithilfe mehrerer SSBOs (Vgl. Abschnitt 2.2.3) realisiert. Es werden Index-Buffer und Vertex-Buffer eines Modells gebunden. Zu Beginn des Compute-Shaders werden drei aufeinanderfolgende Indizes aus dem Index-Buffer ausgelesen und zum Auslesen der Vertexinformationen aus dem Vertex-Buffer verwendet. Alle Berechnungen erfolgen in Voxelgitterkoordinaten. Daher werden die Vertexpositionen nach der Transformation durch die Modelmatrix in das Voxelgitterkoordinatensystem umgerechnet. Die Normale $\mathbf{n}$ wird dann aus dem Kreuzprodukt der zwei aufeinanderfolgenden Kanten $\mathbf{v}_1 - \mathbf{v}_0$ und $\mathbf{v}_2 - \mathbf{v}_1$ berechnet.

Es muss bestimmt werden, ob das Dreieck nach der Projektion in Richtung einer Hauptachse noch im oder gegen den Uhrzeigersinn definiert ist. Zu diesem Zweck wird das Vorzeichen des Skalars der Dreiecksnormalen betrachtet, das der Projektionsachse entspricht. Es wird davon ausgegangen, dass das Polygon gegen den Uhrzeigersinn definiert wird und die Normale aus dem Kreuzprodukt zweier aufeinanderfolgender Kanten berechnet wird. Ist das Vorzeichen positiv, so hat das Dreieck seine Orientierung beibehalten und ist gegen den Uhrzeigersinn definiert. Ist das Vorzeichen negativ, so hat sich die Orientierung umgekehrt, da in der Projektion die Rückseite des Dreiecks betrachtet wird.

Da die Anzahl der ausgeführten Compute-Shader von der Anzahl der Dreiecke des Modells abhängt und der Index-Buffer eine lineare Struktur besitzt, wurden die Größen der Arbeitsgruppen ebenfalls linear gewählt. Quellcode 12 zeigt die Wahl einer möglichen Arbeitsgruppengröße und die Berechnung der mit *glDispatchCompute()* zu entsendenden Arbeitsgruppen. Es müssen dabei die maximal zulässigen Werte beachtet werden. Dies ermöglicht einen wie in Quellcode 5 gezeigten Zugriff auf den Index-Buffer.

```
// local group size of compute shader
layout (local_size_x=1024, local_size_y=1, local_size_z=1) in;

// amount of local groups to dispatch
int numGroupsX = ceil(numTriangles / localSizeX);
int numGroupsY = 1;
int numGroupsZ = 1;
```

Quellcode 12: Festlegen der Anzahl der auszuführenden Comput-Shader

Es werden für das in diesem Abschnitt beschriebene Verfahren in der zugrundeliegenden Arbeit von [Schwarz, 2012] noch einige Vorschläge zur zusätzlichen Beschleunigung gemacht, die ebenfalls implementiert wurden.

- **Bounding Box überlappt nur eine Voxelspalte**. In diesem Fall werden auf jeden Fall alle Überlappungskandidaten in der Spalte (x-, y- oder z-Richtung) geschnitten und können ohne weitere Tests gefüllt werden.
- **Bounding Box überlappt nur eine Voxelschicht**. In diesem Fall muss für alle Überlappungskandidaten in der Schicht (parallel zur xy-, xz- oder yz-Ebene) nur noch der Überlappungstest in der 2D-Projektion zur korrespondierenden Hauptachse ausgeführt werden.

3.3.5 Grenzen

Die Nutzbarkeit des Verfahrens ist von dem Größenverhältnis der Dreiecke zum Voxelgitter abhängig. Da für jedes Dreieck ein zunächst unbekannter Aufwand entsteht, ist eine möglichst gleiche Beschaffenheit dieser zu empfehlen. Durch eine große Bounding Box des Dreiecks wird insbesondere bei einer geneigten Lage des Dreiecks im Voxelgitter eine große Menge von Überlappungskandidaten erzeugt. Bei Tests der Implementierung konnte es bei sehr großen Dreiecken sogar zu Treiber- oder Systemabstürzen kommen. In Abschnitt 4.2 wird der Einfluss von großen Dreiecken auf die Performanz gemessen.

3.4 Texturatlanten mit Compute-Shadern

Auch der texturatlasbasierte Ansatz wurde mithilfe von Compute-Shadern realisiert. Der ursprüngliche Ansatz soll für jedes gültige Texel einen Vertex-Shader ausführen, der jeweils einen Fragment-Shader ausführt. Mithilfe von Compute-Shadern können Vertex-Shader und Fragment-Shader zusammengefasst werden, indem ein Compute-Shader pro gültigem Vertex aufgerufen wird.

3.4.1 Vorgehen

Es wird der Vertex-Buffer des Texturatlas, der die validen Texel-Positionen enthält, als SSBO gebunden und ein Compute-Shader pro Vertex ausgeführt. Mithilfe der globalen Identifikationsnummer kann ein Vertex eindeutig angesprochen werden. Mit der x- und y-Koordinate des Vertex wird dann das passende Texel aus dem Texturatlas gelesen. Die gelesene Weltposition wird in Voxelgitter-Koordinaten transformiert und das entsprechende Bit mithilfe der Bitmaske und der atomaren ODER-Verknüpfung geschrieben. Das Vorgehen erfolgt bis auf das Füllen des Voxelgitters analog zu 3.2. Das Füllen des Voxelgitters erfolgt nicht über die Rasterisierungspipeline. Stattdessen wird ein Compute-Shader erstellt, der Zugriff auf den Vertex-Buffer und Texturatlas erhält. Jeder Compute-Shader liest für einen Vertex die im Texturatlas eingetragene Position und berechnet die Voxelgitterposition. Die korrespondierende Bitmaske wird dann in die Voxelgittertextur mithilfe einer atomaren logischen ODER-Verknüpfung eingetragen.

3.4.2 Details der Implementierung

Die Anzahl der auszuführenden Compute-Shader ist genau gleich der Anzahl der Vertices. Diese entspricht der Anzahl der validen Texel des Texturatlas. Um einen ähnlichen Zugriff auf das Buffer-Objekt zu benutzen wie in Quellcode 5 beschrieben, wurden die Größen wie in Quellcode 13 gezeigt bestimmt.

```
// local group size of compute shader
layout (local_size_x=1024, local_size_y=1, local_size_z=1) in;

// amount of local groups to dispatch
int numGroupsX = ceil(numVertices / localSizeX);
int numGroupsY = 1;
int numGroupsZ = 1;
```

Quellcode 13: Festlegen der Anzahl der auszuführenden Compute-Shader

```

struct Vert{float x; float y; float z;};

layout(std140, binding=0) buffer  vBuffer{Vert v[ ];} vertices;
layout(r32ui,  binding=0) uniform uimage2D            voxelGrid;
layout(r32ui,  binding=1) uniform readonly uimage1D   bBitmaskLUT;

uniform sampler2D texAtlas;
uniform mat4      worldToVoxelMatrix;

void main()
{
    // read vertex
    Vert vertex   = vertices.v[gl_GlobalInvocationID.x];
```

```
15
16      // read world position from texture atlas
17      vec4 pos      = texture(texAtlas, vec2(vertex.x, vertex.y));
18
19      // transform position to voxel indices in voxel grid
20      ivec3 voxelID = ivec3((worldToVoxelMatrix * pos).xyz);
21
22      // retrieve slice index
23      int slice     = voxelID.z;
24
25      // read bitmask corresponding to slice index
26      uint byte     = imageLoad(bitmaskLUT, slice).r;
27
28      // retrieve x / y coordinates of target texel
29      ivec2 writeTo = ivec2(voxelID.x, voxelID.y);
30
31      // OR with value currently written in voxel grid texture
32      uint before   = imageAtomicOr(voxelGrid, writeTo , byte);
33  }
```

Quellcode 14: Compute-Shader zur Texturatlas Voxelisierung

Quellcode 14 zeigt den implementierten Compute-Shader. In Zeile 20 werden die Voxelgitter-Koordinaten gelesen. An dieser Stelle wird im rasterisierungsbasierten Verfahren der Fragment-Shader erzeugt. Das Eintragen in das Voxelgitter erfolgt mithilfe einer atomaren logischen ODER-Verknüpfung auf dem aktuellen Texturwert.

3.5 Hierarchie-Aufbau mithilfe von Mip-Mapping

Bei der Verwendung von Voxelgittern zur Beschleunigung von Ray-Tracing hat es sich bewährt, eine hierarchische Repräsentation zu verwenden, um leere Bereiche schnell zu überspringen. Das folgende Verfahren basiert auf dem von [Thiedemann et al., 2011] beschriebenen Vorgehen, welches sich sich auf das von [Forest et al., 2009] vorgestellte Verfahren zur Generierung einer Octree-Hierarchie stützt. Es kann im Anschluss an das Füllen des Voxel-Gitters ausgeführt werden und ist auf eine texturbasierte Repräsentation des Voxelgitters ausgelegt. Zugrunde liegt die Vorgehensweise des *Mip-Mapping* einer Textur in immer kleinere, gröbere Repräsentationen derselben. Die Ergebnisstruktur aus mehreren Stufen, bzw. Level, wird auch als *MIP-Map* bezeichnet.

3.5.1 Vorgehen

Das auszuführende Mip-Mapping arbeitet von dem halbaufgelösten Texturlevel 1 bis zum höchsten Level n, welches die Auflösung von 1x1 Pixel besitzt. Zur Veranschaulichung sei dabei n gleich dem Exponenten von 2^n, welches die Auflösung der Voxelgittertextur in Höhe und Breite sei. In je-

dem Texturlevel k mit $1 \leq k \leq n$ ergibt sich eine Auflösung von 2^{n-k} in Höhe und Breite.

Der in Texturlevel k an der Stelle x,y einzutragende Pixelwert $\mathrm{p}_k(x, y)$ berechnet sich dann aus der Veroderung der vier umliegenden Pixelwerte des nächstniedrigeren, feineren Texturlevels $n - 1$:

$$\begin{aligned} \mathrm{p}_k(x,y) = \quad & \mathrm{p}_{k-1}(2x, \quad 2y) \\ | \quad & \mathrm{p}_{k-1}(2x+1, \quad 2y) \\ | \quad & \mathrm{p}_{k-1}(2x, \quad 2y+1) \\ | \quad & \mathrm{p}_{k-1}(2x+1, \quad 2y+1) \end{aligned} . \tag{4}$$

Für die Pixelkoordinaten x,y in Texturlevel k gelte dabei $0 \leq x, y < 2^{n-k}$.

Da das Mip-Mapping auf einer 2D-Textur ausgeführt wird, jeder Texel aber die Tiefeninformation einer ganzen Voxelspalte enthält, bleibt die Tiefenauflösung in jedem Texturlevel gleich. Eine Schicht des Voxelgitters wird folglich nur in x- und y- Richtung zusammengefasst. Der Aufbau dieser MIP-Map-Struktur ist daher nur bedingt mit der einer Octree-Struktur zu vergleichen. Um einen bestimmten Abschnitt in der Tiefe auf Belegtheit zu prüfen, muss zunächst eine passende Bitmaske berechnet werden. Mithilfe einer logischen UND-Operation kann dann die gleiche Funktionalität eines Octrees simuliert werden.

3.5.2 Details der Implementierung

In OpenGL wird die Texturerstellung mit *glTexStorage()* ausgeführt. Dadurch kann zu Beginn bereits die Anzahl der gewünschten Texturlevel festgelegt, allokiert und ein einheitliches Datenformat bestimmt werden. Es ist zu beachten, dass die Anzahl der Texturlevel die Basistextur einschließt. Die Anzahl der außerdem benötigten MIP-Map-Level wird aus der Auflösung der Basistextur berechnet.

Um sich bei der Wahl der Auflösung der Voxelgittertextur nicht nur auf Zweierpotenzen zu beschränken, wird die in Quellcode 15 gezeigte Berechnung angewendet.

```
1 int numMipMapLevels = ceil(log2(max(resX, resY)));
```

Quellcode 15: Berechnung der Anzahl der MIP-Map-Level

Das Verfahren wurde mithilfe eines Compute-Shaders realisiert. Es wird jeweils das Texturlevel k und das nächsthöher aufgelöste Texturlevel $k - 1$ an eine *image unit* gebunden. Für jedes Texturlevel von $k = 1$ bis n wird dieser Shader einmal aufgerufen, wobei eine Instanz pro zu berechnendem Texelwert aufgerufen wird. Dadurch ergeben sich für Texturlevel k die in Quellcode 16 gezeigten Größen der Arbeitsgruppen. Dabei enthalten die ganzzahligen Variablen **resX_k** und **resY_k** die Auflösung der Textur auf diesem Level.

```
// local group size of compute shader
layout (local_size_x=32, local_size_y=32, local_size_z=1) in;

// amount of local groups to dispatch
int numGroupsX = ceil(resX_k / localSizeX);
int numGroupsY = ceil(resY_k / localSizeY);
int numGroupsZ = 1;
```

Quellcode 16: Festlegen der Anzahl der auszuführenden Compute-Shader

Der Compute-Shader liest und verrechnet aus dem höheren Texturlevel die vier benachbarten Texelwerte nach dem in Gleichung 4 vorgegebenen Schema. Quellcode 17 zeigt den verwendeten Compute-Shader.

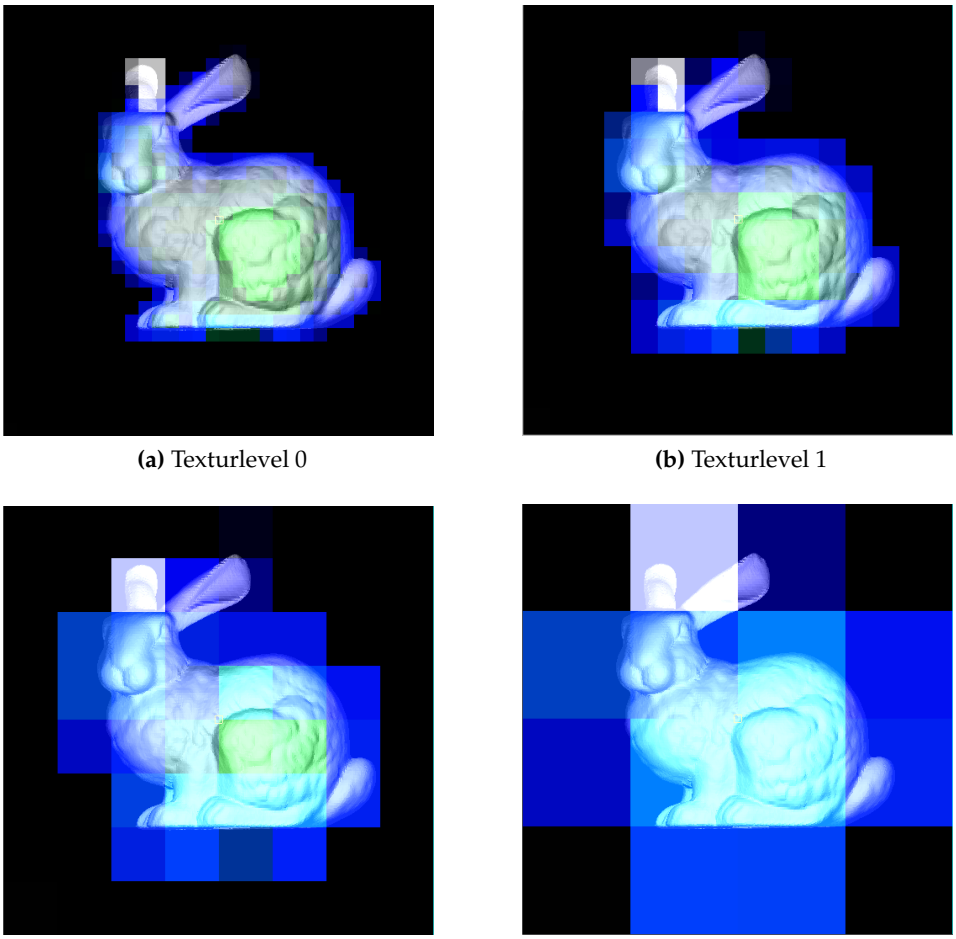
```
// specify image to mipmap
layout(binding = 0, r32ui) readonly uniform uimage2D mipmap_base;

// specify image to write to
layout(binding = 1, r32ui) writeonly uniform uimage2D
    mipmap_target;

void main()
{
    // use x & y index of global invocation as image coordinates
    ivec2 index = ivec2(gl_GlobalInvocationID.xy);

    // load values
    uvec4 valL  = imageLoad(mipmap_base, index * 2);
    uvec4 valR  = imageLoad(mipmap_base, index * 2 + ivec2(1, 0));
    uvec4 valTL = imageLoad(mipmap_base, index * 2 + ivec2(0, 1));
    uvec4 valTR = imageLoad(mipmap_base, index * 2 + ivec2(1, 1));

    // OR values
    uvec4 value = uvec4(0);
    value.r = valL.r | valR.r | valTL.r | valTR.r;

    // write value
    imageStore( mipmap_target, ivec2(index), value );
}
```

Quellcode 17: Compute-Shader zur Berechnung eines Texelwerts eines MIP-Map-Levels

Abbildung 7 zeigt die ersten vier durch den Compute-Shader erzeugten Texturlevel eines Voxelgitters. Da die Werte in der Tiefe nicht zusammengefügt werden, bleiben die beteiligten Farbkanäle erhalten. Die additive Darstellung führt zu einer Aufhellung des Bildes.

4 Vergleich

Anhand verschiedener Aspekte können die Voxelisierungsverfahren verglichen werden. Insbesondere die beanspruchte Zeit einer vollständigen



(a) Texturlevel 0

(b) Texturlevel 1

(c) Texturlevel 2

(d) Texturlevel 3

Abbildung 7: Die ersten vier Texturlevel des Voxelgitters

Voxelisierung der Szene kann als Indiz für die Eignung in einer interaktiven Anwendung dienen. Auch die Qualität der voxelisierten Szene ist bei den Verfahren nicht unbedingt gleich. Die Verfahren entscheiden auf verschiedene Weisen, ob ein Voxel gesetzt wird oder nicht. Die Verfahren sollen anhand einer Reihe von Modellen getestet werden, deren Beschaffenheiten übliche Eigenschaften von Modellen in interaktiven Anwendungen abdecken.

Das Testsystem ist wie folgt konfiguriert: Es besitzt eine Nvidia GTX 750Ti Grafikkarte, einen Intel Core i5 750 Prozessor, 8 GB RAM und das Betriebssystem Windows 7 64-bit. Der installierte Grafikkartentreiber ist die Version 344.11.

Die Compute-Shader werden mit der größtmöglichen lokalen Arbeitsgruppengröße von 1024 Compute-Shadern aufgerufen.

4.1 Testszenen

Die zum Vergleich verwendeten Testszenen umfassen immer ein Objekt im Zentrum des Voxelgitters. Die Dimensionen des Voxelgitters werden dabei der Auflösung angepasst, sodass es eine uniforme Struktur besitzt. Dies impliziert, dass das Frustum der Voxelisierungskamera in der Tiefe verkürzt wird, da die Auflösung in der Tiefe auf 32 Voxel begrenzt ist. Der Texturatlas wurde für jedes Testobjekt mit einer Auflösung von 512 Pixeln in Breite und Höhe gewählt.

Die zu voxelisierenden Testmodelle unterscheiden sich in der Anzahl der Dreiecke und aus dem Texturatlas generierten Vertices, in der Größe und in topologischen Eigenschaften.

- **Simples Quad**. Ein Viereck bestehend aus 2 Dreiecken, die der Voxelisierungskamera zugewandt sind. Es soll die Effizienz in Bezug auf große Dreiecke beeinflussen. Der Texturatlas besitzt keine Pixel ohne UV-Mapping, daher werden durch ihn $512 * 512 = 262144$ Vertices generiert. Die gemittelte Zeit zur Aktualisierung des Texturatlas beträgt 0.165 Millisekunden.

- **Komplexes Quad**. Ein Viereck bestehend aus 11858 Dreiecken. Es wurde aus dem simplen Quad erstellt, indem es mehrere Male unterteilt wurde. Es soll als Vergleich zum simplen Quad stehen, da es die gleiche Topologie aufweist, jedoch eine höhere Anzahl von Dreiecken besitzt. Auch dieses Modell besitzt ein lückenloses UV-Mapping, wodurch 262144 Vertices generiert werden. Die gemittelte Zeit zur Aktualisierung des Texturatlas beträgt 0.252 Millisekunden.

- **Achsenparalleler Würfel**. Ein Würfel bestehend aus 12 Dreiecken. Die Seiten sind achsenparallel zur Voxelisierungskamera ausgerichtet. Er soll verwendet werden, um die Robustheit der Verfahren gegenüber achsenparallelen Geometrien zu testen. Der Texturatlas generiert 174592 Vertices. Die gemittelte Zeit zur Aktualisierung des Texturatlas beträgt 0.144 Millisekunden.

- **Stanford Bunny**. Ein 3D-Scan einer Hasen-Figur aus dem *Stanford 3D Scanning Repository*[24]. Das Ausgangsmodell wurde mithilfe von Blender von toten Vertices, die zu keinen Dreiecken gehören, bereinigt und ein UV-Mapping generiert. Das Modell besitzt 69451 Dreiecke. Der Texturatlas generiert 140318 Vertices. Die gemittelte Zeit zur Aktualisierung des Texturatlas beträgt 0.274 Millisekunden.

- **Stanford Happy Buddha**. Ein 3D-Scan einer Buddha-Statue aus dem *Stanford 3D Scanning Repository*. Das Modell besitzt 1087716 Dreiecke

[24] `http://graphics.stanford.edu/data/3Dscanrep/` (Zugriff: 29.09.2014)

und ist damit das hochauflösendste Modell, das zum Vergleich verwendet wurde. Es besitzt zudem viele konkave und sehr detailreiche Stellen. Es wurde ebenfalls ein UV-Mapping mithilfe von Blender generiert. Der Texturatlas generiert 128058 Vertices. Die gemittelte Zeit zur Aktualisierung des Texturatlas beträgt 2.758 Millisekunden.

Abbildung 8 zeigt die Testmodelle aus Sicht der Voxelisierungskamera. Abbildung 8d zeigt den achsenparallelen Würfel zusätzlich aus einer schrägen Ansicht. Das Voxelgitter wird dabei durch die hellblaue Umrandung dargestellt. Um die Verkürzung des Voxelisierungsfrustums bei höheren Auflösungen auszugleichen, wird das Modell, wie in Abbildung 9 dargestellt, in der Tiefe gestaucht. Dadurch liegt es bei jeder Auflösung vollständig innerhalb des Voxelgitters und beansprucht die gleichen Bereiche des Voxelgitters.

4.2 Performanz

In OpenGL gibt es die Möglichkeit, mithilfe von sogenannten *Query Objects* die Zeit zu stoppen, die von einer Sequenz von OpenGL Kommandos beansprucht wird. Die genaue Zeit eines Rendervorgangs durch *glDrawElements()* oder der Ausführung eines Compute-Shader-Programms durch *glDispatchCompute()* kann dadurch ermittelt werden.

Tabelle 1 zeigt die gemessenen Voxelisierungszeiten.

Zusätzlich zeigt die erste Teiltabelle die benötigte Zeit des Compute-Shaders zum Löschen des Voxelgitters, wobei jeder Texturwert mit 0 überschrieben wird.

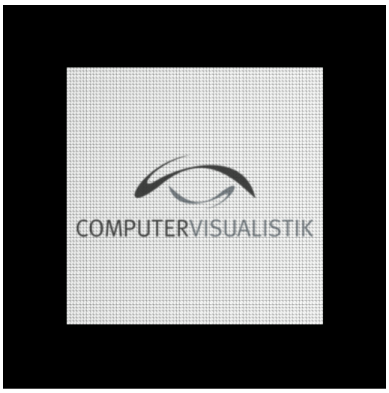
Die Zeiten der Voxelisierung entsprechen der Mittelung der Messungen der letzten einhundert Voxelisierungsprozesse. Es ist zu beachten, dass es vermutlich aufgrund wechselnder Leistungsphasen der Grafikkarte zu geringen Schwankungen von etwa einer Millisekunde kam. Dennoch ist ein grundsätzlicher Vergleich der gemessenen Zeiten möglich, da sich tendenzielle Verhalten bei bestimmen Modelleigenschaften bemerkbar machen.

Es zeigt sich deutlich, dass das Compute-Shader-Verfahren nicht für große Dreiecke ausgelegt ist, die viele Voxel überlappen. Der im Test aufgetretene Worst Case wurde durch das simple Quad bei hoher Voxelgitterauflösung hervorgerufen. Dabei verliert der Compute-Shader seinen Nutzen von Parallelisierbarkeit, da nur zwei Dreiecke bearbeitet werden und folglich nur zwei Compute-Shader-Instanzen aufgerufen werden. Bei hohen Voxelgitterauflösungen führt die steigende Anzahl von Überlappungskandidaten schließlich zu einer Bearbeitungszeit von über 160 Millisekunden, was etwa 6 Bildern pro Sekunde entspricht. Ist die Anzahl der Dreiecke und deren Größenverhältnis zu einem Voxel jedoch ausgewogen, wie bei dem komplexen Quad, so kann die Performanz sogar die des Rasterisierers übersteigen.

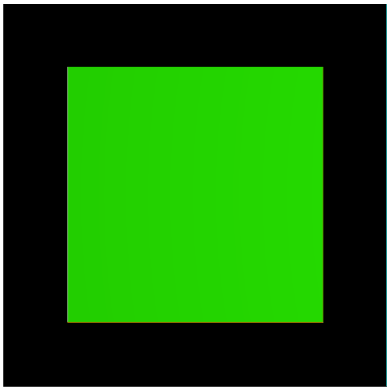
Leeren des Voxelgitters					
Auflösung	32	64	128	256	512
Löschvorgang	0.006	0.006	0.008	0.013	0.047
Simples Quad					
Methode	32	64	128	256	512
Slicemapping	0.164	0.224	0.177	0.165	0.177
Compute	0.677	3.695	10.166	40.614	160.533
Texturatlas	1.215	1.339	1.916	2.607	0.650
TA Compute	0.328	0.446	0.208	0.232	0.177
Komplexes Quad					
Methode	32	64	128	256	512
Slicemapping	0.185	0.190	0.194	0.364	0.198
Compute	0.070	0.077	0.088	0.976	0.410
Texturatlas	1.333	1.291	1.204	2.321	0.710
TA Compute	0.358	0.359	0.223	1.378	0.186
Achsenparalleler Würfel					
Methode	32	64	128	256	512
Slicemapping	0.160	0.166	0.169	0.173	0.183
Compute	2.282	5.165	14.010	48.338	84.09
Texturatlas	1.452	1.316	0.923	1.946	0.873
TA Compute	0.246	0.211	0.170	0.164	0.163
Stanford Bunny					
Methode	32	64	128	256	512
Slicemapping	0.370	0.368	0.812	0.850	0.729
Compute	0.357	0.456	1.237	2.000	1.484
Texturatlas	1.194	0.975	1.434	1.114	0.525
TA Compute	0.200	0.802	0.759	0.717	0.127
Stanford Happy Buddha					
Methode	32	64	128	256	512
Slicemapping	2.821	3.441	3.439	3.428	3.024
Compute	2.507	3.418	3.906	5.056	5.903
Texturatlas	0.725	0.838	0.700	0.497	0.476
TA Compute	0.133	0.153	0.135	0.120	0.098

Tabelle 1: Voxelisierungszeiten (in Millisekunden) der Testmodelle bei verschiedenen Voxelgitterauflösungen. Die Zahl im Spaltenkopf zeigt die Auflösung des Voxelgitters in x- und y-Richtung. Bei den Texturatlas-Methoden ist die Zeit für das Aktualisieren des Texturatlas zu beachten.

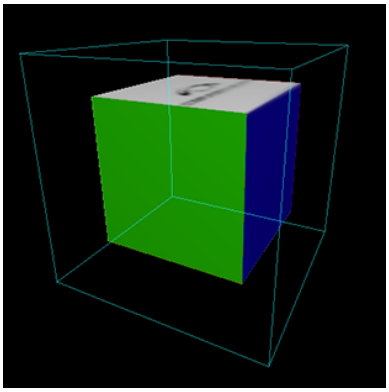
(a) Simples Quad



(b) Komplexes Quad



(c) Achsenparalleler Würfel



(d) Achsenparalleler Würfel, schräge Ansicht

(e) Stanford Bunny

(f) Stanford Happy Buddha

Abbildung 8: Testmodelle aus Sicht der Voxelisierungskamera im 32x32x32 Voxelgitter

Die Voxelisierung mithilfe des Texturatlas zeigt insbesondere bei den

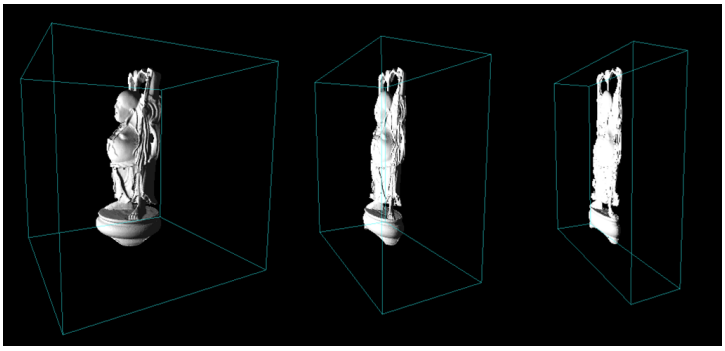


Abbildung 9: Verkürzung des Voxelisierungsfrustums und Stauchung des Objekts in der Tiefe

simplen und komplexen Quads den Nachteil einer im Verhältnis zum Voxelgitter zu hohen Anzahl von generierten Vertices. Jeder generierte Vertex wird in das Voxelgitter eingetragen, sodass bei einer geringen Voxelgitterauflösung viele überflüssige Schreibaufrufe erfolgen. Dementsprechend zeigt sich, dass das Rendern von 262144 Punkten langsamer ist als die beim Slicemapping erfolgende Rasterisierung von 2 Dreiecken.

Auch beim achsenparallelen Würfel zeigen sich die bisher beschriebenen Beobachtungen.

Eine weitere Auffälligkeit im Vergleich zwischen den beiden Voxelisierungsverfahren mithilfe des Texturatlas ist, dass die Compute-Shader-Variante um einen Faktor von bis zu 4 schneller zu sein scheint. Möglicherweise ist dies der übersprungenen Vertex-Shader-Phase zu verdanken. Es bleibt bei diesen beiden Verfahren dennoch zu beachten, dass zusätzlich die Aktualisierung des Texturatlas ausgeführt werden muss. Beispielsweise dauert der gesamte Vorgang mit der Compute-Shader-Variante der texturatlasbasierten Voxelisierung des Stanford Happy Buddha bei einer Voxelgitterauflösung von 512x512x32 mit Aktualisierung des Texturatlas $0,098ms + 2,758ms = 2,856ms$.

Das Stanford Happy Buddha Modell zeigt die fortbestehende Effizienz des Compute-Shader-Verfahrens bei einer großen Dreiecksanzahl von geringer Größe. Es zeigt sich durchweg, dass die Slicemapping-Methode für jedes Modell den vergleichbaren Aufwand eines gewöhnlichen Renderaufrufs erzeugt.

4.3 Qualität

Auch die Qualität der nach der Ausführung der Methoden jeweils erhaltenen Voxelgitter-Strukturen kann verglichen werden. Durch die teilweise abweichenden Entscheidungswege der Methoden, die zu füllenden Voxel zu bestimmen, können je nach Methode manche Voxel gesetzt werden, die

in anderen Methoden nicht gesetzt werden. Um zu vergleichen, welche Methoden die vollständigste Oberflächenvoxelisierung ausführen, werden die gefüllten Voxel nach der Ausführung der Methoden gezählt. Eine vollständige Oberflächenvoxelisierung ist als konservative Oberflächenvoxelisierung zu interpretieren.

Zu diesem Zweck wurde ein Compute-Shader implementiert, der jedes Texel der Voxelgittertextur liest und die gesetzten Voxel zählt und in einer *Atomic Counter*[25] Variable aufaddiert. Anschließend an die Auswertung wird die Zahl aus dem zugehörigen Buffer-Objekt gelesen, wobei vorher eine Speicherbarriere sicherstellt, dass die Schreibvorgänge abgeschlossen sind.

Tabelle 2 zeigt die Anzahl der gefüllten Voxel bei verschiedenen Voxelgitterauflösungen. Eine hohe Anzahl spricht dabei für eine vollständigere Oberflächenvoxelisierung. Es werden also mehr Voxel gefüllt, die die Oberfläche des Modells schneiden.

Die erste Teiltabelle zeigt die Gesamtanzahl der Voxel zu einer bestimmten Auflösung des Voxelgitters.

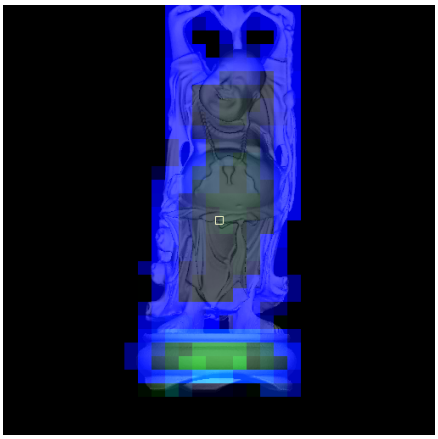
Es kann bereits festgestellt werden, dass das Compute-Shader-Verfahren die meisten gefüllten Voxel und damit die vollständigste Voxelisierung vorweisen kann. Unabhängig von Modell und Voxelgitterauflösung wurden stets alle geschnittenen Voxel gefüllt.

Das simple und komplexe Quad wurde von allen Verfahren nahezu identisch voxelisiert. Es gibt eine Abweichung bei den texturatlasbasierten Verfahren, welche möglicherweise auf Rundungsfehler bei der Fragment-Erstellung zurückzuführen sind.

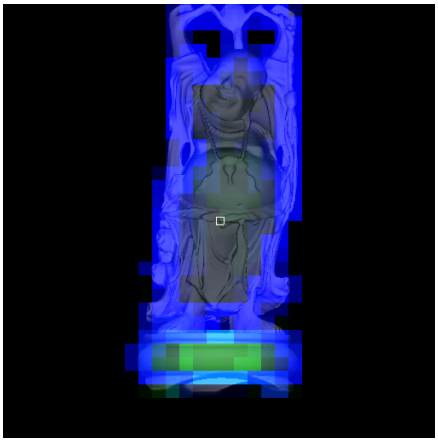
Bei dem achsenparallelen Würfel wird deutlich, dass die rasterisierungsbasierte Slicemapping-Methode Geometrien, die parallel zur Voxelisierungsrichtung liegen, nicht erfasst. Abbildung 11 zeigt, dass die Voxel nicht gefüllt werden, die nur von den zur Voxelisierungsrichtung parallelen Seiten des Würfels geschnitten werden. Zu beachten ist, dass die Seiten auch bei höheren Auflösungen des Voxelgitter nicht erfasst werden. Die Anzahl der übergangenen Voxel nimmt aber aufgrund der geringen Tiefenauflösung einen geringeren Anteil an der Gesamtanzahl. Alle anderen Verfahren konnten diese Seiten bis zu einer Auflösung von 256x256x32 vollständig erfassen. In der höchsten Auflösung des Voxelgitters besitzen die texturatlasbasierten Methoden geringere Füllzahlen. Die Anzahl der Vertices des Texturatlas reichen nicht mehr aus, um alle Voxel vollständig zu erfassen.

Bei beiden Stanford-Modellen ist die Anzahl der gefüllten Voxel bei der rasterisierungsbasierten Slicemapping-Methode bis zu einer Auflösung von 256x256x32 geringer. Auch hier zeigt sich, dass auf den im Winkel zur Voxelisierungsrichtung stärker abfallenden Seiten des Modells zunehmend Voxel übersprungen werden. In den texturatlasbasierten Verfahren werden

[25] OpenGL Spezifikation: 7.7 Atomic Counter Buffers



(a) Compute-Shader



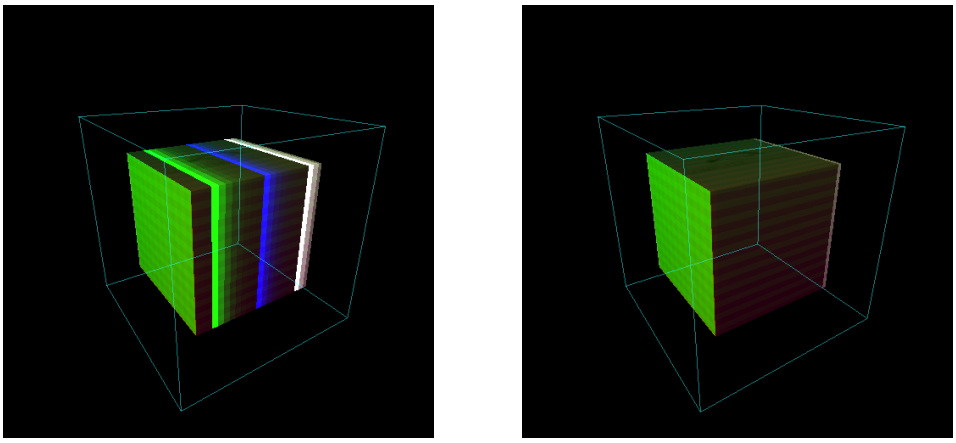
(b) Texturatlas

(c) Slicemapping

Abbildung 10: 32x32x32 Voxelgitter nach Ausführung verschiedener Voxelisierungsmethoden.

Gesamtanzahl Voxel					
Auflösung	32	64	128	256	512
Voxel gesamt	32768	131072	524288	2097152	8388608
Simples Quad					
Methode	32	64	128	256	512
Slicemapping	968	3528	14792	57800	233928
Compute	968	3872	14792	59168	233928
Texturatlas	968	3916	14878	59168	233928
TA Compute	968	3872	14792	59168	233928
Komplexes Quad					
Methode	32	64	128	256	512
Slicemapping	968	3528	14792	57800	233928
Compute	968	3872	14792	59168	233928
Texturatlas	968	3916	14878	59168	234270
TA Compute	968	3872	14792	59168	233928
Achsenparalleler Würfel					
Methode	32	64	128	256	512
Slicemapping	1452	5292	22188	86700	350892
Compute	3132	9248	28988	102432	378172
Texturatlas	3132	9248	28988	102388	127948
TA Compute	3132	9248	28988	102388	117082
Stanford Bunny					
Methode	32	64	128	256	512
Slicemapping	951	3817	15085	60227	241074
Compute	2088	6201	20250	71246	264889
Texturatlas	1991	5877	18943	65994	175250
TA Compute	1991	5875	18945	65994	175214
Stanford Happy Buddha					
Methode	32	64	128	256	512
Slicemapping	802	3112	12605	50282	201148
Compute	1731	5343	17154	60030	221840
Texturatlas	1776	5101	16181	55735	138497
TA Compute	1733	5104	16176	55735	138477

Tabelle 2: Anzahl der durch die Voxelisierungsmethoden gefüllten Voxel bei verschiedenen Voxelgitterauflösungen und Testmodellen. Die Zahl im Spaltenkopf zeigt die Auflösung des Voxelgitters in x und y-Richtung.



(a) Compute-Shader **(b)** Slicemapping

Abbildung 11: Seitliche Ansicht des 32x32x32 Voxelgitter nach Voxelisierung des achsenparallelen Würfels durch verschiedene Voxelisierungsverfahren.

sie im Gegensatz dazu jedoch meistens erfasst. Dies ist auf die hohe Verteilung der Texturatlas-Vertices auf der Oberfläche zurückzuführen. Auch hier reicht in der höchsten Voxelgitterauflösung die Anzahl der generierten Vertices jedoch nicht mehr aus, sodass größere Lücken entstehen.

4.4 Fazit

Alle Verfahren sind unter den passenden Umständen in einer Echtzeitanwendung verwendbar, da sich die Voxelisierung selbst bei komplexer Geometrie binnen weniger Millisekunden ausführen lässt. Daher sollte, ausgehend von der geplanten Nutzung des Voxelgitters und den Eigenschaften der Geometrien, die Auswahl des Verfahrens oder der Ausschluss von Verfahren erfolgen.

Die vollständigste Voxelisierung kann bei einer beliebigen Auflösung mit dem Compute-Shader-Verfahren erzielt werden. Für jedes Dreieck werden alle geschnittenen Voxel gefunden und gefüllt.

Kann für die Anwendung nicht garantiert werden, dass keine großen Dreiecke voxelisiert werden müssen, so ist bei steigender Voxelgitterauflösung die Voxelisierung mithilfe des Texturatlas eine Alternative. Das manuelle Finden einer geeigneten Texturatlasauflösung birgt dabei zusätzlichen Aufwand.

Steht die Vollständigkeit der Voxelisierung nicht an erster Stelle, sondern eine beständige Schnelligkeit der Voxelisierung, so ist ebenfalls der Texturatlas ein gutes Mittel der Wahl. Zwar steht die Geschwindigkeit dem rasterisierungsbasierten Slicemapping-Verfahren nach, die Qualität der Vo-

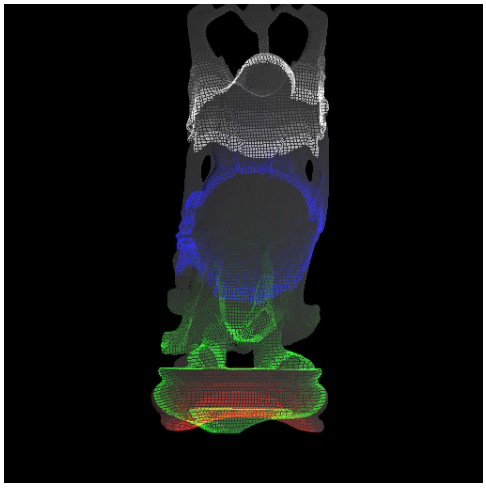


Abbildung 12: 512x512x32 Voxelgitter nach Voxelisierung des Stanford Happy Buddha durch ein Texturatlasbasiertes Verfahren. Das Objekt wurde zur besseren Ansicht etwas geneigt. Es entstehen Lücken zwischen den durch die generierten Vertices erfassten Oberflächenpunkten.

xelisierung ist jedoch beständiger.

Das rasterisierungsbasierte Slicemapping-Verfahren kann in der ursprünglichen Form, wie es hier implementiert wurde, nicht uneingeschränkt für vollständige Voxelisierungen empfohlen werden, wenn mit einer willkürlichen Zusammenstellung der Modelle gerechnet werden muss. Je mehr eine Geometrie in der Voxelisierungsrichtung liegt, desto größere Lücken können im Voxelgitter entstehen.

Für Anwendungsfälle, in denen eine vollständige Voxelisierung keine notwendige Voraussetzung ist, wie etwa in der in Abschnitt 5.1 beschriebenen Anwendung, bieten die anderen Verfahren keinen nennenswerten Mehrwert gegenüber dem rasterisierungsbasierten Slicemapping-Verfahren.

5 Beispielanwendungen

5.1 Transmittance Shadow Mapping

Mithilfe der in Abschnitt 3.1 beschriebenen Slicemap-Repräsentation eines Voxelgitters lassen sich lichtdurchlässige Materialien simulieren. Eine dazu geeignete Vorgehensweise wird als Anwendungsfall von [Eisemann und Décoret, 2006] beschrieben. Sie ergänzt das von [Williams, 1978] begründete Verfahren des *Shadow Mapping* und stützt sich auf die mit *Deep Shadow Maps* von [Lokovic und Veach, 2000] beschriebenen Konzepte zur Darstellung von teilweiser Lichtdurchlässigkeit. Um eine realistische Lichtabsorb-

tion von teilweise lichtdurchlässigen Materialien, wie etwa dichtem Blätterwerk, zu simulieren, werden entlang eines Lichtstrahls die durchdrungenen Schichten gezählt und ausgewertet.

5.1.1 Shadow Mapping

Zur Generierung einer Shadow Map wird die Szene aus der Sicht der Lichtquelle gerendert, wobei für jedes Pixel nur der Tiefenwert gespeichert wird. Eine gewöhnliche Shadow Map speichert in diesem Sinne die Distanz zum nächsten Verdecker in Richtung des Lichtstrahls. Punkte entlang dieses Lichtstrahls können durch den Vergleich des Tiefenwertes mit der gespeicherten Tiefe in den verschatteten oder beleuchteten Bereich eingeordnet werden. Auf diese Weise kann ein Pixel beim Rendern aus Sicht der Hauptkamera durch Rückprojektion in die Sicht der Lichtquelle auf Verschattung untersucht werden. Es wird dabei nur zwischen totaler Verschattung und direkter Beleuchtung unterschieden. Es ist nicht möglich, mit einer einzelnen Shadow Map den Effekt von teilweiser Lichtdurchlässigkeit durch mehrere Schichten, etwa durch getöntes Glas oder dichtes Blätterwerk, zu simulieren.

5.1.2 Erweiterung um schichtweise Lichtabsorption

Um diesen Effekt anzunähern, kann eine Slicemap ähnlich wie eine Deep Shadow Map verwendet werden. Während Deep Shadow Mapping als Beschleunigung für Offline-Rendering vorgestellt wurde, kann das Prinzip mithilfe einer Slicemap auch in Echtzeit angewandt werden. Ein Punkt soll in die Shadow Map projiziert werden und die Lichtintensität mithilfe seines Tiefenwertes bestimmt werden. Der Lichtstrahl verliert an Intensität, wenn er auf ein Hindernis stößt. Unter der Annahme, dass jedes Hindernis die gleiche Lichtdurchläßigkeit besitzt, kann die Lichtintensitätsfunktion in Abhängigkeit zu den durchdrungenen Schichten formuliert werden. Die Lichtintensitätsfunktion sei dann $(1 - \sigma)^n$, wobei n die Anzahl der durchdrungenen Schichten und σ der Lichtabsorptionsfaktor des Materials ist. Im Falle der gewöhnlichen Shadow Map wäre σ also 1, da das Licht nach dem ersten Hindernis seine gesamte Intensität verliert.

Es gilt also, zur Beleuchtung eines Punktes die Anzahl der zur Lichtquelle hin durchdrungenen Schichten zu errechnen. Zu diesem Zweck wird während der Shadow Map Generierung keine Tiefenkarte geschrieben, sondern eine Slicemap generiert. Bei der Auswertung werden die gesetzten Bits von dem Oberflächenpunkt bis zur Lichtquelle gezählt. Der Punkt wird wie gewöhnlich in die Shadow Map, bzw. Slicemap, projiziert. Es wird aus dem Tiefenwert die entsprechende Schicht i berechnet. Für alle Schichten $0 \leq j < i$ wird nun der eingetragene Wert mit der Bitmaske abgeglichen. Immer wenn ein gesetztes Bit gefunden wird, wird ein Zähler n inkremen-

tiert. Ist die Lichtquelle erreicht, wird n in die Lichtintensitätsfunktion eingesetzt und das Pixel entsprechend beleuchtet.

Quellcode 18 zeigt einen Auszug aus dem Fragment-Shader, der die Lichtstärke eines Punktes berechnet. In diesem Fall wird auf die Verwendung einer Lookup-Textur verzichtet und die Bitmaske jeder Schicht mithilfe einer Exponentialfunktion zur Basis 2 berechnet.

```
1  uint voxelCollumn = texture( voxelGrid, shadowmapCoords.xy ).r;
2
3  int startVoxel = int( shadowmapCoords.z * gridResolutionZ );
4  int intersectedSlices = 0;
5
6  for (int i = startVoxel; i >= 0; i--)
7  {
8      uint bitMask = uint( exp2( i ) );
9      if ( bitMask & voxelCollumn != 0)
10     {
11         intersectedSlices ++;
12     }
13 }
14
15 float lightIntensity = pow( ( 1.0 - sigma ), intersectedSlices );
```

Quellcode 18: Fragment-Shader des transmissive Shadow Mapping

5.1.3 Ergebnisse

Als Testszene wurde mithilfe der Modellierungssoftware Blender ein Baum erstellt, für dessen Blätterwerk teilweise Lichtdurchlässigkeit simuliert werden soll. Um zu verhindern, dass auch der Baumstamm als lichtdurchlässiges Material interpretiert wird, wird dieser in eine gewöhnliche Shadow Map gerendert. Das Blätterwerk wird dann mithilfe des in Abschnitt 3.1 beschriebenen rasterisierungsbasierten Slicemapping-Verfahrens in eine Slicemap gerendert, wobei als Voxelisierungskamera eine orthographische Projektion aus Sicht der Lichtquelle verwendet wird.

Das Modell des Blätterwerks besitzt 30960 Dreiecke. Der Baumstamm besitzt weitere 9840 Dreiecke, der Boden wird durch ein Quadrat mit 2 Dreiecken dargestellt. Die Slicemap und die Shadow Map besitzen eine Auflösung von jeweils 1024x1024 Pixel. Die Slicemap besitzt 32 Schichten, die gleich im Frustum der Voxelisierungskamera aufgeteilt werden.

Dies impliziert, dass nahe beieinander liegende Blätter die gleiche Schicht der Slicemap füllen könnten. Bei der Auswertung der Lichtintensität wird zunächst die gewöhnliche Shadow Map überprüft, um zu testen, ob das Pixel von dem Stamm verdeckt wird. Ist dies nicht der Fall, so wird die oben beschriebene Auswertung der Lichtintensitätsfunktion vorgenommen. Der Lichtabsorptionsfaktor σ wird dabei als uniforme Variable an den Fragment-Shader übergeben.

(a) deaktiviert **(b)** aktiviert

Abbildung 13: Bilder der Szene ohne und mit aktiviertem Transmittance Shadow Mapping mit $\sigma = 0.3$

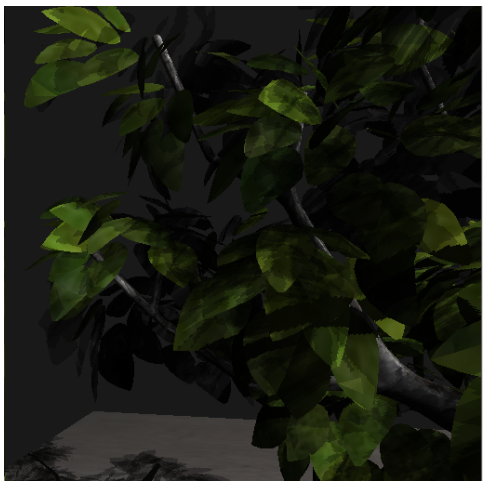


Abbildung 14: Detailaufnahme des Blätterwerks mit aktiviertem Transmittance Shadow Mapping mit $\sigma = 0.3$

Abbildung 13 zeigt den erzielten Effekt. Auf dem Boden und im Blätterwerk zeigt sich deutlich die unterschiedliche Lichtintensität. Abbildung 14 zeigt eine Detailaufnahme des Blätterweks. Es ist erkennbar, dass tiefere Stellen dunkler sind als äußere Stellen, die von wenigen Blättern verdeckt werden.

5.2 Erweitertes Reflective Shadow Mapping

Reflective Shadow Mapping nach [Dachsbacher und Stamminger, 2005] ist eine Lösung zur Annäherung von indirekter Beleuchtung in interaktiver Umgebung. Das ursprüngliche Verfahren des Shadow Mapping nach [Williams, 1978] bestimmt die für eine Lichtquelle sichtbaren Bereiche durch das Rendern einer Tiefenkarte aus Sicht der Lichtquelle. Es wird die Annahme gestellt, dass nur von Licht angestrahlte Oberflächen auch indirektes Licht abgeben können. Oberflächen werden als diffuse Reflektoren interpretiert. Jede für die Lichtquelle sichtbare Position wird selbst als kleine Lichtquelle interpretiert, die ihre nahe Umgebung beleuchtet. Dabei wird bei der Berechnung des indirekten Lichteinfalls für einen Oberflächenpunkt allerdings nicht mehr überprüft, ob eine indirekte Lichtquelle verdeckt ist. Mithilfe einer Voxelrepräsentation der Szene kann das Verfahren durch eine Strahlenverfolgung um diesen Test erweitert werden.

5.2.1 Reflective Shadow Mapping

Zur Generierung einer Reflective Shadow Map wird die Szene aus Sicht der Lichtquelle gerendert. Eine Reflective Shadow Map speichert für jedes Pixel zusätzlich zur Tiefe d_p die Weltposition x_p, die Normale n_p, und den reflektierten Strahlungsfluss Φ_p des beleuchteten Oberflächenpunktes p.

Berechnung des indirekten Lichts

Der Strahlungsfluss (engl. *radiant flux*) Φ_p beschreibt die Lichtmenge, die von der Oberfläche diffus in den Raum zurückgeworfen wird. Dazu wird zunächst berechnet, welche Lichtmenge die Lichtquelle durch das betreffende Pixel abgegeben hat. Dieser Wert ist im Falle parallel einfallenden Lichts konstant, im Falle einer Spot-Lichtquelle nimmt er aufgrund des abnehmenden Raumwinkels mit dem Cosinus zur Leuchtrichtung ab. Die dann von der Oberfläche tatsächlich reflektierte Lichtmenge ergibt sich aus der Multiplikation mit dem Reflexionsfaktor des Materials. Abbildung 15 zeigt den Flux-Buffer einer Spot-Lichtquelle. Der Reflexionsfaktor wird dabei und in den folgenden Abschnitten zur Vereinfachung für alle Materialen auf 1.0 gesetzt.

Die Strahlungsstärke eines Pixellichts in einen bestimmten Raumwinkel ω berechnet sich mithilfe der Gleichung $I_p(\omega) = \Phi_p \max\{0, \langle n_p|\omega\rangle\}$. Folglich kann die Beleuchtungsdichte an einem Oberflächenpunkt x mit der Normalen n ausgehend von dem Pixellicht p berechnet werden mit

$$E_p(x,n) = \Phi_p \frac{\max\{0, \langle n_p|x - x_p\rangle\} \max\{0, \langle n|x_p - x\rangle\}}{\|x - x_p\|^4}. \tag{5}$$

Die indirekte Strahlungsintensität am Oberflächenpunkt x mit der Oberflächennormalen n berechnet sich dann aus der Summe der Beleuchtung

Abbildung 15: Flux-Buffer der Reflective Shadow Map einer Spot-Lichtquelle

durch alle existierenden Pixellichter p:

$$E(x,n) = \sum_{\text{pixels } p} E_p(x,n). \tag{6}$$

Die Menge aller Pixel ist in der Regel zu groß, um diese Gleichung in Echtzeit effizient auszuwerten. Stattdessen soll mithilfe einer gewichteten Stichprobe von einigen hundert Pixeln eine Lösung angenähert werden. Die Wahl der Stichprobe erfolgt nach der Annahme, dass einflussreiche indirekte Lichtquellen nah an dem beleuchteten Oberflächenpunkt liegen müssen. Nach der Projektion eines Oberflächenpunktes in die Shadow Map kann davon ausgegangen werden, dass auch die einflussreichen indirekten Lichtquellen in der Projektion in der Nähe liegen.

Sampling Pattern Generierung

Zunächst wird der Oberflächenpunkt x in die Shadow Map projiziert, sodass die Texturkoordinaten (s,t) bekannt sind. Es werden dann zufällig Stichproben aus der Umgebung von (s,t) gezogen, wobei die Stichprobendichte mit der quadratischen Distanz zu (s,t) abnimmt. Dies wird erreicht, indem die Stichproben in Polarkoordinaten relativ zu (s,t) gezogen werden. Wenn ξ_1 und ξ_2 zwei gleichverteilte Zufallszahlen sind, wird eine Stichprobe an der Position:

$$(s + r_{\max}\xi_1 \sin(2\pi\xi_2), t + r_{\max}\xi_1 \cos(2\pi\xi_2)) \tag{7}$$

gezogen. Dabei ist $r_{\max}$ der maximale Versatz zu (s,t). Die Gewichtung zum Ausgleich der abnehmenden Stichprobendichte erfolgt dann mit ξ_1^2 und einer abschließenden Normalisierung. Dieses Stichprobenmuster (engl.

sampling pattern) kann einmal berechnet werden und für jede Berechnung des indirekten Lichts wiederverwendet werden. Dadurch wird Rauschen verhindert, aber es können Streifen sichtbar werden, wenn die Stichprobengröße zu klein ist.

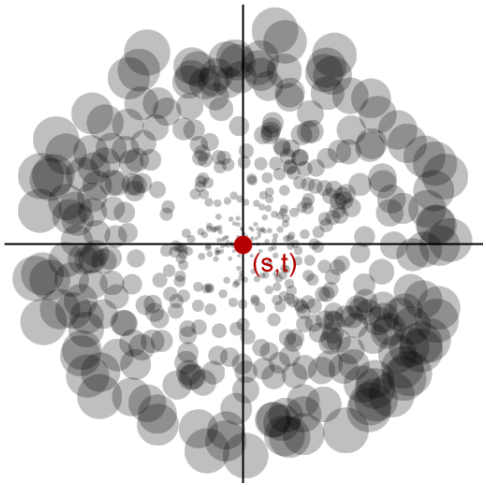
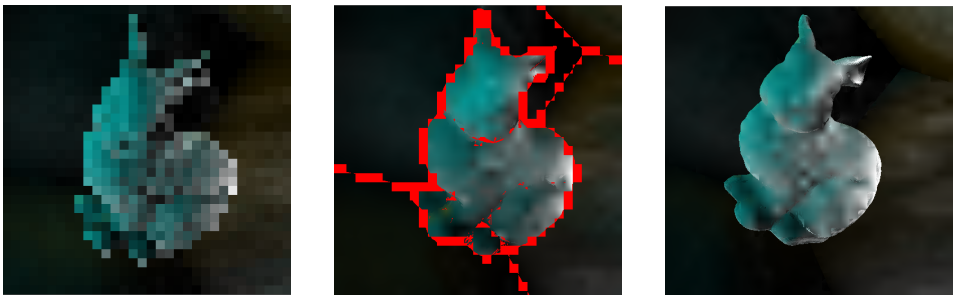


Abbildung 16: Das generierte Sampling Pattern aus [Dachsbacher und Stamminger, 2005]

Screen-Space Interpolation

Um den Rechenaufwand weiter zu verringern, kann ein Interpolationsverfahren angewendet werden, das die Berechnung des indirekten Lichts auf mehrere Renderpasses aufteilt. Dazu wird die Berechnung des indirekten Lichts zunächst auf ein Bild der Kamerasicht in geringer Auflösung angewendet. Es wird dann ein voll aufgelöstes Bild der Kamerasicht gerendert und für jedes Pixel überprüft, ob sich das indirekte Licht aus dem niedrig aufgelösten Bild annähern lässt. Wenn die beteiligten, niedrig aufgelösten Pixel eine ähnliche Normale und Weltposition besitzen, kann interpoliert werden. Sonst wird das Pixel in diesem Renderpass verworfen und das indirekte Licht in einem letzten Renderpass mit dem vollständigen Verfahren errechnet. Damit die Interpolation stattfinden kann, müssen drei oder vier umliegende Texel geeignet sein.

Abbildung 17 zeigt einen beispielhaften Verlauf des Interpolationsverfahrens. Zunächst wird in Abbildung 17a die Berechnung des indirekten Lichts in geringer Auflösung (32x32) vorgenommen. Abbildung 17b zeigt das Ergebnis der Interpolation der Werte in voller Auflösung (512x512). Sind die umliegenden Werte nicht geeignet, um eine plausible Interpolation durchzuführen, werden sie verworfen. Zur Darstellung werden diese Pixel hier rot dargestellt. Dies ist insbesondere an den Kanten der Wände



(a) indirektes Licht in niedriger Auflösung

(b) Interpolation in voller Auflösung. Rote Pixel wurden verworfen.

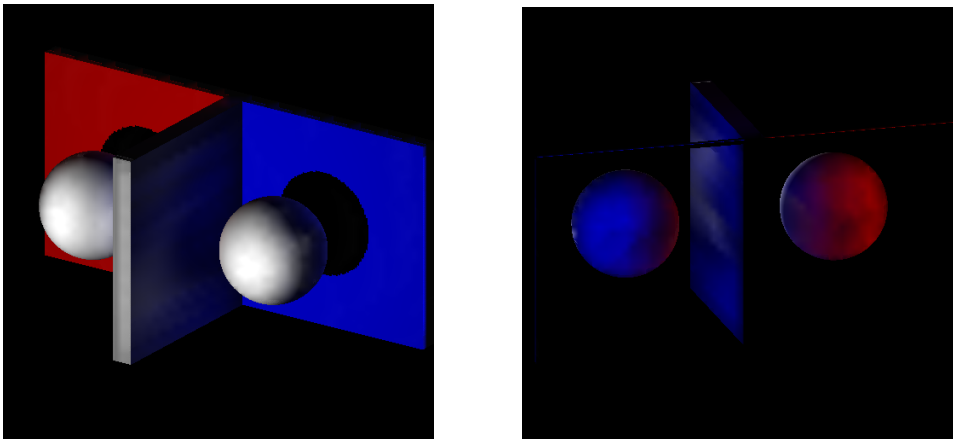
(c) indirektes Licht der verbleibenden Pixel in voller Auflösung

Abbildung 17: Verflauf des Screen-Space Interpolationsverfahren

und an den Umrissen des Modells der Fall. In Abbildung 17c erfolgt die Berechnung des indirekten Lichts für die verbliebenen Pixel.

5.2.2 Test auf Verdeckung

Während der Erfassung der indirekten Lichtquellen erfolgt die Berechnung des Einflusses aufgrund der Ausrichtung und Distanz zum betrachteten Pixel. Dabei erfolgt kein Test auf Verdeckung des Pixels und der Lichtquelle zueinander. Deshalb kann eine indirekte Lichtquelle der Reflective Shadow Map auch Einfluss haben, wenn tatsächlich ein Hindernis zwischen Oberfläche und Lichtquelle das Licht blockieren würde.



(a) Frontansicht

(b) Rückansicht

Abbildung 18: Reflective Shadow Mapping mit Okklusions-Testmodell

Abbildung 18 zeigt Reflective Shadow Mapping mit einer Testszene, in

der dieser Fall eintritt. Die Szene (Abbildung 18a) besteht aus zwei weißen Kugeln, die sich vor einer roten und einer blauen Fläche befinden. Zwischen den Kugeln befindet sich eine graue Trennwand. In Abbildung 18b wird das Modell schräg von hinten betrachtet, wobei die farbigen Flächen nicht dargestellt werden. Die Lichtquelle ist direkt auf die Flächen gerichtet, wodurch die Kugeln indirekt von ihnen beleuchtet werden. Rotes Licht von der Fläche erreicht die Kugel auf der anderen Seite der Trennwand. Analog erfolgt dies auch für das blaue Licht.

Mithilfe eines mit der Szene gefüllten Voxelgitters lässt sich der Algorithmus um einen Test auf Verdeckung erweitern. Während das indirekte Licht gesammelt wird, ist für jede Stichprobe Anfangspunkt und Endpunkt in Weltkoordinaten bekannt. Es soll mithilfe einer Strahlenverfolgung überprüft werden, ob sich ein belegtes Volumenelement zwischen Anfang und Ende befindet.

5.2.3 3D-DDA Traversierung

Ein einfacher und schneller Traversierungsalgorithmus durch ein Voxelgitter ist die in [Amanatides und Woo, 1987] beschriebene Variante eines *Digital Differential Analyzer* (abgekürzt DDA) Algorithmus. Der vorgestellte Algorithmus traversiert ausgehend von einem Startvoxel das Voxelgitter in Strahlrichtung, ohne Voxel zu überspringen. Dazu wird der Strahl in Intervalle eingeteilt, wovon jedes Intervall die Spannbreite eines Voxels umfasst. Der Algorithmus arbeitet in zwei Phasen: Initialisierung und inkrementelles Traversieren des Strahls. Der Strahl sei definiert durch eine Geradengleichung in Paramterdarstellung

$$\mathbf{u} + t\mathbf{v} \tag{8}$$

mit $t \geq 0$.

Initialisierung

In der Initialisierungsphase wird zunächst das Voxel V bestimmt, in dem der Strahl in $\mathbf{u}$ seinen Ursprung hat. Die ganzzahligen Werte **x**,**y** und **z** werden mit den Indizes des Startvoxels initialisiert.

Die Variablen **stepX**, **stepY** und **stepZ** werden entsprechend der Strahlrichtung mit 1 oder -1 initialisiert. Sie zeigen an, ob der betreffende Voxelindex erhöht oder verringert werden muss, wenn der Strahl eine entsprechende Voxelgittergrenze überschreitet. Sie werden anhand der Vorzeichen der x-, y- und z-Komponenten der Strahlrichtung $\mathbf{v}$ gesetzt.

Die Variablen **tMaxX**, **tMaxY** und **tMaxZ** werden entsprechend der Werte für den Paramter t gesetzt, an denen der Strahl die entsprechende Voxelgittergrenze überschreitet. **tMaxX** enthält also den Wert für t, an dem

der Strahl das erste Mal eine vertikale Grenze parallel zur yz-Ebene überschreitet. **tMaxY** und **tMaxZ** erhalten die entsprechenden Werte für eine Überschreitung einer Grenze parallel zur xz- und xy-Ebene. Der minimale dieser Werte gibt an, wie weit der Strahl traversiert werden kann, ohne das aktuelle Voxel zu verlassen.

Zuletzt werden die Variablen **tDeltaX**, **tDeltaY** und **tDeltaZ** berechnet. **tDeltaX** gibt an, welchem Intervall von t die Erhöhung der $x-$ Komponente um die Breite eines Voxels entspricht. Das heißt, wie weit kann der Strahl traversiert werden, bis auf der x-Achse die Breite von genau einem Voxel traversiert wurde. Dieser Wert berechnet sich aus dem Verhältnis der Breite eines Voxels zu dem Wert der x-Komponente des Richtungsvektors **v**. Die Werte von **tDeltaY** und **tDeltaZ** werden analog zur y- und z-Achse berechnet.

Traversierung

Die fortlaufende inkrementelle Traversierung des Strahls durch das Voxelgitter geschieht mithilfe einer einfachen Schleife, wie in Quellcode 19 dargestellt.

```
while(shouldTraverse){
    if( tMaxX < tMaxY ){
        if( tMaxX < tMaxZ ){
            x = x + stepX;
            tMaxX = tMaxX + tDeltaX;
        }else{
            z = z + stepZ;
            tMaxZ = tMaxZ + tDeltaZ;
        }
    }else{
        if( tMaxY < tMaxZ ){
            y = y + stepY;
            tMaxY = tMaxY + tDeltaY;
        }else{
            z = z + stepZ;
            tMaxZ = tMaxZ + tDeltaZ;
        }
    }
}
```

Quellcode 19: 3D-DDA Traversierung

Die Schleifenbedingung kann dabei selbst bestimmt werden, etwa dass **x**, **y** und **z** gültige Voxelindizes im Voxelgitter sind. Mit jedem Schleifendurchlauf wird der nächste Wert für t bestimmt, an dem eine Voxelgitter-Grenze überschritten wird. Der aktuelle Voxelindex wird entsprechend der überschrittenen Grenze um einen Schritt erhöht bzw. erniedrigt. Der betreffende Wert für **tMax** wird um den entsprechenden Wert von **tDelta** auf die

nächste Grenze gesetzt. Auf diese Weise enthalten die Variablen **x**, **y** und **z** immer die Indizes des nächsten Voxels entlang des Strahls.

5.2.4 Hierarchische Traversierung

Da es viele Bereiche geben kann, in denen leere Voxel traversiert werden müssen, kann eine hierarchische Traversierung möglicherweise schneller sein. Basierend auf einer MIP-Map-Struktur des Voxelgitters, wie in Abschnitt 3.5 beschriebenen, wird von [Thiedemann et al., 2011] eine hierarchische Traversierungsmethode beschrieben, die dem Schnittest eines Strahls mit einem Voxel dient.

Der Algorithmus arbeitet auf dem normalisierten Koordinatensystem zwischen 0 und 1 auf allen Koordinatenachsen, wobei 1 der äußersten Kante des Voxelgitters auf dieser Achse entspricht. Zunächst wird der Startpunkt des Strahls auf die Voxelgittertextur projiziert und das betreffende Texel auf dem aktuellen Texturlevel eingelesen. Es wird das minimale umgebende Rechteck des Texels berechnet, welches entsprechend dem MIP-Map-Level mehrere Voxelspalten umfasst und sich in der Tiefe von 0 bis 1 erstreckt. Es wird dann der Schnittpunkt mit dem Strahl berechnet. Da der Strahl innerhalb der Bounding Box startet, muss es sich dabei um einen Schnittpunkt handeln, an dem der Strahl die Bounding Box verlässt.

Um zu testen, ob sich auf dem Strahlabschnitt innerhalb der Bounding Box ein gesetztes Voxel befindet, kann eine Bitmaske verwendet werden. Die Bitmaske muss alle Schichten des Voxelgitters abdecken, die der Strahlabschnitt schneidet. Diese adaptive Bitmaske wird mit dem Texelwert der MIP-Map-Textur durch eine logische UND-Operation verglichen. Ist das Ergebnis ungleich 0, so besteht eine Überschneidung mit einem gesetzten Voxel.

Wird keine Überschneidung festgestellt, wird der Strahlursprung an den Schnittpunkt gesetzt und das aktuelle Texturlevel auf das Nächstgröbere gesetzt. Wird eine Überschneidung festgestellt und das aktuelle Level ist nicht das feinste Level 0, so wird das aktuelle Level auf das Nächstfeinere gesetzt und der Test wiederholt. Ist das aktuelle Level gleich 0 und die Überschneidung festgestellt, so wird dies als Ergebnis des Tests zurückgeliefert. Ein Aktivitätsdiagramm der Traversierung ist in Abbildung 19 dargestellt.

Abbildung 19 zeigt ein Aktivitätsdiagramm der Traversierung.

5.2.5 Details der Implementierung

Das Verfahren wurde mithilfe von *Deferred Rendering* realisiert. Die Szene wird aus der Sicht von Lichtquelle und Hauptkamera nur jeweils einmal gerendert. Nachfolgende Renderpasses erfolgen durch das Rendern eines

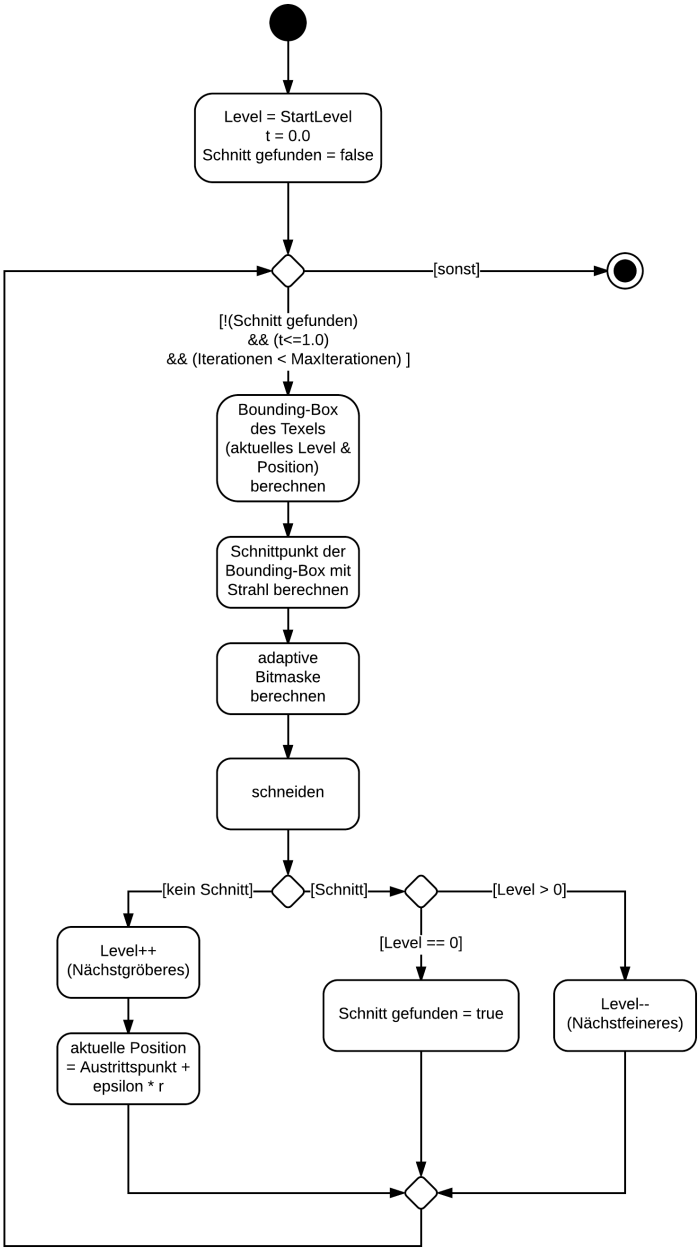


Abbildung 19: Aktivitätsdiagramm der MIP-Map-Traversierung nach [Thiedemann, 2010]

bildschirmfüllenden Dreiecks im Screen-Space. Dabei werden Informationen über Position, Normale und Farbe der Oberfläche in Texturen eines Framebufferobjekts geschrieben und den Shadern in folgenden Renderpasses bei Bedarf als Uniform Variable übergeben. Dieses Texturen werden im Nachfolgenden auch als *G-Buffer* bezeichnet. Während des Renderns der Reflective Shadow Map aus Sicht der Lichtquelle werden Weltposition, Weltnormale und Strahlungsfluss in Texturen eines weiteren Framebufferobjekts geschrieben. In einem finalen Compositing-Renderpass werden die einzelnen Ergebnisse zum Ergebnisbild zusammengefügt.

Die Voxelisierung erfolgt mithilfe von Compute-Shadern, wie in Abschnitt 3.3 beschrieben.

Berechnung des Strahlungsflusses

Der Strahlungsfluss nimmt für eine Spot-Lichtquelle mit Cosinus des Winkels zur Leuchtrichtung ab. Um auch den Öffnungswinkel β des Spots zu berücksichtigen, wird der Strahlungsfluss zusätzlich noch auf das Interval von dem Cosinus des halben Öffnungswinkels bis 1.0 skaliert. Der Strahlungsfluss in Leuchtrichtung beträgt dann 1.0 und an dem halben Öffnungswinkel 0.0. Auf diese Weise gibt das Frustum der Lichtquelle auch den Radius des Lichtkegels der Spot-Lichtquelle vor.

Quellcode 20 zeigt einen Ausschnitt aus dem Fragment-Shader, der die Texturen der Reflective Shadow Map füllt. Der minimale Cosinus des halben Öffnungswinkels wird als uniforme Variable **uniformMinCosine** $= \cos(\beta/2.0)$ übergeben.

```
1  // direction to surface and light direction in view space
2  vec3 surfaceDirection = normalize(viewPosition);
3  vec3 lightDirection   = vec3(0.0, 0.0, -1.0);
4
5  // cosine of angle between light and surface direction
6  float flux = dot(surfaceDirection, lightDirection);
7
8  // set into proportion to minimal cosine
9  flux = (flux - uniformMinCosine) / (1.0 - uniformMinCosine);
10 flux = min(1.0, max(0.0, flux));
```

Quellcode 20: Berechnung des Strahlungsflusses für eine Spot-Lightquelle

Abschließend würde der Strahlungsfluss **flux** noch mit dem diffusen Reflexionsfaktor des Materials multipliziert werden, welcher hier jedoch immer gleich 1.0 gesetzt wurde. In die Textur wird dann in den RGB-Kanälen die mit **flux** skalierte diffuse Oberflächenfarbe eingetragen, in den Alpha-Kanal **flux** selbst.

Sampling Pattern Generierung

Um Rauschen zu vermeiden und Rechenzeit zu vermindern, wird nur ein Sampling Pattern relativ zu (s, t) generiert und für jedes Pixel verwendet. Die Generierung wird im Hauptprogramm vorgenommen und die Werte werden in einer 1D-RGB-Textur im Fließkommaformat auf die GPU geladen. Die Zufallszahlen werden dabei mithilfe der C++ Funktion $\mathrm{std::rand()}$ generiert und die Versätze nach Gleichung 7 berechnet.

```
// generate uniform random number in [0.0, 1.0]
float r1 = (float) std::rand() / (float) RAND_MAX;
float r2 = (float) std::rand() / (float) RAND_MAX;

// generate polar coordinates and weight
float offset_s = r_max * r1 * sin(2.0f * PI * r2);
float offset_t = r_max * r1 * cos(2.0f * PI * r2);
float weight   = r1 * r1;
```

Quellcode 21: Generierung einer Stichprobe

Quellcode 21 zeigt die Generierung einer Stichprobe. Der Versatz **offset_s** zu s, der Versatz **offset_t** zu t und der normalisierte Gewichtungsfaktor **weight** werden dabei jeweils in den R-, G- und B-Kanal des Texels eingelesen. Insgesamt werden genau so viele Stichproben generiert, wie während der Auswertung des indirekten Lichts benötigt werden. Die Summe der Gewichtungsfaktoren ergibt auf diese Weise immer 1.0, und es muss im Shader keine weitere Normalisierung vorgenommen werden.

Berechnung des indirekten Lichts

Die Berechnung des Indirekten Lichts wird auf mehrere Renderpasses aufgeteilt. Dazu wird ein weiterer temporärer Framebuffer mit geringer Auflösung verwendet, der gebunden wird und eine Ausgabetextur erhält. Für jedes Pixel wird ein Fragment-Shader aufgerufen, der nach der Projektion in die Reflective Shadow Map in einer Schleife die vorgegebene Anzahl von Stichproben zieht und verrechnet.

```
// retrieve sampling properties for next sample
float texParam     = float(i) / float(uniformNumSamples);
vec4  properties   = texture(uniformSamplingPattern, texParam);
vec2  sampleUV     = centerUV + properties.xy;
float sampleWeight= properties.z;

// retrieve world position, normal and flux from RSM
vec3 samplePosition = texture(uniformPositionMap, sampleUV).xyz;
vec3 sampleNormal   = texture(uniformNormalMap  , sampleUV).xyz;
vec4 sampleFlux     = texture(uniformFluxMap    , sampleUV);
```

Quellcode 22: Ziehung der nächsten Stichprobe

Quellcode 22 zeigt, wie mithilfe des als 1D-Sampler gebundenen Sampling Patterns die **i**-te Stichprobe aus der Reflective Shadow Map gezogen wird. Die Variable **centerUV** ist dabei die projizierte UV-Koordinate des Pixels in der Reflective Shadow Map. Die Variable **uniformNumSamples** ist dabei die Anzahl der zu ziehenden Stichproben. Die mit **uniform** beginnenden 2D-Sampler sind die Texturen der Reflective Shadow Map.

```
1  // vectors between surface point and sample point
2  vec3 sampleToSurface = surfacePosition - samplePosition;
3  vec3 surfaceToSample = samplePosition - sampleToSurface;
4
5  // radiant intensities between the surfaces
6  float radIntSampleSurface = dot(sampleNormal,  sampleToSurface);
7  float radIntSurfaceSample = dot(surfaceNormal, surfaceToSample);
8  radIntSampleSurface = max(0.0, radIntSampleSurface);
9  radIntSurfaceSample = max(0.0, radIntSurfaceSample);
10
11 // compute irradiance at surface due to sample point light
12 vec3 sampleIrradiance = sampleFlux.rgb;
13 sampleIrradiance *= radIntSampleToSurface;
14 sampleIrradiance *= radIntSurfaceToSample;
15 sampleIrradiance /= pow(length(sampleToSurface), 4);
16
17 // weigh with sample influence
18 sampleIrradiance *= sampleWeight;
19
20 // add to total irradiance at surface point
21 irradiance += sampleIrradiance;
```

Quellcode 23: Einfluss einer Stichprobe auf das indirekte Licht

Quellcode 23 zeigt einen Ausschnitt der Auswertung einer Stichprobe im Fragment-Shader. Um eine in Zeile 15 möglicherweise auftretende Singularität durch Nulldivision zu verhindern, wird die **samplePosition** zuvor um ein geringes Stück entlang der negativen Normalen versetzt. Die Variable **irradiance** speichert fortlaufend die bisher errechnete Bestrahlungsstärke und wird nach Auswertung aller Stichproben als Ausgabewert geschrieben.

Nachdem das niedrig aufgelöste Bild mit der indirekten Beleuchtung geschrieben wurde, folgt die Interpolation in einer voll aufgelösten Kamerasicht. Dazu wird ein Framebuffer-Objekt gebunden und zunächst der Tiefen-Buffer gelöscht. Es wird der Tiefentest aktiviert, damit das Schreiben von Pixeln auch zu dem Schreiben des Tiefen-Buffers führt.

Im Shader sollen dann geeignete Pixelwerte bilinear interpoliert werden. Dazu werden für ein Pixel die vier diagonal zu ihm liegenden Texel des niedrig aufgelösten Bildes gelesen und ihr Einfluss auf die Endfarbe berechnet. Spannen die vier beteiligten Punkte ein Quadrat der Seitenlänge 1 auf und wird der Ursprung auf den linken unteren Punkt gesetzt, so kann allgemein der interpolierte Funktionswert an der Stelle (x, y) mit

$0 \leq x, y \leq 1$ mit Formel 9 berechnet werden. Der Funktionswert entspricht in diesem Fall dem Wert des betreffenden Pixels und die vier Ecken liegen auf den vier Texelmittelpunkten.

$$\begin{aligned} f(x,y) = \quad & f(0,0) \quad (1-x) \quad (1-y) \\ + \quad & f(1,0) \quad x \quad (1-y) \\ + \quad & f(0,1) \quad (1-x) \quad y \\ + \quad & f(1,1) \quad x \quad y \end{aligned} \tag{9}$$

Da auch der Fall eintreten kann, dass nur zwischen drei Texeln interpoliert werden soll, muss der Anteil des vierten Texels herausgerechnet werden.

```
1 if (distance(samplePos, pixelPos) >= uniformDistanceThreshold
2    || dot(sampleNrm,    pixelNrm) <= uniformNormalThreshold)
3 {
4     validSamples--;
5     interpolationBias = 1.0 / ( 1.0 - sampleInfluence );
6     sampleInfluence = 0.0;
7 }
```

Quellcode 24: Bewertung der Tauglichkeit eines Pixels zur Interpolation

Quellcode 24 zeigt den Test, ob ein Texel zur Interpolation geeignet ist. Der Test wird für jedes der vier niedrig aufgelösten Texel ausgeführt. Die beiden Grenzwerte werden jeweils als uniforme Variable übergeben. Die Positionen und die Normalen im Weltkoordinatensystem werden aus den entsprechenden Texturen des G-Buffers gelesen. Ist ein Texel nicht geeignet, so wird die mit 1.0 initialisierte Variable **interpolationBias** angepasst und der Einfluss auf 0.0 gesetzt, sodass der betreffende Summand wegfällt. Auf diese Weise wird das Interpolationsergebnis zwischen den drei verbliebenen Texel normalisiert. Fällt die mit dem Wert 4 initialisierte ganzzahlige Variable **validSamples** unter 3, so wird das Fragment verworfen.

Dank des aktivierten Schreibens des Tiefen-Buffers kann ein anschließender Renderpass mithilfe des Tiefentests die verbleibenden Pixel schreiben. Dieser Renderpass verwendet den gleichen Fragment-Shader für Berechnung des indirekten Lichts wie das niedrig aufgelöste Kamerabild.

Verdeckungstests

Der Test auf Verdeckung wird als Teil des Fragment-Shaders zur Berechnung des indirekten Lichts für jede Stichprobe ausgeführt. Bevor der in Quellcode 23 gezeigte Code den Einfluss einer Stichprobe berechnet, wird zwischen Oberflächenpunkt und Stichprobenposition ein Strahl verfolgt. Wenn der Strahl auf ein belegtes Voxel trifft, so wird die Stichprobe übersprungen, da sie keinen Einfluss auf das indirekte Licht haben kann.

Die Traversierung geschieht bei beiden Verfahren im normalisierten Koordinatensystem. Start- und Endpunkt des Strahles werden nicht direkt auf Oberfläche und Stichprobenposition gesetzt, da das korrespondierende Voxel per Definition bereits belegt ist. Stattdessen wird der Startpunkt etwas entlang der Normalen der Oberfläche verschoben und der Endpunkt etwas entlang der Normalen der Stichprobe. Ein Abbruch der Traversierung erfolgt, wenn eine maximale Anzahl von Iterationen überschritten wird oder der aktuelle Parameter $t > 1.0$ ist. Es wird dann als Ergebnis eine Verdeckungsfreiheit zurückgeliefert.

Die Berechnung der notwendigen Werte für den 3D-DDA Algorithmus erfolgt wie in Abschnitt 5.2.3 beschrieben. Die Voxelindexvariablen **x**, **y** und **z** korrespondieren jedoch nicht mit ganzzahligen Voxelindizes, sondern mit normalisierten Voxelgitterkoordinaten. Die **step**-Variablen besitzen dementsprechend die Größe eines Voxels im normalisierten Koordinatensystem.

Um ein Voxel auszuwerten, müssen sich **x**,**y** und **z** zwischen 0.0 und 1.0 befinden. Sie können dann als Texturkoordinaten verwendet werden. Die Variablen **x** und **y** werden als Texturkoordinate des Voxelgitters verwendet. Die Variable **z** wird verwendet, um die passende Bitmaske aus der 1D-Textur auszulesen. Andernfalls wird das Voxel übersprungen und mit der nächsten Iteration fortgefahren.

Im Falle der hierarchischen Traversierung muss außer Start- und Endpunkt im normalisierten Koordinatensystem auch noch das Startlevel der Mipmap festgelegt werden. Es gibt möglicherweise Startlevel, die schneller zu einem Ergebnis führen.

Um die Bounding Box eines Texels des aktuellen Mipmap-Levels k im normalisierten Koordinatensystem zu berechnen, muss die Texturauflösung $resX$ und $resY$ bekannt sein. Für die Höhe h und Breite b ergibt sich dann $w_k = \frac{2^k}{resX}$ und $h_k = \frac{2^k}{resY}$. Die Tiefe ist, wie oben bereits erläutert, immer 1.0. Mithilfe der Position des aktuellen Startpunktes entlang des Strahls lässt sich dann die linke, untere Ecke der Bounding Box berechnen.

Um den Austrittspunkt des Strahls aus der Bounding Box des Texels zu bestimmen, kann die Anzahl der zu testenden Ebenen mithilfe der Strahlrichtung **v** (siehe Gleichung 8) auf maximal drei Ebenen beschränkt werden. Da der Strahl die Bounding Box in Strahlrichtung immer nur verlassen kann, können mithilfe der Vorzeichen der Skalare von **v** die geschnittenen Seiten der Bounding Box bestimmt werden. Dies sind die Seiten, deren achsenparallelen Ebenennormalen die gleichen Vorzeichen besitzen. Ist ein Skalar der Strahlrichtung gleich 0, so kann der Schnitttest auf der betreffenden Achse übersprungen werden, da der Strahl orthogonal zu ihr verläuft.

In Abschnitt 3.1.2 wurde gezeigt, wie eine 1D-Lookup-Textur von Bitmasken, die zu einer bestimmten Tiefe korrespondieren, erstellt wird. Zur Generierung der benötigten adaptiven Bitmaske kann eine alternative 1D-

Lookup-Textur von Bitmasken verwendet werden. Eine Bitmaske soll einer bestimmten Spanne von Voxeln, bzw. Schichten, entsprechen.

$$\mathrm{b}_i = \sum_{0}^{i} 2^i \tag{10}$$

Gleichung 10 zeigt die Berechnung des Wertes der Bitmaske b der korrespondierenden Schicht i. In der Bitmaske sind demnach alle Bits bis zu der korrespondierenden Schicht i gleich 1. Um nun die Bitmaske zu erhalten, die alle Schichten des Strahlabschnitts abdeckt, müssen die Bitmasken für Startschicht und Endschicht des Strahlabschnitts mit einer logischen XOR-Operation verknüpft werden. Dabei muss für die nähere Schicht i die Bitmaske b_{i-1} gewählt werden.

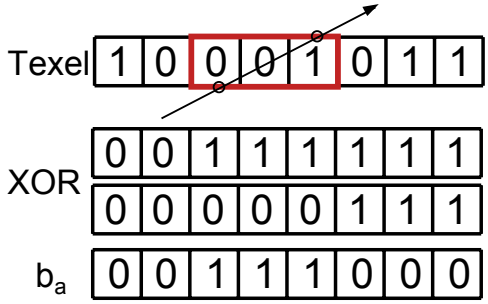


Abbildung 20: Berechnung der adaptiven Bitmaske b_a zu einem Strahlabschnitt

Abbildung 20 zeigt schematisch die Erzeugung der adaptiven Bitmaske zu einem Strahlabschnitt. Die adaptive Bitmaske b_a wird durch die bitweise logische XOR-Operation der beiden eingelesenen Bitmasken erzeugt. Die Bitmaske der näheren Schicht wird so gewählt, sodass das Ergebnis die korrekte Spanne von Schichten abdeckt. Diese kann nun durch eine bitweise logische UND-Operation mit dem Texelwert verknüpft werden.

5.2.6 Ergebnisse

Abbildung 21 zeigt einige Ergebnisbilder des implementierten Reflective Shadow Mapping. Bei direkter Beleuchtung (Abbildung 21a) werden der Lichtquelle abgewandte Oberflächen dunkel schattiert. Ist das Reflective Shadow Mapping aktiviert (Abbildung 21b), so werden sie mit der Farbe der angestrahlten Oberflächen beleuchtet, wie hier mit der Farbe der grünen Wand hinter dem Modell. In Abbildung 21c wird das Modell fast ausschließlich indirekt durch die umgebenden, bunten Wände beleuchtet. Abbildung 21d zeigt die Beleuchtung des Modells durch eine Spot-Lichtquelle.

Abbildung 22 zeigt das Okklusions-Testmodell bei aktiviertem Verdeckungstest (Vgl. Abbildung 18). In der rechten Abbildung ist der Test auf

(a) ohne RSM

(b) mit RSM

(c) indirektes Licht

(d) Spot-Lichtquelle

Abbildung 21: Screenshots der Implementierung von Reflective Shadow Mapping

Verdeckung deaktiviert, wodurch die beiden Kugeln nur von der jeweiligen Farbflächen indirekt beleuchtet werden. Alle Strahlen, die durch die Trennwand verlaufen, werden durch den Verdeckungstest mithilfe des Voxelgitters verworfen.

Abbildung 23 zeigt einen Vergleich der implementierten Traversierungsverfahren. Die hierarchische MIP-Map-Traversierung (Abbildung 23b) verwirft deutlich weniger Strahlen, da Strahlen, die außerhalb des Voxelgitters enden, nicht verfolgt werden. Die 3D-DDA-Traversierung (Abbildung 23a) verwirft hingegen mehr Strahlen, da die Traversierung auch erfolgt, wenn der Strahl außerhalb des Voxelgitters verläuft.

In der Regel wurden für die Aufnahmen folgende Grenzwerte und Einstellungen verwendet: Die Reflective Shadow Map und der G-Buffer besit-

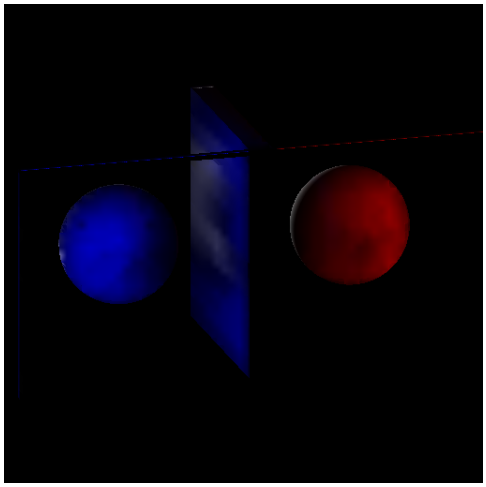
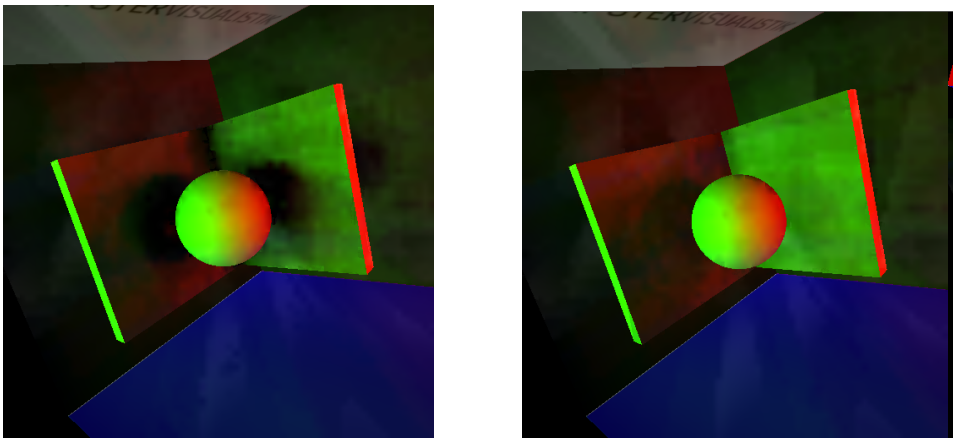


Abbildung 22: Okklusions-Testmodell mit Reflective Shadow Mapping mit aktiviertem Test auf Verdeckung



(a) 3D-DDA **(b)** hierarchisch

Abbildung 23: Reflective Shadow Mapping mit Test auf Verdeckung mit unterschiedlichen Traversierungsverfahren.

zen eine Auflösung von 512x512. Während der Berechnung des indirekten Lichts werden pro Pixel 100 Strahlen über 15 Iterationen durch das Voxelgitter verfolgt. Der Versatz der Start- und Endpunkte beträgt 0.3 Einheiten entlang der jeweiligen Normalen der Oberfläche. Das uniforme Voxelgitter besitzt eine Auflösung von 64x64x32 und umschließt das Testmodell in der Mitte des Raumes. Das zur Interpolation ausgewertete Bild besitzt eine Auflösung von 64x64. Die Schwellwerte lagen meist bei 0.9 für die Normalen und bei 1.0 für die Distanz. Die Strahlen wurden über 15 Iterationen verfolgt.

Mithilfe von Atomic Counter Variablen wurde die Anzahl von verfolgten Strahlen gezählt. In der Testszene mit dem Stanford Bunny Modell werden bei diesen Einstellungen während der Interpolation meist ca. 11500 bis 18000 Pixel verworfen. Dadurch liegt die Anzahl der verfolgen Strahlen meist zwischen 1500000 und 3000000. In Abbildung 21d wurden bei einem Schwellwert von 0.7 für die Normalen insgesamt 2514400 Strahlen verfolgt. Dabei wurden während der Interpolation 21048 Pixel verworfen.

In der Konfiguration von Abbildung 17 wurden bei einem Schwellwert von 0.3 für die Normalen insgesamt 185660 Strahlen verfolgt. Im Ausgangsbild zur Interpolation (Abbildung 17a) werden $32 * 32 = 4096$ Pixel mit je 100 Strahlen ausgewertet. Nach der Interpolation werden noch 17542 weitere Pixel mit je 100 Strahlen ausgewertet.

Im ungünstigsten Fall werden $64 * 64 * 100 + 512 * 512 * 100 = 26624000$ Strahlen verfolgt, wenn während der Interpolation alle Pixel verworfen werden. Bei der Verfolgung von 150 Strahlen läge dieser Wert bereits bei 39936000.

6 Ausblick

Im Zuge dieser Arbeit wurde eine kleine Auswahl von Voxelisierungsmethoden auf der GPU vorgestellt, verglichen und in Beispielanwendungen verwendet. Dennoch wurde nur in einen Teil der Möglichkeiten und aktuellen Entwicklungen in diesem Bereich Einblick gegeben. Die Rasterisierungspipeline zur Voxelisierung zu verwenden, ist noch immer eine beliebte Methode. Gleichzeitig hat sich auch gezeigt, dass die Verwendung von GPGPU-basierten Ansätzen Potenzial besitzt. Daher sollen in diesem Abschnitt mögliche Erweiterungen und Verbesserungen der vorgestellten Verfahren, alternative Verfahren und weiterführende verwandte Themen angesprochen werden.

6.1 Erweiterungen

Mögliche Erweiterungen der implementierten Verfahren können sich auf verschiedene Aspekte beziehen. Nachfolgend werden daher mögliche Erweiterungen im Bezug auf bestimmte Problemstellungen erläutert.

6.1.1 Rasterisierung und zur Voxelisierungsrichtung parallele Geometrien

Mit dem Texturatlas-Verfahren und der Voxelisierung mithilfe von Compute-Shadern wurden bereits zwei aktuellere Alternativen zum rasterisierungsbasierten Slicemapping-Verfahren vorgestellt. Das Problem der rasterisierungsbasierten Voxelisierungsverfahren mit zur Projektionsrichtung parallelen Geometrien hat zahlreiche andere Lösungsansätze motiviert.

Ein früheres, alternatives Verfahren zum Slicemapping wurde bereits von [Dong et al., 2004] vorgestellt. Es verwendet eine ähnliche Kodierung der voxelisierten Szene in 2D-Texturen. Es werden drei Voxelgitter mit orthogonalen Voxelisierungsrichtungen gefüllt und anschließend die Ergebnisse in einer 2D-Textur vereint. Mit diesem Ansatz steigt jedoch auch die benötigte Rechenzeit. Jedoch ist dies ein möglicher Ansatz, um auch die im Zuge dieser Arbeit umgesetzte Implementierung potentiell robuster zu machen. In einem von [Gaitatzes et al., 2011] vorgestellten Verfahren wird der Geometry-Shader verwendet, um zur Voxelisierungsrichtung parallele Geometrien zu erfassen. In der Veröffentlichung nach [Hasselgren et al., 2005] wird eine Lösung präsentiert, um konservative Rasterisierung von Primitiven mithilfe des Vertex-Shaders zu emulieren. Dadurch kann die Anzahl der gesetzten Voxel erhöht werden, da mehr Fragmente produziert werden.

6.1.2 Solid Voxelization

Die im Zuge dieser Arbeit verglichenen Verfahren führen zu einer Oberflächenvoxelisierung der Modelle. Um eine solide Voxelisierung zu erreichen (Vgl. Abschnitt 2.1.2) sind zusätzliche Erweiterungen nötig. Für zwei der hier vorgestellten Verfahren existieren bereits entsprechende Ansätze.

Eine Erweiterung des rasterisierungsbasierten Slicemapping-Verfahrens auf eine solide Voxelisierung wurde von den beiden Autoren des Verfahrens selbst in [Eisemann und Décoret, 2008] vorgestellt. Mit einer ähnlichen Herangehensweise wird in [Schwarz, 2012] auch eine Erweiterung der Voxelisierung mithilfe von DirectCompute erläutert, die sich für das Compute-Shader-basierte Verfahren übertragen lassen könnte.

Es kommt bei beiden Erweiterungen die zusätzliche Anforderung an die Modelle, keine Löcher zu besitzen, hinzu.

Für die texturatlasbasierte Voxelisierung wurde bislang keine spezifische Erweiterung für solide Voxelisierung präsentiert. Doch es könnte untersucht werden, ob mit dem gleichen Ansatz robuste Ergebnisse produziert werden können.

6.1.3 Slicemaps mit höherer Voxelgittertiefe

Bereits in der ursprünglichen Veröffentlichung des rasterisierungsbasierten Slicemapping-Verfahrens in [Eisemann und Décoret, 2006] wurde die begrenzte Voxelgittertiefe angemerkt. Diese kann durch zusätzlich an das Framebuffer-Objekt gebundene Texturen und größere Datenformate erhöht werden. Doch auch diese Möglichkeiten sind beispielsweise durch die maximale Anzahl von Framebuffer-Texturen begrenzt. Insbesondere mit der Verwendung der neuen Schreibzugriffe durch *imageLoadStore* (Vgl. Abschnitt 2.2.1) und Compute-Shadern ist die Verwendung des Framebuffers nicht

mehr notwendig. Denkbar wäre die in [Schwarz, 2012] vorgeschlagene Verwendung eines linearen Arrays oder Buffers. Voxel können mithilfe einer simplen Indizierung angesprochen werden. Auf diese Weise kann eine beliebige Tiefe des Voxelgitters ermöglicht werden. Mögliche Einbußen oder Vorteile in Geschwindigkeit und Nutzbarkeit sind zu erforschen.

6.1.4 Nicht-Binäre Voxeldaten und alternative Voxelgitterstrukturen

Die in dieser Arbeit verwendeten Verfahren führten immer zu einer binären Voxelisierung. Doch auch andere Daten könnten, unabhängig von der Bestimmung der gesetzten Voxel, gespeichert werden. Dann jedoch wird der Aspekt des benötigten Speicherplatzes zu einem wesentlichen Faktor.

In [Thiedemann et al., 2012] werden beispielsweise zusätzlich zur binären Voxelisierung in einer 2D-Textur noch weitere 3D-Texturen verwendet, um beim Setzen eines Voxels noch Informationen zu der Position und der Normalen der Oberfläche zu speichern. Diese Informationen werden dann bei der Berechnung des indirekten Lichts verwendet.

Doch auch die Verwendung von alternativen Datenstrukturen anstelle von Textur-Objekten ist denkbar. Eine sparsame Voxelisierung mithilfe einer eigenen hierarchischen Datenstruktur ist insbesondere für große Voxelgitter interessant. Auch wenn die in einem Voxel zu speichernden Informationen dynamische Größen besitzen können, stoßen die in dieser Arbeit verwendeten Strukturen schnell an ihre Grenzen. Dies kann beispielsweise der Fall sein, wenn zur Beschleunigung von Ray-Tracing dynamische Listen von Dreiecks-Referenzen gespeichert werden müssen.

In [Gaitatzes et al., 2011] wird eine Datenstruktur vorgestellt, die speziell auf die Verwendung von Oberflächenvoxelisierung zugeschnitten ist. Sie soll Funktionalität, Speicherverwaltung und Zugriff auf die Voxelisierung kapseln. Pro Voxel soll die Speicherung von bis zu 1024 Bit von beliebiger Information zugelassen werden. Weiterhin soll die Integration in bestehende Anwendungen besonders einfach sein. Es steht also zur Debatte, wie eine generische Datenstruktur zu Voxelisierungszwecken für verschiedene Anwendungsfälle von Nutzen sein kann. Es werden außerdem weitere Verfahren vorgestellt, um die Rasterisierungspipeline zur Oberflächenvoxelisierung zu nutzen.

6.1.5 Erforschung des Potenzials der Compute-Shader Ansätze

In dieser Arbeit wurde ein besonderer Wert auf die Verwendung der allgmein zu programmierenden Compute-Shader gelegt. Es konnte bereits gezeigt werden, dass sich Compute-Shader zur Echtzeit-Voxelisierung verwenden lassen. Doch es ist ebenfalls erkenntlich geworden, dass das in Abschnitt 3.3 vorgestellte Verfahren noch verbessert werden kann.

Eine in [Schwarz, 2012] bereits vorgeschlagene Lösung für die bisher durch große Dreiecke verursachte Verlangsamung könnte in der Tessellation liegen, sodass die Dreiecke, wenn nötig, bis zu einer sinnvollen Größe aufgeteilt werden. Eine andere vorgeschlagene Lösung könnte in der Aufteilung der Voxelisierung eines Dreiecks in mehrere Teilschritte liegen. Das Voxelgitter soll dazu in gleichgroße Bereiche eingeteilt werden. Die von einem Dreieck überlappten Voxel werden bestimmt und mit einer Referenz auf das Dreieck versehen. Anschließend soll dann pro Bereich eine Arbeitsgruppe die Überlappungstests mit den referenzierten Dreiecken durchführen. Beide Ansätze wurden jedoch noch nicht erprobt.

In [Schwarz und Seidel, 2010] wird zudem praktisch das gleiche Verfahren mithilfe von CUDA verwirklicht. Weiterführend wird zudem noch eine Lösung zur sparsamen soliden Voxelisierung vorgestellt.

Im Zuge dieser Arbeit wurde zudem das in [Thiedemann et al., 2011] vorgestellte Verfahren auf die Nutzung von Compute-Shader übertragen. Die Tests lassen zudem die Vermutung zu, dass das Verfahren durch den Verzicht auf die Nutzung der Rasterisierungspipeline vieleicht sogar beschleunigt werden kann. Es ist daher interessant, ob sich dies in weiterführenden Tests bestätigen lässt.

6.2 Voxel Cone Tracing

In Abschnitt 5.2 wurde ein gefülltes Voxelgitter verwendet, um die Darstellung von indirektem Licht zu verbessern. Auch andere, auf die Simulation von indirekter Beleuchtung abzielende Verfahren, stützen sich insbesondere zu Beschleunigungszwecken auf voxelbasierte Repräsentationen der Szene. Eines ist das von [Crassin et al., 2011] vorgestellte Verfahren des *Voxel Cone Tracing*.

Das Verfahren ist im Zusammenhang mit dieser Arbeit besonders interessant, da es viele der behandelten oder angesprochenen Methoden, Strukturen und Spezialisierungen in einem Verfahren vereint. Es ist sehr umfangreich, sodass daher nur einige Aspekte genannt werden sollen, die im Zuge dieser Arbeit ebenfalls in den Grundzügen angerissen wurden. Die tatsächliche Umsetzung ist auf die speziellen Anforderungen des Verfahrens angepasst worden.

Das Verfahren arbeitet auf einer Sparse Voxel-Octree-Repräsentation der Szene auf der Grundlage der von [Laine und Karras, 2010] und [Crassin et al., 2009] vorgestellten Datenstrukturen.

Diese Datenstrukturen weisen eine zeigerbasierte sparsame Octree-Struktur auf. Die Knoten des Octrees verweisen auf Texturspeicher, in denen Informationen über Oberflächennormale, Farbe und Materialeigenschaften gespeichert werden; sie ist also mehrwertig. Weiterhin erlauben zusätzliche Nachbarschaftsreferenzen eine effiziente Traversierung des Octrees in beliebige Richtungen und Hierarchiestufen.

Die hierarchische Berechnung der Knotenwerte wird durch einen texturbasierterten Mip-Mapping-Ansatz realisiert.

Gleichzeitig werden die Pointer in einem linearen Speicher dynamisch mithilfe von Atomic Countern verwaltet.

Die Voxelisierung der Szene erfolgt mithilfe der Rasterisierungspipeline, wobei die Geometrie mehrmals aus orthogonalen Richtungen rasterisiert wird, um eine möglichst vollständige Oberflächenvoxelisierung zu erzielen.

Die Berechnung der indirekten Beleuchtung verwendet das namensgebende Verfahren des Voxel-Cone-Tracing. Statt einen Strahl zu einem nahen Oberflächenpunkt zu verfolgen, wird mit steigender Entfernung zum Startpunkt die hierarchisch immer grober zusammengefasste Information über die Umgebung verwendet. Bildlich kann sich ein Strahl dadurch als Gebilde in der Form eines Trichters (engl. *cone*) vorgestellt werden, dessen immer größer werdender Radius der immer gröber zusammengefassten Information über die Umgebung entspricht. Anders als bei Reflective Shadow Mapping sind dadurch nicht etwa viele hundert Strahlen zum Annähern eines plausiblen Lichteindrucks nötig, sondern es reicht die Verfolgung einiger weniger Voxel-Cones.

6.3 Line-Space

Der *Line-Space* oder Linienraum ist eine diskret parametrisierte Darstellung des Raumes, die den Raum vollständig beschreibt, indem alle möglichen diskretisierten Linien durch ihn erfasst werden. Wird für den Raum eine uniforme Voxelgitterstruktur von begrenzten Dimensionen in Höhe, Breite und Tiefe definiert, so kann der Linienraum generiert werden. Jede äußere Seite des Voxelgitters besitzt dann eine durch die gewählte Auflösung begrenze Anzahl von Teilflächen. Wird zwischen zwei Teilflächen eine Verbindung gezogen, so können für diese mithilfe einer 3D-Scan-Konvertierung der Linie alle geschnittenen Voxel bestimmt werden. Auf diese Weise kann für einen beliebigen Strahl aus der Startposition und der Strahlrichtung direkt bestimmt werden, welche Voxel dabei geschnitten werden.

Andersherum können alle Linien durch das Voxelgitter bestimmt werden, die ein bestimmtes Voxel durchlaufen. Werden in einem diskreten 2D-Koordinatensystem auf einer Koordinatenachse alle möglichen Startflächen und auf der anderen alle möglichen Endflächen indiziert aufgetragen, so ergibt sich ein diskreter Parameterraum aller Linien durch das Voxelgitter: der Linienraum.

Es liegt ein Vergleich zur *Hough-Transformation* von 2D-Geraden nahe, bei der der Parameterraum alle möglichen Geraden erfasst, indem der Abstand und der Winkel aller möglichen Geraden durch das Bild diskret erfasst wird. Jedes Pixel des Parameterraums entspricht dann genau einer möglichen Geraden im Bild. So können für jedes beliebige Pixel des Bildrau-

mes (Vgl. Voxel des Voxelgitters) alle möglichen Geraden im Parameterraum bestimmt werden (Vgl. Linien des Linienraumes), die das Pixel schneiden.

In diesem Sinne kann eine binär voxelisierte Szene in den Linienraum transformiert werden. Ein Pixel des Linienraumes enthält dann etwa die Information, ob die Linie ohne Überschneidung mit einem gesetzten Voxel traversiert werden kann. Dies könnte erhebliche Vorteile etwa bei der Strahlenverfolgung von Ray-Tracing-Verfahren bieten. Es müsste nicht jeder Strahl einzeln verfolgt werden, sondern eine einzige Anfrage an den Linienraum könnte genügen.

Es ist von Interesse zu erforschen, ob es Verfahren gibt, die von diesem Konzept tatsächlich profitieren, und ob sich diese Transformation mithilfe der GPU beschleunigen lässt. Die im Zuge dieser Arbeit implementierten Voxelisierungsverfahren bieten einen idealen Ansatzpunkt. Es wurde bereits in der Diplomarbeit von [Lemke, 2014] eine hierarchische Datenstruktur mithilfe des Line-Space-Konzeptes entworfen, die das Ray-Tracing auf der CPU beschleunigen konnte. Eine mithifle von Compute-Shadern realisierte Transformation von 2D-Kantenbildern in den Line-Space wurde in der Bachelorarbeit von [Hunz, 2013] implementiert.

Literatur

Tomas Akenine-Möller. Fast 3D Triangle-Box Overlap Testing. *Journal of Graphics Tools*, 6:29–33, 2001.

John Amanatides und Andrew Woo. A Fast Voxel Traversal Algorithm for Ray Tracing. In *In Eurographics '87*, pages 3–10, 1987.

Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, und Elmar Eisemann. GigaVoxels : Ray-Guided Streaming for Efficient and Detailed Voxel Rendering. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*, Boston, MA, Etats-Unis, feb 2009. ACM, ACM Press. to appear.

Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, und Elmar Eisemann. Interactive Indirect Illumination Using Voxel-based Cone Tracing: An Insight. In *ACM SIGGRAPH 2011 Talks*, SIGGRAPH '11, pages 20:1–20:1, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0974-5.

Carsten Dachsbacher und Marc Stamminger. Reflective Shadow Maps. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, I3D '05, pages 203–231, New York, NY, USA, 2005. ACM. ISBN 1-59593-013-2.

Zhao Dong, Wei Chen, Hujun Bao, Hongxin Zhang, und Qunsheng Peng. Real-time voxelization for complex polygonal models. In *Computer Gra-*

phics and Applications, 2004. PG 2004. Proceedings. 12th Pacific Conference on, pages 43–50, Oct 2004.

David Eberly. Intersection of convex objects: The method of separating axes. `www.magic-software.com`, 2001.

Elmar Eisemann und Xavier Décoret. Fast Scene Voxelization and Applications. In *ACM SIGGRAPH 2006 Sketches*, SIGGRAPH '06, New York, NY, USA, 2006. ACM. ISBN 1-59593-364-6.

Elmar Eisemann und Xavier Décoret. Single-pass GPU Solid Voxelization for Real-time Applications. In *Proceedings of Graphics Interface 2008*, GI '08, pages 73–80, Toronto, Ont., Canada, Canada, 2008. Canadian Information Processing Society. ISBN 978-1-56881-423-0.

Shiaofen Fang, Shiaofen Fang, Hongsheng Chen, und Hongsheng Chen. Hardware Accelerated Voxelization. *Computers and Graphics*, 24:200–0, 2000.

Vincent Forest, Loïc Barthe, und Mathias Paulin. Real-Time Hierarchical Binary-Scene Voxelization. *Journal of Graphics Tools*, 14(3):21–34, 2009.

Athanasios Gaitatzes, Pavlos Mavridis, und Georgios Papaioannou. Two Simple Single-pass GPU methods for Multi-channel Surface Voxelization of Dynamic Scenes. *Proceedings of the 19th Pacific Conference on Computer Graphics and Applications (short papers - Pacific Graphics 2011)*, 2011.

Jon Hasselgren, Tomas Akenine-Möller, und Lennart Ohlsson. Conservative Rasterization. In Matt Pharr, editor, *GPU Gems 2*, pages 677–690. Addison-Wesley, 2005.

Jochen Hunz. The Possibilities of Compute Shaders - an Analysis. Bachelorarbeit, Universität Koblenz-Landau, Campus Koblenz, 2013.

Arie Kaufman und Eyal Shimony. 3D Scan-conversion Algorithms for Voxel-based Graphics. In *Proceedings of the 1986 Workshop on Interactive 3D Graphics*, I3D '86, pages 45–75, New York, NY, USA, 1987. ACM. ISBN 0-89791-228-4.

John Kessenich, Dave Baldwin, und Randi Rost. The OpenGL® Shading Language. Technical report, The Khronos Group Inc., 2012. URL `http://www.opengl.org/registry/doc/GLSLangSpec.4.30.6.pdf`. Zugriff: 28.09.2014.

Samuli Laine und Tero Karras. Efficient Sparse Voxel Octrees – Analysis, Extensions, and Implementation. NVIDIA Technical Report NVR-2010-001, NVIDIA Corporation, feb 2010.

Orion Sky Lawlor und Laxmikant V. Kalée. A Voxel-based Parallel Collision Detection Algorithm. In *Proceedings of the 16th International Conference on Supercomputing*, ICS '02, pages 285–293, New York, NY, USA, 2002. ACM. ISBN 1-58113-483-5.

Paul Lemke. Effiziente Schnitttestanfragen durch eine Line-Space-Hierarchie. Diplomarbeit, Universität Koblenz-Landau, 2014.

Duoduo Liao und Shiaofen Fang. Fast Volumetric CSG Modeling Using Standard Graphics System. In *Proceedings of the Seventh ACM Symposium on Solid Modeling and Applications*, SMA '02, pages 204–211, New York, NY, USA, 2002. ACM. ISBN 1-58113-506-8.

Tom Lokovic und Eric Veach. Deep Shadow Maps. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 385–392, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co. ISBN 1-58113-208-5.

John D Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E Lefohn, und Timothy J Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer graphics forum*, 26(1):80–113, 2007.

Juan Pineda. A Parallel Algorithm for Polygon Rasterization. In *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '88, pages 17–20, New York, NY, USA, 1988. ACM. ISBN 0-89791-275-6.

Michael Schwarz. Practical Binary Surface and Solid Voxelization with Direct3D 11. In Wolfgang Engel, editor, *GPU Pro 3: Advanced Rendering Techniques*, pages 337–352. A K Peters/CRC Press, Boca Raton, FL, USA, 2012.

Michael Schwarz und Hans-Peter Seidel. Fast Parallel Surface and Solid Voxelization on GPUs. In *ACM SIGGRAPH Asia 2010 Papers*, SIGGRAPH ASIA '10, pages 179:1–179:10, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0439-9.

Mark Segal und Kurt Akeley. The OpenGL®Graphics System: A Specification (Version 4.3 (Core Profile). Technical report, The Khronos Group Inc., 2012. URL `http://www.opengl.org/registry/doc/glspec43.core.20120806.pdf`. Zugriff: 28.09.2014.

Sinje Thiedemann. Voxelbasierte globale Beleuchtung in dynamischen Szenen. Diplomarbeit, Universität Koblenz-Landau, 2010.

Sinje Thiedemann, Niklas Henrich, Thorsten Grosch, und Stefan Müller. Voxel-based Global Illumination. In *Symposium on Interactive 3D Graphics and Games*, I3D '11, pages 103–110, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0565-5.

Sinje Thiedemann, Niklas Henrich, Thorsten Grosch, und Stefan Müller. Real-Time Near-Field Global Illumination Based on a Voxel Model. In Wolfgang Engel, editor, *GPU Pro 3: Advanced Rendering Techniques*, pages 209–229. A K Peters/CRC Press, Boca Raton, FL, USA, 2012.

Lance Williams. Casting Curved Shadows on Curved Surfaces. *SIGGRAPH Comput. Graph.*, 12(3):270–274, August 1978. ISSN 0097-8930.