UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Optimierung Voxel-basierter globaler Beleuchtung unter OpenGL 4.4

Bachelorarbeit

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

vorgelegt von
Kevin Kremer

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

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
Kevin Kremer
(Mat. Nr. 211 200 397)

Thema: Optimierung Voxel-basierter Globaler Beleuchtung unter OpenGL 4.4

Um eine realistische Beleuchtung einer Szene zu erreichen, muss neben der direkten Beleuchtung auch indirekte Beleuchtung berücksichtigt werden. Mit diesem Schwerpunkt befasst sich das Verfahren der Voxel-basierten Globalen Beleuchtung, welches die Szene in Voxel einteilt und anhand den entstehenden Szeneninformationen mittels Raytracing eine Globale Beleuchtung berechnet. Dadurch kann eine realistisch wirkende Szene effizient in Echtzeit beleuchtet werden.

Ansatz dieser Arbeit ist es, dieses Globale Beleuchtungsverfahren unter OpenGL 4.4 umzusetzen. Ziel dieser Arbeit ist dabei, anhand neuer, vorhandener Funktionalität durch die aktuelle OpenGL-Version eine verbesserte Performanz des Verfahrens herbeizuführen.

Insbesondere wird die Thematik behandelt, inwiefern bestimmte neue Funktionen in OpenGL 4.4 zu einer Leistungssteigerung führen und in welchem Aspekt des Beleuchtungsverfahren sich diese äußert.

Die inhaltlichen Schwerpunkte der Arbeit sind:

1. Recherche über Globale Beleuchtungsverfahren und OpenGL 4.4
2. Implementierung der Voxel-basierten Globalen Beleuchtung
3. Optimierung des Verfahrens unter OpenGL 4.4
4. Evaluation hinsichtlich der Performanz
5. Dokumentation der Ergebnisse

Koblenz, 01.04.2014

– Kevin Kremer –

– Prof. Dr. Stefan Müller –

Zusammenfassung

Die vorliegende Arbeit befasst sich mit der Anwendung und Optimierung globaler Beleuchtung in dreidimensionalen Szenen. Dabei wird nicht nur die direkte Beleuchtung in Abhängigkeit einer oder mehrerer Lichtquellen, sondern auch indirekte Beleuchtung durch umliegende Objekte berücksichtigt. Schwerpunkt dieser Arbeit ist es, die Ergebnisse eines globalen Beleuchtungsverfahrens durch die Implementation unter OpenGL 4.4 zu verbessern. Dies geschieht mithilfe einer Voxelisierung der Szene. Durch eine Traversierung der entstehenden Voxel-Struktur werden zusätzliche Informationen der Szene entnommen, was zu einer realistisch wirkenden globalen Beleuchtung beiträgt.

Abstract

The present thesis covers the implementation and optimization of global illumination in three-dimensional scenes. Global illumination does not only consider direct illumination dependent on one or more light sources, but also indirect illumination which is emitted by surrounding objects in the scene. The thesis focuses on the implementation of a global illumination method and its improvement using OpenGL 4.4. This is done by a voxelization of the scene. By traversing the resulting voxel structure, additional information is taken from the scene, which contributes to a plausible global illumination.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Die Computergrafik ist ein sich rasant entwickelndes Fachgebiet. Während vor etwa 20 Jahren gerade erst Videospielkonsolen mit 3D-Grafik auf den Markt kamen, besitzen einige 3D-Anwendungen heutzutage eine Grafik, die sich dem Photorealismus immer weiter annähert. Neben vielen anderen Faktoren hängt dies sowohl mit der immer weiter steigenden Effizienz von Grafikprozessoren und deren Schnittstellen, als auch der stetigen Weiterentwicklung von Beleuchtungsverfahren zusammen.

Ein Mittel, um eine 3D-Szene realistischer zu gestalten, ist die Berücksichtigung globaler Beleuchtung. Die Simulation dieser ist ein großes Thema im Bereich der photorealistischen Computergrafik. Dabei müssen physikalisch korrekte Annahmen einer Szene getroffen und umgesetzt werden, die sowohl von der Lichtquelle als auch von allen Objekten innerhalb der Szene und ihrer Beschaffenheit abhängen. Beispiele für die Umsetzung globaler Beleuchtung in 3D-Szenen finden sich in Abbildung 1 [1].

Es existieren viele verschiedene Verfahren, welche versuchen, eine solche globale Beleuchtung in Echtzeit zu simulieren. Ein Beispiel dafür ist das Verfahren von Dachsbacher und Stamminger [2], welches von der Lichtquelle aus sichtbare Objekte als Ausstrahler indirekten Lichts betrachtet und mit der direkten Beleuchtung kombiniert. Ein Ansatz von Thiedemann [7] baut auf diesem Verfahren auf und erstellt eine Voxelstruktur einer Szene, um die globale Beleuchtung innerhalb eines Nahfelds zu berechnen.

Moderne Grafikschnittstellen wie OpenGL bieten eine immer weiter steigende Funktionalität in Zusammenarbeit mit stetig optimierter Grafikhardware. Dies ermöglicht zum Einen eine vereinfachte Implementation, zum Anderen performante Ergebnisse bei der Umsetzung globaler Beleuchtungsverfahren.

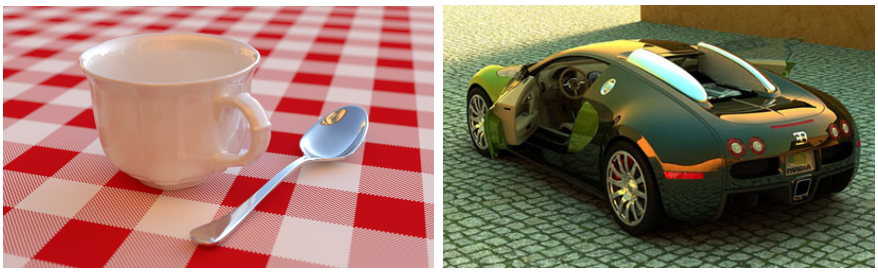


Abbildung 1: Globale Beleuchtung sorgt dafür, dass gerenderte Bilder heutzutage sehr realistisch wirken.

[1] Linke Grafik entnommen von www.peterkutz.de. Rechte Grafik aus http://www.club-3d.com/index.php/produkte/leser.de/product/geforce-gtx-570.870.html?page=4. Beide Grafiken zuletzt abgerufen am 17.09.2014.

1.2 Zielsetzung

Ziel dieser Arbeit ist es, ein Verfahren zur Berechnung globaler Beleuchtung unter OpenGL 4.4 zu implementieren und auf dreidimensionale Szenen in Echtzeit anzuwenden. Hierbei soll ein Ansatz verwendet werden, welcher auf der Repräsentation der Szene in einem Voxelgitter aufbaut. Diese Voxelstruktur wird für die Szene im Voraus erstellt und benutzt, um die indirekte Beleuchtung zu berechnen. Dabei wird die indirekte Beleuchtung auf das Nahfeld um das Empfängerobjekt reduziert. Das theoretische Hintergrundwissen zur Voxelisierung einer Szene sowie zur Berechnung der direkten und indirekten Beleuchtung soll dabei vermittelt und deren praktische Umsetzung aufgezeigt werden.

Zur Implementation des Verfahrens wird ein breites Spektrum an Funktionalität benutzt, die unter Verwendung der Programmiersprache C++ und OpenGL-Version 4.4 vorhanden ist. Die Ergebnisse werden ausgewertet und unter Betrachtung verschiedener Zusammenhänge miteinander verglichen, um zu ermitteln, in welchen Bereichen Verbesserungen bei Verwendung der Voxelstruktur erreicht werden können.

2 Grundlagen

2.1 Direkte und indirekte Beleuchtung

In der Realität besteht Licht aus Wellen, die sich im Raum verteilen und dabei auf Objekte treffen. Je nach Wellenlänge des Lichts und Beschaffenheit der Objekte wird dieses Licht reflektiert. Mit unseren Augen können wir diese Reflexion wahrnehmen und erkennen dadurch die Farbe eines Objekts unter der gegebenen Beleuchtung.

In Computerprogrammen, die 3D-Bilder erzeugen, wird Licht jedoch meist durch Strahlen anstelle von Wellen simuliert, da diese leichter zu berechnen sind. Dabei lassen sich 2 Arten der Beleuchtung unterscheiden. Bei der *direkten Beleuchtung* wird ein Lichtstrahl betrachtet, der direkt von einer Lichtquelle auf die Oberfläche eines Objekts und von dort aus auf ein Auge bzw. eine Kamera trifft. Wird ein Objekt an einem Punkt seiner Oberfläche von keinem Lichtstrahl getroffen, erscheint dieser Punkt schwarz. Treffen jedoch ein oder mehrere Lichtstrahlen auf dieses Objekt, reflektiert es abhängig von seiner Beschaffenheit Licht weiter in den Raum, welches vom Auge aufgenommen werden kann. Wie genau das Licht reflektiert wird, hängt unter Anderem von den spiegelnden und diffusen Eigenschaften des Objekts ab. Während spiegelnde Oberflächen Licht nur in eine Richtung reflektieren, geben diffuse Oberflächen das eingehende Licht in viele verschiedene Richtungen weiter. Eine direkte Beleuchtung betrachtet also nur das Licht, das direkt von einer Lichtquelle auf Oberflächen landet.

Indirekte Beleuchtung hingegen berücksichtigt jedoch das Licht, dass von Objekten reflektiert wird und anschließend auf anderen Objekten im Raum landet, bevor es letztendlich auf das Auge trifft. Diese Reflexion von Objekt zu Objekt kann theoretisch beliebig oft stattfinden. Das dadurch entstehende Abfärben der Farbe eines Objekts wird auch als *Color Bleeding* bezeichnet. Da es jedoch in der Praxis nicht möglich ist, beliebig viele Strahlen innerhalb einer Szene zu berechnen, wird dieses indirekte Licht oft nur auf eine bestimmte Anzahl Reflexionen oder eine gewisse Länge der Lichtstrahls beschränkt.

Durch eine Kombination aus direktem und indirektem Licht in einer Szene soll eine *globale Beleuchtung* erzielt werden. Dabei wird versucht, sämtliche Ausbreitungen von Strahlen in einem Raum korrekt zu simulieren. Mathematische Grundlage der globalen Beleuchtung bietet die sogenannte *Rendergleichung* (*rendering equation*), die erstmals von Kajiya [5] eingeführt wurde . Mit ihr lässt sich berechnen, wie viel Licht ein Oberflächenpunkt von einem anderen Oberflächenpunkt erhält. Die Gleichung lautet wie folgt:

$$L(x, x') = g(x, x') \cdot \left(L_e(x, x') + \int_S b(x, x', x'') L(x', x'') \mathrm{d}x'' \right) \tag{1}$$

Im Folgenden werden die einzelnen Terme der Gleichung erklärt:

- $L(x, x')$ wird als *Energiefluss* bezeichnet und beschreibt, wie viel Licht, das von Punkt x' gesendet wird, an Punkt x ankommt.
- $g(x, x')$ beschreibt, wie die Punkte in der Szene zueinander liegen und wird daher als *geometrischer Term* bezeichnet.
- $L_e(x, x')$ ist der *Emissionsterm*. Er gibt an, wie viel Licht von Punkt x' an Punkt x ausgestrahlt wird.
- Der Term $b(x, x', x'')$ wird als *Streuungsterm* bezeichnet. Er gibt an, wie viel Licht von x'' aus auf den Punkt x' trifft und von dort aus an den Punkt x reflektiert wird.
- S beschreibt letztendlich die Anzahl aller Flächen, die sich in der Szene befinden.

Ein Ziel globaler Beleuchtungsverfahren ist es, die Ergebnisse der Rendergleichung möglichst gut anzunähern, um dadurch mathematisch korrekte Ergebnisse bei der Beleuchtung einer Szene zu erzielen.

2.2 Raytracing

Als Raytracing (deutsch: Strahlenverfolgung) wird in der Computergrafik ein Verfahren bezeichnet, das über das Aussenden von Strahlen die Verdeckung und Beleuchtung in einer Szene ermittelt. Dabei kann automatisch auch die Verschattung von Objekten mitberechnet werden. Aus Sicht eines Augpunkts werden Strahlen in die Szene geschossen, um zu überprüfen, auf welche Objekte diese treffen. Ein Strahl $\vec{s}$ besteht dabei aus einem Startpunkt $\vec{u}$ und einer Richtung $\vec{v}$. Ein Punkt auf diesem Strahl kann durch folgende Gleichung berechnet werden:

$$\vec{s} = \vec{u} + t\vec{v} \tag{2}$$

Dabei ist t ein Parameter, der beschreibt, an welcher Stelle des Strahls sich der Punkt befindet. t liegt im Intervall $[0, 1]$. Die Überprüfung auf Objekte entlang des Strahls wird mit sogenannten Schnittpunkttests durchgeführt. Das vorderste, getroffene Objekt entlang des Strahls ist dann das Objekt, das von dem Augpunkt aus sichtbar ist. Nun können ausgehend von dem getroffenen Schnittpunkt weitere Strahlen in die Szene geschossen

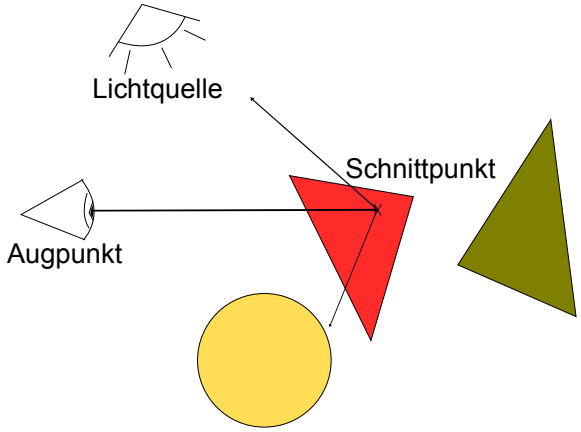


Abbildung 2: Das Prinzip eines Raytracing-Vorgangs. Aus Sicht eines Augpunkts wird ein Strahl in die Szene geschossen. Der erste Punkt auf der Oberfläche eines Objekts, das entlang dieses Strahls getroffen wird, wird als Schnittpunkt festgehalten. Anhand dieses Punktes wird dann die Farbe für den Pixel festgelegt, aus dem der Strahl entsprungen ist. Vom Schnittpunkt aus können dann weitere Strahlen in die Szene reflektiert werden, um die indirekte Beleuchtung oder Verschattungen durch andere Objekte zu ermitteln.

werden, um zum Beispiel Verschattungen des Objekts durch andere Objekte oder die Beleuchtung durch eine Lichtquelle zu ermitteln. Der Raytracing-Vorgang wird in Abbildung 2 verdeutlicht. Durch das Aussenden mehrerer Strahlen pro Pixel wird versucht, ein realistisches 3D-Bild einer Szene wie in Abbildung 3 [2] zu berechnen.

Raytracing lässt sich in einem Programm leicht implementieren. Allerdings besitzen die meisten Raytracing-Algorithmen eine sehr hohe Laufzeit, was die Anwendung von klassischem Raytracing in Echtzeit-Anwendungen bei komplexen Szenen nicht möglich macht. Aus diesem Grund existieren viele Optimierungsansätze, die u.A. mithilfe spezieller Datenstrukturen versuchen, die Laufzeit von Raytracing-Algorithmen zu erhöhen, um dadurch die Beleuchtung in Echtzeit berechnen zu können.

[2] Grafik entnommen von http://kb-en.radiantzemax.com/Knowledgebase/Photorealistic-Rendering-Using-Zemax-and-POV-Ray, zuletzt abgerufen am 17.09.2014

Abbildung 3: Beispiel für ein mittels Raytracing gerendertes Bild

2.3 Voxelisierung

Wenn 3D-Modelle für eine Anwendung erstellt werden, bestehen diese aus Polygonen, also einer großen Menge von Dreiecken verschiedener Größe. Indem immer mehr solcher Dreiecke für ein Modell verwendet werden, versucht man, dessen Qualität zu verbessern und es realistischer erscheinen zu lassen.

Ein Problem mit dieser Darstellungsform zeigt sich jedoch, wenn man versucht, bestimmte Algorithmen auf diese Polygone anzuwenden. Will man die Beleuchtung einer dreidimensionalen Szene mithilfe eines einfachen Raytracing-Verfahrens bestimmen, stellt man fest, dass das Rendern der Szene sehr zeitaufwändig ist. Dies hängt damit zusammen, dass besonders die Schnittpunkttests eines einfachen Raytracers einen hohen Aufwand besitzen, da der Strahl gegen jedes Primitiv der Szene getestet werden muss. Gerade bei komplexen Szenen mit vielen Primitiven kann ein solches Verfahren Bilder nicht in Echtzeit rendern.

Eine Idee ist es daher, die Szene in eine einfachere Datenstruktur zu bringen, um sich damit nachfolgende Berechnungen wie Schnittpunkttests zu erleichtern und dadurch die Performanz zu erhöhen. Eine beliebte Datenstruktur ist zum Beispiel das *Voxelgitter*. In diesem werden die Polygone der Szene in einzelne *Voxel* unterteilt, welche zusammengesetzt das gesamte Voxelgitter ergeben, das die Szene vollständig abdeckt. Ein Beispiel dazu liefert Abbildung 4.

Ein Voxel lässt sich definieren als Würfel einer beliebigen Größe, welcher Informationen über Elemente innerhalb dieses Voxels enthält. Befinden sich ein oder mehrere Objekte z.B. im kanonischen Volumen (d.h. in einem Einheitswürfel, bei dem die xyz-Koordinaten jedes Objekts im Wertebereich

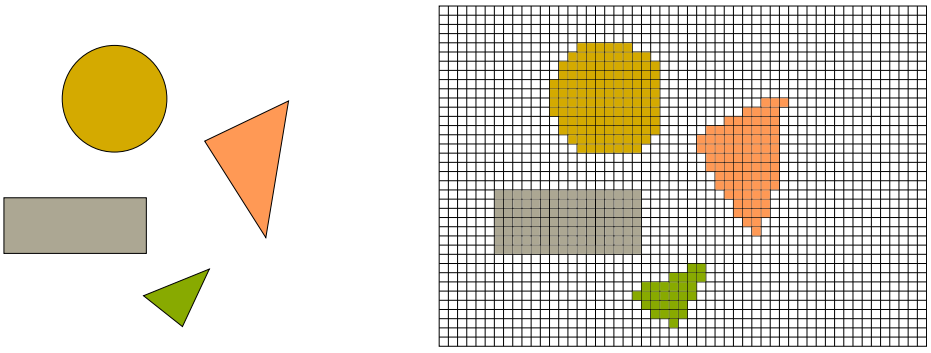


Abbildung 4: Erstellung eines Voxelgitter in 2D-Ansicht. Objekte im Raum (links) werden an der jeweiligen Position in ein vorher festgelegtes, einheitliches Gitter eingesetzt (rechts). Die einzelnen Voxel speichern Informationen über die darin befindlichen Objekte. Dadurch lässt sich direkt abfragen, ob an einer bestimmten Stelle im Voxelgitter ein Objekt vorliegt.

$[-1,-1,-1]\times[1,1,1]$ liegen), kann man die Szene also in einem n^3 Voxelgitter darstellen, wobei n die Anzahl der Voxel in x-Richtung sei. Ein Voxelgitter muss jedoch nicht kubisch sein; je nach Anwendungsfall kann z.B. die Tiefe eines Voxelgitters eine höhere oder niedrigere Auflösung besitzen.

Bei der *Voxelisierung* spricht man demnach von einem Verfahren, welches eine aus Polygonen bestehende Szene in ein solches Voxelgitter transformiert. Dabei unterscheidet man zwischen einer *binären* und einer *mehrwertigen Voxelisierung*. Bei einer binären Voxelisierung wird eine Szene so in Voxel unterteilt, dass ein Voxel nur die Information enthält, ob sich in diesem ein Objekt bzw. Teil eines Objektes befindet oder nicht. Im Gegensatz dazu werden bei der mehrwertigen Voxelisierung zusätzliche Informationen in den Voxeln gespeichert. Das ist besonders nützlich, wenn man z.B. die exakte Position eines Voxels oder die Normale eines sich an dieser Stelle befindenden Objektes abfragen möchte. Ob ein Objekt in einem Voxel enthalten ist, lässt sich also dadurch feststellen, dass in diesem Voxel bestimmte Werte vorliegen und sie somit nicht leer sind. Für diese Informationen wird jedoch mehr Speicheraufwand benötigt, und Methoden zum Speichern dieser müssen in der verwendeten API vorhanden sein.

Die simpel aufgebaute Struktur des Voxelgitters kann nun benutzt werden, um darauf ausgerichtete Algorithmen anzuwenden. Dabei wird der Rechenaufwand erheblich vereinfacht; Probleme wie z.B. das Traversieren einer Voxelstruktur können durch triviale und zugleich effiziente Algorithmen gelöst werden. Mittlerweile gibt es viele verschiedene Voxelisierungsverfahren, die eine Szene ohne großen Rechenaufwand in ein Voxelgitter einteilen, sodass der zusätzliche Aufwand der Erstellung eines solchen Gitters möglichst gering bleibt. Probleme zeigen sich jedoch, wenn die Auflösung des Voxelgitters

zu klein gewählt wird. Dann können die Voxel einen zu großen Bereich der Szene abdecken, in welchem sich somit mehrere verschiedene Objekte befinden. Auch könnte die Information, dass in diesem Voxel ein Objekt enthalten ist, dort gespeichert werden, obwohl nur ein kleiner Teil des Voxels tatsächlich von einem Objekt belegt ist. Ist die Auflösung des Voxelgitters jedoch zu hoch, steigt der Aufwand des Voxelisierungsverfahrens ebenso wie die Laufzeit aller auf diesem Gitter aufbauenden Algorithmen, was die Performance stark beeinflussen kann.

3 Erstellung des Voxelgitters

In diesem Abschnitt wird ein Verfahren behandelt, welches eine dreidimensionale Szene, bestehend aus Polygonen, in eine Repräsentation als dreidimensionales, reguläres Voxelgitter transformiert. Das hier präsentierte Verfahren orientiert sich an dem von Thiedemann [7] verwendeten Ansatz, in dem die Informationen über die Position der Objekte in einem 2D-Texturatlas gespeichert werden. Die dort enthaltenen Koordinaten werden dann eingelesen und an entsprechender Stelle in das Voxelgitter eingesetzt. Diese Methode hat den Vorteil, dass das entstehende Voxelgitter unabhängig davon ist, von welchem Blickwinkel aus es erstellt wurde. Auch Objekte, die von diesem Blickpunkt aus durch andere Objekte verdeckt werden, werden in das Gitter eingetragen. Andere Methoden, wie die von Passalis [6] präsentierte, verwenden *Depth Peeling* zur Erstellung der Voxelstruktur. Im Gegensatz zu dieser Variante ist die Laufzeit des hier verwendeten Verfahrens jedoch unabhängig von der Tiefenkomplexität der Szene.

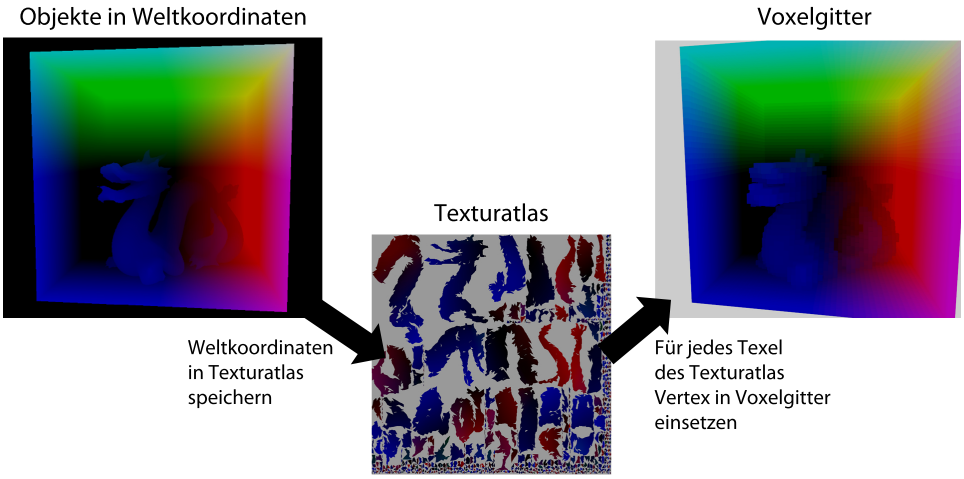


Abbildung 5: Ablauf der Voxelisierung mithilfe von Texturatlanten. Zuerst werden die Objekte in eine eigene Textur gerendert, sodass dort die Weltkoordinaten des Objekts enthalten sind. Anschließend wird für jeden Texel des Texturatlas ein Vertex erstellt und in das Voxelgitter eingesetzt.

Mithilfe dieses Verfahrens kann sowohl ein binäres, als auch ein mehrwertiges Voxelgitter erstellt werden. Eine binäre Darstellung des Voxelgitters lässt sich erzeugen, wenn die im Texturatlas gespeicherten Positionen in eine 2D-Textur übertragen werden. In dieser Textur werden die Bits des RGBA-Kanals dazu verwendet, das Vorhandensein einer Geometrie an einer bestimmten Tiefe im Voxelgitter zu codieren. Um eine mehrwer-

tige Darstellung des Voxelgitters zu verwenden, kann die Position aus dem Texturatlas in eine 3D-Textur gespeichert werden. In dieser lassen sich dann zusätzliche Informationen über Position, Farbe oder Normale der voxelisierten Geometrie speichern, was bei der nachfolgenden Berechnung von Nutzen sein kann. Beide Varianten werden in diesem Abschnitt behandelt, wobei für die spätere Nutzung im Beleuchtungsverfahren eine mehrwertige Voxelisierung verwendet wird. Dies begründet sich dadurch, dass mit der ständigen Weiterentwicklung der OpenGL-API die Unterstützung von 3D-Texturen immer weiter ausgebaut wird und die Verwendung dieser Variante unter OpenGL 4.4 daher einen modernen Ansatz liefert.

Somit besteht die Idee des hier behandelten Voxelisierungsverfahrens aus zwei Schritten: Im ersten Schritt werden die Weltpositionen der vorhandenen Geometrie in einen Texturatlas übertragen. Jedes Objekt der Szene besitzt einen eigenen Texturatlas, weswegen dieser Schritt für alle Objekte innerhalb der Szene ausgeführt wird. Im zweiten Schritt wird für jeden gültigen Texel des Texturatlas ein Punkt an entsprechender Stelle in das Voxelgitter eingesetzt. Einen Überblick über den Ablauf der Voxelisierung liefert Abbildung 5.

3.1 Erstellung des Texturatlas

Der erste Schritt des Voxelisierungsverfahrens besteht aus dem Anlegen eines *Texturatlas*. Ein Texturatlas wird verwendet, um die Oberfläche eines 3D-Modells in zweidimensionaler Darstellung abzubilden und in einer Textur zu speichern. In den meisten Fällen wird dabei der Farbwert an einem bestimmten Punkt des Objekts in der Textur vermerkt, um diesen später für die Berechnung der Beleuchtung zu verwenden. Für den Fall dieses Voxelisierungsverfahrens wird jedoch die Position in Weltkoordinaten in die Textur geschrieben, was zur Folge hat, dass die Oberfläche des Objekts diskretisiert wird. Um einen Texturatlas zu erstellen, ist es notwendig, dass ein Objekt bereits festgelegte Texturkoordinaten für jeden Vertex besitzt. Diese bestimmen, welche Stelle im Texturatlas welchen Vertex des Objekts repräsentiert. Die Abbildung eines Vertex auf einen Texel der Textur wird auch als *UV-Mapping* bezeichnet. Ein solches UV-Mapping kann sowohl automatisch im Programm generiert werden, als auch im Voraus durch externe Software wie z.B. Blender oder 3ds Max kreiert und anschließend in das Programm importiert werden. In dieser Arbeit wurde Blender verwendet, um die Abbildung der verwendeten Modelle auf UV-Koordinaten zu erstellen. Es ist möglich, einen Texturatlas für die gesamte Szene zu rendern, allerdings muss dieser dann jedes Mal neu berechnet werden, wenn Objekte der Szene hinzugefügt oder entfernt werden. Daher ist es vorzuziehen, für jedes Objekt einen eigenen Texturatlas zu verwenden.

Im Vorgang des Renderns in einen Texturatlas wird vorab für jedes Objekt eine Textur angelegt und an einen Framebuffer gebunden, damit in

diese Textur gerendert werden kann. Dabei muss die Auflösung des Texturatlas in Abhängigkeit der Auflösung des Voxelgitters angepasst werden. Ist die Auflösung des Texturatlas zu gering, entstehen Lücken im Voxelgitter. Ist die Auflösung jedoch zu hoch, werden mehrere Texel des Texturatlas auf einen Voxel abgebildet, was für unnötigen Rechenaufwand sorgt und die Performance negativ beeinflusst. Da die Textur Weltkoordinaten enthalten soll und diese auch negativ sein können, muss die angelegte Textur Floating-Point-Zahlen unterstützen. In der Renderschleife wird dann für jedes Objekt der entsprechende Framebuffer als Renderziel gewählt. Die Textur wird vorab mit einem festgelegten Wert mittels dem Befehl `glClear()` geleert. Dies ist notwendig, damit später ermittelt werden kann, welche Werte im Texturatlas gültige Positionen des Objekts sind.

Im Vertex Shader werden die Texturkoordinaten des Objekts, welche Werte im Intervall $[0, 1]$ annehmen können, in *Normalized Device Coordinates* umgewandelt, damit diese sich im Bereich von $[-1, 1]$ befinden. Zudem wird die Position des Vertex in Weltkoordinaten berechnet und an den Fragment Shader übergeben. Wird eine mehrwertige Voxelisierung durchgeführt, können hier zusätzliche Werte, wie z.B. die zugehörige transformierte Normale berechnet und übergeben werden. Hierbei werden diese Informationen in einem separaten Texturatlas gespeichert. Dafür ist es allerdings notwendig, mehrere Texturen pro Objekt anzulegen und dem Framebuffer-Objekt hinzuzufügen. In OpenGL 4.4 wird das Rendern in mehrere Texturen pro Framebuffer-Objekt unterstützt, was diese Option möglich macht. Im Fragment Shader werden lediglich die übergebenen Daten pro Fragment abgefragt und in ein oder mehrere Ergebnisbilder gespeichert.

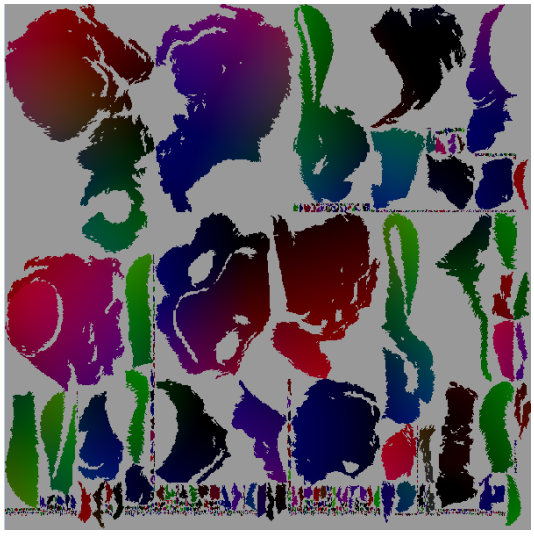


Abbildung 6: Texturatlas, in dem die Positionen eines Objekts in Weltkoordinaten in eine 2D-Textur gespeichert sind.

Nach dem in diesem Kapitel beschriebenen Verfahren liegen nun die Positionen (oder auch zusätzliche Informationen wie Normale und Farbe) der Vertices eines Objekts in Weltkoordinaten in einem oder mehreren Texturatlanten vor (siehe Abbildung 6). Diese Daten werden nun verwendet, um das Objekt an den entsprechenden Positionen in ein Voxelgitter einzusetzen.

3.2 Binäre Voxelisierung

Dieser Abschnitt befasst sich mit der Erstellung eines Voxelgitters aus dem zuvor berechneten Texturatlas. Das dabei entstehende Gitter enthält Informationen darüber, ob sich in einem Voxel ein Teil eines Objekts befindet oder nicht. Im Falle einer binären Voxelisierung enthält ein Voxel lediglich die Information 1, falls ein Objekt enthalten ist, oder 0, falls dies nicht der Fall ist. Die Tatsache, dass ein Voxel nur zwei verschiedene Werte annimmt, kann genutzt werden, um ein Voxelgitter mithilfe von Bits innerhalb eines Farbkanals einer RGBA-Textur zu erstellen. Diese Idee wird in den Verfahren von Eisemann [3] sowie Thiedemann [7] verwendet.

Um die binäre Voxelisierung umzusetzen, wird zunächst eine 2D-Textur angelegt, in welche das Voxelgitter gespeichert werden soll. Diese enthält Daten des Typs *unsigned Integer*, was es erlaubt, pro Texel 128 Bit (32 Bit pro Farbkanal) zur Verfügung zu haben. Diese 128 Bit werden genutzt, um die Tiefenposition im Voxelgitter zu simulieren, während die UV-Koordinaten der Textur den x- und y- Positionen entsprechen. Zusätzlich wird vorab eine 1D-Bitmaske erstellt und als eindimensionale Textur an den Shader übergeben. Diese erfüllt später die Funktion, den Tiefenwert eines Punktes in der 2D-Textur so abzubilden, dass in Abhängigkeit des Tiefenwerts das entsprechende Bit im Voxelgitter gesetzt wird. Die Abbildung wird so gewählt, dass der rote Farbkanal (also die ersten 32 Bits) eines Texels die vorderen 32 Tiefenwerte im Voxelgitter repräsentiert, der grüne Farbkanal (die darauf folgenden 32 Bits) die nächsten 32 Tiefenwerte usw. Somit sind mit diesem Verfahren maximal 128 unterschiedliche Tiefenwerte darstellbar. Des Weiteren wird für jeden Texel des Texturatlas ein Vertex generiert, um die Werte aus dem Texturatlas im Vertex Shader einzeln transformieren zu können.

Innerhalb der Renderschleife wird die Textur, in der das Voxelgitter gespeichert werden soll, als Renderziel gebunden. Eine Voxelisierungskamera wird angelegt, welche das Koordinatensystem des Voxelgitters definiert. Der Blickwinkel der Kamera kann identisch mit dem Blickwinkel sein, der für die finale Beleuchtung verwendet wird; er kann aber auch davon abweichen, solange die Objekte, die voxelisiert werden sollen, in dem Frustum liegen, das durch die Kamera aufgespannt wird. Zudem wird der Framebuffer angewiesen, eine logische OR-Operation für den anschließenden Rendervorgang zu verwenden. Dies wird mit den folgenden Anweisungen bewerkstelligt:

```
glLogicOp(GL_OR);
glEnable(GL_COLOR_LOGIC_OP);
```

Die Verwendung der OR-Operation sorgt dafür, dass die Tiefenwerte der Vertices nicht überschrieben werden, wenn mehrere Vertices die gleiche x- und y-Koordinate besitzen. Stattdessen wird lediglich ein einzelnes Bit innerhalb des Kanals pro Vertex gesetzt.

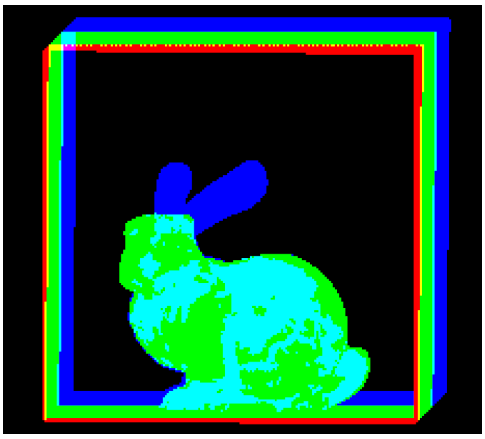


Abbildung 7: Ergebnistextur der binären Voxelisierung. Die Auflösung des Voxelgitters beträgt $256^2 \times 128$. Die Vertices werden so in das Voxelgitter eingesetzt, dass die Bits in den Farbkanälen der Textur die Tiefenposition im Voxelgitter codieren. Dabei sind rote Werte von der Voxelisierungskamera aus gesehen vorne, schwarze Werte (d.h. ein Voxel wurde in den Alpha-Kanal eingesetzt) hinten. Farbmischungen entstehen dadurch, dass mehrere Bits in einem Texel gesetzt werden können, wenn sich Voxel hintereinander befinden.

Im Vertex Shader der binären Voxelisierung wird zunächst die Position des aktuellen Vertex, die als RGBA-Wert im Texturatlas gespeichert wurde, aus diesem entnommen. Anschließend wird geprüft, ob dieser Punkt ein gültiger Punkt im Texturatlas ist und daher wirklich von dem Objekt stammt. Hierbei werden die Standardwerte aussortiert, mit denen der Texturatlas vorab geleert wurde. Damit nicht versehentlich Werte aussortiert werden, die tatsächlich zu dem Objekt gehören, sollte zum Leeren der Textur ein Wert verwendet werden, der nicht in der Reichweite der Werte liegt, die in den Texturatlas gespeichert werden sollen. Wird ein ungültiger Texel gefunden, erhält der zugehörige Vertex eine Position, die aus der Sicht der Voxelisierungskamera nicht mehr sichtbar ist und daher in den weiteren Schritten der OpenGL-Pipeline geclippt wird. Ist der Texel gültig, wird der enthaltene Wert mit einer orthografischen Projektion transformiert und an den Fragment Shader übergeben. Zudem wird die z-Koordinate der trans-

formierten Position auf das Intervall $[0, 1]$ abgebildet und ebenfalls an den Fragment Shader übertragen. Dieser verwendet nun die als 1D-Textur erstellte Bitmaske, um für die gegebene z-Position den Wert in der Bitmaske abzufragen, das entsprechende Bit zu setzen und in die Voxeltextur einzutragen. Da der Fragment Shader mit einer logischen OR-Operation arbeitet, können hier mehrere Bits pro Fragment gesetzt werden, anstatt sich gegenseitig zu überschreiben. Das bewirkt, dass auch Objekte, die aus Sicht der Voxelisierungskamera durch andere Objekte verdeckt sind, in das Voxelgitter eingetragen werden. Eine 2D-Textur, die als Ergebnis der binären Voxelisierung entstanden ist, lässt sich in Abbildung 7 betrachten.

3.3 Mehrwertige Voxelisierung

Will man in einem Voxel nicht nur die Sichtbarkeitsinformation eines Objekts, sondern gleichzeitig weitere Informationen über Position, Farbe etc. speichern, spricht man von einer mehrwertigen Voxelisierung. Dieser Abschnitt befasst sich mit einem solchen Voxelisierungsansatz auf Basis eines zuvor erstellten 2D-Texturatlas.

Die Daten, die in dem Voxelgitter enthalten sein sollen, lassen sich in einer 3D-Textur speichern. Eine 3D-Textur besitzt im Gegensatz zu 2D-Texturen eine Anzahl von *layers* oder *Schichten*, welche die Tiefe der Textur simulieren. Jede Schicht der 3D-Textur ist dabei eine eigene 2D-Textur und kann auch einzeln angesprochen werden (siehe Abbildung 8 für eine genauere Beschreibung). Unter OpenGL lassen sich mithilfe der Variable `gl_Layer` die Schichten der 3D-Textur im *Geometry Shader* und seit GLSL-Version 4.3 auch im Fragment Shader ansprechen. 3D-Texturen lassen sich ebenso an einen 3D-Framebuffer anheften, wodurch es ermöglicht wird, in diese zu rendern. Die so gefüllten Texturen können dann für weitere Verwendungszwecke auch in anderen Renderdurchläufen genutzt werden. Werden Mipmap-Level einer 3D-Textur erstellt, werden für jedes höhere Mipmap-Level Höhe, Breite und Tiefe der Textur halbiert. Alternativ können in OpenGL auch 2D-Array-Texturen verwendet werden. Diese besitzen denselben Aufbau wie 3D-Texturen; jedoch hat eine 2D-Array-Textur eine feste Anzahl Schichten in jedem Mipmap-Level und somit immer die gleiche Tiefe. Da diese Eigenschaft in einem späteren Schritt des Verfahrens benötigt wird, werden im Folgenden 2D-Array-Texturen verwendet.

Vor Beginn des Rendervorgangs wird ein 3D-Framebuffer angelegt und eine 2D-Array-Textur an diesen gebunden. Wenn neben der Position eines voxelisierten Objekts noch weitere Informationen im Voxelgitter gespeichert werden sollen, müssen zusätzliche Texturen angelegt und an diesen Framebuffer angehängt werden. Im Falle des Speicherns von Positionen und ähnlichen Werten muss die Textur Floating-Point-Zahlen unterstützen, da die Koordinaten auch negativ sein können. Die Auflösung der 2D-Array-Textur entspricht der Auflösung des Voxelgitters, das erstellt werden soll.

Anschließend wird für jeden Texel im Texturatlas ein Vertex erzeugt, damit dieser im Vertex Shader einzeln abgefragt und transformiert werden kann.

Während des Rendervorgangs wird der zuvor erstellte 3D-Framebuffer als Renderziel gewählt und der Texturatlas als `uniform`-Variable an den Shader übergeben. Die verwendete 3D-Textur wird vorab mit einem Wert geleert, welcher sich eindeutig von den Werten unterscheidet, mit denen das Gitter gefüllt werden soll. Das ermöglicht es, bei der späteren Verwendung des Voxelgitters leere und nicht-leere Voxel voneinander unterscheiden zu können, ohne eine zusätzliche Variable dafür setzen zu müssen. Nun werden die einzelnen Vertices als Punkte in die 2D-Array-Textur gerendert, um das vollständige Voxelgitter zu ermitteln.

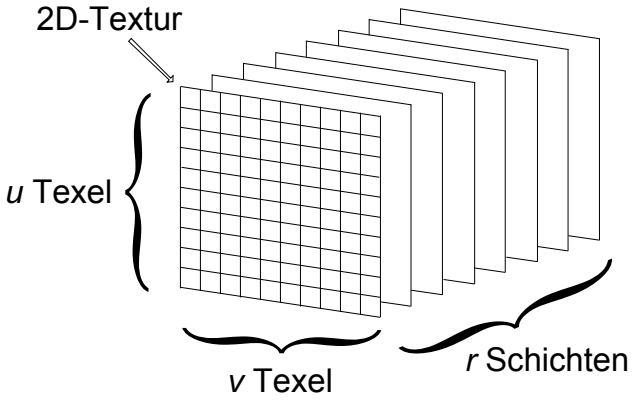
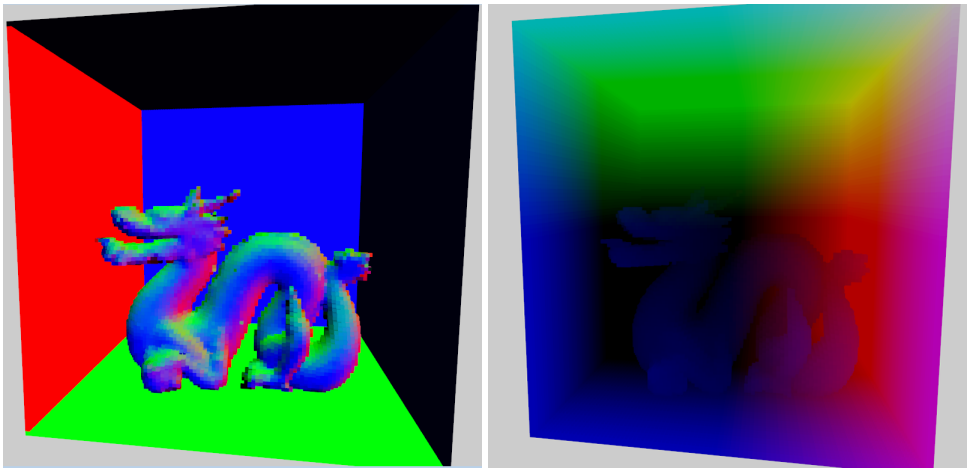


Abbildung 8: Aufbau einer 3D-Textur. Eine 3D-Textur besitzt r Schichten, die hintereinander angeordnet sind. Jede Schicht der 3D-Textur wird dabei durch eine eigene 2D-Textur mit $u \cdot v$ Texel repräsentiert. Somit sind in einer 3D-Textur insgesamt $u \cdot v \cdot r$ Texel enthalten, in die verschiedene Informationen gespeichert werden können.

Im Vertex Shader werden zuerst die im Texturatlas gespeicherte Position sowie gegebenenfalls weitere Informationen über Normale oder Farbe pro Vertex abgefragt. Wie beim Ansatz der binären Voxelisierung wird anschließend geprüft, ob dieser Texel ein gültiger Texel ist und somit tatsächlich von einem Objekt der Szene stammt. Da der Texturatlas vorher mit einem einheitlichen Wert geleert wurde, der außerhalb der Reichweite der darin gespeicherten Werte liegt, lässt sich dies durch eine simple Abfrage im Vertex Shader überprüfen. Ist der aktuell geprüfte Wert ungültig, wird dem zugehörigen Vertex eine Position zugewiesen, die vom Blickwinkel der Voxelisierungs-

kamera nicht sichtbar ist. Dieser Wert wird dann automatisch durch den weiteren Ablauf der OpenGL-Pipeline geclippt. Wird ein gültiger Wert gefunden, werden dessen Koordinaten mit einer orthografischen Projektion transformiert und in der Variable `gl_Position` gespeichert. Weiterhin wird die transformierte z-Position ausgelesen und zusammen mit den aus dem Texturatlas ausgelesenen Werten an einen *Geometry Shader* übergeben.

Zur weiteren Berechnung der Position der Vertices in der 2D-Array-Textur ist es notwendig, einen Geometry Shader der Grafikpipeline hinzuzufügen. Dieser kann nach dem Vertex Shader aufgerufen werden und besitzt die Funktion, zusätzliche Geometrie in Abhängigkeit der vorhandenen Vertices zur Laufzeit zu erzeugen oder zu entfernen. Zudem können hier bestimmte GLSL Built-In-Variablen wie z.B. `gl_Layer` angesprochen werden, welche die Zuweisung eines Vertex zu einer bestimmten Schicht der dreidimensionalen Textur ermöglicht. Da diese Funktionalität für dieses Verfahren notwendig ist, wird ein Geometry Shader verwendet.



(a) (b)

Abbildung 9: Ergebnisse einer mehrwertigen Voxelisierung. Die Auflösung des Voxelgitters beträgt 128^3. In (a) wurden die Normalen, in (b) die Position der Objekte in 3D-Weltkoordinaten in den Voxeln gespeichert.

Im Geometry Shader werden die Daten, die aus einem oder mehreren Texturatlanten ausgelesen wurden, unverändert an den Fragment-Shader übergeben. Die vom Vertex Shader übergebene z-Koordinate wird nun verwendet, um die Zuweisung des Vertex zu einer bestimmten Ebene der 2D-Array-Textur zu ermitteln. Die z-Position des Vertex befindet sich nach der orthografischen Projektion im Bereich $[-1, 1]$; die Variable `gl_Layer` nimmt jedoch nur Integer-Werte im Intervall $[0, n-1]$ an, wobei n die Anzahl der Schichten der Textur sei. Zur Skalierung der Werte in diesen Bereich

wird die z-Position zuerst in den Wertebereich $[0, 1]$ transformiert, mit $n-1$ multipliziert und zum Datentyp Integer gecastet.

Der Fragment Shader erhält die aus dem Texturatlas extrahierten Daten und setzt diese an entsprechender Stelle in eine oder mehrere 2D-Array-Texturen ein. Dabei werden die Werte pro Fragment automatisch in die vorher per `gl_Layer` festgelegten Schichten der Textur eingesetzt. Das Ergebnis des Renderpasses ist ein vom Blickpunkt unabhängiges Voxelgitter, in dem jeder Voxel genau einem Texel der dreidimensionalen Textur entspricht und somit mehrere Werte enthalten kann. Beispiele für ein Voxelgitter, das durch die mehrwertige Voxelisierung entstanden ist, lassen sich in Abbildung 9 finden.

4 Traversierung der Voxelstruktur

Dieser Abschnitt beschreibt ein Verfahren, welches die Voxel eines Voxelgitters nach einem Algorithmus durchläuft. Dies wird dazu verwendet, einen Schnitttest eines Strahls mit diesen Voxeln durchzuführen. Hierbei soll bestimmt werden, ob ein Strahl, der von einem beliebigen Punkt in der Szene aus gesendet wird, entlang seiner Richtung auf ein Objekt in der Szene stößt. Diese Information kann verwendet werden, um zu bestimmen, ob Objekte von einer Position aus sichtbar sind oder nicht, was unter Anderem bei der Berechnung globaler Beleuchtung von Bedeutung ist.

Im Folgenden wird die Implementierung eines Algorithmus behandelt, der eine Voxelstruktur anhand der Position und Richtung eines ausgehenden Strahls traversiert und dabei ermittelt, ob ein Objekt getroffen wird. Des Weiteren wird ermittelt, an welcher Position in der Welt sich dieses Objekt befindet. Dies ist für spätere Verwendungszwecke notwendig. Anschließend wird die Verwendung einer Mipmap-Hierarchie der Voxelstruktur für dieses Verfahren angesprochen, welche die Laufzeit des Algorithmus zusätzlich verbessern kann. In beiden Abschnitten wird sich auf die Implementierung unter der Verwendung von 3D-Texturen beschränkt; somit ist eine vorausgehende mehrwertige Voxelisierung notwendig.

4.1 Voxel-Traversierungs-Algorithmus

In diesem Abschnitt wird ein Algorithmus zum Traversieren eines Voxelgitters entlang einer bestimmten Richtung behandelt. Der hier beschriebene Algorithmus baut auf dem *digital differential analyzer*-Algorithmus (*DDA*) auf. Dieser ist ein typischer, in der Computergrafik genutzter Algorithmus, der vor allem zum Rasterisieren von Linien im zweidimensionalen Raum verwendet wird. Von Fujimoto [4] wurde dieser Algorithmus auf den dreidimensionalen Raum erweitert, was es ermöglicht, diesen auf Strahlen in einem Raytracing-Verfahren anzuwenden; das dort beschriebene Verfahren wird als 3D-DDA-Algorithmus bezeichnet. Amanatides [1] stellt in seinem Paper ein Verfahren vor, welches ebenfalls eine Voxel-Traversierung auf Basis des DDA-Algorithmus im dreidimensionalen Bereich darstellt. Dabei werden bei der Traversierung innerhalb der Hauptschleife die Anzahl der Anweisungen auf nur wenige Floating-Point- und Integer-Vergleiche sowie Additionen beschränkt, um eine Beschleunigung des Algorithmus herbeizuführen. Im Folgenden wird der Algorithmus erläutert, der für dieses Verfahren der Voxelbasierten globalen Beleuchtung benutzt wird. Dabei werden an bestimmten Stellen in der Welt Strahlen erzeugt, die in verschiedene Richtungen gehen. Der Algorithmus wird in dem Verfahren also verwendet, um entlang der Strahlenrichtung die Werte innerhalb des Voxelgitters abzufragen und so Schnittpunkte zwischen dem Strahl und Objekten der Szene feststellen zu können.

Um den hier beschriebenen Algorithmus ausführen zu können, ist das Vorhandensein einer 3D-Textur oder 2D-Array-Textur als Repräsentation des Voxelgitters notwendig. Dabei ist es erforderlich, dass die gefüllten Voxel von den leeren Voxeln eindeutig unterscheidbar sind. Da eine mehrwertige Voxelisierung verwendet wird, sollten die leeren Voxel also einen Wert besitzen, welcher von den Werten in den gefüllten Voxeln abweicht. Das wird in diesem Fall dadurch bewerkstelligt, dass die Voxel im 3D-Gitter Informationen über die Position des Voxels in Weltkoordinaten enthalten. Die 3D-Textur wird vor dem Erstellen des Voxelgitters mit einem Wert geleert, den kein Objekt der Szene in Weltkoordinaten annimmt. Dadurch wird vermieden, dass Voxel versehentlich als leer bestimmt werden, was die Ergebnisse des Algorithmus stark verfälschen kann.

Der Ablauf des Algorithmus beginnt mit dem Bestimmen der Ursprungsposition $\vec{u}$ und Richtung $\vec{v}$ des Strahls, der das Voxelgitter traversiert. Die Position entlang eines Strahl r lässt sich anhand seiner Richtung und seines Ursprungs durch folgende Gleichung bestimmen:

$$\vec{r} = \vec{u} + t\vec{v} \tag{3}$$

Der Parameter t gibt an, an welcher Position des Strahls man sich aktuell befindet. t liegt im Intervall $[0, 1]$, was bedeutet, dass der Wert für $t = 0$ die Anfangsposition des Strahls bestimmt, während beim Einsetzen von $t = 1$ in die Gleichung der Endpunkt des Strahls als Ergebnis geliefert wird. Der Abstand, in dem die Positionen entlang des Strahls abgefragt werden, muss ebenfalls ermittelt und an die Größe der Voxel angepasst werden. Innerhalb der Hauptschleife wird das Voxel abgefragt, welches an der aktuellen Position des Strahls zu finden ist. Ist dieses Voxel leer, wird die aktuelle Position entlang des Strahles nach vorne verschoben, damit im nächsten Iterationsschritt ein neues Voxel überprüft werden kann. Befindet sich jedoch Information an der Stelle des Voxels und ist dieser somit nicht leer, wird ein Treffer vermerkt und die enthaltene Information aus dem Voxel entnommen, um sie im weiteren Verlauf des Programms zu verwenden. Der Algorithmus bricht ab, wenn entweder ein gefülltes Voxel gefunden wurde oder die Anzahl der Schritte entlang des Strahls zu groß ist. Es ist möglich, die verwendete Schrittzahl dabei beliebig einzustellen, was dafür genutzt werden kann, die Performance des Algorithmus anzupassen. Ein Beispielablauf des Algorithmus wird in Abbildung 10 noch einmal verdeutlicht.

Da das Voxelgitter als dreidimensionale Textur gespeichert ist, kann diese als uniform-Variable in einen Shader geladen und verwendet werden. Dadurch ist es möglich, den Algorithmus vollständig innerhalb des Fragment Shaders zu implementieren.

Zu Beginn des Algorithmus wird eine Variable des Typs `bool` initialisiert. In dieser soll im Laufe des Algorithmus vermerkt werden, ob ein Objekt

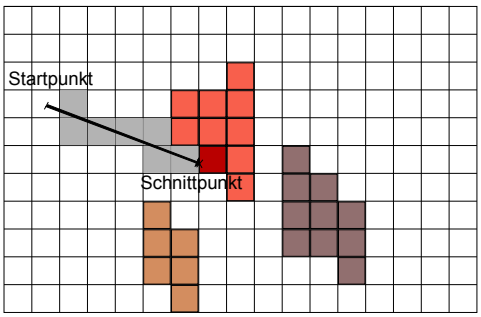


Abbildung 10: Ablauf der Voxel-Traversierung in 2D-Ansicht. Der Startpunkt des Strahls wird bestimmt und an die entsprechende Position im Voxelgitter gesetzt. Von dort aus werden nun die Voxel entlang der Richtung des Strahls traversiert. Das erste Voxel wird auf seinen Inhalt überprüft. Ist dieses Voxel leer, wird das nächste noch nicht geprüfte Voxel betrachtet, das in Richtung des Strahls liegt. Die geprüften Voxel sind in der Grafik grau markiert. Der Algorithmus fährt solange fort, bis ein gefülltes Voxel getroffen oder die vorher definierte Anzahl der Schritte entlang des Strahls überschritten wird. In diesem Fall wurde ein gefülltes Voxel entlang des Strahls gefunden (in der Grafik dunkelrot markiert) und somit ein Schnittpunkt des Strahls mit einem Objekt der Szene festgestellt.

entlang des Strahls getroffen wurde oder nicht. Anschließend werden Startpunkt und Richtung des Strahls ermittelt, der das Voxelgitter durchlaufen soll. Die Weltposition des Startpunkts muss dabei für jede Koordinate in den Bereich $[0, 1]$ transformiert werden, damit ein späterer Texturzugriff anhand der aktuellen Position entlang des Strahls ermöglicht wird. Die dafür notwendige Transformation kann vereinfacht werden, wenn sichergestellt wird, dass sich die xyz-Koordinaten der Objekte der Szene jeweils bereits in Weltkoordinaten im Bereich $[-1, 1]$ befinden. Anschließend wird bestimmt, in welchem Abstand die Werte entlang des Strahls abgefragt werden müssen. Dieser Abstand sollte dabei in Abhängigkeit der Voxelgröße und der Richtung des Strahls gewählt werden.

Des Weiteren ist es wichtig, *Selbstverdeckungen* von Objekten zu vermeiden. Dieses Problem kann entstehen, wenn der Traversierungsalgorithmus an der exakten Startposition des Strahls beginnt. Ist dies der Fall, wird im ersten Iterationsschritt des Algorithmus das Voxel, in dem der Strahl startet, auf dessen Inhalt überprüft. Wenn der Strahl jedoch bewusst von einem bestehenden Objekt aus gesendet wird, kann dieses fälschlicherweise als getroffenes Objekt festgelegt werden. Der Algorithmus bricht dann schon nach der ersten Iteration ab und ermittelt das Objekt, welches den Strahl sendet, als Treffer entlang des Strahls. Um solche Selbstverdeckungen zu

vermeiden, muss ein bestimmter Offset gewählt werden, sodass der erste Voxel, der auf seinen Inhalt überprüft wird, nicht derjenige ist, bei dem der Strahl beginnt, sondern etwas weiter entlang der Strahlenrichtung liegt.

Sobald der tatsächliche Startpunkt auf dem Strahl ermittelt wurde, beginnt die Iteration in einer bestimmten Schrittgröße entlang des Strahls. Dabei wird vorab eine Variable `steps` initialisiert, welche festlegt, wie viele Iterationsschritte der Algorithmus maximal durchführen soll. Dies kann Rechenzeit sparen, wenn nach vielen Iterationen noch kein Treffer gefunden wurde; allerdings wird dadurch die effektive Reichweite des Strahls eingeschränkt, was bedeutet, dass unter Umständen ein Treffer im Voxelgitter verfehlt wird. In jedem Iterationsschritt wird dann zuerst der Wert an der aktuellen Strahlenposition in der 3D-Textur, die das Voxelgitter enthält, abgefragt. Beim Aufruf des Befehls `texture()` für 3D-Texturen muss die Tiefenkoordinate der Textur dabei als Wert zwischen $[0, n-1]$ übergeben werden, wobei n die Anzahl der Schichten der Textur sei. Somit muss die z-Koordinate der aktuellen Position vorher in diesen Wertebereich transformiert werden. Als Ergebnis der Texturabfrage erhält man den Wert, der in dem Voxel an der Stelle gespeichert wurde, die momentan entlang des Strahls überprüft wird. Dieser Wert enthält nun entweder die Information über die Weltposition eines voxelisierten Objekts, oder aber den default-Wert, welcher vor Erstellung des Voxelgitters in die Textur eingetragen wurde. Durch eine `if`-Abfrage lässt sich nun unterscheiden, ob es sich um einen gültigen Wert handelt und somit ein Objekt getroffen wurde, oder ob der Wert ungültig und das Voxel somit leer ist. Wurde ein Objekt getroffen, wird dies mithilfe der zuvor initialisierten `bool`-Variable im Programm vermerkt und die Position des Treffers für spätere Verwendungszwecke gespeichert. Daraufhin wird die Schleife verlassen. Befindet sich kein Objekt an Stelle des Voxels, wird die zu überprüfende Position in Abhängigkeit der Richtung des Strahls verschoben und die Schleife fortgesetzt, bis ein Objekt gefunden oder die festgelegte Anzahl an Iterationen überschritten wird.

4.2 Nutzung einer Mipmap-Hierarchie

Im Folgenden wird die Erstellung und Anwendung einer Datenstruktur auf Basis von Mipmaps erläutert, die es ermöglicht, die Traversierung eines Voxelgitters durch den zuvor beschriebenen Algorithmus zu beschleunigen. Das Ziel bei der Verwendung einer Mipmap-Struktur ist es, die Existenz von Objekten innerhalb eines größeren Bereichs der Voxelstruktur feststellen zu können. Befinden sich in einem großen Teilbereich des Voxelgitters keine Objekte, kann dieser Bereich in einem einzigen Iterationsschritt übersprungen werden, was die Traversierung des Voxelgitters beschleunigen kann.

Die hier beschriebene Methode kann auf ein Voxelgitter angewandt werden, welches durch eine mehrwertige Voxelisierung entstanden ist und in einer 3D-Textur oder 2D-Array-Textur gespeichert wurde. Im Folgenden

wird sich dabei auf die Verwendung einer 2D-Array-Textur beschränkt. Die Idee bei der Erstellung der Mipmap-Hierarchie ist es, die verschiedenen Mipmap-Level einer dreidimensionalen Textur zu nutzen, um die Existenz von Geometrie auf verschiedenen Levels anzeigen zu können. Die Verwendung einer 2D-Array-Textur hat zur Folge, dass die Anzahl der Texel in x- und y-Richtung auf jedem nächsthöheren Level halbiert werden, während die Anzahl der Texel in z-Richtung unverändert bleibt. Das hat zur Folge, dass vier Texel (zwei in x- und zwei in y-Richtung) eines Mipmap-Levels auf einen Texel des nächsthöheren Mipmap-Levels abgebildet werden müssen. Dazu werden jeweils vier benachbarte, als Texel repräsentierte Voxel auf jedem Mipmap-Level abgefragt, um sie als gefüllte oder leere Voxel zu identifizieren. Die Ergebnisse der einzelnen Texel werden miteinander durch eine Oder-Operation verknüpft und in das nächsthöhere Mipmap-Level eingetragen. Das heißt, sobald mindestens eines der vier Texel einer Mipmap-Ebene mit einem gültigen Wert gefüllt ist, wird auch das korrespondierende Texel der höheren Mipmap-Ebene gefüllt. Auf der höchsten Mipmap-Ebene besitzt die Textur jeweils nur ein Texel in x- und y-Richtung, während die Anzahl der Texel in z-Richtung identisch zu der Anzahl in der ursprünglichen Textur ist. Das macht es möglich, in Richtung der Tiefe genauer zu entscheiden, ob ein gefülltes Voxel getroffen wird. Der Aufbau der Mipmap-Hierarchie wird als 2D-Beispiel in Abbildung 11 erläutert.

Unter der OpenGL-API gibt es die Möglichkeit, Mipmap-Level einer Textur automatisch generieren zu lassen; allerdings werden die Texel dabei nicht durch Oder-Operationen verknüpft, was bedeutet, dass die Mipmap-Texturen in diesem Fall manuell erstellt werden müssen. Dazu müssen für jedes Mipmap-Level Texturen initialisiert und an einen eigenen Framebuffer gebunden werden. Für jedes Level wird dann das Ergebnis der Operation in den Shadern berechnet und in die Framebuffer-Textur geschrieben. An den Vertex Shader werden dafür die Vertices eines *Screen-Filling Quad* übergeben. Das ist ein Rechteck, welches den Zweck erfüllt, den Bereich des Framebuffers vollständig zu bedecken, damit in jedes Texel der Framebuffer-Textur geschrieben werden kann. Da eine dreidimensionale Textur verwendet wird, muss ein solches Rechteck für jede Schicht der 2D-Array-Textur erstellt werden. Im Vertex Shader werden lediglich die Position und Texturkoordinaten der Vertices in den benötigten Wertebereich transformiert und an einen Geometry Shader übergeben. Der Geometry Shader besitzt die Funktion, die übergebenen Vertices anhand der jeweiligen z-Koordinaten den Schichten der 2D-Array-Textur zuzuordnen, und gibt die aus dem Vertex Shader erhaltene Texturkoordinate an den Fragment Shader weiter. Diese wird dort verwendet, um die Werte vier benachbarter Texel des vorherigen Mipmap-Levels abzufragen. Für jedes dieser Texel wird nun bestimmt, ob sie mit gültigen Werten gefüllt sind und somit Geometrie im entsprechenden Voxel vorhanden ist. Findet sich unter den vier Texeln mindestens eines mit gültigem Wert, wird das Texel der aktuellen Mipmap-Ebene ebenfalls mit diesem

Wert gefüllt. Diese Bedingung wird durch eine `OR`-Operation zwischen den vier Texeln überprüft.

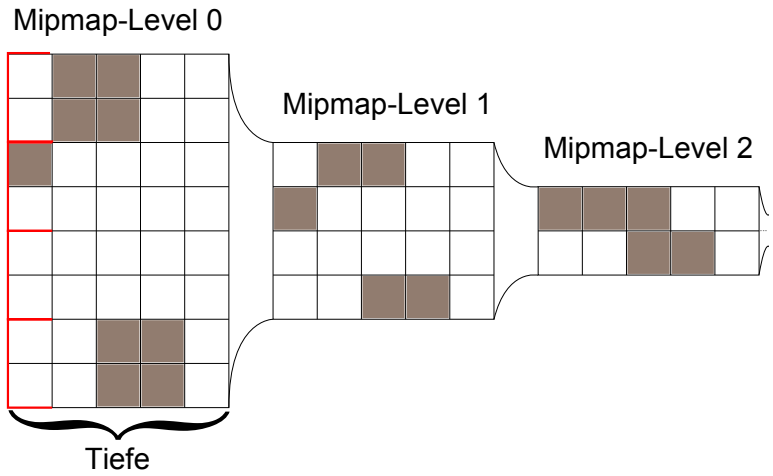


Abbildung 11: Erstellung der Mipmap-Hierarchie (hier in zweidimensionalen Raum beschrieben). Die ursprüngliche Textur (Mipmap-Level 0) besitzt 8 Texel in der Breite und 5 Texel in der Tiefe. Da die Höhe der Textur hier nicht betrachtet wird, werden jeweils 2 benachbarte Texel (rot markiert) auf ihren Inhalt überprüft und die Ergebnisse miteinander durch eine logische Oder-Operation zusammengefasst. Die Tiefe der Textur wird auf jedem Level beibehalten. Das Ergebnis der Operation wird in die Textur des nächsten Mipmap-Levels geschrieben.

Führt man die Shader für jedes Mipmap-Level aus, erhält man eine Mipmap-Hierarchie. Auf jedem Level der Hierarchie kann ermittelt werden, ob in einem bestimmten Bereich, den das jeweilige Level abdeckt, ein oder mehrere Objekte vorhanden sind. Diese Information kann nun zur Traversierung der Voxelstruktur genutzt werden. Dazu wird der in Abschnitt 4.1 beschriebene Algorithmus angepasst. Anstatt den Inhalt der Voxelstruktur für jedes Voxel, der entlang eines Strahls liegt, einzeln abzufragen, kann diese Abfrage auf einem höheren Mipmap-Level durchgeführt werden, um direkt einen größeren Bereich des Voxelgitters abzudecken. Findet sich in diesem Bereich kein gefülltes Voxel, kann der gesamte Bereich übersprungen werden. Wird festgestellt, dass innerhalb des Bereichs ein oder mehrere Objekte vorliegen, wird das Mipmap-Level reduziert und die Überprüfung auf dem niedrigeren Mipmap-Level durchgeführt. Der Algorithmus stoppt, wenn die Anzahl der Schritte entlang des Strahls überschritten wird oder ein gefülltes Voxel auf dem Mipmap-Level mit der höchsten Auflösung gefunden und somit ein Schnittpunkt zwischen dem Strahl und einem Objekt festgestellt wird.

Die Verwendung einer Mipmap-Hierarchie für die Voxel-Traversierung besitzt den Vorteil, dass dadurch große Bereiche, in denen sich keine Objekte befinden, schnell übersprungen werden können. Besitzt eine Szene wenig Geometrie oder ist die Auflösung des Voxelgitters hoch, kann es vorkommen, dass nur wenige Voxel gefüllt sind. In diesem Fall können viele Voxel übersprungen und Rechenzeit gespart werden. Allerdings muss die Hierarchie für dynamische Szenen ständig angepasst und die Inhalte der Mipmap-Level somit für jeden Frame neu berechnet werden. Hierfür wird viel Rechenzeit benötigt, da für jedes Mipmap-Level ein Vertex, ein Geometry und ein Fragment Shader ausgeführt werden müssen. Dieser Aufwand könnte verringert werden, wenn eine Zuweisung eines Vertex zu einer Schicht der 3D-Textur schon im Vertex Shader möglich wäre. Unter OpenGL 4.4 ist diese Zuweisung jedoch nur im Geometry Shader möglich. Da der Geometry Shader hier jedoch keine weitere Funktion erfüllt, würde durch diese Anpassung die Verwendung eines Geometry Shaders vollständig entfallen. Ein weiterer Nachteil ist es, dass die Erstellung einer 3D-Mipmap-Hierarchie für jede Ebene zusätzlichen Speicherplatz verwendet, was gerade bei komplexen Anwendungen, die ohnehin viel Speicherplatz benötigen, zu einem Problem führen kann. Abschließend sollte erwähnt werden, dass aufgrund des zusätzlichen Aufwandes, den die Erstellung der Mipmap-Hierarchie für 3D-Texturen darstellt, diese Methode der Voxel-Traversierung nicht im weiteren Verfahren verwendet wird.

5 Globale Beleuchtung

Mithilfe der heutzutage verfügbaren Mittel ist es unmöglich, die exakte Beleuchtung einer Szene in einer Echtzeit-Anwendung zu berechnen. Aus diesem Grund müssen für solche Anwendungen im Voraus viele Annahmen über die Szene getroffen und Vereinfachungen durchgeführt werden, um eine realistische Beleuchtung anzunähern. In diesem Abschnitt wird sich damit befasst, wie eine globale Beleuchtung in einer Echtzeit-Anwendung simuliert werden kann. Dabei wird zuerst auf die Berechnung der direkten Beleuchtung der Objekte einer Szene durch eine oder mehrere Lichtquellen eingegangen. Im Anschluss wird ein Verfahren zur Erstellung von *Reflective Shadow Maps* (RSM) erläutert, welches von Dachsbacher und Stamminger [2] eingeführt wurde. Diese werden benutzt, um die indirekte Beleuchtung einer Szene zu berechnen. Darauf folgend wird beschrieben, wie die Verwendung von Reflective Shadow Maps durch die Nutzung einer Voxelstruktur optimiert werden kann, um Artefakte zu vermeiden und die indirekte Beleuchtung nur auf eine bestimmte Reichweite zu beschränken.

5.1 Direkte Beleuchtung

Für die Berechnung der gesamten globalen Beleuchtung ist es zuerst notwendig, die direkte Beleuchtung von Objekten durch Lichtquellen zu ermitteln. Dabei wird im Folgenden nur auf die Beleuchtung *diffuser* Materialien eingegangen. Diffuses Material zeichnet sich dadurch aus, dass es auftreffende Strahlen einer Lichtquelle in viele verschiedene Richtungen reflektiert. Dabei ist die Leuchtdichte für jede Richtung gleich und somit unabhängig von der Position des Betrachters (Lambertsches Gesetz). Im Gegensatz dazu besitzen *spiegelnde* Materialien die Eigenschaft, eingehendes Licht nur in genau eine Richtung zu reflektieren. Somit ist deren Beleuchtung abhängig vom Blickwinkel auf die Oberfläche. Werden diffuse Oberflächen beleuchtet, erscheinen sie matt, während Oberflächen mit spiegelnden Eigenschaften an bestimmten Stellen einen Glanz hervorrufen.

Für die direkte Beleuchtung der Szene wird daher ein Shading verwendet, das auf dem Lambertschen Gesetz zur Berechnung diffuser Reflexion basiert. Die Strahlungsstärke einer diffusen Oberfläche hängt dabei vom Cosinus des Winkels zwischen der Oberflächennormale und dem Vektor zwischen der Lichtposition und der Position an der Oberfläche des Objektes ab. Sei $\vec{n}$ die Oberflächennormale des Objekts und $\vec{l}$ der Vektor zwischen Lichtposition und Objekt, dann kann der Cosinus des Winkels θ zwischen den beiden Vektoren durch

$$cos\theta = \frac{\vec{n} \cdot \vec{l}}{|\vec{n} \cdot \vec{l}|} \tag{4}$$

berechnet werden. Gilt für beide Vektoren $|\vec{n}| = |\vec{l}| = 1$, so kann die Berechnung des Winkels auf

$$cos\theta = \vec{n} \cdot \vec{l} \tag{5}$$

vereinfacht werden, was somit lediglich die Berechnung des Skalarprodukts zwischen den beiden Vektoren zur Folge hat.

Um nun die direkte Beleuchtung L_d an einem Punkt der Oberfläche eines Objekts zu bestimmen, wird die diffuse Materialfarbe M_d eines Objektes mit dem Cosinus für jede Lichtquelle der Szene durch

$$L_d = \sum_{i=1}^{n} M_d cos\theta_i \tag{6}$$

berechnet, wobei n die Anzahl der verwendeten Lichtquellen bezeichnet.

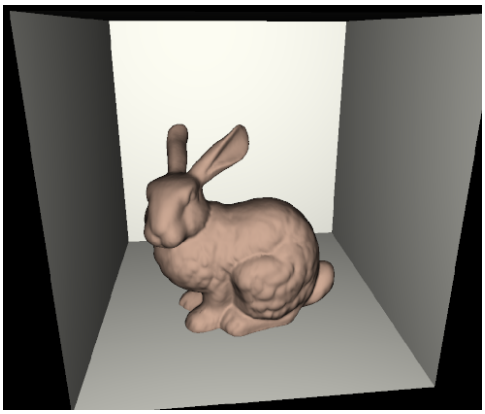


Abbildung 12: Direkte Beleuchung diffuser Objekte nach dem Lambertschen Gesetz.

Die direkte Beleuchtung kann in OpenGL pro Fragment im Fragment Shader berechnet werden, wenn die Oberflächennormalen für jeden Vertex des Objekts durch den Vertex Shader übergeben werden. Abbildung 12 zeigt die berechnete direkte Beleuchtung einer Szene für diffuse Materialien nach der hier beschriebenen Methode. Die Ergebnisse der direkten Beleuchtung können in einer 2D-Textur, die zuvor an einen Framebuffer gebunden wurde, gespeichert werden, um diese für die Berechnung der indirekten Beleuchtung weiterzuverwenden. Ebenfalls ist es möglich, die direkte Beleuchtung zusammen mit der in Abschnitt 3.3 beschriebenen mehrwertigen Voxelisierung zu berechnen und dabei die resultierenden Werte in die einzelnen Voxel einzutragen. Dabei muss jedoch beachtet werden, dass bei der mehrwertigen Voxelisierung die Möglichkeit besteht, dass mehrere Daten in das gleiche Voxel eingetragen werden und bestehende Werte überschrieben werden. Ist das der Fall, gehen Informationen über die direkte Beleuchtung

verloren und Berechnungen, die auf diesen Informationen aufbauen, werden verfälscht. Diese Methode ist daher nur vorzuziehen, wenn sichergestellt werden kann, dass in kein Voxel mehrfach Daten eingetragen werden.

5.2 Reflective Shadow Maps

Eine Methode zum Erzeugen glaubwürdiger indirekter Beleuchtung ist das Erstellen von Reflective Shadow Maps (RSM), die von Dachsbacher und Stamminger [2] eingeführt wurden. Die Idee dahinter ist es, die Szene aus Sicht der Lichtquelle zu rendern, um indirekte Beleuchtung anhand den von der Lichtquelle aus sichtbaren Objekten zu berechnen. Dies geschieht unter der Annahme, dass nur Objekte, die direkt von der Lichtquelle aus sichtbar sind, Licht an die anderen Objekte der Szene abgeben können und somit Ausstrahler indirekten Lichts sind. Die Erstellung einer Reflective Shadow Map funktioniert ähnlich wie die einer gewöhnlichen Shadow Map und wird durch die Verwendung von Texturen realisiert. Während in einer Shadow Map nur die Tiefe eines Objekts gespeichert wird, werden in eine Reflective Shadow Map auch Informationen über die Position, Normale und direktes Licht der Objekte eingetragen. Die Werte aus der Reflective Shadow Map werden dann über ein *Sampling*-Verfahren abgefragt und zur Berechnung der indirekten Beleuchtung verwendet.

Eine gewöhnliche Shadow Map wird erstellt, indem die Objekte einer Szene in Weltkoordinaten in ein Koordinatensystem übertragen werden, das die Szene aus Sicht der Lichtquelle beschreibt. Dazu werden die Weltkoordinaten durch eine perspektivische Matrix in das Koordinatensystem der Lichtquelle projiziert. Dies hat zur Folge, dass nur noch die Objekte sichtbar sind, die von der Lichtquelle zu erkennen sind und somit direkt beleuchtet werden.

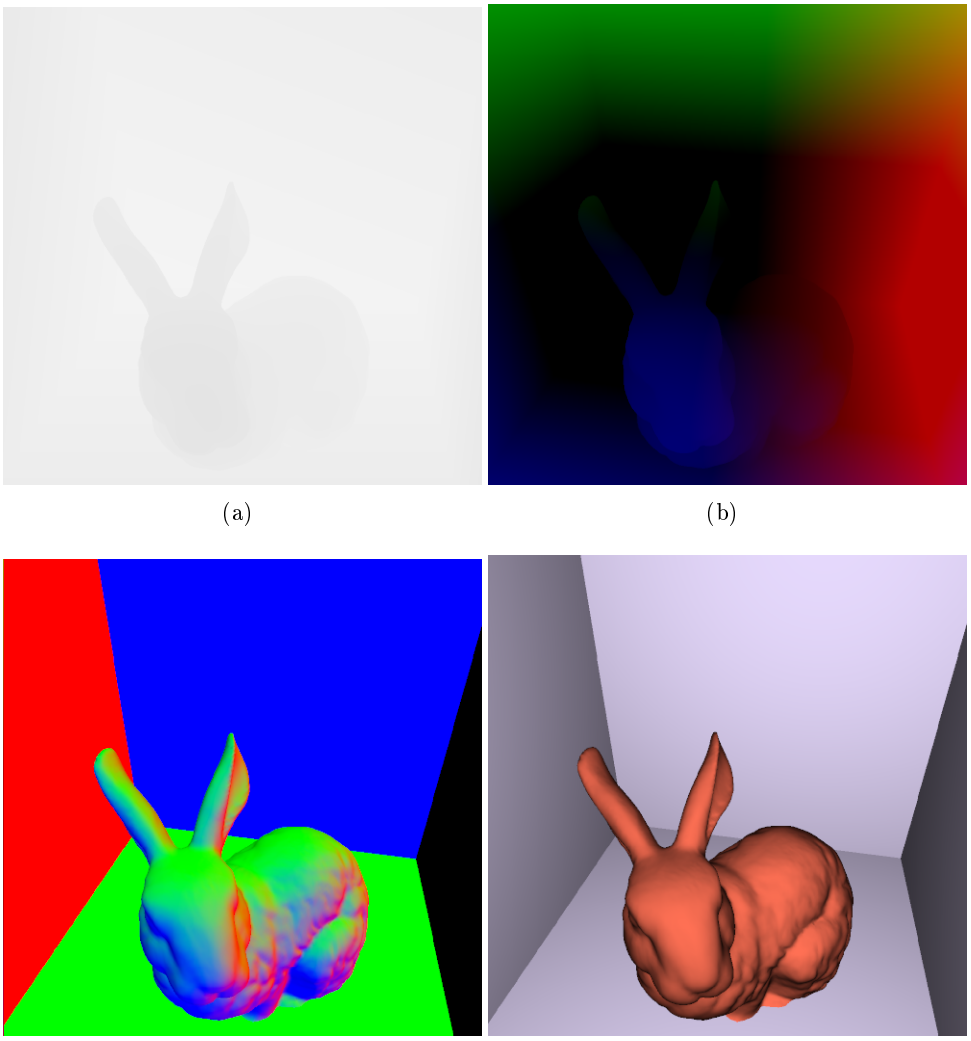
Die Erstellung einer Reflective Shadow Map gleicht bis zu diesem Punkt der Erstellung einer gewöhnlichen Shadow Map. Der Unterschied der Verfahren befindet sich darin, dass in eine Shadow Map die Tiefe der sichtbaren Objekte gespeichert wird, um später feststellen zu können, ob sich ein Objekt im Schatten der Lichtquelle befindet oder nicht. Hingegen werden in eine Reflective Shadow Map verschiedene Daten der von der Lichtquelle aus sichtbaren Objekte eingetragen. Für die Verwendung einer Reflective Shadow Map in diesem globalen Beleuchtungsverfahren werden die Position des Objekts in Weltkoordinaten sowie dessen Normale und die direkte Beleuchtung an der jeweiligen Stelle gespeichert. Dabei werden die gespeicherten Daten verwendet, um daraus die indirekte Beleuchtung einer Szene zu ermitteln.

Um eine Reflective Shadow Map unter OpenGL zu erstellen, ist es notwendig, einen Framebuffer zu initialisieren, mit dem in nur einem Renderpass in mehrere Texturen geschrieben werden kann. Das ermöglicht das Speichern von Weltkoordinaten, Normalen und direkter Beleuchtung in je-

weils separaten Texturen. Die verwendete Art der Textur ist dabei abhängig vom Typ der Lichtquelle. Ist die Lichtquelle ein Spotlight, kann das Ergebnis in eine 2D-Textur gerendert werden. Für Punktlichtquellen sind jedoch *Cube Maps* geeignet. Cube Maps bestehen aus einer 2D-Textur für jede Seite eines Würfels und somit insgesamt 6 Texturen. Da eine Punktlichtquelle in jede beliebige Richtung Licht aussendet, kann eine solche Cube Map um die Punktlichtquelle gelegt werden, damit alle beleuchteten Objekte in dieser festgehalten werden können. Weil ein Spotlight nur in eine Richtung Licht ausstrahlt, genügt in diesem Fall eine 2D-Textur, die die Objekte entlang dieser Richtung speichert. Sobald alle Texturen initialisiert wurden, muss für jede Lichtquelle anhand ihrer Position eine View Matrix angelegt werden, um die Vertices der Objekte in das Koordinatensystem der Lichtquelle übertragen zu können. Für jedes Objekt der Szene wird nun die Grafik-Pipeline durchlaufen. Im Vertex Shader werden die Position und Normale pro Vertex in Weltkoordinaten transformiert und an den Fragment Shader übergeben. Weiterhin wird die Position des Vertex mithilfe der zuvor erstellten View Matrix und einer perspektivischen Projektion in das Koordinatensystem aus Sicht der Lichtquelle transformiert und das Ergebnis der GLSL Built-In Variable
`gl_Position` zugewiesen. Das hat zur Folge, dass die Berechnungen des Fragment Shaders für jedes Fragment aus der Sicht der Lichtquelle durchgeführt werden. Der Fragment Shader erhält die Weltkoordinate und Normale des Fragments und setzt die Werte in das entsprechende Framebuffer-Attachment ein, um die jeweiligen Texturen zu füllen. Des Weiteren werden die übergebenen Werte verwendet, um die direkte Beleuchtung pro Fragment wie in Abschnitt 5.1 beschrieben zu berechnen und ebenfalls in eine Framebuffer-Textur einzutragen. Das Ergebnis der Erstellung der Reflective Shadow Map sind drei Texturen für Weltkoordinate, Normale und direktes Licht der Objekte einer Szene aus Sicht der Lichtquelle (siehe Abbildung 13). Diese Informationen können nun benutzt werden, um die Berechnung der indirekten Beleuchtung einer Szene zu ermitteln.

5.3 Indirekte Beleuchtung

Im Folgenden wird behandelt, wie die in Abschnitt 5.2 erstellten Reflective Shadow Maps genutzt werden können, um eine glaubwürdige indirekte Beleuchtung der Szene zu berechnen. Dabei wird nur der *erste Bounce* des indirekten Lichts betrachtet. Das heißt, nur die erste Reflexion eines Lichtstrahls durch ein Objekt wird in die Beleuchtung mit einbezogen. Alle folgenden Reflexionen, die theoretisch die indirekte Beleuchtung beeinflussen, werden aus Performancegründen nicht beachtet. In der hier verwendeten Methode sollen alle Texel der Reflective Shadow Map als Sender indirekten Lichts berücksichtigt werden. Eine *Sampling*-Strategie wird genutzt, um die Werte der Reflective Shadow Map so abzufragen, dass die Fläche der Szene



(a) (b)

(c) (d)

Abbildung 13: Reflective Shadow Maps. Die Szene wird aus Sicht einer Lichtquelle gerendert. Dabei werden Informationen über (a) Tiefenposition, (b) Position in Weltkoordinaten, (c) Normale und (d) direktes Licht in jeweils einer Textur gespeichert. Die Tiefenposition kann alternativ auch direkt aus den Weltkoordinaten entnommen werden.

gleichmäßig abgedeckt wird. Die Werte der Samples werden dann mit der direkten Beleuchtung der Szene verrechnet, wodurch eine globale Beleuchtung simuliert wird.

Um die indirekte Beleuchtung schnell zu berechnen, werden N Samples aus der Reflective Shadow Map entnommen. Dabei wird eine *Halton-Sequenz* genutzt, um die Werte aus der Reflective Shadow Map gleichmäßig abzufragen. Eine Halton-Sequenz wird in der Statistik verwendet, um Punkte in einem Raum zu erzeugen, die quasi-zufällig verteilt sind. Im Falle der Reflective Shadow Map wird sie genutzt, um die Werte der Pixel an bestimmten Stellen in der Textur abzufragen. Die Halton-Sequenz besitzt dabei die Eigenschaft, dass aufeinanderfolgende Werte der Sequenz einen großen Abstand zueinander besitzen. Diese Eigenschaft verhindert, dass von den N entnommenen Samples viele in einer bestimmten Region der Textur liegen. Wäre das der Fall, hinge die indirekte Beleuchtung stark von einem bestimmten Teil der Szene ab anstatt von allen Bereichen der Szene.

Die aus der Reflective Shadow Map entnommenen N Samples werden genutzt, um N Strahlen zu berechnen, die von der Oberfläche eines Objekts in Richtung der Positionen der Samples gehen. Dabei sei $\mathbf{x}$ die Empfängerposition an der Oberfläche des Objekts, für den die indirekte Beleuchtung berechnet werden soll. Bei diesem Algorithmus wird nicht auf die Entfernung zwischen den Samples und der Empfängerposition geachtet, da theoretisch von allen Objekten der Szene, die beleuchtet werden, unabhängig von der Entfernung, indirektes Licht auf die Oberfläche treffen kann. Die N Strahlen müssen so gewählt werden, dass sie in der oberen Hemisphäre der Oberfläche liegen. Dies wird durch die Berechnung des Skalarprodukts zwischen der Oberflächennormalen und dem Strahl festgestellt. Nur Objekte, die von der Oberfläche des Objekts aus sichtbar sind, können als Sender indirekten Lichts in Frage kommen. Die Wahl der Samples aus einer Reflective Shadow Map sorgt dafür, dass letztendlich nur Samples entnommen werden, die durch eine oder mehrere Lichtquellen beleuchtet werden. Somit lassen sich alle Samples der Reflective Shadow Map, die in der oberen Hemisphäre der Oberfläche liegen, als Sender indirekten Lichts festlegen. Die Werte der Samples werden nun in Abhängigkeit des Einfallswinkels θ des zugehörigen Strahls aufsummiert und ergeben so die indirekte Beleuchtung L_i an der Position $\mathbf{x}$. Sie lässt sich somit berechnen durch

$$L_i(\mathbf{x}) = \frac{m_d(\mathbf{x})}{N} \sum_{i=1}^{N} \tilde{L}_i(\mathbf{x}, \omega_i) cos\theta, \tag{7}$$

wobei $m_d(\mathbf{x})$ die diffuse Materialfarbe am Empfängerpunkt x ist. ω_i bezeichnet die Richtung des Strahls, der zu Sample i geht. Der Wert $\tilde{L}_i(\mathbf{x}, \omega_i)$ gibt die indirekte Beleuchtung an, die der Punkt x aus der Richtung ω_i erhält. Diese Beleuchtung ergibt sich aus dem Wert, der sich an der Position des Samples in der Reflective Shadow Map befindet. Der Farbwert, der als

Ergebnis der indirekten Beleuchtung entsteht, wird zu der am Empfängerpunkt x berechneten direkten Beleuchtung addiert. Dadurch ergibt sich die globale Beleuchtung an Punkt x. Abbildung 14 verdeutlicht an einem Beispiel, wie das indirekte Licht in einer Szene berechnet wird.

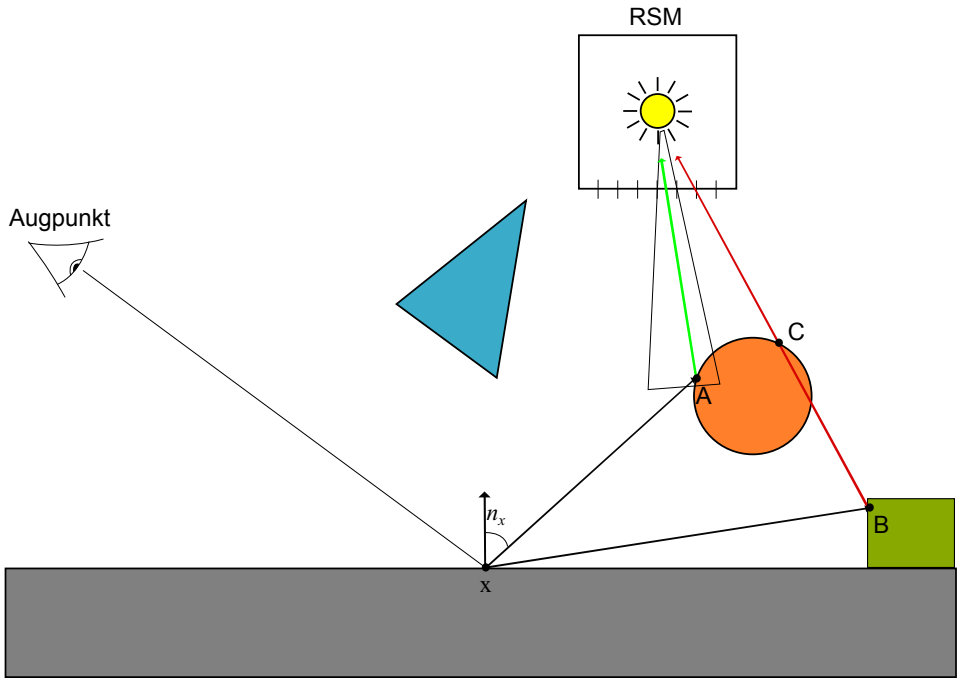


Abbildung 14: Indirekte Beleuchtung durch Reflective Shadow Maps. Vom Augpunkt aus ist Punkt x sichtbar. Deshalb wird für diesen die indirekte Beleuchtung für einen Bounce berechnet. Aus einem Texel der RSM wird ein Sample entnommen, welches Punkt A enthält. Dieser wird somit direkt von der Lichtquelle beleuchtet. Ein Vektor wird von Punkt x zu Punkt A gezogen, um anhand der Normalen n_x zu vergleichen, ob sich A in der oberen Hemisphäre von x befindet. Da dies der Fall ist, wird das reflektierte Licht von Punkt A in die indirekte Beleuchtung von x mit einbezogen. Punkt B ist zwar von x aus sichtbar, wird jedoch von Punkt C in Richtung der Lichtquelle verdeckt, weshalb er in der RSM nicht enthalten ist. Er gibt daher kein indirektes Licht an x ab.

Um die indirekte Beleuchtung in einer gegebenen Szene zu berechnen, wird vorab ein *G-Buffer* angelegt. Der G-Buffer enthält Informationen über Position, Normale und Materialfarbe aller Objekte der Szene, die von einer bestimmten Kameraperspektive aus sichtbar sind. Um diesen zu erstellen, wird die gesamte Szene aus der gewählten Kameraposition gerendert, die jeweiligen Werte ermittelt und in das entsprechende Fragment eingetragen. Daraus entstehen jeweils eine Textur für Position, Normale und Mate-

rialfarbe. Diese Texturen werden nun an das Shader-Programm gebunden, das die Berechnung des indirekten Lichts vornimmt. Zusätzlich werden die Texturen der Reflective Shadow Map als `uniform sampler2D` an den Shader übergeben. Eine weitere 2D-Textur, bestehend aus zufälligen Zahlen aus der Menge $[0,1]\times[0,1] \subset \mathbb{R}^2$, wird erstellt und ebenfalls für die Berechnungen im Shader verwendet. Diese erfüllt den Zweck, Positionen der Samples aus der Reflective Shadow Map zufällig auszuwählen, sodass benachbarte Fragmente nicht benachbarte Samples zugewiesen bekommen. Zusätzlich werden Werte einer 2D-Halton-Sequenz berechnet, deren Länge anhängig von der Anzahl der verwendeten Samples ist. Diese sorgt dafür, dass die benutzten Samples aus möglichst verschiedenen Bereichen der Reflective Shadow Map stammen.

Da die verschiedenen Informationen über die Objekte der Szene, die zur Beleuchtung benötigt werden, in einem G-Buffer gespeichert werden, muss der Rendervorgang zur Berechnung der indirekten Beleuchtung nur für ein *Screen Filling Quad* aufgerufen werden. Dieses Quader deckt den ganzen Bereich des Bildschirms ab, in den gerendert wird. Dadurch kann eine Beleuchtung pro Fragment anhand der Informationen aus dem G-Buffer stattfinden. Dieser Vorgang wird auch als *Deferred Shading* bezeichnet. Im Vertex Shader werden lediglich die UV-Koordinaten des Rechtecks ermittelt und an den Fragment Shader übergeben, damit diese interpoliert werden und jeder Texel in der G-Buffer-Textur abgefragt werden kann. Im Fragment Shader werden nun die Informationen aus dem G-Buffer anhand der UV-Koordinaten des Fragments entnommen. Anschließend wird über die Anzahl der gewünschten Samples iteriert. Innerhalb der Schleife wird ein Texel der Reflective Shadow Map anhand der Zufallstextur und der Halton-Sequenz festgelegt. An diesem Texel werden nun die gespeicherte Position und Normale eines Objekts, das von der Lichtquelle aus sichtbar ist, entnommen und somit die Position des Samples bestimmt. Ein Vektor wird erstellt, der von der Position an Stelle des Fragments zur Position des Samples geht. Daraufhin wird das Skalarprodukt zwischen diesem Vektor und der dem Fragment zugehörigen Normalen berechnet. Da beide Vektoren vorab normalisiert werden, ist das Ergebnis des Skalarprodukts der Cosinus des Winkels zwischen den beiden Vektoren. Ist der Cosinus größer als 0, bedeutet das, dass die Sample-Position in der oberen Hemisphäre aus Sicht des Empfängerpunkts liegt und somit indirektes Licht an diesen abgibt. Ist der Cosinus kleiner oder gleich 0, ist das Sample von dem Punkt aus in der unteren Hemisphäre und das von dort ausgestrahlte indirekte Licht kann diesen Punkt nicht treffen. Somit wird dieses Sample nicht weiter für die Berechnung verwendet. Ist dies der Fall, ist es vorteilhaft, eine neue Sample-Position zu bestimmen, damit die Anzahl der verwendeten Samples für jedes Fragment gleich bleibt. Sobald ein Sample gefunden wurde, das indirektes Licht an den Empfänger abgibt, wird aus der Reflective Shadow Map das direkte Licht an der Position des Samples entnommen. In Abhängigkeit des zuvor ermittelten Cosinus wird dieser Wert dann zu den Ergebnissen der

anderen Samples addiert. Der hier beschriebene Vorgang wird für die festgelegte Anzahl der Samples durchgeführt. Der daraus resultierende Farbwert wird mit der Materialfarbe des Fragments verrechnet und durch die Anzahl der Samples geteilt.

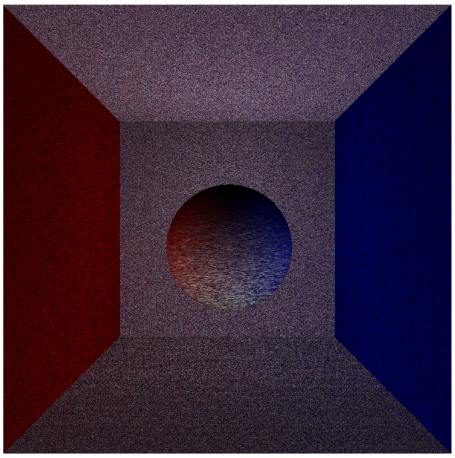


Abbildung 15: Ergebnis der indirekten Beleuchtung einer Szene. Zur Berechnung wurden hier 8 Samples pro Pixel aus der Reflective Shadow Map entnommen. Das Ergebnisbild ist aufgrund der geringen Anzahl verwendeter Samples verrauscht und muss vor der weiteren Verwendung mit einem Filter geglättet werden.

Zu beachten ist, dass in einer Echtzeit-Anwendung die Anzahl der Samples pro Fragment nicht zu hoch werden darf, da jedes weitere Sample die Laufzeit des Verfahrens beeinträchtigt. Das sorgt jedoch dafür, dass das indirekte Licht verrauscht wirkt (siehe Abbildung 15), da an jedem Pixel Samples aus unterschiedlichen Stellen der Reflective Shadow Map entnommen werden. Um solch einem Rauschen entgegenzuwirken, muss die Textur, in der das verrauschte indirekte Licht gespeichert ist, durch einen Filter geglättet werden. Dafür wird ein *Geometrie-sensitiver Filter* verwendet. Dieser Filter funktioniert ähnlich wie ein Gauß-Filter, bei dem über jeden Pixel des Bildes iteriert und anhand der Nachbarwerte des Pixels ein neuer Wert für diesen Pixel berechnet wird. Wie stark benachbarte Pixel diesen Wert beeinflussen, hängt dabei von einer Gauß-Verteilung ab. Diese hat den Effekt, dass die Pixel, die näher am Ursprungspixel liegen, stärker gewichtet werden. Zusätzlich zur Gewichtung anhand der Gauß-Verteilung wird bei dem Geometrie-sensitiven Filter der Wert eines Pixels in Abhängigkeit der Normalen gewichtet. Die Normale wird dabei aus dem G-Buffer entnommen. Es wird der Cosinus des Winkels zwischen den Normalen am Ursprungspixel und am Nachbarpixel berechnet. Ist der Cosinus groß, zeigen die Normalen in eine ähnliche Richtung; das Nachbarpixel wird daher stärker in die Berechnung des neuen Farbwerts mit einbezogen als ein Nachbarpixel, dessen

Normale in eine unterschiedliche Richtung zeigt. Wird dieser Filter auf die Textur angewandt, wird der Rauscheffekt, der durch die geringe Anzahl der Samples entsteht, verringert. Das geglättete indirekte Licht wird schließlich zum direkten Licht addiert und ergibt dadurch die gesamte globale Beleuchtung der Szene (siehe Abbildung 16).

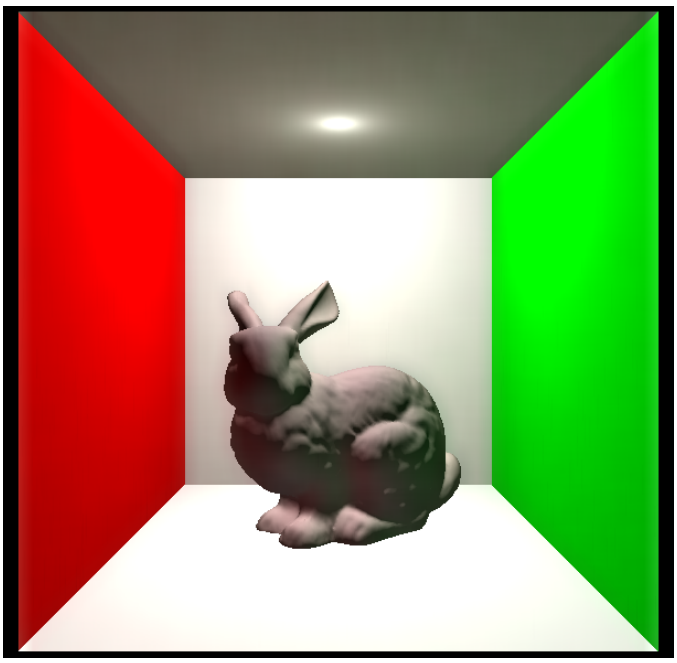


Abbildung 16: Globale Beleuchtung einer Szene unter Verwendung von Reflective Shadow Maps (60 fps).

Das hier beschriebene Verfahren zur Berechnung der indirekten Beleuchtung verwendet somit die Informationen aus Reflective Shadow Maps, um Punkte in der Welt zu finden, die indirektes Licht auf die Oberfläche eines Objekts reflektieren. Durch die Verwendung der Reflective Shadow Map wird sichergestellt, dass nur Punkte, die auch selbst beleuchtet werden, Licht an andere Objekte abgeben können. Weiterhin wird darauf geachtet, dass nur Samples in die Berechnung einbezogen werden, die aus Sicht der Oberfläche in der oberen Hemisphäre liegen. Dadurch wird versucht, eine glaubwürdige globale Beleuchtung zu erzielen. Allerdings berücksichtigt dieses Verfahren nicht den Fall, dass Objekte zwischen der Empfängerposition und der Sample-Position liegen können. In diesem Fall würde das indirekte Licht durch das dazwischenliegende Objekt abgefangen werden und der Empfängerpunkt kein indirektes Licht von diesem Sample erhalten. Der folgende Abschnitt befasst sich mit einer Optimierung, welche die hier beschriebene Methode erweitert, damit solche Objekte erkannt werden können.

5.4 Optimierung durch Voxel-Traversierung

In Abschnitt 5.3 wurde besprochen, wie mithilfe von Reflective Shadow Maps indirektes Licht simuliert wird. Dabei wurde ein Vektor von der aktuellen Empfängerposition zu mehreren Positionen in der Reflective Shadow Map berechnet und dann ermittelt, ob diese Positionen vom Empfängerpunkt aus sichtbar sind und somit indirektes Licht an diesen Punkt abgeben. Dabei gilt als Kriterium, dass die Position des Samples von einer oder mehreren Lichtquellen aus sichtbar und in der oberen Hemisphäre aus Sicht des Empfängerpunkts liegen muss. Es gibt jedoch weitere Faktoren, die beeinflussen können, ob ein Objekt indirektes Licht an ein anderes Objekt abgibt. In dem im vorherigen Abschnitt erläuterten Verfahren wird nicht der Fall behandelt, dass Objekte zwischen der Sample-Position und der Empfängerposition liegen können. Tritt dieser Fall ein, würde das indirekte Licht des Samples anstatt an der Empfängerposition auf der Oberfläche des dazwischenliegenden Objektes landen. Das zuvor beschriebene Verfahren berechnet in diesem Fall jedoch die indirekte Beleuchtung so, als würde das dazwischenliegende Objekt nicht existieren.

In diesem Abschnitt wird daher ein Optimierungsschritt erläutert, der einen solchen Fall abfängt, damit die Berechnung des indirekten Lichts mittels Reflective Shadow Maps korrigiert wird. Das soll zu einer glaubwürdigeren Darstellung des indirekten Lichts in einer Szene beitragen. Zudem wird die indirekte Beleuchtung auf die direkte Umgebung der Objekte beschränkt. Das hier beschriebene Verfahren wurde auch von Thiedemann [7] zur Berechnung einer globalen Beleuchtung verwendet. Hierbei wird eine Darstellung der Szene als Voxelgitter verwendet, deren Erstellung zuvor in Abschnitt 3.3 beschrieben wurde. Der Vektor, der zur Berechnung des indirekten Lichteinflusses zwischen der Empfängerposition und der Sample-Position berechnet wurde, wird dabei verwendet, um eine Traversierung der Voxelstruktur mittels dem in Abschnitt 4.1 beschriebenen Algorithmus durchzuführen. Dazu wird das Voxelgitter entlang der Richtung des Vektors durchlaufen und auf Schnittpunkte mit Objekten der Szene getestet. Sobald ein Objekt entlang dieses Strahls getroffen wurde, wird überprüft, ob der getroffene Punkt derjenige ist, der als Sample aus der Reflective Shadow Map entnommen wurde. Ist dies der Fall, befindet sich kein Objekt zwischen Empfängerposition und Sample-Position. Sind die Positionen jedoch unterschiedlich, lässt sich darauf schließen, dass ein Objekt dazwischen liegt, welches das indirekte Licht des Samples erhält. Infolgedessen wird dieses Sample nicht für die Berechnung des indirekten Lichts am Empfängerpunkt verwendet. In Abbildung 17 wird dargestellt, welche Samples in die Berechnung des indirekten Lichts einbezogen und wann diese verworfen werden.

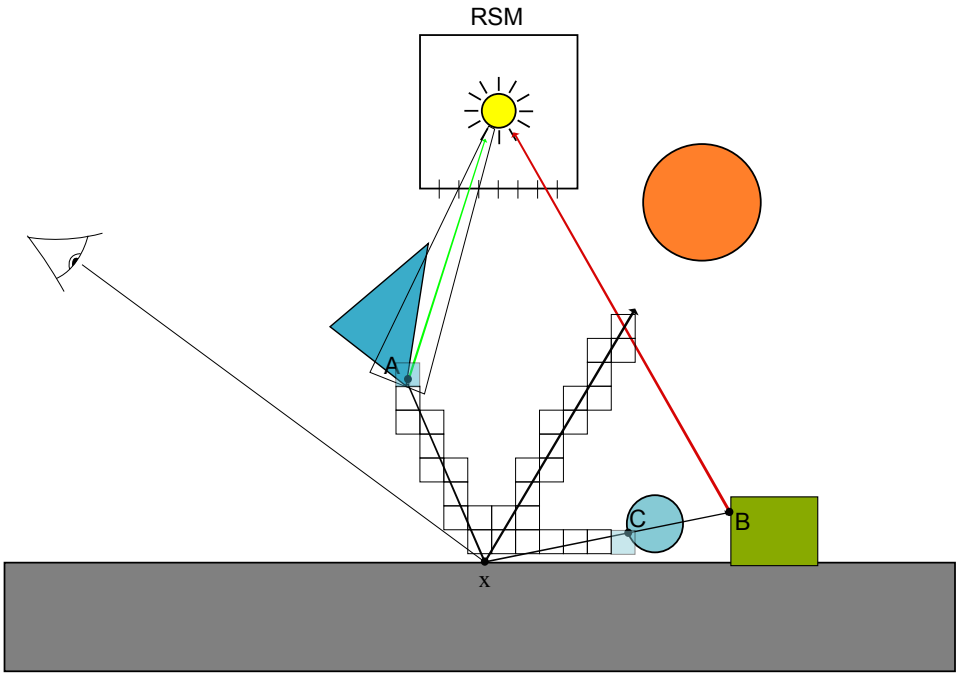
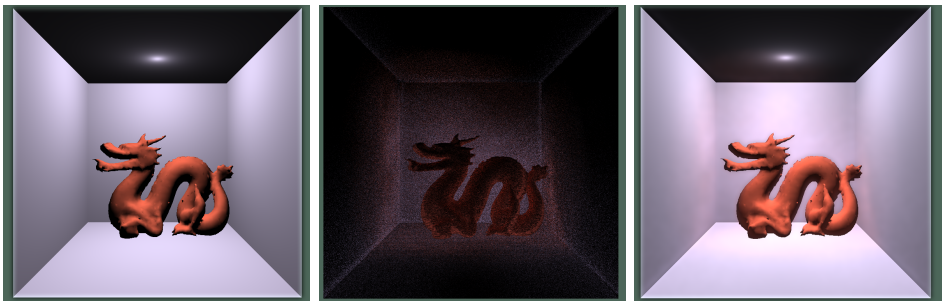


Abbildung 17: Indirekte Beleuchtung unter Verwendung eines Voxelgitters. Für Punkt x soll die indirekte Beleuchtung berechnet werden. Aus der Reflective Shadow Map wird ein Texel als Sample entnommen, in dem Punkt A enthalten ist. Das Voxelgitter wird von x aus in Richtung von A traversiert. Dabei wird ein gefülltes Voxel getroffen, das den Punkt A enthält. Somit reflektiert A indirektes Licht an den Empfängerpunkt x. Anschließend wird Punkt B aus der RSM entnommen. Der Punkt liegt aus Sicht von x in der oberen Hemisphäre. Daher wird auch hier das Voxelgitter in Richtung von B traversiert. Dabei wird ein Voxel getroffen, das Punkt C enthält. Da sich die beiden Punkte um mehr als einen gewissen Schwellwert unterscheiden, wird festgestellt, dass sie nicht gleich sind. Somit liegt ein Objekt zwischen x und B, weshalb B kein indirektes Licht an x abgibt.

Das optimierte Verfahren zur Berechnung der indirekten Beleuchtung einer Szene beginnt mit den selben Schritten wie in Kapitel 5.3 beschrieben. Die Informationen über Weltposition, Normale und Materialfarbe werden pro Fragment aus einem G-Buffer entnommen. Anschließend wird in einer Schleife über die Anzahl der verwendeten Samples iteriert. Innerhalb der Schleife werden die Samples an bestimmten Stellen, die durch die Zufallstextur und die Halton-Sequenz bestimmt werden, aus der Reflective Shadow Map entnommen. Zwischen der Weltposition des Fragments und der Sample-Position wird ein Vektor erstellt und mit der Normalen des Fragments mithilfe des Skalarprodukts verrechnet, um zu bestimmen, ob das Sample aus Sicht des Empfängerpunkts in der oberen Hemisphäre liegt. Ist dies der Fall, ist das Sample ein potentieller Sender indirekten Lichts an die Empfängerposition. Nun muss überprüft werden, ob das Sample von der Empfängerposition aus sichtbar ist oder sich ein Objekt zwischen den beiden Punkten befindet. Um dies zu bestimmen, wird ein Strahl erstellt, der von der Empfängerposition in Richtung der Sample-Position geht. Der Startpunkt des Strahls ist dabei die Position des Fragments in Weltkoordinaten. Die Richtung des Strahls wird durch den Vektor zwischen dieser Position und der Position des Samples bestimmt. Mithilfe dieser Informationen wird nun das zuvor erstellte Voxelgitter entlang des Strahls wie in Abschnitt 4.1 beschrieben durchlaufen. Dabei ist die Anzahl der Iterationsschritte entlang des Strahls abhängig von einer Variable `steps`, die vom Nutzer festgelegt wird. Werden nur wenige Schritte entlang des Strahls getätigt, beschränkt sich der Algorithmus nur auf einen Bereich in der Nähe des Objekts, von dem aus der Strahl gesendet wird. Dadurch können aber Objekte, die weiter entlang des Strahls liegen, verfehlt werden. Wird `steps` so gewählt, dass der Strahl vollständig durchlaufen wird, trifft er garantiert auf ein Objekt, da seine Richtung von der Position des Samples abhängig ist und so mindestens dieses im Voxelgitter treffen muss. Aus Gründen der Performance kann jedoch nicht in jedem Fall der Strahl vollständig abgelaufen werden, was eine Beschränkung auf das Nahfeld um die Empfängerposition notwendig macht, wenn die indirekte Beleuchtung für eine Echtzeit-Anwendung ermittelt werden soll.

Trifft der Strahl nun auf ein gefülltes Voxel, muss ermittelt werden, ob die Position an der Stelle des Voxels mit der Position des Samples aus der Reflective Shadow Map übereinstimmt. Dies wird dadurch vereinfacht, dass aufgrund der Verwendung einer mehrwertigen Voxelisierung die Weltposition direkt an der entsprechenden Stelle im Voxelgitter vorhanden ist. Somit muss nur die Position am getroffenen Voxel abgefragt werden und mit der Position des Samples verglichen werden. Liegt der Abstand zwischen den beiden Positionen unterhalb eines gewählten Schwellwerts ϵ, so wird angenommen, dass der getroffene Punkt tatsächlich dem Punkt aus der Reflective Shadow Map entspricht. Infolgedessen wird die direkte Beleuchtung dieses Samples, die ebenfalls in der Reflective Shadow Map enthalten ist, abgefragt und anhand der Gleichung (7) mit den restlichen Samples verrechnet, um die

indirekte Beleuchtung am Empfängerpunkt zu ermitteln. Die Ergebnisse werden anschließend wie in Abschnitt 5.3 beschrieben mittels eines *Geometrie-sensitiven Filter* gefiltert und zum direkten Licht addiert. Daraus ergibt sich die globale Beleuchtung der Szene, wie in Abbildung 18 zu sehen ist.



(a) (b) (c)

Abbildung 18: Globale Beleuchtung, beschränkt auf die direkte Umgebung der Objekte. Das verwendete Voxelgitter besitzt eine Auflösung von 128^3. (a) Die berechnete direkte Beleuchtung. (b) Indirekte Beleuchtung der Szene. Durch die Verwendung von 8 Samples pro Pixel ergibt sich ein verrauschtes Bild. (c) Das Rauschen des indirekten Lichts wird geglättet und zur direkten Beleuchtung addiert, um die globale Beleuchtung der Szene zu ermitteln (48 fps).

6 Evaluation

6.1 Performance der Texturatlas-Voxelisierung

Dieser Abschnitt befasst sich mit der Auswertung der Performance der binären und mehrwertigen Voxelisierung. Die Laufzeit beider Algorithmen ist dabei abhängig von der Größe des Texturatlas, der für die Voxelisierung verwendet wird. Je höher die Auflösung ist, desto mehr Vertices müssen generiert und in das Voxelgitter eingesetzt werden. Zudem muss die Auflösung des Texturatlas an die des Voxelgitters angepasst werden (siehe Abbildung 19). Ist die Auflösung des Texturatlas zu klein im Vergleich zum Voxelgitter, werden zu wenige Voxel gefüllt. Das voxelisierte Modell ist dann nicht "wasserdicht", das bedeutet, es entstehen Löcher im Voxelgitter. Ist die Auflösung jedoch zu groß, kann es passieren, dass mehrere Vertices in dasselbe Voxel geschrieben werden, was die Performance des Algorithmus verschlechtert. Ein gutes Verhältnis zwischen Auflösung des Texturatlas und der des Voxelgitters ergibt sich, wenn ein Texel im Texturatlas in Weltkoordinaten etwa den gleichen Platz einnimmt wie ein Voxel im Voxelgitter.

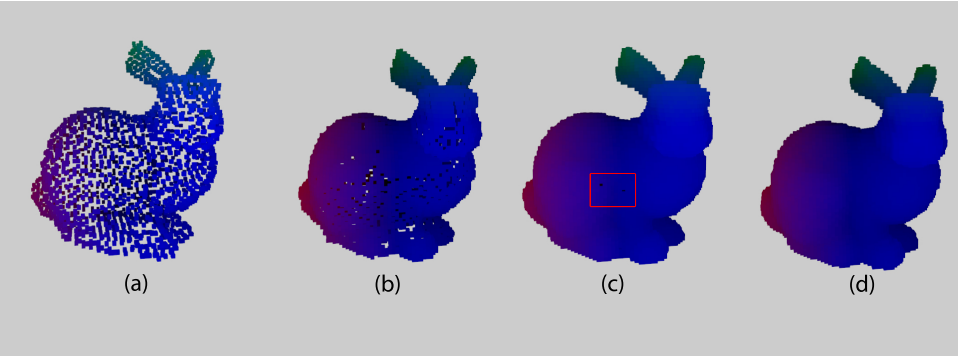


Abbildung 19: Nutzung verschiedener Auflösungen des Texturatlas zur Voxelisierung. Auflösung des Voxelgitters ist 128^3. Auflösung des Texturatlas: (a) 64^2, (b) 112^2, (c) 224^2, (d) 368^2. Die Auflösung der Texturatlanten in den beiden linken Bildern ist zu klein, wodurch große Löcher im Voxelgitter entstehen. (c) liefert ein ausreichend gefülltes Voxelgitter, in dem jedoch auch kleine Lücken entstehen können. In (d) wird der Hase ohne Lücken dargestellt, was allerdings durch die höhere Anzahl gerenderter Vertices zulasten der Performance geht.

In manchen Fällen kann das Verfahren beschleunigt werden, wenn die ungültigen Texel des Texturatlas schon vor der Übergabe der Vertices an den Vertex Shader aussortiert werden. Gerade dann, wenn der Texturatlas eine hohe Auflösung besitzt oder das UV-Mapping des Objektes viele freie Stellen in der Textur lässt, kann diese Variante je nach verwendetem Algorithmus eine Verbesserung der Performance des Verfahrens herbeiführen.

Der Ansatz der binären Voxelisierung besitzt den Vorteil, dass durch die geschickte Verwendung einzelner Bits innerhalb einer 2D-Textur viel Speicherplatz gespart wird. Das kann entscheidend sein, wenn sehr viele Objekte in einer Szene verwendet werden, da für jedes Objekt ein oder mehrere Texturatlanten erstellt werden müssen. Je größer der benötigte Speicherplatz für jeden Texturatlas ist, desto schneller sind die verfügbaren Ressourcen aufgebraucht. Die Verwendung der RGBA-Kanäle einer 2D-Textur sorgt allerdings dafür, dass maximal 128 Bits für die Codierung der Tiefenposition verwendet werden können. Dieser Wert ist für die meisten Szenen ausreichend, für Szenen mit hoher Tiefenkomplexität kann jedoch eine höhere Auflösung nötig sein.

Die Verwendung einer 3D-Textur zur Speicherung mehrerer Daten im Voxelgitter besitzt den Vorteil, dass die angelegte Textur eine beliebige Auflösung haben kann und daher mehr Tiefenpositionen verwendet werden können als bei der binären Voxelisierung. Diese Flexibilität kann bei einer hohen Auflösung allerdings zu Lasten der Performance gehen. Die Möglichkeit, mehrere Daten in der Textur abzuspeichern, kann Rechenzeit bei der späteren Verwendung des Voxelgitters sparen. Alle benötigten Informationen zu einem Voxel können dann einfach durch eine Texturabfrage erhalten werden, anstatt für jedes Voxel im Nachhinein Berechnungen durchführen zu müssen. Dafür wird jedoch eine deutlich größere Menge an Speicherplatz benötigt, um die zusätzlichen Informationen pro Voxel unterbringen zu können. Betrachtet man jede Schicht einer 3D-Textur oder 2D-Array-Textur als eigene 2D-Textur und sei n die Tiefe des Voxelgitters, so müssen bei der mehrwertigen Voxelisierung im Vergleich zur binären Voxelisierung n-mal mehr 2D-Texturen angelegt werden. Je nach Komplexität des Programmes und benötigter Tiefeninformation der Szene kann dieser zusätzliche Speicheraufwand für Probleme sorgen.

Weiterhin ist es anzumerken, dass es bei der mehrwertigen Voxelisierung zu Problemen kommen kann, wenn mehrere Objekte in ein Voxel eingetragen werden. Verwendet man eine binäre Voxelisierung, wird nur die Sichtbarkeit eines Objekts in das Voxel gespeichert. Wird ein weiteres Objekt in ein binäres Voxel eingetragen, ändert sich der Inhalt des Voxels nicht. Bei einer mehrwertigen Voxelisierung wird die Information, die über das erste Objekt gespeichert wurde, jedoch durch die Information des nächsten Objekts überschrieben. Der Inhalt der einzelnen Voxel ist damit abhängig von der Reihenfolge, in welcher die Objekte in das Voxelgitter eingesetzt werden. Benutzt man das Voxelgitter z.B. zum Speichern der direkten Beleuchtung der Objekte, kann die mehrwertige Voxelisierung inkonsistente Ergebnisse bei der finalen Beleuchtung hervorrufen. Dieses Problem äußert sich in besonderem Maße, wenn das Voxelgitter eine niedrige Auflösung besitzt oder sich in der Szene viele dünne Objekte befinden. Das sorgt dafür, dass die Breite dieser Objekte kleiner ist als die Voxelbreite und mehrere Objekte mit verschiedenen Eigenschaften in einem Voxel landen können.

Auflösung Voxelgitter	Auflösung Texturatlas	Zeit binär	Zeit mehrwertig
$64^2 \times 128$	224×224	1.38 ms	1.47 ms
128^3	368×368	1.84 ms	2.23 ms
$256^2 \times 128$	768×768	3.44 ms	5.68 ms
$512^2 \times 128$	1280×1280	6.20 ms	13.35 ms

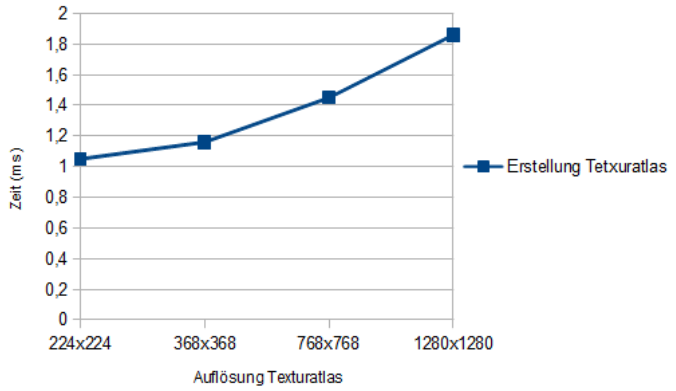
Tabelle 1: Vergleich der Zeitmessungen von binärer und mehrwertiger Voxelisierung mithilfe eines Texturatlas. Das Testmodel besteht aus 208K Vertices. Da bei der binären Voxelisierung maximal 128 Bits durch die Nutzung des RGBA-Kanals verwendet werden können, wird die Tiefenauflösung des Voxelgitters auf 128 festgelegt. Die Laufzeit steigt mit der Auflösung des Texturatlas, da für jeden Texel ein Vertex an die Grafik-Pipeline übertragen wird. Bei der mehrwertigen Voxelisierung ist dies sehr ausschlaggebend, da hier die Vertices zusätzlich durch den Geometry Shader laufen müssen.

Ein weiteres Problem der Verwendung von 3D-Texturen liegt in der Verwendung unter der OpenGL-API. In Version 4.4 ist es nur innerhalb eines Geometry Shaders möglich, Werte zur GLSL Built-In Variable `gl_Layer` zuzuweisen. Für dieses Verfahren wird allerdings außer dieser Variable keine weitere Funktionalität des Geometry Shaders benötigt. Somit muss bei der Erstellung eines Voxelgitters ein zusätzlicher Pass durch den Geometry Shader durchgeführt werden, nur um `gl_Layer` auf den gewünschten Wert zu setzen. Dies bedeutet zusätzlichen Overhead, da im Geometry Shader für jeden übergebenen Vertex zusätzliche Anweisungen ausgeführt werden müssen und die Grafik-Pipeline dementsprechend angepasst werden muss. Dieser Overhead könnte vermieden werden, wenn `gl_Layer` sich auch im Vertex Shader verwenden lassen würde. Tatsächlich ist dies durch die OpenGL-Extension `AMD_vertex_shader_layer` möglich; diese Erweiterung kann allerdings nur unter der Verwendung von AMD-Grafikkarten benutzt werden. Eine Unterstützung dieser Extension für Grafikkarten anderer Hersteller wäre daher äußerst nützlich und könnte eine Beschleunigung des Verfahrens hervorrufen.

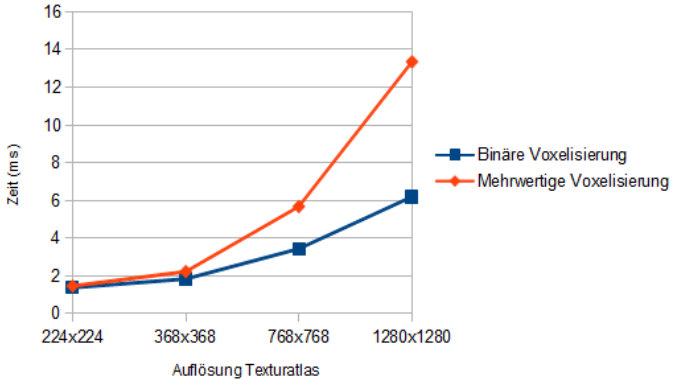
Befinden sich in der zu voxelisierenden Szene sowohl statische als auch dynamische Objekte, kann eine Beschleunigung durch eine Vorvoxelisierung der statischen Objekte herbeigeführt werden. Dazu werden in einem ersten Renderpass alle statischen Objekte in das Voxelgitter eingesetzt, bevor die eigentliche Renderschleife beginnt. Zur Laufzeit müssen lediglich die dynamischen Objekte der Szene in das Gitter eingefügt werden. Je nach Verhältnis zwischen statischen und dynamischen Objekten in der Szene kann diese Aufspaltung der Rendervorgänge zu einer erheblichen Beschleunigung des Verfahrens beitragen.

In Abbildung 20 sowie in Tabelle 1 werden die Laufzeiten der verschiedenen Voxelisierungsmethoden unter Verwendung verschiedener Auflösungen

des Voxelgitters und des Texturatlas miteinander verglichen. Das Testsystem besteht dabei aus einer NVIDIA GeForce GTX 760 (2048 MB RAM), AMD Phenom II X4 955 CPU(4 × 3,2 GHz), 4 GB RAM unter Windows 7. Aus diesen Tests ergibt sich, dass die binäre Voxelisierung gerade bei hohen Auflösungen weniger Rechenaufwand benötigt. Im weiteren Verfahren wird trotzdem die mehrwertige Voxelisierung verwendet, da die entstehende Datenstruktur als 3D-Textur leichter für weitere Algorithmen nutzbar ist. Dabei ist der Rechenaufwand immer noch niedrig genug, sodass die Nutzung für die globale Beleuchtung immer noch in Echtzeit erfolgen kann. Zudem ist abzusehen, dass durch zusätzliche Funktionalität in kommenden OpenGL-Versionen die Unterstützung von 3D-Texturen weiter ausgebaut wird, was die Laufzeit dieses Verfahrens verringern kann.



(a)



(b)

Abbildung 20: Diagramm zum Vergleich der Laufzeit (a) der Erstellung des Texturatlas, (b) des gesamten Voxelisierungsvorgangs unter Berücksichtigung verschiedener Auflösungen des Texturatlas. Verwendet wurde ein Testmodell mit 208K Vertices.

6.2 Auswertung des Beleuchtungsverfahrens

Im Folgenden wird das beschriebene globale Beleuchtungsverfahren bezüglich dessen Performance ausgewertet. Dabei werden Unterschiede betrachtet, die die Verwendung einer Voxelstruktur zur Beleuchtung mit sich bringt. Das verwendete Testsystem ist eine NVIDIA GeForce GTX 760(2048 MB RAM), AMD Phenom II X4 955 CPU(4 $\times$ 3,2 GHz), 4 GB RAM mit Windows 7. Modelle wurden zum Teil in der Open-Source-Software Blender erstellt bzw. modifiziert und über die C++-Bibliothek Assimp in das Programm geladen.

Die Laufzeit des Verfahrens ist von vielen verschiedenen Faktoren abhängig. Ein besonders ausschlaggebender Faktor ist dabei die Anzahl der verwendeten Samples, die pro Pixel aus der Reflective Shadow Map entnommen werden (siehe Abbildung 21). Diese Anzahl sollte daher möglichst gering gehalten werden, damit die globale Beleuchtung in Echtzeit berechnet werden kann. Standardmäßig werden in den Beispielen 8 Samples pro Pixel gewählt. Dieser Wert kann je nach Verwendungszweck angepasst werden; bei einer zu geringen Sample-Anzahl kann jedoch ein Rauschen der indirekten Beleuchtung im Bild sichtbar werden, wenn der verwendete Filter nicht dementsprechend angepasst wird.

Bei Verwendung der Voxelstruktur zur globalen Beleuchtung wird die Performance zusätzlich von der maximalen Anzahl der Iterationsschritte entlang der Strahlen durch das Voxelgitter beeinflusst. Einen Vergleich der Ausführungszeiten bei verschiedenen Schrittzahlen liefert Tabelle 2. Dabei beeinflusst die Schrittzahl die maximale Distanz, in die sich das indirekte Licht ausbreitet. Je kleiner somit die Anzahl der Iterationsschritte ist, desto mehr wird das Color Bleeding durch die indirekte Beleuchtung nur auf die direkte Umgebung der Objekte beschränkt. Eine hohe Schrittzahl sorgt für eine größere Reichweite der indirekten Beleuchtung, die sich immer mehr den Ergebnissen ohne Verwendung der Voxelstruktur annähert (siehe Abbildung 23).

Des Weiteren beeinflusst auch die Auflösung der Reflective Shadow Map die Laufzeit des Verfahrens. In Abbildung 22 wird die Erstellung einer Reflective Shadow Map mit verschiedenen Auflösungen verglichen. Dabei ist zu berücksichtigen, dass der dazugehörige Rendervorgang ebenso wie die Erstellung des G-Buffers und der Texturatlanten stets abhängig von der Komplexität der Szene sind. Zur Berechnung der Beleuchtung hingegen werden die Informationen aus dem G-Buffer verwendet, weshalb hier die Performance nur noch von der Anzahl der Pixel im Ergebnisbild abhängig ist.

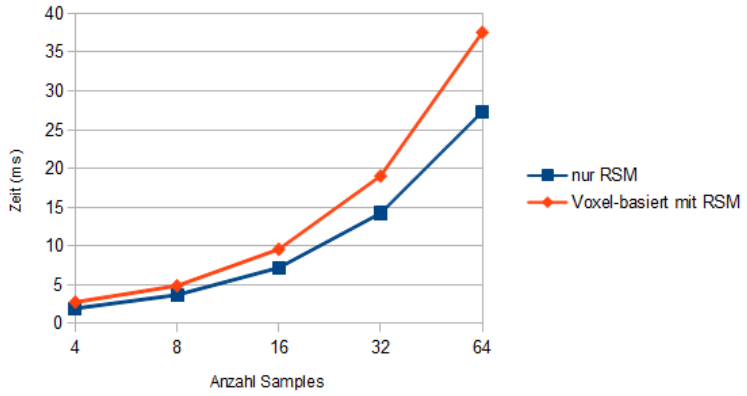


Abbildung 21: Indirekte Beleuchtung mit und ohne Traversierung der Voxelstruktur unter Berücksichtigung der verwendeten Sample-Anzahl. Durch die zusätzliche Voxel-Traversierung ergibt sich bei der Voxel-basierten Beleuchtung eine etwas höhere Laufzeit des Algorithmus.

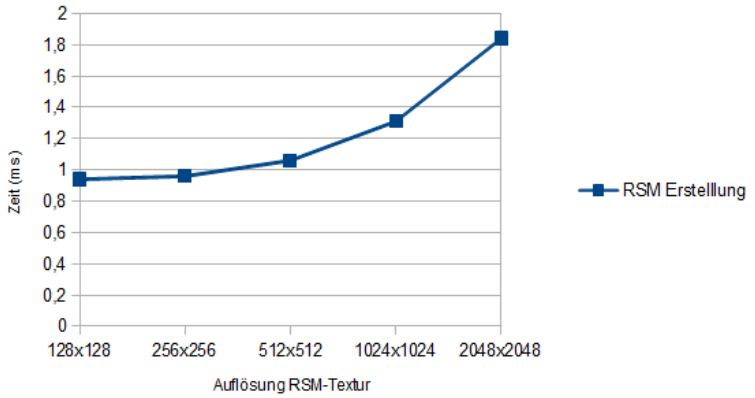
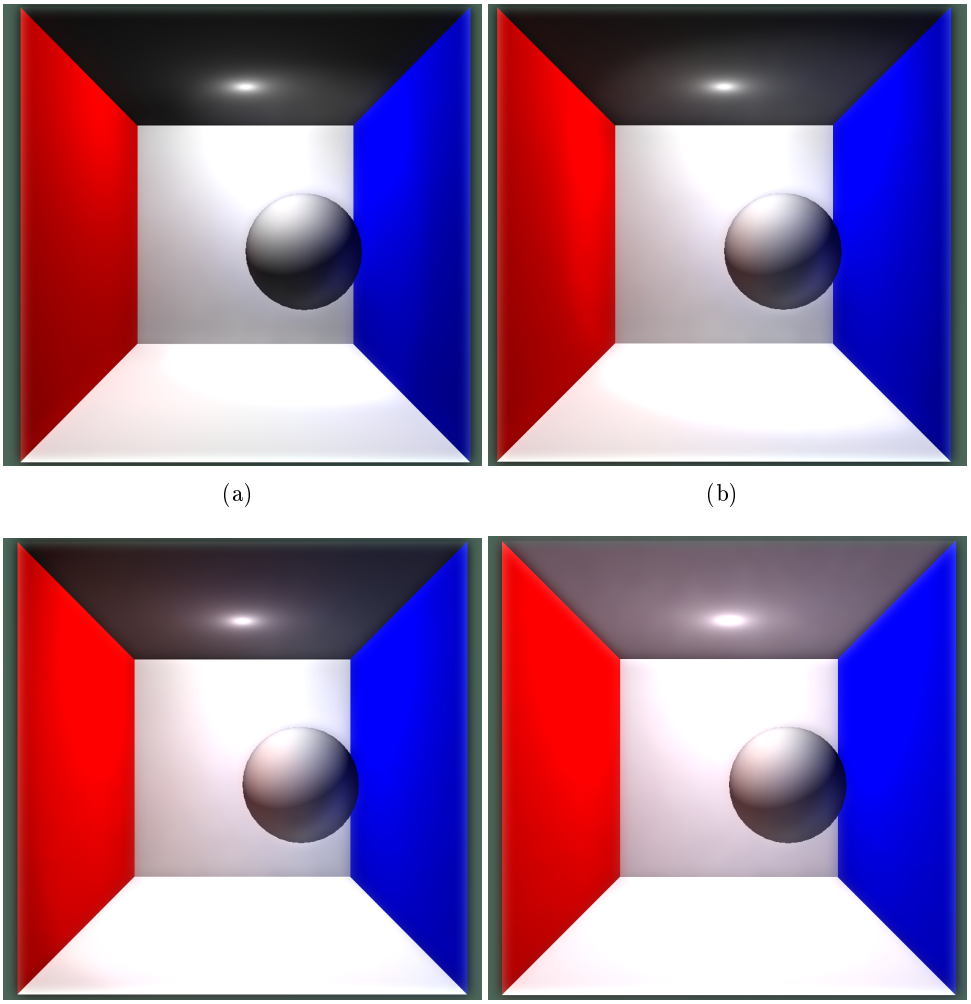


Abbildung 22: Vergleich der Laufzeit der RSM-Erstellung bei verschiedenen Texturauflösungen.



(a) (b)

(c) (d)

Abbildung 23: Globale Beleuchtung mithilfe der Voxel-Traversierung. (a) 48 Iterationsschritte; das Color Bleeding der roten Wand ist auf der Kugel nicht zu erkennen. (b) 64 Iterationsschritte; (c) 86 Iterationsschritte; (d) globale Beleuchtung lediglich mit der Verwendung von Reflective Shadow Maps.

Anzahl Iterationsschritte	Zeit (in ms)
12	3.40
24	4.87
48	8.86
96	16.98

Tabelle 2: Zusammenhang zwischen Ausführungszeit der indirekten Beleuchtung mithilfe der Voxelstruktur und der maximalen Schrittzahl entlang den Lichtstrahlen. Wird die Schrittzahl zu hoch, kann die Performance des Algorithmus einbrechen. Eine geringe Schrittzahl resultiert darin, dass potentielle Schnittpunkte in der Voxelstruktur verfehlt werden können und das indirekte Licht nur auf ein Nahfeld um die Objekte beschränkt wird.

7 Fazit

Diese Bachelorarbeit zeigt, wie ein globales Beleuchtungsverfahren auf Basis eines Voxelisierungsverfahrens unter OpenGL 4.4 implementiert werden kann. Dabei ist die Verwendung dieses Verfahrens in ausreichender Framerate in interaktiven Anwendungen möglich. Eine Voxelisierung wurde durchgeführt, um Elemente einer Szene in Voxel einzuteilen und die Berechnung des indirekten Lichts zu vereinfachen. Dabei wurden zwei verschiedene Voxelisierungsmethoden, die binäre sowie die mehrwertige Voxelisierung, vorgestellt. Beide Methoden arbeiten dabei auf Basis eines Texturatlas, welcher Informationen über die Position der Objekte im Raum erhält. Dadurch können auch Objekte voxelisiert werden, die aus Sicht der Kamera verdeckt sind, was bei Screen-Space-Verfahren nicht möglich ist. Die beiden präsentierten Voxelisierungsmethoden wurden miteinander verglichen, um Vor-und Nachteile bei der jeweiligen Verwendung aufzuzeigen. Dabei wurde die mehrwertige Voxelisierung für die weitere Verwendung vorgezogen, da sie eine besonders leicht verwendbare Datenstruktur als Ergebnis liefert.

Des Weiteren wurde ein Verfahren vorgestellt, mit dem die Voxelstruktur in eine bestimmte Richtung traversiert werden kann. Dieser Algorithmus kann genutzt werden, um die indirekte Beleuchtung einer Szene zu berechnen. Dazu werden Strahlen von der Oberfläche eines Objekts in den Raum geschossen und ermittelt, ob und auf welche Objekte diese treffen. Zur Berechnung der indirekten Beleuchtung werden außerdem Reflective Shadow Maps verwendet, welche Informationen über die Szene aus Sicht der Lichtquelle festhält. Die direkte Beleuchtung der Szene für diffuse Objekte wird ebenfalls berechnet. Zusammen mit der indirekten Beleuchtung wird daraus die globale Beleuchtung ermittelt. Dabei wurde die globale Beleuchtung mit und ohne Verwendung der Voxelstruktur simuliert, um festzustellen, welche Änderungen die Nutzung von Voxeln als Datenstruktur hervorruft. Zu beachten ist, dass die Laufzeit der Voxel-basierten Beleuchtung stark von der

Reichweite abhängig ist, in der Objekte Licht an ihre Umgebung abgeben. Die Reichweite kann dabei je nach Anwendungsfall angepasst werden, um eine indirekte Beleuchtung nur auf eine nahe Umgebung der Objekte zu beschränken.

Bei der hier präsentierten Methode zur Voxel-basierten Simulation von globaler Beleuchtung sind weitere Optimierungsvorgänge möglich. Die Wahl eines geschickteren Traversierungsalgorithmus, der bestimmte Regionen im Voxelgitter überspringt, falls dort keine Objekte vorhanden sind, könnte das Verfahren allgemein beschleunigen. Durch die Auslagerung von Berechnungen in *Compute Shader*, die seit Version 4.3 von OpenGL unterstützt werden, könnten ebenfalls Verbesserungen der Performance hervorgerufen werden. Generell ist anzunehmen, dass in zukünftigen Versionen der OpenGL-API Funktionalität hinzugefügt wird, die das Verfahren weiter beschleunigt. Das könnte beispielsweise dafür sorgen, dass die Verwendung eines Geometry Shaders zum Erstellen eines mehrwertigen Voxelgitters mithilfe einer 3D-Textur überflüssig wird. Optisch kann eine Verbesserung hinzugefügt werden, wenn die Berechnung des indirekten Lichts nicht nach einer Reflexion stoppt, sondern die Lichtstrahlen mehrmals von Objekten der Szene abprallen (*Multi-Bounce Indirect Lighting*). Während sich in dieser Implementation auf diffuse Materialien beschränkt wurde, ist die Erweiterung des Verfahrens auf spiegelnde Materialien denkbar.

Abschließend ist zu sagen, dass globale Beleuchtungsverfahren ein Thema sind, bei dem laufend Weiterentwicklungen stattfinden. Viele dieser Verfahren beschränken sich mittlerweile auf eine Berechnung dynamischer Szenen in Echtzeit und die Ergebnisse wirken immer realistischer. Gerade die Verwendung von Voxeln erfährt dabei eine immer weiter steigende Beliebtheit. Es ist abzusehen, dass in diesem Gebiet in Zukunft noch viele Fortschritte erzielt werden.

Literatur

[1] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. In *In Eurographics '87*, pages 3–10, 1987.

[2] Carsten Dachsbacher and 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.

[3] Elmar Eisemann and Xavier Décoret. Fast scene voxelization and applications. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, I3D '06, pages 71–78, New York, NY, USA, 2006. ACM.

[4] A. Fujimoto, Takayuki Tanaka, and K. Iwata. Tutorial: Computer graphics; image synthesis. chapter ARTS: Accelerated Ray-tracing System, pages 148–159. Computer Science Press, Inc., New York, NY, USA, 1988.

[5] James T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, August 1986.

[6] Georgios Passalis, Theoharis Theoharis, George Toderici, and Ioannis A. Kakadiaris. General voxelization algorithm with scalable gpu implementation. *J. Graphics Tools*, pages 61–71, 2007.

[7] Sinje Thiedemann, Niklas Henrich, Thorsten Grosch, and Stefan Müller. Voxel-based global illumination. *Symposium on Interactive 3D Graphics and Games*, pages 103–110, 2011.