UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

Globale Beleuchtung im Bildraum unter besonderer Berücksichtigung der Sichtbarkeitsbestimmung

Studienarbeit

im Studiengang Computervisualistik

vorgelegt von
Sinje Thiedemann

Betreuer: Prof. Dr. Stefan Müller, Dipl.-Inform. Niklas Henrich
Institut für Computervisualistik, AG Computergraphik

Koblenz, im September 2009

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 Studienarbeit
Sinje Thiedemann (Matr.-Nr. 205 210 268)

**Thema: Globale Beleuchtung im Bildraum
unter besonderer Berücksichtigung der Sichtbarkeitsbestimmung**

Die Simulation einer globalen Beleuchtung im dreidimensionalen Objektraum ist sehr rechenintensiv und hängt von der Komplexität der Szene ab. Dabei ist besonders die Berechnung der Sichtbarkeit aufwändig, d. h. der Test, ob sich zwei Punkte in der Szene gegenseitig sehen können.

Verfahren, die die globale Beleuchtung vom Objektraum in den Bildraum (Screen-Space / Image-Space) verlagern, umgehen das Problem der Szenenkomplexität und haben somit einen wesentlichen Geschwindigkeitsvorteil. Auf diese Weise erzeugte Effekte sind zwar naturgemäß nicht physikalisch korrekt, da die aus der Sicht der Kamera verdeckte Geometrie ignoriert wird, dennoch können sie für die menschliche Wahrnehmung überzeugend sein und realistisch wirken. Schlagworte hierfür sind „Fake"-Global-Illumination oder auch „Quasi"-Global-Illumination. Ein bekanntes Beispiel für ein bildraum-basiertes Verfahren zur Annäherung einer globalen Beleuchtung mithilfe weicher Schatten ist Screen Space Ambient Occlusion (SSAO).

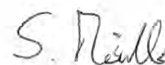
In dieser Studienarbeit soll untersucht werden, inwieweit sich die Sichtbarkeitsbestimmung im Bildraum nicht nur für nah gelegene Geometrie wie beim Ambient Occlusion, sondern in Bezug auf die gesamte Szene realisieren lässt. Aktuelle Ansätze werden dahingehend untersucht und das geeignetste Verfahren wird als Grundlage für die Implementierung eines Screen-Space Global Illumination Testszenarios genutzt (zum Beispiel „Screen-Space Radiosity").

Das umgesetzte Verfahren wird anhand verschiedener Testszenen bewertet.

Schwerpunkte dieser Arbeit sind:

1. Recherche aktueller Screen-Space Methoden zur Bestimmung der Sichtbarkeit
2. Einarbeitung in GPU-Programmierung (GLSL)
3. Implementierung eines Programms zur Analyse und Bewertung der recherchierten Sichtbarkeitstests
4. Integration eines ausgewählten Sichtbarkeitstests in ein Screen-Space GI Testszenario
5. Test des implementierten Verfahrens mit verschiedenen Szenen und Bewertung
6. Dokumentation der Ergebnisse

Koblenz, den 01. April 2009



- Prof. Dr. Stefan Müller-

Inhaltsverzeichnis

1 Einführung **1**

2 Beleuchtung **4**
2.1 Lokale Beleuchtungsmodelle 4
2.2 Globale Beleuchtungsmodelle 4
2.3 Ambient Occlusion 5
2.3.1 Definition 5
2.3.2 Berechnung 6
2.3.3 Bent Normal 7
2.3.4 Monte-Carlo-Integration 8
2.3.5 Existierende Verfahren 9
2.4 Obscurances 12

3 Voxelisierung mit der GPU **14**
3.1 Einführung 14
3.2 Oberflächenvoxelisierung 14
3.2.1 Die Voraussetzungen 14
3.2.2 Der Ablauf 16
3.3 Körpervoxelisierung 17
3.4 Erstellung von Slicemap-Mipmaps 19

4 Sichtbarkeit **22**
4.1 Problemstellung 22
4.2 Sichtbarkeitstest im Objektraum (Referenzlösung) 23
4.3 Sichtbarkeitstest im Bildraum (Screen-Space) 23
4.4 Sichtbarkeitstest im Voxelraum (Voxel-Space) 27
4.5 Bildraum vs. Voxelraum 29

5 Implementierung **31**
5.1 Überblick 31
5.2 G-Buffer erzeugen 32
5.3 Ambient Occlusion und indirektes Licht 34
5.3.1 Sampling 34
5.3.2 Gewichtung der Samples 38
5.3.3 Screen-Space AO und Screen-Space GI 39
5.3.4 Voxel-Space AO und Hybrid-GI 41
5.4 Blur 43
5.5 Finale Kombination 44

6 Ergebnisse **45**
6.1 Qualität 45
6.2 Performance 52
6.3 Erweiterungsmöglichkeiten 55

6.4 Ausblick: Strahlschnitttest im Voxelraum 55
6.5 Fazit 57

Literaturverzeichnis **58**

1 Einführung

"I am sure that at some point we will be ray tracing complex scenes on our palm pilots. Until then we'll continue to create efficient cheats and tricks to get the visual advantages of global illumination without the time and expense."
Hayden Landis, Industrial Light & Magic [Lan02, S. 97 ff.]

Die fotorealistische Bildsynthese beschäftigt sich mit der Frage, wie die Ausbreitung und Interaktion von Licht in einer virtuellen Szene so simuliert werden kann, dass ein menschlicher Betrachter das entstandene Bild nicht von einem Foto unterscheiden kann. Der Entstehungsprozess wird Rendering genannt. Rendering ist die Abbildung der dreidimensionalen Objekte einer virtuellen Szene mit all ihren definierten Eigenschaften auf ein zweidimensionales Bild. Hierbei sind folgende Faktoren zu berücksichtigen: Licht, Materialien, Geometrie, Kamera bzw. Betrachter.

Insbesondere die Simulation des Lichts ist dafür ausschlaggebend, ob eine Szene realistisch oder unrealistisch wirkt, selbst bei sehr detaillierter Modellierung der in ihr befindlichen Objekte. Es gibt zahlreiche Modelle und Verfahren, um die Beleuchtung eines Punktes in der Szene zu berechnen. Je nachdem, welche Informationen für diese Beleuchtung genutzt werden, handelt es sich um eine lokale oder globale Beleuchtung. Global ist sie dann, wenn außer den lokal für den aktuell betrachteten Punkt vorhandenen Informationen wie Position und Oberflächennormale auch Informationen über alle anderen Punkte mit in die Berechnung eingehen. Da andere Oberflächenpunkte je nach ihrer Materialeigenschaft jedoch auch wieder Licht reflektieren und damit andere Punkte beleuchten können, handelt es sich bei der Berücksichtigung solcher Interreflexionen um ein rekursives Problem: Um einen Punkt korrekt beleuchten zu können, müssen andere Punkte bereits beleuchtet sein [AMHH08, S. 379].

Entsprechend hoch ist der nötige Rechenaufwand, um eine globale Beleuchtungssimulation durchzuführen, vor allem bei physikalisch korrekten oder basierten Ansätzen. Für statische Szenen mit nicht-verformbaren Objekten, die sich ebenso wie alle Lichtquellen nicht bewegen, lassen sich die Beleuchtungsinformationen vorberechnen, in geeigneten Datenstrukturen ablegen und später für die Darstellung wieder abrufen.

Dies ist jedoch für dynamische Szenen nur unter bestimmten Voraussetzungen und nur teilweise möglich. In voll dynamischen Umgebungen mit sich bewegenden und animierten Objekten oder sich bewegenden Lichtquellen muss die Lichtsimulation fortlaufend neu berechnet werden.

Auch auf heutiger Hardware sind die von Hayden Landis im Eingangszitat genannten „Tricks" und vereinfachende Annahmen erforderlich, um dynamische Szenen interaktiv oder sogar in Echtzeit global zu beleuchten. Akenine-Möller et al. definieren Rendering als interaktiv ab einer Bilder-

zeugungsrate von 6 Frames pro Sekunde (fps), Echtzeit als 15 fps aufwärts [AMHH08, S. 1]. Nicht nur Anwendungen wie Spiele, in denen Echtzeit zwingend erforderlich ist, profitieren von Verfahren, die die Berechnung einer globalen Beleuchtung beschleunigen. Neben den komplett computergenerierten Animationsfilmen kommen in immer mehr Kinofilmen Spezialeffekte zum Einsatz. Die Produktionsfirmen sparen durch eine Beschleunigung des Renderings – selbst mit ihren Renderfarmen – Zeit und damit Geld.

Die Vereinfachungen und Näherungen sind immer ein Kompromiss zwischen der dadurch erzielten höheren Performance und der Bildqualität. Je nach Anwendung sind darüber hinaus wichtige Kriterien, ob ein ggf. zeitintensiver Vorverarbeitungsschritt notwendig ist oder ob komplizierte Datenstrukturen während des Renderings zu verwalten sind. Daraus ergibt sich auch, wie flexibel ein Verfahren ist und ob die Szenengeometrie zum Beispiel in spezieller Form vorliegen muss. Je nachdem, wie stark die Vereinfachung ist, sind Artefakte möglich, die das menschliche Auge stören oder gar nicht wahrgenommen werden. Auch hier ist es anwendungsabhängig, ob kleinere Fehler akzeptabel sind.

Verfahren im Objektraum arbeiten direkt auf der Szenengeometrie und sind abhängig von ihrer Komplexität. Eine Möglichkeit zur Beschleunigung ist daher, die tatsächliche Geometrie durch einfachere Geometrie anzunähern.

Um die Lichtberechnungen von der Szenenkomplexität zu entkoppeln, entstanden Algorithmen im Bildraum, die als grobe Szenenrepräsentation den Tiefenpuffer nutzen, d. h. die Tiefenwerte aller von der Kamera aus sichtbaren Objekte. Zudem beschränkt man sich in der Regel auf einen Radius, innerhalb dessen andere Szenenelemente berücksichtigt und außerhalb lie-

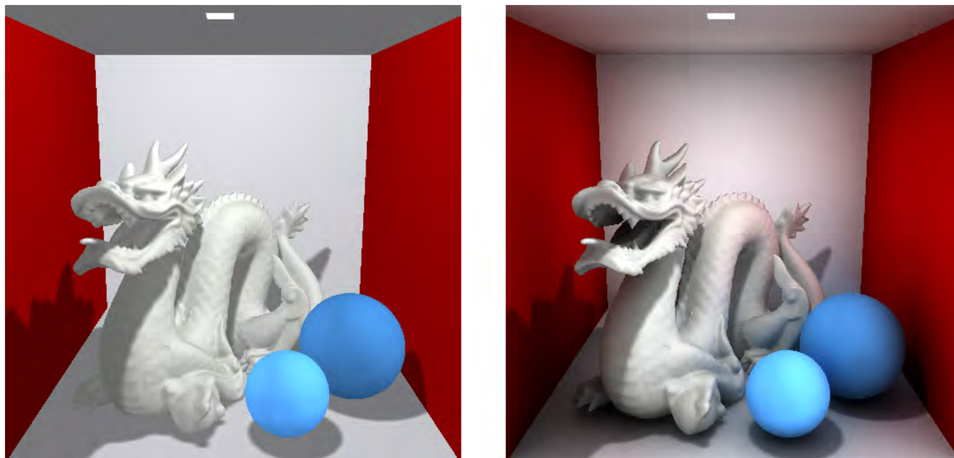


Abbildung 1: *Links:* Lokale Beleuchtung mit konstantem ambienten Licht und Shadow Mapping. *Rechts:* Angenäherte globale Beleuchtung im Voxel- und Bildraum.

gende ignoriert werden.

Für das Rendering mit globaler Beleuchtung sind Sichtbarkeitsanfragen essentiell: Können sich zwei Elemente der Szene sehen? Wird ein Punkt aus einer bestimmten Richtung verdeckt? Im Bildraum ist diese Information zwar nur begrenzt verfügbar, da er von der Kamera aus nicht sichtbare Teile nicht erfasst. Trotzdem können ansprechende Resultate bei gleichzeitiger Echtzeitfähigkeit erzielt werden. Eine viel benutzte Charakterisierung der Ergebnisse solch vereinfachender „Fake"-Verfahren ist „plausibel für die menschliche Wahrnehmung". Sehr populär ist ein Verfahren namens Ambient Occlusion, das in dieser Studienarbeit eine wesentliche Rolle spielt. Mithilfe von Ambient Occlusion lassen sich sehr weiche Schatten und eine globale Beleuchtung annäherungsweise berechnen.

Diese Studienarbeit befasst sich mit zwei Arten der Szenenrepräsentation (Bildraum und Voxelraum) und wie diese für eine Sichtbarkeitsbestimmung genutzt werden können. Dafür wurden bekannte Ambient-Occlusion-Verfahren untersucht und zwei ausgewählte Techniken implementiert. Diese wurden anschließend auf „Fake"-GI erweitert, das heißt, indirektes Licht wurde berücksichtigt (erste Indirektion). Die Umsetzung geschah im Kontext eines Deferred Shadings, das sich für bildraumbasierte Effekte anbietet. Für die Implementierung wurden OpenGL und die OpenGL Shading Language (GLSL) genutzt, sowie Qt für die Benutzeroberfläche der entstandenen Anwendung.

2 Beleuchtung

2.1 Lokale Beleuchtungsmodelle

Die *direkte Beleuchtung* bildet die Grundlage für jede weiterführende Beleuchtung. „Direkt“ heißt, dass nur die definierten Lichtquellen herangezogen werden, um die Szene zu beleuchten. Unter die direkte Beleuchtung fällt auch die Berechnung von *Schatten*, die durch Verdeckung der Lichtquellen entstehen.

Für Punktlichtquellen gilt: Entweder ist die Lichtquelle von einem Punkt aus gesehen verdeckt (Punkt liegt im Schatten) oder sichtbar (kein Schatten). Dieses binäre Ergebnis produziert harte Schattenkanten. In der Realität gibt es keine Punktlichtquellen außer der Sonne, die aber auch nur näherungsweise als eine solche betrachtet werden kann aufgrund ihrer großen Entfernung. Um realistische Schatten zu erzeugen, werden flächige Lichtquellen eingesetzt, durch deren Abtastung Halbschatten und weiche Schattenverläufe möglich sind.

Ist der Punkt unverdeckt oder wird die Sichtbarkeit der Lichtquelle nicht beachtet, wird ein *lokales Beleuchtungsmodell* für die Beleuchtung des Punktes herangezogen. Dieses beschreibt, wie das Material bei Interaktion mit Licht reagiert: Wie stark wird das Licht beim Auftreffen auf eine Oberfläche absorbiert, reflektiert oder transmittiert (und dabei vermutlich gebrochen)? Lokal bedeutet, dass jeder Punkt „für sich“ beleuchtet wird, unabhängig von den anderen Punkten in der Szene. Nur die Informationen über die Lichtquellen (Typ, Position, Lichtfarbe) und über die eigene Position, Oberflächennormale und Materialeigenschaften gehen in die Auswertung des lokalen Beleuchtungsmodells mit ein.

Materialien lassen sich mit Hilfe der *Bidirectional Reflectance Distribution Function* (BRDF) modellieren. Diese gibt für jeden Einfalls- und Ausfallswinkel an, wie das Verhältnis zwischen einfallender Beleuchtungsstärke und reflektierter Leuchtdichte ist, kann aber noch von weiteren Größen wie dem betrachteten Ort auf der Oberfläche abhängig sein. Sie lässt sich durch geschlossene mathematische Formeln oder durch Tabellen ausdrücken, in denen die jeweiligen Werte durch meist tausende Messungen erfasst wurden.

Für ideal diffuse Materialien (*Lambert-Strahler*) ist die BRDF konstant. Sie hängt nur von dem Reflexionsgrad des Materials ab. Für jede Einfallsrichtung wird die gleiche Leuchtdichte in alle Richtungen reflektiert, sodass das Material aus allen Richtungen betrachtet gleich hell erscheint. Ein weiteres klassisches Beleuchtungsmodell ist das Phong-Beleuchtungsmodell, das zusätzlich zur diffusen Reflexion Spiegelungen bzw. Glanz mit einbezieht.

2.2 Globale Beleuchtungsmodelle

In der Realität können Bereiche, die durch die direkte Beleuchtung dunkel bleiben, von ihrer (nahen) Umgebung beleuchtet werden. Licht, das auf eine

diffuse farbige Fläche – zum Beispiel eine rote Wand – fällt, wird in alle Richtungen reflektiert und beleuchtet somit indirekt andere Objekte in der Szene. Objekte in der Nähe der Wand erscheinen demnach leicht rötlich. Lokale Beleuchtungsmodelle berücksichtigen solche indirekten Phänomene nicht.

Die einfachste Möglichkeit, ein nicht näher definiertes indirektes Licht (gestreutes Umgebungslicht) zu simulieren, ist das Aufaddieren einer für die gesamte Szene konstanten Farbe. Dieser *ambiente Term* ist die gröbste Art der Annäherung einer *globalen Beleuchtung* (engl. *global illumination* = GI). Das Ergebnis ist jedoch weit von Wirklichkeitstreue entfernt.

Zwei klassische Ansätze zur globalen Beleuchtungssimulation sind Radiosity-Systeme und Path Tracing. Bei einer Radiosity-Simulation darf die Szene nur aus diffusem Material bestehen. Sie wird in viele kleine Flächenelemente (Patches) geteilt, zwischen denen Strahlungsaustausch stattfindet. Path Tracing geht über das klassische Ray Tracing hinaus, indem viele Strahlen durch jedes Pixel geschossen werden, allerdings pro Strahl nur ein einziger – zwar zufällig ausgewählter, gleichwohl aber vom jeweiligen Auftreffpunkt abhängiger – „Weg“ durch die Szene verfolgt wird. Um ein rauschfreies Ergebnis zu erhalten, sind sehr viele Strahlen pro Pixel nötig, entsprechend hoch ist die nötige Rechenleistung.

2.3 Ambient Occlusion

2.3.1 Definition

Ambient Occlusion (AO, deutsch: Umgebungsverdeckung) ist ein viel genutztes Verfahren zur Annäherung globaler Beleuchtung. Es deckt zwar nur eine Untermenge der globalen Effekte ab, kann aber dennoch den Realismusgrad eines Bildes beträchtlich steigern.

Das Verfahren modelliert, wie stark ein Punkt von der ihn umgebenden Geometrie verdeckt wird. Diesem Verdeckungsgrad entsprechend erreicht ihn viel oder wenig Licht eines imaginären diffusen Umgebungslichts, das die Szene aus allen Richtungen gleichmäßig beleuchtet (wie ein komplett bewölkter Himmel). Für die verdeckenden Objekte hat sich der Begriff *Occluder* eingebürgert.

Es ist ein Verfahren, das rein geometrisch basiert arbeitet und somit vollständig von der Beleuchtung durch die Lichtquellen einer Szene entkoppelt ist. Beide Berechnungen werden unabhängig voneinander ausgeführt und am Ende kombiniert, zum Beispiel durch eine einfache Multiplikation des Verdeckungsgrades mit der Farbe des lokal beleuchteten Punktes. Alternativ können die für jeden Oberflächenpunkt der Szene bestimmten Ambient-Occlusion-Werte als variierender ambienter Term angesehen werden.

Für Schatten, die durch Verdeckung der Lichtquellen entstehen, muss ein ergänzendes Schattenverfahren wie z. B. Shadow Mapping eingesetzt werden.

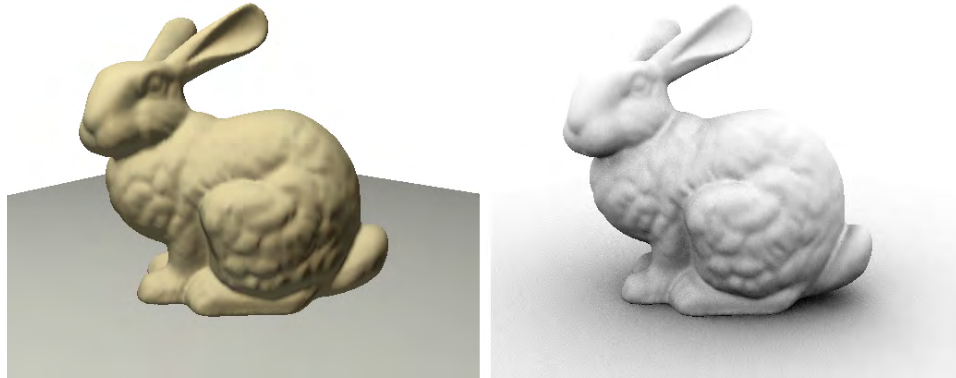


Abbildung 2: AO am Beispiel des Stanford Bunnys

Das Ergebnis von Ambient Occlusion sind zum einen extrem weiche Schatten (siehe Abbildung 2 (rechts), der Boden unter dem Bunny), die Kontaktstellen zwischen nahen Objekten und somit die räumliche Anordnung der Objekte verdeutlichen können. Zum anderen wird für einzelne Objekte durch Selbstverdeckung erkennbar, ob Stellen gewölbt oder gekrümmt sind. Die Heuristik „Vertiefungen sind dunkler als Erhebungen" („dark means deep" [LB99]) wird von Menschen zwar nicht alleine zur Erkennung von Oberflächenbeschaffenheiten herangezogen, spielt jedoch eine nicht außer Acht zu lassende Rolle.

2.3.2 Berechnung

Der Verdeckungsgrad wird für jeden Punkt P wie folgt bestimmt [AMHH08, S. 375]:

$$AO(P,\vec{n}) = \frac{1}{\pi} \int_{\vec{\omega}\in\Omega} \underbrace{V(P,\vec{\omega})}_{\text{Sichtbarkeit}} \cdot \underbrace{(\vec{\omega} \circ \vec{n})}_{\widehat{=}\text{ Cosinus des Winkels zwischen Normale } \vec{n} \text{ und Richtung } \vec{\omega}} d\vec{\omega} \tag{1}$$

[BS09, S. 426] weisen darauf hin, dass die Berechnung auch ohne Cosinus-Gewichtung möglich ist. In dem Fall lautet die Formel

$$AO(P,\vec{n}) = \frac{1}{2\pi} \int_{\Omega} V(P,\vec{\omega})\ d\omega. \tag{2}$$

Im Allgemeinen kommt jedoch die Variante mit Cosinusgewichtung zum Einsatz. Welche optischen Unterschiede sich ergeben, kann Abbildung 23 auf S. 35 in Abschnitt 5.3.1 Sampling entnommen werden.

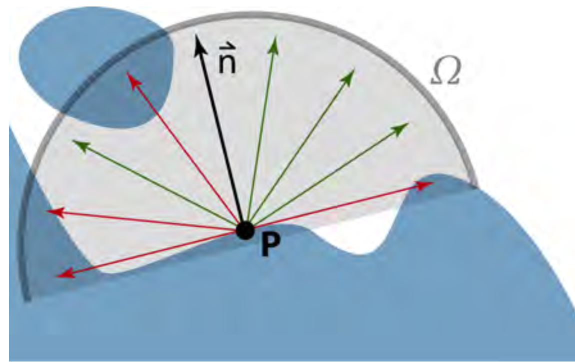


Abbildung 3: Ambient Occlusion Prinzip in 2D: An der Normalen $\vec{n}$ ausgerichtete Hemisphäre Ω um einen Punkt P, innerhalb welcher der Verdeckungsgrad an Punkt P bestimmt wird. Rote Richtungen sind verdeckt, grüne unverdeckt.

$AO(P, \vec{n})$ nimmt Werte aus $[0, 1]$ an, die prozentuale Verdeckung der Hemisphäre des Punktes P. Ein Wert von 1 bedeutet, dass der Punkt komplett unverdeckt („offen“) ist, bei einem Wert von 0 ist der Punkt komplett verdeckt. Aufgrund dieser Art der Berechnung (größerer Ergebniswert bedeutet weniger Verdeckung), ist ein häufiger Einwand gegen die Bezeichnung Ambient *Occlusion*, dass eigentlich der Grad der „Offenheit“ bestimmt wird – allerdings hat sich der Name Ambient Occlusion eingebürgert.

Oft wird statt des gesamten vorderen Halbraums die an der Normalen eines Punktes ausgerichtete Hemisphäre betrachtet, also eine Halbkugel mit festem, endlichen Radius. Ohne Beschränkung auf einen Radius wären geschlossene Räume schwarz, da hier jeder Punkt in allen Richtungen verdeckt wäre. Ein anderer Ansatz, um dieses unerwünschte Ergebnis zu vermeiden, ist die Abschwächung des Verdeckungsgrades (siehe Abschnitt 2.4 Obscurances, S. 12). Der geeignete Radius muss manuell pro Szene bestimmt werden.

2.3.3 Bent Normal

Während der Berechnung kann auch die sogenannte *Bent Normal* (deutsch: geneigte Normale) durch Mittelung aller unverdeckten Richtungen bestimmt werden. Sie zeigt die Richtung, in der sich im Durchschnitt die wenigsten oder keine Occluder befinden. Im Allgemeinen zeigt die Bent Normal potentiell in eine andere Richtung als die eigentliche – zuvor definierte – Oberflächennormale. Sie kann für eine spätere Beleuchtung genutzt werden, häufig in Kombination mit einer diffusen Environment Map. Abbildung 4 zeigt den Unterschied zwischen den Normalen.

Es gibt allerdings Konstellationen, in denen die Ausrichtung der Bent Normal nicht definiert ist (mathematisch zwar schon, der Bedeutung nach aber nicht). Somit kann es passieren, dass sie in eine eigentlich verdeckte

Richtung zeigt. Wenn zum Beispiel über dem Punkt oben mittig in der Hemisphäre ein Occluder schwebt, und der Punkt ringsherum in den flachen Richtungen unverdeckt ist, heben sich diese im Durchschnitt auf. Die Bent Normal wird wieder der ursprünglichen Oberflächennormalen ähneln, obwohl genau diese Richtung verdeckt ist.

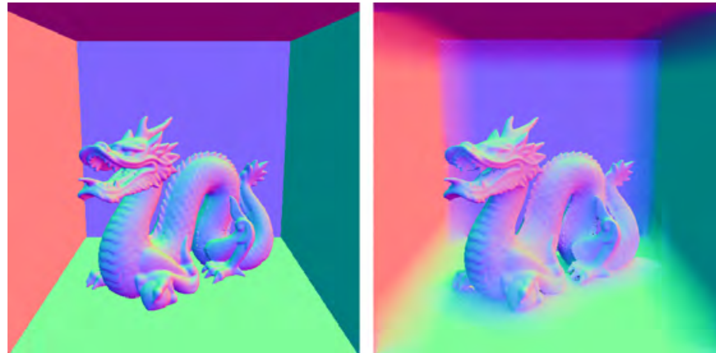


Abbildung 4: Links: die definierten Normalen; rechts: Bent Normals, berechnet mit SSAO (Ray Marching)

2.3.4 Monte-Carlo-Integration

Das Ambient-Occlusion-Integral kann nicht analytisch gelöst werden. Daher wird oftmals Monte-Carlo-Integration zur Annäherung der Lösung eingesetzt, also die diskrete Abtastung der Hemisphäre mittels zufällig generierter Richtungen (= Stichproben oder *Samples*). Die allgemeine Formulierung für das Abschätzen eines Integrals mittels Monte-Carlo-Integration zeigt Gleichung (3). Der rechte Teil der Gleichung wird auch als Monte-Carlo-Schätzer bezeichnet.

$$\int_a^b f(x)dx \approx \frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)} \tag{3}$$

mit

N	Anzahl der Samples
X_i	i-tes Sample
$p(X_i)$	Dichtefunktion

Die Ambient-Occlusion-Formeln (1) und (2) integrieren jeweils über eine Hemisphäre Ω mit Raumwinkel 2π. Für die Monte-Carlo-Integration müssen als Samples zufällige Richtungen innerhalb der Hemisphäre ausgewählt werden, die jeweils einen gewissen Raumwinkel repräsentieren.

Die Samples für eine Einheits-Hemisphäre mit Radius = 1 sind in Kugelkoordinaten ausgedrückt zweidimensional, nämlich ein Winkelpaar (ϕ, θ) (genannt Azimut- und Polarwinkel). Sie beschreiben im dreidimensionalen Raum innerhalb einer Kugel oder einer Hemisphäre eine Richtung und können in kartesische Koordinaten (x, y, z) umgerechnet werden.

Es bietet sich an, die Dichtefunktion ähnlich dem Integranden des Integrals zu wählen (*Importance Sampling*). Zur Generierung der Samples für die Abschätzung des Integrals (2) ist eine konstante Dichtefunktion $p(\phi, \theta) = \frac{1}{2\pi}$ sinnvoll, die in einem uniformen Sampling in Bezug auf den Raumwinkel der Hemisphäre resultiert. Jede Richtung wird also mit gleicher Wahrscheinlichkeit erzeugt. Für die Abschätzung des Integrals (1) ist dagegen ein cosinusgewichtetes Sampling mit der Dichtefunktion $p(\phi, \theta) = \frac{cos\theta}{\pi}$ (mehr Samples in Richtung der Normalen) zweckmäßig. Der Cosinus-Term des Integrals nimmt dann hohe Werte an, wenn der Winkel zwischen Normale und Richtung klein ist. Bei sehr großem Winkel würde ein Sample nicht viel zur Summe beitragen.

In beiden Fällen werden die Ergebnisse der Sichtbarkeitstests aufaddiert und durch die Anzahl der Samples geteilt:

$$\frac{1}{N} \sum_{i=1}^{N} V(X_i)$$

2.3.5 Existierende Verfahren

Es gibt zahlreiche Algorithmen zur Berechnung des Verdeckungsgrades. Sie werden kategorisiert in *Inside-out*-Verfahren und *Outside-in*-Verfahren. Die Bezeichnung weist darauf hin, aus welchem Blickwinkel die Berechnung des Verdeckungsgrads geschieht. Inside-out heißt, dass der Punkt von sich aus nach außen in die Umgebung „schaut“. Dieser Betrachtungsweise entspricht auch die sehr rechenaufwändige Referenzlösung mittels Ray Tracing. Jene wird daher nur im Offline-Rendering eingesetzt. Outside-in bedeutet, dass der betreffende Punkt von außen betrachtet wird. Das Umgebungslicht (repräsentiert durch mehrere imaginäre Lichtquellen, von denen aus die Szene gerendert wird) „testet“, wieviel es von dem Punkt sieht.

Seit Hayden Landis das Konzept von Ambient Occlusion im Jahr 2002 veröffentlichte, welches damals in einem Offline-Renderer zum Einsatz kam, wurde die Technik rasch populär. Die Tendenz ging und geht immer mehr hin zu Verfahren, die AO in Richtung Interaktivität und sogar Echtzeit in voll dynamischen Szenen bewegen.

Martin Knecht unterscheidet in seinem „State of the Art Report on Ambient Occlusion“ aus dem Jahr 2007 [Kne07] zwischen *objektbasierten*, *vertexbasierten* und *bildbasierten* Methoden. Erstere arbeiten auf ganzen Objekten, zweitere betrachten einzelne Vertices und letztgenannte nutzen den blickwinkelabhängigen Bildraum. Im Jahr 2009 stellten Reinbothe et al. das

voxelbasierte „Hybrid Ambient Occlusion" vor [RBA09]. Sie nutzen den voxelisierten Objektraum und den Bildraum für ein intelligentes Hochskalieren der nicht vollaufgelösten Ambient-Occlusion-Textur.

Vertexbasiert ist zum Beispiel ein von Michael Bunnell 2005 vorgestelltes Verfahren [Bun05], das die Geometrie in Form vieler Scheiben annähert. Jede Scheibe ist an einem Vertex zentriert und gemäß der Normalen dieses Vertex ausgerichtet. Das Verfahren ist auf eine ausreichend hohe Auflösung der Meshes angewiesen, da die Annäherung durch Scheiben ansonsten zu grob ist.

Bildbasierte Verfahren nutzen den Tiefenpuffer bzw. eine Vielzahl an Texturen, die zusammen oft als *G-Buffer* bezeichnet werden (von Geometrie-Buffer). Abbildung 5 zeigt Beispiele für mögliche Texturen bzw. Texturkanäle.

Abbildung 5: Beispiele für G-Buffer-Texturen bzw. Kanäle; von links nach rechts: Farbe/Material, Beleuchtungsstärke, Normalen, Tiefe, Kamerakoordinaten

Exkurs: G-Buffer und Deferred Shading

Deferred Shading bezeichnet das Prinzip, zunächst die Szene in Texturen zu rendern und anschließend auf diesen Texturen zu arbeiten, um das fertige Bild zu erzeugen.

Der G-Buffer wird beim Deferred Shading im ersten Pass erzeugt. Alle nötigen Informationen (Tiefe, Kamerakoordinaten oder Weltkoordinaten; Normale; Farbe; Material) werden im Fragment Shader in dafür zuvor angelegte Texturen (= Render Targets eines Framebuffers) geschrieben.

Dabei fallen – je nach Bildauflösung – große Datenmengen an, weshalb eine möglichst geschickte Anordnung und Nutzung aller Kanäle in den Texturen angestrebt wird. Bei einer späteren Weiterverarbeitung, die den Zugriff auf die Texturen benötigt, können Grafikkarten mit wenig Texturspeicher und Bandbreite ausgebremst werden.

Vor diesen eben beschriebenen ersten Render-Pass kann ein „Early-Z"-Pass geschoben werden. Währenddessen wird alles außer dem Schreiben der Tiefenwerte in den Tiefenpuffer deaktiviert. Beim anschließenden

> Rendern der Szene werden über den Tiefentest frühzeitig nicht-sichtbare Pixel aussortiert, sofern die Hardware „Early-Z“ unterstützt, d. h. das Verwerfen eines Fragments früh in der Pipeline, vor dem Fragment Shader. Es ist zu bedenken, dass die Geometrie doppelt gerendert werden muss – auch wenn der „Nur-Tiefen“-Pass schnell ist, kostet dieser Zeit. Deshalb ist diese Vorgehensweise nur dann sinnvoll, wenn der anschließende Pass extrem komplizierte Berechnungen im Fragment Shader beinhaltet.

Das erste Mal wurde Ambient Occlusion im Bildraum im Kontext des Deferred Shadings in Cryteks CryEngine 2 bzw. ihrem Computerspiel *Crysis* unter dem Namen **Screen Space Ambient Occlusion** (SSAO) vorgestellt [Mit07], Details in: [Kaj09]. Die Idee ist, nur den Tiefenpuffer als Grundlage für eine Per-Pixel-Verdeckungsberechnung im Fragment Shader zu nutzen. Der Tiefenpuffer stellt eine grobe Repräsentation der Szenengeometrie dar. Das Verfahren spannt um jeden Punkt eine Kugel mit gewissem Radius auf, in der Samples (Punkte) generiert werden. Für jedes Sample wird getestet, ob es von dem betrachteten Punkt aus sichtbar oder verdeckt ist. Die Ergebnisse werden akkumuliert, wobei die Hälfte der Samples als ungültig angenommen wird, da sich diese im hinteren Halbraum des Punktes befinden. Wie die Sichtbarkeit im Bildraum bestimmt werden kann, ist in Abschnitt 4.3 (S. 23 ff.) beschrieben. Ein allgemeines Problem besteht dabei darin, dass von der Kamera aus nicht-sichtbare Objekte nicht erfasst sind.

Seit dieser „Geburtsstunde“ erfreuen sich AO-Verfahren im Bildraum großer Beliebtheit. Solche sind äußerst flexibel und können auf alles angewendet werden, was in einen Tiefenpuffer rasterisiert werden kann. Es gibt weder Einschränkungen für die Geometrie noch die Notwendigkeit spezielle Datenstrukturen zu nutzen oder langwierige Vorverarbeitungsschritte durchzuführen.

Screen-Space Directional Occlusion wurde 2009 von Ritschel et al. [RGS09] vorgestellt. Es bietet die interessante Neuerung, die Entkopplung von direkter Beleuchtung und ambienter Verdeckung aufzuheben und stattdessen beides zeitgleich durchzuführen. Es entstehen dadurch potentiell gerichtete und farbige Verschattungen, die im Worst Case genauso aussehen wie „normales“ Ambient Occlusion.

Im Rahmen von NVIDIA's **Image-Space Horizon-Based Ambient Occlusion** [BSD08] (Details in: [BS09]) stellen die Autoren das **Ray Marching** als Referenzlösung für Ambient Occlusion im Bildraum vor. Die Werte des Tiefenpuffers der Szene repräsentieren ein Höhenfeld (*height field*). Ray Marching ist ein Brute-Force Algorithmus, der jeden für die Annäherung des Ambient-Occlusion-Integrals erzeugten Strahl schrittweise daraufhin testet, ob und an welcher Stelle er das Höhenfeld schneidet. Der erste Sample-Punkt, der unterhalb des Höhenfeldes liegt, wird als Schnittpunkt angenommen.

Ray Marching wird ebenfalls in dem bereits erwähnten **Hybrid Am-**

bient Occlusion eingesetzt. Anstelle des Tiefenpuffers als grobe Szenenrepräsentation nutzen Reinbothe et al. eine dynamische **Voxelisierung** der Szene, die ebenfalls in nur einer 2D-Textur speicherbar ist. Auch hier wird jeder Strahl auf den ersten Schnittpunkt mit der Szene getestet. Dies ist der erste Sample-Punkt, der innerhalb eines vollen Voxels liegt. Die Voxelisierung (genaueres dazu in Abschnitt 3 auf S. 14 ff.) wird pro Frame in nur einem einzigen anfänglichen Pass durchgeführt. Der Vorteil besteht darin, dass nicht nur von der Kamera aus sichtbare Objekte, sondern die gesamte Geometrie von der „Voxelisierungskamera“ erfasst wird und für die Ambient-Occlusion-Berechnung genutzt werden kann.

2.4 Obscurances

Ambient Occlusion ist eine Vereinfachung von Obscurances (engl. *to obscure* = verdecken), die zum ersten Mal 1998 von Zhukov et al. vorgestellt wurden [ZIK98]. Sie definieren Obscurance als geometrische Eigenschaft, die angibt, wie offen ein gegebener Oberflächenpunkt ist.

Der einzige wesentliche Unterschied ist, dass bei Obscurances der Abstand zum ersten Occluder berücksichtigt wird. Der binäre Visibilitätsterm aus dem Ambient-Occlusion-Integral wird ersetzt durch eine Distanzabbildungsfunktion, die von dem Abstand zum ersten Occluder in der untersuchten Richtung abhängig ist. Das heißt, es muss nicht mehr nur ermittelt werden, ob eine Richtung verdeckt ist, sondern welches der erste Occluder ist bzw. wo sich der erste Schnittpunkt in dieser Richtung befindet. Das Sichtbarkeitsergebnis wird pro Sample berechnet, indem die stetige Distanzabbildungsfunktion auf den Abstand angewendet wird.

$$Obsc(P,\vec{n}) = \frac{1}{\pi} \int\limits_{\vec{\omega} \in \Omega} \underbrace{\rho(d(P,\vec{\omega},d_{max}))}_{\substack{\text{Distanz-}\\ \text{abbildungs-}\\ \text{funktion}}} \cdot (\vec{\omega} \circ \vec{n})\ d\vec{\omega} \qquad (4)$$

mit	$d(P,\vec{\omega},d_{max})$	Abstand zum ersten Schnittpunkt in Richtung $\vec{\omega}$
	d_{max}	maximaler Abstand von Punkt P, innerhalb dessen nach Schnittpunkten gesucht wird

$\rho()$ hat einen Wertebereich von [0,1] (siehe Abbildung 6), $d()$ liegt innerhalb $[0, d_{max}]$. Die Annahme ist, dass nahe Objekte einen Punkt stärker verdecken als weit entfernte. Der Wert von $\rho()$ ist dann 1, wenn der erste gefundene Schnittpunkt einen Abstand $\geq d_{max}$ hat. Aus diesem Grunde kann die Schnittpunktsuche direkt auf einen Radius von d_{max} beschränkt werden. Im Falle keines Fundes innerhalb dieses Bereichs ist $d() = d_{max}$. Es ist natürlich auch möglich, $d_{max} = \infty$ zu setzen, falls keine solche Einschränkung des Suchradius gewünscht ist. Je nachdem, welche Abmessungen die

Szene hat, muss ein passender Wert für d_{max} manuell bestimmt werden, um ansprechende Ergebnisse zu erzielen.

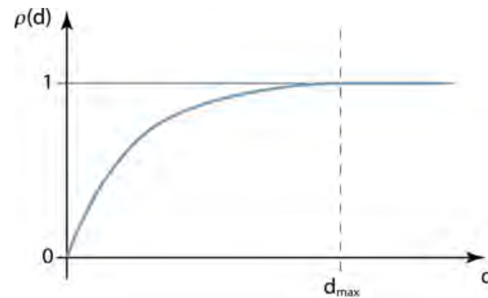


Abbildung 6: Beispiel-Plot einer Distanzabbildungsfunktion nach [ZIK98, S. 47]

Viele Verfahren nennen sich zwar Ambient Occlusion, da dies der populärere Begriff ist. Sie nutzen für die Berechnung aber eigentlich die Formel für Obscurances, sofern der für die Formel benötigte Abstand zum ersten Schnittpunkt bekannt ist. Die Abbildung 25 auf S. 39 zeigt verschiedene Funktionen und die Ergebnisse.

Durch eine Erweiterung des Obscurance-Algorithmus können die Farb-/ Materialinformationen an den Schnittpunkten zum Einsammeln von farbigem indirektem Licht genutzt werden [Mén07, S. 47 ff.] (*Color Bleeding*, siehe Abbildung 7).

Abbildung 7: *Links:* Ambient Occlusion, *rechts:* Obscurances mit Color Bleeding. Bilder entnommen aus [Mén07, S. 31]

3 Voxelisierung mit der GPU

3.1 Einführung

Der Begriff *Voxel* ist an das Kunstwort *Pixel* (von Picture Element) angelehnt und bezeichnet ein diskretes Volumenelement. Die zweidimensionalen Pixel entstehen durch Diskretisierung eines kontinuierlichen Bildes, die dreidimensionalen Voxel durch die Diskretisierung eines Raumes und der darin befindlichen Objekte. Im Normalfall handelt es sich bei der Zellenstruktur, in der die Voxel angeordnet sind, um ein reguläres Gitter. Sie können beliebige Daten enthalten. Ebenso wie bei Bildern und Pixeln ist die Auflösung des dreidimensionalen Gitters ein wichtiges Kriterium dafür, wie gut die diskreten Voxel die kontinuierlichen Objekte annähern.

In der Medizin werden Volumendatensätze zum Beispiel mit Computertomographen (CT) generiert.

Aber auch außerhalb der medizinischen Bildverarbeitung bieten Voxel interessante Möglichkeiten. Anwendungsbeispiele aus der Computergrafik sind Partikelsimulationen, Berechnungen von Schnittvolumen zweier Objekte, Kollisionserkennung oder auch Sichtbarkeitsanfragen für beispielsweise Ambient Occlusion.

Da die virtuellen Objekte in der Computergrafik aber in aller Regel als Polygonnetze vorliegen, müssen diese zunächst in eine geeignete Voxelrepräsentation überführt, d. h. voxelisiert, werden.

Das in diesem Abschnitt beschriebene Verfahren zur Voxelisierung von Meshes auf der GPU wurde von Eisemann und Décoret vorgestellt. 2006 veröffentlichten sie das Paper *Fast Scene Voxelization and Applications* [ED06], zwei Jahre später *Single-Pass GPU Solid Voxelization for Real-Time Applications* [ED08]. Das zweite modifizierte das erste Verfahren so, dass Voxel nicht nur für die Meshoberflächen generiert werden, sondern auch für das Innere eines geschlossenen Objektes.

Ziel dieser Voxelisierung ist es, die Information zu erfassen, ob sich an einer bestimmten diskreten Stelle in der Szene etwas befindet, also eine binäre Angabe über die Existenz von Objekten in einer Szene.

Im Folgenden wird zunächst die Oberflächenvoxelisierung (*boundary voxelization*) erläutert, danach die Körpervoxelisierung (*solid voxelization*), auf die sich der Algorithmus durch minimale Änderungen übertragen lässt.

3.2 Oberflächenvoxelisierung

3.2.1 Die Voraussetzungen

Das Verfahren arbeitet mit bitweisen logischen Operationen und beruht darauf, dass seit Shader Model 4.0 der Datentyp (Unsigned) Integer auf der GPU verfügbar und der Zugriff auf (Unsigned) Integer Texturen möglich ist.

Bei Verwendung von OpenGL ist mindestens Version 2.0 erforderlich, da erst ab dieser Version das interne Texturformat für Unsigned Integer unterstützt wird.

Eine Voxelisierung mit OpenGL und GLSL benötigt:

- ein Framebuffer-Objekt mit einer 2D-Textur als Render Target,
  internalFormat = `GL_RGBA32UI_EXT`,
  pixelFormat = `GL_RGBA_INTEGER_EXT`,
  pixelType = `GL_UNSIGNED_INT`
- eine 1D-Textur (*bitmask*), deren Texel den gleichen Datentyp wie jene des Render Targets haben
  (Aufbau der Textur: siehe Abbildung 9 auf S. 17)
- eine Grafikkarte, die Shader Model 4.0 unterstützt
- das zu voxelisierende 3D-Modell

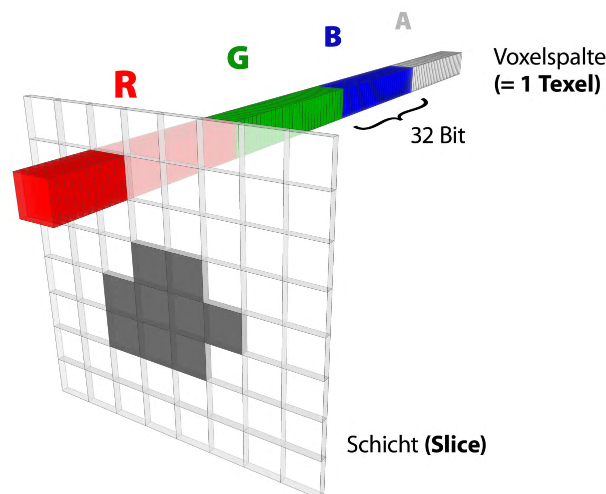


Abbildung 8: Slicemap mit Auflösung 8×8: Die beispielhaft herausgegriffene Schicht besteht aus 8×8 Bits, welche jeweils anzeigen, ob sie ein volles (1, hier: dunkelgrau) oder leeres (0, hier: hellgrau) Voxel repräsentieren. Ein Texel der Slicemap hat 128 Bits, es gibt also 128 solcher Schichten.

Die gesamte voxelisierte Szene wird in der 2D-Unsigned-Integer-Textur abgespeichert. Diese Voxel-Textur wird als *Slicemap* (slice = Schicht) bezeichnet und im Folgenden so genannt. Das Prinzip ist in Abbildung 8 schematisch dargestellt. Alle k-ten Bits der Texel der Slicemap bilden zusammen eine Schicht.

Ein Bit entspricht einem Voxel und enthält die Information, ob dieser voll (1) oder leer (0) ist. Ein Texel mit dem internen Format `GL_RGBA32UI_EXT` enthält 128 Bits (= 4 Kanäle · 32 Bits pro Kanal). Dies entspricht 128 Voxel-Schichten in z-Richtung.

3.2.2 Der Ablauf

Die Slicemap ist das Render Target des zum Zeitpunkt der Voxelisierung genutzten Framebuffer-Objekts. Um ein reguläres Voxelgitter zu erhalten, in dem die Voxel würfel- bzw. quaderförmig sind, ist eine Voxelisierungskamera mit *orthographischer Projektion* zu setzen. Das Frustum dieser Kamera umfasst die Szene bzw. den Teil der Szene, der voxelisiert werden soll. Die Auflösung der Slicemap in x- und y-Richtung entspricht der Auflösung des zum Zeitpunkt der Voxelisierung gesetzten *Viewports*. Die Auflösung in z-Richtung ist die Anzahl der *Bits pro Texel*, also 128 bei dem beschriebenen internen Format. Eine Slicemap der Auflösung 512×512 enthält $512 \times 512 \times 128$ Voxel. Die Auflösung in z-Richtung kann durch den Einsatz von Multiple Render Targets erhöht werden. Mit zwei Slicemaps, also zwei Render Targets, beträgt die Auflösung in z-Richtung folglich 256.

Beim Rendern der gesamten Geometrie dürfen weder Vertices durch Culling noch Fragments am Ende der Pipeline verworfen werden. Das Culling wird ausgeschaltet mit `glDisable(GL_CULL_FACE)` und es wird kein Render-Buffer für den Tiefentest an das Framebuffer-Objekt gehängt. Somit gelangt jeder innerhalb des Voxelisierungsfrustums gelegene und rasterisierte Vertex zum Fragment Shader und jedes Fragment in den Framebuffer.

Listing 1: Voxelisierung: GLSL Vertex Shader

```
1  #version 120
2  varying float mappedZ;
3
4  void main()
5  {
6    // fixed function pipeline
7     gl_Position  = ftransform();
8     // map z-coordinate to [0,1]
9     mappedZ = (gl_Position.z+1.0)/2.0;
10 }
```

Jeder Punkt wird mit `ftransform()` (= `gl_ModelViewProjectionMatrix * gl_Vertex`) in homogene Koordinaten transformiert. Bei einer orthographischen Projektion entspricht dies bereits dem kanonischen Volumen mit Koordinaten von -1 bis 1 in allen drei Raumrichtungen. Der z-Wert kann deshalb direkt in Fensterkoordinaten umgerechnet und an den Fragment Shader weitergereicht werden (siehe Listing 1).

Der lineare z-Wert `mappedZ` zwischen 0 und 1 wird im Fragment Shader genutzt, um zu bestimmen, an welcher Stelle der 128 Bits des aktuellen

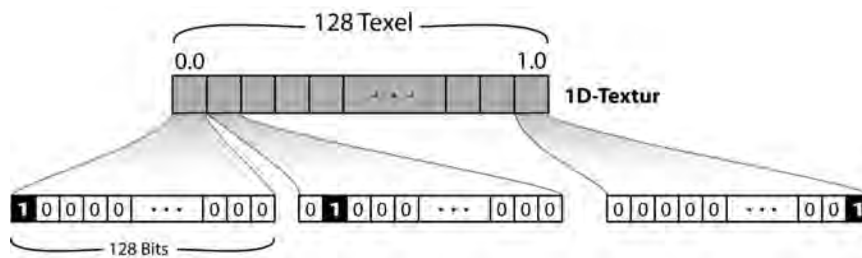


Abbildung 9: Bitmaske für die Oberflächenvoxelisierung

Texels in der Slicemap ein Bit auf 1 gesetzt wird (voller Voxel). Dafür wird die Bitmaske (1D-Textur) benutzt.

Ein Lookup in der Bitmaske mit *Texturkoordinate* = ***mappedZ*** ermittelt die dem z-Wert entsprechende Bitposition.

Listing 2: Voxelisierung: GLSL Fragment Shader

```
#version 120
// Shader Model 4.0 for Integer Arithmetics
#extension GL_EXT_gpu_shader4 : require

varying float mappedZ;
varying out uvec4 result;

uniform usampler1D bitmask;

void main()
{
   result = texture1D(bitmask,mappedZ);
}
```

Die Bits werden am Ende der Pipeline mit den bereits im Framebuffer vorhandenen Werten durch eine logische Operation verodert. Die OpenGL-Befehle zum Einschalten und Definieren der logischen Operation lauten `glEnable(GL_COLOR_LOGIC_OP)` und `glLogicOp(GL_OR)`.

Auf diese Weise werden alle Flächen voxelisiert. Problematisch sind allerdings Flächen, die parallel oder nahezu parallel zur Blickrichtung der Kamera verlaufen, weil diese nicht oder kaum erfasst werden (Abbildung 10).

3.3 Körpervoxelisierung

Bei der Oberflächenvoxelisierung bleibt das Innenleben der Objekte leer. Vor allem bei der Bestimmung, ob sich etwas an einer Stelle in der Szene befindet, ist es sinnvoll, auch die Voxel innerhalb eines Meshes zu füllen. Liegt bei der Frage „Befindet sich etwas an Position xy in der Szene?“ die Position innerhalb eines Objektes, sollte die Antwort eigentlich „ja“ lauten, weil das Objekt die Position umschließt. Im Falle der Oberflächenvoxelisierung wäre die Antwort jedoch „nein“, da alle Voxel im Objektinneren leer sind.

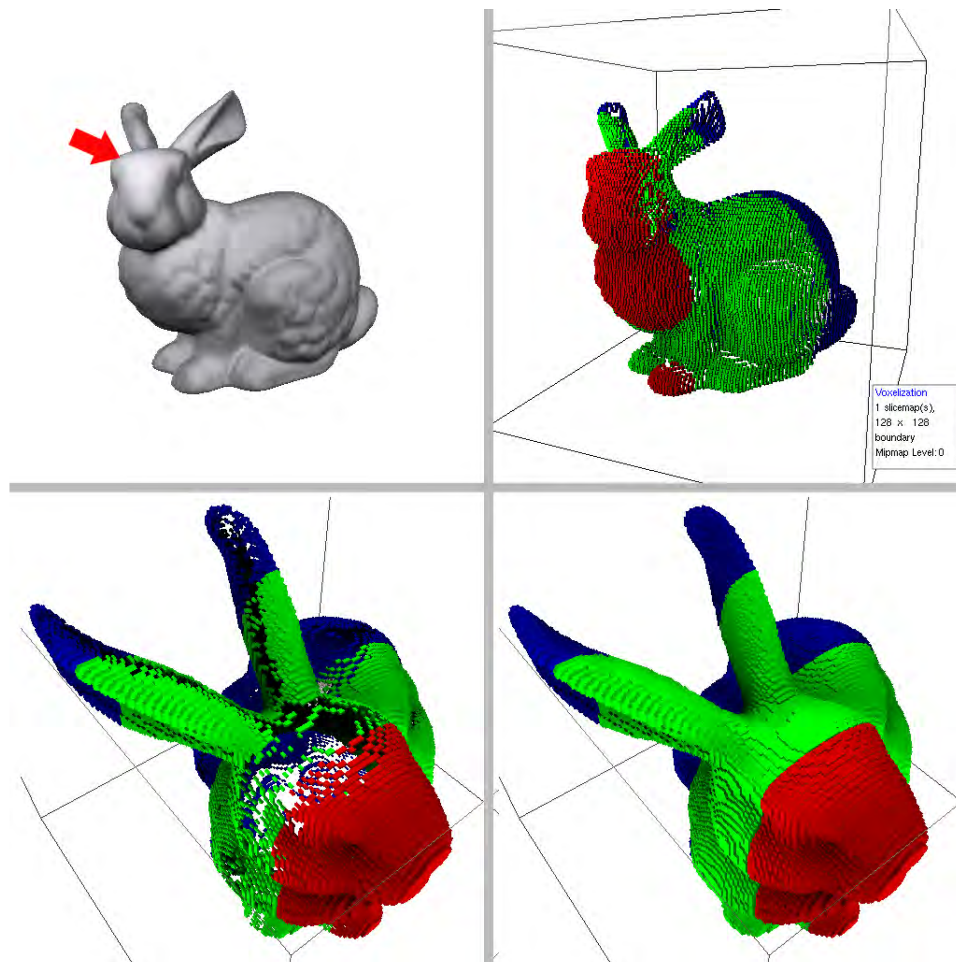


Abbildung 10: Zur Kamerablickrichtung parallele Flächen werden bei der Oberflächenvoxelisierung kaum erfasst. *Oben links:* Das zu voxelisierende Stanford Bunny. *Oben rechts:* Oberflächenvoxelisierung, Voxel dargestellt als Quader. Die Farben symbolisieren, in welchem Kanal (Rot, Grün, Blau oder Alpha) das Voxel abgespeichert wurde. *Unten links:* Die im Bild oben links mit dem roten Pfeil markierte Oberseite des Bunny-Kopfes weist in der Voxelrepräsentation eine große Lücke auf. *Unten rechts:* Bei der Körpervoxelisierung tritt dieses Problem nicht auf.

Mit einer kleinen Änderung des obigen Algorithmus wird das Objektinnere gefüllt. Es muss lediglich eine andere Bitmaske und eine andere logische Operation verwendet werden, nämlich das Exklusiv-Oder `glLogicOp(GL_XOR)`. Die 128 Bits der Bitmasken-Texel folgen dem Muster:
`[11111...1]`, `[01111...1]`, `[00111...1]`,... , `[0000...01]`.

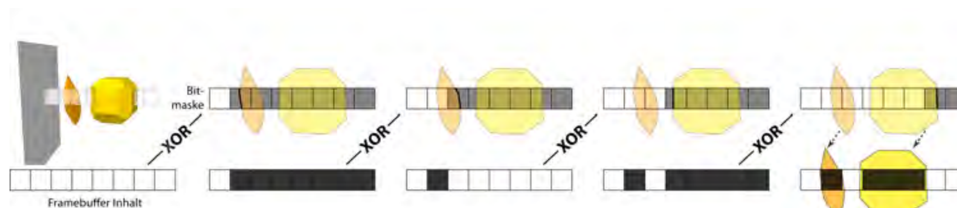


Abbildung 11: Veranschaulichung des Prinzips der Körpervoxelisierung. Die Grafik zeigt von links nach rechts die Füllschritte für ein Texel (eine Voxelspalte) der Slicemap. Das Texel hat der Einfachheit halber nur 8 Bits, ebenso der Framebuffer, welcher anfangs leer ist (alle Bits sind 0). In der Grafik bedeutet weiß = 0, grau = 1. Bei dem Ergebnis (ganz rechts) ist ein Detail zu erkennen: Die Voxel sind um etwa eine halbe Voxellänge in z-Richtung relativ zur Geometrie verschoben. Durch einen Offset lassen sich Geometrie und Voxel zur Deckung bringen. (Abbildung nach [ED08, S. 3], dort findet sich auch eine mathematische Begründung für die Voxelisierung via XOR-Operation.)

Das Mesh muss die Anforderung erfüllen, *watertight* zu sein (anschaulich: „wasserdicht“ = keine Löcher). [ED08, S. 3] definieren ein Modell als watertight, wenn für jeden zusammenhängenden Bereich im Raum gilt, dass all seine Punkte entweder innenliegend oder außenliegend sind. Gemäß des *Jordan Theorems* liegt ein Punkt im Inneren, wenn für jeden beliebigen Strahl, dessen Ursprung dieser Punkt ist, die Anzahl der Schnittpunkte mit dem Modell ungerade ist. Analog liegt er außen, wenn die Anzahl der Schnittpunkte gerade ist. Abbildung 12 veranschaulicht diesen Sachverhalt.

Löcher im Mesh würden zu einer fehlerhaften Voxelisierung führen, die in Abbildung 13 gezeigt wird.

3.4 Erstellung von Slicemap-Mipmaps

Aktuelle Grafikkarten unterstützen die automatische Erzeugung von Mipmaps bei Integer-Texturen nicht. Da aber ohnehin die volle Kontrolle über die Art der Reduzierung gegeben sein muss, wird ein eigener Fragment Shader zum sukzessiven Zusammenführen der Voxelspalten eingesetzt. Räumlich gesehen handelt es sich um eine Reduzierung in x- und y-Richtung, nicht jedoch in z-Richtung, da die Texel ihr Format von 128 Bits behalten. Somit entsteht eine Datenstruktur, die als Quadtree angesehen werden kann (statt des üblichen Octrees für Voxelhierarchien). Um einen echten Octree zu er-

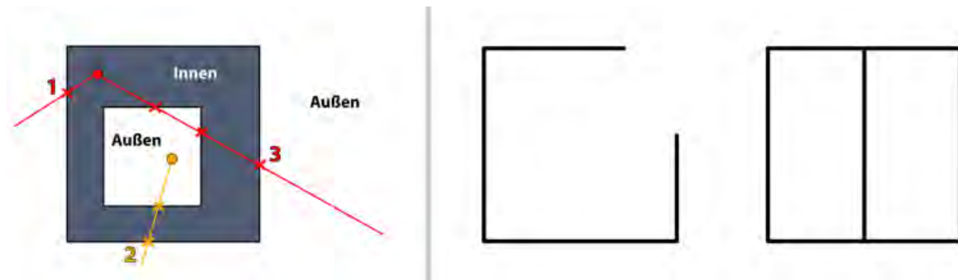


Abbildung 12: *Links:* Jordan Theorem: Beim Szenario „Kleine Box in großer Box" wird der Zwischenraum der Boxen als „Innen" angenommen: Ausgehend von dem Punkt im Inneren des Objekts (dunkelblau) haben alle Strahlen eine ungerade Anzahl an Schnittpunkten mit den Objektgrenzen. *Rechts:* Geometrie, die nicht watertight ist, da Innen und Außen nicht wohldefiniert ist (nach [ED08, S. 2])

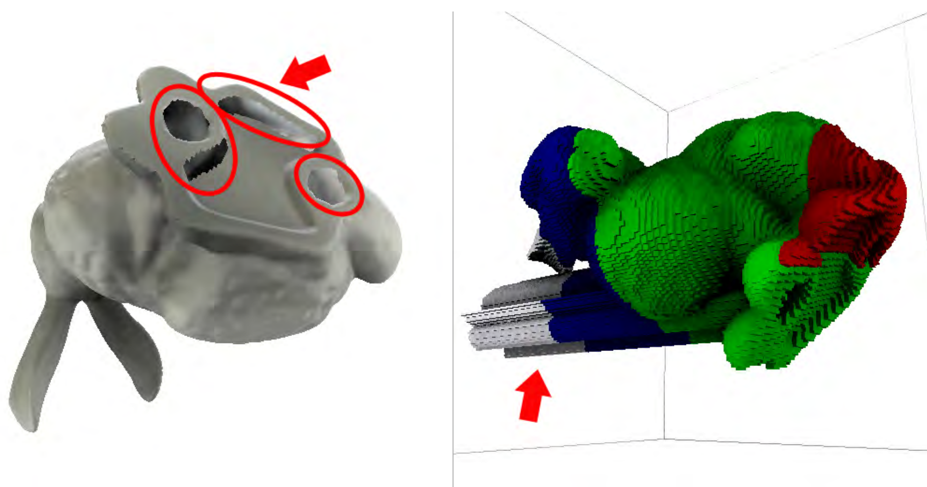


Abbildung 13: Beispiel einer fehlerhaften Voxelisierung: Das Original-Mesh des Stanford Bunnys ist nicht watertight. Die Tonfigur hatte unten Öffnungen. Die Voxelisierungskamera blickt durch die Löcher auf die eigentliche Rückseite, interpretiert diese aber als Vorderseite eines dort beginnenden Objektes und füllt den Raum dahinter entsprechend mit Voxeln. So entstehen die herausragenden „Säulen" im rechten Bild.

halten, müssten die Voxel auch in z-Richtung zusammengeführt werden.

Die bei der Voxelisierung erzeugte Slicemap muss eine quadratische Auflösung von $2^n \times 2^n$ haben. Der Mipmap-Level *0* hat die Auflösung $2^n \times 2^n$, der höchste Mipmap-Level n die Auflösung 1×1. Ausgehend von Mipmap-Level *0* wird in jedem Schritt die Auflösung in jeder Dimension halbiert, indem jeweils eine Gruppe von 2×2 Voxelspalten zu einer Voxelspalte zusammengeführt wird. Dies geschieht durch einfache Veroderung der Bits der vier Texel des nächstniedrigeren (im vorherigen Schritt erzeugten) Levels.

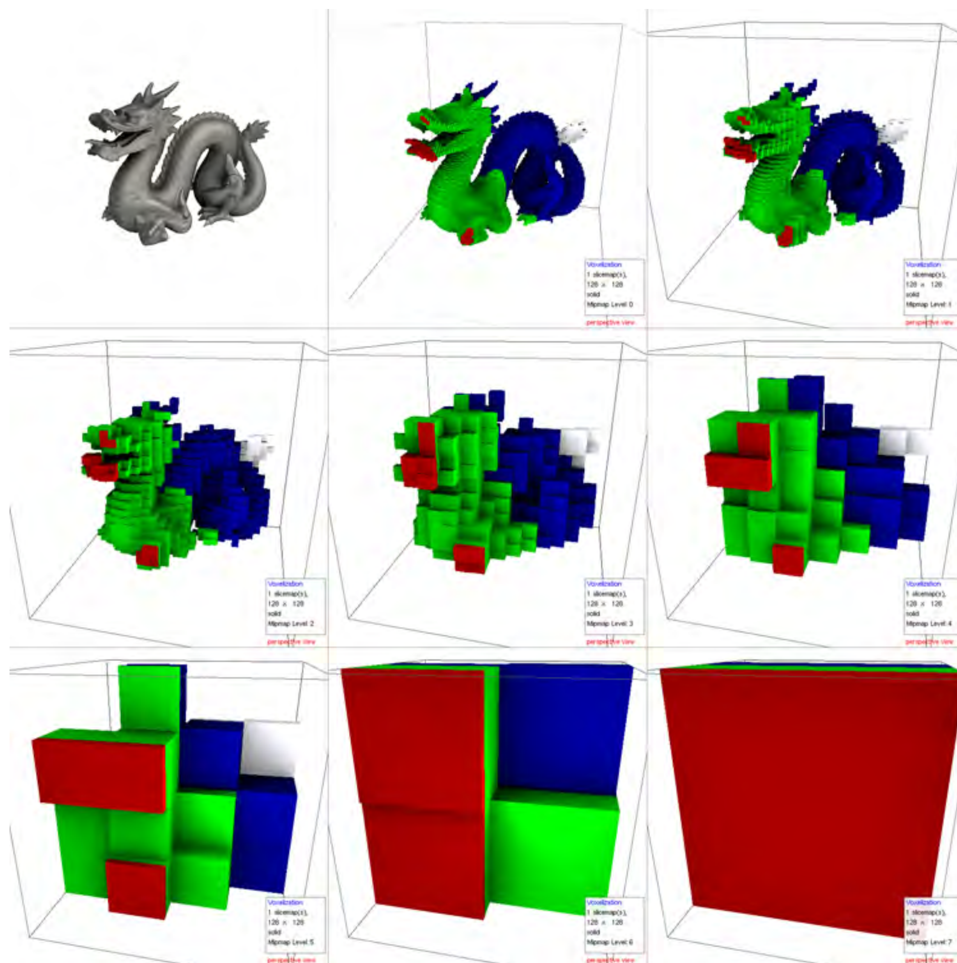


Abbildung 14: Voxel-Mipmaps des Dragons von Level 0 bis maximales Level 7. Initiale Slicemap-Auflösung: 128×128 (2^7)

Im Rahmen dieser Studienarbeit wurde versucht, die Mipmaps für einen verbesserten Sichtbarkeitstest im Voxelraum zu nutzen; genaueres dazu in Abschnitt 6.4 Ausblick, S. 55.

4 Sichtbarkeit

4.1 Problemstellung

Je nach zu lösendem Problem hat der Begriff Sichtbarkeit (Visibilität) in der Computergrafik eine andere Bedeutung.

Bei der Rasterisierung stellt sich die Frage: *Was ist auf dem Bildschirm sichtbar?* Am Ende der Rendering-Pipeline muss für jeden Punkt, der die Viewport-Transformation (Abbildung in Bildschirmkoordinaten) durchlaufen hat, entschieden werden, ob er dargestellt werden soll. Nur für die Kamera sichtbare Punkte werden dargestellt, alle verdeckten verworfen. Ein bekanntes Verfahren zur Lösung dieses Problems ist der Z-Buffer-Algorithmus. Um zu ermitteln, ob ein Punkt verdeckt oder sichtbar ist, wird der Tiefenpuffer (Z-Buffer) herangezogen, in dem die Tiefenwerte all jener zuvor gerenderten Objekte, die am weitesten „vorne" (nah an der Kamera) liegen, gespeichert sind. Ist der Tiefenwert des zu prüfenden Punkts größer als der aktuell an seiner Bildschirmkoordinate eingetragene Tiefenwert, ist er weiter von der Kamera entfernt, somit verdeckt, und wird nicht dargestellt. Diese Art der Sichtbarkeit bezieht sich also darauf, welches der in der Szene befindlichen Polygone an einem Bildpunkt (Pixel) dargestellt werden muss.

Die in dieser Studienarbeit relevante Art der Sichtbarkeitsanfrage bezieht sich auf *Sichtbarkeit innerhalb einer virtuellen Szene*. Dabei werden Punktpaare (zum Beispiel zwei Oberflächenpunkte) auf gegenseitige Sichtbarkeit oder eine Richtung von einem Punkt aus auf Unverdecktheit untersucht. Die Sichtbarkeitsinformation ist jeweils binär: Entweder sehen sich die Punkte oder nicht, entweder ist die Richtung unverdeckt oder nicht - eine teilweise Verdeckung ist nicht möglich. Zwei Punkte sehen sich dann nicht, wenn ein Objekt ihre Verbindungslinie schneidet. Ein solches verdeckendes Objekt wird *Occluder* genannt.

$$V(P,Q) = \begin{cases} 1 & \text{falls die Punkte P und Q sich sehen können} \\ 0 & \text{sonst} \end{cases} \tag{5}$$

Eine Richtung $\vec{\omega}$ ist von Punkt P aus gesehen dann unverdeckt, wenn sich kein Occluder in dieser Richtung befindet.

$$V(P,\vec{\omega}) = \begin{cases} 1 & \text{falls Richtung } \vec{\omega} \text{ unverdeckt} \\ 0 & \text{sonst} \end{cases} \tag{6}$$

Betrachtet man statt Punkten infinitesimal kleine Flächenelemente, die eine Normale haben, können von vornherein alle Punkte als nicht sichtbar (alle Richtungen als verdeckt) klassifiziert werden, die im hinteren Halbraum des betrachteten Elements liegen. Potentiell sichtbare Elemente müssen im vorderen Halbraum liegen bzw. als unverdeckte Richtungen werden nur solche in Erwägung gezogen, die in den vorderen Halbraum zeigen.

In Abschnitt 2.3.2 wurde der Visibilitätstest als zentrales Element des Ambient-Occlusion-Integrals vorgestellt. Auch hierbei wird nur der vordere Halbraum betrachtet bzw. sogar nur eine durch einen endlichen Radius begrenzte Hemisphäre.

Ein weiteres Anwendungsbeispiel sind Schattenfühler beim Ray Tracing. Zu beachten ist: Die erste Anfrage des Ray Tracers „Finde den ersten Schnittpunkt des Strahls“ ist komplexer als ein reiner Sichtbarkeitstest, bei dem danach gefragt wird, ob es überhaupt einen Schnitt des Strahls gibt.

4.2 Sichtbarkeitstest im Objektraum (Referenzlösung)

Im Objektraum liefert ein Ray Tracer die Referenzlösung des Sichtbarkeitstests. Schneidet der Strahl $Ray(t) = P + t \cdot \vec{\omega}$ mit $t \in]0..R]$ und beliebigem Suchradius R die Szene, so ist die Richtung $\vec{\omega}$ verdeckt. Dabei ist es irrelevant, an welcher Stelle des Strahls der Schnittpunkt gefunden wurde, es muss nicht der vorderste sein. Sobald ein Schnittpunkt gefunden wurde, kann die Suche beendet werden. Im schlimmsten Fall (kein Schnittpunkt und unendlicher Suchradius) müssen alle Objekte getestet werden, bis feststeht, dass wirklich kein Schnittpunkt zu finden ist. Durch Beschleunigungsstrukturen kann die Suche jedoch auf bestimmte Objekte begrenzt werden.

4.3 Sichtbarkeitstest im Bildraum (Screen-Space)

Der Sichtbarkeitstest im Bildraum hat als Grundlage den Tiefenpuffer oder Kamerakoordinaten-Puffer.

Wenn nur die Tiefenwerte gespeichert wurden anstelle vollständiger Kamerakoordinaten (x, y, z), können die Kamerakoordinaten rekonstruiert werden. Eine mögliche Rekonstruktionsmethode ist es, den Shadern Informationen über das Kamerafrustum beim Zeichnen eines bildschirmfüllenden Vierecks zu geben:

- Die Tiefenwerte müssen in der Form $|z|/zFar$) vorliegen (z in Kamera-Koordinaten).
- Beim Zeichnen des bildschirmfüllenden Quads wird jeder der vier Ecken ein Sichtstrahl von der Kamera aus (0,0,0) zu der entsprechenden Ecke der Far-Plane[1] mitgegeben (z. B. als Normale oder selbst definiertes Attribut an den Vertex Shader).
- Die Sichtstrahlen werden (nicht normalisiert) an den Fragment Shader weitergereicht, also interpoliert für jeden Pixel.
- Im Fragment Shader wird über *(Tiefe·Sichtstrahl)* die Kamerakoordinate ermittelt, die an dieser Pixelposition im Bild zu finden ist.

[1] Formeln hierfür siehe OpenGL Lighthouse 3D: View Frustum Culling Tutorial http://www.lighthouse3d.com/opengl/viewfrustum/index.php?gaplanes

Die beschriebene Rekonstruktionsmethode wurde in dieser Studienarbeit während der Implementierung getestet, um den G-Buffer um eine Textur zu verkleinern. Sie brachte jedoch keinen erkennbaren Performance-Vorteil, sodass der Einfachheit halber die Variante des Abspeicherns vollständiger Kamerakoordinaten gewählt wurde.

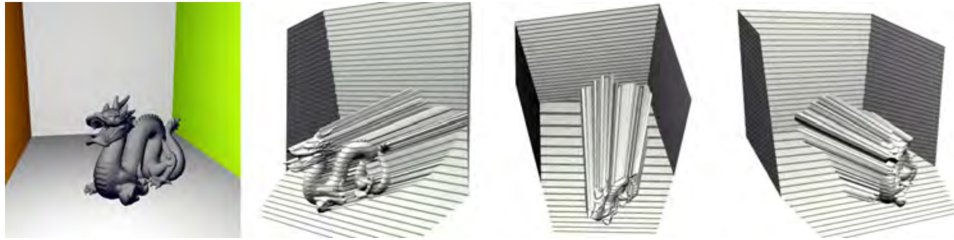


Abbildung 15: Visualisierung des Kamerakoordinaten-Buffers. Links die Szene, wie die Kamera sie sieht. Rechts drei Ansichten des als Triangle-Strip gerenderten Kamerakoordinaten-Buffers. Deutlich ist zu sehen, wie der Dragon in die Tiefe ragt.

Der **Ray Marching-Algorithmus im Bildraum** startet bei Kamerakoordinate P und läuft schrittweise entlang eines Strahls. Für jeden Schritt wird geprüft, ob Geometrie geschnitten wurde.

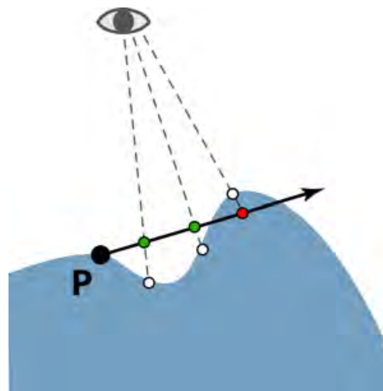


Abbildung 16: Ray Marching im Bildraum: Blau dargestellt ist das Höhenfeld. Die Sample-Positionen auf dem Strahl sind unverdeckt (grün) oder verdeckt (rot). Die weiß dargestellten Positionen auf dem Höhenfeld ergeben sich aus dem auf S. 25 beschriebenen Projektionsverfahren. Der Vergleich zwischen diesen Positionen und den Strahl-Samples liefert das Ergebnis, ob verdeckt oder unverdeckt. Abbildung nach [BS09, S. 430]

Für jede Sample-Position $P + t \cdot Ray$ soll also ermittelt werden, ob der Strahl an dieser Stelle unterhalb oder oberhalb des Höhenfeldes verläuft. Da die Suche einen begrenzten Radius hat, ist ein Endpunkt E auf dem Strahl bekannt. Das Abtasten des Strahls und die Generierung der Samples kann

auf zwei Wegen geschehen [BS09, S. 430]. Beide Methoden liefern identische Ergebnisse:

1. **3D-Samples in Kamerakoordinaten** $Sample_{cam} = P_{cam} + t \cdot Ray_{cam}$ mit einem 3D-Strahl der Länge *Radius*. Jedes 3D-Sample wird mit der zum Zeitpunkt des G-Buffer-Renderings gesetzten Projektionsmatrix und anschließender manueller Transformation in Texturkoordinaten überführt. Mit dieser Texturkoordinate wird auf die Tiefentextur zugegriffen und die „reale" Tiefe an der Stelle ermittelt. Der Vergleich von $Sample_{cam}.z$ und $RealPos_{cam}.z$ zeigt, ob sich der Sample-Punkt auf dem Strahl unter- oder oberhalb des Höhenfeldes befindet. In OpenGL blickt die Kamera nach der View-Transformation entlang der negativen z-Achse, also sind alle Tiefenwerte negativ. Ein größerer Tiefenwert bedeutet demnach „näher an der Kamera". Ein Schnitt liegt vor, wenn $RealPos_{cam}.z > Sample_{cam}.z$, denn die Geometrie liegt näher an der Kamera als der Punkt auf dem Strahl. Somit befindet sich der Strahl bereits innerhalb der Geometrie.
   Anders formuliert: Bewegt sich die Position auf dem Strahl durch die Projektion[2] von der Kamera weg, war an der Stelle keine Geometrie. Wenn sie sich zur Kamera hin bewegt, war an der Stelle Geometrie [RGS09].

   oder

2. **2D-Samples in Texturkoordinaten** $Sample_{uv} = P_{uv} + t \cdot Ray_{uv}$ mit einem 2D-Strahl durch vorherige Projektion des Start- und Endpunktes in Texturkoordinaten. In diesem Fall muss nicht mehr jedes Sample projiziert werden, sondern nur noch jeder Strahl. Folglich gibt es weniger Matrix-Multiplikationen im Shader. Das 2D-Sample wird direkt zum Texturzugriff benutzt, um $RealPos_{cam}$ zu ermitteln. Für den Vergleich zwischen dieser 3D-Position mit einer 3D-Strahl-Position ist es nötig, parallel zum 2D-Sample auch das 3D-Sample zu berechnen gemäß obiger Methode.

Ein Problem ist, dass nur der erste von der Kamera aus sichtbare Punkt pro Pixel bekannt ist – alles, was dahinter liegt, aber nicht. Es ragt sozusagen alles, was vorne nah an der Kamera ist, unendlich tief in die Szene hinein und wird dort als Geometrie erkannt. Das heißt, der Test kann ergeben, dass sich zwei Punkte nicht sehen, obwohl sich zwischen ihnen eigentlich kein Occluder befindet, im Screen-Space aber ein vorderes Objekt imaginär bis nach hinten ragt. Dadurch entstehende Artefakte bei Ambient Occlusion sind in Abschnitt 6 (Ergebnisse) geschildert.

[2]von Kamerakoordinaten in Texturkoordinaten und Auslesen der realen Kamerakoordinaten an dieser Texturkoordinate

Eine Lösung des Problems ist *Depth Peeling*. Dabei werden durch mehrfaches Rendering Schichten (Texturen) erzeugt, die den herkömmlichen Tiefenpuffer durch Tiefen ergänzen, die hinter dem vordersten Pixel liegen. Außer Tiefen können natürlich auch noch andere Informationen pro Schicht gespeichert werden, aber der Speicherbedarf ist enorm.

Wie kann mit zwei Schichten der Sichtbarkeitstest im Bildraum verbessert werden? Angenommen, die Geometrie ist so beschaffen, dass die Oberfläche eindeutig das Objektinnere vom Objektäußeren trennt. Dann gehört der Tiefenwert z_1 der ersten Schicht zu einer Vorderseite und der Tiefenwert z_2 der dahinter liegenden zweiten Schicht zu der zugehörigen Rückseite. Für einen Sample-Punkt muss daher ausgewertet werden, ob er sich hinter z_1, aber vor z_2 befindet, also im Inneren eines Objektes [RGS09, S. 3 f.]. Dieser Ansatz kann mit weiteren Schichtenpaaren fortgeführt werden.

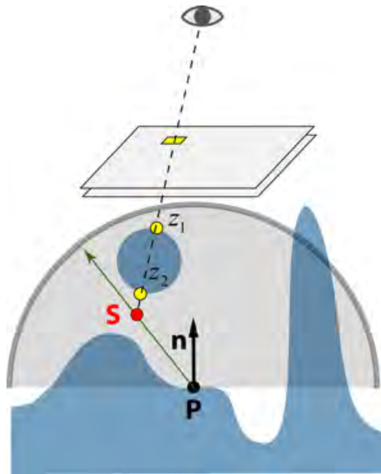


Abbildung 17: Sichtbarkeitstest und Depth Peeling mit 2 Schichten: Das Sample S (rot) ist von Punkt P aus gesehen eigentlich unverdeckt. Es würde aber bei nur einer Schicht als verdeckt angenommen, da es sich aus Sicht der Kamera hinter einem Occluder befindet. Mit 2 Schichten ergibt die Auswertung des z-Werte-Paars (gelb), dass das Sample unverdeckt ist, da es sich nicht zwischen den beiden Werten befindet. Grafik nach [RGS09, S. 4]

Ein anderer Ansatz ist das Rendern der Szene aus zusätzlichen Kamerapositionen und -blickwinkeln. [RGS09, S. 4] beschreiben die Details einer möglichen Umsetzung. Ein nennenswerter Vorteil gegenüber Depth Peeling besteht darin, dass so auch Informationen über Polygone gewonnen wird, die ganz oder nahezu parallel zur Standard-Kamera verlaufen. Außerdem erfassen die zusätzlichen Kameras Objekte, die außerhalb des mit der Standard-Kamera gerenderten Bildes liegen. Somit lassen sich Probleme am Bildrand lösen, die daher stammen, dass es über die außerhalb der Tiefentextur liegende Geometrie keine Information gibt.

4.4 Sichtbarkeitstest im Voxelraum (Voxel-Space)

Die geschilderten Probleme im Bildraum entspringen immer einer mangelnden Information über im Bild nicht erfasste Geometrie.

Im Voxelraum werden sämtliche Objekte von einer Voxelisierungskamera erfasst. Anders als bei der Nutzung des Tiefenpuffers (1 Layer, keine zusätzlichen Kameras) liegt hier auch Information über die Szenengeometrie vor, die von der Kamera aus nicht direkt sichtbar ist. Dies ist Geometrie, die von vorderen Objekten verdeckt wird oder gar außerhalb des rasterisierten Bereichs der darstellenden Kamera liegt.

Die voxelisierte Szene (eine Slicemap, Körpervoxelisierung) wird unterstützend zum G-Buffer herangezogen. Zur Voxelisierung und zur finalen Darstellung der Szene kann jeweils eine eigene Kamera benutzt werden:

Voxelisierungskamera zur Voxelisierung mit orthographischer Projektion und eigener View-Transformation. Im Folgenden wird nur eine einfache statische View-Transformation betrachtet, sodass das quaderförmige Frustum die gesamte Szene umschließt. Komplizierter wird es, wenn die Voxelisierungskamera dynamisch so positioniert wird, dass ihr Frustum immer mindestens so viel von der Szene umschließt wie jenes der Darstellungskamera.

Darstellungskamera zum Rendering des G-Buffers mit perspektivischer Projektion und eigener (von der View-Transformation der Voxelisierungskamera abweichender) View-Transformation zum dynamischen Bewegen.

Wie lassen sich im Screen-Space erfasste Kamerakoordinaten der Darstellungskamera und Voxel-Space in Einklang bringen?

Die Projektion der Kamerakoordination der Darstellungskamera in Slicemap-Texturkoordinaten ist komplexer als im bildbasierten Fall. Im Bildraum genügt die Projektionsmatrix der einzigen darstellenden Kamera. Um die Kamerakoordinaten in Slicemap-Koordinaten zu transformieren, müssen sie...

- zunächst mit der *inversen View-Matrix der Darstellungskamera* in Weltkoordinaten gebracht werden,
- anschließend mit der *View-Matrix der Voxelisierungskamera* in die Voxelisierungskamera-Koordinaten
- und dann mit der *Projektionsmatrix der Voxelisierungskamera* (und Überführung in Fensterkoordinaten) in Slicemap-Texturkoordinaten

...transformiert werden (siehe Abbildung 18).

Die drei Matrizen sollten bereits im Vertex Shader multipliziert und das Ergebnis als `varying` Variable an den Fragment Shader weitergereicht werden.

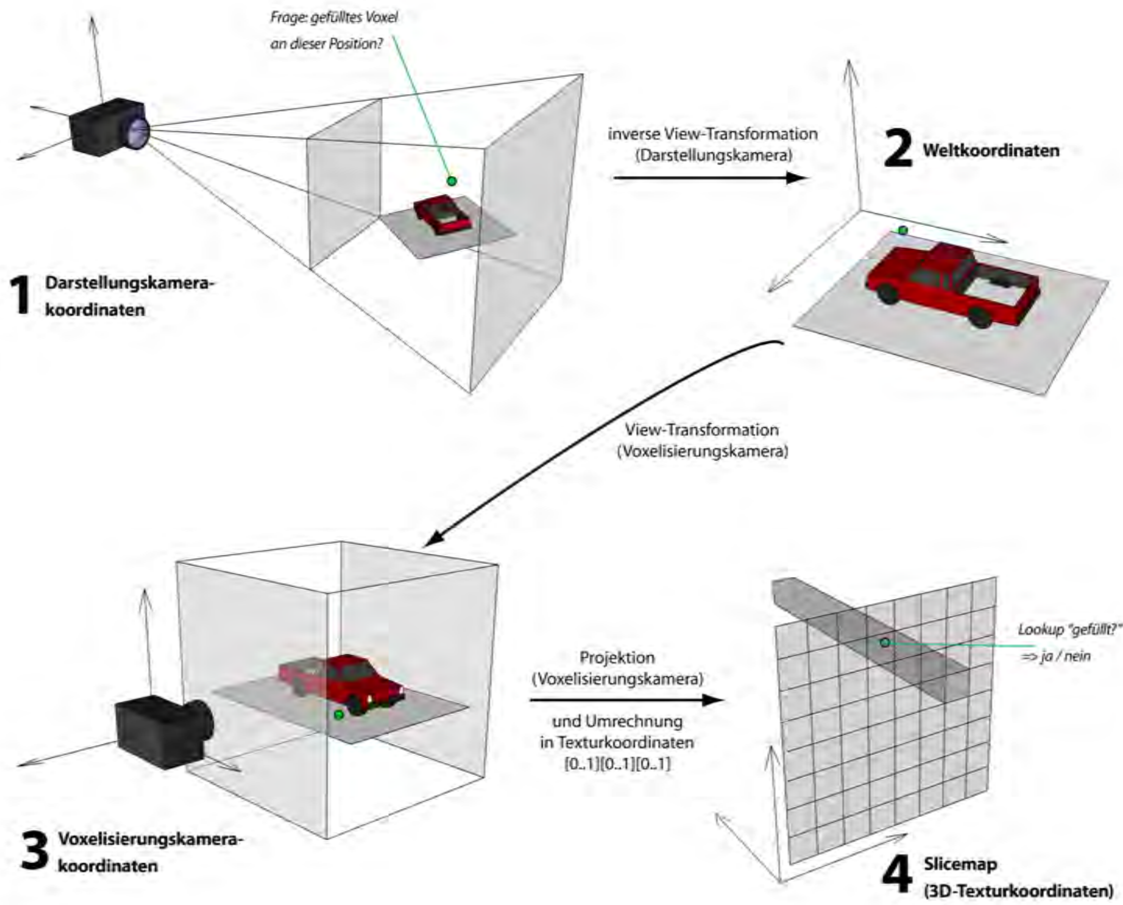


Abbildung 18: Die Transformationsschritte

Es entsteht eine Texturkoordinate mit 3 Komponenten: Die ersten beiden (s, t) werden dafür gebraucht, das entsprechende Texel (die Voxelspalte) aus der Slicemap abzurufen. Nur ein spezielles Voxel innerhalb der 128 Voxel dieses Texels soll daraufhin überprüft werden, ob es gefüllt ist, also ob sich hier bei der Voxelisierung Geometrie befand. Mit der dritten Komponente der Texturkoordinate wird ein Lookup in einer 1D-Textur gemacht, die der in Abbildung 9 auf S. 17 dargestellten Bitmask entspricht. Eine bitweise Verundung des entsprechenden Texels aus dieser Lookup-Textur mit dem Texel aus der Slicemap liefert einen 128 Bit-Wert (aufgeteilt in einen 4-Komponenten-Vektor mit je 32 Bits pro Komponente), in dem entweder alle Bits 0 sind (leeres Voxel an der Stelle), oder aber in dem ein Bit gesetzt ist (volles Voxel an der Stelle).

4.5 Bildraum vs. Voxelraum

Bildraum (einfachster Fall ohne Depth-Peeling und zusätzliche Kameras):

- keine komplizierten Transformationen: nur eine Kamera, nur ein einziger Raum
- kann auf alles angewendet werden, was in den Z-Buffer rasterisiert werden kann; keine Einschränkungen der Meshes
- falsche Ergebnisse möglich wegen „nach hinten ragender vorderer Objekte“
- blickwinkelabhängig, hat keine Information über verdeckte Geometrie und nicht-rasterisierte (außerhalb des Bildes befindliche) Geometrie

Voxelraum:

- nicht blickwinkelabhängig
- kennt (näherungsweise) die gesamte Szenengeometrie, auch verdeckte oder außerhalb des Sichtfelds der Darstellungskamera liegende Objekte
- Slicemap kann als diskretes Depth-Peeling angesehen werden, bei dem die Anzahl der Schichten in z-Richtung vorgegeben ist
- fehleranfälliger aufgrund der verschiedenen Räume (Voxelisierungskamera, Darstellungskamera) und der Transformationen
- zusätzlicher Pass für die Voxelisierung und zusätzliche Textur (Slicemap)
- vorgestellte Voxelisierungsmethode erfordert Meshes, die watertight sind, und Shader Model 4.0 wegen der Integer-Texturen

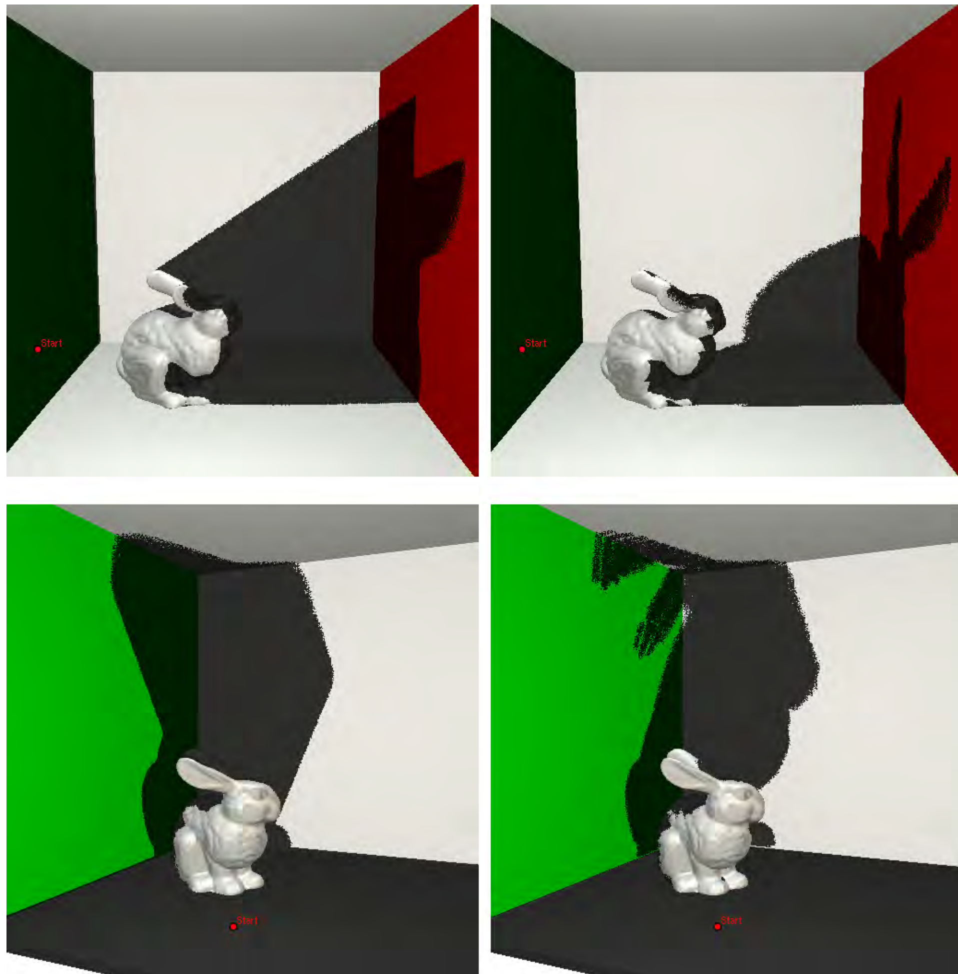


Abbildung 19: Vergleich der Sichtbarkeitstests: *Links:* im Bildraum, *rechts:* im Voxelraum. Die Bilder entstanden, indem ausgehend von einem fest definierten Startpixel (rot markiert) ermittelt wurde, welche anderen Pixel er im Bild sieht. Methode: Ray Marching, je 50 Schritte (mit Jittering) entlang eines Strahls.

Beide Verfahren:

- keine weiteren auf der CPU zu haltenden und zu pflegenden Datenstrukturen
- keine Vorverarbeitung
- geeignet für dynamische Szenen

5 Implementierung

5.1 Überblick

In dieser Studienarbeit wurden folgende Verfahren umgesetzt:

- Ambient Occlusion mit Ray Marching im Bildraum und Voxelraum
- Indirektes Licht (erste Indirektion) im Bildraum einsammeln (Sichtbarkeitsbestimmung dabei entsprechend des jeweiligen Ambient-Occlusion-Verfahrens)
- Sichtbarkeitstests im Bildraum und Voxelraum („Was sieht ein Pixel?“)

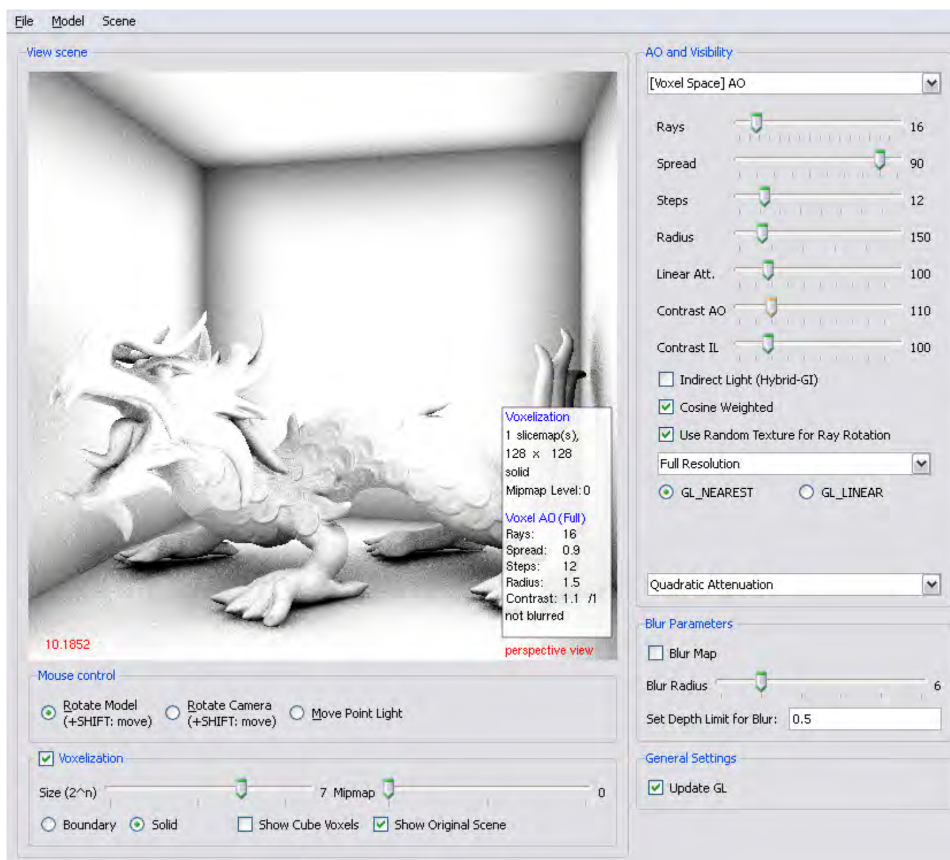


Abbildung 20: Screenshot der Anwendung, in der die genannten Verfahren betrachtet werden können. Die Verfahren hängen von vielen manuell zu setzenden Parametern ab, sodass ebenso viele Steuerelemente für das Justieren der Werte benötigt werden.

Die in Abbildung 20 gezeigte Anwendung entstand in der Entwicklungsumgebung Visual Studio 2008 mit C++, OpenGL (Open Graphics Library)

und GLSL (OpenGL Shading Language). Das Laden von 3D-Modellen ermöglicht der Wavefront-OBJ-Loader GLM von Nate Robins[3]. Die Benutzeroberfläche wurde mit der Bibliothek Qt[4] (Version 4.5.1) realisiert.

Die wichtigsten Schritte, die bei jedem Zeichenaufruf abgearbeitet werden, sind in Abbildung 21 dargestellt. Die Wahlmöglichkeiten des Benutzers, sich nur bestimmte Texturen statt des finalen Bildes anzeigen zu lassen, sind der Übersicht halber nicht abgebildet. Außerdem kann die voxelisierte Szene in Form vieler Quader dargestellt werden, was insbesondere zum Debuggen nützlich war.

Die Voxelisierung wird nur durchgeführt, wenn anschließende Schritte die voxelisierte Szene (Slicemap) als Eingabe benötigen. Abschnitt 3 (S. 14 ff.) beschreibt die Erzeugung der Slicemap samt Implementierungsdetails, sodass dieser Schritt hier nicht weiter erläutert wird.

Bei der Implementierung der Verfahren wurde besonders darauf geachtet, die Algorithmen und deren Eingabewerte für Bild- und Voxelraum so ähnlich wie nur möglich zu halten, um faire Vergleiche ziehen zu können.

Alle wesentlichen Schritte (Voxelisierung, G-Buffer-Erstellung, Ambient Occlusion und indirektes Licht, Blur und Kombination der erzeugten Texturen) sind als Shader implementiert.

5.2 G-Buffer erzeugen

In der Szene (Cornell-Box mit 3D-Modell(en)) gibt es eine Punktlichtquelle und zwei gerichtete Lichtquellen, die die Grundhelligkeit der direkten Beleuchtung erhöhen. Optional kann Shadow Mapping eingeschaltet werden, was einen zusätzlichen Render-Pass aus Sicht der Punktlichtquelle erfordert.

Für das Rendern der Szene wird ein Framebuffer-Objekt (FBO) mit mehreren Render Targets genutzt. Für die an das FBO gehängten Texturen (*color attachments*) gelten bestimmte Regeln in Bezug darauf, wie oder ob interne Texturformate miteinander kombiniert werden dürfen. In der hier vorgestellten Implementierung wurden vier Render Targets mit jeweils 3 Kanälen (RGB) und dem internen Format `GL_RGB32F_ARB` verwendet. Diese vier Texturen bilden den G-Buffer. Eine Änderung des G-Buffer-Layouts zieht viele Änderungen in anderen Programmteilen nach sich. Dennoch musste eine solche in der letzten Implementierungsphase durchgeführt werden, denn der zunächst angedachte einzelne Kanal für die Speicherung der Beleuchtungsstärke reichte für farbige Lichtquellen nicht aus.

Die finale Zusammensetzung des G-Buffers sieht wie folgt aus: je eine Textur für die BRDF ($f_{r,d}$= Reflexionsgrad/π), für die direkte Beleuchtung ($L_{dir} = f_{r,d} \cdot E_{dir}$), für Normalen in Kamerakoordinaten und für Positionen in Kamerakoordinaten.

[3] enthalten im *Tutors source code package*: http://www.xmission.com/~nate/tutors.html

[4] http://qt.nokia.com/

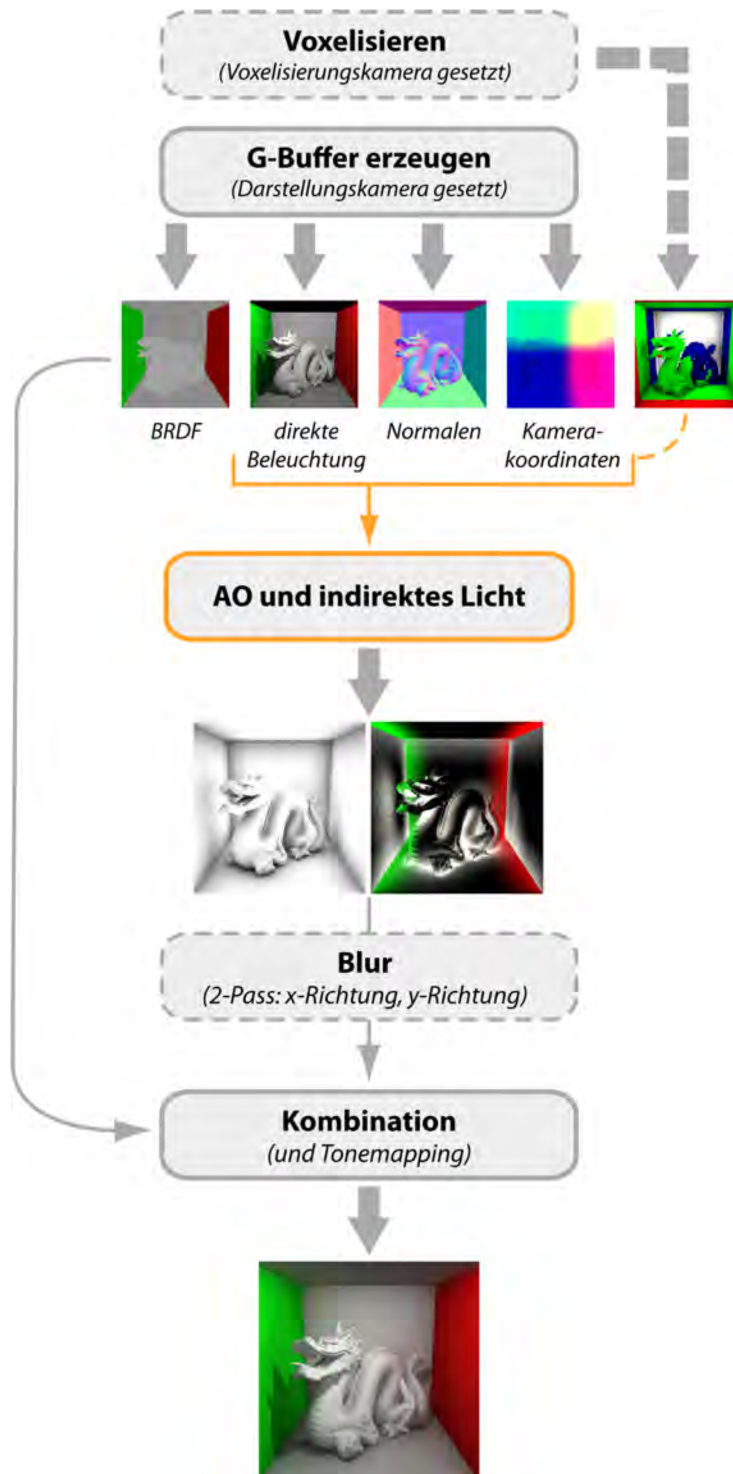


Abbildung 21: Rendering-Ablauf: Skizze der jeweiligen Schritte pro Frame.

Ausgehend von einer Szene mit rein diffusen Materialen wird für die direkte/lokale Beleuchtung das Lambert-Beleuchtungsmodell eingesetzt. Die Lichtquellen sind mit ihrer Lichtstärke I (Einheit: Candela) definiert. Eine 100-Watt-Glühlampe erzeugt beispielsweise ca. 1000 Lumen, was für eine Punktlichtquelle $\frac{1000lm}{4\pi sr} \approx 80cd$ sind.

Die photometrische Formel für die Beleuchtungsstärke ist $E = \frac{I \cdot \cos\theta}{d^2}$, wobei d der Abstand zu der Lichtquelle und θ der Winkel zwischen Oberflächennormale und dem Vektor in Richtung Lichtquelle ist. Gerichtete Lichtquellen haben keine solche Abschwächung. Als Alternative zu den gerichteten Lichtquellen wurde der Lookup in einer stark geblurrten diffusen Environment Map implementiert.

5.3 Ambient Occlusion und indirektes Licht

5.3.1 Sampling

Um zu ermitteln, wie stark die Hemisphäre eines Punktes verdeckt ist, wird Monte-Carlo-Integration eingesetzt, deren Grundlage die Erzeugung von Samples mit einer bestimmten Dichte ist.

Die theoretischen Grundlagen wurden in Abschnitt 2.3.4 (S. 8 f.) erläutert. Es wurde bereits darauf hingewiesen, dass die Wahl der Dichtefunktion, mit der die Samples generiert werden, beeinflusst, wie schnell und gut sich der geschätzte Wert der tatsächlichen Lösung annähert. „Schnell“ heißt, dass möglichst wenig Samples benutzt werden müssen.

Die Basis für die Erzeugung von Samples bilden im Allgemeinen (0,1)-verteilte Zufallszahlen. Diese sind im Intervall $[0, 1]$ so verteilt, dass jeder Wert mit gleicher Wahrscheinlichkeit angenommen werden kann. Hier spielt die Güte der erzeugten Zufallszahlen eine wichtige Rolle. Statt echter oder Pseudo-Zufallszahlen, wie der C++-Zufallszahlengenerator sie liefert, können auch deterministisch erzeugte *Low Discrepancy Sequenzen* eingesetzt werden. Jene Diskrepanz ist ein mathematisch definiertes Maß für die Qualität einer Sequenz – je niedriger die Diskrepanz, desto besser sind die Zahlen verteilt.

Für die implementierten Ambient-Occlusion-Verfahren wurde die Low Discrepancy *Hammersley-Sequenz* verwendet, da die Anzahl der Samples vorab bekannt ist und sie eine bessere Diskrepanz als die *Halton-Sequenz* aufweist. Später wurde zu Vergleichzwecken auch die Halton-Sequenz eingebaut. Es zeigte sich, dass die Hammersley-Sequenz tatsächlich die bessere Wahl ist (siehe Abbildung 22).

Das i-te n-dimensionale Element von insgesamt N Elementen der Hammersley-Sequenz ist definiert als $X_i = (\frac{i}{N}, \Phi_2(i), \Phi_3(i), \Phi_5(i), ..., \Phi_b(i))$. Die Funktion $\Phi_b(i)$ heißt *radikal inverse Funktion* mit $i \in \mathbb{N}$ und Basis b. Als Basen sind Primzahlen zugelassen. Die Zahlen i liegen normalerweise im Dezimalsystem vor. Die radikal inverse Funktion konvertiert i zunächst ins Zahlen-

Abbildung 22: Konstruktion der Samples mit ... *links:* Hammersley-Sequenz, *Mitte:* Halton-Sequenz, *rechts:* C++-Pseudo-Zufallszahlen. (Verfahren: Voxel-Space AO, Ray Marching)

system mit Basis b, spiegelt diese Zahl am Komma und konvertiert die gespiegelte Zahl wieder ins Dezimalsystem. Das Ergebnis ist eine Zahl zwischen 0 und 1.

Ausgehend von einer zweidimensionalen Hammersley-Sequenz wurden uniformes und cosinusgewichtetes Sampling der Hemisphäre umgesetzt (siehe Abbildung 23).

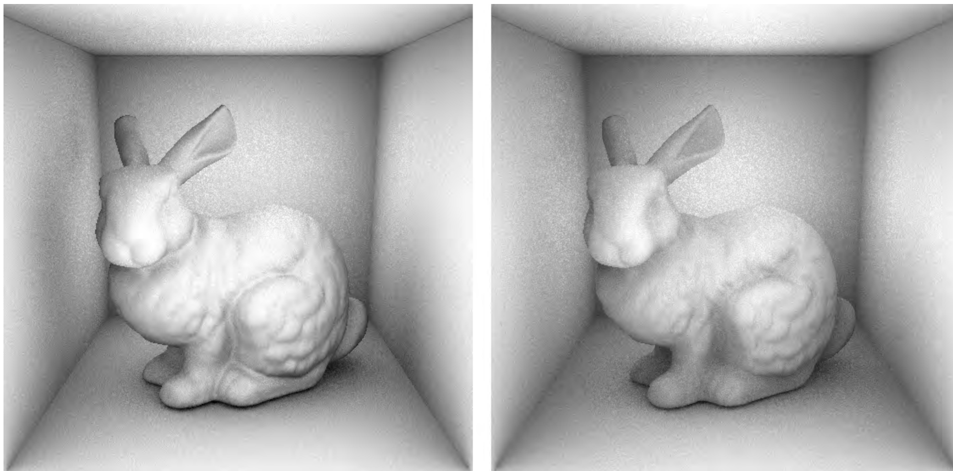


Abbildung 23: *Links:* Cosinusgewichtetes Sampling. *Rechts:* Uniformes Sampling (ohne Cosinus-Term)

Dem Shader wird die Sequenz statt der fertigen Samples als `uniform`-Variable übergeben, da die Werte der Sequenz pro Pixel modifiziert werden. Das Vorgehen hierfür heißt *Cranley-Patterson-Rotation* [PH04, S. 348]:

$$X_i' = (X_i + \xi_i) \bmod 1$$

ξ_i ist eine Zufallszahl zwischen 0 und 1, die pro Pixel aus einer kleinen gekachelten Random-Textur gelesen wird. Würde man die Samples vorab be-

rechnen und als uniform-Variable an den Shader reichen, wäre das Sampling-Muster für jeden Punkt gleich. Abbildung 24 zeigt den dabei auftretenden Effekt.

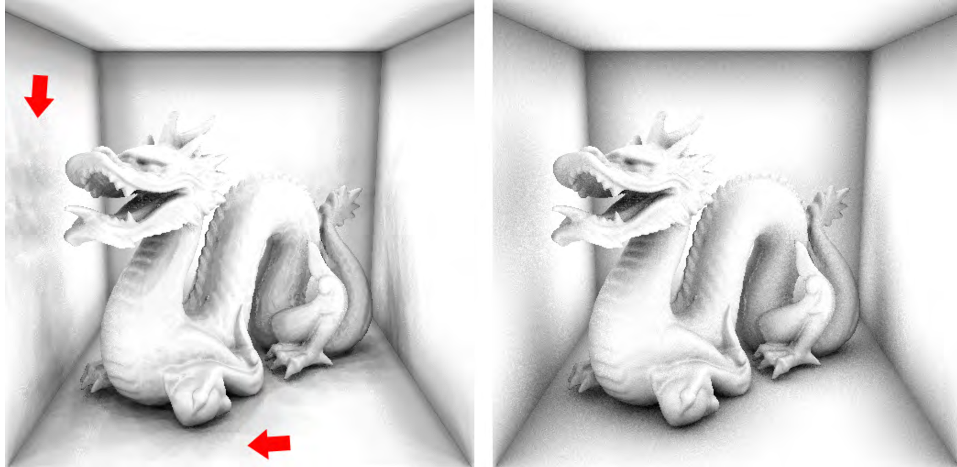


Abbildung 24: Ohne Rotation *(links)* treten die mit den Pfeilen markierten Artefakte auf, mit Rotation *(rechts)* wandeln sich diese zu Rauschen. (Verfahren: Voxel-Space AO, Ray Marching, Sampling mit Hammersley-Sequenz)

Nach [PH04, S. 650 f., 656 f.] sind die Kugelkoordinaten bzw. kartesischen Koordinaten der Sampling-Richtungen wie folgt zu berechnen:

Uniformes Sampling mit konstanter Dichte

$$\begin{aligned} \text{Kugelkoordinaten:} \quad \theta &= \cos^{-1}\xi_1 \\ \phi &= 2\pi\xi_2 \\ \text{Kartesische Koordinaten:} \quad x &= \cos(2\pi\xi_2)\sqrt{1-\xi_1^2} \\ y &= \sin(2\pi\xi_2)\sqrt{1-\xi_1^2} \\ z &= \xi_1 \end{aligned}$$

Cosinusgewichtetes Sampling mit Dichte proportional zu $\cos(\theta)$

Das cosinusgewichtete Sampling der Hemisphäre erzeugt erst uniform verteilte Punkte *(x,y)* in einem Einheitskreis und hebt diese dann senkrecht in die dritte Dimension auf die Oberfläche der Hemisphäre.

$$\begin{aligned} \text{Kartesische Koordinaten:} \quad x &= \cos(2\pi\xi_2)\sqrt{\xi_1} \\ y &= \sin(2\pi\xi_2)\sqrt{\xi_1} \\ z &= \sqrt{1-x^2-y^2} \end{aligned}$$

Eine Herleitung findet sich in der genannten Quelle. Die so erzeugten kartesischen Koordinaten (x, y, z) entsprechen Einheitsvektoren. Sie sind um eine Hemisphäre mit Normale (0,0,1) orientiert und müssen daher noch der tatsächlichen Normalen entsprechend ausgerichtet werden.

Listing 3: Erzeugung der Richtungssamples: Auszug aus Ambient Occlusion GLSL Fragment Shader

```
#version 120
#define PI 3.1415926535897932384626433832795

uniform sampler2D randomTex;
uniform vec2 samplingSequence[128]; // 2D Low Discr. Sequence
uniform int numRays;
uniform bool useCosWeight;

void main()
{
   [...]
   // TBN Basis
   // get normal N, compute tangent Tan and bitangent BiTan

   // get random numbers [0,1]
   vec3 rand = texture2D(randomTex,mod(gl_FragCoord.st,64.0)
       /64.0).rgb;

   // distribute rays over hemisphere
   for(int i = 0; i<numRays; i++)
   {
      // Cranley-Patterson Rotation
      float u1 = fract(samplingSequence[i].x + rand.x);
      float u2 = fract(samplingSequence[i].y + rand.y);
      float r;
      if(useCosWeight)
         // cosine weighted sampling
         r = sqrt(u1);
      else
         // uniform sampling
         r = sqrt(1.0f - u1 * u1);
      float phi = 2 * PI * u2;
      // cartesian coordinates
      float x = cos(phi) * r;
      float y = sin(phi) * r;
      float z = sqrt(max(0.0,1.0-x*x-y*y));

      // transform sampling direction to camera space
      vec3 ray = x*Tan+y*BiTan+z*N;

      // march ray [...]
   }
}
```

Die Methode zur Ausrichtung der Richtungen mit Hilfe einer Tangent-Bitangent-Normal Basis stammt von NVIDIA[5], siehe Listing 4. Ergänzt wurde die Berechnung durch die Zeilen 4 und 5, um den Bug zu beheben, dass bei Normalen mit Richtung (0,0,1) oder (0,0,-1) die Bitangente durch das Kreuzprodukt zu (0,0,0) degeneriert und infolgedessen ebenso die Tangente (falsches Ergebnis siehe rechts: Die Rückwand der Cornell-Box wird nicht korrekt dargestellt). Die initiale „beliebige" Tangente darf nicht mit der Normalen übereinstimmen.

Listing 4: Berechnung der TBN Basis

```
1 vec3 N = texture2D(normalTex,gl_TexCoord[0].st).xyz;
2 //arbitrary vector Tan not coinciding with the normal
3 vec3 Tan = vec3(0.0,0.0,1.0);
4 if(N.x==0.0 && N.y==0.0)
5   Tan = vec3(0.0,1.0,0.0);
6 vec3 BiTan = normalize(cross(N,Tan)); // bitangent
7 Tan = cross(BiTan,N); // tangent
```

Jede Richtung wird mit einer benutzerdefinierten Schrittzahl und innerhalb eines ebenfalls benutzerdefinierten Radius stückweise auf Schnittpunkte mit Occludern abgetastet.

5.3.2 Gewichtung der Samples

Beim Ray Marching wird der erste Punkt, der sich innerhalb von Geometrie befindet, als Schnittpunkt angenommen. Deshalb können die einzelnen Sichtbarkeitsergebnisse jeweils entsprechend des Abstandes zum ersten Schnittpunkt gewichtet werden (vgl. Abschnitt 2.4 Obscurances, S. 12). Abbildung 25 zeigt die Ergebnisse verschiedener Distanzabbildungsfunktionen. Auf der y-Achse ist das Gewicht eines Samples abgetragen.

Dass der grundsätzliche Verlauf der Funktionen aus Abbildung 25 zu jenem in Abbildung 6 (S. 13) gespiegelt ist, liegt an der konkreten Berechnung des Verdeckungswertes. Im Bild stehen dunkle Farb- oder Grauwerte für „verdeckt". Bei den in dieser Studienarbeit implementierten Verfahren werden nur die verdeckten Samples gewichtet und aufaddiert. Es ergeben sich hohe Werte (größere Helligkeit) für verdeckte Bereiche, sodass der akkumulierte Wert am Ende invertiert werden muss.

[5]NVIDIA Direct3D SDK 10.5 Sample *Screen Space Ambient Occlusion* (gemeint ist damit Horizon-Based Ambient Occlusion) http://developer.download.nvidia.com/SDK/10.5/direct3d/samples.html

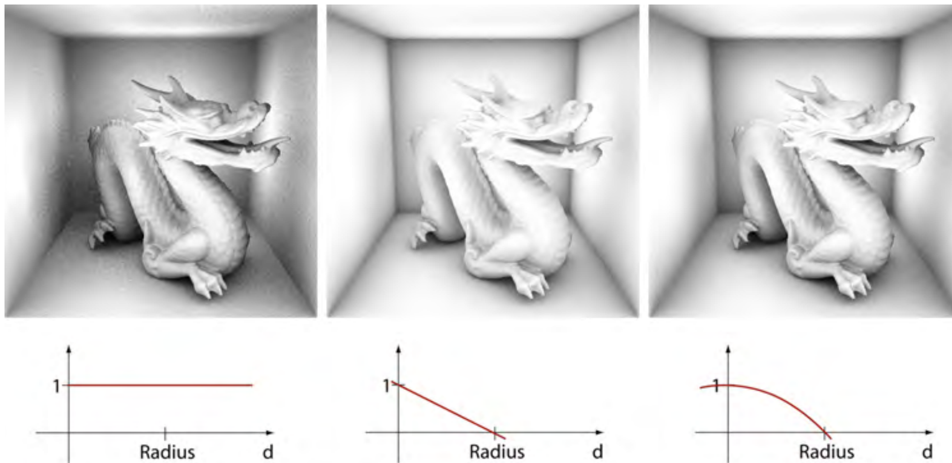


Abbildung 25: Von links nach rechts: keine Abschwächung, lineare Abschwächung, quadratische Abschwächung. (Verfahren: Voxel-Space AO, Ray Marching)

5.3.3 Screen-Space AO und Screen-Space GI

Die Implementierung von Screen-Space AO und GI (kurz SSAO und SSGI) ist getreu ihren Namen auf den Bildraum beschränkt. Es findet kein Depth-Peeling statt und es gibt nur eine Kamera. Die Methode zur Sichtbarkeitsermittlung ist das auf S. 24 geschilderte Ray Marching. Bei Screen-Space AO werden die Sichtbarkeitsergebnisse im Falle eines gefundenen Schnittpunktes mit einer der genannten Gewichtungsfunktionen gewichtet, aufaddiert und diese Summe durch die Anzahl der verschossenen Strahlen geteilt. Das Ergebnis muss schließlich invertiert werden, um zu erreichen, dass verdeckte Bereiche im Bild dunkel erscheinen.

Der Fragment Shader für Screen-Space AO wird für jeden Pixel beim Zeichnen eines bildschirmfüllenden Rechtecks aufgerufen. Er benötigt als Eingaben: die Textur mit den Kamerakoordinaten, die Textur mit den Normalen, eine kleine Zufallstextur, die Projektionsmatrix der Kamera, die zum Zeitpunkt der G-Buffer-Erzeugung gesetzt war, eine 2D-Low-Discrepancy-Sequenz und zahlreiche benutzerdefinierte Parameter (z. B. Strahlanzahl, Schrittanzahl, Kontrast).

Das Ergebnis schreibt er in eine an ein FBO geknüpfte Graustufen-Textur vom Typ `GL_LUMINANCE32F_ARB`. Die Textur hat die Auflösung des Viewports, der während des Zeichnens des bildschirmfüllenden Rechtecks gesetzt ist. Der Aufwand von Screen-Space-Verfahren hängt vor allem von der Auflösung ab; die Geschwindigkeit beim Rendern kleinerer Auflösungen erhöht sich entsprechend. Daher kann die Ambient-Occlusion-Textur in voller, halber oder viertel Auflösung (volle Auflösung = Auflösung der G-Buffer-

Texturen) erstellt werden. Mit „halber Auflösung“ ist gemeint, dass sowohl Breite als auch Höhe halbiert werden. Eigentlich handelt es sich um eine Viertelung der Pixelanzahl und somit im besten Fall um eine Vervierfachung der Geschwindigkeit. Für das Hochskalieren der Textur auf volle Auflösung wird ein geometrie-sensitiver Blur genutzt, mehr dazu in Abschnitt 5.4 auf S. 43.

Listing 5 zeigt, wie ein Strahl schrittweise durch Tiefenvergleiche auf Schnittpunkte getestet wird. Es ist die 3D-Samples-Methode, die in Abschnitt 4.3 auf S. 25 beschrieben wird. Mit `j+rand.z` in Zeile 13 wird die Schrittweite pro Pixel „gejittert“.

Listing 5: SSAO: Schritte entlang eines 3D-Strahls in Kamerakoordinaten

```
float ao = 0.0; // init ao for accumulation
for(int i=1; i<=numRays; i++)
{
   // contruct i-th ray

   bool hit = false;
   float hitDist = 0.0;
   //steps along ray
   for(float j=1.0; j<=numSteps; j++)
   {
      if(!hit)
      {
         vec3 samplePos = P + (j+rand.z)/numSteps*R*ray;
         vec3 realPos = getEyePosition(samplePos);

         // compare depths: hit?
         if(realPos.z-eps > samplePos.z)
         {
            float d = distance(P,realPos);
            if (d<R) // sample inside radius?
            {
               hit = true;
               hitDist = d;
            }
         }
      }
   }
   if(hit)
   {
      float attenuation;
      // compute attenuation
      ao += attenuation;
   }
}
finalAO = 1.0-contrast*ao/float(numRays);
```

Die zentrale Funktion zur Projektion eines 3D-Samples in Texturkoordinaten für den Lookup in der Kamerakoordinaten-Textur ist wie folgt definiert:

Listing 6: SSAO: Projektion eines 3D-Samples

```
1  vec3 getEyePosition(in vec3 p)
2  {
3     vec4 clipCoord = projMatrix * vec4(p,1.0);
4     vec3 nDeviceCoord = clipCoord.xyz / clipCoord.w;
5     vec2 lookupCoord = (nDeviceCoord.xy+1.0)/2.0;
6     return texture2D(eyePosTex,lookupCoord).xyz;
7  }
```

Zu beachten ist hierbei, die Projektionsmatrix zu verwenden, die zum Zeitpunkt der G-Buffer-Erzeugung gesetzt war, anstelle der Built-In Matrix `gl_ProjectionMatrix`. Denn letztere ist für das Zeichnen des bildschirmfüllenden Rechtecks gesetzt.

Der Screen-Space-GI-Shader schreibt in eine RGBA-Textur (32-bit floating point) – die ersten drei Kanäle enthalten das indirekte Licht, der Alpha-Kanal den Ambient-Occlusion-Term. Zusätzlich zu den eben genannten Eingaben benötigt er noch die Textur mit den Leuchtdichten der direkten Beleuchtung, um das indirekte Licht einzusammeln. Gesucht ist die Beleuchtungsstärke

$$E = \int_{\Omega} L_{\omega_e} \cos\theta_e \, d\omega \;\approx\; \pi \frac{1}{N} \sum_{i=1}^{N} L(X_i).$$

Die rechte Seite der Gleichung ist der Monte-Carlo-Schätzer für die Beleuchtungsstärke der indirekten Beleuchtung bei cosinusgewichtetem Sampling. L_{ω_e} bzw. $L(X_i)$ ist die Leuchtdichte der direkten Beleuchtung an dem ersten mit Ray Marching gefundenen Schnittpunkt entlang einer Richtung ω_e bzw. X_i. Sie wird der Textur an der durch Projektion des Schnittpunkts ermittelten Koordinate entnommen und parallel zu dem Ambient-Occlusion-Term akkumuliert. Somit hat der SSGI Shader einen Texturzugriff mehr pro Strahl, für den ein Schnittpunkt gefunden wurde, und nur wenige Operationen mehr.

5.3.4 Voxel-Space AO und Hybrid-GI

Die Autoren von *„Hybrid Ambient Occlusion“* [RBA09] nennen ihr voxelbasiertes Ambient-Occlusion-Verfahren hybrid, da im Voxelraum die Sichtbarkeit bestimmt wird, die AO-Textur aber mit geringerer Auflösung erstellt und mit einem *Joint Bilateral Upsampling* Filter [KCLU07] im Bildraum hochskaliert wird.

Im Folgenden wird trotzdem die Bezeichnung *Voxel-Space AO* verwendet, da die Sichtbarkeitsbestimmung der zentrale Punkt von Ambient Occlusion ist.

Im Falle der globalen Beleuchtung handelt es sich allerdings tatsächlich um eine Mischform aus Voxelraum und Bildraum, sodass diese im Folgenden

Hybrid-GI genannt wird. Das Einsammeln des indirekten Lichts basiert auf der Slicemap und der Textur mit den Leuchtdichten der direkten Beleuchtung. Die Sichtbarkeit wird mit Hilfe der voxelisierten Szene bestimmt, die Information über die Leuchtdichte an einem Punkt allerdings aus der Textur gelesen, sodass hier nur die rasterisierte Farbe verfügbar ist. Die Voxel selbst enthalten nur die binäre Information voll/leer, aber keine weiteren Attribute wie Farbe oder Leuchtdichte an der Stelle. Würde den Voxeln mehr als nur 1 Bit zur Verfügung gestellt, könnten diese auch mehr Informationen tragen. Die Auflösung in z-Richtung würde sich aber stark verringern; mehr Slicemaps wären nötig und der Vorteil der ursprünglich kompakten Speicherung wäre nicht mehr vorhanden.

Die voxelbasierten Shader benötigen mehr Eingaben als jene, die im Bildraum arbeiten: zusätzlich noch

- die Slicemap,
- eine 1D-Lookup-Textur für das Auslesen eines Voxels aus der Slicemap
- und 3 Matrizen für die Transformation von Kamerakoordinaten in Slicemap-Texturkoordinaten: die inverse View-Matrix der Darstellungskamera, die View-Matrix der Voxelisierungskamera und die Projektionsmatrix der Voxelisierungskamera.

Jene 3 Matrizen werden bereits im Vertex Shader miteinander multipliziert und das Ergebnis wird an den Fragment Shader weitergereicht. Der Hybrid-GI-Shader benötigt zudem die Projektionsmatrix der Darstellungskamera, um die Texturkoordinate für den Lookup der Leuchtdichte berechnen zu können.

Die wesentliche Änderung zum Screen-Space-Shader besteht darin, dass der Tiefenvergleich (Listing 5, Zeile 14-17) durch den Funktionsaufruf `isFilledVoxel(samplePos)` ersetzt wird. Die Funktion ist wie folgt definiert:

Listing 7: Voxel-Space: Liegt ein gefülltes Voxel an einer bestimmten Position vor?

```
1  bool isFilledVoxel(in vec3 p) // p in eye-space
2  {
3     p.z += 0.025; // shift towards camera (removes artifacts)
4     vec4 clipCoord = transformMatrix * vec4(p,1.0);
5     vec3 lookupCoord = (clipCoord.xyz+1.0)/2.0;
6  
7     uvec4 val = texture2DLod(slicemap,lookupCoord.xy,0);
8     uvec4 mask= texture1D(lookupMask,lookupCoord.z);
9     return (any(notEqual(uvec4(0),val & mask)));
10 }
```

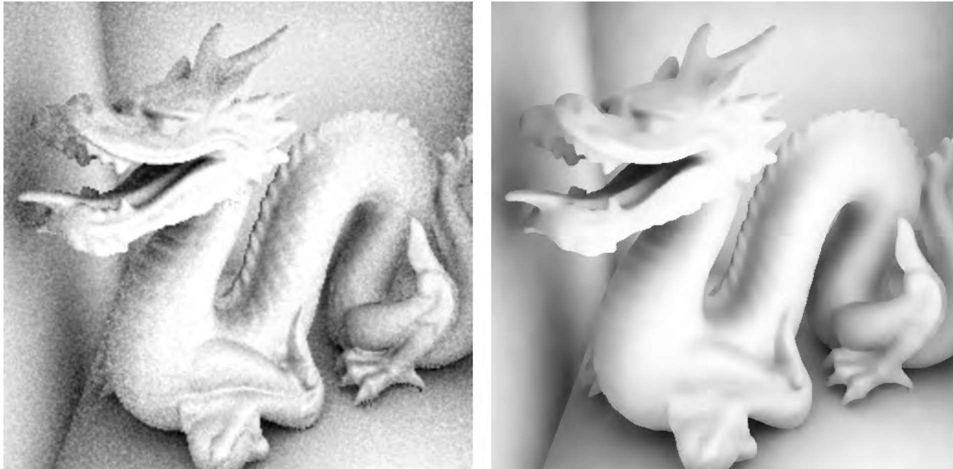


Abbildung 26: Geometrie-sensitiver Blur: *Links* die AO-Textur mit halber Auflösung (256x256), *rechts* die zugleich hochskalierte und weichgezeichnete Textur (512x512). Maskenbreite: 21 Pixel

5.4 Blur

Abhängig von der Kombination der Parameter Strahlanzahl, Schritte entlang eines Strahls und vor allem Radiusgröße ist das Ergebnis mehr oder weniger verrauscht. Das Rauschen tritt bei der Voxelraum-Variante etwas stärker auf. Eine wahrscheinliche Ursache ist, dass die Oberfläche der Szenenrepräsentation durch die Voxelisierung „klötzchenartig“ (eckig) ist statt „glatt“ wie im Bildraum, obwohl auch dieser diskretisiert ist.

Abhilfe schafft ein geometrie-sensitiver, kantenerhaltender Weichzeichner (Blur). Dieser erhält zusätzlich zu der verrauschten Textur als Eingabe auch die Normalen- und Kamerakoordinaten-Textur. So lässt sich erreichen, dass nicht über Kanten, Objektgrenzen oder starke Sprünge in den Normalen hinweg verwischt wird. Das Rauschen wird unterdrückt, aber auch Feinheiten gehen verloren. Letzteres stellt im Allgemeinen kein großes Problem dar, da Ambient Occlusion generell wenig Details erfasst.

Reinbothe et al. [RBA09] erweitern das von Kopf et al. vorgestellte Joint Bilateral Upsampling [KCLU07]. Ein bilateraler Filter ist ein nicht-linearer Filter, bei dem zwei spezielle Funktionen die Bildwerte innerhalb der Maske abhängig von ihrer Lage (*domain*) und Helligkeit (*range*) gewichten. Reinbothe et al. definieren nun eine Reihe von Gewichtungsfunktionen, die Normalen, Kamerakoordinaten, Pixelpositionen und Grauwerte als Eingabe erhalten. Eine zweite, leicht modifizierte Version wurde erstellt, um auch RGBA-Texturen weichzeichnen zu können.

Obwohl der Filter nicht separierbar ist, wird er trotzdem näherungsweise in zwei Passes auf die verrauschte AO-/GI-Textur angewendet, um die Geschwindigkeit zu erhöhen. Im ersten Pass wird in x-Richtung mit einer

$m \times 1$-Maske gefiltert, im zweiten Pass in y-Richtung mit einer $1 \times m$-Maske und mit der Ergebnis-Textur des ersten Passes als Eingabe. m ist die benutzerdefinierte Maskengröße in Pixeln.

Die Anwendung des Blurs ist insbesondere dann notwendig, wenn die AO-/GI-Textur nicht voll aufgelöst ist (siehe Abbildung 26). Aber auch voll aufgelöste Texturen können von einer Rauschunterdrückung profitieren.

5.5 Finale Kombination

Am Ende müssen die während des Renderings erzeugten Komponenten zusammengesetzt und auf einen für den Bildschirm darstellbaren Bereich abgebildet werden. Dafür kommt der Tonemapper von Drago et al. [DMAC03] zum Einsatz. Der letzte Schritt ist eine Gamma-Korrektur $(R, G, B, A)^{\frac{1}{\gamma}}$ mit einem angenommenen Gamma-Wert $\gamma = 2.2$.

Kombiniert werden müssen:

- Leuchtdichte der direkten Beleuchtung (L_{dir})
- Leuchtdichte eines ambienten Umgebungslichtes (L_{amb})
- diffuse BRDF ($f_{r,d}$)
- Ambient-Occlusion-Term (AO)
- Beleuchtungsstärke der indirekten Beleuchtung (E_{ind}), damit ist gegeben: $L_{ind} = f_{r,d} \cdot E_{ind}$

Akenine-Möller et al. schlagen vor, in rein diffusen Umgebungen die direkte Beleuchtung mit einem ambienten Licht und dem AO-Term wie folgt zu kombinieren [AMHH08, S. 376]:

$$L_{out} = f_{r,d} \cdot AO \cdot \pi L_{amb} + L_{dir}$$

πL_{amb} ist die ambiente Beleuchtungsstärke.

Durch Experimentieren wurde schließlich folgende Formel zur Kombination aller genannten Faktoren gefunden:

$$L_{out} = f_{r,d} \cdot (AO \cdot \pi L_{amb} + k \cdot E_{ind}) + L_{dir} \cdot AO$$

Der benutzerdefinierte Parameter k skaliert das indirekte Licht. Würde der Ambient-Occlusion-Term nur das ambiente Licht verdunkeln, müsste dieses sehr hell sein, damit ein Effekt sichtbar wird. Daher wird der AO-Term zusätzlich auf die direkte Beleuchtung angewendet. Da Ambient Occlusion kein physikalisch korrektes Konzept ist und alle Berechnungen nur näherungsweise ausgeführt werden, wurde mehr Wert auf ein optisch ansprechendes Ergebnis gelegt. Dieses ist nur durch Nutzerinteraktion (passende Wahl der Parameter) zu erreichen. Ein Beispielergebnis ist in Abbildung 27 zu sehen.

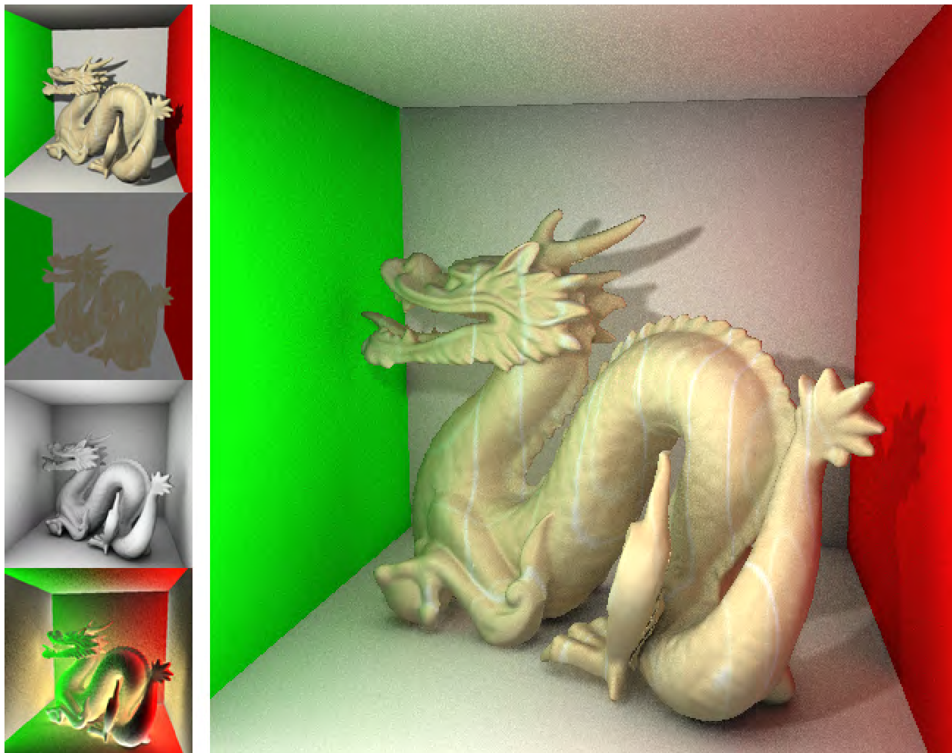


Abbildung 27: Finale Kombination: Beispiel eines Endergebnisses (die Streifen auf dem Drachen stammen von einer Textur)

6 Ergebnisse

6.1 Qualität

Im Folgenden werden Unterschiede und Besonderheiten der Bildraum- und Voxelraum-Verfahren in Bezug auf die Bildqualität betrachtet. Beide Varianten liefern bei richtiger Wahl der Parameter überzeugende Ergebnisse.

Im Bildraum entstehen *Artefakte* durch verdeckte Objekte und am Bildrand. Besonders auffällig sind springende Schatten (*popping shadows*) bei bewegten Bildern: Schatten können sprungartig verschwinden oder auftauchen. Bei statischen Bildern sind sie weniger auffällig. Abbildung 28 (Mitte) zeigt einen derartigen Effekt. Beim Bewegen des Bunnys oder der Kamera würde sich der Fehler an der Wand hinter dem Bunny mitbewegen. Auch der Kontaktschatten zwischen Bunny und Boden wäre nicht stabil. Die Sprünge sind jedoch nicht abgehackt, sondern eher weich und daher, je nachdem, wie ausgeprägt sie sind, tolerierbar. Ob Artefakte mehr oder weniger stark auffallen, ist natürlich auch von der Wahrnehmung des Betrachters abhängig.

Das voxelbasierte Ambient Occlusion dagegen bleibt aus jeder Blick-

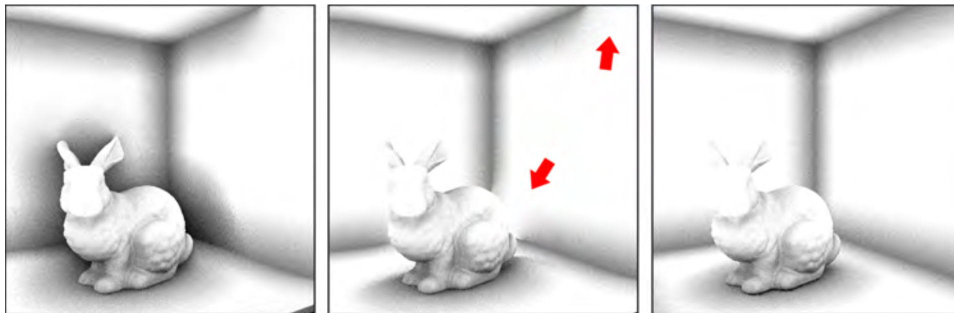


Abbildung 28: *Links und Mitte:* Screen-Space AO, *rechts* Voxel-Space AO. Identische Parameterwahl bei beiden Verfahren. Abschwächung der Samples: keine. Das Bild *links* entstand ohne Radiuscheck („Befindet sich ein Sample innerhalb des Suchradius?“). Beim Bild in der *Mitte* war dieser Radiuscheck aktiviert. Es ist klar ersichtlich, dass der Radiuscheck im Bildraum unabdingbar ist. Im Voxelraum (Bild *rechts*) ist ein solcher Radiuscheck nicht nötig. Auftretende Artefakte im Bildraum sind mit Pfeilen gekennzeichnet: fehlerhafte AO-Berechnung an der Wand durch das verdeckende Bunny und am Bildrand (fehlende Verschattung).

richtung und bei Objektbewegungen stabil, da die Sichtbarkeitsbestimmung nicht von der Kamerablickrichtung oder den für die Kamera sichtbaren Objekten abhängig ist.

Für SSGI und Hybrid-GI gilt: *Indirektes Licht* kann nur von im Bild sichtbaren Flächen reflektiert werden. Ein Beispiel zeigt Abbildung 29. Interessant ist der Unterschied zwischen SSGI und Hybrid-GI bei diesem Grenzfall. SSGI ignoriert weitestgehend das kleine Stück rote Wand als Quelle indirekten Lichtes. Der Grund ist, dass die Wand im Bildraum nicht als stark verdeckendes Objekt erkannt wurde. Bei Hybrid-GI ist dies anders, da die Wand im Voxelraum als Occluder erkannt wird. Der Auftreffpunkt wird zurück in den Bildraum projiziert und an der entsprechenden Stelle kann die Farbe ausgelesen werden. Befindet sich die Wand allerdings außerhalb des Bildes, liegt auch der projizierte Punkt außerhalb des Bildes, sodass hier nur die geclampte Randfarbe gelesen werden kann.

Die *Wahl der Parameter* trägt entscheidend dazu bei, wie gut die Qualität des finalen Bildes ist und wie stark der Ambient-Occlusion- oder GI-Effekt sichtbar wird. Insbesondere bei den Parametern zur Steuerung der Anzahl der Samples (Strahlen, Schritte) und Auflösungsgröße sind Qualität und Performance gegeneinander abzuwägen. Verfahren im Voxelraum sind rauschanfälliger als im Bildraum und benötigen daher tendenziell mehr Samples. Abbildung 30 zeigt die Ergebnisse verschiedener Strahl-/Schrittanzahlen.

Sobald die Framerate in der Anwendung unter 6 fps fällt, wird während einer Benutzerinteraktion (Bewegen eines Objektes, der Kamera oder der Punktlichtquelle) die Auflösung der AO-/GI-Textur auf ein Viertel reduziert.

Abbildung 29: Grenzfallbetrachtung: *Oben* SSGI, *unten* Hybrid-GI. *Links* ist die rote Wand noch knapp im Bild zu sehen und wirft daher indirektes rotes Licht auf die hintere Wand und den Drachen. Der Effekt auf dem Drachen ist nur bei Hybrid-GI deutlich zu erkennen. Warum SSGI hier den Color Bleeding-Effekt kaum zeigt, wird auf S. 46 erläutert. *Rechts* ist die rote Wand aus dem Bild verschwunden, sodass sie kein indirektes rotes Licht mehr reflektieren kann.

Nach Beendigung der Interaktion wird sie wieder auf die vorherige gewählte Auflösung gesetzt.

Im Allgemeinen ist ein großer Radius insbesondere bei GI erforderlich, um gute Ergebnisse zu erreichen (vgl. Abbildung 32). Kleine Radien sind für das indirekte Licht nicht sinnvoll, denn auch in der Realität ist zu beobachten, dass farbige Objekte auf weiter entfernte abstrahlen.

Zu beachten ist: Je größer der Radius ist, desto verrauschter wird das Ergebnis, wenn nicht die Samples entsprechend vervielfacht werden, um den

größeren Raum abdecken zu können. Das Rausch-Problem ist wie erwähnt bei den voxelbasierten Verfahren viel größer. Abbildung 31 zeigt Ambient Occlusion mit verschiedenen Radien.

Abbildung 30: Wahl der Strahl- und Schrittanzahl (Verfahren: Voxel-Space AO, in jedem Bild: Radius = 3.0, Slicemap-Auflösung: 128^2, Bildauflösung: 512^2)

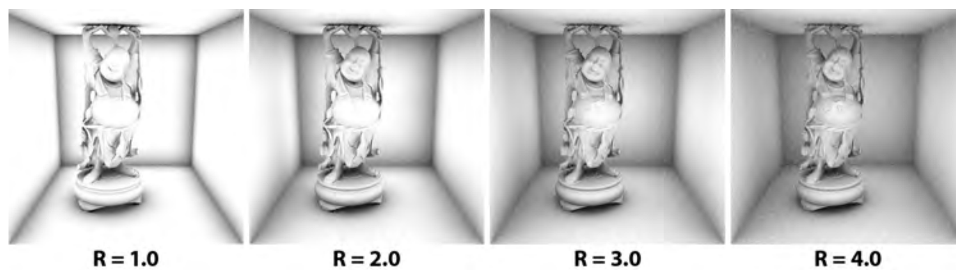


Abbildung 31: Radiusgröße (AO). Zur Orientierung: Die Cornell-Box hat die Breite 5. (Verfahren: Voxel-Space AO, in jedem Bild: 20 Strahlen, 20 Schritte, Slicemap-Auflösung: 128^2, Bildauflösung: 512^2)

Ein typisches Artefakt von SSGI wird exemplarisch in Abbildung 33 gezeigt. Abhängig von der Radiusgröße erscheinen leuchtende Flecken.

In den Bildern sind pixelige Kanten sichtbar. *Anti-Aliasing* ist bei Deferred Shading nicht trivial. Die erste angedachte naive Lösung, einfach ein FBO mit Multisampling zu benutzen, erwies sich als falsch, da anti-aliased Normalen und Positionen keine Bedeutung haben und zu gravierenden Darstellungsfehlern führen. Es gibt verschiedene Ansätze, Anti-Aliasing bei Deferred Shading zu realisieren, sie wurden im Rahmen dieser Studienarbeit jedoch nicht weiter betrachtet.

Insgesamt lässt sich sagen, dass mit beiden GI-Varianten optisch sehr ansprechende Ergebnisse möglich sind. Beispiele zeigen die Abbildungen 27, 34 - 37 und Abbildung 1 in der Einführung.

Das indirekte Licht in den in Abbildung 34 gezeigten Bildern wurde verstärkt, um den Color-Bleeding-Effekt hervorzuheben. Bei solch statischen

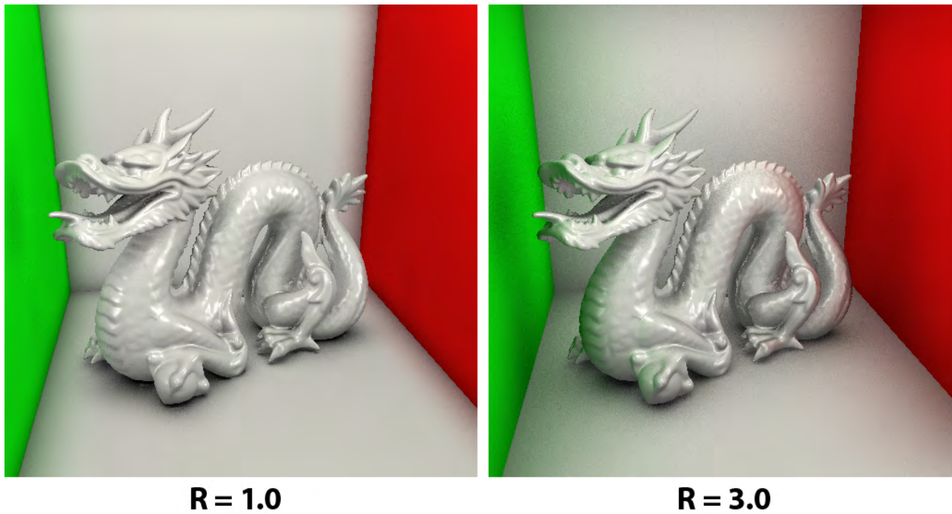


Abbildung 32: Große Radien (in Relation zur Szenengröße) sind für GI sinnvoll. (Hier: Hybrid-GI)

Bildern sieht SSGI den Hybrid-GI Ergebnissen meistens sehr ähnlich, daher wird hier nur die Hybrid-GI Variante gezeigt. Der Vorteil von Hybrid-GI, die bessere Konstanz bei Objekt- oder Kamerabewegungen, kommt hier nicht zur Geltung.

Abbildung 36 vergleicht SSGI mit Hybrid-GI in einer Szene, deren Objekt-Wand-Konstellation der Grenzfallbetrachtung aus Abbildung 29 entspricht. Somit ist bei dem mit SSGI erzeugten Bild (Mitte) auf dem Schaf kein Color Bleeding zu sehen. Bei einem anderen Betrachtungswinkel (rechts) erscheint auch bei SSGI das Color Bleeding. Jedoch ist bei beiden SSGI-Bildern auf dem Bunny kaum orangefarbenes Licht auszumachen.

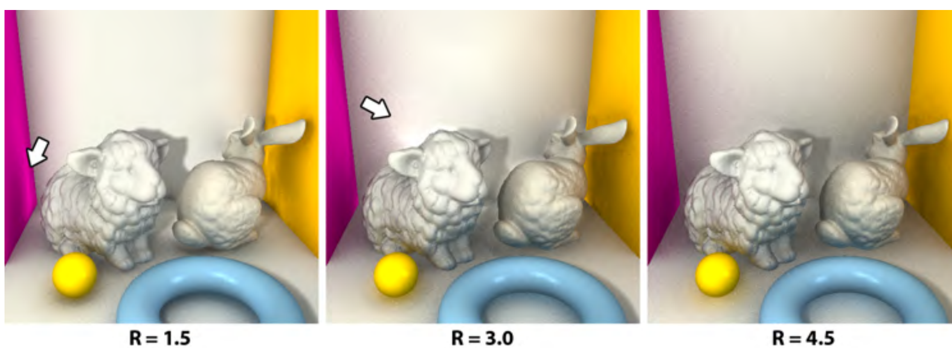


Abbildung 33: SSGI: Leuchtende Flecken. Mögliche Ursache: Fehlerhafte Verschattung und zu viel indirektes Licht an den Stellen

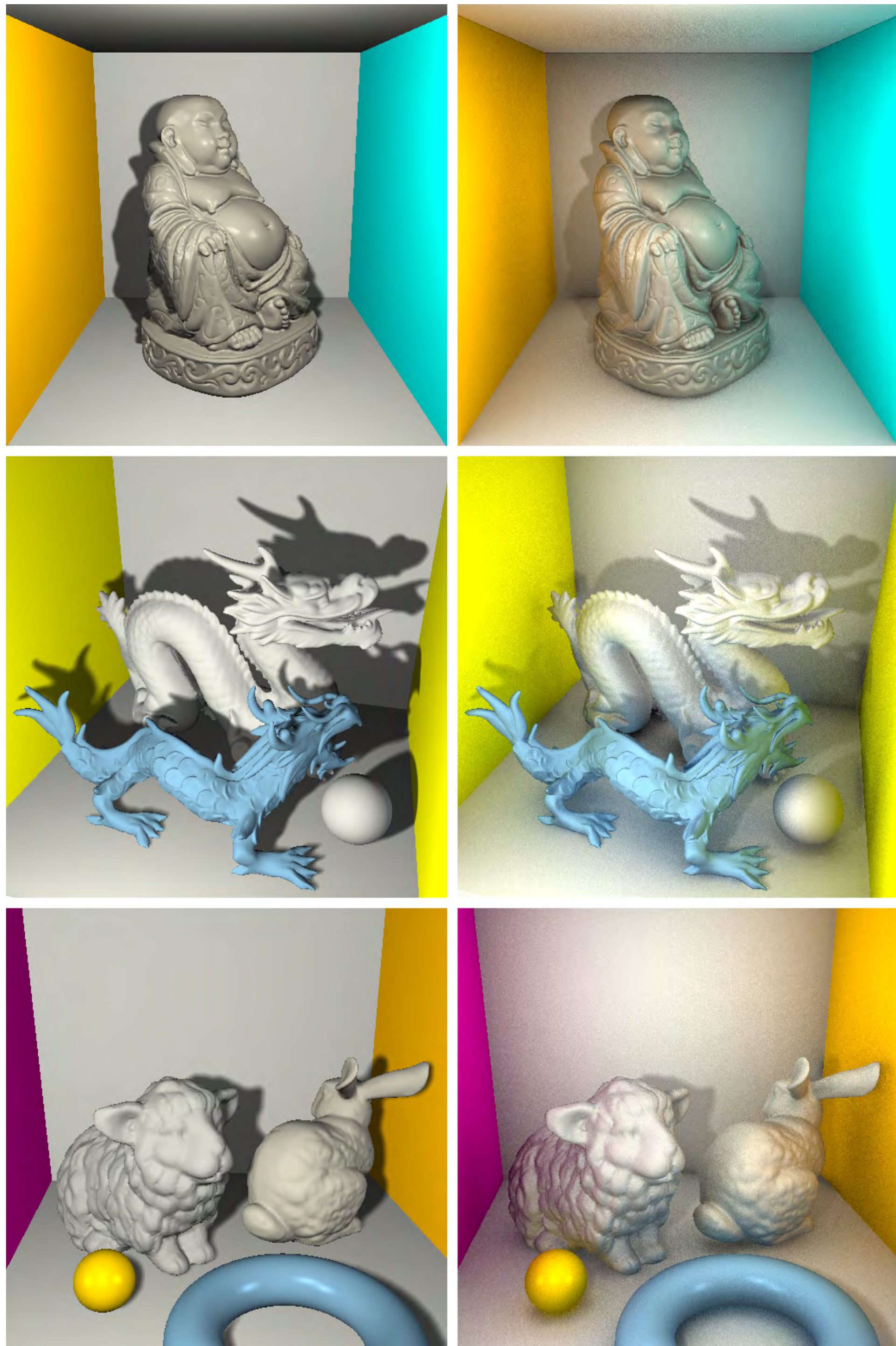


Abbildung 34: *Links:* Direkte Beleuchtung mit Shadow Mapping.
Rechts: Hybrid-GI. FPS siehe Tabelle 2.

Abbildung 35: Die Szenen aus Abbildung 34 nur mit Voxel-Space AO

Abbildung 36: Vergleich Hybrid-GI / SSGI. In der Mitte fehlt das Color Bleeding auf dem Schaf. Dieses wird erst sichtbar, wenn die pinkfarbene Wand ein beträchtliches Stück weiter ins Bild gewandert ist.

Abbildung 37: *Links:* Direkte Beleuchtung ohne Shadow Mapping, eine Punktlichtquelle. *Rechts:* SSAO Ray Marching

6.2 Performance

Ray Marching Methoden liefern qualitativ hochwertige Ergebnisse, haben jedoch auch einen hohen Aufwand, der abhängt von der Anzahl der Strahlen und der Schritte entlang eines Strahls. Entsprechend viele Texturzugriffe sind nötig (Anzahl der Zugriffe = Strahlanzahl×Schrittanzahl, z. B. $32 \times 16 = 512$ (!) bei 32 Strahlen und 16 Schritten). Die Zugriffe sind aufgrund der Art, wie die Samples erzeugt werden, zufällig von Strahl zu Strahl und liegen je nach Radiusgröße weit auseinander. Somit ist die Wahrscheinlichkeit für Textur-Cache-Misses extrem hoch.

Die übliche Methode, um den Aufwand zu verringern, ist das Rendern der AO-/GI-Textur in geringerer Auflösung mit anschließendem Blurring. Mit dem vorgestellten Blur sind die Ergebnisbilder zwar nicht mehr verrauscht, haben aber trotz Geometrie-Sensitivität einen sehr weichen und teils detailärmeren Eindruck (siehe Abbildung 38). Die Verwendung einer GI-Textur mit halber Auflösung und Blur brachte in den Tests mindestens eine Verdopplung bis Verdreifachung der Framerate (vgl. Tabelle 2).

Abbildung 38: *Links:* Buddha mit voll aufgelöster GI-Textur, *rechts* mit halber Auflösung und Blur

Dynamisches Branching, insbesondere ein frühzeitiges bedingtes Verlassen einer Schleife mit einem `break`, ist auf der GPU immer noch teuer. In Listing 5 (S. 40) könnte die `for`-Schleife – sobald ein Schnittpunkt gefunden wurde – nach Zeile 13 mit einem `break` verlassen werden. Während der Implementierung stellte sich heraus, dass dadurch die Performance des Fragment Shaders um bis zu 20 % gesenkt wird. Anstelle eines bedingten Verlassens der Schleife werden daher am Anfang der Schleife nur die folgenden Berechnungen abgefangen, falls schon ein Schnittpunkt gefunden wurde.

Dem Ray Marching gegenüber stehen SSAO-Varianten wie jene von Crytek, die wesentlich weniger Samples einsetzen (meist weniger als 32). Sie sind

nicht strahlbasiert, sondern verteilen nur einzelne Punkte zufällig im Raum. Deshalb erzielen sie meist ungenauere Ergebnisse bei jedoch höherer Performance. Testweise wurde ein Punkte-Sampling im Bildraum implementiert, bei dem eine gewisse Anzahl von Punkten zufällig (auf Grundlage einer 3D-Hammersley-Sequenz) in der Hemisphäre eines betrachteten Oberflächenpunktes verteilt werden. Die Ergebnisse sind in Abbildung 39 zu sehen. Die Messungen der Frameraten sind in Tabelle 1 aufgeführt.

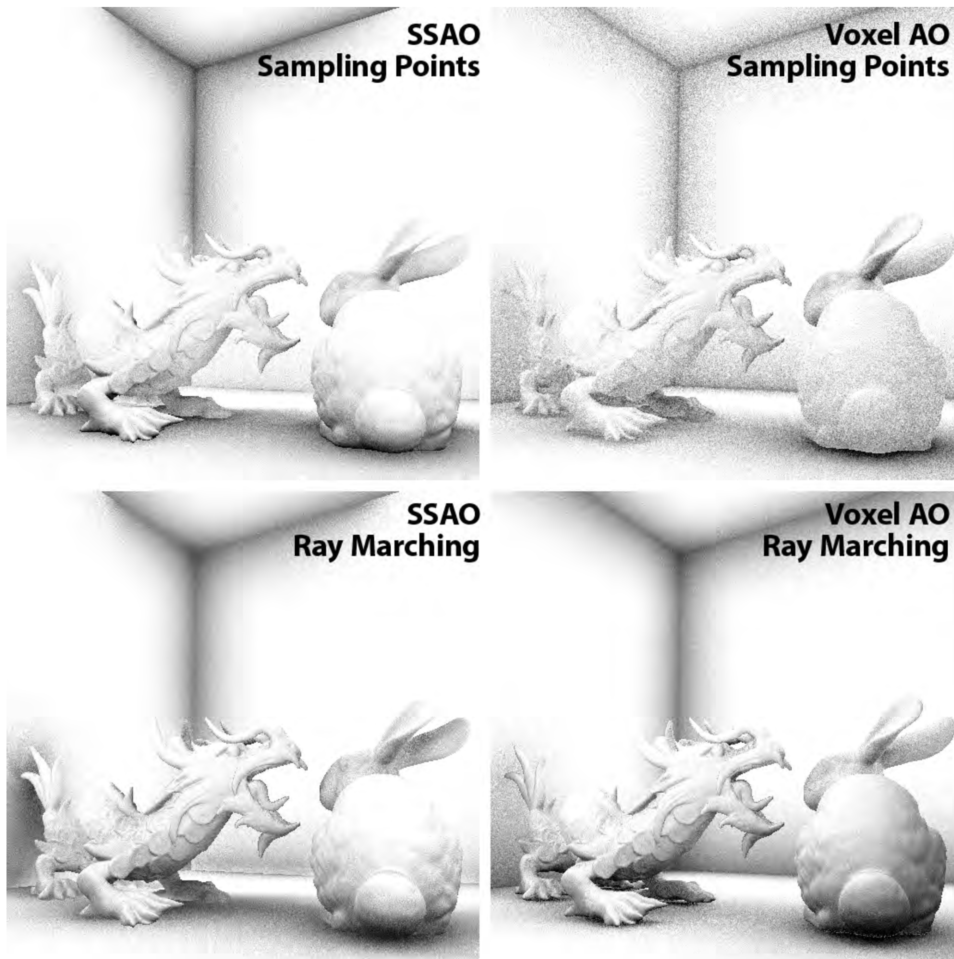


Abbildung 39: Vergleich Ray Marching (je 12 Strahlen, 10 Schritte) und einzelne Samples (32 Punkte in Hemisphäre). Radius = 1,5. Slicemap-Auflösung bei Voxelisierung: 128^2. Das Sampling einzelner Punkte liefert im Voxelraum stark verrauschte Bilder. Im Bildraum können beide Sampling-Methoden ohne ein derartiges Problem eingesetzt werden; im Voxelraum ist nur Ray Marching empfehlenswert.

Die Performance-Messungen wurden auf zwei Systemen durchgeführt:

System 1	**System 2**
NVIDIA GeForce 8600 GTS (256 MB), Intel Core 2 Duo 6420 (2x2,13 GHz), 2 GB RAM	NVIDIA GeForce GTX 260 (896 MB), AMD Phenom II X4 940 (4x3,00 GHz), 4 GB RAM

Tabelle 1: fps: Ray Marching vs. Sampling Points (Abbildung 39)

	Sampling Points		**Ray Marching**	
SSAO	24.2	79.9	9.6	35.7
VoxelAO	26.5	82.2	13.8	46.0

Voxel-Space AO ist etwas schneller als SSAO, da der Lookup in der 128^2-Slicemap (4 Kanäle à 32 Bit) schneller ist als in der 512^2-Kamerakoordinaten-Textur (3 Kanäle à 32 Bit). Es wurde nicht untersucht, wie Qualität und Performance bei einer ebenso kleinen Kamerakoordinaten-Textur wären.

Tabelle 2: fps: Szenen aus Abbildung 34

Szene	**Parameter**	**Frames per Second (fps)** direkte Bel.		AO		GI			
Buddha (200 000 Dreiecke)	32/32 4.0	55.3	146.0	1.9	7.9	1.8	7.4	V	*full*
				1.5	6.3	1.1	5.9	S	
						5.3	18.7	V	*half +*
						3.5	16.2	S	*Blur 12*
Sheep+ Bunny (200 000 Dreiecke)	24/20 3.5	77.6	209.9	3.9	16.4	3.6	15.4	V	*full*
				3.1	12.9	2.2	12.3	S	
						9.3	33.0	V	*half +*
						6.5	29.2	S	*Blur 10*
Dragons (287 000 Dreiecke)	32/20 3.5	43.0	106.8	3.0	11.8	2.7	11.0	V	*full*
				2.4	10.0	1.7	9.4	S	
						6.9	23.7	V	*half +*
						5.1	22.0	S	*Blur 12*

Zu Tabelle 2: V = Voxel-Space, S = Screen-Space, *full/half* = voll/halb aufgelöste AO-/GI-Textur, *Blur 12* = Blur aktiviert mit Blurradius 12 Pixel. Die Performance beeinflussende Parameter sind Strahlen/Schritte, Radius (von oben nach unten in der Parameterspalte eingetragen), die AO-/GI-Textur-Auflösung und bei aktiviertem Blur der Blurradius (siehe Spalte ganz rechts). Alle Bilder haben eine Auflösung von 512×512. Die Slicemap hat immer eine Auflösung von 128^2. (Diese Auflösung ist ebenfalls ein Performance beeinflussender Parameter.)

6.3 Erweiterungsmöglichkeiten

Mit einem geschickteren G-Buffer-Layout lässt sich Texturspeicher sparen. Die Normalen und die Positionen könnten in einer einzelnen Textur zusammengeführt werden: 2 Kanäle für die Normale, 1 Kanal für die Tiefe (Positionen müssen rekonstruiert werden, siehe Abschnitt 4.3 auf S. 23). Dafür würde ausgenutzt, dass die Normalen mit Länge 1 vorliegen und die dritte Komponente somit aus den ersten beiden bestimmbar ist. Das Vorzeichen der dritten Komponente kann als Vorzeichen des Tiefenwerts gespeichert werden, sofern alle Tiefenwerte dasselbe Vorzeichen besitzen und erkennbar ist, ob es umgedreht wurde. Falls sich durch die G-Buffer-Änderung keine nennenswerte Performance-Steigerung erzielen lässt, wären die Gründe dafür zu suchen.

Das eindeutige Bottleneck ist das Sampling, also die zahlreichen Texturzugriffe. NVIDIAs Horizon-Based AO erreicht durch ein geschickteres Sampling ähnliche Ergebnisse wie Ray Marching im Bildraum und ist dabei wesentlich schneller. Es wäre interessant zu untersuchen, inwieweit sich dieses Sampling im Voxelraum durchführen lässt. Ebenfalls wäre die Übertragung von Screen-Space Directional Occlusion auf den Voxelraum interessant. Im folgenden Abschnitt wird eine Idee für einen schnelleren bzw. genaueren Strahlschnitttest im Voxelraum erläutert.

Die Samples werden erst auf der GPU ausgehend von einer Low-Discrepancy-Sequenz und einer Zufallszahl erzeugt. Effizienter wäre es, bereits fertig berechnete Samples zu übergeben und diese zufällig zu modifzieren. Während der Implementierung wurde derartiges versucht, jedoch dominierte starkes Rauschen das Ergebnis, sodass keine zufriedenstellende Lösung gefunden wurde.

Des Weiteren könnten die Bent Normals für die Beleuchtung herangezogen werden - entweder für die direkte Beleuchtung oder für die indirekte mit einem Lookup in diffusen Environment Maps.

Das implementierte Shadow Mapping ist verbesserungswürdig. Mit dem derzeit verwendeten Model-Loader können keine komplexen Szenen vernünftig geladen werden. Bisher sind nur quadratische Auflösungen möglich.

Als letzter Punkt wäre zu nennen: Anti-Aliasing wurde in dieser Studienarbeit nicht umgesetzt, wäre aber ein wünschenswertes Feature.

6.4 Ausblick: Strahlschnitttest im Voxelraum

In den vorherigen Abschnitten ist deutlich geworden, dass sich im Voxelraum gute und vor allem konstante Ergebnisse erzielen lassen. Dafür ist allerdings ein geeignetes Vorgehen für den Schnitt von Strahlen mit der Voxelszene erforderlich. Bei Ambient Occlusion ergibt beispielsweise das spärliche Sampling einzelner Punkte im Halbraum oder in der Hemisphäre keine guten Ergebnisse. Der Grund ist, dass diese Art des Samplings nur einen

zufälligen Punkt entlang eines Strahls testet und daraufhin für die gesamte Richtung entscheidet, ob sie verdeckt ist. Ray Marching testet mehrere Voxel entlang eines Strahls. Optimal wäre es, *alle* Voxel entlang des Strahls zu testen. Je nach Strahllänge und Slicemap-Auflösung wäre dieses Vorgehen bei einfachem Ray Marching sehr langsam.

Während der Studienarbeit wurde daher die Idee entwickelt, die in Abschnitt 3.4 (S. 19 ff.) beschriebenen Slicemap-Mipmaps zu nutzen, um einen Strahl mit der Voxelszene zu schneiden. Die Visualisierung der Mipmaps (Abbildung 14 auf S. 21) zeigt, dass die vollen Voxel der höheren Mipmapstufen anzeigen, dass sich auf niedrigerer Ebene innerhalb dieses Bereichs mindestens ein volles Voxel (also letztendlich Geometrie) befindet. Leere Voxel auf höheren Mipmapstufen wiederum bedeuteten, dass sich in dem gesamten Bereich auf allen niedrigeren Ebenen keine vollen Voxel und somit keine Geometrie befindet. In diesen leeren Bereichen wird garantiert kein Schnittpunkt gefunden, sodass sie beim Schnitttest übersprungen werden können. Bereiche, in denen sich mindestens ein volles Voxel befindet, müssen dagegen genauer untersucht werden.

Die Mipmaps ähneln einem Quadtree, da sie durch schrittweise Reduzierung der x- und y-Auflösung, nicht jedoch der z-Auflösung, entstanden. Die Suche nach einem Schnittpunkt eines Strahls mit den Voxeln wird somit zu einer Suche im Baum. Ein Knoten auf Mipmapstufe n ist ein Texel der Slicemap-Mipmap dieses Levels. Er hat 4 Kinder auf Stufe n-1.

Wichtig ist für die Betrachtung, sich ein Texel (Voxelspalte) als Quader vorzustellen (wie in Abbildung 8 auf S. 15). Diese Quader sind Axis-Aligned-Bounding-Boxes (AABBs) im Voxelisierungskamera-Koordinatensystem. Die Position und Abmessungen eines Quaders sind durch das Voxelisierungskamera-Frustum, die Texelkoordinaten, die Slicemap-Auflösung und das aktuelle Mipmap-Level gegeben.

Der grobe Ablauf des Algorithmus ist wie folgt: Ein AABB-Schnitttest eines Strahls mit einem Quader liefert entweder keinen Schnitt oder den Eintritts- und/oder Austrittspunkt. Bei keinem Schnitt werden die Geschwisterknoten untersucht. Bei einem Schnitt wird das Strahlstück als Voxelspalte konstruiert, und zwar mit vollen Voxeln zwischen den z-Werten des Ein- und Austrittspunkts. Die Verundung des Strahlstücks mit dem Texel des geschnittenen Quaders ergibt, ob der Strahl auf diesem Level volle Voxel geschnitten hat. Falls ja, werden die 4 Kindknoten untersucht, sofern nicht schon der niedrigste Mipmap-Level erreicht wurde. In diesem Fall wäre die Suche mit dem Ergebnis „Schnitt gefunden“ beendet. Die Details, wie und unter welchen Bedingungen im Baum auf- oder abgestiegen wird, werden hier nicht erläutert.

Die Erwartung ist, dass die Berechnung genauer und unter Einbeziehung dieser Genauigkeit schneller als das herkömmliche Ray Marching ist.

6.5 Fazit

In der Studienarbeit wurde untersucht, wie Sichtbarkeitstests im Kontext eines Deferred Shadings durchgeführt werden können. Sichtbarkeitstests arbeiten rein geometrisch, ihre Grundlage bildet die Szenenrepräsentation. Zwei Arten, die Geometrie näherungsweise zu repräsentieren – (1) Bildraum und (2) Voxelraum – wurden genauer betrachtet und im Rahmen von Ambient Occlusion für die Sichtbarkeitsbestimmung eingesetzt. Darauf aufbauend ließen sich globale Beleuchtungseffekte im Bildraum realisieren.

Die Ergebnisse beider implementierter Verfahren sind visuell überzeugend. Das voxelbasierte Verfahren kennt näherungsweise die gesamte Szene, sodass die Sichtbarkeitstests blickwinkelunabhängig sind. Es liefert daher vor allem in dynamischen Szenen und bei Kameraschwenks stabile Ergebnisse. Dem gegenüber stehen potentiell springende Schatten im Bildraum. Die Integration indirekten Lichts beruht wiederum nur auf Bildraum-Information, sodass die gezeigten Color-Bleeding-Effekte sowohl für das voxel- als auch das bildraumbasierte GI-Verfahren von den im Bild zu sehenden Flächen abhängt. Hybrid-GI schneidet bei den betrachteten Grenzfällen besser ab.

Die Verfahren sind gegeneinander abzuwägen in Bezug auf Genauigkeit und Flexibilität. Voxel-Space AO ist genauer, erfordert aber spezielle Meshes. Reine Bildraum-Verfahren können auf alles angewendet werden, was in den Bildraum rasterisiert wurde.

SSAO wird bereits in aktuellen Computerspielen eingesetzt (Crysis, Burnout Paradise) und ist für das kommende StarCraft 2 fest eingeplant. Es scheint nur eine Frage der Zeit zu sein, bis sich auch Screen-Space-Verfahren für globale Belechtungseffekte wie Color Bleeding in künftigen Spielegenerationen als Standard etablieren.

Literatur

[AMHH08] Akenine-Möller, Thomas ; Haines, Eric ; Hoffman, Naty: *Real-Time Rendering*. 3. A K Peters, Ltd., 2008

[BS09] *Kapitel* 6.2 Image-Space Horizon-Based Ambient Occlusion. In: Bavoil, Louis ; Sainz, Miguel: *Shader X7: Advanced Rendering Techniques*. Course Technology, 2009, S. 425–444

[BSD08] Bavoil, Louis ; Sainz, Miguel ; Dimitrov, Rouslan: Image-space horizon-based ambient occlusion. In: *SIGGRAPH '08: ACM SIGGRAPH 2008 talks*, ACM, 2008. – DOI: http://doi.acm.org/10.1145/1401032.1401061 (letzter Zugriff am 10.08.2009)

[Bun05] *Kapitel* 14 Dynamic Ambient Occlusion and Indirect Lighting. In: Bunnell, Michael: *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005, 223-233. – Online unter http://download.nvidia.com/developer/GPU_Gems_2/GPU_Gems2_ch14.pdf (letzter Zugriff am 04.06.2009)

[DMAC03] Drago, Frederic ; Myszkowski, Karol ; Annen, Thomas ; Chiba, Norishige: Adaptive Logarithmic Mapping For Displaying High Contrast Scenes. In: Brunet, Pere (Hrsg.) ; Fellner, Dieter W. (Hrsg.): *The European Association for Computer Graphics 24th Annual Conference: EUROGRAPHICS 2003* Bd. 22(3). Granada, Spain : Blackwell, 2003 (Computer Graphics Forum 3), S. 419–426. – Online unter http://www.mpi-inf.mpg.de/resources/tmo/logmap/logmap.pdf (letzter Zugriff am 17.09.2009)

[ED06] Eisemann, Elmar ; Décoret, Xavier: Fast Scene Voxelization and Applications. In: *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM SIGGRAPH, 2006, 71-78. – Online unter http://artis.imag.fr/Publications/2006/ED06 (letzter Zugriff am 10.08.2009)

[ED08] Eisemann, Elmar ; Décoret, Xavier: Single-pass GPU Solid Voxelization and Applications. In: *GI '08: Proceedings of Graphics Interface 2008* Bd. 322, Canadian Information Processing Society, 2008 (ACM International Conference Proceeding Series), 73-80. – Online unter http://artis.imag.fr/Publications/2008/ED08a (letzter Zugriff am 10.08.2009)

[Kaj09] *Kapitel* 6.1 Screen-Space Ambient Occlusion. In: Kajalin, Vladimir: *Shader X7: Advanced Rendering Techniques*. Course Technology, 2009 (ShaderX), S. 413–424

[KCLU07] Kopf, Johannes ; Cohen, Michael ; Lischinski, Dani ; Uyttendaele, Matt: Joint Bilateral Upsampling. In: *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2007* 26 (2007), Nr. 3. – Online unter http://johanneskopf.de/publications/jbu/paper/FinalPaper_0185.pdf (letzter Zugriff am 17.09.2009)

[Kne07] Knecht, Martin: *State of the Art Report on Ambient Occlusion*. Technical University of Vienna, 2007. – Online unter http://www.cg.tuwien.ac.at/research/publications/2007/knecht-2007-ao/knecht-2007-ao-paper.pdf (letzter Zugriff am 06.05.2009)

[Lan02] Landis, Hayden: *Production-Ready Global Illumination*. ACM SIGGRAPH Course Notes, Course 16 (RenderMan in Production), Juli 2002. – Online unter http://www.debevec.org/HDRI2004/landis-S2002-course16-prodreadyGI.pdf (letzter Zugriff am 24.8.2009)

[LB99] Langer, Michael S. ; Bülthoff, Heinrich H.: Perception of shape from shading on a cloudy day / Max-Planck-Institut für biologische Kybernetik. Tübingen, Deutschland, Oktober 1999 (73). – Forschungsbericht. – Online unter http://www.kyb.mpg.de/publications/pdfs/pdf1539.pdf (letzter Zugriff am 04.09.2009)

[Mit07] Mittring, Martin: Finding next gen: CryEngine 2. In: *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, ACM, 2007, 97–121. – Online unter http://ati.amd.com/developer/SIGGRAPH07/Chapter8-Mittring-Finding_NextGen_CryEngine2.pdf (letzter Zugriff am 06.05.2009)

[Mén07] Méndez Feliu, Àlex: *Fast Photorealistic Techniques to Simulate Global Illumination in Videogames and Virtual Environments*, Universitat Politècnica de Catalunya, Diss., April 2007. – Online unter http://ima.udg.edu/~amendez/thesis/thesis.pdf (letzter Zugriff am 04.09.2009)

[PH04] Pharr, Matt ; Humphreys, Greg: *Physically Based Rendering: From Theory to Implementation*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2004

[RBA09] Reinbothe, Christoph ; Boubekeur, Tamy ; Alexa, Marc: Hybrid Ambient Occlusion. In: *EUROGRAPHICS 2009 Areas Papers* (2009). – Online unter http://perso.telecom-paristech.fr/~boubek/papers/HAO/HAO.pdf (letzter Zugriff am 06.05.2009)

[RGS09] Ritschel, Tobias ; Grosch, Thorsten ; Seidel, Hans-Peter: Approximating Dynamic Global Illumination in Image Space. In: *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games*, ACM, 2009, 75–82. – Online unter http://www.mpi-inf.mpg.de/~ritschel/Papers/SSDO.pdf (letzter Zugriff am 06.05.2009)

[ZIK98] Zhukov, Sergej ; Inoes, Andrej ; Kronin, Grigorij: An Ambient Light Illumination Model. In: Drettakis, George (Hrsg.) ; Max, Nelson (Hrsg.): *Rendering Techniques '98*, Springer-Verlag Wien New York, 1998 (Eurographics), S. 45–56