Informatik
Hauptcampus

H O C H
S C H U L E
T R I E R

Bildraum Approximation globaler Beleuchtungseffekte mit Vulkan

Screen Space Global Illumination Approximation with Vulkan

Florian Oetke

Bachelor-Abschlussarbeit

Betreuer: Prof. Dr. Christof Rezk-Salama

Trier, 11.12.2017

Vorwort

Diese Arbeit sowie die beigefügte Implementierung entstand zwischen März und September 2017[1] als Abschlussarbeit meines Bachelorstudiums der Informatik an der Fachhochschule Trier.

Mein persönliches Ziel beim Verfassen dieser Arbeit war es nicht nur die generellen Grenzen der Einsetzbarkeit von inhärent lokalen Screen-Space-Verfahren zur Berechnung von globalen Beleuchtungsphänomenen zu prüfen, sondern auch mich in die für mich neue Vulkan-API einzuarbeiten.

Ich möchte mich an dieser Stelle bei allen bedanken, die durch ihre persönliche oder fachliche Unterstützung zu dieser Arbeit beigetragen haben.

Mein besonderer Dank gilt meiner Mutter Monika Oetke und meiner Schwester Nicole Oetke ohne deren Unterstützung (u.a. bei der Korrektur) die Sätze, Fußnoten, Klammern und Gedankenstriche in dieser Arbeit deutlich unhandlichere Ausmaße angenommen hätten und ohne deren Rückhalt ich vermutlich mindestens zweimal unterwegs das Thema gewechselt hätte.

Darüber hinaus möchte ich meinem Arbeitgeber der Rebel Creations AG und Dr. Klaus Netter im besonderen dafür danken, dass sie die zur Durchführung dieser Arbeit notwendige Hardware bereitgestellt haben.

Florian Oetke

Trier, 01. Oktober 2017

[1] Nicht eingerechnet das gute halbe Jahr davor, welches ich mit Nachforschungen zum Thema Echtzeitbeleuchtungsverfahren, auf der Suche nach einer geeigneten Grundlage für diese Arbeit verbracht habe.

Kurzfassung

In dieser Arbeit wird ein Bildraumverfahren zur Approximation indirekter Beleuchtung in dynamischen Szenen vorgestellt. Sowie zu einer vollwertigen globalen Beleuchtungslösung erweitert und abschließend seine Tauglichkeit für komplexe Echtzeitanwendung geprüft.

Das Verfahren basiert auf einer lokalen Monte-Carlo-Integration im Bildraum. Hierbei wird trotz einer relativ großen Anzahl von Samples eine hohe Geschwindigkeit erreicht, indem mittels Mip-Mapping nur lokale Textur-Samples zur Berechnung benötigt werden und so eine hohe Cachelokalität erreicht wird.

Der wesentliche Beitrag dieser Arbeit ist die Portierung des Verfahrens auf eine physikalisch plausible bidirektionale Reflektanzverteilungsfunktion (Cook-Torrance) sowie die dafür notwendige Erweiterung um eine Lösung für spekulare Reflextionen mittels Bildraum-Raytracing.

Bei der abschließenden Bewertung liegt der Fokus vor allem auf der Performance und Skalierbarkeit des Verfahrens um die Einsetzbarkeit auf schwächeren Plattformen zu bewerten. Anschließend werden basierend auf dieser Analyse mögliche Einsatzbereiche im Hinblick auf die inhärenten Probleme von Bildraumverfahren aufgezeigt.

With this Bachelor's Thesis, I present a screen-space algorithm for computing indirect illumination in dynamic scenes. Furthermore I describe the nessesary extensions to create a complete lighting solution and analyse its usability for complex real-time applications by integrating it into a deferred rendering pipeline.

The algorithm is based on screen-space local Monte-Carlo integration and on mip maps. In this way it only uses local texture samples, therefore, maximizing Cache locality to achieving high Efficiency, despite requiring a relatively large number of samples.

My main contribution in this thesis is the conversion of the original algorithm to a physically based BRDF (Cook-Torrance) and extending it with a solution for specular reflections based on screen-space raycasting.

The focus of the final evaluation is the performance and scalability of the approach to assess its usability on less powerful hardware. As well as an analysis of the possible applications with regard to the inherent issues of Screen-space algorithms is conducted.

Inhaltsverzeichnis

1

Problemstellung

Die subjektive Qualität einer dreidimensionalen Szene ist – abgesehen von stark stilisierten Darstellungen – primär von ihrer Realitätsnähe abhängig, d.h. in wie weit sich menschliche Wahrnehmungsheuristiken erfolgreich auf die Szene anwenden lassen um die Beziehung der Teilobjekte zueinander zu interpretieren. Eine bedeutende Rolle spielt dabei vor allem die Beleuchtung der Objekte sowie Schatten und die Interaktion von Licht zwischen den Teilobjekten durch Reflexionen und indirekte Beleuchtung (siehe Abb 1.1). [Pue89]

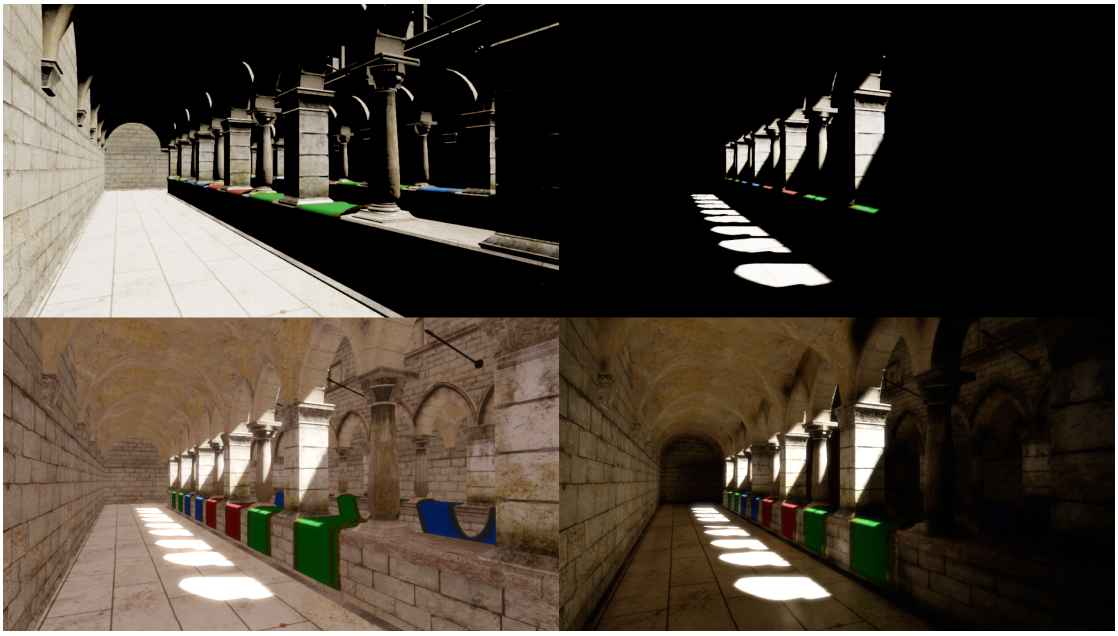


Abb. 1.1: Ein Beispiel für die Bedeutung der Beleuchtung einer Szene anhand der in dieser Arbeit entwickelten Anwendung. Von links oben im Uhrzeigersinn: Nur direkte Beleuchtung, mit Schatten, mit konstanter Umgebungsbeleuchtung und mit indirekter Beleuchtung.
Im Endergebnis ist ein größerer Teil der Szene erkennbar, ohne das dadurch die Lichtstimmung beeinträchtigt wird oder manuelle Anpassungen der Beleuchtung nötig sind. Durch die Interaktion des Lichts mit den Oberflächen lassen sich außerdem deutlich mehr Materialeigenschaften (Rauheit, Metall, etc.) ablesen und räumliche Beziehungen zwischen den Objekten herstellen.

Während direkte Beleuchtungseffekte durch aktuelle physikbasierte lokale Beleuchtungsmodelle bereits für die meisten Anwendungsfälle in Echtzeit hinreichend gut approximiert werden können, stellen globale Beleuchtungsmodelle in Echtzeitszenarien wie Computerspielen nach wie vor oft ein Problem dar. Dies ist dadurch bedingt, dass für diese komplexe Berechnungen auf einem nicht unerheblichen Teil der Szene erforderlich sind. Zusätzlich erschwert wird das Problem im Fall von Spielen durch ihren interaktiven Charakter, welcher die Wiederverwendung bereits berechneter Ergebnisse und Beschleunigungsstrukturen erschwert, sowie die durch ihre Interaktivität bedingten hohen Echtzeitanforderungen von maximal 33 Millisekunden pro Frame. Darüber hinaus ist die Hardware der Endanwender in diesem Bereich der Informatik vergleichsweise heterogen. Um den Entwicklungsaufwand für derartige globale Beleuchtungsverfahren zu rechtfertigen, müssen diese daher auch auf schwächerer Hardware zumindest einen gewissen Mehrwert bieten. [Kaj86]

Daher liegt der Fokus dieser Arbeit auf der Entwicklung einer Lösung, welche eine dynamische Berechnung globaler Beleuchtungseffekte auch auf leistungsschwächerer Hardware ermöglicht[1]. Hierbei müssen naturgemäß Abstriche bei der Qualität gemacht werden. Es soll allerdings versucht werden, diese soweit möglich zu minimieren und störende Artefakte, wie temporale Inkonsistenzen (Flackern) und offensichtliche Beleuchtungsfehler, zu vermeiden.

Als Basis des entwickelten Beleuchtungsverfahrens kommen eine Reihe von Bildraum-Verfahren[2] zur Anwendung. Diese weisen aufgrund der begrenzten Informationen[3] zwar gewisse visuelle Artefakte auf, sind allerdings i.d.R. weniger rechenaufwendig und vor allem unabhängig von der Größe und Komplexität der Szene.

Die Grundlage bildet dabei die Arbeit von Soler, Hoel und Rochet [SHR10] zur Echtzeitberechnung diffuser indirekter Beleuchtung. Dieses Verfahren wird im folgenden modifiziert um die inhärenten Defizite des verwendeten Screen-Space-Ansatzes zu kompensieren und um eine vollwertige physikalisch basierte bidirektionale Reflektanzverteilungsfunktion (BRDF) erweitert.

Verwandte Verfahren

Eines der ersten Beleuchtungsverfahren mit Unterstützung für globale Beleuchtungseffekte ist das von Kajiya [Kaj86] als Ansatz für eine numerische Lösung seiner Rendergleichung entwickelte Path-Tracing Verfahren. Bei diesem werden ausgehend von jedem Pixel des Ergebnisbildes eine größere Anzahl von Strahlen ausgesendet[4] um die Farbe dieses Punktes zu bestimmen und der Mittelwert aus

[1] Eine detaillierte Anleitung zur Installation und Verwendung der entwickelten Anwendung befindet sich in Anhang B.

[2] engl. Screen-Space-Algorithm. Algorithmen die ihre Ergebnisse anhand des Ergebnisses der Rasterisierung und nicht der eigentlichen Geometrie berechnen.

[3] z.B. keine Daten zu Objekten direkt hinter der Kamera.

[4] Auch Raytracing. Ein Strahl wird von einem Ausgangspunkt bis zum ersten Schnittpunkt mit der Geometrie oder einer festgelegten Maximallänge verfolgt.

diesen Samples bestimmt um das finale Ergebnis zu erhalten. Jeder dieser Strahlen folgt dabei der einfachen Regel: [Kaj86]

- Wenn nichts getroffen wurde ist die ermittelte Farbe schwarz.
- Ansonsten wird bis zu einer maximalen Rekursionstiefe ein weiterer Strahl vom getroffenen Punkt aus in eine zufällige Richtung auf der Hemisphäre ausgesendet, das Ergebnis mit der BRDF an dem Punkt moduliert und zur Emission am Punkt addiert.

Bei diesem Vorgehen handelt es sich somit um ein Monte-Carlo-Integrationsverfahren. Das heißt es gehört zu einer Klasse von zufallsbasieren Berechnungsverfahren, deren Ergebnisse mit einer gewissen (i.d.R. nach oben beschränkten) Wahrscheinlichkeit falsch sein können. Dieser inhärente Fehler wird in diesem Fall durch die Berechnung des Mittelwerts einer großen Anzahl von Samples praktisch stark reduziert. [Kaj86]

Diese simple Implementierung des Verfahrens benötigt allerdings (vor allem bei nicht diffusen Oberflächen) durch den hohen Fehler auch eine entsprechend hohe Anzahl an Samples. Um diese und den damit verbundenen Rechenaufwand zu reduzieren, wird dieses Verfahren daher oft mit Importance Sampling kombiniert. Hierbei werden die Richtungen der Strahlen durch eine quasi-zufälligen Sequenz bestimmt, deren Werte vermehrt in Wertebereichen liegen die zum Erfolgt führen. Doch auch mit diesen Verbesserungen sind realistische Ergebnisse durch Path-Tracing auf dem heutigen Stand der Technik nicht ohne weiteres in Echtzeit auf üblicher Endanwender-Hardware zu realisieren. [Laf95]

Ein weiteres Verfahren, welches häufig in Spielen eingesetzt wird und auch zur Beleuchtung dynamischer Objekte verwendet werden kann, sind Light-Probes. Hierbei wird die gesamte Lichteinstrahlung an vorher festgelegten, statischen Punkten bestimmt und in geeigneter Form gespeichert. Dies kann z.B. durch Rendern der Szene aus Sicht des Messpunktes in eine Cubemap erfolgen. Obwohl hierdurch theoretisch nur die Beleuchtung an den Messpunkten vorliegt, stellen sie i.d.R. auch eine hinreichend gute Approximation der Beleuchtung in ihrer unmittelbaren Umgebung dar. Durch eine ausreichende Menge an Messpunkten lässt sich somit die gesamte Beleuchtung einer Szene – inkl. dynamischer Objekte – erreichen. Allerdings ist die Berechnung der Lichteinstrahlung pro Messpunkt relativ aufwändig. Daher werden oft nur statische Objekte bei der Berechnung der Lichteinstrahlung an den Messpunkten beachtet und das Ergebnis möglichst vorberechnet. [SKS02]

Ein Verfahren das größere Ähnlichkeit mit dem hier vorgestellten Algorithmus aufweist ist das 2016 von Mara, McGuire, Nowrouzezahrai und Luebke [MMN+16] vorgestellte Deep G-Buffer basierte Verfahren, welches ebenfalls auf dem Verfahren von Soler, Hoel und Rochet [SHR10] aufbaut. Dieses Verfahren konzentriert sich allerdings verstärkt auf den Einsatz eines mehrschichtigen G-Buffers zur Kompensation der Screen-Space-Artefakte[5]. Das Verfahren von Mara, McGuire, Nowrouzezahrai und Luebke [MMN+16] weißt zwar ein grundsätzlich gutes Laufzeitver-

[5] z.B. Oberflächen die durch andere Objekte (teilweise) verdeckt sind.

halten[6] auf und liefert überzeugende Ergebnisse, baut allerdings zumindest bei der Darstellung der Sponza-Szene stark auf statischen Light-Probes auf, die in dynamischen Szenen zu zusätzlichem Rechenaufwand führen können. Das Verfahren ist also zumindest in der vorgestellten Form eher unterstützend zu etablierten Verfahren zu sehen.

Das letzte Verfahren, das hier kurz vorgestellt werden soll ist Voxel Cone Tracing. Bei diesem wird zuerst die gesamte zu beleuchtende Szene in ein volumetrisches Raster aus Voxeln zerlegt, welches anschließend mit der eingehenden Beleuchtung angereichert wird. Als Ergebnis erhält man eine volumetrische Datenstruktur in der jedem Voxel die Farbe der beleuchteten Oberflächen in seinem Inneren zugewiesen ist. Mittels Cone-Tracing[7] in diesem Datensatz lassen sich anschließend Informationen zur Umgebungsverdeckung, indirekter Beleuchtung sowie Reflexionen berechnen. Die Ergebnisse dieses Verfahrens sind i.d.R. qualitativ sehr hochwertig und lassen sich auf aktuellen Computern problemlos (auch für dynamische Szenen) in Echtzeit berechnen. Der Speicherbedarf der Datenstruktur stellt allerdings, trotz Techniken zur Abschwächung dieser Problematik[8], nach wie vor oft noch ein Problem dar. Auch sind umfangreiche Änderungen der Datenstruktur aufgrund von dynamischen Objekten (Revoxelisation) nicht immer in Echtzeit auf schwächerer Hardware möglich. Trotz der allgemein recht hohen Qualität ist das Ergebnis außerdem auch nicht frei von Artefakten. Diese treten meist in Folge der Quantisierung auf das Voxel-Gitter auf und äußern sich u.a. in Licht, das durch geschlossene Oberflächen/Kanten scheint (Light-Bleeding) und eine zu geringe Auflösung von Spiegelungen. [CNS+11]

[6] ≤ 7 ms für die Berechnung der indirekten Beleuchtung in 1080p auf einer NVIDIA GeForce GTX 980.
[7] Ähnlich zu Raytracing wobei der unendlich dünne Strahl durch einen Zylinder ersetzt wird. Dies wird oft durch mehrere Schnitttests mit einer Kugel umgesetzt, deren Radius mit zunehmender Entfernung vom Startpunkt wächst.
[8] Clipmaps, Sparse Voxel Octrees

2

Lösungsansatz

Um eine solide mathematische Grundlage für das weitere Vorgehen zu schaffen, erfolgt in einem ersten Schritt, analog zur Arbeit von Soler, Hoel und Rochet [SHR10], eine Herleitung der verwendeten Formeln über die Rendergleichung von Kajiya [Kaj86]. Mit dieser lässt sich approximativ die Beleuchtung an einem Punkt der Szene, unter Berücksichtigung der gesamten Umgebung beschreiben. [Kaj86]

$$L_0(\vec{x}, \vec{\omega_0}) = L_e(\vec{x}, \vec{\omega_0}) + \int_\Omega f_r(\vec{x}, \vec{\omega_i}, \vec{\omega_0}) \cdot L_i(\vec{x}, \vec{\omega_i}) \cdot (\vec{\omega_i} \bullet \vec{n}) d\vec{\omega_i} \tag{2.1}$$

Hierbei gibt $L_0(\vec{x}, \vec{\omega})$ an wie viel Licht vom Punkt $\vec{x}$ aus in die Richtung $\vec{\omega_0}$ ausgestrahlt wird (Strahlungsdichte). Dies setzt sich zusammen aus dem vom Punkt x selbst emittierten (L_e) und dem aus anderen Richtungen reflektiertem Licht. Das reflektierte Licht ergibt sich aus einem Integral über alle Richtungen der Hemisphere über der Oberfläche des zu beleuchtenden Punktes. In diesem beschreibt $f_r(\vec{x}, \vec{\omega_i}, \vec{\omega_0})$ die BRDF der Oberfläche, $L_i(\vec{x}, \vec{\omega_i})$ die aus der Richtung $\vec{\omega_i}$ an Punkt $\vec{x}$ ankommende Strahlungsdichte, $\vec{\omega_i}$ die aktuell betrachtete Richtung und $\vec{n}$ die Oberflächennormale an Punkt $\vec{x}$. [Kaj86]

Die Berücksichtigung der Eigenemission und der direkten Beleuchtung durch analytische Lichtquellen erfolgt losgelöst von der indirekten Beleuchtung in einem vorhergehenden Schritt. Bei der Berechnung der direkten Beleuchtung kommt – wie auch bei der indirekten – das Cook-Torrance-Beleuchtungsmodell[1] zum Einsatz. Hierbei handelt es sich um ein Mikrofacettenmodell, bei dem eine Oberfläche über eine große Anzahl mikroskopisch kleiner perfekter Spiegel (Facetten) modelliert wird. Die Berechnung (Gleichung 2.2) erfolgt dabei aufbauend auf den drei Hauptfaktoren F, D und G (Gleichungen 2.3, 2.4 und 2.5)[2]. Diese Faktoren beschreiben den Fresnel-Effekt (Reflexionsvermögen abhängig vom Betrachtungswinkel), die Wahrscheinlichkeitsverteilung der Mikrofacetten (Microfacet Distribution), sowie die Selbstabschattung (Geometrical Attenuation). Im Vergleich zu älteren verbreiteten Beleuchtungsmodellen wie dem Phong-Modell handelt sich sich beim Cook-Torrance-Modell um ein physikalisch plausibles Beleuchtungsmodell[3], wodurch im

[1] Auch als Torrance-Sparrow-Beleuchtungsmodell bezeichnet.
[2] Die Verwendung dieser Funktionen erfolgt aus Gründen der Lesbarkeit i.d.R. ohne Angabe der Parameter.
[3] Das heißt es ist positiv definit, reziprok und energieerhaltend.

Allgemeinen ein deutlich realistischeres Ergebnis entsteht, welches außerdem unabhängig von der Beleuchtung plausible bleibt. [CT82]

Nach Einsetzten der Cook-Torrance BRDF ergibt sich damit für die indirekte Beleuchtung die Gleichung: [CT82]

$$L_0(\vec{x},\vec{\omega_0}) = L_{\text{direkt}}(\vec{x},\vec{\omega_0}) + \int_\Omega \left(\frac{F\cdot D\cdot G}{4\cdot(\vec{n}\bullet\vec{\omega_i})\cdot(\vec{n}\bullet\vec{\omega_0})} + \frac{C\cdot(1-F)}{\pi}\right) \cdot L_i(\vec{x},\vec{\omega_i})\cdot(\vec{n}\bullet\vec{\omega_i})d\vec{\omega_i} \tag{2.2}$$

Hierbei bezeichnet C das Rückstrahlvermögen von diffusen Oberflächen[4] (d.h. ihre Farbe bei gleichmäßiger Beleuchtung mit weißem Licht), L_{direkt} das Ergebnis der direkten Beleuchtung und F, D und G jeweils den Fresnel, Microfacet Distribution und Geometrical Attenuation Faktor mit den folgenden Approximationen:

$$F(F0,\vec{h},\vec{\omega_0}) = F0 + (1-F0)\cdot(1-\vec{h}\bullet\vec{\omega_0})^5 \tag{2.3}$$

Mit dem Reflexionsvermögen bei Betrachtung senkrecht zur Ebene $F0$ und dem Halfway-Vector[5] $\vec{h}$.

$$D(\alpha,\vec{n},\vec{h}) = \frac{\alpha^4}{\pi\cdot((\vec{n}\bullet\vec{h})^2\cdot(\alpha^4-1)+1)^2} \tag{2.4}$$

Für die Oberflächenrauheit α.

$$G(\vec{n},\vec{\omega_0},\vec{\omega_i},\alpha) = \frac{\vec{n}\bullet\vec{\omega_0}}{(\vec{n}\bullet\vec{\omega_0})\cdot(1-k(\alpha))+k(\alpha)}\cdot\frac{\vec{n}\bullet\vec{\omega_i}}{(\vec{n}\bullet\vec{\omega_i})\cdot(1-k(\alpha))+k(\alpha)} \tag{2.5}$$

Mit $k = \frac{(\alpha+1)^2}{8}$.

Das größte Problem bei der Lösung dieser Gleichung stellt das Integral über alle Winkel der Hemisphäre dar, welches für jeden zu beleuchtenden Punkt gelöst werden muss. Selbst bei Verwendung eines Bildraumverfahrens müsste bei einem naiven Ansatz trotzdem noch für jedes Pixel jedes andere analysiert werden. Bei einer Auflösung von 1920×1080 müssten somit über $4\cdot10^{12}$ Pixel ausgelesen und analysiert werden.

Da es sich hierbei folglich um den begrenzenden Faktor des Verfahrens handelt, konzentrieren sich alle Teilschritte primär darauf, die Anzahl an betrachteten Punkten (Samples) zu reduzieren und das Abrufen der notwendigen Samples zu optimieren.

Abweichend von der Arbeit von Soler, Hoel und Rochet [SHR10] wird die diffuse und spekulare indirekte Beleuchtung getrennt voneinander behandelt. Dies eröffnet weitere Möglichkeiten zur Skalierung der Darstellungsqualität in Abhängigkeit von der verwendeten Hardware und ist vor allem aufgrund der sehr unterschiedlichen

[4] Ist 0 für rein spekulare Oberflächen wie Metalle.
[5] Der Vektor, der auf halbem Weg zwischen $\vec{\omega_0}$ und $\vec{\omega_i}$ liegt.

Anforderungen der beiden Phänomene sinnvoll. So weist die diffuse Beleuchtung vorwiegend niederfrequente Änderungen auf, während bei der spekularen Beleuchtung – gerade bei glatten Oberflächen mit nahezu perfekten Reflexionen – auch hohe Frequenzen auftreten können. Des Weiteren erfordert die Berechnung der diffusen Beleuchtung Lichtinformationen aus der gesamten Hemisphäre über der Oberfläche, während spekulare Reflexionen i.d.R. (d.h. bei perfekten und nahezu perfekten Spiegelungen) nur Informationen aus einem eingeschränkten Bereich benötigen und mit zunehmender Größe des relevanten Bereichs schnell an subjektiver Intensität verlieren. Durch diese Trennung entstehen die beiden Gleichungen

$$L_{\text{spec}}(\vec{x}, \vec{\omega_0}) = \int_{\Omega} \frac{F \cdot D \cdot G}{4 \cdot (\vec{n} \bullet \vec{\omega_i}) \cdot (\vec{n} \bullet \vec{\omega_0})} \cdot L_i(\vec{x}, \vec{\omega_i}) \cdot (\vec{n} \bullet \vec{\omega_i}) d\vec{\omega_i} \tag{2.6}$$

$$L_{\text{diffuse}}(\vec{x}, \vec{\omega_0}) = \int_{\Omega} L_i(\vec{x}, \vec{\omega_i}) \cdot (\vec{n} \bullet \vec{\omega_i}) d\vec{\omega_i} \tag{2.7}$$

welche in einem abschließenden Schritt kombiniert werden zur Gleichung der gesamten Beleuchtung:

$$L_0(\vec{x}, \vec{\omega_0}) = L_{\text{direkt}}(\vec{x}, \vec{\omega_0}) + L_{\text{spec}}(\vec{x}, \vec{\omega_0}) + L_{\text{diffuse}}(\vec{x}, \vec{\omega_0}) \cdot \frac{C \cdot (1 - F)}{\pi} \tag{2.8}$$

Darüber hinaus wird es durch dieses Vorgehen möglich die Berechnung der indirekten Beleuchtung in niedrigerer Auflösung durchzuführen ohne dabei Details aus den hochfrequenten Farbinformationen der Oberfläche (C und $F0$) zu verlieren.

2.1 Diffuse indirekte Beleuchtung

Bei dem Verfahren von Soler, Hoel und Rochet [SHR10] – welches die Basis für die Berechnung der diffusen indirekten Beleuchtung bildet – handelt es sich um einen Gathering-Ansatz, d.h. für jedes Pixel wird eine Menge von Pixeln aus der Eingabe analysiert um den Ausgabewert zu berechnen. Sowohl die Ein- als auch die Ausgabe des Algorithmus ist eine 2D RGB-Textur, welche die aktuelle Beleuchtung[6] beschreibt. Im Falle der Eingabe werden darüber hinaus Informationen zum sichtbaren Teil der Szene (Normalen und Tiefeninformationen/Positionen) benötigt. [SHR10]

Da der Algorithmus auf Samples im Screen-Space arbeitet, muss die Gleichung 2.7 in Form einer begrenzten Anzahl von Oberflächen in der Szene ausgedrückt werden. Dies erfolgt über die Einführung eines Terms v welcher die Sichtbarkeit zwischen zwei Punkten angibt sowie einer Diskretisierung über eine Summe von N Flächen der Größe ds innerhalb der Szene. Des Weiteren kommt hierbei ein auf den positiven Wertebereich beschränkter Kosinus $\cos_+$ zum Einsatz. [SHR10]

$$L_{\text{diffuse}}(\vec{x}, \vec{\omega_0}) \approx \sum_{i=1}^{N} L_i(\vec{x}, \vec{\omega_i}) \cdot \frac{\cos_+ \theta_i \cdot \cos_+ \theta_i'}{||\vec{x} - \vec{x_i}||^2} \cdot v(\vec{x}, \vec{x_i}) ds \tag{2.9}$$

[6] Zum Beispiel das Ergebnis der direkten Beleuchtung.

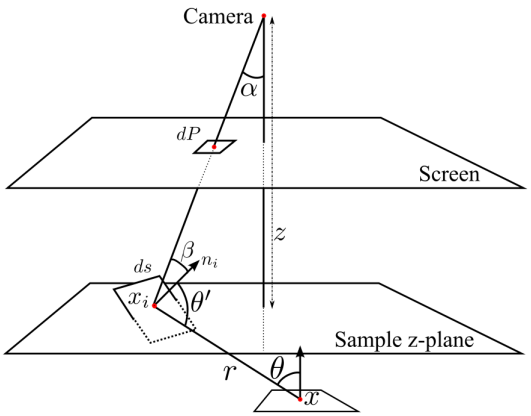
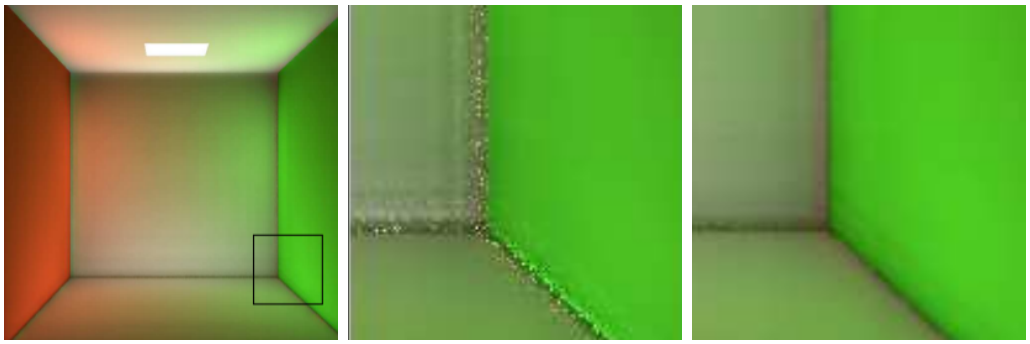


Abb. 2.1: Darstellung des zur Berechnung der Beleuchtung am Punkt $\vec{x}$ verwendeten View-Space-Koordinatensystems und der Variablen.
Dabei ist $\vec{x}_i$ der aktuell betrachtete Punkt, ds der Differentialbereich im View-Space (in Metern) und dP der Differentialbereich im Screen-Space (in normalisierten Gerätekoordinaten). [SHR10]

Die Größe des Differentialbereichs ds wird entsprechend Abb. 2.1 aus der Größe des Bereichs im Screen-Space und den Parametern der Kamera berechnet: [SHR10]

$$ds = \frac{z^2 \cdot 4 \cdot \tan\frac{f_h}{2} \cdot \tan\frac{f_v}{2} \cdot dP}{W \cdot H} \cdot \frac{\cos\alpha}{\cos\beta} \tag{2.10}$$

Hierbei steht z für die Z-Koordinate (im View-Space) des betrachteten Punktes $\vec{x}_i$, f_h/f_v für das horizontale bzw. vertikale Sichtfeld[7] und W/H für die Breite/Höhe des Bildschirms in Pixeln. Der linke Quotient bestimmt die Größe einer Fläche von dP Pixeln mit einer Tiefe von z, sofern diese senkrecht zur Blickrichtung liegt und der rechte Quotient korrigiert diesen Wert für davon abweichende Neigungswinkel. [SHR10]



(a) Nur indirekte Beleuchtung (b) Mit Gleichung 2.9 (c) Mit Gleichung 2.11

Abb. 2.2: Unter Verwendung der Gleichung 2.9 kann es zu Artefakten (b) kommen, welche durch die Approximation in Gleichung 2.11 minimiert werden können (c). [SHR10]

In Gleichung 2.9 liegt aufgrund der Division durch den Abstand der Punkte ($\frac{\cos_+\theta_i \cdot \cos_+\theta_i'}{\|\vec{x}-\vec{x}_i\|^2}$) eine Definitionslücke vor. Diese führt zu Instabilität, welches sich

[7] Field of View (FOV)

in Bildrauschen ausdrückt, wenn sich die beiden Punkte stark annähern (siehe Abb. 2.2). Die Lösung aus der Arbeit von Soler, Hoel und Rochet [SHR10] ist es den problematischen Teil der Gleichung durch die Approximation in Gleichung 2.11 zu ersetzen. In dieser Implementierung wurde allerdings abweichend davon die einfachere Approximation in Gleichung 2.12 eingesetzt, da diese im vorliegenden Fall zu weniger Artefakte und einem subjektiv besseren Gesamtergebnis führt. [SHR10]

$$\frac{\cos_+\theta_i \cdot \cos_+\theta_i' \cdot ds}{\|\vec{x}-\vec{x}_i\|^2} \approx \frac{R^2 \cdot \cos_+\theta_i}{\|\vec{x}-\vec{x}_i\|^2 + R^2} \quad \text{mit} \quad R^2 = \frac{\cos_+\theta_i' \cdot ds}{\pi} \tag{2.11}$$

$$\frac{\cos_+\theta_i \cdot \cos_+\theta_i' \cdot ds}{\|\vec{x}-\vec{x}_i\|^2} \approx \frac{\cos_+\theta_i \cdot \cos_+\theta_i' \cdot ds}{\|\vec{x}-\vec{x}_i\|^2 + 0{,}1} \tag{2.12}$$

Die Berechnung des Sichtbarkeitsterms $v(\vec{x},\vec{x}_i)$ erfordert einen Raycast zwischen den beiden Punkten und gehört somit zu den aufwändigsten Teilen der Gleichung. Soler, Hoel und Rochet [SHR10] schlagen zur Lösung des Terms eine Voxelisierung der sichtbaren Szene nach Eisemann und Décoret [ED08] und anschließenden Raycast des Voxelrasters vor [SHR10]. Dazu wird allerdings ein weiterer Renderpass über die Szene und eine nicht unerhebliche Speichermenge[8] benötigt. Da der Algorithmus primär durch die Speicherbandbreite begrenzt wird, schlägt sich dies auch in den von Soler, Hoel und Rochet [SHR10] festgestellten Renderzeiten nieder. Da darüber hinaus zwar direkte größere Schatten essentiell für die subjektive Qualität sind, die Bedeutung kleinerer indirekter Schatten allerdings deutlich geringer ist, scheint eine Berechnung dieses Terms zum jetzigen Zeitpunkt wenig attraktiv [YCK+09]. Bei Szenen mit hoher Komplexität (siehe Abb. 2.3) kommt es durch den Verzicht auf diesen Term allerdings zu nicht unerheblichen Fehlern. In derartigen Fällen wäre eine denkbare Lösung den Raycast nur auf weiter entfernte Punkte zu beschränken oder eine alternative, abstraktere Repräsentation der Szene zu definieren[9] und die Sichtbarkeit über diese prüfen.

Unter Anwendung der oben genannten Gleichungen und Approximationen ergibt sich somit die folgende Gleichung für L_{diffuse}.

$$\begin{aligned} L_{\text{diffuse}}(\vec{x},\vec{\omega}_0) \approx \sum_{i=1}^{N} L_i(\vec{x},\vec{\omega}_i) \cdot \frac{\cos_+\theta_i \cdot \cos_+\theta_i'}{\|\vec{x}-\vec{x}_i\|^2 + 0{,}1} \cdot v(\vec{x},\vec{x}_i) \\ \cdot \frac{z^2 \cdot 4 \cdot \tan\frac{f_h}{2} \cdot \tan\frac{f_v}{2} \cdot dP}{W \cdot H} \cdot \frac{\cos\alpha}{\cos\beta} \end{aligned} \tag{2.13}$$

Abschließend ist die Aufgabe der hier definierten Funktion L_{diffuse} im Gesamtkontext, für jeden Pixel des Eingangsbildes die eingehende Bestrahlungsstärke (Irradiance), durch Betrachtung der anderen Pixel im Eingangsbild zu bestimmen. Um das Sammeln der relevanten Pixel zu beschleunigen setzt das Verfahren auf

[8] Vorgeschlagen wurden vier zusätzliche Texturen mit je 32 Bit, also bei 1920×1080 zusätzlich ca. 32 MiB.

[9] Zum Beispiel eine Menge von Axis Aligned Bounding Boxes (AABBs), welche den Lichttransfer einschränken.

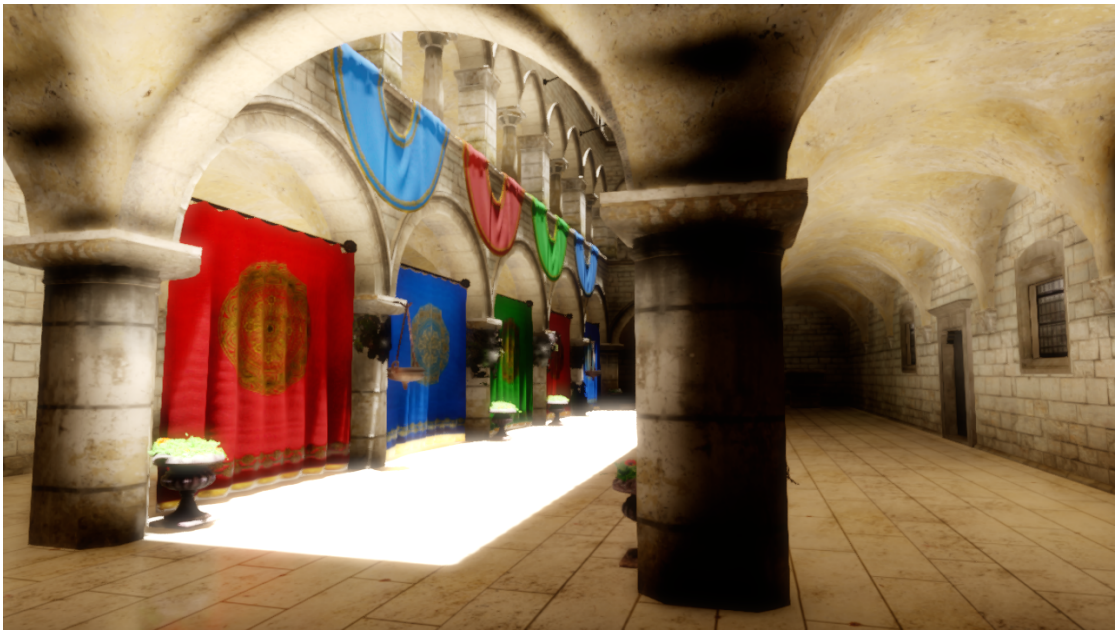


Abb. 2.3: Durch den Verzicht auf die Berechnung der tatsächlichen Sichtbarkeit zwischen den einzelnen Punkten kann es zu einem beträchtlichen Fehler durch Licht, das durch andere Objekte scheint, kommen. In diesem Fall wird die rechte Seite zu stark durch Licht von der linken Seite beleuchtet. Da dies nur auftritt wenn zwei getrennte Teile der Szene mit sehr unterschiedlicher Beleuchtung gleichzeitig sichtbar sind, wird diesem Fehler keine hohe Priorität beigemessen.

einen hierarchischen Ansatz. Dazu wird eine Bildpyramide (MIP-Map)[10] verwendet und jeweils nur Samples aus einem begrenzten Umkreis bezogen um die Cache-Lokalität[11] zu waren. Auf weitere Details der konkreten Implementierung sowie die Integration in das Gesamtsystem wird in Kapitel 3.6.1 genauer eingegangen.

Da die Ein- und Ausgabe des Verfahrens das selbe Format aufweisen (ein 2D RGB Bild der beleuchteten Szene), ist es ohne großen zusätzlichen Aufwand möglich eine unbegrenzte Anzahl von Reflexionsschritten der Beleuchtung (Bounces) zu berechnen, indem die Ausgabe vom letzten Frame mit der Eingabe des aktuellen Frames kombiniert wird. Die Details dieses Kombinationsschittes sowie die dafür nötigen zusätzlichen Anpassungen werden in Kapitel 3.6 genau behandelt.

[10] Eine Reihe von skalierten Kopien des Ausgangsbildes, wobei die Kantenlänge jedes Bildes halb so groß ist wie die seines Vorgängers.

[11] Zugriffe werden soweit möglich auf Texel beschränkt, die in der Textur nahe beieinander liegen. Dadurch wird die Wahrscheinlichkeit gesteigert, dass diese sich bereits im Cache der GPU statt nur im vergleichsweise langsamen VRAM befinden.

2.2 Spekulare indirekte Beleuchtung

Bei der Berechnung der spekularen indirekten Beleuchtung (Gleichung 2.6) liegt eine ähnliche Situation wie bei der diffusen Beleuchtung vor. So ist auch hier die Hauptaufgabe der Funktion das Sammeln der eingehenden Bestrahlungsstärke aus der Umgebung. Abweichend davon liegen hier allerdings vornehmlich hochfrequente Signale (perfekte oder annähernd perfekte Spiegelungen) vor und die berücksichtigte Bestrahlung über die Hemisphäre ist nicht konstant, sondern bestimmt sich aus dem Betrachtungswinkel und den Materialeigenschaften (Form und Größe der spekularen Keule). Daher erfolgt die Berechnung, wie zu Beginn des Kapitels erläutert, getrennt und verwendet andere Approximationen. Da es bei diesem Teil der Gleichung gewisse Parallelen dazu gibt, bedienen sich einige der angeführten Vorgehensweisen an Verfahren aus dem Bereich des Image-Based-Lightings (IBL).

Analog zu L_{diffuse} liegt der Hauptaufwand auch bei L_{spec} in der Berechnung des Integrals für alle Richtungen der Hemisphäre. In einem ersten Schritt wird das Integral daher ebenfalls mit einer Summe über N-Punkten approximiert.

$$L_{\text{spec}}(\vec{x}, \vec{\omega_0}) \approx \frac{1}{N} \sum_{i=1}^{N} \frac{F \cdot D \cdot G}{4 \cdot (\vec{n} \bullet \vec{\omega_i}) \cdot (\vec{n} \bullet \vec{\omega_0})} \cdot L_i(\vec{x}, \vec{\omega_i}) \cdot (\vec{n} \bullet \vec{\omega_i}) \tag{2.14}$$

Um eine ausreichende Qualität zu gewährleisten sind allerdings selbst bei entsprechend intelligenter Wahl der Punkte[12], noch eine hohe Zahl an Samples (> 16) notwendig [Kar13]. Erschwerend kommt hier außerdem noch hinzu, dass für jede Richtung die vollständige BRDF gelöst werden muss. Daher wird diese Summe durch Aufteilung in ein Produkt von zwei Summen weiter approximiert, welche anschließend separat gelöst werden können. Diese Approximation ist zwar nur für ein konstantes L_i[13] korrekt, allerdings für die meisten in der Praxis auftretenden Fällen hinreichend genau. [Kar13]

$$L_{\text{spec}}(\vec{x}, \vec{\omega_0}) \approx \left(\frac{1}{N} \sum_{i=1}^{N} L_i(\vec{x}, \vec{\omega_i}) \right) \left(\frac{1}{N} \sum_{i=1}^{N} \frac{F \cdot D \cdot G}{4 \cdot (\vec{n} \bullet \vec{\omega_i}) \cdot (\vec{n} \bullet \vec{\omega_0})} \cdot (\vec{n} \bullet \vec{\omega_i}) \right) \tag{2.15}$$

$$L_{\text{spec light}}(\vec{x}, \vec{\omega_0}) = \left(\frac{1}{N} \sum_{i=1}^{N} L_i(\vec{x}, \vec{\omega_i}) \right) \tag{2.16}$$

$$L_{\text{spec brdf}}(\vec{x}, \vec{\omega_0}) = \left(\frac{1}{N} \sum_{i=1}^{N} \frac{F \cdot D \cdot G}{4 \cdot (\vec{n} \bullet \vec{\omega_i}) \cdot (\vec{n} \bullet \vec{\omega_0})} \cdot (\vec{n} \bullet \vec{\omega_i}) \right) \tag{2.17}$$

Der entscheidende Vorteil an diesem Vorgehen ist, dass die Summe 2.17 nun unabhängig von der Umgebung L_i ist. Dies erlaubt es die gesamte rechte Summe in die Form aus Gleichung 2.18 zu bringen, vorzuberechnen und erst im letzten Schritt mit dem Ergebnis der linken Gleichung und dem $F0$-Werte der Oberfläche gemäß Gleichung 2.19 zu verrechnen. Hierdurch wird nicht nur der Rechenaufwand zum Lösen der Funktion L_{spec} minimiert, sondern es bleiben auch mehr Details des

[12] Importance Sampling
[13] Das heißt eine gleichmäßig weiße Umgebung.

mitunter relativ hochfrequenten $F0$-Wertes (Oberflächenfarbe bei Metallen) und der Normale $\vec{n}$ erhalten. [Kar13]

Als Ergebnis der genannten Umformungen und Approximationen erhalten wir die Funktion f_{brdf} über der Rauheit α, dem Betrachtungsvektor $\vec{\omega_0}$ und der Oberflächennormale $\vec{n}$, welche als Ergebnis den Wert der BRDF sowie einen Skalierungsfaktor für den Fresnel-Faktor liefert[14].

$$f_{\text{brdf}}(\alpha, \vec{\omega_0}, \vec{n}) = \begin{pmatrix} \int_\Omega \frac{D \cdot G}{4 \cdot (\vec{n} \bullet \vec{\omega_i}) \cdot (\vec{n} \bullet \vec{\omega_0})} \cdot (1 - \vec{\omega_0} \bullet \vec{h})^5 \cdot (\vec{n} \bullet \vec{\omega_i}) d\vec{\omega_i} \\ \int_\Omega \frac{D \cdot G}{4 \cdot (\vec{n} \bullet \vec{\omega_i}) \cdot (\vec{n} \bullet \vec{\omega_0})} \cdot (1 - (1 - \vec{\omega_0} \bullet \vec{h})^5) \cdot (\vec{n} \bullet \vec{\omega_i}) d\vec{\omega_i} \end{pmatrix} \tag{2.18}$$

Diese Funktion kann anschließend mittels der folgenden Gleichung mit L_{diffuse} (Gleichung 2.13) und dem linken Teil der Summe aus Gleichung 2.15 zur gesamten indirekten Beleuchtung am Punkt $\vec{x}$ kombiniert werden.

$$\begin{aligned} L_0(\vec{x}, \vec{\omega_0}) &= L_d(\vec{x}, \vec{\omega_0}) + L_{\text{indirekt}}(\vec{x}, \vec{\omega_0}) \\ L_{\text{indirekt}}(\vec{x}, \vec{\omega_0}) &= \frac{C \cdot L_{\text{diffuse}}(\vec{x}, \vec{\omega_0}) \cdot (1 - F0 \cdot f_{\text{brdf}}(\alpha, \vec{\omega_0}, \vec{n})_1)}{\pi} \\ &\quad + L_{\text{spec light}} \cdot (F0 \cdot f_{brdf}(\alpha, \vec{\omega_0}, \vec{n})_1 + f_{\text{brdf}}(\alpha, \vec{\omega_0}, \vec{n})_0) \end{aligned} \tag{2.19}$$

Zur Berechnung der Summe in $L_{\text{spec light}}$ (Gleichung 2.16) werden Raycast auf dem Tiefenpuffer[15] im Screen-Space durchgeführt und alle verfügbaren Pixel im relevanten Umkreis (abhängig von der Rauheit und Distanz) des ersten Treffers ermittelt (siehe Kapitel 3.6.3).

[14] Diese und die folgenden Gleichungen kombinieren die bisherigen Gleichungen mit den Funktionen 2.3, 2.4 und 2.5 aus Kapitel 2.

[15] Textur mit dem Abstand des Pixels zur Kamera für jedes Pixel.

2.3 Scalable Ambient Obscurance

Durch den Verzicht auf die Bestimmung der Sichtbarkeit zwischen den Punkten bei der Berechnung der diffusen indirekten Beleuchtung kommt es zu einer – abhängig von der Tiefenkomplexität der Szene relativ stark ausgeprägten – Überschätzung der eingehenden Beleuchtung. Diese wird unter Umständen noch zusätzlich dadurch verstärkt, dass die Berechnung der diffusen Beleuchtung in einer niedrigeren Auflösung durchgeführt wird, was zu einem geringfügigen Weichzeichnen führt.

Um diesem Effekt entgegenzuwirken wird das Ergebnis der indirekten Beleuchtung mit einem Umgebungsverdeckungsterm[16] moduliert. Dieser wird basierend auf dem "Scalable Ambient Obscurance"-Verfahren (SAO) von McGuire, Mara und Luebke [MML12] berechnet. Als Eingabe dient der aktuelle Tiefenpuffer, sowie (abweichend vom original Verfahren) die Oberflächennormalen. Da ähnlich wie bei der diffusen Beleuchtung ein hierarchischer Ansatz zur Anwendung kommt, müssen diese beiden Texturen ebenfalls mit MIP-Maps ausgestattet sein. Das Ergebnis des Algorithmus ist eine Textur mit Werten im Intervall (0,1), welche die Sichtbarkeit der Umgebung vom Punkt aus angeben (1 = gesamte Hemisphäre sichtbar). [MML12]

Zur Berechnung der Sichtbarkeit werden N (= 16) Samples $\vec{s}_i$ an festgelegten Positionen um den aktuellen Punkt c ausgewertet. Die Samples liegen dabei auf einer Spirale um eine möglichst gute Abdeckung bei gleichzeitig optimaler Cache-Lokalität zu erreichen und sind definiert durch: [MML12]

$$\begin{aligned} \text{Sei } \alpha_i &= \frac{i + 0{,}5}{N} \\ \theta_i &= 2\Pi\alpha_i\tau + \phi \\ \vec{s}_i &= \vec{c} + r'\alpha_i \cdot \begin{pmatrix} \cos\theta_i \\ \sin\theta_i \end{pmatrix} \end{aligned} \tag{2.20}$$

Der Parameter r' bestimmt sich dabei durch den gewünschten Radius der Umgebungsverdeckung r projiziert in den Screen-Space. Diese Projektion kann z.B. über $r' = -rS'/c_z$ aus dem Radius im World-Space, der Z-Position des aktuellen Punktes $\vec{c}$ und der Konstanten S' – welche die Pixelgröße eines 1m großen Objektes von 1m Entfernung aus angibt – berechnet werden. [MML12]

Über die Konstante τ kann die Anzahl von Umdrehungen der Spirale gesteuert werden. Der optimale Wert, welcher eine möglichst gleichmäßige Verteilung der Samples sicherstellt, hängt von der Anzahl der Samples ab und kann aus verschiedenen Quellen bezogen werden. [MMN+16]

Die letzte Variable in der Gleichung 2.20 ϕ ist ein Rotationsoffset, der für jedes Pixel zufällig gewählt werden sollte[17] um das Auftreten von sich wiederholenden Mustern zu vermeiden. [MML12]

Für jedes dieser Samples wird sowohl die Normale $\vec{n}_i$ als auch der Tiefenwert ausgelesen und die View-Space Position $\vec{s}_i$ aus diesem rekonstruiert. Die entspre-

[16] Engl. Ambient Occlusion oder Ambient Obscurance.
[17] Zum Beispiel mittels Hashing der aktuellen Bildschirmkoordinaten.

chenden Texturen werden dabei auf dem MIP-Level $m_i = \lfloor \log_2(r' \cdot \alpha_i / q') \rfloor$ mit $q' = 2^4$ ausgelesen. [MML12]

Basierend auf diesen Daten der Samples, berechnet sich die Sichtbarkeit $A(\vec{c})$ an jedem Punkt nach der Gleichung: [MML12]

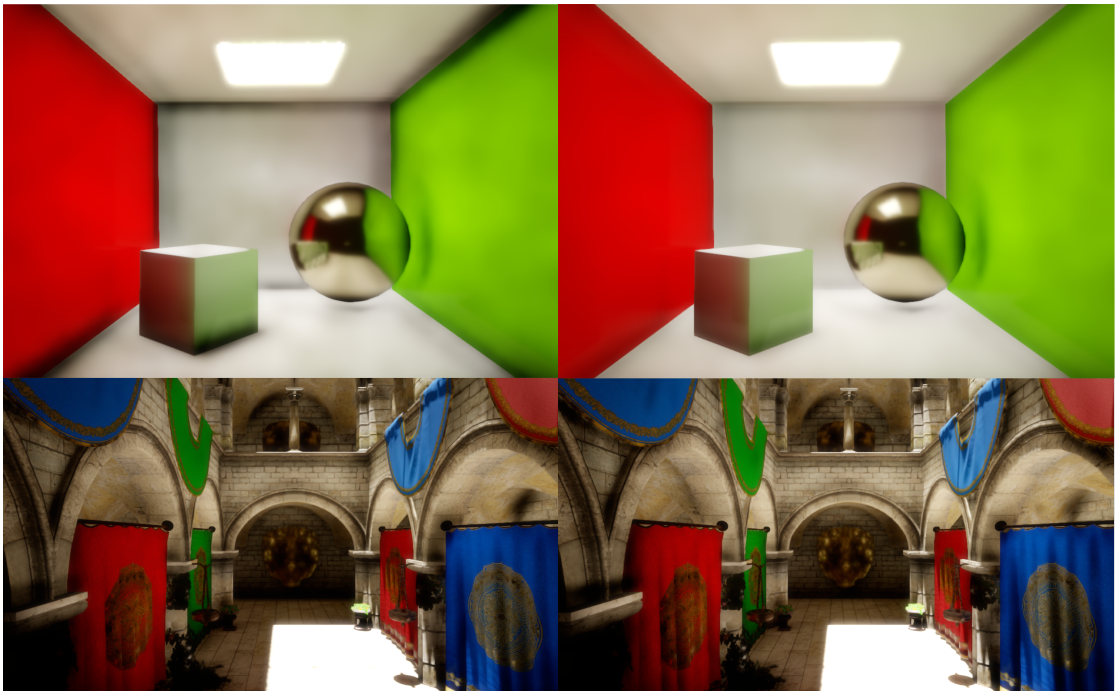
$$
\begin{aligned}
A(\vec{c}) &= max\left(0; 1 - \frac{5}{r^6 N}\sum_{i=0}^{N} max(r^2 - \vec{v} \bullet \vec{v}; 0)^3 \cdot max(\frac{\vec{v} \bullet \vec{n} - \beta}{\vec{v} \bullet \vec{v} + \epsilon}; 0) \cdot \sigma\right)^k \\
&\text{mit} \\
&\vec{v} = \vec{s} - \vec{c} \\
&\sigma = S_{\text{ss}}(0{,}01; 0{,}2; \|\vec{n} \bullet \vec{n_i}\|) \cdot 0{,}6 + 0{,}4 + S_{\text{ss}}(0{,}8; 1; \|\vec{n} \bullet \vec{n_i}\|) \cdot 2
\end{aligned}
\tag{2.21}
$$

Hierbei beschreibt $S_{\text{ss}}(l,r,x)$ eine smoothstep-Funktion, welche x auf das Intervall (l,r) beschränkt und zwischen diesen beiden Werten eine flüssige Interpolation mit einer Steigung von 0 für $x = l$ und $x = r$ gewährleistet und definiert ist als:

$$
\begin{aligned}
S_{\text{ss}}(l, r, x) &= S_1((x - l)/(r - l)) \\
S_1(x) &= \begin{cases} 0 & x \le 0 \\ 3x^2 - 2x^3 & 0 \le x \le 1 \\ 1 & 1 \le x \end{cases}
\end{aligned}
\tag{2.22}
$$

Die Variation der Konstanten β, k und ϵ kann neben der von r dazu verwendet werden die Größe, Intensität sowie den Kontrast der Umgebungsverdeckung an spezifische Bedürfnisse der Szene anzupassen. Während der Analyse haben sich bei allen getesteten Szenen die Werte $\beta = 0{,}04$, $k = 2$, $\epsilon = 0{,}01$ und $r = 1{,}8$ als befriedigend erwiesen. [MML12]

Aufgrund der besonderen Anforderungen an die Umgebungsverdeckung waren einige Modifikationen der von McGuire, Mara und Luebke [MML12] vorgestellten Vorgehensweise notwendig. Diese dienen vor allem dazu den Kontrast in nahezu vollständig verdeckten Bereichen (z.B. hinter den Bannern) zu erhöhen, ohne das gleichzeitig die Ecken zu dunkel werden (siehe Abb. 2.4). Diese Änderungen sind neben einem Exponent $k \neq 1$ und entsprechend gewählten Konstanten β, ϵ und r der zusätzliche Skalierungsfaktor σ. Um diesen Faktor zu berechnen, wird der Winkel zwischen der Normalen an den Punkten c und s_i bestimmt. Mittels dieses Winkels können anschließend die beiden Extremsituationen (ca. 90°/270° und ca. 0°/360° Winkel) unterschieden und ein der Situation angemessener Faktor gewählt werden, der die Intensität der Verdeckung wie gewünscht moduliert. Außerdem wurde, abweichend vom empfohlenen Vorgehen, die Normale n nicht über den Differenzenquotient aus dem Tiefenpuffer berechnet, sondern die Fragment-Normale aus dem Ergebnis des Normal-Mapping verwendet. Dies führt zwar zu geringfügigen temporalen Inkonsistenzen, berücksichtigt dafür allerdings auch Details, die nicht Teil der Geometrie sind. Was vor allem aufgrund der fehlenden Ambient-Occlusion-Maps bei den verwendeten Modellen für zusätzliche Details sorgt.



(a) (b)

Abb. 2.4: Vergleich der ursprünglichen (a) und modifizierten (b) Umgebungsverdeckung

Die Berechnung der Umgebungsverdeckung erfolgt nur in einer geringeren Auflösung[18]. Hierdurch sowie durch die geringe Anzahl an Samples, enthält die so berechnete Lösung einige Artefakte. Daher wird aus dieser anschließend durch einen bilateralen gaußschen Weichzeichner eine stückweise glatte Lösung rekonstruiert. Bei diesem Schritt werden außerdem zusätzliche Gewichte aus den Tiefenwerten berechnet um den Weichzeichner auf zusammenhängende Flächen zu beschränken und nicht über Objektkanten hinweg zu arbeiten. Um diesen Schritt zusätzlich zu beschleunigen, wird bei der Weichzeichnung darüber hinaus von der linearen Texturinterpolation der Hardware Gebrauch gemacht. Hierdurch kommt es zwar zu einer geringfügigen Glättung über Kanten hinweg, dieser Effekt fällt allerdings in der Praxis kaum ins Gewicht und ist in Anbetracht der dadurch verbesserten Geschwindigkeit zu vernachlässigen. [MML12]

[18] In dieser Implementierung standardmäßig bei der halben Auflösung des Eingabebildes.

3

Implementierung

Zur Bewertung des in Kapitel 2 vorgestellten Beleuchtungsverfahrens wurde im Rahmen dieser Arbeit ein 3D-Renderer als Basis entwickelt und anschließend um das beschriebene Verfahren ergänzt. Hierbei wurden einige Optimierungen – sowohl der Geschwindigkeit als auch der Qualität – implementiert sowie zahlreiche weitere Designentscheidungen getroffen, welche im folgenden Kapitel genauer vorgestellt und analysiert werden.

Der Fokus der Implementierung liegt dabei klar auf Umsetzung und Test des entwickelten Beleuchtungsverfahrens. Es wurde zwar trotzdem Wert auf eine saubere und erweiterbare Struktur gelegt, auf eine umfangreiche Fehlerbehandlung (z.B. bei nicht unterstützen Textur-Formaten und GPU-Features) wurde allerdings vorerst verzichtet. Auf die Unterstützung von mehreren Threads wurde ebenfalls noch kein Wert gelegt, da zumindest in der aktuellen Ausprägung durch die vergleichsweise geringe Belastung der CPU keine Verbesserung der Performance zu erwarten war.

Die Implementierung erfolgte unter Verwendung der Programmiersprache C++17 und der Grafikschnittstelle Vulkan. Abgesehen von grundlegenden Basiskomponenten[1], welche mit einigen Modifikationen aus früheren Projekten übernommen wurden, erfolgte die gesamte Entwicklung innerhalb der Bearbeitungszeit des Projekts und baut auf den folgenden externen Bibliotheken auf:

- **Open Asset Import Library (assimp)**: Eine Bibliothek zum Laden, Speichern und Vorverarbeiten von 3D-Modellen, die dazu benötigt wird Modelle in unterschiedlichen Formaten zu laden und in ein effizientes enginespezifisches Format zu konvertieren.
- **OpenGL Mathematics (GLM)**: Eine Sammlung von Mathematik Klassen (Vektoren, Matrizen, ...) und Funktionen, welche speziell auf die Verwendung mit OpenGL und Vulkan ausgelegt sind.
- **Guidelines Support Library (GSL)**: Eine Erweiterung der C++-Standardbibliothek um Typen und Funktionen aus den C++-Core Guidelines.
- **moodycamel::ConcurrentQueue**: Eine in C++11 geschriebene Lock-Free multi-producer multi-consumer Queue, welche u.a. beim Verwalten von Events

[1] Erweiterungen der Standardbibliothek (util-Komponente); Dateiverwaltung (asset-Komponente); Input-System (input-Komponente); grundlegende Projektstruktur und Verwaltungsklassen (u.a. core-Komponente).

durch Benutzereingaben und im Entity-Component-System (ECS) zum Einsatz kommt.
- **Nuklear**: Eine Immediate-Mode GUI-Bibliothek, mit der die Debug-Ausgaben und Steuerungselemente realisiert wurden.
- **PhysicsFS**: Eine Abstraktionsschicht zum vereinfachten strukturierten Zugriff auf das Dateisystem und diverse Containerformate, welche die Grundlage für das Asset-System bildet.
- **Simple DirectMedia Layer (SDL)**: Eine Bibliothek zum plattformunabhängigen Zugriff auf Eingabegeräte, Audio- und Grafik-Hardware, die derzeit vor allem für die Eingabebehandlung sowie das Öffnen von Fenstern verwendet wird.
- **SF2**: Eine Bibliothek zum Laden und Speichern von Datenstrukturen (struct/class) im JSON-Format. Diese baut auf Makros und Templates auf um mit einer möglichst geringen Menge an zusätzlichen Annotationen nahezu beliebige Datenstrukturen zu de-/serialisieren. Sie kommt u.a. im Assets-System und dem ECS zu Einsatz.
- **stb_image**: Eine Bibliothek zum Laden von Bildern im JPG, PNG, TGA, BMP, PSD, GIF, HDR und PIC-Format während der Vorverarbeitung von Modellen.

3.1 Architektur

Das Projekt ist auf der obersten Ebene unterteilt in drei Unterprojekte: Die Beispiel-Anwendung (`src/demo`), eine Konsolenanwendung mit der Modelle in das intern genutzte Format konvertiert werden können (`src/mesh_converter`), sowie die eigentliche Engine (`src/mirrage`). Die Beispielanwendung und der Converter sind beide relativ simpel aufgebaut. Im Gegensatz dazu unterteilt sich die Engine noch weiter in die Komponenten:

- **asset**: Der Asset-Manager, welcher einen strukturierten und vereinfachten Zugriff auf das Dateisystem ermöglicht. Unter anderem besteht hierüber die Möglichkeit beliebige über SF2 annotierte Klassen zu laden und zu speichern. Der Zugriff auf bestimmte Dateien erfolgt hierbei über Asset-IDs (AIDs), bestehend aus einem Typ und einem Namen, welche optional über `*.map` Dateien auf bestimmte Pfade abgebildet werden können. Des Weiteren wird der Dateisystemzugriff hierbei bewusst auf die in der `archives.lst` (im aktuellen Arbeitsverzeichnis) genannten Verzeichnisse und einen separaten Ordner für Schreibzugriffe beschränkt.
- **core**: Die Hauptkomponente der Engine über die alle globalen Subsysteme (Asset-Manager, Audio, Input, etc.) initialisiert und zugänglich gemacht werden, sowie ein Zustandsautomat zum Verwalten der unterschiedlichen Anzeigen/Menüs/Spielmodi.
- **ecs**: Das Entity-Component-System, mit dem die Objekte innerhalb einer Spielwelt verwaltet werden.

- **graphic**: Eine dünne Abstraktionsschicht über der Vulkan-API und einige häufig benötigte Klassen wie Texturen und Meshes, die auf dieser aufbauen.
- **gui**: Übernimmt die Initialisierung von Nuklear, sowie die Integration mit der Input-/ Grafikschnittstelle und stellt zusätzliche GUI-Elemente zur Verfügung.
- **input**: Eine Abstraktionsschicht über der Eingabebehandlung mit SDL, die einen saubereren Zugriff auf Benutzereingaben ermöglicht, diese in Anwendungsspezifische Ereignisse übersetzt und sie über den asynchrone Message-Bus an andere Systeme weiterleitet.
- **renderer**: Die Implementierung des 3D-Renderers, aufbauend auf der u.a. durch die graphic-Komponente bereitgestellten Abstraktionen.
- **utils**: Eine Sammlung hilfreicher Datenstrukturen und Funktionen, die von den anderen Komponenten benötigt werden.

3.2 Vulkan Integration

Bei Vulkan handelt es sich um eine von der Khronos Group entwickelte neue Grafik-Schnittstelle für die plattformübergreifende Entwicklung von grafischen Anwendungen. Im Gegensatz zu früheren Schnittellen wie OpenGL, arbeitet Vulkan deutlich näher an der tatsächlichen Hardware. Hierdurch soll sowohl die Komplexität der Grafiktreiber reduziert, als auch eine stabilere und i.d.R. höhere Performance erreicht werden. [Ove17; VK59]

Parallelen zu OpenGL

Trotz dieser größeren Hardwarenähe und der zusätzlichen Kontrollmöglichkeiten, ist die grundsätzliche Struktur der API allerdings trotzdem in Ansätzen vergleichbar mit älteren Schnittstellen wie OpenGL (Core Profile). [Ove17; VK59]

Der markanteste Unterschied beim Einsatz von Vulkan ist die deutlich umfangreichere Konfiguration bei der Initialisierung. Während es bei OpenGL einen globalen zustandsbehafteten OpenGL-Context gibt, wird unter Vulkan jeglicher Zustand in Objekten gespeichert. Das erste dieser Objekte ist die `VkInstance`, welche als Schnittstelle zum Vulkan-Treiber dient und alle anwendungsspezifischen Daten[2] speichert. Mittels diesem kann außerdem die verfügbare Hardware (`VkPhysicalDevice`) aufgelistet (`vkEnumeratePhysicalDevices`) und näher untersucht (`vkGetPhysicalDevice...`) werden. Sobald basierend auf diesen Informationen ein geeignetes `VkPhysicalDevice` ausgewählt wurde, muss mittels `vkCreateDevice` eine logische Verbindung (`VkDevice`) zu diesem hergestellt werden, über welche die folgenden Befehle mit der Hardware interagieren. Dies ermöglicht es nicht nur die für die Anwendung optimale Hardware[3] auszuwählen, sondern erlaubt es auch, nur die Features der Grafikkarte zu aktivieren, die auch tatsächlich benötigt werden und ggf. Funktionen in der Anwendung zu deaktivieren, die auf von der Grafikkarte nicht unterstützten Features aufbauen. [Ove17; VK59]

[2] Beinhalten u.a. Name und Version der Anwendung/Engine sowie ggf. verwendete Erweiterungen.
[3] Zum Beispiel dedizierte, integrierte oder externe Grafikkarte.

Um auf ein Fenster oder auf den Bildschirm (Surface) zu zeichnen, sind darüber hinaus i.d.R. eine oder mehrere `VkSwapchainKHR` notwendig. Diese stellt eine Abstraktion der Zeichenflächen (Presentable Images) und somit z.B. die Verbindung zu einem Fenstermanager dar. Daher ist die Erzeugung der Swapchain abhängig von der konkreten Plattform und als separate Erweiterung in Vulkan umgesetzt[4]. [Ove17; VK59]

Größere Parallelen zu OpenGL gibt es dagegen bei der Erstellung und Verwendung von Texturen (`VkImage` / `VkImageView`) und Buffern (`VkBuffer`). Analog zum Vorgehen unter OpenGL müssen diese vor ihrer Verwendung zuerst mittels `vkCreateImage`, `vkCreateImageView` und `vkCreateBuffer` erstellt und zum Einsatz in Zeichenbefehlen durch `vkCmdBindVertexBuffers`, `vkCmdBindIndexBuffer` und `vkCmdBindDescriptorSets` gebunden werden. Anders als unter OpenGL müssen die Daten und Metadaten zu Texturen und Buffern allerdings bereits bei der Erstellung angegeben werden und das Binden dient lediglich für den Zugriff durch Zeichenoperationen. Außerdem erfolgt das Binden nur im Kontext einer konkreten Menge von Zeichenoperationen, da es unter Vulkan keinen globalen Zustand mit aktuell gebundenen Objekten mehr gibt. Auch ist die Speicherverwaltung und Übertragung der Daten an die Grafikkarte manuell durchzuführen. Darüber hinaus sind Texturen unter Vulkan in die vier Abstraktionen `VkImage`, `VkImageView`, `VkSampler` und `VkDescriptorSet` getrennt. Hierbei stellt ein `VkImage` die eigentlichen Daten eines Bildes dar. Diese können allerdings nicht direkt, sondern nur durch ein `VkImageView` genutzt werden, welches einen zusammenhängenden Bereich eines Bildes[5] repräsentiert. Um diese in einem Fragment-Shader auszulesen, ist i.d.R. außerdem ein `VkSampler` nötig, der detaillierte Angaben zum Auslesen der Textur enthält[6]. Um diese Informationen schließlich in einem Zeichenbefehl verwenden zu können, müssen sie in einem `VkDescriptorSet` zusammengefasst werden. Dort werden die Informationen zu einem oder mehreren, der in den Shadern verwendeten Daten[7], gespeichert um diese zusammen für eine Zeichenoperation zu binden. [Ove17; VK59]

Die Durchführung von Zeichenbefehlen weist ebenfalls gewisse Parallelen zu OpenGL auf, insofern als dass die nötigen Texturen/Buffer gebunden und schließlich ein Zeichenbefehl z.B. mit `vkCmdDraw` oder `vkCmdDrawIndexed` aufgezeichnet wird. Im Gegensatz zu OpenGL erfolgt diese Aufzeichnung der Befehle allerdings nicht implizit durch Manipulation eines globalen Zustandsautomaten, sonder explizit durch Anlegen und Befüllen einer Warteschlange von Befehlen (`vkCommandBuffer`). Hierdurch ist es zum einen möglich mehrere `vkCommandBuffer` in unterschiedlichen Threads parallel aufzuzeichnen und später zu kombinieren (z.B. mittels `vkCmdExecuteCommands`) oder auch konstante `vkCommandBuffer` nur einmal aufzuzeichnen und später wiederzuverwenden. Die Verarbeitung von Befehlen auf der Grafikkarte ist, losgelöst davon, organisiert in eine Menge von `VkQueue`, die verschiedene Operationen wie Datenübertragung, Zeich-

[4] Zu erkennen ist dies an dem *KHR*-Suffix, welches für von der Khronos Group definierte Erweiterungen steht.
[5] MIP-Level, Array-Layer, Komponenten und Bildformat
[6] Beinhalten u.a die Art der Interpolation bei Vergrößerung/Verkleinerung, das Wrapping, Angaben zum Auslesen der MIP-Level und zur anisotrope Filterung.
[7] Zum Beispiel Texturen, Uniform-Buffer und Storage-Buffer.

nen und/oder Compute-Shader ausführen können. Zur tatsächlichen Ausführung müssen die in einem `vkCommandBuffer` aufgezeichneten Befehle also mittels `vkQueueSubmit` an eine bestimmte `VkQueue` übertragen werden. Dies macht es u.a. möglich auch eine GPU-seitig parallele Verarbeitung von Befehlen zu realisieren, sofern dies von der Hardware unterstützt wird. [Ove17; VK59]

Da nicht nur unterschiedliche `VkQueues` sondern auch mehrere Befehle auf der selben `VkQueue` parallel oder in abweichender Reihenfolge verarbeitetet werden könnten, ist leider auch eine umfangreiche manuelle Synchronisation abhängiger Operationen notwendig. Hierzu existieren neben traditionelleren Synchronisations-Primitiven (Fence, Semaphore) auch Pipeline-Barriers, welche eine sehr detaillierte Angabe von Abhängigkeiten zwischen einzelnen Schritten der Grafikpipeline erlauben. [Ove17; VK59]

Ein großes Problem bei der Verwendung von OpenGL-artigen APIs ist, dass dynamische Zustandsänderungen[8] oft mit hohen Kosten verbunden sind, da Shader während des laufenden Frames neu kompiliert oder Hardwarekomponenten neu konfiguriert werden müssen. Da dies zu deutlichen Schwankungen der Bildrate führen kann, setzen aktuelle Treiber i.d.R. Heuristiken ein um derartige Verhaltensweise von Anwendungen frühzeitig zu erkennen. Dies ist allerdings mit einer hohen Komplexität auf Seiten der Treiber verbunden, oft abhängig von konkreten Anwendungen (z.B. Abgleich des Programmnamens) und kann diese Probleme trotzdem nicht in allen Fällen abfangen. Unter Vulkan wurde zur Beseitigung dieser Probleme ein besserer Weg eingeschlagen, indem alle potentiell relevanten Zustandsänderungen im voraus definiert werden müssen. Dies erfolgt durch Konstruktion der folgenden Objekte, die zur Laufzeit während Zeichenoperationen mittels `vkCmdBeginRenderPass` bzw. `vkCmdBindPipeline` gebunden werden können um einen konkreten Zustand der Grafikpipeline zu konfigurieren: [Ove17; VK59]

- `vkRenderPass`: Beschreibt eine Menge von Attachments (z.B. Render-Targets) sowie ihre Verwendung (Shader/Subpasses) und Abhängigkeiten zwischen ihnen. Hierdurch ist es z.B. möglich den Geometrie- und den Beleuchtungs-Pass bei einem Deferred-Renderer in einem `vkRenderPass` mit mehreren Subpasses durchzuführen, was es Tiled-Rendering basierten GPU-Architekturen ggf. erlaubt Schreiboperationen in den GBuffer weg zu optimieren. [Ove17; VK59]
- `vkPipeline`: Definiert den gesamten statischen Zustand der Grafikpipeline während einer Menge von Operationen. Hierzu zählen bei einer Graphic-Pipeline z.B. der verwendete Shader, das Vertex-Layout, Informationen zum Blending, Depth-Test, Tesselation sowie der `vkRenderPass` und Subpass in dem die `vkPipeline` verwendet werden wird. [Ove17; VK59]
- `vkPipelineLayout`: Beschreibt u.a. die Anzahl und das Layout der in einer `vkPipeline` verwendeten `VkDescriptorSets`. [Ove17; VK59]

Begründung des Einsatzes

Diese nicht unerhebliche zusätzliche Komplexität wird allerdings in vielen Anwendungsfällen durch die ebenfalls nicht unerheblichen Vorteile mehr als ausgeglichen.

[8] Zum Beispiel Änderung von Uniform-Variablen, Blending-Einstellungen, Shader oder Render-Targets.

So ist der CPU-Overhead durch die größere Hardwarenähe deutlich geringer als unter OpenGL. Außerdem bestehen dadurch, dass alle relevanten Daten vorab festgelegt werden, mehr Möglichkeiten zur Optimierung durch den Treiber und Bildraten fallen im Allgemeinen konsistenter aus. Außerdem bietet Vulkan u.a. durch die explizite Synchronisation im Gegensatz zu OpenGL deutlich mehr und bessere Möglichkeiten zur Verteilung von Aufgaben auf mehrere CPU-Threads. [Ove17; VK59]

Bei der Entscheidung für dieses Projekt war allerdings vor allem die moderne und subjektiv angenehmere API ausschlaggebend. Darüber hinaus sind gerade die Render-Passes für die globale Beleuchtung ausreichend komplex, dass sich Vorteile durch die Vorabspezifikation ergeben haben sollten.

Alle Details der Vulkanimplementierung an dieser Stelle zu erläutern würde zu weit führen. Auf einen der größten Unterschiede zu OpenGL, die implementierte Resourcenverwaltung, soll allerdings noch eingegangen werden.

Speicherverwaltung

Die grundlegendste Resource, welche manuell verwaltet werden muss, ist der von der GPU verwendete Arbeitsspeicher. Dieser ist abhängig von seiner Anbindung[9] und Eigenschaften[10] in mehrere Heaps und Memory-Types aufgeteilt. Ein Heap stellt dabei eine bestimmte, feste Menge von Speicher dar (z.B. den RAM der GPU oder den GPU-sichtbaren Teil des Arbeitsspeichers der CPU), der jeweils eine Menge von Memory-Types unterstützt. Über den API-Befehl `vkAllocateMemory` kann ein Speicherbereich mit einer bestimmten Größe von einem dieser Memory-Types angefordert werden, was allerdings ggf. mit nicht unerheblichem Rechenaufwand verbunden ist. Darüber hinaus ist es zu empfehlen die Anzahl der gleichzeitig verwendeten Speicherbereiche so gering wie möglich zu halten, da es zum einen eine Hardware/Treiber abhängige Anzahl maximaler Allokationen[11] gibt und abhängig von der Plattform evtl. auch zusätzliche Performancebeeinträchtigungen bei einer großen Zahl von Allokationen[12] bestehen können. [Heb16; VK59]

Die implementierte Speicherwaltung besteht aus einer Reihe von Pools, getrennt nach Memory-Type und zu erwartender Lebenszeit des angeforderten Speichers[13]. Jeder dieser Pools verfügt über eine Liste von Speicherblöcken, welche über eine Buddy-Speicherverwaltung in einzelne Allokationen aufgeteilt werden. Bei dieser Art der Speicherverwaltung wird der Speicher in Blöcke der Größe 2^k aufgeteilt, welche entweder in kleinere Blöcke gleicher Größe aufgespalten oder bei benachbarten Blöcken wieder zu einem verbunden werden können. Zu Beginn besteht die Speicherverwaltung nur aus einem Block, welcher den gesamten Speicherbereich umfasst. Bei der Anforderung von Speicher wird nun die angeforderte Menge auf

[9] Z.B. direkt an die GPU angebunden oder nur über den PCIe-Bus zugänglich.
[10] Z.B. Support für Lazy-Allocations, Sichtbarkeit von der CPU/GPU, gecached, etc.
[11] Auf einigen Plattformen nur 4096 Allokation.
[12] Unter älteren Versionen des Windows Display Driver Model (WDDM) bestehen z.B. pro Submit Kosten die linear mit der Anzahl der allozierten Speicherbereiche wachsen. [Heb16]
[13] Derzeit getrennt nach temporärem, normalem und sehr langlebigem Speicher um Fragmentierung zu vermeiden.

die nächstgrößere Zweierpotenz aufgerundet und der nächste freie Block zurückgeliefert. Falls es keinen entsprechenden Block gibt, werden ggf. größere Blöcke aufgeteilt, bis ein passender Block entsteht oder es wird ein Fehler zurück geliefert, falls kein Block dieser Größe mehr reserviert werden kann[14]. Wenn Speicher wieder freigegeben werden soll, läuft dieser Prozess umgekehrt ab, d.h. der entsprechende Block wird freigegeben und solange mit seinem Nachbarn vereinigt wie möglich. [Kno65]

Der Vorteil dieses Vorgehens liegt vor allem in der vergleichsweise trivialen Implementierung, dem geringen Aufwand für die Anforderung und Freigabe von Speicher ($O(\log n)$ über der Größe des Speichers) und der geringen externen Fragmentierung[15]. Im Regelfall weißt diese Art der Speicherverwaltung ein hohes Maß an interner Fragmentierung[16] auf, wenn Speicher mit einer Größe angefordert wird, der keiner Zweierpotenz entspricht. Da es sich bei dieser Anwendung allerdings bei den meisten angeforderten Speicherbereichen um Texturdaten[17] handelt, stellt dies im vorliegenden Fall kein schwerwiegendes Problem dar. [Kno65]

Laden von Texturen und Modellen

Wie eingangs bereits erwähnt, ist das effiziente Laden von Texturen und Modellen ebenfalls mit zusätzlichem Aufwand verbunden. Um Daten schnell auf der GPU verarbeiten zu können, müssen diese in einem Speicherbereich liegen, der direkt an diese angebunden ist (Device-Local). In diesen Speicherbereich kann allerdings – ausgenommen bei Unified-Memory-Architectures wie in vielen Intel und Smartphone GPUs – nicht von der CPU aus geschrieben werden. Außerdem muss der Zugriff auf den Speicher und die erstellte Resource (z.B. `vkBuffer` oder `vkImage`) ggf. mit folgenden oder vorangegangenen Operationen synchronisiert werden. Darüber hinaus erfordern Bilder noch zusätzliche Operationen (u.a. Kopieren/Konvertieren der Bilddaten in ein passendes Format mit `vkCmdCopyBufferToImage`) bevor sie verwendet werden können. Diese Aufgabe wird vom `Transfer_manager` übernommen, welcher abhängig von der vorliegenden Speicher-Architektur die zu ladenden Resourcen – ggf. über einen Zwischenspeicher (Staging-Buffer) – in den lokalen Speicher der GPU kopiert und sich um die Synchronisation und nötige Konvertierungen kümmert. [Heb16; VK59]

Dieses Vorgehen ist allerdings für Daten wie Uniforms und dynamische Geometrie, die sich in jedem Frame ändern mit einem zu hohen Aufwand und Verzögerungen verbunden. Für diese Daten sind aktuell, abhängig von ihrer Größe, drei Verfahren vorgesehen. [Heb16; VK59]

[14] In diesem Fall wird ein neuer GPU-Speicherblock angefordert und durch eine neue Buddy-Speicherverwaltung übernommen.

[15] Fragmentierung die z.B. entsteht wenn sich Lücken zwischen reservierten Speicherbereichen bilden, sodass der verfügbare Speicher in kleinere Teilbereiche getrennt wird, was das Reservieren eines größeren zusammenhängenden Speicherbereichs verhindern kann auch wenn noch ausreichend viel freier Speicher zur Verfügung stehen würde.

[16] Ungenützter Speicher innerhalb von reservierten Speicherbereichen.

[17] In der Regel mit Seitenlängen bei denen es sich um Zweierpotenzen handelt und 1, 2 oder 4 Byte pro Texel.

Sehr kleine Uniform-Daten, welche sich mehrmals pro Frame ändern, sollten nach Möglichkeit in Form von Push-Constants übertragen werden. Diese werden direkt in den Command-Buffer gespeichert und können effizient in Shadern gelesen werden. Sie sind allerdings abhängig von der Hardware unter Umständen stark in ihrer Größe limitiert (garantierte Größe von nur 128 Byte). Folglich werden Push-Constants z.Z. primär für Variablen von Post-Effekten und die Transformationsmatrizen von Objekten verwendet. [Heb16; VK59]

Größere dynamische Uniforms und andere Daten wie Vertices/Indices können somit nicht als Push-Constants übertragen werden, sondern müssen in einem `vkBuffer` gespeichert und manuell aktualisiert werden. Für derartige Daten mit einer Größe unter 64 KiB ist die `Dynamic_buffer` Klasse vorgesehen, die es erlaubt die neuen Daten des Buffers direkt in den Command-Buffer einzubetten. Dies erfordert deutlich weniger Synchronisation, ist allerdings auf besagte 64 KiB pro Buffer begrenzt und verursacht unter Umständen große Command-Buffer sowie evtl. damit verbundene Performanceprobleme. [Heb16; VK59]

Für große dynamische Daten gibt es schließlich den `Streamed_buffer`, welcher keine Größenbeschränkung aufweist. Dieser verwendet einen Ring-Buffer aus `vkBuffern`, von denen einer pro Frame mit neuen Daten befüllt wird. Dadurch wird garantiert, dass der nächste zu beschreibende Buffer nicht mehr von der GPU gelesen wird, ohne zusätzliche Synchronisation zwischen CPU und GPU zu benötigen. [Heb16; VK59]

3.3 Basis Renderer

Zur Darstellung der Szene und als Basis für das Beleuchtungsverfahren wurde ein Deferred-Renderer[18] entwickelt. Diese Entscheidung ist vor allem dadurch begründet, dass für die Umsetzung des Verfahrens zur indirekten Beleuchtung ohnehin ein G-Buffer[19] benötigt wird. Der Aufbau des verwendeten G-Buffers ist in Abb. 3.1 beschrieben und benötigt 240 Bit pro Pixel an Speicher. Da für die indirekte Beleuchtung sowie einige der Post-Effekte eine MIP-Map-Pyramide für die Texturen des G-Buffers benötigt wird, beläuft sich die gesamte Größe bei einer Auflösung von 1920×1080 somit auf ca. 80 MiB.

Der Renderer ist im Code in der Klasse `renderer::Deferred_renderer` umgesetzt, welche wiederum aus einzelnen Passes[20] besteht. In diesen sind die Zeichenbefehle (z.B. Modelle und UI-Elemente) sowie die Beleuchtung und Post-Effekte annähernd unabhängig von einander implementiert und können somit auch weitestgehend einzeln gezielt aktiviert und deaktiviert werden.

[18] Ein 3D-Renderer der "Deferred Shading" verwendet. Ein Zeichenverfahren bei dem im Gegensatz zum Forward-Rendering das Zeichnen der Modelle und die Beleuchtungsberechnungen in getrennten Schritten durchgeführt werden.

[19] Der Geometry-Buffer, welcher beim Zeichen der Modelle geschrieben und u.a. für die Berechnung der Beleuchtung herangezogen wird. Er ist eine Menge von Texturen und besteht i.d.R. mindestens aus den Oberflächennormalen, Position/Tiefe, Albedo sowie ggf. zusätzlichen Materialeigenschaften.

[20] Nicht zu verwechseln mit den Render-Passes aus Vulkan. Die hier beschriebenen Passes agieren auf einer abstrakteren Ebene und bestehen u.U. auch aus mehreren Vulkan-Render-Passes.

[21] Gespeichert als Octahedron-Normalen nach Meyer, Süßmuth, Sußner, Stamminger und Greiner [MSS+10].

<table>
<tr><td colspan="2">Tiefe</td><td></td><td></td><td></td><td></td></tr>
<tr><td colspan="2">Letzte Tiefe</td><td></td><td></td><td></td><td></td></tr>
<tr><td>Albedo (R)</td><td>Albedo (G)</td><td>Albedo (B)</td><td>Material-ID</td><td></td><td></td></tr>
<tr><td colspan="2">Normal</td><td>Rauheit</td><td>Metall</td><td></td><td></td></tr>
<tr><td colspan="2">Diffuse (R)</td><td colspan="2">Diffuse (G)</td><td colspan="2">Diffuse (B)</td></tr>
<tr><td colspan="2">Farbe A (R)</td><td colspan="2">Farbe A (G)</td><td colspan="2">Farbe A (B)</td></tr>
<tr><td colspan="2">Farbe B (R)</td><td colspan="2">Farbe B (G)</td><td colspan="2">Farbe B (B)</td></tr>
</table>

Abb. 3.1: Die Texturen des G-Buffers: Ein 16 Bit Tiefen-Puffer des aktuellen sowie des letzten Frames, die Farbkanäle (Albedo) in je 8 Bit, eine 8 Bit Material-ID, die materialspezifischen Daten (beim Standard Material für opake Objekte: die in 16 Bit komprimierte Oberflächennormale[21], die Rauheit (Roughness) und der Faktor zur Unterscheidung von Metallen und Isolatoren (Metalness)) sowie das Ergebnis der Beleuchtungsberechnungen (nur der diffuse Teil sowie zwei Farb-Puffer für Ping-Pong-Rendering).

Da die benötigten GPU-Features bereits zur Initialisierung des Vulkan-Devices feststehen müssen, diese sich allerdings evtl. erst aus den aktiven Passes ergeben und der `Deferred_renderer` wiederum zur Initialisierung bereits das Vulkan-Device benötigt, mussten die Aufgaben durch Einsatz des Factory-Patterns weiter aufgeteilt werden. Hierzu wird die Klasse `Deferred_renderer_factory` beim Initialisieren der Anwendung instantiiert und übernimmt die Auswahl der passenden GPU[22], ihre Initialisierung sowie das abschließende Zeichnen auf den Bildschirm pro Frame (`vkSubmit`). Als Parameter erhält diese Klasse eine Liste aller benötigten Passes als `Pass_factory`[23]. Diese verfügen über Funktionen um die Tauglichkeit einer GPU anhand ihrer verfügbaren Features zu bewerten, für den Pass nötige Features bei der Initialisierung des Devices zu Aktivieren und Konfigurieren und schließlich eine konkrete Instanz des zugehörigen `Pass` zu erzeugen. Letzteres geschieht für jeden `Deferred_renderer` welcher aus der Factory erzeugt wird. Während es pro Anwendung i.d.R. nur eine Instanz der `Deferred_renderer_factory` geben sollte, können zur selben Zeit prinzipiell beliebig viele `Deferred_renderer` Instanzen aktiv sein um z.B. schnell zwischen unterschiedlichen Spielmodi wechseln zu können oder es zu ermöglichen mehrere logisch getrennte Spielmodi neben/über einander zu zeichnen.

[22] Diese erfolgt basierend auf einer Liste von `vkPhysicalDevice`, welche alle unterstützen GPUs des Systems enthält und Einblick in die von ihnen unterstützten Funktionen und Hardwaredetails – wie dem verfügbaren Speicher – ermöglicht.

[23] Eine vollständige Auflistung der implementierten Passes befindet sich in Anhang C.

3.4 Beleuchtungsmodel

Als zugrundeliegendes Beleuchtungsmodell des Renderers wird das in Kapitel 2 beschriebene Cook-Torrance-Beleuchtungsverfahren verwendet. In der aktuellen Ausführung des Renderers kommt für dieses ein vereinfachtes physikalisches Material-Model basierend auf einem Roughness-Metal-Workflow zum Einsatz, wie er z.B. auch in der UnrealEngine verwendet wird. Hierzu wird jedem Modell (oder Untermodell) ein Material zugeordnet, welches aus einer Material-ID, einer Albedo sowie einer Material-Data Textur besteht. Die Material-ID ermöglicht die Wahl des Shaders welcher zur Darstellung verwendet wird. Aktuell umfasst dies einen Standard-Shader für opake Objekte und einen Shader für Objekte die lediglich Licht emittieren. Das Vorgehen lässt sich allerdings in Zukunft auch um komplexere Effekte (Sub-Surface-Scattering, Transparenz, nur teilweise emittierende Oberflächen) erweitern. Für opake Objekte besteht die Material-Data Textur, analog zum Layout des G-Buffers, aus der Oberflächennormalen (hier im Tangent-Space), der Rauheit und der Metalness.

Die für die Berechnung notwendigen Materialeigenschaften sind neben der Oberflächennormale, das diffuse Rückstrahlvermögen C, das Reflexionsvermögen bei Betrachtung senkrecht zur Ebene $F0$ sowie die Rauheit der Oberfläche α. Die vereinfachende Beobachtung des Roughness-Metal-Workflows ist, dass $F0$ bei allen dielektrischen Oberflächen frequenzunabhängig ist[24] und mit wenigen Ausnahmen (z.B. Edelsteine) auf den Bereich $\leq 10\%$ beschränkt ist. Somit lässt sich der $F0$-Wert aller dielektrischen Oberflächen hinreichend gut mit einem konstanten Wert von 4% approximieren. Größere Varianzen und höhere Werte treten dagegen primär bei Metallen auf, welche im Gegenzug keine diffusen Reflexionen aufweisen (C ist immer schwarz). Basierend auf diesen Beobachtungen lassen sich die beiden Werte C und $F0$ mit je drei Komponenten als ein Albedo-Wert C_{albedo} und ein Metalness-Faktor m kodieren um sowohl Speicherplatz zu sparen, als auch eine intuitivere Bearbeitung der Texturen zu ermöglichen. Die Berechnung der Werte erfolgt somit gemäß:

$$\begin{aligned} C &= C_{albedo} \cdot (1 - m) \\ F0 &= 0{,}04 \cdot (1 - m) + C_{albedo} \cdot m \end{aligned} \tag{3.1}$$

3.4.1 Schatten bei direkter Beleuchtung

Zur Berechnung der Schatten bei der lokalen Beleuchtung kommt ein Standard-Shadow-Mapping Verfahren mit Percentage-Closer-Filtering (PCF) zum Einsatz. Bei diesem wird die Intensität der lokalen Beleuchtung mit einem Sichtbarkeitsfaktor skaliert. Die Berechnung dieses Faktors erfolgt durch Zeichnen der gesamten Szene aus Perspektive der Lichtquelle[25], wobei nur der Abstand der nächsten Oberfläche zur Kamera erfasst wird. Die so entstandene Textur wird als Shadowmap bezeichnet. Bei der Beleuchtung kann anschließend die Positionen des aktuell zu

[24] Die Farbe des $F0$ Wertes ist also in diesen Fällen immer ein Grauwert.
[25] Einmal beim Start der Anwendung und bei Änderungen der Lichtposition.

beleuchtenden Punktes in das Koordinatensystem der Lichtquelle transformiert und mit der entsprechenden Position in der Shadowmap verglichen werden. Die Sichtbarkeit an dem Punkt ist somit 0 wenn sein Abstand zur Lichtquelle größer als der aus der Shadowmap ist und ansonsten 1. [Fer05]

Durch die geringe Auflösung der Shadowmap in Verbindung mit dem u.U. sehr steilen Betrachtungswinkel von Oberflächen kommt es allerdings zu Artefakten, welche sich durch Schatten auf eigentlich hellen Oberflächen äußern (Shadow-Acne). Um diesen Artefakten zu begegnen kommt ein statischer Offset von 0,00035 zum Einsatz, welcher vom Abstand des aktuellen Punktes abgezogen wird. Dieser Offset allein genügt allerdings noch nicht um die Artefakte vollständig zu bereinigen und bei der Wahl eines zu großen Offset entsteht ein sichtbarer Abstand zwischen den Schatten und den Objekten die sie werfen (Peter-Panning). Daher kommt zusätzlich noch ein dynamischer Offset zur Anwendung, welcher die Position des Punktes im World-Space um die mit einem Faktor von 0,06 skalierte Oberflächennormale (Normal-Offset) verschiebt. Dadurch ergibt sich ein zufriedenstellendes Gesamtergebnis, bei dem sowohl Shadow-Acne minimiert wird, als auch der Kontakt der Schatten zu den Objekten gewahrt bleibt. [Hol10]

Durch das bisher geschilderte Verfahren sind allerdings nur Schatten mit harten Kanten, d.h. ohne flüssige Farbverläufe zwischen Licht und Schatten, realisierbar. Für weiche Schatten wird das bisherige Verfahren um PCF erweitert. Dabei wird auch der Bereich im unmittelbaren Umfeld des aktuellen Punktes in der Shadowmap ausgelesen, mit dem Punkt verglichen und das arithmetische Mittel der Sichtbarkeitsfaktoren berechnet. Zu diesem Zweck werden im Renderer bis zu 16 Samples berechnet, deren Position anhand einer zufällig rotierten Poisson-Disc[26] bestimmt wird. Durch Einsatz der rotierten Poisson-Disc werden sowohl sichtbare Pixel-Artefakte, als auch übermäßiges Rauschen minimiert. Außerdem werden, bei entsprechender Hardwareunterstützung, auch noch pro Sample die 4 jeweils umliegenden Texel mit betrachtet[27]. [Fer05]

In der Realität ist die Härte eines Schatten von der Größe der Lichtquelle sowie den Abständen zwischen der Lichtquelle und den beteiligten Objekten abhängig. Im Falle des Sonnenlichts wird von einer unendlich weit entfernten Lichtquelle ausgegangen, wodurch die Größe und der Abstand der Lichtquelle weitgehend irrelevant wird. Daher erfolgt über vier weitere Shadowmap-Samples aus dem Umfeld des Punkte eine Abschätzung der durchschnittlichen Distanz der schattenwerfenden Oberflächen vor dem Punkt. Anhand der Differenz dieser Distanz zur Position des Punktes kann anschließend der Radius der Poisson-Disc für die eigentliche Schattenberechnung passend variiert werden. [Fer05]

Durch die zufällige Wahl sowie geringe Anzahl der Samples kommt es zu einem nicht unerheblichen Rauschen an den Schattenübergängen. Dieses wird allerdings vollständig von der temporalen Kantenglättung erkannt und bereinigt.

[26] Eine Menge von Punkten mit einem festen maximalen Abstand, welche einer Poisson-Verteilung unterliegen.

[27] Erfolgt mittels bilineare Filterung in Hardware mittels `sampler2DShadow` in GLSL.

3.5 Tone Mapping

Zur realistischen Darstellung einer Szene ist, vor allem bei Verwendung eines physikalisch plausiblen Beleuchtungsmodells, ein größerer Wertebereich für die Farbkanäle als die von den meisten Monitoren unterstützen 255 Werte (8 Bit) nötig. Im hier implementierten Renderer erfolgt die Beleuchtung daher mit 16 Bit pro Farbkanal und in einem linearen Farbraum. Zur Darstellung auf dem Bildschirm muss dieses Format allerdings wieder in die zumeist 8 Bit (im Gamma-Space) des Monitors konvertiert werden. Die Konvertierung vom linearen in den Farbraum des Monitors erfolgt über die Hardware bzw. den Vulkan-Treiber durch Wahl der entsprechenden Textur-Formate. Für eine qualitativ hochwertige Reduzierung des Dynamikumfangs (von 16 auf 8 Bit) ist allerdings ein manuelles Tone-Mapping erforderlich. [Pet10]

Im Renderer kommt hierzu ein globaler Tone-Mapping-Filter mit einer filmischen Kurve[28] zum Einsatz, um einen kontrastreichen Gesamteindruck und intensivere Schatten zu erhalten. [Dui10; Pet10]

Um den größeren Dynamikumfang der Szene auch bei sich ändernden Lichtverhältnissen sinnvoll auf den geringen Dynamikumfang des Bildschirms abzubilden, muss darüber hinaus die Anpassung des menschlichen Auges an unterschiedliche Helligkeiten berücksichtigt werden. Bei Tone-Mapping-Verfahren kann die Anpassung an die Helligkeit der Umgebung analog zur Photographie über die Belichtungszeit (Exposure) realisiert werden. Bei statischen Szenen ist es oft sinnvoll diesen Wert manuell festzulegen um exakt die gewünschte Lichtstimmung zu erreichen. Bei größeren oder dynamischen Szenen ist dieses Vorgehen allerdings sehr schnell nicht mehr umsetzbar. Daher kommt ein automatisches Verfahren zur Berechnung der Belichtungszeit zum Einsatz. Hierzu wird das bisher berechnete Bild analysiert, die durchschnittliche Helligkeit bestimmt und diese in einer neuen Textur gespeichert. Dieser Wert wird nun über die Zeit mit dem letzten berechneten Wert interpoliert um die langsame Anpassung des Auges an abrupte Helligkeitsunterschiede zu simulieren. Aus diesem Helligkeitswert l lässt sich anschließend beim Tone-Mapping mit folgender Formel die Belichtungszeit e ableiten: [Pet10]

$$\begin{aligned} k &= 1{,}03 - \frac{2}{2 + \log_{10}(l+1)} \\ e &= \frac{k}{l} \end{aligned} \tag{3.2}$$

Obwohl die Farben durch das Tone-Mapping insgesamt relativ gut repräsentiert werden, geht durch den geringen Dynamikumfang des Bildschirms trotzdem noch ein Effekt verloren: Die Überstrahlung von im Vergleich zur Umgebung extrem hellen Oberflächen.

Dieser Effekt ist vor allem beim Einsatz von indirekter Beleuchtung wichtig, damit helle Bereiche auch nach der Anpassung der Belichtungszeit ihren subjektiv hohen Kontrast behalten (siehe Abb. 3.2). Um diesen Effekt zu simulieren kommt

[28] Engl. Filmic-Curve. Eine grob auf der charakteristischen Kurve von analogem Film basierende Abbildungskurve. [Dui10]

zusätzlich ein Bloom-Filter zu Einsatz. Bei diesem wird das bisher berechnete Bild (vor dem Tone-Mapping) einem separaten Tone-Mapping-Schritt mit einer niedrigen Belichtungszeit unterzogen um die Bereiche mit hoher Helligkeit zu selektieren. Das Ergebnis dieses Schrittes wird anschließend mit einem Tiefpassfilter bearbeitet und multipliziert mit einem zusätzlichen Faktor zum Endergebnis (nach dem Tone-Mapping) addiert.

(a) Mit Bloom (b) Ohne Bloom

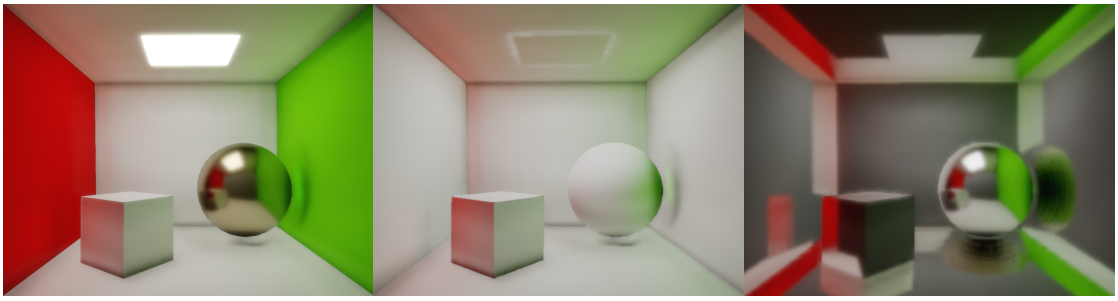
Abb. 3.2: Vergleich einer beleuchteten Szene mit (a) und ohne (b) Bloom. Die Helligkeitsunterschiede sind vor allem in den helleren Bereichen (eckige Säule) etwas deutlicher zu erkennen.

3.6 Globale Beleuchtung

Die Berechnung der globalen Beleuchtung erfolgt gesammelt im `Gi_pass` und besteht aus den folgenden Teilschritten (siehe Abb. 3.4 für einen detaillierten Graph aller Teilschritte):

1. Temporale Reprojektion der Ergebnisse des letzten Frames und Integration in die Eingabe für diesen Frame. Sowie allgemeine Vorbereitungen für die eigentlichen Berechnungen.
2. Berechnung der eingehenden Beleuchtung für die diffuse Beleuchtung (siehe Abb. 3.3).
3. Berechnung des eingehenden Beleuchtung für die spekulare Beleuchtung (siehe Abb. 3.3).
4. Kombination der berechneten eingehenden diffusen und spekularen Beleuchtung und Berechnung der finalen globalen Beleuchtung.

Die Eingabe für den Pass bildet neben der intern gespeicherten, zuletzt berechneten indirekten Beleuchtung der konstruierte G-Buffer inkl. eine Textur mit dem diffusen Ergebnis der direkten Beleuchtung. Das Verfahren könnte auch direkt auf dem normalen Ergebnis der Beleuchtung inkl. des spekularen Teils arbeiten. Die spekulare Beleuchtung ist allerdings abhängig vom Blickwinkel und somit für alle Punkte, die nicht am selben Punkt wie die Kamera liegen nicht korrekt. Dieser Fehler verstärkt Überstrahlungs-Artefakte (Fireflys) massiv und führt allgemein zu einem weniger korrekten und störanfälligeren Ergebnis.



(a) Endergebnis (b) Diffuse Einstrahlung (c) Spekulare Einstrahlung

Abb. 3.3: Die finale Beleuchtung (a) sowie die berechnete diffuse (b) und spekulare (c) Einstrahlung an jedem sichtbaren Punkt der Szene. Bei der Szene handelt es sich um eine Cornell-Box ohne analytische Lichtquelle, mit einer Kupfer-Kugel, einem weißen Würfel, drei weißen, einer grünen und einer roten Wand.

Das zentrale Problem bei der Berechnung von globaler Beleuchtung ist die Anzahl der nötigen Samples um ein ausreichend hochauflösendes und stabiles Ergebnis zu gewährleisten (kein Rauschen oder Flackern). Um dieses Problem abzuschwächen können die nötigen Samples über mehrere Frames verteilt werden, da selbst bei hochgradig dynamischen Szenen von einer gewissen temporalen Kohärenz[29] ausgegangen werden kann. Dazu wird, analog zum Vorgehen bei der temporalen Kantenglättung (siehe Kapitel 3.7), ein History-Buffer mit dem letzten Ergebnis der Beleuchtung gespeichert und dieser über die Zeit mit den aktuellen Werten mittels $c_t = c_t + (c_{t-1} - c_t) \cdot \tau$ geblendet. Die Glättung und Reprojektion erfolgt dabei analog zum in Kapitel 3.7 geschilderten Vorgehen. Es wird allerdings auf eine Einschränkung der Werte auf den aktuellen Wertebereich der Umgebung verzichtet und stattdessen ein Abgleich mit dem Tiefen-Puffer des letzten Frames durchgeführt um falsch zugeordnete Pixel zu verwerfen. Hierzu wird ein Faktor $0{,}1 \leq \tau \leq 0{,}98$ automatisch basierend auf der Ähnlichkeit der Tiefenwerte und der Bildrate gewählt. Außerdem wird dieses Verfahren hier nicht dazu verwendet das Endergebnis zu glätten, sondern wird auf die gesammelte diffuse und spekulare Beleuchtung einzeln angewendet. Dadurch kann hier eine höhere Glättung bei weniger auffälligen Artefakten erreicht werden. Der Speicherbedarf ist allerdings auch minimal höher, da zwei statt nur einem History-Buffer benötigt werden. Dies ermöglicht es außerdem für diffuse und spekulare Beleuchtung unterschiedliche Gewichte bei der Glättung zu verwenden und so eine zu starke Weichzeichnung von Reflexionen unter Bewegung zu vermeiden.

Der letzte Schritt ist die Berechnung der tatsächlichen indirekten Beleuchtung aus der gesammelten diffusen und spekularen Einstrahlung. Es wurde versucht in diesem Schritt so viele der nötigen Berechnungen wie möglich durchzuführen, damit zum einen das Sammeln der Samples in einer möglichst niedrigen Auflösung erfolgen kann und zum anderen trotzdem die hohen Frequenzen aus dem G-Buffer (F0, Albedo, Normale) erhalten bleiben. Die finale Beleuchtung erfolgt basierend auf dem Cook-Torrance-Verfahren unter Anwendung von Gleichung 2.19. Das Endergebnis wird schließlich noch auf einen sinnvollen Wertebereich (derzeit

[29] Das heißt die meisten Objekte befinden sich für längere Zeit an einer ähnlichen Position.

0-10 pro Komponente) eingeschränkt um die Auswirkungen der Akkumulation von Berechnungsfehlern einzuschränken. Das Verfahren arbeitet zwar im allgemeinen korrekt, im Laufe der Entwicklung haben sich aber wiederholt Situationen ergeben in denen es zu einer Feedbackschleife gekommen ist, was ohne eine entsprechende Einschränkung zu einer exponentiellen Saturation aller betroffenen Texturen führt.

Da das Beleuchtungsverfahren ausschließlich im Screen-Space arbeitet, fehlen naturgemäß jegliche Informationen zu Objekten die aktuell nicht sichtbar sind. Dies fällt vor allem bei Kamerabewegungen negativ auf, da es dabei zu abrupten Änderungen der Beleuchtung kommen kann, sobald ein helles Objekte nicht mehr sichtbar ist. Um diesen Effekt abzuschwächen und die Beschränkung auf sichtbare Objekte zu verschleiern, wird die Szene mit einem minimal höheren Sichtfeld (FOV) gezeichnet und erst im TAA-Schritt auf den finalen FOV-Wert eingeschränkt. Die problematischsten Bereiche am Rand sind somit nicht mehr unmittelbar sichtbar und auch Objekte die gerade so außerhalb des Sichtfeldes liegen, haben noch einen Einfluss auf die sichtbare Beleuchtung. Da bei gleicher Auflösung mit einem höheren FOV gerendert wird, stehen weniger Pixel für die selbe Fläche zur Verfügung, was nach der Reprojektion auf das finale FOV zu sichtbaren Pixeln führen kann. Diese Artefakte lassen sich allerdings weitestgehend durch das temporale Supersampling im TAA-Schritt ausgleichen.

Eine schematische Darstellung aller relevanten Teilschritte des Renderers ist in Abb. 3.4 dargestellt. Die in dieser verwendeten Texturen umfassen:

1. Tiefenpuffer (inkl. MIP-Maps)
2. Albedo (inkl. MIP-Maps)
3. Material-Daten (inkl. MIP-Maps)
4. Ergebnis der direkten Beleuchtung (inkl. Eigenemission)
5. Gesammelte spekulare indirekte Beleuchtung (14) des letzten Frames
6. Gesammelte diffuse indirekte Beleuchtung (16) des letzten Frames
7. Tiefenpuffer des letzten Frames
8. Ergebnis der direkten Beleuchtung des aktuellen und der indirekten Beleuchtung des letzten Frames (inkl. MIP-Maps)
9. Gesammelte diffuse eingehende Beleuchtung für k=n
10. Gesammelte diffuse eingehende Beleuchtung für k=n-1
11. Gesammelte diffuse eingehende Beleuchtung für k=n (vergrößert durch Joint-Bilateral-Upsampling)
12. Gesammelte diffuse eingehende Beleuchtung für k=0
13. Gesammelte diffuse eingehende Beleuchtung für k=1 (vergrößert durch Joint-Bilateral-Upsampling)
14. Gesammelte spekulare eingehende Beleuchtung
15. Berechnete Umgebungsverdeckung
16. Gesamte gesammelte diffuse eingehende Beleuchtung moduliert mit der Umgebungsverdeckung
17. Finale globale Beleuchtung
18. Kombinierte globale und lokale Beleuchtung
19. Finaler Frame

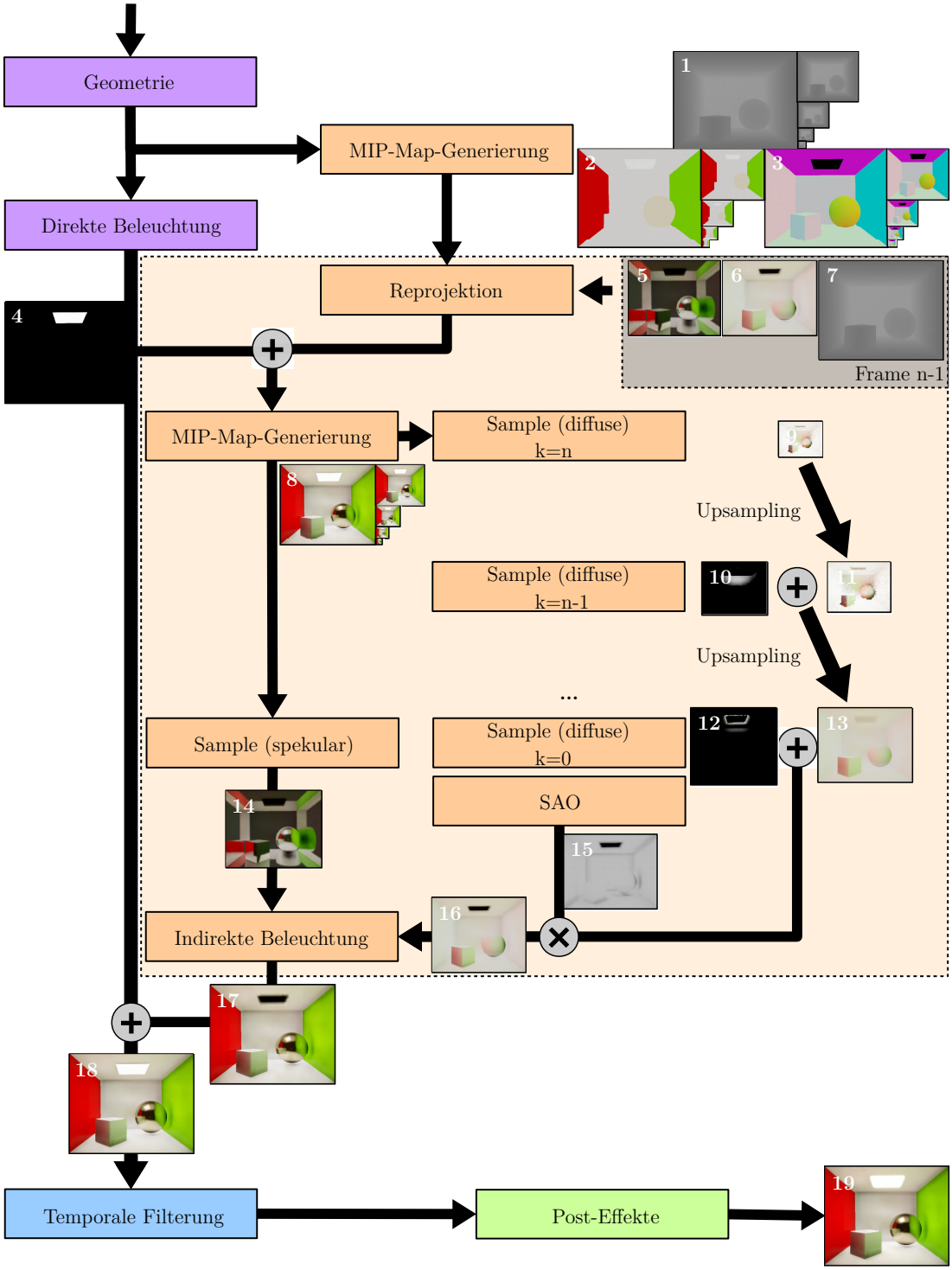
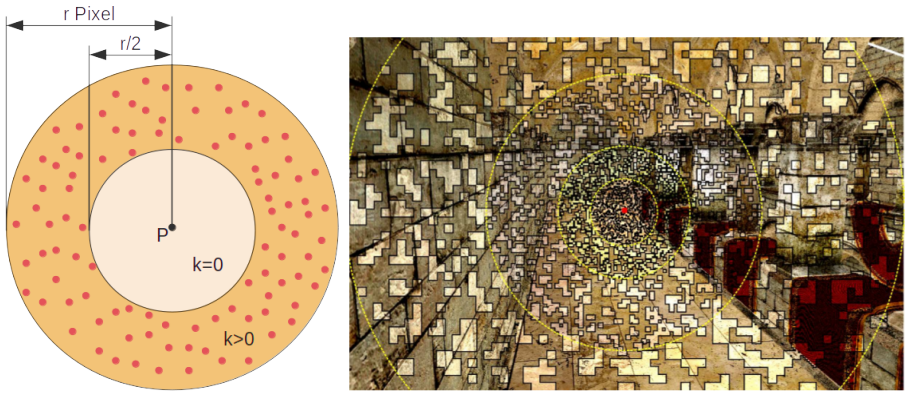


Abb. 3.4: Eine schematische Darstellung der gesamten Beleuchtungsberechnung. Zeigt die einzelnen Teilschritte, ihre Interaktionen sowie die zwischen ihnen ausgetauschten Texturen/Daten. Die Post-Effekte (Tone-Mapping und Bloom) werden hier nur verkürzt wiedergegeben und die Teilschritte der diffusen globalen Beleuchtung (GI Diffuse Samples $k = n-2$ bis $k = 1$) wurden ebenfalls ausgelassen. Zu beachten ist außerdem, dass die mit MIP-Maps versehenen Texturen des GBuffers (1 – 3) von allen Teilschritten der globalen Beleuchtung (oranger Kasten) gelesen werden und der *GI Reprojektion* Schritt neben den Eingabedaten des aktuellen auch auf die Ergebnisse des letzten Frames zurückgreift.

3.6.1 Diffuse Beleuchtung

Die Aufgabe dieses Teilschrittes ist die Lösung von Gleichung 2.13 mittels Quasi-Monte-Carlo-Sampling der Eingabetextur mit der beleuchteten Szene. Dies erfolgt vereinfacht ausgedrückt über das Aufsummieren einer großen Anzahl von Messwerten an einer quasi-zufälligen Sequenz von Punkten. Als Sequenz wurde dabei eine am aktuellen Punkt zentrierte Spirale – analog zum in Kapitel 2.3 erläuterten Vorgehen bei der Berechnung der Umgebungsverdeckung – gewählt. Der Rechenaufwand für diesen Schritt wäre aufgrund der großen Anzahl von weit von einander entfernten Texeln sehr hoch[30], weshalb die Berechnung über einen hierarchischen Ansatz erfolgt.

Hierzu werden zuerst MIP-Maps für den benötigten Teil des G-Buffers (Farbe, Tiefe und Normalen) berechnet und diese anschließend von der höchsten Stufe (kleinste Auflösung) an durch einen lokal arbeitenden Shader verarbeitet. Dieser bestimmt für jeden Punkt eine Menge von Messpunkten in einem festen Umkreis in Pixeln (40px) um den aktuellen Punkt (siehe Abb. 3.5), löst das innere Produkt aus Gleichung 2.13 für jeden davon und summiert die Ergebnisse auf.



(a) (b)

Abb. 3.5: Der Bereich aus dem die Samples für den Punkt P bezogen werden (a) sowie alle Bereiche, die zur Beleuchtung am roten Punkt beitragen (b). Mit Ausnahme des ersten MIP-Levels (k=0) werden die Samples aus einem Ring begrenzt durch den Radius r und $\frac{r}{2}$ bezogen. [SHR10]

Nach jedem dieser Schritte wird das Ergebnis durch einen Upsampling-Filter auf die Auflösung des nächsten MIP-Levels vergrößert und die Berechnung auf diesem Level erneut durchgeführt und additiv zusammen geblendet, sodass am Ende im Schnitt jeder Bereich auf dem Bildschirm zur Beleuchtung jedes Punktes beiträgt. Durch das mehrfache Vergrößern der höheren Level werden Einflüsse von weiter entfernten Flächen stärker geglättet als solche von nahegelegenen und darüber hinaus glatte Übergänge zwischen den unterschiedlichen Leveln erzeugt. Außerdem tragen Punkte im direkten Umfeld mehr zum Endergebnis bei als weiter

[30] Das Auslesen von Texeln, die nahe beieinander liegen erfolgt i.d.R. über den Cache der GPU statt direkt über den RAM, was die Geschwindigkeit deutlich erhöht. Dies ist für weiter entfernte Texel allerdings nicht mehr zwangsläufig der Fall.

entfernte Punkte, da die Anzahl der Samples pro Level konstant bleibt. Dies deckt sich mit der menschlichen Wahrnehmung, da Details der indirekte Beleuchtung von nahe beieinander liegenden Flächen i.d.R. auffälliger sind als jene von weiter entfernten Flächen.

Die gesammelten Beleuchtungsinformationen aus jedem Level k müssen allerdings noch mit einem Faktor 2^{2k} skaliert werden um levelübergreifend eine gleichbleibende Beleuchtungsstärke zu erhalten. Um einen – aus künstlerischer Sicht oft erwünschten – stärker betonten Farbtransfer zwischen benachbarten Flächen zu erhalten, kann dieser Faktor geringfügig modifiziert werden um die Einstrahlung von näher gelegenen Punkten zu priorisieren (siehe Abb. 3.6).

(a) 2^{2k} (b) $2^{(k_{\max}-k_{\min}-1)\cdot 2}$

Abb. 3.6: Die berechnete diffuse indirekte Beleuchtung an jedem Punkt sowie das Endergebnis, jeweils skaliert mit einem Faktor von 2^{2k} (a) und $2^{(k_{\max}-k_{\min}-1)\cdot 2}$ (b). Hierbei sind $k_{\min}$ und $k_{\max}$ der kleinste bzw. größte Wert von k über alle betrachteten MIP-Level. In der Implementierung kann mittels eines Faktors fließend zwischen diesen beiden Exponenten interpoliert werden, um den gewünschten visuellen Eindruck festzulegen.

Um den gesamten Bildschirm abzudecken und so dunklere Bereiche an den Rändern zu vermeiden, sollte des höchste verwendete MIP-Level eine Diagonale von r nicht überschreiten. Die Berechnung muss allerdings nicht zwingend bei MIP-Level 0 beginnen, sondern kann variiert werden um eine höhere Geschwindigkeit auf Kosten der Details bei näher gelegenen Flächen zu erreichen. Da es sich bei der diffusen Reflexion um ein sehr niederfrequentes Phänomen handelt, können bei Verwendung eines hinreichend guten Upsampling-Verfahrens auch bereits bei niedrigen Auflösungen qualitativ hochwertige Resultate erzielt werden. Diese Implementierung verwendet standardmäßig eine fixe Auflösung von ca. 960×500[31]

[31] Sofern nötig wird diese auf einen glatten Bruchteil der endgültigen Auflösung auf- oder abgerundet.

für das niedrigste MIP-Level, was bei allen getesteten Auflösungen gute Ergebnisse geliefert hat und eine möglichst auflösungsunabhängige Laufzeit und Qualität garantiert. Ein Ausschnitt des Shaders zur Bestimmung der diffusen indirekten Einstrahlung befindet sich in Anhang D.3.

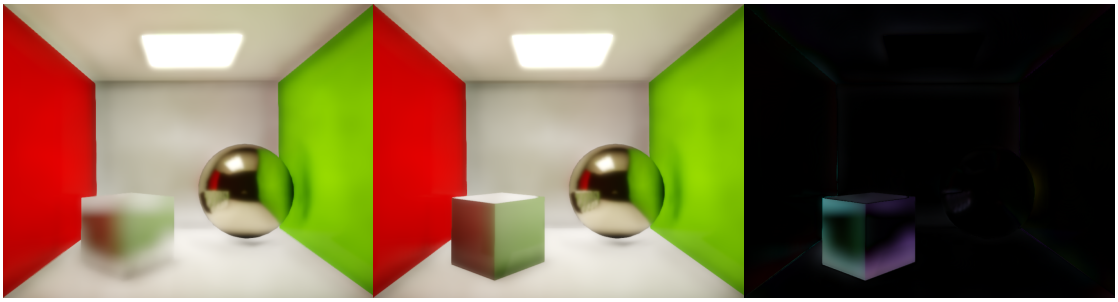
Für das Vergrößern der Teilergebnisse kommt ein modifizierter Joint-Bilateral-Upsampling-Filter zum Einsatz. Dieser erhält als Eingabe einen hoch aufgelösten Tiefen-Puffer $\widetilde{I}$ sowie eine niedrig aufgelöste Beleuchtungslösung S und berechnet daraus eine hoch aufgelöste Lösung $\widetilde{S}$ indem ein gaußscher Filter simultan auf $\widetilde{I}$ und S angewendet wird um einen zusätzlichen Gewichtungsfaktor anhand des Tiefenwertes zu bestimmen. [KCL+07]

Zusätzlich wird außerdem noch die Oberflächennormale $\widetilde{N}$ mit einbezogen, um eine differenziertere Trennung an Kanten mit ähnlicher Position aber stark abweichenden Richtungen zu erhalten (siehe Abb. 3.7). Da außerdem die niedrig aufgelösten Normalen N und Tiefeninformationen I vorliegen, welche zur Berechnung von S verwendet wurden, werden diese zum Bestimmen des Gewichtungsfaktors mit den hoch aufgelösten Daten abgeglichen. Dies führt nicht nur zu geringfügig besseren Ergebnissen, sondern erhöht auch die Cache-Lokalität der Textur-Zugriffe und damit die Ausführungsgeschwindigkeit des Filters.

Die Berechnung der hoch aufgelösten Lösung an Punkt p erfolgt somit durch den folgenden Filter über die Punkte Ω in Pixelkoordinaten der hoch aufgelösten Texturen und mit $p\downarrow$ bzw. $q\downarrow$ in den Koordinaten der niedrig aufgelösten Lösung:

$$\widetilde{S} = \frac{1}{k} \sum_{q\downarrow in \Omega} S_{q\downarrow} f(\|p\downarrow - q\downarrow\|) g(\|\widetilde{N}_p \bullet N_{q\downarrow}) h(\|\widetilde{I}_p - I_{q\downarrow}\|) \tag{3.3}$$

Hierbei ist k die Summe aller berechneten Gewichtungsfaktoren oder 1 falls die Summe einen Schwellenwert ϵ unterschreitet und g, h, f gaußsche Filter-Kerne. [KCL+07]



(a) Ohne Normalen (b) Mit Normalen (c) Differenz

Abb. 3.7: Die indirekt Beleuchtete Cornell-Box ohne (a) und mit (b) dem zusätzlichen Gewichtungsfaktor basierend auf der Oberflächennormale sowie die Differenz der beiden Bilder (c). Ohne den Faktor kommt es vor allem bei dem Würfel, aber auch vereinzelt bei anderen Kanten, zu inkorrekten Farbverläufen über Objektkanten hinweg.

Da das Upsampling einen der aufwändigsten Teilschritte darstellt, mussten einige zusätzliche Optimierungen angebracht werden. So wird zum Abgleich der Normalen auf einen gaußschen Filter sowie eine Dekodierung der Normalen verzichtet.

Dies ist zwar allgemein unzulässig, stellt im vorliegenden Fall allerdings eine hinreichend gute Approximation dar, da lediglich der Unterschied der Normalen auf einen bestimmten Schwellenwerten hin geprüft wird. Darüber hinaus wurde die Berechnung der Gewichtungsfaktoren über eine manuelle Vektorisierung und unter Verwendung des GLSL-Befehls `textureGather(...)` zum gleichzeitigen Auslesen mehrerer Pixel so weit wie möglich parallelisiert. Ein Auszug des entsprechenden Shaders befindet sich in Anhang D.1.

3.6.2 Generierung der MIP-Maps

Für die oben beschriebenen Berechnungen sind MIP-Maps von Teilen des G-Buffers nötig. Deren Berechnung erfordert einiges an zusätzlicher Aufmerksamkeit, da sich durch die normale hardwarebeschleunigte Berechnung von MIP-Maps basierend auf linearer Interpolation für Tiefenwerte und Normalen, Werte ergeben würden, die im Ausgangsbild nicht vorkommen und somit fehlerhafte Beleuchtungsergebnisse liefern würden. Bei einer Kante mit zwei benachbarten Punkten mit sehr unterschiedlichen Tiefenwerten wäre der berechnete Wert auf dem nächsthöheren MIP-Level z.B. ein Tiefenwert in der Luft zwischen diesen beiden Punkten. Eine häufige Alternative bei der Berechnung von MIP-Maps von Tiefenwerten ist daher die Verwendung des Minimums, Maximums oder nächsten Nachbarn (Nearest-Neighbor). Dies führt allerdings ebenfalls dazu das kleine Details u.U. vergrößert werden und abhängig von der Perspektive auftauchen/verschwinden, was zu auffälligen temporalen Inkonsistenzen führt. [SHR10]

Zur Berechnung der MIP-Maps für die Normalen und den Tiefenpuffer wird daher das von Soler, Hoel und Rochet [SHR10] vorgestellte Abstimmungsverfahren verwendet. Dieser Filter berechnet für jeden Pixel p_i der dem aktuellen im vorherigen Level entspricht einen Score und verwendet nur den Tiefen-/Normalen-Wert des Pixels mit dem höchsten Score. Hierzu wird der Unterschied der Normalen N und Tiefenwerte Z zur Umgebung des Pixels betrachtet und unter Verwendung der Gauß-Funktionen g_1, g_2 und g_3 verrechnet: [SHR10]

$$s(p_i) = \sum_{p_j : \|p_j - p_i\| \leq R} g_1(\|p_i - p_j\|) g_2(1 - N_j \bullet N_i) g_3(Z_j - Z_i) \tag{3.4}$$

Dieser Filter bevorzugt große konsistente Bereiche, was im allgemeinen zu temporal stabileren und rauschärmeren Ergebnissen führt (siehe Abb. 3.8). [SHR10]

Der vorgestellte Filter liefert zwar qualitativ hochwertige Ergebnisse, ist allerdings durch das notwendige Sammeln von Informationen im Umkreis des Pixels vergleichsweise aufwändig. Daher gibt es zusätzlich die Option dieses Verfahren erst ab einem späteren MIP-Level einzusetzen und die vorherigen Level durch Nearest-Neighbor Interpolation zu generieren. Die hierdurch entstehenden Artefakte sind bei den ersten 1–2 Leveln noch zu vernachlässigen, bedeuten durch die exponentiell abnehmende Auflösung mit jedem Level allerdings unter Umständen einen nicht unerheblichen Performancevorteil.

Da die Berechnung der MIP-Maps von Normalen und Tiefenwerten durch die Scoring-Funktion inhärent verbunden ist und diese auch von anderen Schritten

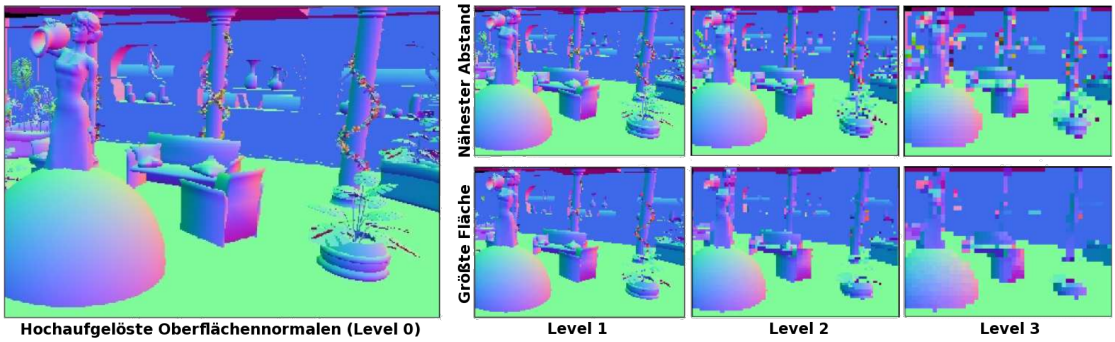


Abb. 3.8: Beispiel für die Filterung der Ausgangsnormalen (links) durch Nearest-Neighbor-Interpolation (oben) und durch das beschriebene Abstimmungsverfahren (unten). [SHR10]

wie der Umgebungsverdeckung benötigt werden, erfolgt ihre Generierung bereits unmittelbar nach der direkten Beleuchtung in einem einzelnen zusätzlichen Pass (mit einem Render-Pass pro Level). Im Gegensatz dazu wird die MIP-Map der eingehenden Beleuchtung direkt mit hardwarebeschleunigtem Blitting und linearer Interpolation berechnet, da die hierdurch entstehende Weichzeichnung bei den Farbwerten weniger problematisch und in vielen Fällen (Reflexionen) sogar erwünscht ist. Die MIP-Map der eingehenden Beleuchtung wird außerdem auch erst zu Beginn des GI-Passes berechnet, da in diese noch die reprojezierten Ergebnisse der Beleuchtung des letzten Frames einfließen müssen.

Analog zur in Kapitel 3.6.1 beschriebenen Implementierung des Upsampling-Verfahrens, wurde auch beim MIP-Mapping versucht über Vektorisierung so parallel wie möglich zu arbeiten. Darüber hinaus wurde hier außerdem eine zusätzliche Approximation verwendetet, indem nicht jeder umgebende Punkt mit dem aktuellen abgeglichen wird. Statt dessen wird ein Durchschnittswert für den Tiefenwert und die Achsen der kodierten Normalen bestimmt und nur dieser mit dem Punkt abgeglichen. Der relevante Code dieses Shaders befindet sich ebenfalls in Anhang D.1.

3.6.3 Spekulare Beleuchtung

Analog zu Kapitel 3.6.1 ist die Aufgabe dieses Teil-Passes die Lösung von Gleichung 2.15. Wie bereits in Kapitel 2.2 erläutert, kommt hierzu ein Verfahren aus dem Bereich des Image-Based-Lighting zur Anwendung, bei dem die Lösung für einen Teil des Integrals (f_{brdf}) vorberechnet wird. Hierzu werden die Textur-Koordinaten (X = $N \bullet V$, Y = Rauheit) als Eingabewerte interpretiert, das Integral in einem Fragment-Shader für jeden Eingabewert mittels Importance-Sampling gelöst und anschließend in einer 512×512 großen RG-Textur (16 Bit pro Farbkanal) gespeichert. Für jeden Punkt in dieser Textur werden 1024 Richtungen basierend auf einer Hammersley-Sequenz bestimmt, der relevante Teil der BRDF gelöst und der Durchschnitt über alle Teilergebnisse berechnet. Diese Textur wird bei Programmstart einmal vorberechnet und kann anschließend bei der finalen Kombination der indirekten Beleuchtung ausgelesen und mit den zur Laufzeit bestimmten Beleuchtungsdaten kombiniert werden. [Kar13]

Ähnlich wie beim diffusen Teil erfolgt auch hier die Bestimmung der Beleuchtungsinformationen über das Sammeln von Samples im Screen-Space Eingabe-Buffer. Im Unterschied dazu ist der relevante Bereich, aus welchem die Samples bezogen werden, bei nahezu perfekten Spiegelungen deutlich kleiner, das Ergebnis allerdings folglich auch hochfrequenter. Entsprechend kommt hierfür ein Raycasting-Verfahren im Screen-Space zur Anwendung. Da dies für zunehmend raue Oberflächen zu mehr Fehlern und zusätzlichem Rechenaufwand durch einen größeren relevanten Bereich führt, wird ab einer gewissen Rauheit und Distanz zwischen Reflektor und reflektierter Oberfläche abgebrochen und der diffuse Beleuchtungswert (skaliert mit $\frac{1}{4\pi^2}$) als Approximation verwendet.

Der erste Schritt ist die Berechnung des Schnittpunktes des an der Normalen des aktuellen Punktes reflektierten Sichtstahls mit der Szene, mittels des optimierten Ray-Tracing-Verfahrens von McGuire und Mara [MM14]. Anschließend wird aus dem Abstand zwischen dem Ausgangs- und dem Schnittpunkt sowie der Rauheit am Ausgangspunkt die Größe der reflektierten Fläche im Screen-Space nach Hermanns, Franke und Kuijper [HFK16] approximiert. Für diese wird anschließend das MIP-Level abgeschätzt, aus dem das Sample gelesen werden muss um einen entsprechend großen Bereich abzudecken. [HFK16]

Eine bessere Approximation wäre z.B. durch das Screen-Space Cone-Tracing Verfahren von Hermanns, Franke und Kuijper [HFK16] möglich und würde vor allem bei raueren Oberflächen und Szenen mit hoher Tiefenkomplexität bessere Ergebnisse liefern. Dies wurde allerdings in der vorliegenden Anwendung, auf Grund des damit verbundenen erhöhten Komplexität, nicht umgesetzt. Statt dessen wird nach der Berechnung der Reflexionen noch ein 3×3-Weichzeichner angewendet, der die meisten Artefakte ausreichend gut verschleiert. Der Rechenaufwand für diesen sowie für das Ray-Tracing selbst, fallen relativ gering aus, da sie in einer geringeren Auflösung durchgeführt und erst im letzten Schritt auf die finale Auflösung hoch skaliert werden.

3.7 Temporale Kantenglättung

Wie viele andere Lösungen für globale Beleuchtung leidet auch dieses unter dem Problem, dass bei einer zu geringen Anzahl von Samples Artefakte wie Rauschen und Flackern auftreten, die Anzahl der Samples allerdings einen erheblichen negativen Einfluss auf die Performance hat. Dieses Problem besteht ebenfalls bei der Schattenberechnung für die direkte Beleuchtung sowie allgemein bei der Rastarisierung und Beleuchtung[32]. Vor allem in den letzten beiden Fällen werden diese Undersampling-Artefakte (Aliasing) i.d.R. durch Einsatz eines entsprechenden Kantenglättungs oder Anti-Aliasing (AA) Verfahrens bereinigt.

Eine häufige Lösung für Aliasing an Kanten von Objekten ist Multisample Anti-Aliasing (MSAA), welches allerdings einen nicht unerhebliche zusätzlichen Rechenaufwand (u.a. bei Speicher und Bandbreite) nach sich zieht und bei Deferred-Shading nicht trivial zu implementieren ist. Eine effiziente Alternative hierzu

[32] Z.B. Flackern von spekularen Highlights unter Bewegung.

stellen Post-Effekt basierte AA-Verfahren wie Fast Approximate Anti-Aliasing (FXAA) dar. Diese braucht zwar nur geringfügige zusätzliche Ressourcen, sind i.d.R. trivial umzusetzen und können im Gegensatz zu MSAA auch Aliasing innerhalb von Objekten glätten, entfernt allerdings u.U. auch erwünschte Details und können keine temporalen Aliasing-Artefakte wie Flackern bereinigen. Die Artefakte bei den Schatten und der indirekten Beleuchtung bleiben somit nach wie vor bestehen.

Die Lösung welche in dieser Implementierung gewählt wurde, ist der Einsatz von temporaler Kantenglättung bzw. Temporal Anti-Aliasing (TAA). Die Idee dieses Verfahrens ist es zusätzliche Samples über mehrere Frames zu verteilen um die oben genannten Undersampling-Artefakte zu lösen. Hierzu wird ein zusätzlicher History-Buffer benötigt, in welchem der letzte Frame (nach der Kantenglättung) gespeichert wird. In jedem Frame wird dann der aktuelle Frame mit diesem History-Buffer durch $c_t = c_t + (c_{t-1} - c_t) \cdot \tau$ verrechnet, wobei $0{,}88 \leq \tau \leq 0{,}97$ abhängig vom Unterschied der Helligkeit der beiden Bilder gewählt wird. Da die Eingabe c_{t-1} des aktuellen Frames die Ausgabe des letzten ist, wird dadurch eine exponentielle Glättung über alle vergangenen Frames realisiert. [Kar14; Ped16]

Um nicht nur eine allgemeine Glättung durchzuführen, sondern tatsächlich zusätzliche Sub-Pixel-Details zurück zu gewinnen, muss die Kameraperspektive bei jedem Frame um einen Sub-Pixel-Offset verschoben werden. Dieser wird zu Beginn des Frames mit der Projektionsmatrix verrechnet und bei der Kantenglättung wieder abgezogen. Dieser Offset kann prinzipiell beliebig gewählt werden, es ist allerdings wünschenswert sowohl räumlich als auch zeitlich eine gleichmäßige Verteilung der Samples zu gewährleisten um möglichst alle Bereiche gleichmäßig abzutasten und zusätzliche Artefakte zu vermeiden. Entsprechend der Empfehlung von Karis [Kar14] wird der Offset durch eine Halton-Sequenz mit Basis 2 und 3 aus $\mathbb{R}^2$ mit Komponenten im Intervall $(-0{,}125, 0{,}125)$ bestimmt.

Da der TAA-Filter vor dem Tone-Mapping angewendet wird, kommt es zu einem starken Überstrahlen heller Bereiche (Fireflies). Diese Artefakte entstehen dadurch das helle Bereiche bei der Berechnung des Durchschnitts einen höheren Einfluss haben und lassen sich somit weitestgehend lösen, indem die Farbwerte anhand ihrer Helligkeit[33] gewichtet werden. Diese Gewichtung erfolgt im TAA-Shader beim auslesen der Farbwerte mit $T(\text{farbe}) = \frac{\text{farbe}}{1+\text{helligkeit}}$ und beim Speichern des Ergebnisses mit $T^{-1}(\text{farbe}) = \frac{\text{farbe}}{1-\text{helligkeit}}$. [Kar14]

Das bisher geschilderte Verfahren funktioniert allerdings nur für statische Szenen, da ansonsten der korrespondierende Punkt im History-Buffer nicht mehr die selben Koordinaten wie der Punkt im aktuellen Frame hat und es somit zu Schlieren (Ghosting) kommt. Der erste Schritt zur Lösung dieses Problems ist es zusammen mit dem History-Buffer die letzte Kamera-Matrix (View-Matrix) zu speichern und mit dieser den aktuellen Punkt (rekonstruiert anhand der Tiefeninformationen aus dem G-Buffer) in die Vergangenheit zu projizieren. Um zusätzlich zu Kamerabewegungen auch für dynamische Objekte die korrekte Position im History-Buffer zu bestimmen, wird ein Velocity-Buffer benötigt der für jeden Pixel die aktuelle

[33] In diesem Fall berechnet mittels $lum(c) = \sqrt{0{,}229c_r^2 + 0{,}587c_g^2 + 0{,}114c_b^2}$

Geschwindigkeit enthält und somit eine Berechnung der letzten Position ermöglicht.[34]

Die beschriebene Reprojektion ist allerdings selbst in idealen Situationen nicht exakt[35]. Um unter Bewegung trotzdem Schlieren zu vermeiden, wird die Umgebung jedes Punktes im aktuellen Frame untersucht, das Minimum/Maximum der Farbwerte berechnet und der Wert aus dem History-Buffer auf diesen Wertebereich beschränkt. [Kar14; Ped16]

Wie bei den meisten verbreiteten Anti-Aliasing-Verfahren ist auch bei der implementierten TAA-Variante ein gewisser Verlust an Schärfe zu verzeichnen, dieser fällt gleichwohl deutlich geringer aus, als dies bei anderen verbreiteten Verfahren der Fall wäre. Die Berechnung ist indes verglichen mit z.B. FXAA auch relativ komplex und zeitaufwändig (siehe Kapitel 4.2). Außerdem belastet der bei dynamischen Objekten erforderliche Velocity-Buffer zusätzlich das Speicher- und Durchsatzbudget. Dieser Buffer ist allerdings ggf. auch für andere grafische Effekte wie Bewegungsunschärfe nutzbar, womit sich seine Kosten in gewisser Weise amortisieren.

Die Qualität des Ergebnisses spricht insgesamt für sich und ist den zusätzlichen Aufwand in diesem Fall mehr als Wert, zumal das Verfahren in vielen Bereichen gute Ergebnisse liefert in denen traditioneller Algorithmen wie MSAA und FXAA nicht anwendbar sind (z.B. temporal Artefakte). Darüber hinaus vereinfacht der Einsatz von TAA viele der anderen Verfahren, wie die Berechnung der indirekten Beleuchtung und der Schatten immens, da dort nun auch Verfahren und Optimierungen eingesetzt werden können, welche im Normalfall stark durch Rauschen und vergleichbare Artefakte belastet wären (siehe Abb. 3.9).

(a) Mit TAA (b) Ohne TAA (c) Differenz

Abb. 3.9: Vergleich des Endergebnisses mit (a) und ohne (b) die temporale Kantenglättung, sowie die verstärkte Differenz der beiden (c). Während die Unterschiede in dieser Szene als Standbild kaum ersichtlich sind, unterliegen die betroffenen Bereiche (die Übergänge der Schatten und Objektkanten) ohne TAA einem starken Flackern, welches vor allem unter Bewegung stark störend auffällt.

[34] In der aktuellen Implementierung gibt es keine dynamischen Objekte, weshalb die genannte Lösung über einen Velocity-Buffer nicht implementiert wurde.

[35] z.B. bei Flächen die im letzten Frame nicht sichtbar waren.

4

Auswertung

Zur Auswertung des entwickelten Verfahrens werden eine Reihe von Beleuchtungsszenarien im Sponza-Model[1] durch den entwickelten Renderer dargestellt und anschließend analysiert. Da der Fokus auf der Anwendbarkeit des Verfahrens in dynamischen Echtzeitanwendungen liegt, erfolgt die Bewertung basierend auf den Faktoren:

- **Bildqualität:** Unter anderem die physikalische Korrektheit der Ergebnisse, aber vor allem der subjektive optische Gesamteindruck. Da dieser im angestrebten Einsatzbereich i.d.R. von größerer Bedeutung ist.
- **Performance:** Die Ausführungsgeschwindigkeit des Verfahrens auf unterschiedlichen Hardware-Plattformen, welche maßgeblich für die praktische Verwendbarkeit in Echtzeitanwendungen ist.
- **Skalierbarkeit:** Untersuchung in wie weit die Performance auf schwächerer Hardware durch Reduktion der Bildqualität verbessert werden kann, damit das Verfahren auch auf dieser zum Einsatz kommen kann und trotzdem eine ähnliche Beleuchtung gewährleistet wird.

Soweit nicht anders angegeben erfolgten alle Messungen auf einem Computer mit einer aktuellen Kaby-Lake Intel CPU, 32 GiB Arbeitsspeicher und einer NVIDIA GeForce GTX 1060, 970 oder Intel HD Graphics 630. Hierdurch sowie durch die geringe CPU-seitige Komplexität der Anwendung wird somit garantiert, dass lediglich die jeweils verwendete GPU den limitierenden Faktor darstellt.

Zur Analyse der Laufzeit und Bildqualität erfolgt ein Vergleich mit dem physikalisch korrekten offline Renderer LuxRender[2] und der voxelbasierten Echtzeitlösung SEGI[3]. Auf Grund unterschiedlicher Materialmodelle ist ein exakter Vergleich der Ergebnisse allerdings nicht ohne weiteres möglich. Soweit nicht anders angegeben erfolgten alle Messungen mit der NVIDIA GeForce GTX 1060 und einer Auflösung von 1920×1080.

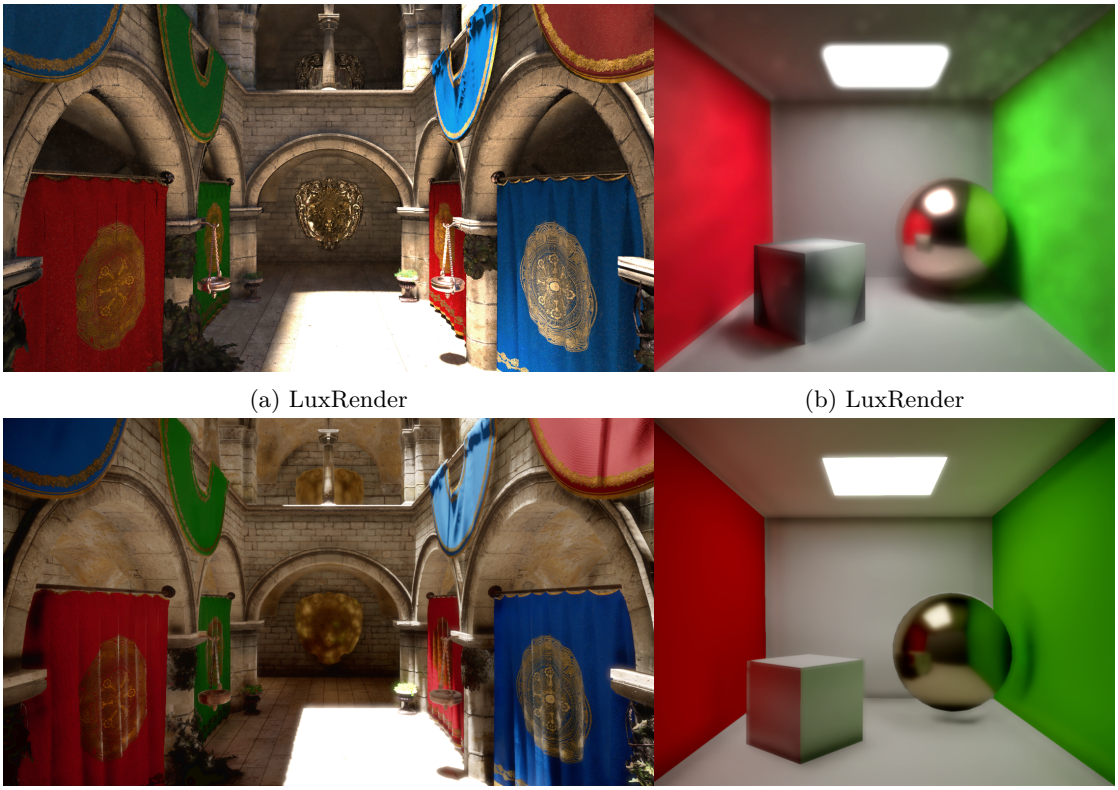
[1] Ein ursprünglich von Frank Meinl erstelltes und von Crytek erweitertes 3D-Modell vom Atrium des Sponza Palastes in Dubrovnik bestehend aus 279.163 Dreiecken. Das Model ist aufgrund seiner Architektur gut zur Evaluierung globaler Beleuchtungsverfahren geeignet. Da der entwickelte Renderer auf einem physikalischen Modell basiert waren darüber hinaus entstehende Material-Texturen notwendig, welche von Alexandre Pestana (`http://www.alexandre-pestana.com/pbr-textures-sponza/`) bezogen und entsprechend der spezifischen Anforderungen angepasst wurden.

[2] Unter Verwendung eines Metropolis Path-Tracers (Hybrid Path).

[3] `http://www.sonicether.com/segi`

4.1 Bildqualität

Die Qualität der Ergebnisse überzeugt in den meisten getesteten Situationen durch einen ansprechenden visuellen Gesamteindruck. Der Unterschied zu einer realistischen Darstellung ist allerdings erwartungsgemäß hoch (siehe Abb. 4.1 und Abb. 4.2). Die Unterschiede sind primär das angesprochene Fehlen indirekter Schatten, auflösungsbedingte Artefakte sowie Fehler auf Grund fehlender Informationen von Objekten außerhalb des Sichtbereichs.



(a) LuxRender (b) LuxRender

(c) Screen-Space Verfahren (d) Screen-Space Verfahren

Abb. 4.1: Vergleich der Ergebnisse des entwickelten Verfahrens mit durch Metropolis Path-Tracing berechneten Referenzen. Die Referenzbilder wurden mittels LuxRender innerhalb von drei Stunden auf dem Testsystem unter Verwendung der NVIDIA GTX 970 berechnet. Es ist zwar eine deutliche Ähnlichkeit zu erkennen, die aufgeführten Schwächen des Verfahrens sind allerdings eindeutig ersichtlich.

Das größte Problem stellt hierbei die fehlende Information zu Objekten außerhalb des Sichtfeldes dar. In Abb. 4.1 äußert sich dies primär durch das Fehlen von Reflexionen und der blauen Beleuchtung durch den Himmel, da dieser auf dem Bild nicht sichtbar ist. Darüber hinaus kommt es bei Bewegungen der Kamera mit unter zu relativ abrupten Änderungen der Beleuchtung, da direkt beleuchtete Flächen aus dem Sichtfeld verschwinden und auftauchen. Auch nimmt der Fehler bei der Beleuchtung wie in Abb. 4.4 massiv zu, je weniger von den direkt beleuchteten Flächen sichtbar ist.

(a) VXGI (b) VXGI

(c) Screen-Space Verfahren (d) Screen-Space Verfahren

Abb. 4.2: Vergleich der Ergebnisse des entwickelten Verfahrens mit denen des voxelbasierten Beleuchtungsverfahrens SEGI. Die sichtbaren Unterschiede ergeben sich hier, neben Abweichungen beim Tone-Mapping, wie auch bei Abb. 4.1 primär durch die fehlende Sichtbarkeitsprüfung und nicht sichtbare Informationen.

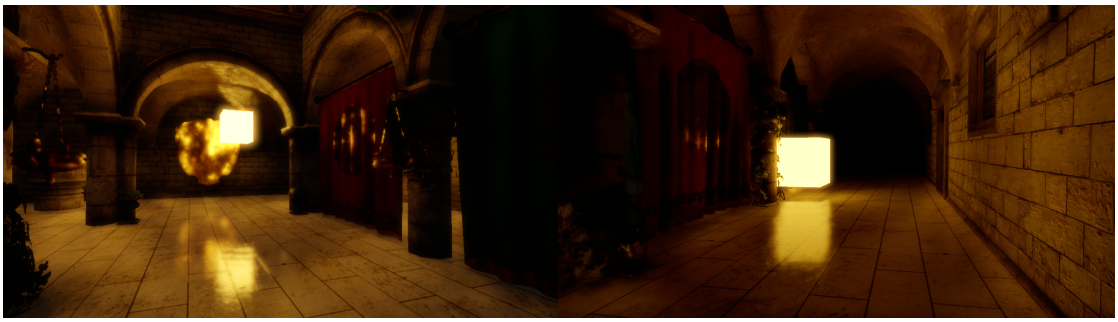


Abb. 4.3: Weitere Ergebnisse des entwickelten Verfahrens in einer Szene ohne analytische Lichtquelle analog zu Abb. 4.2.

Diese Probleme können zwar teilweise durch temporale Filterung und die Durchführung der Beleuchtungsberechnung mit einem höheren Sichtfeld abgeschwächt werden, sind allerdings spätestens bei kompletter Abkehr von den direkt beleuchteten Flächen offenkundig. Wenn sich die Lösung dieser Probleme nicht über eine Einschränkung der zulässigen Kameraperspektiven bewerkstelligen lässt, ist daher ein alternatives Verfahren für diese Fälle nötig. Dieses muss allerdings nicht zwingend im vollen Umfang dynamische Szenen unterstützen, da es lediglich ergänzend arbeitet. Hierzu kann während der Berechnung der indirekten Beleuchtung ein Konfidenzwert aus der Anzahl der erfolgreichen Samples berechnet werden. Mittels diesem kann anschließend im letzten Schritt linear z.B. zum Ergebnis sta-

tischer Lightprobes geblendet werden. Auch die Verwendung eines Deep G-Buffer Verfahrens wie von Mara, McGuire, Nowrouzezahrai und Luebke [MMN+16] vorgeschlagen wäre als Lösung dieses Problems denkbar, deckt allerdings nur einen Teil der problematischen Fälle ab und ist, durch die Notwendigkeit der Beleuchtung eines zusätzlichen G-Buffers, mit für viele Anwendungen zu hohem Rechenaufwand verbunden.

Die geringe Auflösung der intern verwendeten Texturen fällt vor allem bei Reflexionen auf metallischen Oberflächen störend auf, während sie bei der diffusen Beleuchtung nur vereinzelt ins Gewicht fällt. In Verbindung mit der Problematik der fehlenden Informationen ergibt sich allerdings eine ungleichmäßige Beleuchtung (siehe Abb. 4.4) und temporale Inkonsistenzen. Diese können zumindest zum Teil durch die temporale Kantenglättung gelöst werden, auch wenn hier zwischen diesen GI-Artefakten und denen der TAA (Verlust scharfer Kanten, Schlieren unter Bewegung und Nachleuchten von Flächen bei Beleuchtungsänderungen) abgewogen werden muss.

(a) (b)

Abb. 4.4: Ein Beispiel für Artefakte durch unvollständige Informationen zur Szene in Verbindung mit einer zu geringen Auflösung (a). Diese werden vor allem in Szenen ohne oder mit nur geringer direkter Beleuchtung deutlich und äußern sich zu meist in Form von Wolkenmustern. Besonders störend ist dabei die Inkonsistenz dieser Muster unter Bewegung. In Abb. (b) wird der Effekt durch Variation der Größe des Emitters in der Cornell-Box verdeutlicht.

Die Abwesenheit indirekter Schatten äußert sich in Abb. 4.1 primär durch die zu hellen Bereiche im Gang auf der linken und rechten Seite, welche eigentlich durch die Vorhänge von der Einstrahlung vom Boden aus abgeschattet sein müssten. Während kleinere Schatten – wie z.B. an Kanten und hinter den oberen Fahnen – gut durch Umgebungsverdeckungsverfahren wie das implementierte SAO approximiert werden können, ist dies für die größeren Schatten nicht ohne weiteres möglich. Das von Soler, Hoel und Rochet [SHR10] vorgestellte Verfahren schlägt hierfür eine Voxelisierung der Szene in Verbindung mit Raycasts zur Prüfung der Sichtbarkeit vor. Dies ist allerdings in der Praxis mit einem zu hohem Rechenaufwand verbunden. Um die Problematik der fehlenden Schatten in solchen Fälle trotzdem abzuschwächen, wäre ein Ansatz denkbar, bei dem die Sichtbarkeitsprüfung nur auf höheren MIP-Leveln bei der Berechnung der diffusen Beleuchtung

und somit seltener pro Frame erfolgt. Dieser Ansatz erfordert allerdings nach wie vor eine Voxelisierung, was gerade bei komplexeren Szenen mit einem nicht unerheblichen Rechenaufwand verbunden ist. Eine für viele Anwendungsfälle denkbare Alternative wäre es außerdem, die Sichtbarkeitsprüfung gegen eine stark vereinfachte Szene (z.B. eine Menge von 2D-Flächen oder 3D-Boxen) durchzuführen um zumindest die größten Schatten zu simulieren. [SHR10]

Abschließend ist die Bildqualität für die meisten Anwendungsfälle mehr als ausreichend und eine deutliche Verbesserung im Vergleich mit lokalen Beleuchtungsverfahren und einer konstanten Umbgebungsbeleuchtung (Ambient-Light). Für Szenen mit einer hohen Tiefenkomplexität oder einer frei beweglichen Kamera ist allerdings u.U. ein zusätzliches Beleuchtungsverfahren nötig um in allen Situationen eine zufriedenstellende Qualität zu gewährleisten.

4.2 Performance

Die Bewertung des Laufzeitverhaltens erfolgt analog zur Bildqualität über die Darstellung der Sponza- und Cornell-Box-Szene. Hierbei werden die Laufzeiten pro Frame sowie pro Teilschritt (Pass) über Timestamp-Queries der Grafikkarte gemessen, aggregiert und unter anderem mit denen des voxelbasierten SEGI-Verfahrens verglichen.

Um einen möglichst vollständigen Überblick über das Laufzeitverhalten in verschiedenen Situationen und Szenen zu erhalten, wird während der Messungen eine automatische Kamera-Animation durchlaufen. Des Weiteren wird das beschriebene Szenario mit unterschiedlichen Auflösungen und Grafikkarten wiederholt um das allgemeine Verhalten des Verfahrens, sowie seine unteren und oberen Laufzeitschranken, besser abschätzen zu können.

Um eine Grundlinie für die folgenden Analysen zu etablieren erfolgte als erstes eine Messung der Laufzeiten unter Verwendung der NVIDIA GeForce GTX 1060 bei einer Auflösung von 1920×1080, deren Ergebnisse in Abb. 4.5 aufgearbeitet sind. Die gemessenen Werte sind dort in die wesentlichsten Teilschritte aufgeteilt, die in Kapitel 3 erläutert wurden. Nennenswert ist hierbei lediglich die Zusammenfassung aller kleineren Teilschritte (Tonemapping, Bloom sowie die finale Zusammenfassung aller Teilergebnisse) in den Eintrag *Post-Effekte* sowie die Zusammenfassung der einzelnen MIP-Mapping-Schritte in einen.

Wie aus dem Graph in Abb. 4.5 hervorgeht, ist die Berechnung der MIP-Maps und globalen Beleuchtung mit einer Schwankung von maximal 0,6 ms deutlich unabhängiger von der dargestellten Szene als die direkte Beleuchtung. Diese Schwankungen treten primär aufgrund kürzerer und längerer Pfade während des Raycastings für die spekularen Reflexionen sowie generelle Unterschiede bei der Ausführung (Cache-Misses, unterschiedliche Code-Pfade) auf. Die auffällig starken Schwankungen bei der direkten Beleuchtung beruhen darauf, dass die aufwändigen Beleuchtungsberechnungen nur bei Bereichen durchgeführt werden, die nicht im Schatten liegen sowie darauf, dass bei Bewegungen der Lichtquelle eine aufwändige Neuberechnung der Shadowmaps erforderlich wird. Bei einer konstanten

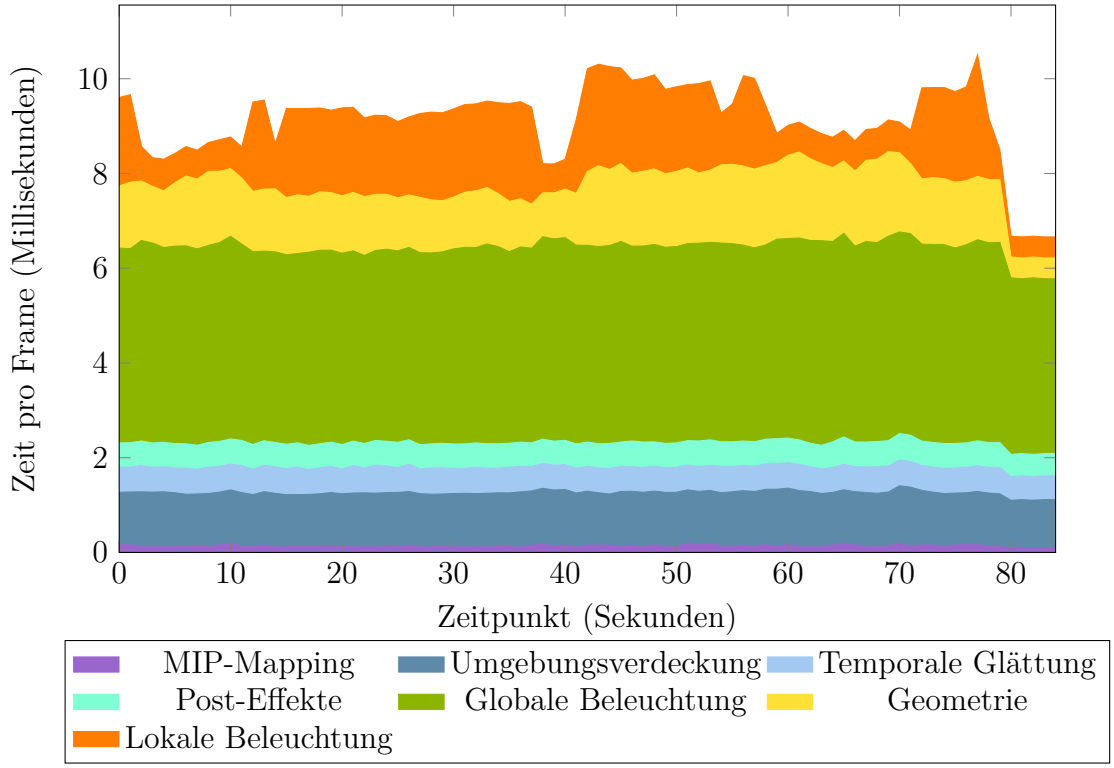


Abb. 4.5: Ein Graph der Berechnungszeit pro Frame im zeitlichen Verlauf, aufgeteilt in die wichtigsten Teilschritte. Die ersten 80 Sekunden der Messwerte umfassen die automatische Kamerafahrt, gefolgt von der statischen Darstellung der Cornell-Box. Während der Kamerafahrt betrug die durchschnittliche Laufzeit pro Frame 9,3 ms, was etwa einer Bildrate von 108 FPS entspricht.

Auflösung, verhält sich das Verfahren somit relativ stabil, macht allerdings inkl. der Umgebungsverdeckung etwa 58 % der benötigten Laufzeit aus.

Der Graph in Abb. 4.6 zeigt die Laufzeiten in der selben Situation aber mit unterschiedlichen Auflösungen. Da die direkte und ein geringer Teil der indirekten Beleuchtung stark von der verwendeten Auflösung abhängen, sind die Messwerte hier zusätzlich in die globale Beleuchtung sowie die Bestimmung der eingehenden diffusen und spekularen Beleuchtung getrennt.

Wie der Graph in Abb. 4.6 verdeutlicht, besteht zwischen der Verwendeten Auflösung und der Berechnung der globalen Beleuchtung eine deutlich weniger stark ausgeprägte Beziehung als beim Rest des Renderers. Lediglich bei der mittleren Auflösung von 1440p liegen die Werte deutlich über denen der Grundlinie. Dieses Phänomen entsteht dadurch, dass die intern für die Berechnung der globalen Beleuchtung verwendeten Texturen eine feste Größe von ca. 540p haben, welche erst am Ende des Prozesses auf die tatsächlich benötigte Auflösung hochgerechnet werden. Um hierbei ein artefaktfreies Ergebnis zu gewährleisten muss die endgültige Auflösung allerdings ein glattes Vielfaches der intern verwendeten sein, was in diesem Fall nur entweder 720p oder 480p als internes Format zulässt. Standardmäßig wird hierbei die höhere der beiden Möglichkeiten bevorzugt, da es bei niedrigeren Auflösungen vermehrt zu Artefakten kommt. Dies kann allerdings über

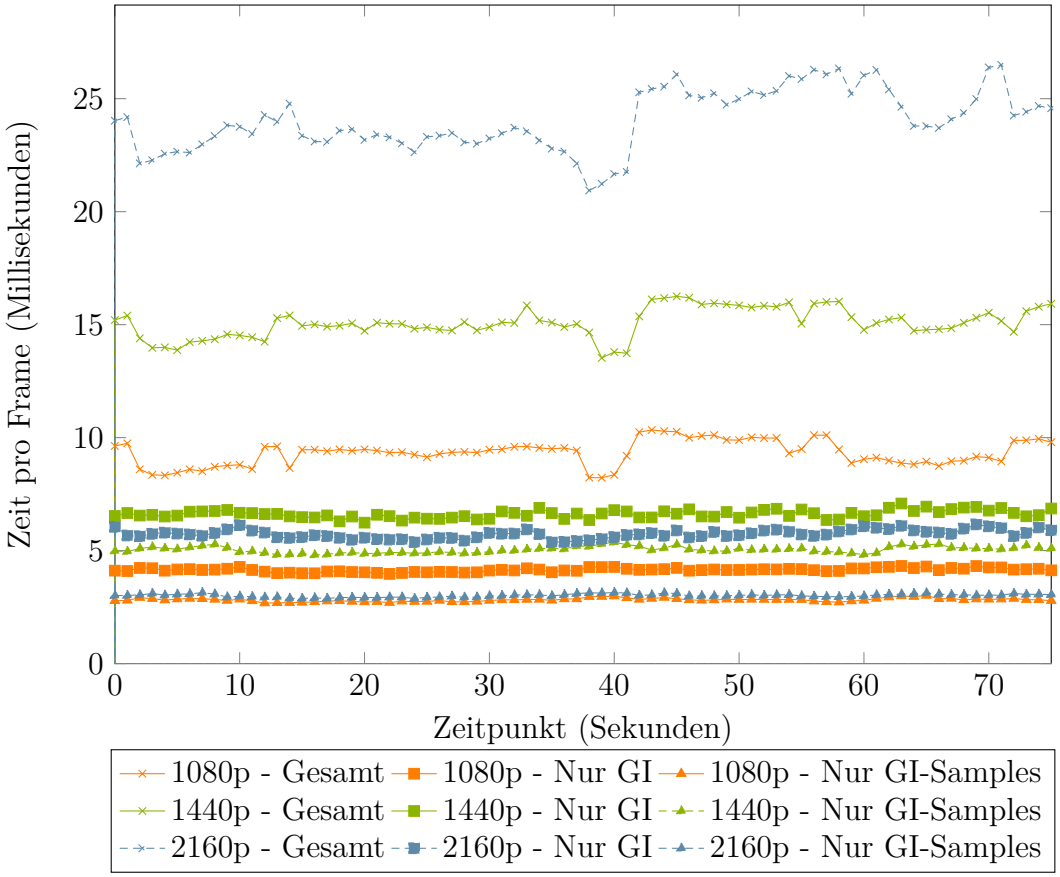


Abb. 4.6: Graph der Berechnungszeit pro Frame im zeitlichen Verlauf bei Bildschirmauflösungen von 1920×1080, 2560×1440 und 3840×2160.

einen Konfigurationsparameter variiert werden, wodurch die Laufzeit des GI-Teils nahezu der bei 1080p entspricht.

Um in gewissem Maße auch auf das Laufzeitverhalten unter Verwendung anderer Grafikkarten schließen zu können, erfolgt zuletzt in Abb. 4.7 ein direkter Vergleich mit einer Mittelklasse-Grafikkarte der letzten Generation (NVIDIA GeForce GTX 970). Die Laufzeiten des GI-Teils befinden sich insgesamt in einer sehr ähnlichen Größenordnung. Die Teilweise erheblichen Abweichungen bei der Gesamtlaufzeit ergeben sich im wesentlichen durch Performanceunterschiede der beiden Grafikkarten beim Aktualisieren der Shadowmap, wie an der hohen Ähnlichkeit der Werte in Bereichen ohne Bewegungen der Lichtquelle zu erkennen ist.

Ein direkter Vergleich mit SEGI ist aufgrund der unterschiedlichen Architektur[4] nur schwer möglich. Nach intensiven Tests haben sich auf dem Testsystem mit einer GTX 1060 und einer Auflösung von 1920×1080 allerdings Bildraten von etwa 54 FPS auf hoher und 86 FPS auf niedriger Detailstufe und in einer ähnlichen Szene ergeben. Dies entspricht Laufzeiten pro Frame von ungefähr 18 ms bzw. 12 ms und

[4] Derzeit ist SEGI noch eine Closed-Source Unity Erweiterung, von der z.Z. nur eine unter Windows lauffähige Demoversion bezogen werden kann.

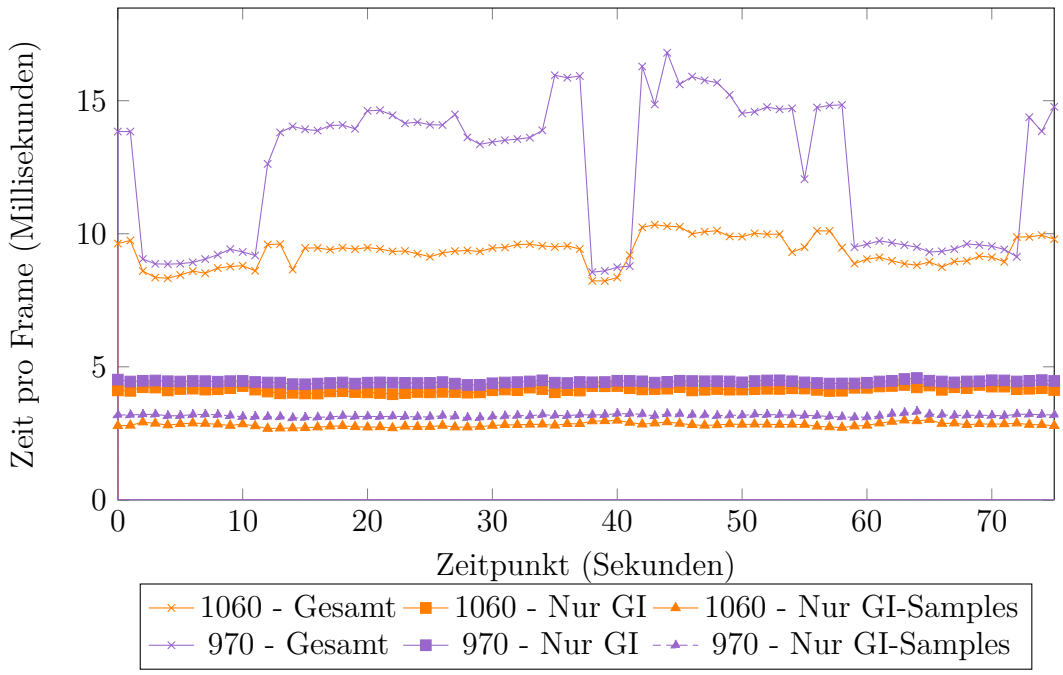


Abb. 4.7: Graph der Berechnungszeit pro Frame im zeitlichen Verlauf unter Verwendung einer GTX 1060 und einer GTX 970.

liegt somit erwartungsgemäß etwas über den 9,3 ms des hier vorgestellten Screen-Space-Verfahrens.

Wie aus den oben genannten Daten hervorgeht, ist das Verfahren zwar mit nicht unerheblichem Rechenaufwand verbunden, im Vergleich mit anderen Verfahren allerdings noch relativ effizient. Auch ist die Laufzeit nahezu unabhängig von der Komplexität der Dargestellten Szene und nur in geringem Maße abhängig von der Bildschirmauflösung. Zwar lässt sich auf Grund der geringen Menge getesteter Hardware keine verlässliche Aussage diesbezüglich treffen aber die gemessenen Daten stellen dennoch in Aussicht, dass das Verfahren auf einer größeren Menge unterschiedlicher Hardware erfolgreich eingesetzt werden kann.

4.3 Skalierbarkeit

Wie aus dem vorherigen Kapitel hervorgeht ist die Laufzeit des Verfahrens weitestgehend unabhängig von der verwendeten Auflösung und der dargestellten Szene. Damit ist die wichtigste Möglichkeit zur Skalierung des Verfahrens auf leistungsschwächerer Hardware das Variieren der Grafikqualität.

Um die Optionen mit dem größten Effekt auf die Laufzeit und gleichzeitig möglichst geringen Auswirkungen auf die Grafikqualität zu identifizieren, wurden in Abb. 4.8 die relevanten Optionen in der Sponza-Szene mit unterschiedlichen Auflösungen und Grafikkarten getestet sowie in Abb. 4.9 auf ihrer Qualitätsunterschiede hin untersucht. Hierbei stellt Konfiguration Nr. 19 die Standardkonfiguration dar, mit der die bisherigen Messungen und Abbildungen erzeugt wurden.

Nummer	MIP-Mapping	GI-Auflösung	Anzahl Samples	Gesamt (NVIDIA 1440p)	MIP-Mapping (NVIDIA 1440p)	Globale Beleuchtung (NVIDIA 1440p)	Gesamt (NVIDIA 1080p)	MIP-Mapping (NVIDIA 1080p)	Globale Beleuchtung (NVIDIA 1080p)	Gesamt (Intel 720p)	MIP-Mapping (Intel 720p)	Globale Beleuchtung (Intel 720p)
1	**schnell**	**niedrig**	**16**	10,4	0,3	2,5	6,8	0,2	2,2	30,8	0,6	8,8
2			**32**	10,5	0,3	2,6	6,9	0,2	2,3	31,0	0,6	9,0
3			**64**	10,6	0,3	2,7	7,0	0,2	2,4	31,4	0,6	9,4
4			**128**	10,7	0,3	2,8	7,2	0,2	2,6	32,3	0,6	10,3
5			**256**	11,0	0,3	3,1	7,8	0,2	3,2	33,9	0,6	12,0
6		**hoch**	**16**	12,3	0,4	4,2	7,0	0,2	2,4	45,5	0,7	23,5
7			**32**	12,8	0,4	4,6	7,2	0,2	2,6	49,3	0,7	27,3
8			**64**	13,7	0,4	5,5	7,8	0,2	3,1	56,5	0,7	34,5
9			**128**	15,6	0,4	7,4	8,8	0,2	4,1	70,6	0,7	48,6
10			**256**	19,0	0,4	10,9	10,8	0,2	6,2	97,3	0,7	75,3
11	**präzise**	**niedrig**	**16**	10,7	0,4	2,5	6,9	0,3	2,2	31,3	1,1	8,9
12			**32**	10,8	0,4	2,6	6,9	0,3	2,3	31,5	1,1	9,1
13			**64**	10,9	0,4	2,7	7,0	0,3	2,4	31,9	1,1	9,5
14			**128**	11,0	0,4	2,8	7,3	0,3	2,7	32,7	1,1	10,3
15			**256**	11,2	0,4	3,0	7,8	0,3	3,2	34,3	1,1	11,9
16		**hoch**	**16**	12,5	0,5	4,2	7,0	0,3	2,4	46,1	1,2	23,6
17			**32**	12,9	0,5	4,6	7,2	0,3	2,6	49,8	1,2	27,2
18			**64**	13,8	0,5	5,5	7,8	0,3	3,1	56,9	1,2	34,4
19			**128**	15,6	0,5	7,3	8,8	0,3	4,2	70,9	1,2	48,5
20			**256**	19,1	0,5	11,0	10,8	0,3	6,2	97,7	1,2	75,2

Abb. 4.8: Auflistung der durchschnittlichen Laufzeiten in Millisekunden, an Preset-Position *Center*, mit unterschiedlichen Qualitätseinstellungen und sowohl mit einer NVIDIA GeForce GTX 1060 als auch einer Intel HD Graphics 630. Die beiden MIP-Mapping-Verfahren bezeichnen das schnelle Nearest-Neighbor-Verfahren und das in dieser Arbeit vorgestellte präzisere abstimmungsbasierte Verfahren. Die Spalte *GI-Auflösung* umfasst die beiden Konfigurationsparameter *High-Resolution GI* und *Diffuse GI MIP* in der Benutzeroberfläche und beschreibt dort die Werte *Aktiviert* und *Level 1* für *hoch* sowie *Deaktiviert* und *Level 2* für *niedrig* (siehe Anhang B zu Details der Anwendungskonfiguration).

Wie aus den Daten in Abb. 4.8 und Abb. 4.9 hervorgeht, ist der Performance-Unterschied zwischen dem präziseren abstimmungsbasiertem und dem schnelleren hardwarebeschleunigten MIP-Mapping-Verfahren auf leistungsfähigeren Grafikkarten zu vernachlässigen ($\leq 0{,}2$ ms). Außerdem wird selbst dieser geringfügige Unterschied in vielen Fällen vollständig durch die parallele Verarbeitung auf der Grafikkarte ausgeglichen. In Anbetracht der minimal besseren Qualität und höheren Stabilität, stellt das schnellere MIP-Mapping-Verfahren daher nur auf sehr schwacher Hardware eine sinnvolle Option da.

Im Gegensatz dazu hat die GI-Auflösung einen sehr starken Einfluss auf die Laufzeit von bis zu einem Faktor von 3. Die Auswirkungen auf die Bildqualität sind allerdings leider ebenso deutlich. Vor allem bei komplexeren reflektierenden Objekten wie dem Löwenkopf fällt die geringere Auflösung der spekularen Reflexionen deutlich auf und verursacht mitunter auffällig zu helle Ergebnisse. Wenn solche Fälle ausgeschlossen werden können, ermöglicht diese Option allerdings auch flüssige Bildraten auf leistungsschwachen Grafikkarten, für die etablierte Verfahren keine Lösung bieten. Potentiell ist es auch möglich nur die Auflösung der diffusen Einstrahlung zu reduzieren[5], was die genannten Artefakte behebt. Die Geschwindigkeitsvorteile fallen damit allerdings auch geringer aus, da das Raycasting und die Kombination der Beleuchtungsinformationen damit nach wie vor in einer höheren Auflösung stattfindet.

Über die Variation der Anzahl von Samples für die diffuse Beleuchtung lässt sich – vor allem bei höherer GI-Auflösung – ebenfalls einiges an Laufzeit einsparen. Unterhalb von 64 Samples bei niedriger bzw. 32 Samples bei hoher GI-Auflösung kommt es allerdings zu stark ausgeprägtem, temporal instabilem Rauschen. Dieses fällt vermehrt in Szenen mit besonders glatten Oberflächen wie der Cornell-Box auf. Abhängig von der konkreten Szene können also auch niedrigere Werte sinnvoll sein. Vor allem bei zu geringer direkter Beleuchtung kann das entstehende Rauschen allerdings auch hier störend auffallen.

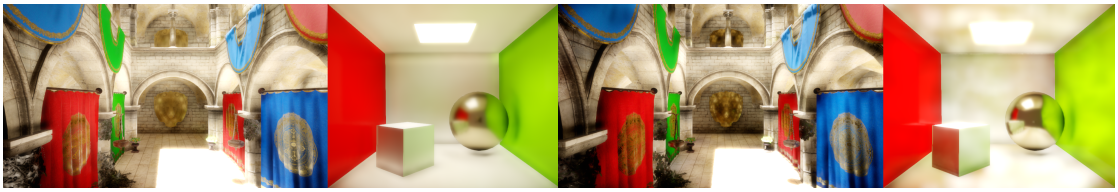
Das vorgestellte Verfahren bietet eine Vielzahl von Konfigurationsmöglichkeiten, welche mit einigen Abstrichen in der Grafikqualität auch den Betrieb auf leistungsschwacher Hardware erlauben. Die niedrigste noch akzeptable Einstellung Nr. 3 (siehe Abb. 4.8) erlaubt mit 31,4 ms (31,85 FPS) selbst noch die Verwendung von Intel-Grafikkarten in interaktiven Anwendungen wie Computerspielen. Mit zusätzlichen Optimierungen und Abstrichen bei anderen Grafikeffekten[6] sollten diese Bildraten auch zuverlässig erreicht werden können.

Nach oben sind prinzipiell zwar keine Grenzen gesetzt, was die Auflösung und Anzahl von Samplen angeht, oberhalb von 128 Samples bei hoher GI-Auflösung sind die Qualitätsverbesserungen allerdings kaum erkennbar. Das beste Verhältnis zwischen Bildqualität und Laufzeit bietet somit aktuell Konfiguration Nr. 19 (siehe Abb. 4.8). Mehr Vorteile würden sich hier ggf. eher durch Verwendung von aufwändigeren Methoden zur Berechnung indirekter Schatten und Kompensation von Artefakten durch nicht sichtbare Objekte ergeben.

[5] Das heißt bei niedriger *Diffuse GI MIP* aber aktivierter *High Resolution GI*.
[6] Wie z.B. Umgebungsverdeckung und Bloom.

(a) Einstellung Nr. 1 (b) Einstellung Nr. 2

(c) Einstellung Nr. 3 (d) Einstellung Nr. 4

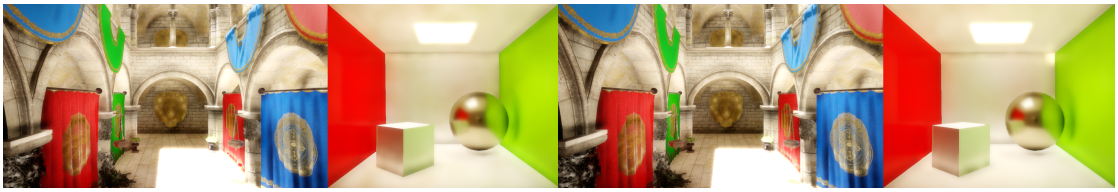


(e) Einstellung Nr. 5 (f) Einstellung Nr. 6

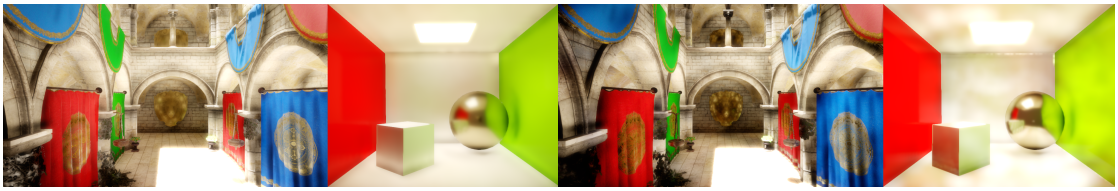
(g) Einstellung Nr. 7 (h) Einstellung Nr. 8

(i) Einstellung Nr. 9 (j) Einstellung Nr. 10

(k) Einstellung Nr. 11 (l) Einstellung Nr. 12



(m) Einstellung Nr. 13 (n) Einstellung Nr. 14



(o) Einstellung Nr. 15 (p) Einstellung Nr. 16

(q) Einstellung Nr. 17 (r) Einstellung Nr. 18

(s) Einstellung Nr. 19 (t) Einstellung Nr. 20

Abb. 4.9: Jeweils ein Bild der dargestellten Sponza-Szene und einer Cornell-Box mit den Konfigurationen aus Abb. 4.8.

5

Fazit

Insgesamt bietet das hier vorgestellte Verfahren eine gute Alternative zu aufwändigeren dynamischen Algorithmen wie Voxel-Cone-Tracing und effizienteren aber statischen Lösungen wie Light-Probes. Dies gilt allerdings nur so lange ein ausreichend großer Teil der Szeneninformationen im aktuellen Frame sichtbar sind. Für Einsatzszenarien mit einer relativ frei vom Benutzer steuerbaren Kamera – wie bei First- und Third-Person Spielen – ist daher ein zusätzliches Verfahren als Fallback notwendig, was allerdings den Nutzen des Verfahrens etwas in Zweifel zieht.

Bei einer eingeschränkteren Perspektive, wie dies z.B. bei Side-Scroller Spielen der Fall ist, stellt sich die Situation dagegen deutlich positiver dar. Hierbei liegt das primäre Problem bei dem Fehlen von indirekten Schatten. Dies fällt potentiell auch durch die häufig auftretende gleichzeitige Darstellung unterschiedlicher, getrennter Bereiche verstärkt negativ auf. Die in Kapitel 4.1 vorgeschlagenen Lösungen sollten allerdings in den meisten Fällen ausreichen um ein zufriedenstellendes Ergebnis zu gewährleisten.

Der ideale Einsatzbereich liegt somit bei jenen Anwendungen, bei denen stets alle relevanten Teile der Szene sichtbar sind und sich vor allem nichts hinter der Kamera befinden kann, wie dies z.B. bei Spielen in der Top-Down-Perspektive der Fall ist. Die größten Vorteile bestehen hier bei dynamischen und komplexen Szenen[1] (z.B. Strategiespiele), welche bei Verfahren, die auf einer Voxelisierung der Szene basierende, mitunter besondere Probleme bereiten.

Ein letztes interessantes Szenario stellt der Einsatz des Verfahrens im Bereich der Augmented-Reality dar. Um dafür zu sorgen, dass sich digitale Objekte natürlich in die reale Szene einfügen, ist eine realistische Beleuchtung hier unerlässlich. Allerdings muss diese mit einer relativ geringen Informationsbasis erfolgen, da aktuelle Systeme i.d.R. lediglich eine Farbaufnahme der Umgebung und Abstandsdaten liefern [Goo17]. Diese Einschränkungen entsprechen somit weitestgehend denen des hier vorgestellten Beleuchtungsverfahrens. Anhand der Abstandswerte lässt sich ein Tiefenpuffer bestimmen und mittels Differenzialquotienten die Oberflächennormalen approximieren. Die Gewinnung der Materialeigenschaften der realen Umgebung stellt hierbei evtl. noch ein Problem dar, ist allerdings nicht zwingend erforderlich, sofern auf eine indirekte Beleuchtung der Umgebung durch digitale

[1] Das heißt viele Objekte mit einer hohen Anzahl von Polygonen.

Objekte verzichtet werden kann und lediglich die digitalen Objekte selbst beleuchtet werden sollen. Auch gibt es bereits Ansätze zur approximativen Extraktion der relevanten Informationen [KK13]. Darüber hinaus wäre es denkbar die reale Beleuchtung durch eine Analyse der Kamera-Historie z.B. als Spherical-Harmonics zu approximieren um auch die problematischen Fälle aus Kapitel 4.1 abzudecken und so eine relativ realitätsgetreue Beleuchtung zu erreichen.

Wie in den Kapiteln 4.2 und 4.3 gezeigt wurde, bietet das Verfahren eine, vor allem im Vergleich mit anderen etablierten dynamischen GI-Verfahren, gute Laufzeit und ist mit einigen Abstrichen bei der Grafikqualität auch auf leistungsschwacher Hardware einsetzbar. Durch weitere Optimierungen sollte es möglich sein, die Laufzeit weiter zu verbessern und somit die nötigen Qualitätsabstriche zu reduzieren.

Aufbauend auf dieser Arbeit sind eine Vielzahl von Erweiterungen denkbar um die Grafikqualität weiter zu verbessern. Die erste dieser Erweiterungen – welche aus Zeitmangel nicht umgesetzt wurde – ist die Berechnung der Reflexionen über Screen-Space Cone-Tracing nach Hermanns, Franke und Kuijper [HFK16], statt wie aktuell lediglich das finale Sammeln der Lichteinstrahlung basierend auf der Rauheit und Entfernung durchzuführen. Hierdurch ließen sich u.U. gerade bei einer höheren Tiefenkomplexität bessere Ergebnisse erzielen.

Auch die Implementierung einer Sichtbarkeitsprüfung – z.B. wie in Kapitel 4.1 vorgeschlagen – wäre noch eine sinnvolle Erweiterung zur Verbesserung des Gesamteindrucks. Dies stellt zwar u.a. auf Grund von Performanceerwägungen ein nicht unerhebliches Problem dar, könnte aber zumindest auf leistungsstärkerer Hardware praktikabel sein.

Des Weiteren wäre es evtl. möglich die Qualität der diffusen indirekten Beleuchtung durch eine bessere Wahl der Samples weiter zu erhöhen. Zum einen könnte hierzu die Anzahl der Samples pro Level variiert werden um mehr Samples auf den höheren Leveln zu sammeln, da diese hier durch die geringe Auflösung weniger problematisch sind und durch den größeren abgedeckten Bereich tendenziell mehr Einfluss auf das Endergebnis haben. Darüber hinaus sollte noch untersucht werden ob eine andere Quelle für die zufällige Verteilung der Samples (Blue-Noise statt White-Noise) zu rauschärmeren Ergebnissen beitragen könnte.

Der letzte Punkt, welcher wie bei den meisten Screen-Space basierten Verfahren, problematisch ist, ist die Unterstützung von transparenten Objekten. Da das Verfahren in seiner aktuellen Form lediglich auf Informationen im G-Buffer – der zu diesem Zeitpunkt keine transparenten Objekte enthält – aufbaut und eine separate Beleuchtung der transparenten Oberflächen (z.B. durch Depth-Peeling) auf Grund des Aufbaus des Verfahrens nicht in annehmbarer Zeit durchführbar wäre, gibt es hierfür keinen trivialen Lösungsansatz. Die einzige derzeit denkbare Lösung wäre es, die Pixel mit transparenten Oberflächen zu unterteilen in jene, welche den Großteil des Hintergrunds verdecken und jene für die dies nicht gilt. Die größtenteils opaquen Pixel könnten dann in den G-Buffer übertragen werden um zumindest eine approximative indirekte Beleuchtung für diese Bereiche zu erhalten, während die so verdeckten Pixel und die zu durchsichtigen Oberflächen über ein alternatives Verfahren beleuchtet werden.

Literaturverzeichnis

[CNS+11] Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green und Elmar Eisemann. Interactive Indirect Illumination Using Voxel Cone Tracing. *Computer Graphics Forum (Proceedings of Pacific Graphics 2011)*, 30(7), September 2011. URL: http://maverick.inria.fr/Publications/2011/CNSGE11b (besucht am 10. 09. 2017).

[CT82] Robert L. Cook und Kenneth E. Torrance. A Reflectance Model for Computer Graphics. *ACM Trans. Graph.*, 1(1):7–24, Januar 1982. ISSN: 0730-0301. DOI: 10.1145/357290.357293.

[Dui10] Haarm-Pieter Duiker. Filmic Tonemapping for Real-time Rendering, 2010. Siggraph 2010 Color Course.

[ED08] Elmar Eisemann und Xavier Décoret. Single-pass GPU solid voxelization for real-time applications. In *Proceedings of Graphics Interface 2008*, GI 2008, Seiten 73–80, Windsor, Ontario, Canada. Canadian Human-Computer Communications Society, 2008. ISBN: 978-1-56881-423-0. DOI: 10.20380/GI2008.10.

[Fer05] Randima Fernando. Percentage-Closer Soft Shadows, 2005. URL: http://developer.download.nvidia.com/shaderlibrary/docs/shadow_PCSS.pdf (besucht am 22. 06. 2017).

[Goo17] Google. Project Tango – Depth Perception, 2017. URL: https://developers.google.com/tango/overview/depth-perception (besucht am 12. 09. 2017).

[Heb16] Chris Hebert. Vulkan Memory Management, 2016. URL: https://developer.nvidia.com/vulkan-memory-management (besucht am 09. 06. 2017).

[HFK16] Lukas Hermanns, Tobias Franke und Arjan Kuijper. *Screen Space Cone Tracing for Glossy Reflections*. In Springer International Publishing, Cham, 2016, Seiten 308–318. ISBN: 978-3-319-39907-2. DOI: 10.1007/978-3-319-39907-2_29. URL: http://dx.doi.org/10.1007/978-3-319-39907-2_29 (besucht am 04. 06. 2017).

[Hol10] Dan Holbert. Normal Offset Shadows, 2010. URL: http://www.dissidentlogic.com/old/#Normal%20Offset%20Shadows (besucht am 06. 09. 2017).

[Kaj86] James T. Kajiya. The Rendering Equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, Seiten 143–150, New York, NY, USA. ACM, 1986. ISBN: 0-89791-196-2. DOI: 10.1145/15922.15902.

[Kar13] Brian Karis. Real Shading in Unreal Engine 4, 2013. URL: http://blog.selfshadow.com/publications/s2013-shading-course/

karis/s2013_pbs_epic_notes_v2.pdf (besucht am 20. 06. 2017). SIGGRAPH 2013 Course: Physically Based Shading in Theory and Practice.

[Kar14] Brian Karis. High Quality Temporal Supersampling, 2014. URL: https://de45xmedrsdbp.cloudfront.net/Resources/files/TemporalAA_small-59732822.pdf (besucht am 23. 08. 2017).

[KCL+07] Johannes Kopf, Michael F. Cohen, Dani Lischinski und Matt Uyttendaele. Joint Bilateral Upsampling. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2007)*, 26(3):to appear, 2007.

[KK13] Peter Kán und Hannes Kaufmann. Differential Irradiance Caching for Fast High-Quality Light Transport Between Virtual and Real Worlds. In *Proceedings of International Symposium on Mixed and Augmented Reality (ISMAR)*, Seiten 133–141. IEEE Computer Society, 2013.

[Kno65] Kenneth C. Knowlton. A Fast Storage Allocator. *Commun. ACM*, 8(10):623–624, Oktober 1965. ISSN: 0001-0782. DOI: 10.1145/365628.365655.

[Laf95] Eric P. Lafortune. *Mathematical Models and Monte Carlo Algorithms for Physically Based Rendering*. Dissertation, Department of Computer Science, KU Leuven, Celestijnenlaan 200A, 3001 Heverlee, Belgium, Februar 1995, Seite 167.

[MM14] Morgan McGuire und Michael Mara. Efficient GPU Screen-Space Ray Tracing. *Journal of Computer Graphics Techniques (JCGT)*, 3(4):73–85, Dezember 2014. ISSN: 2331-7418.

[MML12] Morgan McGuire, Michael Mara und David Luebke. Scalable Ambient Obscurance. In *High-Performance Graphics 2012*, Paris, France, Juni 2012. URL: http://graphics.cs.williams.edu/papers/SAOHPG12/ (besucht am 22. 07. 2017).

[MMN+16] Michael Mara, Morgan McGuire, Derek Nowrouzezahrai und David Luebke. Deep G-Buffers for Stable Global Illumination Approximation. In *HPG*, Juni 2016. URL: http://graphics.cs.williams.edu/papers/DeepGBuffer16 (besucht am 18. 05. 2017).

[MSS+10] Quirin Meyer, Jochen Süßmuth, Gerd Sußner, Marc Stamminger und Günther Greiner. On Floating-point Normal Vectors. In *Proceedings of the 21st Eurographics Conference on Rendering*, Seiten 1405–1409. Eurographics Association, 2010. DOI: 10.1111/j.1467-8659.2010.01737.x.

[Ove17] Alexander Overvoorde. Vulkan Tutorial, 2017. URL: https://vulkan-tutorial.com/ (besucht am 03. 06. 2017).

[Ped16] Lasse Jon Fuglsang Pedersen. Temporal Reprojection Anti-Aliasing in INSIDE, 2016. URL: http://www.gdcvault.com/play/1022970/Temporal-Reprojection-Anti-Aliasing-in (besucht am 23. 08. 2017).

[Pet10] Matt Pettineo. A Closer Look at Tone Mapping, 2010. URL: https://mynameismjp.wordpress.com/2010/04/30/a-closer-look-at-tone-mapping/ (besucht am 02. 08. 2017).

[Pue89] Antonio Medina Puerta. The power of shadows: shadow stereopsis. *J. Opt. Soc. Am. A*, 6(2):309–311, Februar 1989. DOI: 10.1364/JOSAA.6.000309.

[SHR10] Cyril Soler, Olivier Hoel und Frank Rochet. A Deferred Shading Pipeline for Real-Time Indirect Illumination. In *ACM SIGGRAPH 2010 Talks*, Seite 18. ACM, 2010. DOI: 10.1145/1837026.1837049. Talk session: Split Second Screen Space.

[SKS02] Peter-Pike Sloan, Jan Kautz und John Snyder. Precomputed Radiance Transfer for Real-time Rendering in Dynamic, Low-frequency Lighting Environments. *ACM Trans. Graph.*, 21(3):527–536, Juli 2002. ISSN: 0730-0301. DOI: 10.1145/566654.566612.

[VK59] The Khronos Vulkan Working Group. Vulkan® 1.0.59 - A Specification, 2017. URL: https://www.khronos.org/registry/vulkan/specs/1.0-extensions/pdf/vkspec.pdf (besucht am 18. 09. 2017).

[YCK+09] Insu Yu, Andrew Cox, Min H. Kim, Tobias Ritschel, Thorsten Grosch, Carsten Dachsbacher und Jan Kautz. Perceptual Influence of Approximate Visibility in Indirect Illumination. *ACM Trans. Appl. Percept.*, 6(4):24:1–24:14, Oktober 2009. ISSN: 1544-3558. DOI: 10.1145/1609967.1609971.

A

Glossar

AO	Umgebungsverdeckung (engl. Ambient Occlusion oder Ambient Obscurence (AO)). Eine Methode zur approximation der Verschattung von Objekten ohne Beachtung der Lichtrichtung. Beschreibt im Allgemeinen wie groß der sichtbare Anteil der Hemisphäre über einem Punkt ist.
Blitting	Auch Bit-Blit für Bit Block Image Transfer. Eine Methode zum schnellen (i.d.R. hardwarebeschleunigten) Verschieben/Kopieren von Bilddaten.
BRDF	Eine Bidirektionale Reflektanzverteilungsfunktion (engl. Bidirectional Reflectance Distribution Function). Eine Funktion, die das Reflexionsverhalten eines Materials für beliebige Einfalls- und Betrachtungswinkel beschreibt.
Cone-Cast	Eine Methode zum Bestimmen eines oder mehrerer Schnittpunkte zwischen der Szene und einem Kegel, analog zu einem Raycast aber mit einem physischen Körper statt einer unendlich dünnen Halbgeraden. Häufig umgesetzt durch mehrere Schnittprüfungen zwischen der Szene und einer Kugel mit einem Radius abhängig von der bisher zurückgelegten Distanz.
Deferred Rendering	Ein 3D-Renderer der "Deferred Shading" verwendet. Ein Zeichenverfahren bei dem im Gegensatz zum Forward-Rendering das Zeichnen der Modelle und die Beleuchtungsberechnungen in getrennten Schritten durchgeführt werden.
Dynamikumfang	Der Bereich der darstellbaren Farben innerhalb eines Bildformates (d.h. der Quotient aus dem minimal und maximal darstellbaren Wert.)

G-Buffer	Der Geometry-Buffer, welcher beim Zeichen der Modelle geschrieben und u.a. für die Berechnung der Beleuchtung herangezogen wird. Er ist eine Menge von Texturen und besteht i.d.R. mindestens aus den Oberflächennormalen, Position/Tiefe, Albedo sowie ggf. zusätzlichen Materialeigenschaften.
GI	Globale Beleuchtung (engl. Global Illumination). Eine Gruppe von Verfahren zur Simulation von Beleuchtungseffekten, welche im Gegensatz zu lokaler Beleuchtung nicht nur die Lichtquelle und die beleuchtete Fläche, sondern auch andere Objekte der Szene berücksichtigen. Zu diesen Effekten gehören u.a. perfekte und nahezu perfekte Reflexionen, diffuse Reflexionen (indirekte Beleuchtung) und Kaustiken.
IBL	Bildbasierte Beleuchtung (engl. Image-Based Lighting). Ein Verfahren zur Beleuchtung von 3D-Modellen bei dem eine omnidirektionale Repräsentation einer realen Umgebung als Grundlage verwendet wird um eine möglichst realistische Beleuchtung zu erreichen.
Importance Sampling	Eine Methode zur Varianzreduktion bei Monte-Carlo-Verfahren, welches darauf aufbaut das Sampling mit einer quasi-zufälligen Sequenz durchzuführen, deren Werte vermehrt in Wertebereichen liegen, welche den größten Einfluss auf das Endergebnis haben.
Light-Probe	Eine Datenstruktur die omnidirektionale Informationen zur Beleuchtung an einem bestimmten Punkt in einer Szene enthält, welche approximativ auch zur Beleuchtung des umliegenden Volumens verwendet werden können. Häufig gelöst durch Cubemaps, entsprechend aufgeteielte 2D-Texturen oder Spherical Harmonics.
MIP-Map	Akronym für lat. "multum in parvo" für "Viel in wenig". Beschreibt eine Reihe von skalierten Kopien eines Ausgangsbildes, wobei die Kantenlänge jedes Bildes halb so groß ist wie die seines Vorgängers. Diese können von aktuellen Grafikkarten effizient gespeichert und ausgelesen werden und werden u.a. verwendet um Aliasing-Artefakte von Texturen in 3D-Anwendungen zu reduzieren.
Monte-Carlo-Verfahren	Eine Klasse von zufallsbasieren Berechnungsverfahren, deren Ergebnisse mit einer gewissen (i.d.R. nach oben beschränkten) Wahrscheinlichkeit falsch sein können.
Poisson-Disc	Eine Menge von Punkten mit einem festen maximalen Abstand, welche einer Poisson-Verteilung unterliegen.

Raycast	Eine Methode zum Bestimmen eines oder mehrerer Schnittpunkte zwischen der Szene und einem Strahl (i.d.R. eine Strecke oder Halbgerade). Wird oft zur Sichtbarkeitsprüfung zwischen Objekten oder zur Visualisierung einer Szene (siehe Raytracing) eigesetzt.
Raytracing	Ein Verfahren zur Visualisierung dreidimensionaler Szenen, welches auf dem Aussenden einer großen Menge von Strahlen basiert.
Sample	Eine Stichprobe oder auch Abtastwert (engl. Sample) ist ein an einer bestimmten Stelle aus einem (kontunierlichen) Signal ausgelesener Messwert (z.B. ein einzelner Frabwert aus einer Textur).
Screen-Space	Das Koordinatensystem nach der Anwendung der (perspektifischen) Projektionsmatrix und der Normalisierung. Zumeist angegeben in normalisierten Gerätekoordinaten (engl. Normalized Device Coordinates (NDC)) zwischen -1 und 1 oder definitionsabhängig evtl. auch in Pixeln innerhalb des fertigen Bildes.
Screen-Space-Algorithmus	Bezeichnung für eine Klasse von Verfahren welche ausschließlich oder vorrangig mit der 2D-Repräsentation einer Szene nach der Rasterisierung dieser arbeiten.
Speicherfragmentierung	Durch die interne Funktionsweise oder ein bestimmtes Muster von Speicherallokationen und Freigaben entstehende unzusammenhängende Lücken zwischen reservierten Speicherblöcken. Dies sorgt dafür, dass Anfragen evtl. nicht mehr erfüllt werden können, obwohl insgesammt noch ausreichend Speicher zur Verfügung stehen würde.
Tone-Mapping	Operation zur Kompression des Dynamikumfangs eines Bildes, z.B. zur Anzeige auf Ausgabegeräten mit einem geringeren Dynamikumfang, als zur Berechnung benötigt wurde.
Uniform	Über einen oder mehrere Zeichenoperationen konstante Shader-Variable.
Vertices	Eckpunkte bestehend aus einer Position sowie ggf. Texturkoordinaten und Farben aus denen Polygone (zumeist Dreiecke) gebildet werden.
View-Space	Das Koordinatensystem mit der Kamera im Ursprung, d.h. das System nach Anwendung der Kamera-Transformationsmatrix.
Voxel	Ein Datenpunkt in einem dreidimensionalen Gitter, d.h. die dreidimensionale Entsprechnung eines Pixels. Zusammengesetzter Begriff aus Volume und Element.

B

Installations- und Bedienungsanleitung

Im Rahmen dieser Arbeit wurde ein 3D-Renderer sowie eine Beispielanwendung entwickelt. Dies diente u.a. zur Analyse des Verfahrens und wurde verwendet um die meisten der in dieser Arbeit gezeigten Bilder zu generieren. Diesem Dokument sollte ein Archiv mit ausführbaren Binärdateien für Windows und Linux sowie dem gesamte Quellcode der Anwendung beiliegen. Alternativ lässt sich dieses Archiv[1] sowie der Quellcode[2] auch über das Internet beziehen. Damit alle Resourcen gefunden werden, muss die Anwendung vom Arbeitsverzeichnis mit der `archives.lst`-Datei[3] aus ausgeführt werden. Zur Ausführung der Anwendung muss sowohl die verwendete Grafikkarte als auch der installierte Treiber die aktuelle Version der Vulkan-API unterstützen. Darüber hinaus werden mindestens 2 GiB freier Arbeitsspeicher vorausgesetzt und es wird die Verwendung einer aktuellen Mittelklassegrafikkarte empfohlen.

Nach dem Start der Anwendung wird die erste Beispielszene geladen. Die Steuerung erfolgt im Wesentlichen über die Benutzeroberfläche (siehe Abb. B.1) sowie über einige Tastatur-Shortcuts. So kann die aktuelle Szene und Position mit `1` bis `6` durchgeschaltet und mit `F3` eine automatische Animationssequenz gestartet werden (Beenden mit `ESC`). Außerdem ist es möglich über die `F11`-Taste die Benutzeroberfläche auszublenden. Zur Steuerung der Kamera kann mittels `W`, `A`, `S`, `D` oder den Pfeiltasten der aktuelle Kameraausschnitt verschoben und bei gedrückter linker Maustaste die Kamera rotiert werden. Durch betätigen der rechten Maustaste kann darüber hinaus ein Würfel ($1m^3$) der Licht emittiert (lediglich über eine Emissive-Farbe und ohne analytische Lichtquelle) erzeugt werden. Die Anwendung kann entweder über die entsprechende Taste des Fensters oder mittels `ESC` beendet werden.

[1] `https://github.com/lowkey42/mirrage/releases/download/v0.2/mirrage_demo_v0.2.zip`
[2] Im GitHub-Projekt `https://github.com/lowkey42/mirrage` zu finden.
[3] `assets`-Verzeichnis im Quellcode-Archiv oder Stammverzeichnis im beigefügten Archiv mit den Binärdateien.

Die Kontrollelemente der grafischen Benutzeroberfläche bieten die folgenden Funktionen:

- **Preset**: Ermöglicht die Wahl zwischen den unterschiedlichen Szenen und verschiedenen vordefinierten Positionen.
- **Show Profiler**: Erlaubt das Ein- und Ausblenden des Profilers.
- **Directional Light**: Enthält verschiedene Optionen für die analytische Lichtquelle. u.a. die Position (**Elevation**, **Azimuth**), die Größe der weichen Schatten (**Size**) und die Farbe (**Color** und darunter liegendes Auswahlfeld).
- **Graphic Settings**: Erlaubt die Änderung der Fensterauflösung und den Wechsel zwischen Vollbild und Fenster-Darstellung (muss mittels **Apply** bestätigt werden).
- **Renderer Settings**: Bietet die folgenden Einstellungsmöglichkeiten, von denen einige mittels (**Apply** bestätigt werden müssen):
  - **Debug Layer**: Erlaubt den Wechsel zwischen der normalen Ansicht (-1), der gesammelten spekularen Beleuchtung (0) und den MIP-Levels mit der diffusen Beleuchtung (1-N).
  - **Indirect Illumination**: Erlaubt die Deaktivierung der globalen Beleuchtung.
  - **High-Resolution GI**: Bestimmt die in Kapitel 4.2 beschriebene präferierte Auflösung der intern verwendeten Texturen, falls mehrere in Frage kommen.
  - **Minimum GI MIP**: Bestimmt das MIP-Level in dem die Berechnungen für die globale Beleuchtung sowie die Bestimmung der spekularen Samples erfolgen.
  - **Diffuse GI MIP**: Bestimmt das niedrigste MIP-Level, welches zur Berechnung der diffusen Beleuchtung herangezogen werden soll.
  - **Sample Count**: Legt die Anzahl der Samples, die zur Berechnung der Diffusen Beleuchtung pro MIP-Level bestimmt werden fest.
  - **Prioritise Near Samples**: Erlaubt das Priorisieren lokaler Samples wie in Abb. 3.6 beschrieben.
  - **Low-Quality MIP-Levels**: Legt die Anzahl von MIP-Levels fest, welche mit dem schnelleren aber fehlerhaften Nearest-Neighbor-Verfahren bestimmt werden sollen.
  - **Exposure**: Erlaubt das Überschreiben der automatisch bestimmten Belichtungszeit.
  - **Background Brightness**: Legt die Helligkeit des Hintergrunds/Himmels fest.
  - **Ambient Occlusion**: Erlaubt das Aktivieren und Deaktivieren der Umgebungsverdeckung.
  - **Bloom**: Erlaubt das Aktivieren und Deaktivieren des Bloom-Effekts.

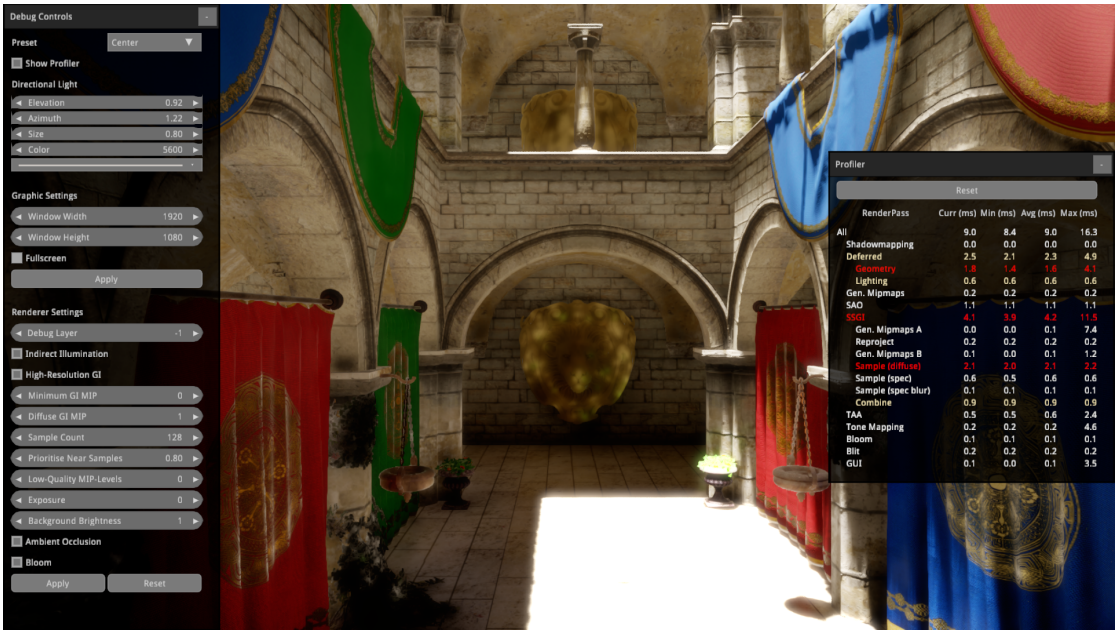


Abb. B.1: Die grafische Benutzeroberfläche der Demo-Anwendung, bestehend aus den Kontrollelementen (links) und dem Profiler mit den Laufzeiten der einzelnen Rendering-Teilschritten (rechts). Der Profiler ist standardmäßig ausgeblendet und kann über den Punkt *Show Profiler* ein- und ausgeblendet werden.

Zur Kompilierung des Quellcodes wird neben CMake (mindestens Version 3.7) ein aktueller C++-Compiler[4] sowie die SDL2-Bibliothek und ein aktuelles Vulkan-SDK benötigt. Für das Kompilieren unter Windows wird der Einsatz des MinGW-Compiler empfohlen[5].

Wenn alle Abhängigkeiten verfügbar sind, kann die Software z.B. mittels `cmake -DCMAKE_BUILD_TYPE=Release PROJEKT_ORDNER` gefolgt von `cmake --build .` kompiliert werden. Es empfehlt sich diese Befehle in einem separaten Ordner auszuführen (Out-of-Source Build). Bei Verwendung des MinGW-Compilers unter Windows muss abweichend davon der folgende Befehl verwendet werden um die Build-Konfiguration zu generieren, damit der korrekte Compiler verwendet wird:

```
cmake -DCMAKE_BUILD_TYPE=Release -G "MinGW Makefiles" PROJEKT_ORDNER
```

Zur Ausführung der kompilierten Demo-Anwendung werden außerdem die verwendeten Modelle aus dem Ordner `extensions` des Archivs mit den Binärdateien benötigt, welche aufgrund ihrer Größe nicht Teil des GIT-Repositories sind.

[4] Es werden u.a. einige C++17 Features verwendet, sodass mindestens Clang 4 oder GCC 7 erforderlich sind. Der Microsoft Visual Studio Compiler (MSCV) wird zum jetzigen Zeitpunkt aufgrund fehlender Features zu Beginn der Entwicklung (N3652, P0292R2, P0195r2, P0135r1 und N4295) sowie weiterer Einschränkungen (u.a. Begrenzung der maximalen Anzahl von Makro-Argumenten) nicht unterstützt.

[5] Ein vollständiges Paket mit diesem Compiler, CMake, SDL2 und dem Vulkan SDK kann von `https://github.com/lowkey42/mirrage/releases/download/v0.2/MinGW_build_environment.zip` bezogen werden. Mittels der beiliegenden `open_distro_window.bat` kann anschließend ein Konsolenfenster geöffnet werden, in dem alle notwendigen Pfade/Variablen gesetzt und alle Binärdateien verfügbar sind.

C

Render-Passes

Derzeit sind die folgenden Passes mittels der aufgeführten Dateien implementiert (mittels `#include` eingebundene Abhängigkeiten ausgenommen):

- **Blit:** Kombiniert das finale Ergebnis der anderen Passes mit denen des Bloom- und Tone-Mapping-Passes.
  Dateien:
  - `src/mirrage/renderer/include/mirrage/renderer/pass/blit_pass.hpp`
  - `src/mirrage/renderer/src/pass/blit_pass.cpp`
  - `assets/core_assets/shader/blit.frag`
  - `assets/core_assets/shader/fullscreen.vert`

- **Bloom:** Extrahiert basierend auf dem Ergebnis des Tone-Mappings die hellsten Bereiche des aktuellen Frames und führt einen Gaußschen Weichzeichner durch um einen Überstrahlungs-Effekt zu erzeugen.
  Dateien:
  - `src/mirrage/renderer/include/mirrage/renderer/pass/bloom_pass.hpp`
  - `src/mirrage/renderer/src/pass/bloom_pass.cpp`
  - `assets/core_assets/shader/bloom_blur.frag`
  - `assets/core_assets/shader/bloom_blur.vert`
  - `assets/core_assets/shader/bloom_filter.frag`
  - `assets/core_assets/shader/fullscreen.vert`

- **Deferred:** Stellt die Geometrie und die direkter Beleuchtung mittels Deferred-Shading dar.
  Dateien:
  - `src/mirrage/renderer/include/mirrage/renderer/pass/deferred_geometry_subpass.hpp`
  - `src/mirrage/renderer/include/mirrage/renderer/pass/deferred_lighting_subpass.hpp`
  - `src/mirrage/renderer/include/mirrage/renderer/pass/deferred_pass.hpp`
  - `src/mirrage/renderer/src/pass/deferred_geometry_subpass.cpp`
  - `src/mirrage/renderer/src/pass/deferred_lighting_subpass.cpp`
  - `src/mirrage/renderer/src/pass/deferred_pass.cpp`
  - `assets/core_assets/shader/fullscreen.vert`
  - `assets/core_assets/shader/light_directional.frag`
  - `assets/core_assets/shader/model.frag`

  - `assets/core_assets/shader/model_emissive.frag`
  - `assets/core_assets/shader/model.vert`

- **Gen-Mipmaps:** Generiert MIP-Maps für die Tiefen- und Material-Textur des G-Buffers.
  Dateien:
  - `src/mirrage/renderer/include/mirrage/renderer/pass/gen_mipmap_pass.hpp`
  - `src/mirrage/renderer/src/pass/gen_mipmap_pass.cpp`
  - `assets/core_assets/shader/fullscreen.vert`
  - `assets/core_assets/shader/gi_mipgen.frag`

- **GI:** Implementiert das in dieser Arbeit vorgestellt GI-Verfahren.
  Dateien:
  - `src/mirrage/renderer/include/mirrage/renderer/pass/gi_pass.hpp`
  - `src/mirrage/renderer/src/pass/gi_pass.cpp`
  - `assets/core_assets/shader/fullscreen.vert`
  - `assets/core_assets/shader/gi_blend.frag`
  - `assets/core_assets/shader/gi_blend_common.frag`
  - `assets/core_assets/shader/gi_integrate_brdf.frag`
  - `assets/core_assets/shader/gi_integrate_brdf.vert`
  - `assets/core_assets/shader/gi_reproject.frag`
  - `assets/core_assets/shader/gi_sample.frag`
  - `assets/core_assets/shader/gi_sample_spec.frag`
  - `assets/core_assets/shader/gi_spec_blur.frag`
  - `assets/core_assets/shader/gi_spec_blur.vert`

- **GUI:** Zeichnet die UI-Elemente.
  Dateien:
  - `src/mirrage/renderer/include/mirrage/renderer/pass/gui_pass.hpp`
  - `src/mirrage/renderer/src/pass/gui_pass.cpp`
  - `assets/core_assets/shader/ui.frag`
  - `assets/core_assets/shader/ui.vert`

- **Shadowmapping:** Generiert soweit notwendig (z.B. nach Bewegung der Lichtquelle) eine neue Shadowmap.
  Dateien:
  - `src/mirrage/renderer/include/mirrage/renderer/pass/shadowmapping_pass.hpp`
  - `src/mirrage/renderer/src/pass/shadowmapping_pass.cpp`
  - `assets/core_assets/shader/shadow_model.frag`
  - `assets/core_assets/shader/shadow_model.vert`

- **SSAO:** Berechnet die Umgebungsverdeckung für den aktuellen Frame.
  Dateien:
  - `src/mirrage/renderer/include/mirrage/renderer/pass/ssao_pass.hpp`
  - `src/mirrage/renderer/src/pass/ssao_pass.cpp`
  - `assets/core_assets/shader/fullscreen.vert`
  - `assets/core_assets/shader/ssao.frag`
  - `assets/core_assets/shader/ssao_blur.frag`
  - `assets/core_assets/shader/ssao_blur.vert`

- **TAA:** Führt die temporale Kantenglättung des aktuellen Frames durch:
  Dateien:
  - `src/mirrage/renderer/include/mirrage/renderer/pass/taa_pass.hpp`
  - `src/mirrage/renderer/src/pass/taa_pass.cpp`
  - `assets/core_assets/shader/taa.frag`
  - `assets/core_assets/shader/fullscreen.vert`

- **Tone-Mapping:** Berechnet die für das Tone-Mapping notwendige, über die Zeit adaptierte, durchschnittliche Helligkeit des aktuellen Frames.
  Dateien:
  - `src/mirrage/renderer/include/mirrage/renderer/pass/tone_mapping_pass.hpp`
  - `src/mirrage/renderer/src/pass/tone_mapping_pass.cpp`
  - `assets/core_assets/shader/fullscreen.vert`
  - `assets/core_assets/shader/luminance.frag`
  - `assets/core_assets/shader/luminance_adapt.frag`

D

Shader Quellcode Auszüge

Dieser Anhang enthält die wichtigsten Ausschnitte aus den Kern-Komponenten des implementierten globalen Beleuchtungsverfahrens.

D.1 Upsampling

Die folgende Funktion enthält die Implementierten des Joint-Bilateral-Upsampling Filters, der u.a. zum Vergrößern der gesammelten diffusen Einstrahlung zwischen den einzelnen Teilschritten dient.

```
1  // Gaußsche Gewichtungs-Funktion für die Differenz der Tiefenwerte
2  vec4 weight_depth(vec4 x, float dev) {
3      return exp(-x*x / (2*dev*dev));
4  }
5
6  // Gewichtungs-Funktion für die Differenz der kodierten Normale
7  vec4 weight_mat_data(vec4 dx, vec4 dy) {
8      return max(vec4(0.005), 1 - smoothstep(0.05, 0.2, (dx*dx+dy*dy)));
9  }
10
11 // Berechnet die UV-Koordinaten der 2x2 Blöcke, die ausgelesen werden sollen und die Gewichte pro Pixel basierend auf
12 //   den hoch und niedrig aufgelösten Normalen und Tiefenwerten.
13 // Gibt die Summe der berechneten Gewichte zurück
14 float calc_upsampled_weights(sampler2D highres_depth_sampler, sampler2D highres_mat_data_sampler,
15                              sampler2D depth_sampler,         sampler2D mat_data_sampler,
16                              vec2      tex_coords,
17                              out vec2 uv_00,     out vec2 uv_10,     out vec2 uv_11,     out vec2 uv_01,
18                              out vec4 weight_00, out vec4 weight_10, out vec4 weight_11, out vec4 weight_01) {
19     // Auslesen des hoch aufgelösten Tiefenwertes und der Normalen
20     float depth = texelFetch(highres_depth_sampler,    ivec2(textureSize(highres_depth_sampler,    0)*tex_coords), 0).r;
21     vec2 normal = texelFetch(highres_mat_data_sampler, ivec2(textureSize(highres_mat_data_sampler, 0)*tex_coords), 0).xy;
22
23     // Berechnung der UV-Koordinaten der 2x2 Blöcke
24     vec2 tex_size = textureSize(depth_sampler, 0);
25     uv_00 = tex_coords + vec2(-1,-1) / tex_size;
26     uv_10 = tex_coords + vec2( 1,-1) / tex_size;
27     uv_11 = tex_coords + vec2( 1, 1) / tex_size;
28     uv_01 = tex_coords + vec2(-1, 1) / tex_size;
29
30     // Initialisierung der Pixel-Gewichte mit denen eines äquivalenten Gauß-Filters
31     weight_00 = vec4(0.125794409230998, 0.132980760133811, 0.125794409230998, 0.118996412547595);
32     weight_10 = vec4(0.125794409230998, 0.106482668507451, 0.100728288549083, 0.118996412547595);
33     weight_11 = vec4(0.100728288549083, 0.085264655436308, 0.100728288549083, 0.118996412547595);
34     weight_01 = vec4(0.100728288549083, 0.106482668507451, 0.125794409230998, 0.118996412547595);
35
36     // Berechnung der Standardabweichung der Tiefenwerte, basierend auf dem hochauflösenden Tiefenwert um übermäßige
37     //   Weichzeichnung in nahen Bereichen zu reduzieren
38     float depth_dev = mix(0.3, 1.5, depth) / global_uniforms.proj_planes.y;
39
40     // Auslesen der niedrig aufgelösten Tiefenwerte und Modulation der Gewichte basierend auf ihrem Unterschied
41     //   zum hoch aufgelösten Tiefenwert
42     weight_00 *= weight_depth(textureGather(depth_sampler, uv_00, 0) - depth, depth_dev);
43     weight_10 *= weight_depth(textureGather(depth_sampler, uv_10, 0) - depth, depth_dev);
44     weight_11 *= weight_depth(textureGather(depth_sampler, uv_11, 0) - depth, depth_dev);
45     weight_01 *= weight_depth(textureGather(depth_sampler, uv_01, 0) - depth, depth_dev);
46
47     // Auslesen der niedrig aufgelösten, kodierten Normalen und Bildung der Differenz zum hochaufgelösten Wert
48     vec4 normal_x_00 = textureGather(mat_data_sampler, uv_00, 0) - normal.x;
49     vec4 normal_x_10 = textureGather(mat_data_sampler, uv_10, 0) - normal.x;
50     vec4 normal_x_11 = textureGather(mat_data_sampler, uv_11, 0) - normal.x;
```

```
51      vec4 normal_x_01 = textureGather(mat_data_sampler, uv_01, 0) - normal.x;
52
53      vec4 normal_y_00 = textureGather(mat_data_sampler, uv_00, 1) - normal.y;
54      vec4 normal_y_10 = textureGather(mat_data_sampler, uv_10, 1) - normal.y;
55      vec4 normal_y_11 = textureGather(mat_data_sampler, uv_11, 1) - normal.y;
56      vec4 normal_y_01 = textureGather(mat_data_sampler, uv_01, 1) - normal.y;
57
58      // Modulation der Gewichte basierend auf dem Unterschied der Normalen zum hoch aufgelösten Wert
59      weight_00 *= weight_mat_data(normal_x_00, normal_y_00);
60      weight_10 *= weight_mat_data(normal_x_10, normal_y_10);
61      weight_11 *= weight_mat_data(normal_x_11, normal_y_11);
62      weight_01 *= weight_mat_data(normal_x_01, normal_y_01);
63
64      // Summe der Gewichte berechnen
65      return dot(weight_00, vec4(1))
66           + dot(weight_10, vec4(1))
67           + dot(weight_11, vec4(1))
68           + dot(weight_01, vec4(1));
69  }
70
71  // Berechnet die Approximation einer hoch aufgelösten Lösung an einem Punkt, basierend auf einer niedrig aufgelösten
72  //   (color_sampler) mittels Join-Bilateral-Upsampling, unter Verwendung der hoch und niedrig aufgelösten Normalen
73  //   und Tiefenwerte
74  vec3 upsampled_result(sampler2D highres_depth_sampler, sampler2D highres_mat_data_sampler,
75                        sampler2D depth_sampler,         sampler2D mat_data_sampler,
76                        sampler2D color_sampler,         vec2      tex_coords) {
77      // Berechnen der UV-Koordinaten und Gewichte
78      vec2  uv_00,     uv_10,     uv_11,     uv_01;
79      vec4  weight_00, weight_10, weight_11, weight_01;
80      float weight_sum = calc_upsampled_weights(highres_depth_sampler, highres_mat_data_sampler,
81                                                depth_sampler,         mat_data_sampler, tex_coords,
82                                                uv_00,     uv_10,      uv_11,     uv_01,
83                                                weight_00, weight_10,  weight_11, weight_01);
84
85      // Fallback zu linearer Interpolation, falls es keine hinreichend gute Übereinstimmung gab
86      if(weight_sum<0.001)
87          return textureLod(color_sampler, tex_coords, 0).rgb;
88
89      // Auslesen und Gewichten der RGB-Werte der 16 umliegenden Pixel
90      float color_r = dot(vec4(1),
91                      textureGather(color_sampler, uv_00, 0) * weight_00
92                    + textureGather(color_sampler, uv_10, 0) * weight_10
93                    + textureGather(color_sampler, uv_11, 0) * weight_11
94                    + textureGather(color_sampler, uv_01, 0) * weight_01);
95
96      float color_g = dot(vec4(1),
97                      textureGather(color_sampler, uv_00, 1) * weight_00
98                    + textureGather(color_sampler, uv_10, 1) * weight_10
99                    + textureGather(color_sampler, uv_11, 1) * weight_11
100                   + textureGather(color_sampler, uv_01, 1) * weight_01);
101
102     float color_b = dot(vec4(1),
103                     textureGather(color_sampler, uv_00, 2) * weight_00
104                   + textureGather(color_sampler, uv_10, 2) * weight_10
105                   + textureGather(color_sampler, uv_11, 2) * weight_11
106                   + textureGather(color_sampler, uv_01, 2) * weight_01);
107
108     return vec3(color_r, color_g, color_b) / weight_sum;
109 }
```

D.2 MIP-Map Generierung

Das folgende Fragment zeigt das implementierte Verfahren zur Generierung der MIP-Maps durch das in Kapitel 3.6.2 vorgestellte Abstimmungsverfahren. Der Shader wird entsprechend für jedes MIP-Level vom zweiten bis zum höchsten aufgerufen und verwendet jeweils das Ergebnis aus dem nächstniedrigeren Level.

```
1   layout(location = 0) in Vertex_data {
2       vec2 tex_coords;
3   } vertex_out;
4
5   layout(location = 0) out vec4 out_depth;
6   layout(location = 1) out vec4 out_mat_data;
7
8   layout(set=1, binding = 0) uniform sampler2D depth_sampler;
9   layout(set=1, binding = 1) uniform sampler2D mat_data_sampler;
10
11  // Gaußsche Gewichtungs-Funktion für die Differenz der Komponenten der kodierten Normale
12  float g1(float x) {
13      float c = 0.1;
14      return exp(-x*x / (2*c*c));
15  }
```

```
16
17  // Gaußsche Gewichtungs-Funktion für die Differenz der Tiefenwerte
18  float g2(float x) {
19      float c = 0.001f;
20      return exp(-x*x / (2*c*c));
21  }
22
23  // Bestimmt den Pixel aus den 4 umliegenden Pixeln, der am ehesten mit den 16 Pixeln der Umgebung übereinstimmt
24  void main() {
25      // Berechnen der UV-Koordinaten der 2x2 Blöcke, die ausgelesen werden sollen
26      vec2 tex_size = textureSize(depth_sampler, 0);
27      const vec2 uv_00 = vertex_out.tex_coords + vec2(-1,-1) / tex_size;
28      const vec2 uv_10 = vertex_out.tex_coords + vec2( 1,-1) / tex_size;
29      const vec2 uv_11 = vertex_out.tex_coords + vec2( 1, 1) / tex_size;
30      const vec2 uv_01 = vertex_out.tex_coords + vec2(-1, 1) / tex_size;
31      const ivec2[] center_offsets = ivec2[4](ivec2(0,0), ivec2(1,0), ivec2(1,1), ivec2(0,1));
32
33      // Auslesen der Tiefenwerte und Berechnung des Scores basierend auf ihrer Ähnlichkeit zum Durchschnitt der Umgebung
34      vec4 depth_00 = textureGather(depth_sampler, uv_00, 0);
35      vec4 depth_10 = textureGather(depth_sampler, uv_10, 0);
36      vec4 depth_11 = textureGather(depth_sampler, uv_11, 0);
37      vec4 depth_01 = textureGather(depth_sampler, uv_01, 0);
38
39      float avg_depth = (dot(vec4(1), depth_00) + dot(vec4(1), depth_10)
40                      + dot(vec4(1), depth_11) + dot(vec4(1), depth_01)) / 16.0;
41
42      vec4 center_depths = vec4(depth_00.y, depth_10.x, depth_11.w, depth_01.z);
43
44      vec4 score = vec4(1.0);
45      score *= vec4(g2(avg_depth - center_depths.x),
46                    g2(avg_depth - center_depths.y),
47                    g2(avg_depth - center_depths.z),
48                    g2(avg_depth - center_depths.w) );
49
50      // Auslesen der X-Komponente der kodierten Normale und Berechnung des Scores basierend auf ihrer
51      //   Ähnlichkeit zum Durchschnitt der Umgebung
52      vec4 normal_x_00 = textureGather(mat_data_sampler, uv_00, 0);
53      vec4 normal_x_10 = textureGather(mat_data_sampler, uv_10, 0);
54      vec4 normal_x_11 = textureGather(mat_data_sampler, uv_11, 0);
55      vec4 normal_x_01 = textureGather(mat_data_sampler, uv_01, 0);
56
57      float avg_normal_x = (dot(vec4(1), normal_x_00) + dot(vec4(1), normal_x_10)
58                          + dot(vec4(1), normal_x_11) + dot(vec4(1), normal_x_01)) / 16.0;
59
60      vec4 center_normals_x = vec4(normal_x_00.y, normal_x_10.x, normal_x_11.w, normal_x_01.z);
61
62      score *= vec4(g1(avg_normal_x - center_normals_x.x),
63                    g1(avg_normal_x - center_normals_x.y),
64                    g1(avg_normal_x - center_normals_x.z),
65                    g1(avg_normal_x - center_normals_x.w) );
66
67      // Auslesen der Y-Komponente der kodierten Normale und Berechnung des Scores basierend auf ihrer
68      //   Ähnlichkeit zum Durchschnitt der Umgebung
69      vec4 normal_y_00 = textureGather(mat_data_sampler, uv_00, 1);
70      vec4 normal_y_10 = textureGather(mat_data_sampler, uv_10, 1);
71      vec4 normal_y_11 = textureGather(mat_data_sampler, uv_11, 1);
72      vec4 normal_y_01 = textureGather(mat_data_sampler, uv_01, 1);
73
74      float avg_normal_y = (dot(vec4(1), normal_y_00) + dot(vec4(1), normal_y_10)
75                          + dot(vec4(1), normal_y_11) + dot(vec4(1), normal_y_01)) / 16.0;
76
77      vec4 center_normals_y = vec4(normal_y_00.y, normal_y_10.x, normal_y_11.w, normal_y_01.z);
78
79      score *= vec4(g1(avg_normal_y - center_normals_y.x),
80                    g1(avg_normal_y - center_normals_y.y),
81                    g1(avg_normal_y - center_normals_y.z),
82                    g1(avg_normal_y - center_normals_y.w) );
83
84      // Bestimmen der Koordinaten des Pixels mit dem höchsten Score
85      int max_index = 3;
86      float s = score.w;
87
88      if(score.x > s) {
89          max_index = 0;
90          s = score.x;
91      }
92      if(score.y > s) {
93          max_index = 1;
94          s = score.y;
95      }
96      if(score.z > s) {
97          max_index = 2;
98          s = score.z;
99      }
100
101     // Schreiben der Tiefenwerte und Materialdaten des gefundenen Pixels
102     out_depth    = texelFetch(depth_sampler,    ivec2(vertex_out.tex_coords * tex_size) + center_offsets[max_index], 0);
103     out_mat_data = texelFetch(mat_data_sampler, ivec2(vertex_out.tex_coords * tex_size) + center_offsets[max_index], 0);
104 }
```

D.3 Diffuse GI-Samples

Der folgende Auszug zeigt, den zur Berechnung der diffusen Einstrahlung implementierten Shader. Der Shader wird entsprechend des in Kapitel 3.6.1 erläuterten Vorgehens einmal pro MIP-Level der Ausgabetextur beginnend vom höchsten Level ausgeführt und verwendet dabei die Teilergebnisse der vorherigen Level.

```
1   layout(location = 0) in Vertex_data {
2       vec2 tex_coords;
3   } vertex_out;
4
5   layout(location = 0) out vec4 out_color;
6
7   layout(set=1, binding = 0) uniform sampler2D color_sampler;
8   layout(set=1, binding = 1) uniform sampler2D depth_sampler;
9   layout(set=1, binding = 2) uniform sampler2D mat_data_sampler;
10  layout(set=1, binding = 3) uniform sampler2D result_sampler;
11  layout(set=1, binding = 4) uniform sampler2D history_weight_sampler;
12  layout(set=1, binding = 5) uniform sampler2D prev_depth_sampler;
13  layout(set=1, binding = 6) uniform sampler2D prev_mat_data_sampler;
14  layout(set=1, binding = 7) uniform sampler2D ao_sampler;
15
16  // 1 wenn dies das letzte MIP-Level ist, in dem noch Samples genommen werden sollen
17  layout (constant_id = 0) const int LAST_SAMPLE = 0;
18  // Der Radius in Pixeln aus dem die Samples genommen werden
19  layout (constant_id = 1) const float R = 40;
20  // Die Anzahl von Samples, die genommen werden sollen
21  layout (constant_id = 2) const int SAMPLES = 128;
22  // 1 wenn nur noch das Upsampling des vorherigen Ergebnisses durchgeführt aber keine Samples mehr berechnet werden sollen
23  layout (constant_id = 3) const int UPSAMPLE_ONLY = 0;
24  // Gibt näheren Samples ein höheres Gewicht um einen intensiveren Farbtransfer zwischen nahegelegenen Oberflächen
25  //   zu erreichen.
26  layout (constant_id = 4) const float PRIORITISE_NEAR_SAMPLES = 0.6;
27
28  // Die Argumente dynamischen Variablen des Shaders. In den Komponenten von zwei Matrizen um das Pipeline-Layout
29  //   zwischen den GI-Passes kompatibel zu halten
30  layout(push_constant) uniform Push_constants {
31      // [3][3] = Intensität der Umgebunsverdeckung (0 = Deaktiviert)
32      mat4 projection;
33
34      // [0][0] = Basis MIP-Level (höher aufgelöste Alternative)
35      // [2][0] = Exponent für alternativen Skalierfaktor (siehe PRIORITISE_NEAR_SAMPLES)
36      // [0][3] = Aktuelles MIP-Level (relativ zum Basis-Level)
37      // [1][3] = Höchstes relevantes MIP-Level (relativ zum Basis-Level)
38      // [2][3] = Vorberechneter Teil des ds-Faktors, der in calc_illumination_from verwendet wird
39      // [3][3] = Basis MIP-Level (ausgewählte Alternative)
40      mat4 prev_projection;
41  } pcs;
42
43  #include "global_uniforms.glsl"
44  #include "normal_encoding.glsl"
45  #include "random.glsl"
46  #include "brdf.glsl"
47  #include "upsample.glsl"
48  #include "raycast.glsl"
49
50  vec3 gi_sample(int lod, int base_mip);
51  vec3 calc_illumination_from(int lod, vec2 tex_size, ivec2 src_uv, vec2 shaded_uv, float shaded_depth,
52                              vec3 shaded_point, vec3 shaded_normal, out float weight);
53
54  // Berechnet die Helligkeit einer RGB-Farbe (zur Normalisierung)
55  float luminance_norm(vec3 c) {
56      vec3 f = vec3(0.299,0.587,0.114);
57      return sqrt(c.r*c.r*f.r + c.g*c.g*f.g + c.b*c.b*f.b);
58  }
59
60  void main() {
61      float current_mip = pcs.prev_projection[0][3];
62      float max_mip     = pcs.prev_projection[1][3];
63      float base_mip    = pcs.prev_projection[3][3];
64
65      // Upsampling des vorherigen Teilergebnisses (falls dies nicht das erste Level ist)
66      if(current_mip < max_mip)
67          out_color = vec4(upsampled_result(depth_sampler, mat_data_sampler,
68                                            prev_depth_sampler, prev_mat_data_sampler,
69                                            result_sampler, vertex_out.tex_coords), 1.0);
70      else
71          out_color = vec4(0,0,0, 1);
72
73      // Samples für dieses Level bestimmen und auswerten (sofern das Ziel-Level noch nicht erreich/überschitten wurde)
74      if(UPSAMPLE_ONLY==0)
75          out_color.rgb += gi_sample(int(current_mip+0.5), int(base_mip+0.5));
76
77      // Letztes Level erreicht (= Endergebnis) => Lineare Interpolation mit letztem Frame
78      if(abs(current_mip - base_mip) < 0.00001) {
79          // Interpolationsfaktor basierend auf dem bei der Reprojektion berechneten Fehler in der Umgebung wählen
80          vec2 hws_step = 1.0 / textureSize(history_weight_sampler, 0);
```

```
81
82          vec4  history_weights = textureGather(history_weight_sampler, vertex_out.tex_coords+hws_step, 0);
83          float history_weight  = min(history_weights.x, min(history_weights.y, min(history_weights.z, history_weights.w)));
84
85          history_weights = textureGather(history_weight_sampler, vertex_out.tex_coords-hws_step, 0);
86          history_weight  = min(history_weight,min(history_weights.x, min(history_weights.y, min(history_weights.z, ↙
        ↳ history_weights.w))));
87
88          // Diffuse Einstrahlung mit der Umgebungsverdeckung modulieren
89          if(pcs.projection[3][3]>0.0) {
90              float ao = texture(ao_sampler, vertex_out.tex_coords).r;
91              ao = mix(1.0, ao, pcs.projection[3][3]);
92              out_color.rgb *= ao;
93          }
94
95          // Normalisieren der diffusen Einstrahlung mit der Helligkeit (reduziert Fire-Fly-Artefakte)
96          out_color.rgb /= (1 + luminance_norm(out_color.rgb));
97
98          // Minimale und Maximale Interpolationsfaktoren basierend auf dem Bildrate bestimmen um Schlierenbildung und
99          //   Flackern zu minimieren
100         float weight_measure = smoothstep(1.0/120.0, 1.0/30.0, global_uniforms.time.z);
101         float weight_min = mix(0.85, 0.1, weight_measure);
102         float weight_max = mix(0.98, 0.85, weight_measure);
103
104         // Interpolationsfaktor wählen und Alpha-Blending mit dem letzten Frame durchführen, der sich bereits in der
105         //   Ergebnistextur befindet
106         out_color *= 1.0 - mix(weight_min, weight_max, history_weight);
107     }
108
109     // Wertebereich auf ein sinnvolles Intervall einschränken um Artefakte durch Feedbackschleifen zu reduzieren
110     out_color = clamp(out_color, vec4(0), vec4(20));
111 }
112
113 const float PI     = 3.14159265359;
114 const float REC_PI = 0.3183098861837906715; // 1/PI
115
116 // Bestimmt die diffuse Einstahlung für das aktuelle Level
117 vec3 gi_sample(int lod, int base_mip) {
118     // UV-Koordinaten in Pixeln berechnen
119     vec2 texture_size = textureSize(color_sampler, 0);
120     ivec2 uv = ivec2(vertex_out.tex_coords * texture_size);
121
122     // Betrachteten Bereich zusätzlich einschränken um Artefakte an den Rändern zu reduzieren
123     if(uv.y >= texture_size.y-1 || uv.x >= texture_size.x-1)
124         return vec3(0, 0, 0);
125
126     // Tiefenwerte und Normalen für den Ziel-Pixel auslesen und seine View-Space Position rekonstruieren
127     float depth  = texelFetch(depth_sampler, uv, 0).r;
128     vec4 mat_data = texelFetch(mat_data_sampler, uv, 0);
129     vec3 N = decode_normal(mat_data.rg);
130     vec3 P = position_from_ldepth(vertex_out.tex_coords, depth);
131
132     // SAMPLES viele Samples in einem Spiralmuster um den aktuellen Pixel auswerten
133     vec3 c = vec3(0,0,0);
134     float samples_used = 0.0;
135     float angle = random(vec4(vertex_out.tex_coords, 0, global_uniforms.time.x*10.0));
136     float angle_step = 1.0 / float(SAMPLES) * PI * 2.0 * 19.0;
137
138     for(int i=0; i<SAMPLES; i++) {
139         float r = max(4, mix(LAST_SAMPLE==1 ? 4.0 : R/2.0, R, float(i)/float(SAMPLES)));
140
141         angle += angle_step;
142         float sin_angle = sin(angle);
143         float cos_angle = cos(angle);
144
145         ivec2 p = ivec2(uv + vec2(sin_angle, cos_angle) * r);
146         float weight;
147         vec3 sc = calc_illumination_from(lod, texture_size, p, uv, depth, P, N, weight);
148
149         c += sc;
150         samples_used += weight;
151     }
152
153     // Skalieren der gesammelten Einstrahlung mit einem Faktor um die geringere Intensität kleinerer MIP-Level
154     //   zu kompensieren
155     float actual_lod = lod - pcs.prev_projection[0][0];
156     float scale_exponent = mix(actual_lod,
157                                pcs.prev_projection[2][0],
158                                PRIORITISE_NEAR_SAMPLES);
159     c *= pow(2.0, scale_exponent*2);
160
161     // Normalisieren der gesammelten Einstrahlung mit der Anzahl der Samples um eine konsistentere Beleuchtung
162     //   unabhängig von der Anzahl zu erreichen.
163     c *= 128.0 / SAMPLES;
164
165     return c;
166 }
167
168 // Berechnet die Einstrahlung die von einem Pixel am Zielpunkt (x) ankommt
169 vec3 calc_illumination_from(int lod, vec2 tex_size, ivec2 src_uv, vec2 shaded_uv, float shaded_depth,
170                             vec3 shaded_point, vec3 shaded_normal, out float weight) {
171     // Tiefenwert und Normale am Ausgangspixel
```

```
172     vec4 mat_data = texelFetch(mat_data_sampler, src_uv, 0);
173     vec3 N = decode_normal(mat_data.rg);
174     float depth  = texelFetch(depth_sampler, src_uv, 0).r;
175 
176     if(depth>=0.9999) {
177         // Pixel gehört zur Skybox => reduzieren des Tiefenwertes, damit die Einstrahlung noch mit einfließt
178         depth = 0.1;
179     }
180 
181     // Rekonstruieren der View-Space Position (x_i) des Ausgangspunktes und Berechnung der Richtung und Distanz^2 zu x
182     vec3 P = position_from_ldepth(src_uv / tex_size, depth);
183     vec3 Pn = normalize(P);
184     vec3 diff = shaded_point - P;
185     vec3 dir = normalize(diff);
186     float r2 = dot(diff, diff);
187 
188     float visibility = 1.0; // v(x,x_i); z.Z. nicht implementiert
189 
190     // Interpolieren von r^2 zu (r^2)^1,25 für weiter entfernte Punkte um das Fehlen des Sichtbarkeitsterm zu kompensieren
191     r2 = mix(r2, pow(r2,1.25), clamp((r2-1)*0.5, 0, 1));
192 
193     float NdotL_src = clamp(dot(N, dir),              0.0, 1.0); // cos(θ')
194     float NdotL_dst = clamp(dot(shaded_normal, -dir), 0.0, 1.0); // cos(θ)
195 
196     // Wenn der Ausgangspunkt ein Emitter ist (mat_data.b=0), wird die Normale ggf. umgekehrt um eine höhere helligkeit
197     //   zu erreichen. Dies approximiert Licht das von der Rückseite emittiert wird, unter der Annahme, dass das Objekt
198     //   symmetrisch ist.
199     NdotL_src = mix(max(clamp(dot(-N, dir), 0.0, 1.0), NdotL_src), NdotL_src, step(0.0001, mat_data.b));
200 
201     // Berechnung der Größe des Differenzialbereichs (ds)
202     float cos_alpha = Pn.z;
203     float cos_beta  = dot(Pn, N);
204     float z = depth * global_uniforms.proj_planes.y;
205     float ds = pcs.prev_projection[2][3] * z*z * clamp(cos_alpha / cos_beta, 0.001, 1000.0);
206 
207     // Multiplizieren aller Faktoren, die die Helligkeit der Einstrahlung beeinflussen
208     weight = visibility * NdotL_dst * NdotL_src * ds / (0.1+r2);
209 
210     // Das vom Ausgangspunkt emittierte Licht auslesen und mit den berechneten Faktoren skalieren
211     vec3 radiance = texelFetch(color_sampler, src_uv, 0).rgb;
212     return max(vec3(0.0), radiance * weight);
213 }
```

Erklärung der Kandidatin / des Kandidaten

☐ Die Arbeit habe ich selbstständig verfasst und keine anderen als die angegebenen Quellen- und Hilfsmittel verwendet.

☐ Die Arbeit wurde als Gruppenarbeit angefertigt. Meine eigene Leistung ist ...

Diesen Teil habe ich selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Namen der Mitverfasser: ...

Datum

Unterschrift der Kandidatin / des Kandidaten