UNIVERSITÄT
KOBLENZ · LANDAU

Fachbereich 4: Informatik

Untersuchung von Verfahren für dynamische Global Illumination in Echtzeit

Bachelorarbeit

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

vorgelegt von

Alexander Gauggel

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

Koblenz, im September 2019

Erklärung

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

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐

.....

(Ort, Datum) (Unterschrift)

Zusammenfassung

Global-Illumination ist eine wichtige Komponente beim Rendering von realistischen Bildern. Der Rechenaufwand für die akkurate Simulation dieser Effekte ist jedoch zu hoch für die Berechnung in Echtzeit. In dieser Arbeit werden Light-Propagation-Volumes, Screen-Space-Reflections und mehrere Varianten von Screen-Space-Ambient-Occlusion als Lösungen für Echtzeitrendering untersucht. Es wird gezeigt, dass alle schnell genug für den Einsatz in Echtzeitanwendungen sind. Die einzelnen Techniken approximieren nur einige Aspekte des Transports von Licht, ergänzen sich jedoch gegenseitig.

Abstract

Global-Illumination is an important part of the rendering of realistic images. However, the computational complexity of an accurate simulation of these effects is too high for the use in real time applications. In this paper Light-Propagation-Volumes, Screen-Space-Reflections and multiple variants of Screen-Space-Ambient-Occlusion are investigated as a solution for real time rendering. It is shown that they are fast enough for the use in real time applications. The various techniques approximate only a few aspects of the light transport, but complement each other.

Inhaltsverzeichnis

1 Einleitung

Die Simulation von Licht ist ein essenzieller Bestandteil des Renderings realistischer Bilder. Im Offline-Rendering wird Licht akkurat simuliert, um photorealistische Bilder zu erzeugen. Allerdings benötigt dieser Prozess je nach Szene Stunden oder Tage.

Trotz großer Entwicklungen in Hardware und Algorithmen, stellt die Berechnung von Global-Illumination in Echtzeit immer noch eine große Herausforderung dar. Oft wird die Beleuchtung komplett vorberechnet. Dies erzwingt nicht nur eine statische Szene, sondern kann auch ein zeitintensiver Prozess sein. Andere Techniken berechnen einzelne Faktoren vor und erlauben Dynamik in gewissen Bereichen. Oft werden einige Aspekte der indirekten Beleuchtung stark vereinfacht oder komplett ignoriert. Manche Techniken liefern zwar gute Ergebnisse, sind aber so rechenaufwändig, dass sie aktuell für den Einsatz in Echtzeitanwendungen in vielen Fällen zu langsam sind.

In dieser Arbeit werden Techniken für die Darstellung von indirekter Beleuchtung untersucht und verglichen. Der Fokus liegt dabei auf der Echtzeitfähigkeit und Dynamik von Licht und Geometrie. Es werden Light-Propagation-Volumes, mehrere Varianten von Screen-Space-Ambient-Occlusion und Screen-Space-Reflections betrachtet.

Im folgenden Kapitel werden die mathematischen und physikalischen Grundlagen der Problemstellung erläutert. Anschließend werden die verschiedenen Techniken im Detail vorgestellt und erklärt. Es folgt ein Kapitel über die programmiertechnische Umsetzung. Abschließend werden die Ergebnisse evaluiert. Zudem wird ein Ausblick für die Verbesserungen der vorgestellten Techniken gegeben sowie einige offene Probleme skizziert.

2 Grundlagen

Das Ziel des Rendering-Prozesses ist es, in die Kamera fallendes Licht zu berechnen und so Bilder zu erzeugen. Eine einfache Herangehensweise ist die Reflectance-Equation [AMHH18]:

$$L_o(p, v) = \int_{l \in \Omega} f(l, v) L_i(p, l)(n \cdot l)^+ dl. \tag{1}$$

Diese berechnet das Licht abhängig vom betrachteten Punkt in der Welt p und dem View-Vektor v. Ω ist die obere Halbkugel über p, $f(l, v)$ ist die Bidirektionale Reflexionsverteilungsfunktion, kurz BRDF, welche das Reflexionsverhalten von Materialien beschreibt. L_i ist das einfallende Licht an Punkt p, aus Richtung l und $(n \cdot l)^+$ ist das Skalarprodukt zwischen der Normale und dem Richtungsvektor, auf den positiven Bereich beschränkt.

Diese Formel berücksichtigt jedoch nicht das aus der Umgebung reflektierte

Licht. Eine vollständigere Formel ist die Rendering-Equation [AMHH18]:

$$L_o(p, v) = L_e(p, v) + \int_{l \in \Omega} f(l, v) L_o(r(p, l), -l)(n \cdot l)^+ dl \tag{2}$$

$L_e(p, v)$ ist das ausgehende Licht von p in Richtung v. Der Term $L_i(p, l)$ wurde ersetzt durch $L_o(r(p, l), -l)$, das ausgehende Licht in Richtung $-l$ von einem anderen Punkt, welcher durch die Raycasting-Funktion $r(p, l)$ gegeben ist. Bei dieser Gleichung wird nicht nur das direkte Licht beachtet. Für jeden Punkt wird auch das Licht berücksichtigt, welches aus der Umgebung reflektiert wird.

Da L_o auf beiden Seiten der Gleichung vorkommt ist die Formel rekursiv. Da es keine Abbruchbedingung gibt, muss das Ergebnis durch eine endliche Anzahl von Iterationen approximiert werden. Zudem ist die Nachbarschaftssuche ein rechenaufwändiges Problem.

Als ein Modell für das Verhalten von Licht dienen Lichtpfade. Hierbei wird Licht als Menge von Strahlen betrachtet. Diese werden von Lichtquellen ausgesendet, in der Szene reflektiert und fallen gegebenenfalls in die Kamera. Eine weit verbreitete Notation stammt aus [Hec90]. Dort werden zwei grundlegende Arten von Reflektionen genutzt, diffus und spekular. Licht wird von der Lichtquelle L ausgestrahlt, diffus D oder spekular S reflektiert und fällt zuletzt in das Auge E.

Lichtpfade werden als Kombination dieser Komponenten notiert. Licht welches direkt in das Auge fällt, wird notiert als LE. Licht das diffus reflektiert und über einen Spiegel in das Auge fällt ist $LDSE$.

Die Rendering-Equation kann als regulärer Ausdruck dargestellt werden: $L(DS)*$ E [AMHH18]. Dieser stellt ausgehendes Licht dar, welches, nach einer beliebigen Anzahl diffuser oder spekularer Reflektionen, in das Auge fällt.

3 Light-Propagation-Volumes

3.1 Konzept

Light-Propagation-Volumes (LPV) ist eine Technik zur Approximation von diffuser, indirekter Beleuchtung in Echtzeit. Hierfür wird Licht in einer Gitterstruktur mit Hilfe von Spherical-Harmonics gespeichert. Die Ausbreitung wird schrittweise berechnet. Die Technik wurde ursprünglich von Anton Kaplanyan entwickelt und erstmals in [Kap09] vorgestellt.

3.2 Spherical-Harmonics

Spherical-Harmonics (SH) sind auf der Kugel definierte harmonische Funktionen. Sie werden in der Computergrafik häufig für die Repräsentation von Licht genutzt [Slo08].

Mathematisch definieren Spherical-Harmonics eine orthonormale Basis über die Einheitskugel S, welche wie folgt definiert ist [Slo08]:

$$S = (x, y, z) = (sin(\theta)cos(\varphi), sin(\theta)sin(\varphi), cos(\theta)) \tag{3}$$

Der erste Teil sind die Koordinaten im Kartesischen-Koordinatensystem (x, y, z), der Zweite ist definiert durch den Polar- und Azimutwinkel (θ, φ) in Kugelkoordinaten.

Die Basisfunktionen sind wie folgt definiert [Slo08]:

$$Y_l^m(\theta, \varphi) = K_l^m e^{im\varphi} p_l^{|m|}(cos\varphi), l \in N, -l \leq m \leq l \tag{4}$$

P_l^m sind die Zugeordneten Legendrepolynome, K_l^m sind Konstanten:

$$K_l^m = \sqrt{\frac{(2l+1)(l-|m|)!}{4\pi(l+|m|)}} \tag{5}$$

Gleichung (4) ist die komplexe Form, in der Computergrafik wird meist die reellwertige Basis genutzt [Slo08]:

$$Y_l^m = \begin{cases} \sqrt{2}RE(Y_l^m) m > 0 \\ \sqrt{2}IM(Y_l^m) m < 0 \\ \quad Y_l^0 \qquad m = 0 \end{cases} = \begin{cases} \sqrt{2}K_l^m cos\,(m\varphi) P_l^m(cos\theta) \quad m > 0 \\ \sqrt{2}K_l^m sin\,(-m\varphi) P_l^{|m|}(cos\theta) m < 0 \\ \quad K_l^0 P_l^m(cos(\theta)) \qquad\qquad m = 0 \end{cases} \tag{6}$$

Der Index l wird als das Band bezeichnet. Jedes Band beinhaltet $2l + 1$ Funktionen welche mit m gekennzeichnet sind. Die Koeffizienten mit $m = 0$ werden als Zonal-Harmonics bezeichnet und sind invariant bei der Rotation um bestimmte Achsen.

Intuitiv kann man Spherical-Harmonics als Polynome auf einer Kugel interpretieren. Polynome können effizient durch ihre Koeffizienten repräsentiert werden und können genutzt werden, um Funktionen zu approximieren. In der Computergrafik wird so oft der Lichteinfall an einer Stelle winkelabhängig gespeichert. Mit steigender Anzahl von Koeffizienten, steigt die Genauigkeit, allerdings erhöht sich auch Speicher- und Rechenaufwand.

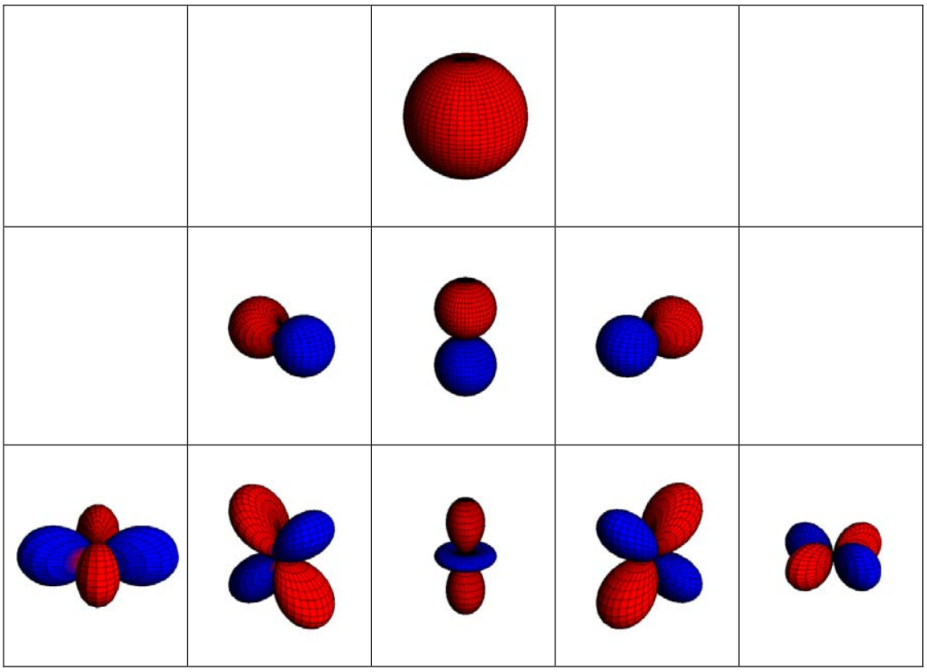


Abbildung 1: Visualisierung der ersten drei SH-Bänder, aus [Slo08]

Eine häufige Anwendung von Spherical-Harmonics ist die Berechnung von reflektiertem Licht [Gre03]. Hierfür muss das Integral über das einfallende Licht $L(s)$ und der BRDF der Oberfläche $t(s)$ gebildet werden:

$$\int_S L(s)t(s)ds \tag{7}$$

Durch die Projektion beider Funktionen in Spherical-Harmonics kann das Integral effizient als das Skalarprodukt der Koeffizienten berechnet werden:

$$\int_S \tilde{L}(s)\tilde{t}(s)ds = \sum_{i=0}^{n^2} L_i t_i \tag{8}$$

3.3 Licht-Injektion

Für die initiale Licht-Injektion wird eine Reflective Shadow Map (RSM) genutzt. Diese wurde erstmals in [DS05] vorgestellt. Hierbei wird die Szene aus Sicht der Lichtquelle gerendert. Im Gegensatz zu traditionellem Shadow-Mapping, wird neben der Tiefe, auch die Farbe der Oberfläche, sowie die Normale abgespeichert.

Jeder Texel der RSM wird als kleine, flächige Lichtquelle interpretiert, in Richtung der Normale n und mit der Intensität $I_p(\omega)$:

$$I_p(\omega) = \frac{\Phi_p}{\pi}(n_p \cdot \omega)^+ \tag{9}$$

Die hier aufgeführte Formel ist nicht das Original aus [KD10] sondern eine korrigierte Fassung aus [Kir10b].

Φ_p ist die Strahlungsleistung des Lichts, berechnet aus der Farbe der Oberfläche und Farbe, sowie Stärke der Lichtquelle. $(n_p \cdot \omega)^+$ ist das Skalarprodukt zwischen Lichtvektor und Normale, auf den positiven Bereich beschränkt.

Für die Repräsentation in SH, werden die ersten beiden Bänder genutzt, was in vier Koeffizienten resultiert (c_0, c_1, c_2, c_3). Die Normale wird in eine in Richtung der Normalen zeigende Kosinus Verteilung projiziert. In [Kap09] wird dafür eine analytische Form angegeben:

$$c_0 = \frac{1}{2\sqrt{\pi}}, \qquad c_1 = -\frac{\sqrt{3}y}{2\sqrt{\pi}}, \qquad c_2 = \frac{\sqrt{3}z}{2\sqrt{\pi}}, \qquad c_3 = -\frac{\sqrt{3}x}{2\sqrt{\pi}} \tag{10}$$

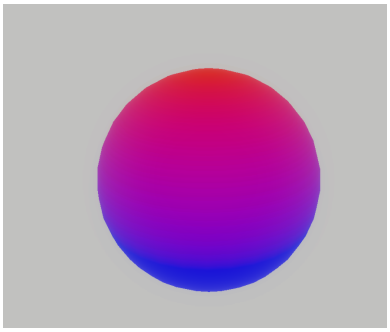


Abbildung 2: Visualisierung des in SH projizierten Up-Vektors, nach Formel (10)

Der Vektor der resultierenden Koeffizienten wird normalisiert. Sie stellen die Richtungseigenschaften des reflektierten Lichts dar. Bei der Strahlungsleistung Φ_p ist zu beachten, dass diese unabhängig von den Eigenschaften der gerenderten RSM sein sollte. Dies bedeutet, dass die Pixelanzahl und der abgedeckte Bereich beachtet werden müssen. Dies wird erreicht, indem mit dem abgedeckten Bereich multipliziert und anschließend durch die Pixelanzahl geteilt wird.

Mithilfe der Tiefe wird die Weltposition jedes Texels aus der RSM rekonstruiert, woraus sich ergibt, in welche Zelle des Volumens das Texel fällt. Die ausgerechneten Koeffizienten werden in den entsprechenden Zellen aufsummiert.

Falls die Normale eines Texels nicht in Richtung des Mittelpunktes der Zelle zeigt, soll das abgestrahlte Licht stattdessen in die Zelle auf die diese zeigt, injiziert werden. Um dies zu erreichen, wird jeder Texel um eine halbe Zellengröße in Richtung seiner Normale verschoben. Dies verhindert, dass Oberflächen sich selbst beleuchten.

3.4 Geometrie-Injektion

Durch die Injektion von Informationen aus der RSM und dem G-Buffer kann eine grobe Repräsentation der Szene erzeugt werden. Dafür werden die Pixelwerte, ähn-

lich wie in Abschnitt 3.3, in ein zusätzliches Geometrie-Volumen injiziert. Hierbei werden nur Position sowie Normale berücksichtigt und keine spektralen Daten gespeichert. Dies resultiert in einem Volumen, welches für jeden Punkt speichert, wie wahrscheinlich in einer Richtung Geometrie vorliegt. Diese Informationen können im Ausbreitungsschritt berücksichtigt werden.

Die genaue Sichtbarkeits-Wahrscheinlichkeit pro Texel kann wie folgt berechnet werden [KD10]:

$$B(\omega) = A_s s^{-2} (n_s \cdot \omega)^+ \tag{11}$$

Hierbei ist A_s die Fläche des Texel, s die Gittergröße des Volumens und n_s die Normale des Texel. Für die Projektion in SH wird ebenfalls Formel (10) verwendet.

3.5 Ausbreitung

Das Ausbreitungsverfahren berechnet schrittweise die Ausbreitung des Lichts. Die erste Iteration verwendet das initiale LPV als Input, nachfolgend wird immer das Ergebnis des vorherigen Schrittes verwendet. Am Ende werden alle Schritte aufaddiert, was in dem finalen Volumen resultiert, welches für den Rendering-Schritt benutzt wird.

Die folgende Beschreibung erklärt den Algorithmus für ein einzelnes Set von Koeffizienten. Bei der tatsächlichen Umsetzung wird er einmal für alle drei Volumen (RGB) angewendet.

Für jede Zelle wird berechnet, wie viel Licht aus ihren sechs direkten Nachbarn auf sie fällt. Hierfür wird pro Seite der Zelle der Lichtfluss berechnet, welcher von der Mitte der Nachbarzelle auf die betrachtete Seite fällt.

Sei $I(\omega)$ die richtungsabhängige Intensität der Nachbarzelle und $V(\omega)$ eine Sichtbarkeitsfunktion. $V(\omega)$ ist für jeden Richtungsvektor vom Mittelpunkt der Nachbarzelle definiert und ist eins, falls er die betrachtete Seite schneidet, sonst null. Der komplette Lichtfluss aus der Nachbarzelle auf die betrachtete Seite ist das Integral über alle Richtungen: $\Phi_f = \int_\omega I(\omega) V(\omega) d\omega$.

Für die effiziente Berechnung in Echtzeit wird dieses Integral approximiert. Statt dem Integral über Sichtbarkeit und Lichtfluss, wird der Raumwinkel $\Delta\omega_f = \int_\omega = V(\omega) d\omega$ genutzt. Dieser wird kombiniert mit dem Lichtfluss in mittlerer Richtung $I(\omega_c)$. Der finale Wert wird berechnet als $\Delta\omega_f \cdot I(\omega_c)$ (korrigierte Formel aus [Kir10b]). Die Raumwinkel können vorberechnet werden. Eine genaue Herleitung der Werte findet sich in [Kir10a].

Der resultierende Wert ist der Lichtfluss, der von der Nachbarzelle auf die Würfeloberfläche fällt. Um diesen Wert in Spherical-Harmonics-Koeffizienten umzurechnen, wird die Richtung der Seite, der Vektor von Mittelpunkt der Zelle zum Mittelpunkt der Seite, mithilfe von Gleichung (10) in SH projiziert. Die resultierenden Koeffizienten werden mit dem Lichtfluss skaliert. Die Koeffizienten werden für alle sechs Nachbarzellen und deren sechs Würfelseiten berechnet und aufaddiert.

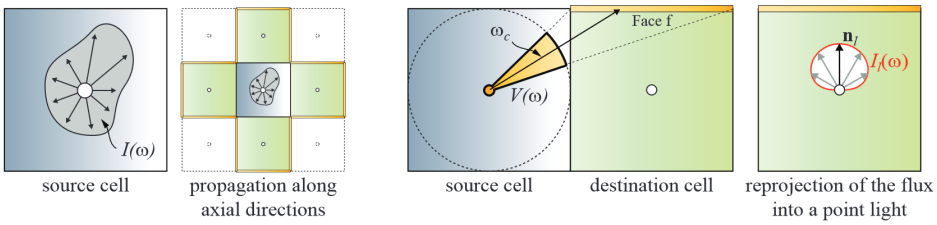


Abbildung 3: Ausbreitungs-Schema, aus [KD10]

3.5.1 Verdeckung

Falls gewünscht, kann im Ausbreitungsschritt die Verdeckung durch Geometrie approximiert werden. Hierfür wird aus dem Geometrie-Volumen (siehe 3.4) die Blockierungs-Wahrscheinlichkeit am Übergang zwischen der Zelle und Nachbarzelle für die Richtung der Lichtausbreitung berechnet. Der Lichtstrom wird anschließend durch die Skalierung mit dieser abgeschwächt. In der ersten Iteration der Ausbreitung wird die Verdeckung nicht beachtet, um zu verhindern, dass Geometrie sich selbst verdeckt.

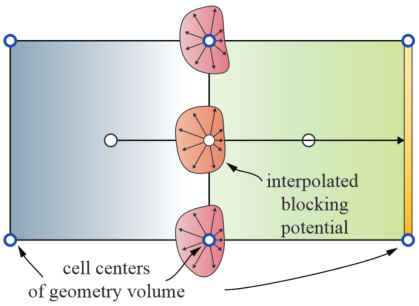


Abbildung 4: Verdeckung, aus [KD10]

3.5.2 Sekundäre Reflektionen

Der Ausbreitungsschritt kann modifiziert werden, um mehrere diffuse Spiegelungen zu approximieren. Um dies zu erreichen, wird das Geometrie-Volumen beim Auslesen eine Zelle in Richtung der Ausbreitung verschoben. Dies ergibt die Wahrscheinlichkeit $B(\omega)$, das Licht an dieser Stelle zu blockieren. Die Wahrscheinlichkeit, dass das Licht hier reflektiert wird, ist $B(-\omega_c)$ [KD10]. Dies ist effektiv das Skalarprodukt zwischen der Ausbreitungsrichtung und der Normalen der Geometrie. Der Wert wird skaliert mit der propagierten Lichtintensität, um die Intensität des reflektierten Lichts zu errechnen. Diese wird auf die Koeffizienten der Ziel-Zelle addiert.

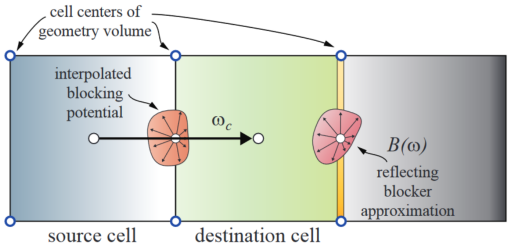


Abbildung 5: Sekundäre Reflektion, aus [KD10]

3.6 Rendering

3.6.1 Diffus

Im Renderschritt wird das einfallende Licht aus dem LPV berechnet, indem zunächst die Position des Pixels im LPV berechnet wird. Anschließend werden die Koeffizienten für RGB aus den 3D-Texturen ausgelesen, wobei diese trilinear interpoliert werden. Die Lichtintensität wird anschließend berechnet, indem die Normale des Pixels mithilfe von Formel (10) in SH-Koeffizienten projiziert wird und das Skalarprodukt zwischen diesen und den Koeffizienten aus dem Volumen gebildet wird.

3.6.2 Spekular

Der spekulare Term kann approximiert werden, indem man in Richtung des Reflektions-Vektors R, das LPV samplet und die Intensität in Richtung $-R$ ausrechnet. Die Sample-Werte werden gemittelt.

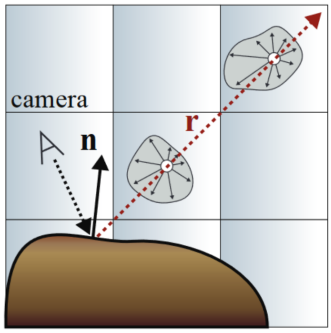


Abbildung 6: Reflektion aus dem LPV, aus [KD10]

Diese Berechnung ist eine Approximation für das einfallende Licht. Jedoch muss die BRDF noch beachtet werden, um die Oberflächeneigenschaften mit einzubringen. Hierfür kann die Split-Sum-Approximation genutzt werden [Kar13].

Diese stellt eine Möglichkeit dar, die Cook-Torrance-BRDF [CT82] für indirekte Beleuchtung anzunähern. Das Integral über die obere Hemisphäre des betrachteten Punktes wird gebildet wie folgt:

$$\int_{l\in\Omega} L_i(l)f(l,v)cos\theta_l dl \tag{12}$$

Für jede Richtung l wird das Licht $L_i(l)$ mit der BRDF $f(l,v)$ verrechnet, wobei v der View-Vektor ist. Das Integral kann mithilfe von Importance-Sampling gelöst werden:

$$\int_{l\in\Omega} L_i(l)f(l,v)cos\theta_l dl \approx \frac{1}{N}\sum_{k=1}^{N}\frac{L_i(l_k)f(l_k,v)cos\theta_{l_k}}{p(l_k,v)} \tag{13}$$

Hier werden N Samples gemittelt, wobei sie nach der Wahrscheinlichkeitsdichtefunktion $p(I_k,v)$ gewichtet werden. Die Split-Sum-Approximation nähert diesen Ausdruck an, indem die Summen des einfallenden Lichts und die BRDF getrennt aufsummiert und danach multipliziert werden.

$$\frac{1}{N}\sum_{k=1}^{N}\frac{L_i(l_k)f(l_k,v)cos\theta_{l_k}}{p(l_k,v)} \approx (\frac{1}{N}\sum_{k=1}^{N}L_i(l_k))(\frac{1}{N}\sum_{k=1}^{N})\frac{f(l_k,v)cos\theta_{l_k}}{p(l_k,v)}) \tag{14}$$

Dadurch sind Licht und BRDF getrennt. Das einfallende Licht wird aus dem LPV berechnet, während die BRDF vorberechnet wird.

Für die BRDF muss über den Winkel zwischen Normale und View-Vektor $cos\theta_v$, dem Oberflächenparameter Rauheit und der Reflektivität F_0 integriert werden. Durch Umformung kann F_0 jedoch aus dem Integral gezogen werden:

$$\int_{l\in\Omega} f(l,v)cos\theta_l dl =$$
$$F_0\int_{l\in\Omega}\frac{f(l,v)}{F(v,h)}(1-(1-v\cdot h)^5)cos\theta_l dl + \frac{f(l,v)}{F(v,h)}(1-v\cdot h)^5 cos\theta_l dl$$

Diese Integrale können vorberechnet werden und in einer 2D-Look-Up-Table gespeichert werden. Zur Laufzeit werden die Werte, abhängig von roughness und $cos\theta_v$ ausgelesen. Die Ergebnisse werden mit dem berechneten F_0 und den Lichtwerten kombiniert, um die finale Beleuchtung zu erhalten.

3.7 Stabilisierung

Unter Bewegung von Kamera oder Licht kann es zu Flackern kommen. Um dies zu verhindern, werden mehrere Techniken verwendet.

Bei Kamera Bewegungen wird das Volumen mit der Kamera mit bewegt, was aufgrund der räumlichen Aufteilung des Lichts zu Artefakten führt. Um dem entgegenzuwirken, wird das Volumen in Zellen-großen Schritten bewegt.

Da die Auflösung der RSM begrenzt ist, kann es durch das begrenzte Sampling der injizierten Lichtquellen bei Veränderungen zu Flackern kommen. Eine Lösung für dieses Problem ist die Verwendung einer höher aufgelösten RSM. Da hohe Auflösungen jedoch einen höheren Berechnungsaufwand mit sich bringen, wird das Volumen zusätzlich über die Zeit interpoliert. Dafür wird das Volumen in jedem Schritt auf die vorherige Position reprojiziert und mit dem vorherigen Volumen linear interpoliert. Dadurch werden Änderungen über einen größeren Zeitraum gestreckt, was für das Auge weniger sichtbar ist [Eng11].

4 Screen-Space-Algorithmen

Screen-Space Techniken werden auf das Bild angewendet, nachdem die Szene fertig gezeichnet wurde. Zusätzlich zum Farbbild können weitere Informationen wie Tiefe und Normalen verwendet werden.

Im Folgenden werden zwei Screen-Space-Techniken zur Approximation von Global-Illumination vorgestellt.

4.1 Ambient-Occlusion

Ambient-Occlusion (AO) ist eine Annäherung für die Verdeckung eines Punktes durch umliegende Geometrie. Für die Berechnung wird die vordere Hemisphäre eines gegebenen Punktes auf Sichtbarkeit geprüft. Die Formel laut [AMHH18] ist wie folgt:

$$k_A(p) = \frac{1}{\pi} \int_{l \in \Omega} v(p, l)(n \cdot l)^+ dl \tag{15}$$

v ist eine Funktion, die die Sichtbarkeit modelliert. Sie gibt null zurück, falls der Strahl von p in Richtung l blockiert ist und eins, falls nicht.

$k_A(p)$ liegt zwischen null und eins und repräsentiert den Anteil der Oberfläche der vorderen Halbkugel, welcher nicht verdeckt ist, gewichtet mit dem Kosinus des Winkels zwischen Richtung und Normale. Dieser Wert wird auf den indirekten Beleuchtungsterm multipliziert.

4.2 Screen-Space-Ambient-Occlusion

Screen-Space-Ambient-Occlusion(SSAO) ist eine Technik zur Berechnung von AO. Hierfür werden zufällige Sampling-Vektoren auf die Welt-Position des Pixels gerechnet und anschließend in Screen-Space projiziert. Die Tiefe des ursprünglichen Punktes wird mit der an der projizierten Stelle ausgelesenen Tiefe verglichen. Ist die ausgelesene Tiefe niedriger als die Ursprüngliche, wird das Sample als verdeckt betrachtet.

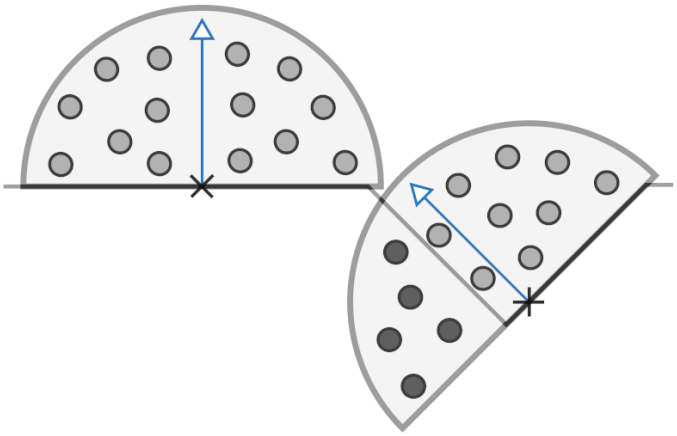


Abbildung 7: Halbkugeln mit Sample-Punkten, verdeckte Samples sind dunkel dargestellt. Nachbildung aus [Cha13]

In Formel (15) sind die Richtungen gewichtet, da sie aus der Reflektions-Gleichung hergeleitet sind [AMHH18]. Dort wird angenommen, dass das einfallende Licht konstant ist. Falls SSAO jedoch mit Techniken kombiniert wird, welche diese Gewichtung bereits bei der Berechnung des einfallenden Lichts berücksichtigen, wird dieser Term nicht beachtet. Dies ist bei der Kombination mit LPVs der Fall.

In [FM08] wird vorgeschlagen, die Samples abzuschwächen je weiter die Distanz von Blocker und verdecktem Punkt entfernt ist. Dadurch wird verhindert, dass Geometrie im Vordergrund in den AO-Faktor des Hintergrunds eingerechnet wird.

4.2.1 Near-Field-Licht-Transport

[TR] stellt eine Erweiterung von SSAO vor, welche eine diffuse Licht-Reflektion in Screen-Space approximiert. Jeder gesampelte Pixel wird als kleine Lichtquelle betrachtet. Das zusätzliche Licht wird wie folgt berechnet:

$$L_{ind}(P) = \sum_{i=0}^{N} \frac{p}{\pi} L_{pixel}(1 - V(\omega_i)) \frac{A_s cos\theta_{si} cos\theta_{ri}}{d_i^2} \tag{16}$$

Hierbei ist d_i die Distanz zwischen Lichtquelle und Oberfläche. A_s ist die Fläche des Senders und wird berechnet mit: $A_s = \pi r_{max}^2 / N$, wobei N die Anzahl der Samples ist und r der Sample-Radius. $cos\theta_{si}$ ist der Winkel zwischen der Richtung des Lichts und der Normale des Senders. Dieser Term wird berücksichtigt, um zu verhindern, dass Punkte ihre hintere Halbkugel beleuchten. $cos_\theta ri$ ist der Winkel zwischen Lichtrichtung und Normale des Empfängers. $V(\omega_i)$ ist die Sichtbarkeitsfunktion in Richtung Sample i, ähnlich wie in Gleichung (15).

Konzeptuell ist diese Reflektion eine Ergänzung zur Beleuchtung aus dem LPV. In [May18] wird das Verhältnis von Far- und Near-Field Beleuchtung aufgeführt.

Far-Field-Beleuchtung ist die Beleuchtung aus der Entfernung, hier wird diese mithilfe des LPV approximiert. Far-Field-Beleuchtung wird durch die direkte Umgebung verdeckt (AO). Near-Field-Beleuchtung ist der Anteil aus der direkten Umgebung. Sie ist effektiv die Beleuchtung von der Geometrie, welche das entfernte Licht blockiert. Hieraus ist ersichtlich, dass beide sich ergänzen. Beide Beiträge können also getrennt berechnet und aufsummiert werden.

4.3 Directional-Occlusion

Ambient-Occlusion ist eine starke Vereinfachung von globaler Beleuchtung. Die Geometrie wird zwar miteinbezogen, allerdings komplett entkoppelt von der tatsächlichen Beleuchtung. Im Unterschied dazu bezieht Directional-Occlusion die Richtung der Verdeckung mit ein. Dadurch können Effekte, wie farbige Schatten, erzielt werden [TR]. Im Folgenden werden einige Techniken zur Approximation von Directional-Occlusion vorgestellt.

4.3.1 Bent-Normals

Eine Bent-Normal ist die mittlere, unverdeckte Richtung eines Punktes [AMHH18]. Sie wird wie folgt berechnet:

$$n_{bent} = \frac{\int_{l\in\Omega} lv(l)(n\cdot l)^+ dl}{||\int_{l\in\Omega} lv(l)(n\cdot l)^+ dl||} \tag{17}$$

Die Samples in Richtung l werden akkumuliert, falls sie unverdeckt sind ($v(l) = 1$). Dabei werden sie mit dem Kosinus des Winkels zwischen Richtung l und der Normale des Punktes n gewichtet. Der Term im Nenner dient zur Normalisierung des Vektors. Die Bent-Normal kann zusätzlich zum AO-Term berechnet werden. Bei der Schattierung kann die Bent-Normal statt der eigentlichen Normale genutzt werden, um akkuratere Ergebnisse zu erzielen.

4.3.2 Spherical-Harmonic-Occlusion

Screen-Space-Spherical-Harmonic-Occlusion (S_3HO), vorgestellt in [HSS11], ist eine Erweiterung des üblichen SSAO-Algorithmus um Richtungsinformationen. Hierfür werden die unverdeckten Samples in Spherical-Harmonics projiziert. Im Beleuchtungsschritt werden diese Sichtbarkeits-Koeffizienten mit den Beleuchtungskoeffizienten integriert, was in SH einem Skalarprodukt entspricht. So wird der Verdeckungsfaktor für jeden Farbkanal einzeln berechnet. Eine ähnliche Technik wird in [O'D11] verwendet, um die Sichtbarkeit in einer expliziten Richtung zu prüfen. Hierfür wird die Richtung in Spherical-Harmonics projiziert und über die Sichtbarkeits-Koeffizienten integriert.

4.4 Screen-Space-Reflections

Screen-Space-Reflections (SSR) ist eine Technik zur Berechnung von Reflektionen mithilfe der bereits vorhandenen Bildinformationen [Gia16]. Hierfür wird aus View- und Normalen-Vektor der Reflektions-Vektor berechnet. Anschließend wird der Schnittpunkt mit dem Depth-Buffer mithilfe von Raymarching gesucht. Dies geschieht indem der Reflektions-Vektor schrittweise abgetastet und auf den Depth-Buffer projiziert wird. Ist die Tiefe im Depth-Buffer geringer als die des Samples des Reflektions-Vektors, wurde der Schnittpunkt gefunden. Die Farbe der Reflektion wird ausgelesen, indem der Schnittpunkt auf den vorherigen Frame reprojiziert wird.

4.5 Filterung

Aufgrund der Berechnungszeit ist die Anzahl der möglichen Samples für Post-Processing-Effekte begrenzt. [FM08] gibt eine typische Anzahl von 8–32 Samples für SSAO an. Dies führt zu offensichtlichen Artefakten, in Form von Bändern. Um dies zu verhindern, werden die Samples pro Pixel zufällig rotiert, wodurch das Band-Muster durch Rauschen ersetzt wird.

Um Rauschen herauszufiltern, werden häufig Blur-Filter benutzt. In [FM08] wird ein Kanten erhaltender Gauß-Filter vorgeschlagen. Dieser sampelt Normale und Tiefe des mittleren Pixels. Für alle weiteren Samples werden ebenfalls Normale und Tiefe ausgelesen. Das Gewicht des Samples wird auf null reduziert, falls die Differenz der Tiefe oder das Skalarprodukt der Normalen unter einem festgelegten Schwellwert liegt. Die Gewichte müssen am Ende renormalisiert werden, um die fehlenden Samples auszugleichen. Der Filter glättet das Rauschen und erhält Details.

4.5.1 Reprojektion

In [Kar14] wird eine Anti-Aliasing Technik präsentiert, welche durch Reprojektion Informationen aus dem vorherigen Bild nutzt. Durch Sub-Pixel-Variationen in der Projektions-Matrix von Bild zu Bild ist es so möglich, das Bild durch mehrfache Samples zu stabilisieren. Neben der Anwendung als Anti-Aliasing ist es auch möglich, dies für andere Techniken zu nutzen, welche auf Sampling basieren. Effektiv ermöglicht dies die Verteilung von Samples über mehrere Bilder.

In [Wes14] wird vorgeschlagen, Reprojektion zu benutzen, um nicht nur die Samples über Zeit zu akkumulieren, sondern auch den genutzten Blur. Hierfür wird das erzeugte Bild erst mit einem Blur-Filter gefiltert, anschließend mit einem temporalen Filter und danach nochmals mit einem Blur-Filter. Dies ermöglicht temporal stabile, weichgezeichnete Ergebnisse bei einer niedrigen Auflösung und Sample-Zahl.

5 Implementierung

5.1 Rendering-Framework

Für die Implementierung der Algorithmen wurde ein selbstgeschriebener Renderer benutzt, welcher in C++ geschrieben ist. Für das Rendering wird OpenGL genutzt, für das Management von Fenster und Inputs, wird GLFW verwendet. Als Mathe-Bibliothek wird die GLM Library benutzt. Das Laden von Modellen geschieht mithilfe von Assimp, Bilder werden mit stb-image und Nvidias nv-dds Bibliotheken geladen. Das UI basiert auf Dear ImGui.

5.2 Implementierung: Light-Propagation-Volumes

Die Ausmaße des Light-Propagation-Volume sind über Auflösung und Skalierung festgelegt. Zur Speicherung der Licht-Koeffizienten dienen 3D-Texturen, welche ein 16-Bit Float-Format nutzen. Pro Farbkanal werden vier Koeffizienten benötigt, daher werden diese in drei Texturen mit jeweils vier Kanälen gespeichert. Die Auflösung der Texturen beträgt standardmäßig 32^3 Pixel. Die Skalierung legt fest, wie viele Meter pro Pixel benutzt werden. Der Wert ist standardmäßig auf einen Pixel pro Meter gesetzt. Höhere Werte sorgen dafür, dass das Volumen einen größeren Bereich abdeckt, jedoch auf Kosten der Präzision. Für die RSM wird eine Auflösung von 256x256 Pixeln benutzt.

Das Geometrie-Volumen hat dieselbe Auflösung wie das Licht-Volumen, allerdings wird hier nicht die Farbe gespeichert, daher wird es in einer einzigen Float-3D-Textur mit vier Kanälen gespeichert.

Das Volumen wird mit der Kamera bewegt. Hierfür wird ein Offset verwendet, welcher bei allen Umrechnungen von Welt- in Volumen-Koordinaten genutzt wird. Wie in Sektion 3.7 beschrieben, wird das Offset dabei auf ein Vielfaches der Skalierung beschränkt, um Artefakte unter Bewegung zu vermeiden.

Die Kamera befindet sich nicht im Zentrum des Volumens. Wie in [Eng11] beschrieben, kann das Volumen entlang der Sichtrichtung verschoben werden, damit der Großteil des berechneten Bereichs im Sichtfeld liegt. Es wird ein 80/20 Verhältnis benutzt, wobei 80% des Volumens im Sichtfeld liegen und 20% dahinter.

```
m_lpvOffset = floor(camera->getPosition() /
↪  m_volumeScale) * m_volumeScale +
↪  floor(camera->getViewDirection() * 0.3f *
↪  m_volumeScale / m_volumeScale) *
↪  m_volumeScale;
```

Listing 1: C++ Code zur Berechnung des Volumen-Offset

5.2.1 Spherical-Harmonics-Tools

Die wichtigste SH-Funktion ist die Projektion eines Richtungs-Vektors in Spherical-Harmonics-Koeffizienten. Dafür wird die folgende GLSL-Funktion verwendet, welche eine direkte Umsetzung von Gleichung (10) darstellt.

```
vec4 calcSHCoefficients(vec3 dir){
        float denom = 2.f * sqrt(pi);
        float c0 = 1.f                  / denom;
        float c1 = -sqrt(3.f) * dir.y / denom;
        float c2 =  sqrt(3.f) * dir.z / denom;
        float c3 = -sqrt(3.f) * dir.x / denom;

        return normalize(vec4(c0, c1, c2, c3));
}
```

Listing 2: GLSL-Funktion für die Berechnung von SH-Koeffizienten

Eine nützliche Hilfsfunktion ist die Auswertung von SH-Koeffizienten in einer gewissen Richtung. Hierfür wird die Richtung in Koeffizienten umgerechnet. Anschließend wird das Integral beider Funktionen gebildet, was durch das Skalarprodukt der beiden Koeffizienten-Mengen berechnet wird.

```
float evaluateDirection(vec3 dir, vec4 coeff){
        return max(dot(calcSHCoefficients(dir), coeff), 0.f);
}
```

Listing 3: GLSL-Funktion zur Auswertung von SH-Koeffizienten in gegebener Richtung

5.2.2 Reflective Shadow Map

Die Reflective Shadow Map ist der erste Schritt, da sie der Input für die Injektion ist. Mit der Position und den Ausmaßen des LPV kann dessen Bounding-Box berechnet werden. Die Light-Matrix der RSM wird mithilfe eines Algorithmus für Cascaded-Shadow-Mapping berechnet, welcher eine optimal zugeschnittene Matrix aus einer Bounding-Box berechnet. Der Algorithmus ist aus [Dim07] entnommen.

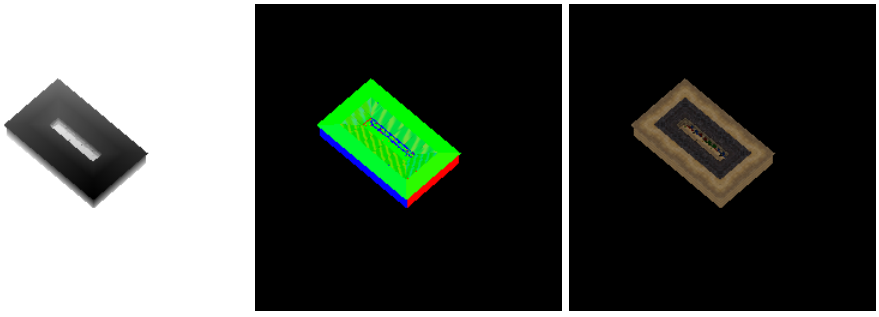
Die View-Matrix wird aus der Lichtrichtung berechnet. Anschließend werden alle Eckpunkte der Bounding-Box mithilfe der View-Matrix und einer Einheitsmatrix, als temporäre Projektionsmatrix, transformiert. Von diesen Punkten werden die maximalen X-, Y- und Z-Koordinaten herausgesucht. Aus den X- und Y- Werten wird eine Crop-Matrix gebildet, welche die Transformation auf den Bereich

der Bounding-Box begrenzt. Die Berechnung ist wie folgt:

$$\begin{pmatrix} S_x & 0 & 0 & O_x \\ 0 & S_y & 0 & O_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{aligned} S_x &= \frac{2}{M_x - m_x} \\ S_y &= \frac{2}{M_y - m_y} \\ O_x &= -0.5(M_x + m_x)S_x \\ O_y &= -0.5(M_y + m_y)S_y \end{aligned} \tag{18}$$

Hierbei sind m_x und m_y die minimalen X- bzw. Y-Werte und M_x und M_y die Maxima. Aus den Z-Werten wird eine orthographische Projektions-Matrix P_z berechnet, mit den Werten als Near und Far. Die finale Projection-Matrix ist $P = CP_z$.

Der Framebuffer der RSM beinhaltet eine Textur für die Farbe der Oberfläche als Unsigned-Byte-Textur mit drei Kanälen, sowie eine Signed-Byte Textur mit drei Kanälen für die Normalen in Weltkoordinaten. Zudem besitzt er einen Depth-Buffer, aus dem später die Welt-Position rekonstruiert wird. Normal-Mapping wird für die RSM nicht berechnet, da die feinen Unterschiede aufgrund der geringen Präzision des Algorithmus nicht sichtbar sind.



(a) Tiefe **(b)** Normalen **(c)** Albedo

Abbildung 8: RSM-Buffer, Wertebereich des Tiefenbuffer wurde zur Visualisierung zusammengestaucht

5.2.3 Injektion

Für die Injektion des Lichts wird die Position jedes Pixels der RSM in das Koordinatensystem des LPV umgerechnet. Anschließend werden Stärke und Richtung des reflektierenden Lichts berechnet und in Koeffizienten umgerechnet. Diese werden auf die entsprechende Stelle in der 3D-Textur addiert.

Bei der normalen Rasterisierung kann man nicht frei wählen, wohin die Daten geschrieben werden. Um die Koeffizienten an die richtige Stelle zu schreiben, wird ein VBO angelegt, welcher so viele Vertices enthält wie die RSM Pixel. Für jedes Pixel-Vertex Paar wird die Weltposition aus der Tiefe der RSM ausgerechnet. Die Weltposition wird anschließend in das Koordinatensystem des Volumens umgerechnet.

```
vec3 worldToCellPosition(vec3 worldPosition, vec3
↪  offset, float scale){
        return (worldPosition - offset) / scale +
        ↪  vec3(0.5f);
}
```

Listing 4: GLSL-Funktion zur Umrechnung von Welt- in Volumen-Koordinaten

Der Vertex wird im Geometry-Shader an die errechnete XY-Volumen-Position geschoben. Die korrekte Tiefe wird durch `gl_Layer` festgelegt. Die Vertices werden mithilfe von `GL_POINTS` gerendert mit einem Radius von einem Pixel. Dies sorgt dafür, dass jeder Vertex mit genau einem Pixel rasterisiert wird.

Im Fragment-Shader werden die Koeffizienten des injizierten Lichts ausgerechnet. Bei diesem Vorgang wird additiv geblendet. Somit werden alle Koeffizienten, die in eine Zelle Fallen, aufaddiert.

```
void main(){
        vec3 light = texelFetch(albedoSampler, p_samplePosition,
        ↪  0).rgb * light.strength * light.color;
        float dot = max(dot(p_normal, light.direction), 0.f);

        vec4 coeff = calcSHCoefficients(p_normal);

        float nTexelsRSM = textureSize(albedoSampler, 0).x *
        ↪  textureSize(albedoSampler, 0).y;
        float lightWeight = u_shadowArea / nTexelsRSM;

        coeff *= dot * lightWeight;
        o_coeffR = coeff * light.r;
        o_coeffG = coeff * light.g;
        o_coeffB = coeff * light.b;
}
```

Listing 5: Fragment-Shader für die Injektion der Licht-Koeffizienten

Die Geometrie-Injektion nutzt die gleiche Methode. Allerdings werden hier zwei Quellen benutzt: die RSM und der G-Buffer. Der G-Buffer wird vor der Injektion auf ein Viertel der ursprünglichen Auflösung skaliert, da die Injektion bei großen Pixelmengen langsam ist.

5.2.4 Ausbreitung

Jede Iteration der Ausbreitung wird in einen eigenen Framebuffer geschrieben, wobei das Ergebnis eines Schrittes der Input für den nächsten ist. Das finale Volumen ist die Summe aller Iterationen.

Um jeden Pixel in der 3D-Textur zu verarbeiten wird ein Screen-Filling-Quad gerendert. In einem Geometry-Shader wird dieses für jeden Slice der Textur kopiert und mit Hilfe von `gl_Layer` an die entsprechende Tiefe geschrieben.

Die folgende GLSL-Funktion berechnet im Pixel-Shader das Licht, welches von einer Nachbarzelle in die Zielzelle fällt, wobei Verdeckung und indirekte Reflektion berücksichtigt werden. Diese Funktion wird für jede Zelle für alle sechs direkten Nachbarn ausgeführt. Die Ergebnisse werden aufaddiert. Die Werte für die Sichtbarkeit stammen aus [Kir10a].

```
vec4 propagateDirectionOccluded(vec3 cellPos, vec3
↪  neighbourDirection, sampler3D coeffSampler, vec3 cellLenght,
↪  float geometry){

        vec4 coeff = vec4(0.f);
        vec3 lightDirection = -neighbourDirection;
        vec4 coeffTexel = texelFetch(coeffSampler, ivec3(cellPos *
        ↪  u_volumeRes), 0) * geometry;

        for(int face = 0; face < 6; face++){
                vec3 faceDirection =
                ↪  offset[face];

                float dot = dot(lightDirection, faceDirection);
                float visibility = 1.f - abs(dot) < 0.1f ? 0.4f :
                ↪  0.42f;
                visibility = dot < -0.1 ? 0 : visibility;

                vec3 centralVector = normalize(lightDirection *
                ↪  0.82f + faceDirection);

                vec4 propagatedLight = (visibility / pi)*
                ↪  evaluateDirection(centralVector, coeffTexel) *
                ↪  calcSHCoefficients(faceDirection);
                coeff += propagatedLight;
        }

        vec4 bounceCoeff = texture(coeffGeometrySampler, cellPos +
        ↪  neighbourDirection * cellLenght, 0);
        vec4 bounce = evaluateDirection(lightDirection,
        ↪  bounceCoeff) * dot(calcSHCoefficients(lightDirection),
        ↪  coeff) * bounceCoeff;
        coeff += bounce;

        return coeff;
}
```

Listing 6: GLSL-Funktion zur Berechnung des Lichtes was von einer Nachbarzelle in eine Zielzelle fällt

5.2.5 Debug-Visualisierung

Zum Debuggen können die Koeffizienten visualisiert werden. Hierfür wird ein Vertex-Array-Object (VAO) angelegt, welches für jeden Texel der Textur einen Vertex enthält. Im Geometry-Shader werden die Vertices an die Stelle der zugehörigen Zelle in Welt-Koordinaten geschoben und zu einem Würfel erweitert. Im Fragment-Shader werden die Koeffizienten der Zelle gelesen und für diffuse Beleuchtung genutzt.

Da die Visualisierung eigene Shader nutzt, muss sie getrennt vom Deferred-Pass geschehen. Nachdem dieser fertig gerendert wurde, wird der Depth-Buffer aus dem G-Buffer in den Haupt-Color-Buffer kopiert. Dies geschieht mit dem Befehl `glBlitFramebuffer`. Anschließend kann die Visualisierung mit korrekter Verdeckung gerendert werden.

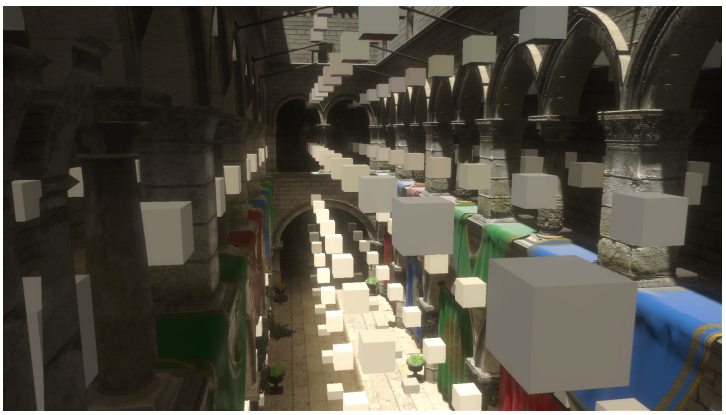


Abbildung 9: Visualisierung der Koeffizienten

Dieselbe Technik kann verwendet werden, um das Geometrie-Volumen darzustellen.

5.2.6 Rendering

Im Rendering-Schritt werden die Pixel mit Hilfe des LPVs schattiert. Dafür wird ein Screen-Filling-Quad gerendert. Das Ergebnis wird in den Color-Buffer geschrieben und dabei additiv geblendet.

Im Fragment-Shader werden die Koeffizienten für den diffusen Anteil aus den entsprechenden Texturen gelesen und trilinear-interpoliert. Um das einfallende Licht zu berechnen, werden sie in Richtung der Normale evaluiert.

```
vec3 cellPosition = worldToCellPosition(pos, u_offset,
↪  u_volumeRes.x);

vec4 coeffR = texture(coeffRSampler, cellPosition, 0);
vec4 coeffG = texture(coeffGSampler, cellPosition, 0);
vec4 coeffB = texture(coeffBSampler, cellPosition, 0);

vec3 incomingLight;
incomingLight.r = evaluateDirection(N, coeffR);
incomingLight.g = evaluateDirection(N, coeffG);
incomingLight.b = evaluateDirection(N, coeffB);
```

Listing 7: GLSL-Ausschnitt zur Berechnung des einfallenden, diffusen Lichts aus dem LPV

Für den Diffusen Anteil wird das Lambertsche Beleuchtungsmodell genutzt. Für den spekularen Anteil wird aus Normal- und View-Vektor der Reflektions-Vektor berechnet. Anschließend werden entlang dieses Vektors Koeffizienten ausgelesen und in Richtung des negativen Reflektions-Vektors ausgewertet. Das Ergebnis wird gemittelt.

```
const float samples = 3;
vec3 specular = vec3(0.f);
for(int i = 1; i <= samples; i++){
        vec3 offset = i * reflection;
        cellPosition = worldToCellPosition(pos + offset, u_offset,
        ↪  u_volumeRes.x);

        coeffR = texture(coeffRSampler, cellPosition);
        coeffG = texture(coeffGSampler, cellPosition);
        coeffB = texture(coeffBSampler, cellPosition);

        specular.r += evaluateDirection(-reflection, coeffR);
        specular.g += evaluateDirection(-reflection, coeffG);
        specular.b += evaluateDirection(-reflection, coeffB);
}
specular /= samples;
```

Listing 8: GLSL-Ausschnitt zur Berechnung des einfallenden, spekularen Lichts

Falls das LPV mit SSR kombiniert wird, wird der spekulare Anteil des LPVs mit $1 - SpecularOcclusion$ multipliziert. Das bedeutet, die Reflektion wird, falls möglich, aus den Screen-Space-Daten berechnet und an Stellen wo keine Informationen vorliegen, mit der Reflektion aus dem LPV aufgefüllt. Dies verhindert, dass die Reflektion doppelt gerendert wird.

Die Integrale der Split-Sum-Approximation werden anfangs einmal in einem

extra Shader berechnet und in eine Textur geschrieben. Hierfür wird eine Float-Textur mit 16-Bit und zwei Channels genutzt.

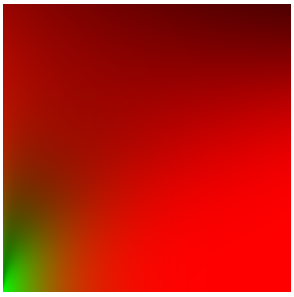
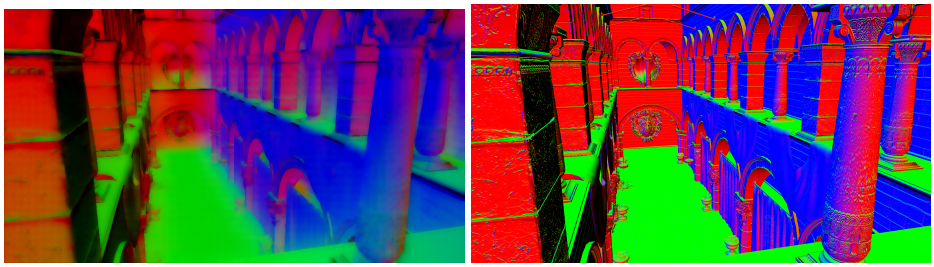


Abbildung 10: LUT für den indirekten spekularen Anteil der BRDF

5.3 SSAO

SSAO wird vor dem Beleuchtungsschritt berechnet. Es werden drei Buffer beschrieben: ein Unsigned-Byte-Buffer für den AO-Faktor mit einem Kanal und ein 32-Bit-Farb-Buffer für das reflektierte diffuse Licht. Der Farb-Buffer benutzt als Format GL_R11F_G11F_B10F, was ein komprimiertes RGB-Format ist. Dies ermöglicht größere Werte als ein Byte-Format, kostet jedoch weniger Speicher und Bandbreite als ein volles Float-Format und ist daher gut für Farb-Buffer geeignet. Zudem gibt es einen optionalen 16-Bit Float-Buffer, mit vier Kanälen, in den die Daten für Directional-Occlusion geschrieben werden. Je nach Technik sind dies die Bent-Normals oder die Sichtbarkeits-SH-Koeffizienten.



(a) berechnete Bent-Normals **(b)** G-Buffer

Abbildung 11: Vergleich der Normalen-Buffer

Der SSAO-Pass wird auf halber Auflösung berechnet, um Rechenzeit zu sparen. Dabei werden auch die Input-Buffer auf die niedrigere Auflösung skaliert. Dadurch wird zusätzlich Bandbreite gespart, zudem ist die Texture-Hit-Rate besser.

Abbildung 12: ungefilterter SSAO-Buffer

Die Blur-Operationen sowie die Reprojektion werden ebenfalls auf halber Auflösung berechnet. Für die Reprojektion wird ein Velocity-Vektor benötigt. In diesem ist pro Pixel der Vektor zur Position des vorherigen Frames abgespeichert. Der Velocity-Buffer wird in einem Shader, zusammen mit dem Rest des G-Buffer, berechnet. Zur Berechnung sind Model- sowie View- und Projection-Matrix des vorherigen Frames nötig, diese werden bereits auf der CPU zusammen multipliziert, um im Shader Rechenarbeit zu sparen. Im Vertex-Shader wird neben der aktuellen auch die vorherige Position des Vertex berechnet.

```
p_posProjCurr = u_modelViewProjectionCurr * vec4(i_pos, 1.f);
p_posProjLast = u_modelViewProjectionLast * vec4(i_pos, 1.f);
```

Listing 9: GLSL-Ausschnitt, aktuelle und vorherige Position werden berechnet und an den Fragment-Shader weitergeben

Beide werden an den Fragment-Shader weitergegeben, wobei sie interpoliert werden. Im Fragment Shader muss die perspektivische Division von Hand ausgeführt werden. Dies ergibt die auf das Bild projizierten Positionen. Um den Velocity-Vektor zu berechnen, wird die aktuelle Position von der alten subtrahiert. Da dies in Normalized-Device-Coordinates geschieht, liegen die Werte im Bereich [-1,1]. Der Velocity-Vektor wird in Bild-Koordinaten [0,1] benötigt, daher muss der Wert noch halbiert werden.

```
vec4 posProjLast = p_posProjLast / p_posProjLast.w;
vec4 posProjCurr = p_posProjCurr / p_posProjCurr.w;
o_velocity = (posProjLast.xy - posProjCurr.xy) * 0.5f;
```

Listing 10: GLSL-Ausschnitt, Berechnung des finalen Velocity-Vektors im Fragment-Shader

Für die Reprojektion ist eine hohe Genauigkeit notwendig, daher wird das Ergebnis in einen 16-Bit-Float-Buffer geschrieben.

Als Rauschen für die zufällige per-Pixel-Rotation wird, wie in [MG16] vorgeschlagen, Blue-Noise verwendet. Dies hat im Gegensatz zu White-Noise eine höhere lokale Varianz, jedoch ohne sichtbare Struktur, wie beispielsweise eine Bayer-Matrix. Eine simple Methode um Blue-Noise zu erzeugen, ist einen High-Pass-Filter auf White-Noise anzuwenden[MG16]. Zudem wird, wie in [Jp16], Rauschen auch auf den Radius gerechnet, wodurch die Samples gleichmäßiger verteilt werden.

Das Rauschen wird pro Frame variiert. Durch den temporalen Filter werden so effektiv mehrere Samples über die Zeit verteilt. Hier werden acht Samples pro Frame benutzt, kombiniert mit acht Variationen des Rauschens.

Wie in [Xu16] beschrieben, wird das aktuelle Bild mit dem vorherigen linear interpoliert, wobei das neue Bild nur gering in das alte geblendet wird, beispielsweise mit 5%.

Ein Problem, was dabei auftritt, ist Ghosting. Dies tritt auf, wenn sich bewegende Objekte falsch reprojiziert werden und das alte Bild das Neue an der falschen Position überlagert. Um dies zu verhindern, wird bei der Anwendung auf Farbbildern jeder reprojizierte Pixel auf den Farbbereich seiner Nachbarn begrenzt, siehe [Kar14]. Das ist jedoch nicht effektiv bei SSAO, da die Bilder stark verrauscht sind und die lokale Varianz dadurch so groß ist, dass der Farbbereich zu groß wird, um Ghosting zu verhindern. Stattdessen wird der Alpha Wert, mit dem die Bilder geblendet werden, dynamisch angepasst, ähnlich wie in [Jp16] und [Mé10]. In [Mé10] wird die Differenz der Werte als Kriterium benutzt, was ähnlich wie das Beschränken auf den Wertebereich der Nachbarschaft aufgrund der Varianz nicht funktioniert. Stattdessen werden, wie in [Jp16], Tiefenwerte verglichen, sowie die Bewegungsgeschwindigkeit der Objekte. Die Geschwindigkeit wird aus dem Velocity-Vektor berechnet. Beide Werte werden multipliziert. Durch die Differenz der Tiefenwerte wird verhindert, dass Informationen aus Vorder- oder Hintergrund genutzt werden. Durch die Kombination mit der Bewegungsgeschwindigkeit wird dafür gesorgt, dass bei stillen Bildern immer korrekt interpoliert wird.

```
float depthDifference = abs(linearDepth(depth) -
↪  linearDepth(lastFrameDepth));
float velocityFactor  = dot(abs(velocity), vec2(1.f)); //squared
↪  distance

float alpha = 0.05f;
float blendStrength = 100; //higher value results in less
↪  ghosting, but less filtering in motion
float blendFactor = clamp(blendStrength * depthDifference *
↪  velocityFactor, 0.f, 1.f);
alpha = mix(alpha, 1.f, blendFactor);

//check if reprojection is within image bounds
const float lowerBound = 0.0001f;
const float upperBound = 0.9999f;
if(uvOffset.x < lowerBound || uvOffset.y < lowerBound ||
↪  uvOffset.x >= upperBound || uvOffset.y >= upperBound) alpha =
↪  1.f;

o_ao = (1.f - alpha) * lastFrameAO + alpha * currentAO;
```

Listing 11: GLSL-Ausschnitt, Fragment-Shader für die temporale Filterung von SSAO

Abbildung 13: finaler SSAO-Buffer

5.4 SSR

SSR wird, wie auch SSAO, auf der halben Auflösung berechnet. Dasselbe temporale Filter-Setup findet Verwendung, allerdings ist der genutzte Gauß-Filter nicht adaptiv. Im ersten Schritt wird durch Raymarching auf dem Tiefen-Buffer der Schnitt-

punkt gesucht. Ähnlich wie bei SSAO, tritt bei niedriger Sample-Anzahl Banding auf. Um dies zu verhindern, wird der Startpunkt des Raymarchings durch Blue-Noise verschoben. Ist ein Schnittpunkt gefunden, wird der Vektor von dem Startpixel zum Pixel des Schnittpunkts in einen 16-Bit-Float-Buffer geschrieben. Zudem wird in einem Unsigned-Byte-Buffer gespeichert ob ein Schnittpunkt gefunden wurde, bezeichnet als Specular-Occlusion. Der Wert ist eins, falls ein Schnittpunkt vorliegt und sonst null. Diese Information wird später genutzt um zu überprüfen, ob der Pixel über Reflektions-Informationen verfügt. An den Rändern wird dieser Faktor abgeschwächt, um die Reflektionen an den Rändern des Bildes weich auszublenden, bevor sie den Bildbereich verlassen.

Darauf folgt das Auslesen der Reflektion. Die gespeicherte Schnittpunktposition wird mit Hilfe des Velocity-Buffers in den vorherigen Frame reprojiziert. Die finale Farbe wird an dieser Stelle aus dem vorherigen Color-Buffer ausgelesen. Dieser wurde vor dem Post-Processing-Schritt abgespeichert.

Der letzte Schritt ist die Beleuchtung der Pixel. Die Cook-Torrance-Split-Sum-Approximation dient als BRDF, wie auch bei dem LPV. Das Ergebnis wird mit der Specular-Occlusion multipliziert und additiv geblendet.

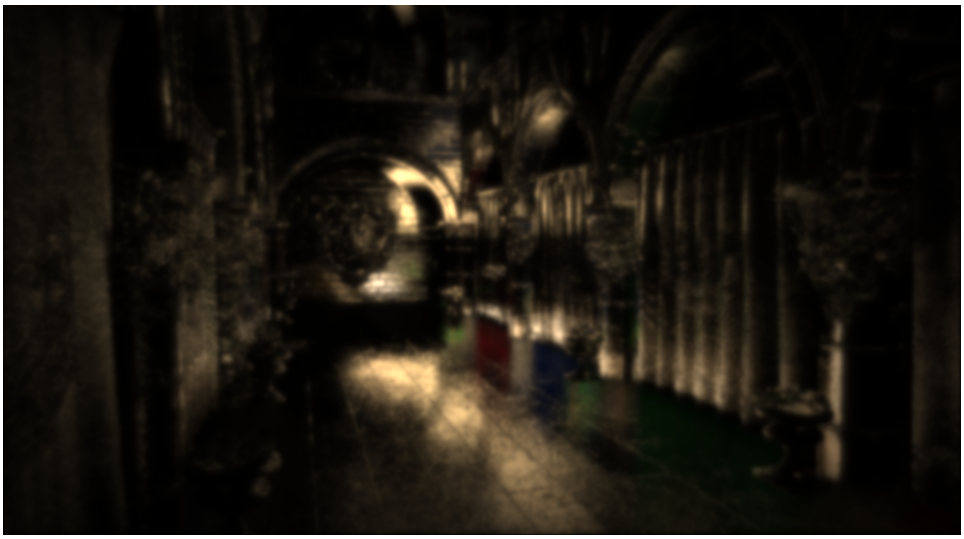


Abbildung 14: finaler SSR-Buffer

6 Evaluation

6.1 Theoretische Evaluation

Für die Analyse der abgedeckten Effekte wird die Notation aus Abschnitt (2) genutzt.

Das LPV modelliert in der grundlegenden Version zwei diffuse Reflektionen $LDDE$. Mit der Erweiterung um den spekularen Anteil kann die letzte und nur

diese, eine spekulare Reflektion sein, $LD(D|S)E$. Die Sichtbarkeit wird nach der ersten diffusen Reflektion ignoriert. Die Erweiterung um das Geometrie-Volumen approximiert die Sichtbarkeit dieser. Durch die Approximation der diffusen Reflektion im Ausbreitungsschritt können N diffuse Reflektionen vor der finalen Reflektion angenähert werden $L(D)\{1, N\}(D|S)E$. Dabei ist N die Anzahl der für den Ausbreitungsschritt verwendeten Iterationen.

Eine Limitierung des LPVs ist, dass die beachteten Lichtquellen L eingeschränkt sind. Da für jede Lichtquelle eine RSM berechnet werden muss, was recht rechenaufwändig ist, wird in der Implementierung nur die Sonne beachtet.

SSAO modelliert die Sichtbarkeit der letzten diffusen Reflektion. Bent-Normals und S_3HO modellieren diese akkurater und können zusätzlich benutzt werden, um die Sichtbarkeit für die finale spekulare Reflektion zu modellieren. Der Near-Field-Licht-Transport modelliert eine zusätzliche diffuse Reflektion mit begrenzter Reichweite. Der vorherige Lichtpfad ist von den sonstigen Berechnungen abhängig, da das vorherige Bild als Input genutzt wird. Sei $\backslash scene$ ein regulärer Ausdruck für die Lichtpfade, welche für das Input-Bild benutzt wurden, dann ist der modellierte Effekt $(\backslash scene)DE$.

SSR ist ebenso abhängig vom Input aus dem vorherigen Bild. Jedoch wird hier eine spekulare Reflektion berechnet $(\backslash scene)SE$. Hierbei wird zusätzlich die Sichtbarkeit beachtet.

6.2 Optische Qualität

Zur Evaluation der optischen Qualität wird eine Ground-Truth benötigt. Für diesen Zweck wird die Open-Source 3D-Software Blender benutzt. Es wurden für beide Renderer die gleichen Modelle und Texturen benutzt und Kamera- und Licht-Parameter wurden abgestimmt. Trotzdem unterscheiden sie sich in den Details der genutzten BRDF und des Tonemappers.

In Blender wurde der Cycles Renderer genutzt, ein Path-Tracer. Das Bild wurde mit 1024 Samples pro Pixel berechnet. Auf Denoising und Post-Processing wurde verzichtet.

(a) Render des Path-Tracer.

(b) Echtzeitrender mithilfe der LPV

Abbildung 15: Vergleich der Renderer

6.2.1 Optische Qualität: LPV

Ein auffälliger Unterschied ist, dass das LPV beim Rendering nicht das Umgebungslicht vom Himmel miteinbezieht. In [KD10] wird vorgeschlagen, entferntes Licht aus Environment-Maps an den Rändern des Volumen zu injizieren. Dies hat jedoch das Problem, dass dieses Licht nach der begrenzten Anzahl an Ausbreitungs-Schritten nicht im Zentrum des Volumens ankommt. Daher ist dieser Ansatz nicht geeignet.

Das Bild des Path-Tracers hat eine deutlich stärkere sichtbare Richtung in der indirekten Beleuchtung. Dies ist gut sichtbar an den unteren Bögen, auf der rechten Seite und den oberen Säulen auf der linken. Dies ist zurückzuführen auf die grobe Unterteilung des Raumes und die geringe Präzision der genutzten Spherical-Harmonics.

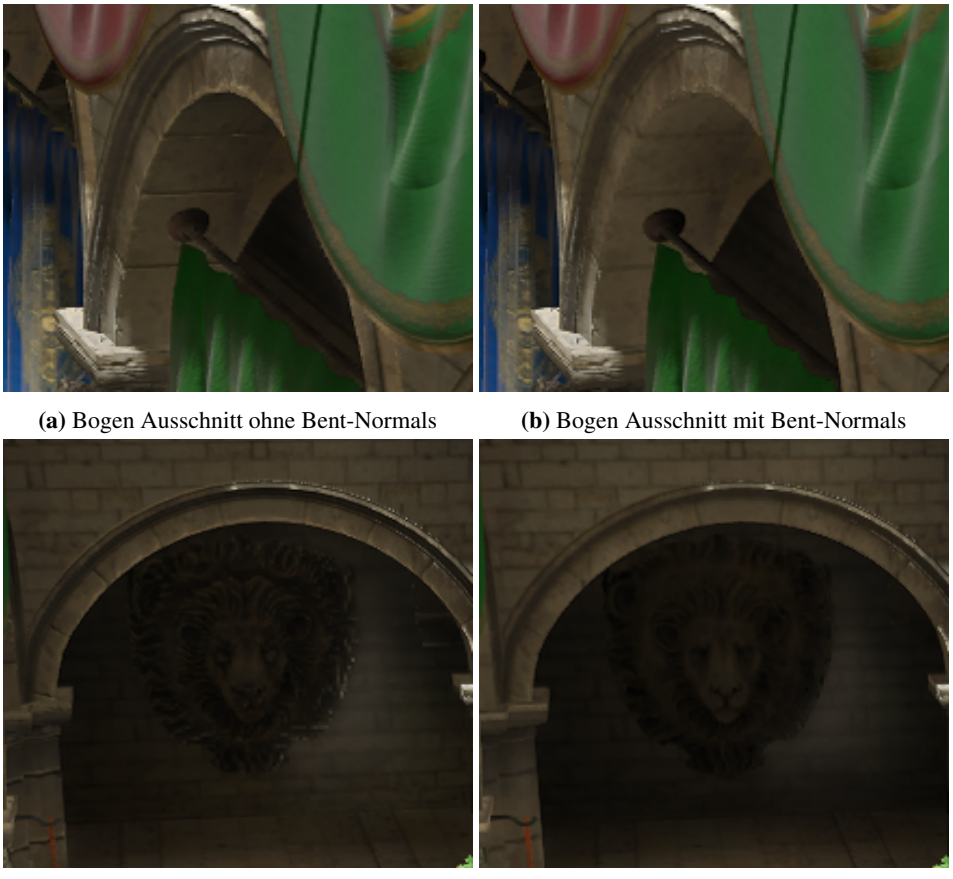
Im Bereich des Löwenkopfes ist das Echtzeit-Bild deutlich dunkler, vermutlich zurückzuführen auf das Fehlen von mehreren indirekten Reflektionen. Im Bereich

der rechten oberen Säulen ist zu sehen, dass das offline gerenderte Bild deutlich stärkeres Color-Bleeding aufweist.

6.2.2 Optische Qualität: SSAO

An dem Bereich hinter den hängenden Flaggen ist zu erkennen, dass SSAO die Stellen nicht so stark abdunkelt wie der Path-Tracer. Zudem ist im Bereich des Bogens, über dem Löwenkopf, die Richtungsabhängigkeit der Schlagschatten erkennbar: auf der linken Seite ist er klar erkennbar, wird aber auf der rechten Seite immer schwächer. Normales SSAO simuliert diesen Effekt nicht, da es Lichtunabhängig operiert und auch die Approximation des Phänomens durch Bent-Normals oder S_3HO ist nicht detailliert genug, um einen solchen Effekt zu erzeugen.

6.2.3 Optische Qualität: Bent-Normals



(a) Bogen Ausschnitt ohne Bent-Normals **(b)** Bogen Ausschnitt mit Bent-Normals

(c) Löwenkopf Ausschnitt ohne Bent-Normals **(d)** Löwenkopf Ausschnitt mit Bent-Normals

Abbildung 16: Vergleich mit und ohne Bent-Normals

Die Bent-Normals hellen einige Bereiche, wie die Unterseite der Bögen, auf. An einigen Stellen werden Reflektionen abgeschwächt, sichtbar an der Seite des Löwenkopfes. Zudem wirkt die Beleuchtung an einigen Stellen weicher, beispielsweise über dem mittleren Bogen. Die Unterschiede sind subtil. Im Vergleich mit der Referenz ist nicht eindeutig feststellbar, ob der Effekt diese besser annähert. Ein Nachteil ist, dass Details in der Beleuchtung verloren gehen. Dies ist eine Folge der Berechnung auf halber Auflösung und Filterung der Normalen.

6.2.4 Optische Qualität: S_3HO

(a) Szene mit SSAO

(b) Szene mit S_3HO

Abbildung 17: Vergleich mit und ohne S_3HO

S_3HO dunkelt einige Bereiche stärker ab als SSAO, sichtbar an der linken Wand. Andere Bereiche, wie die Unterseite der Bögen ist heller. Die starken Reflektionen werden deutlich abgedunkelt. Dies ist deutlich näher an der Referenz. Insbesondere der Löwenkopf ist mit S_3HO deutlich sichtbarer und gleichmäßiger ausgeleuchtet, eine deutliche Annäherung an die Referenz, verglichen mit SSAO. Im Gegensatz zu Bent-Normals wird die Normale nicht modifiziert, daher kommt es nicht zu einem Detailverlust in der Beleuchtung.

6.2.5 Optische Qualität: Near-Field-Licht-Transport

(a) Ausschnitt ohne Licht-Transport **(b)** Ausschnitt mit Licht-Transport

Abbildung 18: Vergleich mit und ohne Near-Field-Licht-Transport

Die Approximation der ersten diffusen Reflektion sorgt dafür, dass helle Bereiche die Umgebung erhellen. Dadurch werden einige dunkle Bereiche aufgehellt. Der Effekt ist deutlich feiner aufgelöst als der, der durch das LPV erzielt wird. Hierbei ist zu beachten, dass der Effekt von Hand angepasst werden muss. In [TR] wird erwähnt, dass dafür der Radius benutzt wird, wodurch das Ergebnis effektiv mit einem einstellbaren Wert multipliziert wird. Für das Vergleichsbild ist ein Wert von 3 genutzt. Mit dem Standwert von 1 ist der Effekt kaum sichtbar.

Abbildung 19: Path-Traced Referenz

Im Vergleich zur Referenz, sorgt der Effekt für eine bessere Approximation. Ohne ihn sind einige Bereiche wie die Unterseite der Vase, der Bereich hinter ihr und der untere, goldene Streifen des Vorhangs deutlich dunkler als in der Referenz.

6.2.6 Optische Qualität: SSR

Im Vergleich zu der Vorlage reflektieren einige Stellen zu stark, gut sichtbar auf der linken Wand und neben dem Löwenkopf. Dies hat mehrere Ursachen. Da beim Raymarching, aus Performance-Gründen, wenig Samples getestet werden, muss die Schrittlänge entsprechend groß sein. Dies führt dazu, dass kleinere Objekte, welche die Reflektion verdecken würden, nicht gefunden werden. Zudem enthält

das vorherige Bild, aus dem die Reflektion gelesen wird, den spekularen Anteil, der in die Kamera fällt. Es wäre korrekter aber auch rechenintensiver, die diffusen und spekularen Anteile des Bildes zu trennen und nur den diffusen Anteil für die Reflektion zu nutzen.

6.3 Artefakte

Das Haupt-Artefakt was bei der Benutzung von LPVs sichtbar ist, ist Light-Leaking. Light-Leaking ist die Bezeichnung für Licht, welches an Stellen auftritt, an denen es verdeckt sein sollte, beispielsweise wenn es durch dünne Geometrie fällt. Dies ist ein Problem, welches viele Algorithmen betrifft, beispielsweise wird in [Hoo16] Light-Leaking bei vorberechneter Beleuchtung beschrieben. Bei LPVs tritt das Problem auf, da die Verdeckung nur approximiert wird und Licht daher nicht komplett blockiert. Zudem sorgt die Gitterstruktur dafür, dass Licht aus verdeckten Bereichen stellenweise zur Interpolation genutzt wird.

Abbildung 20: Light-Leaking auf der Innenseite der Sponza-Szene, aufgehellt zur besseren Sichtbarkeit

Da SSAO und SSR in Screen-Space berechnet werden, leiden sie unter den für diese Techniken üblichen Problemen. Da keine Informationen außerhalb des Bildes verfügbar sind, wird der Effekt an den Rändern des Bildes abgeschnitten. Das gleiche gilt für verdeckte Geometrie. Im Falle von SSAO führt das zu hellen Umrandungen hinter Objekten. Bei SSR können Reflektionen durch Objekte im Vordergrund verdeckt werden.

Im SSAO-Buffer sind nach der Filterung immer noch Strukturen des grundlegenden Rauschens zu erkennen. Durch das variable Rauschen sind weder SSR noch SSAO komplett stabil, auch in unbewegten Bildern ist noch ein leichtes Flackern erkennbar.

Bei SSAO wird der temporale Filter unter Kamerabewegung abgeschwächt, um Ghosting zu verhindern, wodurch das Rauschen sichtbar wird.

Da die Bent-Normals auf halber Auflösung berechnet und gefiltert werden, ver-

liert die Beleuchtung des LPVs aufgrund der gemittelten Werte an Details, verglichen mit den Normalen aus dem G-Buffer.

Der Near-Field-Licht-Transport wird auf halber Auflösung berechnet und hochskaliert. Dadurch kommt es an einigen Kanten zu Artefakten. Beleuchtung aus dem Vordergrund fällt dort in den Hintergrund und umgekehrt. Dies könnte durch Depth-Aware-Upscaling, wie in [SSO12] oder eine ähnliche Technik, behoben werden.

6.4 Performance-Messungen

Die Messungen wurden auf der Sponza-Szene durchgeführt. Diese besteht aus 192.460 Vertices und 262.249 Faces. Als Hardware wurde ein Computer mit einem Ryzen 5 1600 als CPU und einer GTX 1050 als GPU verwendet. Die Messungen wurden mithilfe der Profiling-Tools von NVIDIA Nsight Graphics vorgenommen. Die Szene wurde aus der für die Evaluation genutzten Perspektive gerendert.

Die Auflösung beträgt 1920 x 1080 Pixel, jedoch werden einige Effekte nicht auf voller Auflösung berechnet. Die Renderzeit der mit Cycles erstellten Referenz beträgt 26:16 Minuten.

Pass	Zeit in ms
RSM-Rendering	0.30
Light-Injection	0.05
Geometry-Injection: G-Buffer	0.50
Geometry-Injection: RSM	0.05
Einzelne Propagation-Iteration	0.23
Einzelne Propagation-Iteration: Reflektion/Verdeckung	0.26
Kombinierung der Iterationen	0.07
Temporaler-Filter	0.03
Rendering mit Spekular	1.78
Rendering ohne Specular	0.78
Total: 8 Iterationen, Spekular + Reflektion/Verdeckung	4.83
Total: 8 Iterationen, ohne extra Features	3.07

Von den Renderingschritten ist nur das Rendering der RSM szenenabhängig. Der finale Rendering-Pass ist Auflösungsabhängig. Durch die optionalen Features sowie die wählbare Iterations-Anzahl kann die Technik je nach Anforderungen an die visuelle Qualität und Performance skaliert werden.

Pass	Zeit in ms
SSAO, 8 Samples	0.86
SSAO mit Bent-Normal, 8 Samples	1.03
S_3HO, 8 Samples	1.08
Separated-Adaptive-Gaussian-Blur	0.31
Blur-Total	1.24
Temporal-Filter	0.23
Licht-Transport-Blending	0.16
Total	2.66

Die Geschwindigkeit der Post-Processing Effekte ist weitestgehend szenenunabhängig. Stattdessen skaliert der Rechenaufwand mit der Auflösung. Auffällig ist, dass das Filtern der Ergebnisse mehr Zeit in Anspruch nimmt als die initiale Berechnung.

Die Berechnung der Bent-Normals benötigt etwas mehr Zeit als reines SSAO, S_3HO ist noch ein Stück langsamer. Allerdings sind die Unterschiede recht gering. Der Unterschied zwischen der Basisversion und S_3HO beträgt lediglich 0.22ms.

Pass	Zeit in ms
SSR-Raymarching	0.34
SSR-Retrieve-Reflection	0.19
SSR-Separated-Gaussian-Blur	0.09
Blur-Total	0.36
Temporal-Filter	0.15
SSR-Rendering	0.47
Total	1.51

SSR ist der schnellste Effekt. Die Filterung ist schneller als bei SSAO, da es sich um einen einfachen Gauß-Filter handelt. Der dynamische Variante hat für jedes Sample zwei zusätzliche Textur-Zugriffe für Normale und Tiefe, sowie zusätzliche Branches. Durch die Kombination von Schnittpunktberechnung und dem Auslesen der Reflektion in einen Pass, könnte die Geschwindigkeit durch das Sparen von Bandbreite verbessert werden.

7 Fazit

Das LPV bietet eine komplett dynamische und schnelle Lösung für indirekte Beleuchtung. Das geht jedoch auf Kosten der visuellen Qualität und der Anzahl der möglichen Lichtquellen. Falls diese Einschränkungen für den Anwendungsfall akzeptabel sind, stellt es eine angemessene Lösung dar. Allerdings ist es keine allgemeine Lösung für das Problem der indirekten Beleuchtung.

Voxel-Cone-Tracing [CNS+11] ist eine Lösung, welche ebenfalls dynamische Lichter und Geometrie erlaubt und eine höhere visuelle Qualität erzielt. Dafür ist die Technik rechenintensiver.

In [Ste16] werden Precomputed-Radiance-Transfer-Probes benutzt um eine hohe visuelle Qualität mit dynamischen Lichtquellen und schneller Laufzeit zu erzielen. Dafür kann nur statische Geometrie berücksichtigt werden.

Eine weitere Technik wird in [Yud19] vorgestellt. Dort wird Raytracing auf einer dynamischen Voxel-Repräsentation der Szene genutzt, um hochqualitative Ergebnisse zu erzielen. Die Technik ist schnell und skalierbar. Dynamische Lichtquellen und Geometrie sind möglich, jedoch dürfen diese sich nur langsam verändern.

Letztendlich gibt es eine Vielzahl von Techniken mit verschiedenen Stärken und Schwächen. Für die Wahl muss man die Limitierungen des konkreten Anwendungsfalles betrachten. Eine Lösung, welche alle Anforderungen erfüllt, ist noch nicht vorhanden.

SSAO und SSR sind weniger ganzheitliche Ansätze, sondern komplementäre Techniken. Sie sind gut geeignet andere Methoden zu ergänzen und können stellenweise deutlich akkuratere Ergebnisse erzielen. Dafür entstehen durch die begrenzten Daten sichtbare Artefakte und sie können sehr aufwändig in der Berechnung sein.

8 Ausblick

Für alle in dieser Arbeit vorgestellten und implementierten Techniken existieren eine Vielzahl von Verbesserungen, um sowohl die visuelle Qualität, sowie auch die Geschwindigkeit zu verbessern.

LPV In [KD10] wird vorgeschlagen, mehrere LPV-Kaskaden zu verwenden, ähnlich wie Cascaded-Shadow-Mapping. Dadurch kann ein größeres Gebiet abgedeckt und die Qualität in der Nähe der Kamera verbessert werden.

In [Kap10] wird die Möglichkeit behandelt, das LPV nicht jeden Frame zu aktualisieren. Die Berechnung verschiedener Kaskaden oder Schritte kann auf mehrere Bilder verteilt werden. Dies spart Rechenaufwand.

In [Kir18] wird von Hand platzierte Blocker-Geometrie benutzt anstatt die Geometrie dynamisch zu injizieren. Zudem wird die Geometrie im Rendering Schritt bei der trilinearen Interpolation miteinbezogen um Light-Leaking zu verhindern.

In [Kap09] wird vorgeschlagen, Point-Lights nach dem Ausbreitungsschritt zu injizieren, um effizient eine hohe Anzahl dieser zu rendern. Zur Verbesserung des Rendering-Schrittes kann die Ableitung des Volumens benutzt werden, um die Intensität in Bereichen mit hohem Kontrast abzuschwächen [KD10]. Dadurch kann Light-Leaking reduziert werden.

Eine mögliche Beschleunigung der Injektion von Geometrie und Licht könnte durch das Nutzen der OpenGL Image Load/Store Operationen erzielt werden. Diese würden die Vertex-Point-Methode ersetzen. Dies hätte den Vorteil, dass der Overhead der Geometrie-Pipeline vermieden wäre, dafür müsste man sich selbst um die Synchronisation der Ergebnisse kümmern.

In [CL08] wird eine Methode vorgestellt, um die Cook-Torrance-BRDF in Spherical-Harmonics zu projizieren. Dies könnte beim Rendering des spekularen Anteils genutzt werden, anstatt der Approximation durch mehrmaliges Integrieren in Richtung der Reflektion mit einem Cosine-Lobe und der anschließenden Verrechnung mit der Split-Sum-Approximation.

Eine Alternative dazu wird in [ODo18] vorgestellt. Hier wird aus den SH-Koeffizienten eine virtuelle Lichtquelle geschätzt, aus welcher der spekulare Anteil berechnet wird.

Screen-Space Methoden Eine Methode um die Geschwindigkeit von Screen-Space-Algorithmen zu verbessern, wird in [LB13] vorgestellt. Dort wird beobachtet, dass durch die benutzten Noise-Patterns die Cache-Kohärenz der Textur-Zugriffe leidet, was die Performance verschlechtert. Als Lösung wird vorgeschlagen, das Bild so neu anzuordnen, dass alle Pixel mit dem gleichen Noise-Wert beieinander liegen. Nach der Berechnung werden die Pixel wieder an ihre ursprüngliche Position geschrieben.

Um den Beschränkungen von Screen-Space-Methoden entgegenzuwirken, werden in [MMNL16] Deep-G-Buffer eingeführt. Diese enthalten nicht nur die Informationen, welche am nächsten an der Kamera sind, sondern beinhalten mehrere Layer. Dadurch wird Artefakten, welche durch Verdeckung entstehen, entgegengewirkt.

SSAO Ein ähnlicher Ansatz wird in [AP16] genutzt. Dort wird Ambient-Occlusion in zwei separaten Passes gerendert. Einmal für statische Geometrie und einmal für alle dynamischen Objekte. Dadurch werden Artefakte im Hintergrund um die Objekte vermieden.

Für SSAO gibt es neben der hier vorgestellten Herangehensweise alternative Algorithmen. Beispiele sind Horizon-Based-Ambient-Occlusion[BSD08] und Line-Sweep-Ambient-Obscurance[Tim13]. Ersteres liefert bessere Ergebnisse, während zweites Effizienz-Vorteile aufweist.

In der hier vorgestellten Implementierung wurden die Verdeckungs-Koeffizienten von S_3HO mittels des gleichen temporalen Filters gefiltert, der auch für SSAO benutzt wurde. Da es sich jedoch um SH-Koeffizienten handelt, ist dies nicht mathematisch korrekt. In [HSS11] wird vorgeschlagen einen in Zonal-Harmonics projizierten Gauß-Kern zu nutzen.

SSR In [Sta15] wird eine Erweiterung zur naiven Herangehensweise zu SSR vorgestellt. Dort werden mehr Strahlen pro Pixel benutzt, um genauere Ergebnisse zu erzielen. Dadurch können Effekte wie Contact-Hardening, Abhängigkeiten von der Oberfläche, wie Roughness und Specular-Elongation dargestellt werden. Zur Gewährleistung der Geschwindigkeit wird Importance-Sampling, eine variable, bildabhängige Anzahl von Strahlen pro Bild-Bereich, sowie Strahlen-Teilung unter Nachbarn, benutzt.

In [MM14] wird das Raymarching auf dem Tiefenbuffer verbessert, indem ein modifizierter Digital-Differential-Analyzer benutzt wird. Dieser erlaubt Pixelgroße Schritte, wodurch keine Informationen übersprungen oder einzelne Pixel mehrmals ausgelesen werden.

Allgemein Neben der Rekursionstiefe der berechneten Reflektionen, stellt insbesondere der Specular-Light-Transport eine Herausforderung dar, da den Algorithmen häufig die nötige Präzision fehlt. Viele Techniken sind begrenzt in der Anzahl und bezüglich des Typs der beachteten Lichtquellen. Weitere Effekte, welche in Echtzeit-Algorithmen kaum umgesetzt werden, sind der Einfluss von *participating media* wie Gase, Flüssigkeiten oder Nebel sowie der Einfluss von komplexen Materialien wie durchscheinende oder transparente Objekte.

Von besonderem Interesse ist GPU-Raytracing. Seit Mitte 2018 ist Consumer Hardware mit Hardwarebeschleunigung für Raytracing erhältlich. Erste Ergebnisse indizieren, dass die Technik noch nicht schnell genug für einen holistischen Ansatz ist. Sie wird bisher zur Ergänzung zu traditioneller Rasterisierung genutzt. Mit Verbesserungen in Hardware und Algorithmen könnten solche Ansätze die Brücke zur Qualität von offline Rendering schlagen.

Literatur

[AMHH18] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-Time Rendering, Fourth Edition*. A. K. Peters, Ltd., Natick, MA, USA, 4th edition, 2018.

[AP16] Andrei Tatarinov Alexey Panteleev. Advanced ambient occlusion methods for modern games. *Game Developers Conference*, 2016.

[BB17] Colin Barré-Brisebois. A certain slant of light: Past, present and future challenges of global illumination in games. In *SIGGRAPH Open Problems in Real-Time Rendering course*. ACM SIGGRAPH, 2017.

[BSD08] Louis Bavoil, Miguel Sainz, and Rouslan Dimitrov. Image-space horizon-based ambient occlusion. 07 2008.

[Cha13] John Chapman. Ssao tutorial. `http://john-chapman-graphics.blogspot.com/2013/01/ssao-tutorial.html`, 2013. Online; accessed: 2019-08-15.

[CL08] Hao Chen and Xinguo Liu. Lighting and material of halo 3. In *ACM SIGGRAPH 2008 Games*, SIGGRAPH '08, pages 1–22, New York, NY, USA, 2008. ACM.

[CNS+11] Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, and Elmar Eisemann. Interactive Indirect Illumination Using Voxel Cone Tracing. *Computer Graphics Forum*, 2011.

[CT82] R. L. Cook and K. E. Torrance. A reflectance model for computer graphics. *ACM Trans. Graph.*, 1(1):7–24, January 1982.

[Dim07] Rouslan Dimitrov. Cascaded shadow maps. Technical report, NVIDIA Corporation, 2007.

[DS05] Carsten Dachsbacher and Marc Stamminger. Reflective shadow maps. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, I3D '05, pages 203–213, New York, NY, USA, 2005. ACM.

[Eng11] Wolfgang Engel. *GPU Pro 2*. A. K. Peters, Ltd., Natick, MA, USA, 1st edition, 2011.

[FM08] Dominic Filion and Rob McNaughton. Effects & techniques. In *ACM SIGGRAPH 2008 Games*, SIGGRAPH '08, pages 133–164, New York, NY, USA, 2008. ACM.

[Gia16] Michele Giacalone. Screen space reflections in the surge. `https://de.slideshare.net/MicheleGiacalone1/`

screen-space-reflections-in-the-surge, 2016. Online; accessed: 2019-06-25.

[Gre03] Robin Green. Spherical harmonic lighting: The gritty details. *Archives of the Game Developers Conference*, 03 2003.

[Hab10] John Hable. Uncharted 2: Hdr lighting. *Game Developers Conference*, 03 2010.

[Hec90] Paul S. Heckbert. Adaptive radiosity textures for bidirectional ray tracing. *SIGGRAPH Comput. Graph.*, 24(4):145–154, September 1990.

[Hoo16] JT Hooker. Volumetric global illumination at treyarch. In *SIGGRAPH Advances in Real-Time Rendering in Games course*. ACM SIGGRAPH, 2016.

[HSS11] Sebastian Herholz, Timo Schairer, and Wolfgang Straßer. Screen space spherical harmonics occlusion (s3ho) sampling. 08 2011.

[Jp16] Jp. Temporal reprojection and sao. http://bitsquid.blogspot.com/2015/09/temporal-reprojection-and-sao.html, 2016. Online; accessed: 2019-06-25.

[Kap09] Anton Kaplanyan. Light propagation volumes in cryengine 3. In *SIGGRAPH Advances in Real-Time Rendering in Games course*. ACM SIGGRAPH, 01 2009.

[Kap10] Anton Kaplanyan. Real-time diffuse global illuminationin cryengine 3. In *SIGGRAPH*. ACM SIGGRAPH, 2010.

[Kar13] Brian Karis. Real shading in unreal engine 4 by. 2013.

[Kar14] Brian Karis. High quality temporal supersampling. In *SIGGRAPH Advances in Real-Time Rendering in Games course*. ACM SIGGRAPH, 2014.

[KD10] Anton Kaplanyan and Carsten Dachsbacher. Cascaded light propagation volumes for real-time indirect illumination. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '10, pages 99–107, New York, NY, USA, 2010. ACM.

[Kir10a] Andreas Kirsch. Light propagation volumes. http://blog.blackhc.net/2010/07/light-propagation-volumes/, 2010. Online; accessed: 2019-06-25.

[Kir10b] Andreas Kirsch. Light propagation volumes - corrections. http://data.blog.blackhc.net/2010/07/lpv-corrections.pdf, 2010. Online; accessed: 2019-09-23.

[Kir18] Scott Kircher. Rendering technology in 'agents of mayhem'. *Game Developers Conference*, 2018.

[LB13] Jon Jansen Louis Bavoil. Particle shadows and cache-efficient post-processing. *Game Developers Conference*, 2013.

[Mé10] José María Méndez. A simple and practical approach to ssao. https://www.gamedev.net/articles/programming/graphics/a-simple-and-practical-approach-to-ssao-r2753, 2010. Online; accessed: 2019-06-25.

[May18] Benoît Mayaux. Horizon-based indirect lighting (hbil). Technical report, 2018.

[MG16] Mikkel Svendsen Mikkel Gjoel. Low complexity, high fidelity - inside rendering. *Game Developers Conference*, 2016.

[MM14] Morgan McGuire and Michael Mara. Efficient GPU screen-space ray tracing. *Journal of Computer Graphics Techniques (JCGT)*, 3(4):73–85, December 2014.

[MMNL16] M. Mara, M. McGuire, D. Nowrouzezahrai, and D. Luebke. Deep g-buffers for stable global illumination approximation. In *Proceedings of High Performance Graphics*, HPG '16, pages 87–98, Goslar Germany, Germany, 2016. Eurographics Association.

[O'D11] Yuriy O'Donnell. Deferred screen space directional occlusion. https://kayru.org/articles/dssdo/, 2011. Online; accessed: 2019-07-15.

[ODo18] Yuriy ODonnell. Precomputedglobal illuminationin frostbite. *Game Developers Conference*, 2018.

[Ped16] Lasse Jon Fuglsang Pedersen. Temporal reprojectionanti-aliasing in inside. *Game Developers Conference*, 2016.

[Pol16] Eric Polman. Light propagation volumes. https://ericpolman.com/2016/06/28/light-propagation-volumes/, 2016. Online; accessed: 2019-06-25.

[Slo08] Peter-Pike Sloan. Stupid spherical harmonics (sh) tricks. *Game Developers Conference*, 01 2008.

[SSO12] Sebastian Schwarz, Mårten Sjöström, and Roger Olsson. Depth map upscaling through edge weighted optimization. 02 2012.

[Sta15] Tomasz Stachowiak. Stochastic screen-space reflections. In *SIGGRAPH Advances in Real-Time Rendering in Games course*. ACM SIGGRAPH, 2015.

[Ste16] Nikolay Stefanov. Global illumination in tom clancy's the division. *Game Developers Conference*, 2016.

[Tim13] Ville Timonen. Line-Sweep Ambient Obscurance. *Computer Graphics Forum (Proceedings of EGSR 2013)*, 32(4):97–105, 2013.

[TR] Hans-Peter Seidel Tobias Ritschel, Thorsten Grosch. Approximating dynamic global illumination in image space. TU Clausthal, Institut für Informatik.

[Vid17] Davyd Vidiger. Ssrtgi: Toughest challenge in real-time 3d. `https://80.lv/articles/ssrtgi-toughest-challenge-in-real-time-3d/`, 2017. Online; accessed: 2019-06-25.

[Wes14] Peter Wester. Generating smooth and cheap ssao using temporal blur. `https://www.gamasutra.com/blogs/PeterWester/20140116/208742/Generating_smooth_and_cheap_SSAO_using_Temporal_Blur.php`, 2014. Online; accessed: 2019-06-25.

[Xu16] Ke Xu. Temporal antialiasing in uncharted 4. In *SIGGRAPH Advances in Real-Time Rendering in Games course*. ACM SIGGRAPH, 2016.

[Yud19] Anton Yudintsev. Scalable real-time global illumination for large scenes. *Game Developers Conference*, 2019.