HAW HAMBURG

Bachelorarbeit

Sebastian Brückner

Globale Beleuchtung in 2D mit Hilfe von Echtzeit-Raytracing

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Sebastian Brückner

Globale Beleuchtung in 2D mit Hilfe von Echtzeit-Raytracing

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Philipp Jenke
Zweitgutachter: Prof. Dr.-Ing. Andreas Meisel

Eingereicht am: 24.07.2019

Sebastian Brückner

Thema der Arbeit

Globale Beleuchtung in 2D mit Hilfe von Echtzeit-Raytracing

Stichworte

2D, Raytracing, Global Illumination

Kurzzusammenfassung

Für die 2D Computergrafik existieren kaum Beleuchtungsmodelle. In der 3D Computergrafik rückt währenddessen die Berechnung von globaler Beleuchtung mit Raytracing in Echtzeit in greifbare Nähe. Hier wird das Problem der mangelnden Beleuchtungsmodelle der 2D Grafik mit Raytracing kombiniert um ein Echtzeit System zum Berechnen der globalen Beleuchtung in 2D zu erstellen.

Sebastian Brückner

Title of Thesis

Globale Illumination in 2D using Realtime-Raytracing

Keywords

2D, Raytracing, Global Illumination

Abstract

There is hardly any lighting model for 2D computer graphics. Meanwhile, in the 3D computer graphics, the calculation of global illumination with raytracing in real time is within reach. Here the problem of the lack of lighting models for 2D graphics is combined with raytracing to create a real-time system for calculating global illumination in 2D.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungen

AABB Axis Aligned Bounding Box.

bpp Farbtiefe.

BRDF Bidirectional Reflectance Distribution Function.

BSDF Bidirectional Scattering Distribution Function.

BSSRDF Bidirectional Scattering Surface Reflectance Distribution Function.

BTDF Bidirectional Transmittance Distribution Function.

BVH Hüllkörperhierarchie (engl. Bounding Volume Hierachy).

FPS Bilder pro Sekunde.

k-DOP k-Discretely Oriented Polytopes.

LPV Light Propagation Volumes.

MCI Crude Monte-Carlo-Integration.

OBB Oriented Bounding Boxes.

RGB Multisample anti-aliasing.

RGB Rot Grün Blau.

RR Russian Roulett.

VXGI Voxel Cone Tracing global Illumination.

1 Einleitung

Während die 3D Computergrafik ein sehr aktives Thema in Forschung und Entwicklung ist, ist dies bei der 2D Grafik nicht der Fall. Das Thema 2D Grafik wird dabei momentan wieder wichtiger, da sogenannte Mobile Games einen immer größeren Marktanteil bekommen[1]. Diese Spiele werden sehr häufig noch in 2D Grafik realisiert[2].

Raytracing ist eine Technik, welche eine fotorealistische globale Beleuchtung in der Computergrafik ermöglicht. Aufgrund des hohen Rechenaufwandes ist die Anwendung der Technik bis vor kurzem nur in der offline Bildsynthese möglich gewesen. Neue Entwicklungen beim Thema Raytracing und bessere Hardware ermöglichen jedoch immer höhere Bildraten, was den Einsatz in der Echtzeitgrafik immer näher rücken lässt[3]. Momentan beschränkt sich der Einsatz aber immer noch auf einen hybriden Einsatz für bestimmte Beleuchtungseffekte in Kombination mit normaler auf Rasterung basierender Computergrafik[4].

Ziel dieser Arbeit ist es diese Entwicklungen in einem System, welches die Beleuchtung für eine 2D Szene berechnet, zu bündeln, um für diese Szene eine globale Beleuchtung zu ermöglichen. Die Reduktion auf 2D sollte die Komplexität des Raytracings soweit verringern, dass eine Echtzeit Anwendung möglich ist.

Ziel ist es weder eine komplette Engine noch eine komplette Rendering Pipeline zu schreiben. Vielmehr geht es um ein Framework, welches die Berechnung der globalen Beleuchtung einer Szene ermöglicht. Die Ergebnisse aus diesem System können dann in einer kompletten Rendering Pipeline verwendet werden.

[1] *https://arstechnica.com/gaming/2018/05/mobile-platforms-now-account-for-more-than-half-of-all-game-spending/*, abgerufen am 14.07.19
[2] *https://www.linkedin.com/pulse/why-develop-2d-games-mobile-instead-3d-ahmad-hammad*, abgerufen am 14.07.2019
[3] *https://developer.nvidia.com/rtx/raytracing*, abgerufen am 02.07.2019
[4] *https://www.nvidia.com/en-us/geforce/news/geforce-gtx-dxr-ray-tracing-available-now/*, abgerufen am 08.06.2019

Hierzu werden zuerst die physikalischen und mathematischen Grundlagen der Computergrafik erörtert. Danach wird das Thema globale Beleuchtung vorgestellt und formal beschrieben. Anschließend werden Algorithmen aus der 3D Computergrafik vorgestellt, welche versuchen das Problem der globalen Beleuchtung zu lösen. Ihre Vor- und Nachteile werden dabei herausgestellt. In der Problemstellung wird daraufhin ausgeführt, wie die globale Beleuchtung in 2D übertragen werden kann. Aufbauend darauf wird im Konzept ein Lösungsansatz entwickelt, wie das Problem der globalen Beleuchtung in 2D mit Raytracing gelöst werden kann. Am Ende der Arbeit werden die erzielten Ergebnisse auf ihre Optik und Laufzeiten hin untersucht. Abschließend enthält das Fazit eine Bewertung der Ergebnisse und ein Ausblick auf eventuelle Erweiterungen.

Die Arbeit baut auf meiner Hausarbeit „Echtzeit Raytracing für 2D Grafik“ auf, welche im Sommersemester 2018 als Prüfungsleistung für das Wahlpflichtfach „Computergrafik für Augmented Reality“ an der HAW Hamburg bei Prof.Dr. Philipp Jenke angefertigt wurde.

2 Grundlagen

Hier werden die Grundlagen der Computergrafik und der globalen Beleuchtung erläutert.

2.1 Mathematische Grundlagen

Für Berechnungen der Computergrafik benötigt man mathematische Grundlagen, diese werden hier dargestellt.

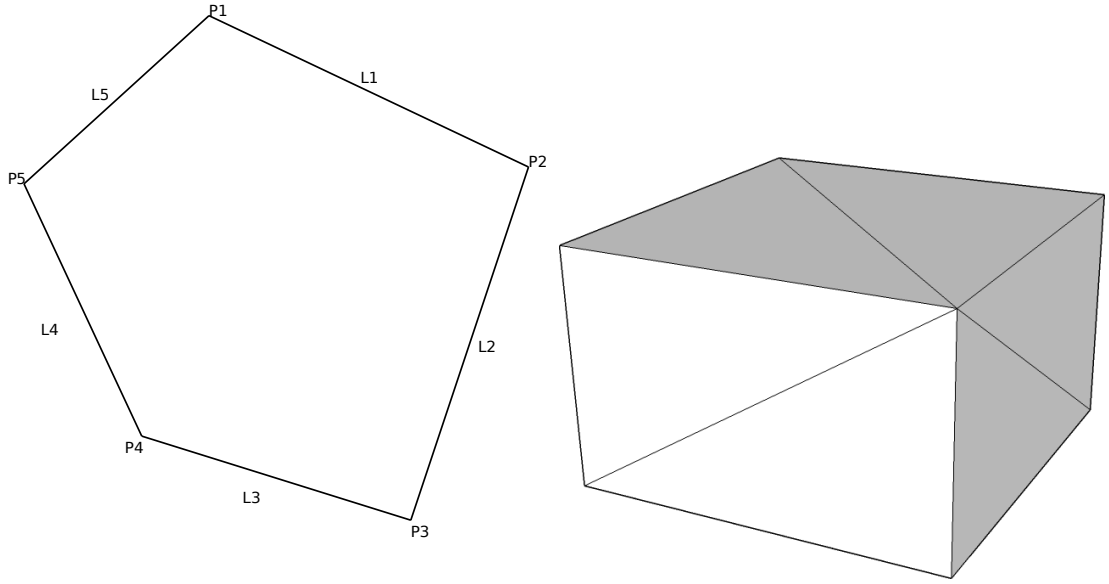
2.1.1 Polygone

Ein Polygon ist eine geometrische Form, welche durch Eckpunkte und die zwischen diesen verlaufenden Kanten definiert wird. In der Computergrafik werden häufig 3D Polygone verwendet welche sich dann aus dreieckigen Facetten zusammensetzen (sog. Meshes, Abb. 2.1). 2D Polygone können folgende für Computergrafik relevante Eigenschaften haben (Abb. 2.2):

- **Überschlagen**: zwei Kanten des Polygons kreuzen sich.
- **Lochfrei**: das Polygon hat kein Loch im Inneren.
- **Kantenzahl pro Punkt**: die Anzahl der Kanten pro Punkt.

2.1.2 Strahl

Als Strahl ist eine Linie, welche auf einer Seite begrenzt ist. Er wird definiert durch den Ursprungspunkt r_o und die Richtung r_d.



(a) 2D Polygon definiert durch die Punkte P1 bis P5 und Kanten L1 bis L5

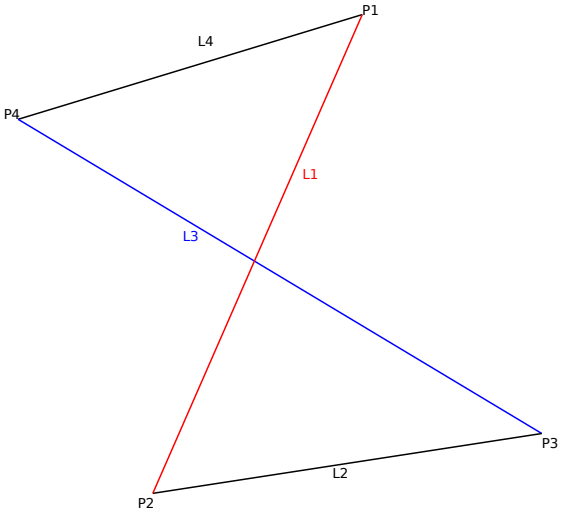
(b) Quader, 3D Polygon aus Dreiecken zusammengesetzt

Abbildung 2.1: 2D und 3D Polygon

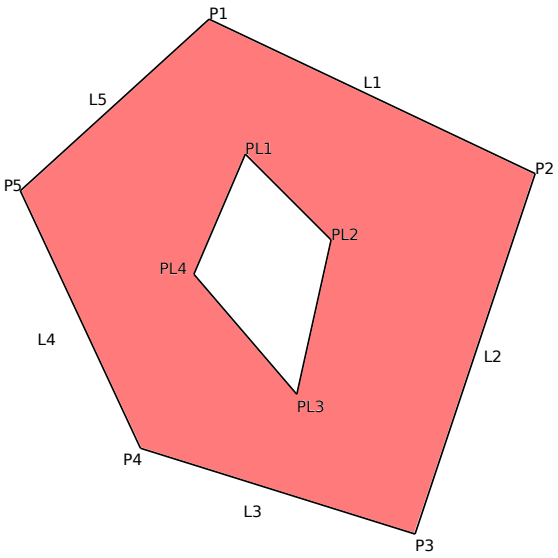
2.1.2.1 Normale

Als Normale (oder auch Normalenvektor) wird ein Vektor bezeichnet, welcher senkrecht auf einer Facette steht (Abb. 2.3). Die Normale einfacher Facette können mit Hilfe des Kreuzproduktes zweier nicht paralleler Kanten der Fläche berechnet werden. Statt die Normale der Facetten zu verwenden, was ein sehr kantiges Bild zur Folge hat, kann eine Punktnormale verwendet werden. Zur Berechnung dieser wird der Durchschnitt der angrenzenden Normalen der Facetten am Punkt gebildet. Um eine geglättete Oberfläche zu erhalten, kann zwischen den Punktnormalen linear interpoliert werden[23]. Als Normale einer Kante bei einem 2D-Polygon, kann ein zur Kante orthogonale Vektor betrachtet werden. Um eine Orthogonale zu einer Kante eines 2D-Polygons zu bestimmen, reicht das einfache Tauschen von x- und y-Werten und ein Vorzeichenwechsel:

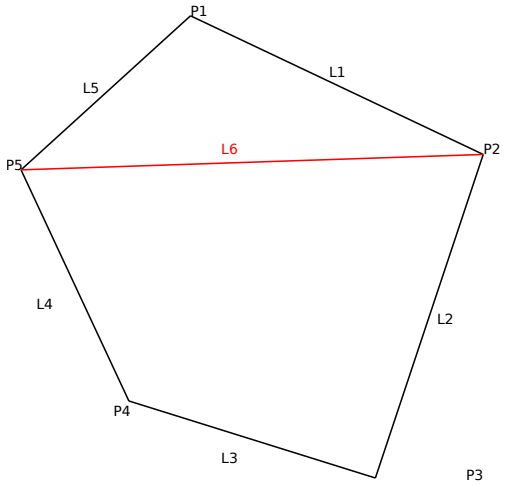
$$\begin{aligned} &\text{(1) Rotation um } +0.5\pi \quad \begin{pmatrix} x \\ y \end{pmatrix} \Rightarrow \quad \begin{pmatrix} -y \\ x \end{pmatrix} \\ &\text{(2) Rotation um } -0.5\pi \quad \begin{pmatrix} x \\ y \end{pmatrix} \Rightarrow \quad \begin{pmatrix} y \\ -x \end{pmatrix} \end{aligned} \tag{2.1}$$



(a) Überschlagenes Polygon: Die Linien L3 und L1 kreuzen sich

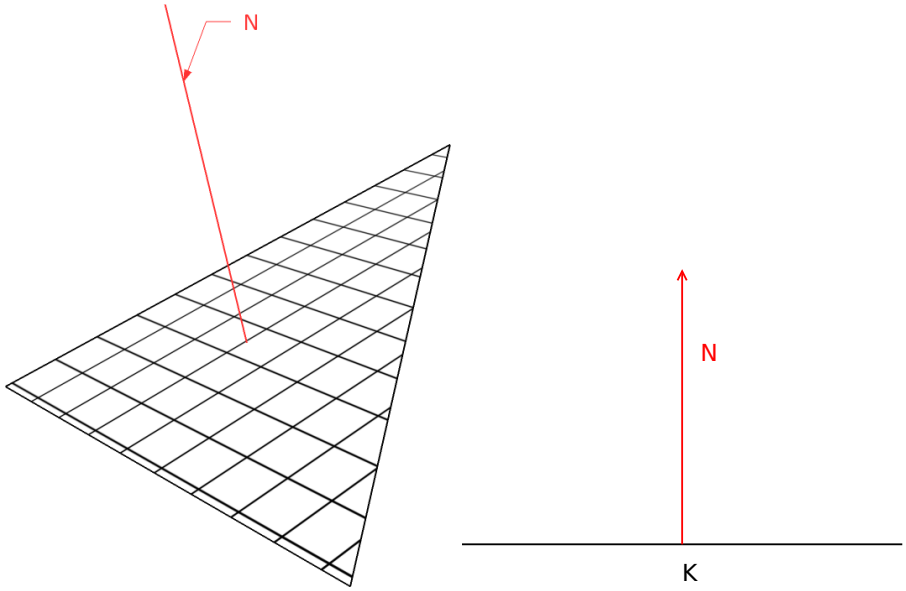


(b) Nicht lochfreies Polygon, die Punkte PL1 bis PL4 bilden ein Loch



(c) Polygon mit Querverbindung L6, die Punkte P5 und P2 sind mit mehr als zwei Kanten verbunden

Abbildung 2.2: Eigenschaften von Polygonen



(a) Oberflächennormale eines 3D Dreickspolygon

(b) Die Normale einer Kante entspricht der Orthogonalen dieser Kante

Abbildung 2.3: Normalen

2.1.2.2 Reflexion an einer Normalen

Für die Berechnung der globalen Beleuchtung ist es wichtig, die Reflexion von Licht berechnen zu können. Sei v der zu reflektierende Lichtstrahl in Form eines Vektors, n die Normale der Kante, an dem Punkt p der Oberfläche des Polygons, an dem er reflektiert wird und s das Vektorprodukt von v und n. Dann gilt:

$$r = v - 2s * n \tag{2.2}$$

Wobei r der reflektierte Strahl in Form eines Vektors mit Ursprung p ist. Es gilt die Regel, das Einfallswinkel gleich dem Ausfallswinkel ist.

2.1.3 Schnitt zwischen Kanten und Strahlen

Die Entfernung des Schnittpunktes zwischen einer Kante und einem Strahl kann mit der folgenden Formel berechnet werden[1]:

[1] *https://rootllama.wordpress.com/2014/06/20/ray-line-segment-intersection-test-in-2d/*, abgerufen am 19.07.2019

$$\begin{aligned} \vec{v_1} &= \vec{r_o} + \vec{p_1} \\ \vec{v_2} &= \vec{p_2} - \vec{p_1} \\ \vec{v_3} &= \vec{r_d} + 0.5\pi \\ dot_{v_2v_3} &= \vec{v2} \cdot \vec{v3} \\ t_1 &= \frac{\vec{v_2} \times \vec{v_1}}{dot_{v_2v_3}} \qquad t_2 = \frac{\vec{v_1} \cdot \vec{v_3}}{dot_{v_2v_3}} \end{aligned} \tag{2.3}$$

t_1 ist dabei die Entfernung zwischen r_o und der Linie, die p_1 und p_2 bilden. t_2 ist die Stelle zwischen p_1 und p_2, an der der Strahl auf die Linie trifft. Zwischen p_1 und p_2 ist $1 \geq t_2 \geq 0$ Der Strahl trifft also wenn $t_1 > 0$ und $1 \geq t_2 \geq 0$.

2.2 Physikalische Grundlagen

Die globale Beleuchtung nähert reale Effekte an. Die physikalischen Effekte, welche als Grundlage für die Berechnung der globalen Beleuchtung dienen, werden hier vorgestellt.

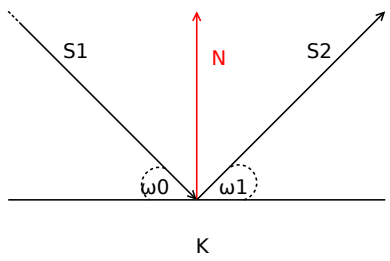
2.2.1 Reflexion

Licht kann entweder diffus oder gerichtet reflektiert werden. Bei einer gerichteten Reflexion wird das Licht entsprechend der Formel *Einfallswinkel* $\omega 0$ *gleich Ausfallswinkel* $\omega 1$ reflektiert (Abb. 2.4). Bei einer perfekt diffusen Reflexion wird das Licht gleichmäßig in alle Richtungen gestreut. Die Reflexionsarten können jedoch auch in Kombination auftreten (Abb. 2.4 (c))[22].

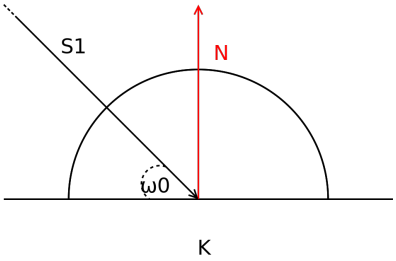
2.2.2 Lambertsches Gesetz

Das *Lambertsche Gesetz* besagt, dass die Bestrahlungsstärke umso kleiner wird, je größer der Winkel zwischen der Oberflächennormalen und der Einfallsrichtung des Lichtes ist. Der Faktor ist hierbei der Kosinus dieses Winkels (Abb. 2.5)[19].

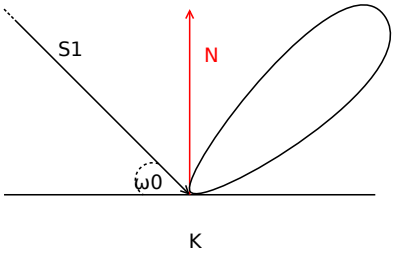
$$I = I_0 * \cos\theta_i \tag{2.4}$$



(a) Perfekt gerichtete Reflexion



(b) Perfekt diffuse Reflexion



(c) Kombination

Abbildung 2.4: Verschiedene Arten der Reflexion des Lichtstrahls S1 an K

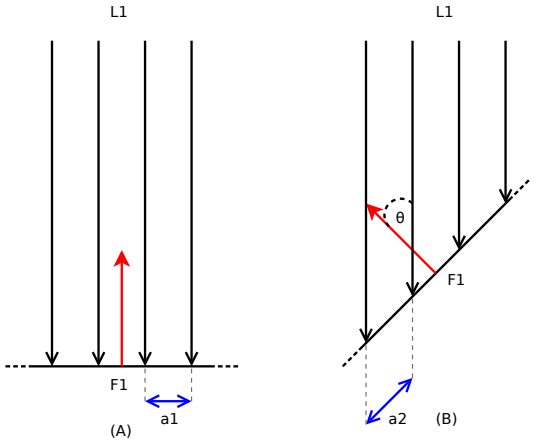


Abbildung 2.5: Prinzip des Lambertschen Gesetzes. Licht trifft auf von L1 auf die Fläche F1. Stellt man nun die Fläche F1 schräg auf, so wird der Abstand a2 im Vergleich zu a1 größer. Das gleiche Licht muss also eine größere Fläche beleuchten. Die Bestrahlungsstärke nimmt ab. θ ist der Winkel zwischen Oberflächennormale und Lichteinfallswinkel

2.3 Globale Beleuchtung

Globale Beleuchtung (engl. Global Illumination) bezeichnet ein Modell der Computergrafik, bei dem nicht nur die direkte Beleuchtung durch Lichtquellen zur Berechnung der

Beleuchtung von Objekten berücksichtigt wird, sondern auch die Licht-Wechselwirkung zwischen Objekten.

2.4 Generelle Anforderungen an globale Beleuchtung

Zur Berechnung von globaler Beleuchtung müssen folgende Dinge berücksichtigt werden[22]:

2.4.1 Kamera

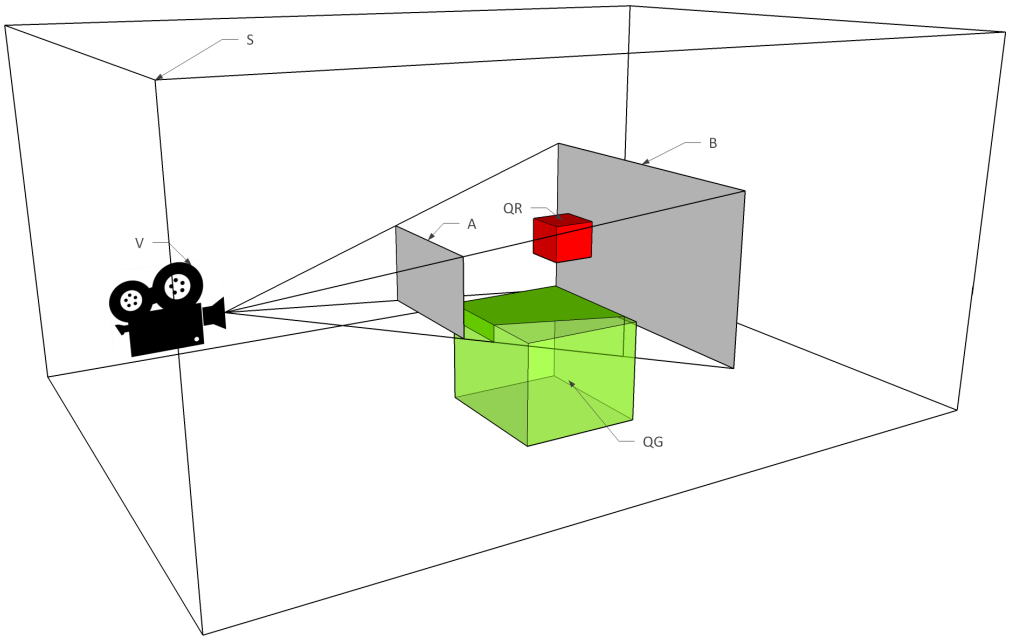


Abbildung 2.6: Skizze des Kameramodells in der 3D Bildsynthese. Nicht im View Frustum liegende Objektbereiche werden durchsichtig dargestellt.

Um eine Szene auf einem Bildschirm abzubilden, muss die Sicht auf die Szene definiert werden. Dies geschieht mit einer virtuellen Kamera. Abbildung 2.6 zeigt eine virtuelle Kamera V in der Szene S. Die Kamera V ist dabei selbst Teil der Szene S. Sie befindet sich also im gleichen Koordinatenraum wie die Objekte der Szene. Dabei erfasst sie nur Teile der Szene welche zwischen der vorderen Clippingebene A und der hinteren Clippingebene B, sowie den Seiten des Kegels liegen. Diesen Bereich nennt man View Frustum. Nur Objekte im View Frustum werden gerendert. Die im View Frustum liegenden Teile der Szene werden auf die vordere Clippingebene A projiziert. Der Quader QR liegt komplett

im View Frustum, der Quader QG nur teilweise. Das von QR und QG reflektierte Licht kann direkt auf die Kamera treffen und wird dort zu Bildsynthese verwendet.

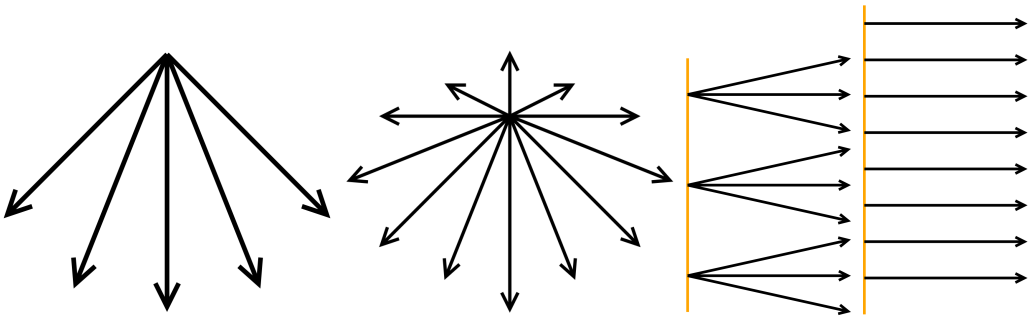
2.4.2 Lichtquellen

Jede Szene benötigt Lichtquellen, welche zumindest Teile der Szene direkt beleuchten, damit diese das Licht reflektieren können. Hierbei muss beschrieben werden, wie diese Lichtquellen das Licht in die Szenen abgeben. Lichtquellen können folgende Eigenschaften haben (Abb. 2.7)[24]:

- **Punktlicht/Volumen:** Lichtquellen können entweder ein Punktlicht sein oder ein Volumen beliebiger Form besitzen. Bei einem Punktlicht geht das Licht von genau einem Punkt aus. Im Falle einer Volumenlichtquelle geht das Licht von mehr als einem Punkt aus, z.B. von der Kante eines Polygons. Dies ermöglicht dann Effekte wie weiche Schatten.
- **Gerichtet/Diffus:** Gerichtete Lichtquellen geben ihr Licht nur in eine Richtung ab bzw. in einem Winkel. Komplett diffuse Lichtquellen jedoch gleichmäßig in alle Richtungen. Eine Lichtquelle muss weder komplett gerichtet noch komplett diffus sein.
- **Leistungsdichte:** Die Leistungsdichte beschreibt, wie viel Licht die Lichtquelle in eine Richtung abgibt. Eine Punktlichtquelle welche rundum strahlt, muss z.B. nicht in alle Richtungen gleich viel Licht abgeben.
- **Abstrahlwinkel:** Der Abstrahlwinkel beschreibt, zwischen welchen Winkeln die Lichtquelle Licht abstrahlt und zwischen welchen nicht. Es kann auch als Sonderform der Leistungsdichte mit Bereichen ohne Leistungsdichte gesehen werden. Dies ermöglicht Spotlights.
- **Ambient light:** Grundhelligkeit einer Szene. Hat keine bestimmte Quelle. Das Licht wird gleichmäßig auf alle Flächen verteilt.

2.4.3 Oberflächenstreuung

Licht wird an der Oberfläche von Objekten reflektiert. Dafür muss die Oberfläche des Objekts so beschrieben sein, dass bestimmt werden kann wie viel Licht das Objekt



(a) Punktlicht mit Abstrahlwinkel von ca. $\frac{1}{2}\pi$ (b) Punktlicht mit unterschiedlicher Leistungsdichte (c) Diffuse Volumenlichtquelle (d) Gerichtete Volumenlichtquelle

Abbildung 2.7: Skizzen Lichtquellen

reflektiert, wie es das Licht reflektiert wird und welchen Anteil des Lichts es reflektiert und welchen es absorbiert.

2.4.4 Transmission

Nicht alle Objekte Reflektieren und/oder absorbieren Licht komplett[22]. Licht kann auch in Objekte eindringen und wieder austreten. Hierbei kann es evtl. gebrochen und gestreut werden. Ein Beispiel hierfür sind dielektrische Materialien wie Glas (Brechung)[27].

2.4.5 Indirekte Beleuchtung

Objekte reflektieren Licht. Das heißt, dass für die Beleuchtung von Objekten nicht nur das Licht der eigentlichen Lichtquellen relevant ist, sondern auch das Licht, welches die Objekte reflektieren.

2.5 Rendergleichung

Eine allgemeine formale Beschreibung für Bildsynthese-Verfahren ist die Rendergleichung[15]. Die Berechnung von globaler Beleuchtung beruht auf dem Lösen der Rendergleichung. Da alle Bildsynthese Algorithmen versuchen das Gleiche zu tun, und zwar die Ausbreitung von Licht in einer Szene zu bestimmen, kann die Rendergleichung, welche dieses

exakt beschreibt, als Grundlage und/oder als Vergleich für deren Qualität herangezogen werden.

Bei der Rendergleichung handelt es sich um eine Approximation der geometrischen Optik. Da es sich hierbei um die geometrische Optik handelt, werden Lichteigenschaften wie Lichtgeschwindigkeit, Zeit und Lichtspektrum außer acht gelassen, was z.B. Effekte wie Beugung unmöglich macht. Statt der von ursprünglich formulierten *Surface Form* wird hier die anschaulichere *Path Form* der Gleichung verwendet[22]. Die Rendergleichung lautet:

$$L_o(x, \omega_o) = L_e(x, \omega_o) + L_r(x, \omega_o) \tag{2.5}$$

$L_o(x, \omega_o)$: ausgehendes Licht von Punkt x in Richtung ω_o
$L_e(x, \omega_o)$: Licht, welches der Punkt x in Richtung ω_o abgibt, wenn er selbst eine Lichtquelle ist
$L_r(x, \omega_o)$: Licht, welches der Punkt x in Richtung ω_o reflektiert.
Der Term L_r wird hierbei durch die Reflexionsgleichung bestimmt:

$$L_r(x, \omega_o) = \int_\Omega f(x, \omega_o, \omega_i) L_i(x, \omega_i) \cos\theta_i d\omega_i \tag{2.6}$$

Das Integral $\int_\Omega$ beschreibt in der Gleichung das Licht, welches aus allen Richtungen (alle möglichen ω_i) auf x trifft. Es beschreibt also eine bedeckende Hemisphäre um x. Bei $f(x, \omega_o, \omega_i)$ handelt es sich um eine Bidirectional Reflectance Distribution Function (BRDF). Eine BRDF beschreibt das Reflexionsverhalten eines Materials, indem sie den Quotienten der Bestrahlungsstärke des Punktes x aus Richtung ω_i und der Strahlungsdichte des Punktes x in Richtung ω_o bestimmt. $L_i(x, \omega_i)$ ist das Licht, welches aus Richtung ω_i auf x trifft. $\cos\theta_i$ dient der Berücksichtigung des Lambertschen Gesetzes. Löst man nun die Rendergleichung für alle Oberflächenpunkte auf allen Objekten der Szene, so hat man eine perfekte globale Beleuchtung berechnet. Dies ist jedoch nicht möglich, da die Oberflächen eines jeden Objektes unendlich viele Punkte besitzt und dadurch, dass jeder Punkt der Szene von jedem anderen abhängig ist, eine unendliche Rekursion entsteht. Dies ist das grundlegende Problem bei der Berechnung von globaler Beleuchtung.

Eine Erweiterung ist das Hinzufügen der Transmission zur Rendergleichung. Bei Transmission muss an der Oberfläche des Objektes also Licht berücksichtigt werden, welches aus dem Objekt austritt. Die Rendergleichung ändert sich also nun zu:

$$L_o(x, \omega_o) = L_e(x, \omega_o) + L_r(x, \omega_o) + L_t(x, \omega) \tag{2.7}$$

$L_t(x, \omega)$: Licht, welches das Objekt an der Stelle x in Richtung ω aus dem Objekt auf dem x liegt, austritt. In $L_t(x, \omega)$ würde die BRDF mit einer Bidirectional Transmittance Distribution Function (BTDF) ersetzt werden[21] und die Hemisphäre ins Innere des Objektes gespiegelt werden. Statt nun ein weiteres Integral für $L_t(x, \omega)$ zu bilden, zieht man die Terme $L_r(x, \omega_o)$ und $L_t(x, \omega)$ zusammen. Um dieses Licht berücksichtigen zu können, muss die Hemisphäre Ω über dem Punkt x durch eine Sphäre S um x ersetzt werden, damit auch Licht, welches von innerhalb des Objektes kommt, vom Integral berücksichtigt wird. Da $L_r(x, \omega_o)$ und $L_t(x, \omega)$ zusammengefasst wurden, müssen auch BRDF und BTDF zusammengefasst werden. Fasst man BRDF und BTDF zusammen, spricht man von einer Bidirectional Scattering Distribution Function (BSDF)[4]. Die BSDF ermöglicht nun Effekte wie Brechung, transparente und diffuse Objekte. Die Rendergleichung lautet nun:

$$L_r(x, \omega_o) = \int_S f'(x, \omega_o, \omega_i) L_i(x, \omega_i) \cos\theta_i d\omega_i \tag{2.8}$$

Wobei es sich bei $f'(x, \omega_o, \omega_i)$ nun um eine BSDF handelt. Die BSDF erlaubt jedoch immer noch keine Volumenstreuung. Hierzu benötigt man eine Bidirectional Scattering Surface Reflectance Distribution Function (BSSRDF)[14].

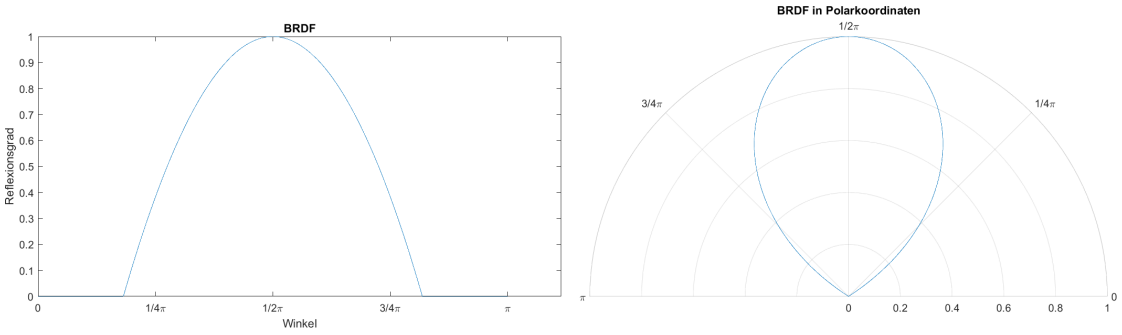


Abbildung 2.8: Beispiel einer BRDF

3 Methoden zur Berechnung globaler Beleuchtung

Es gibt viele Methoden zur Berechnung von globaler Beleuchtung. Diese unterschiedlichen Methoden haben jeweils Vor- und Nachteile oder können globale Beleuchtung nicht vollständig berechnen.

3.1 Lichtpfadnotation

Die Lichtpfadnotation[12] definiert reguläre Ausdrücke, um den Weg des Lichtes von der Lichtquelle zur Kamera zu beschreiben. Hierfür wird folgendes Alphabet definiert:

- **L** Lichtquelle
- **D** *(engl. diffuse)* Reflexion an diffuser Oberfläche
- **S** *(engl. specular*) gerichtete Reflexion an spiegelnder Oberfläche
- **E** *(engl. eye)* Auge/Kamera

Mit diesen kann nun ein Lichtpfad beschrieben werden (z.B. LSDSE Lichtquelle → gerichtete Reflexion → diffuse Reflexion → gerichtet Reflexion → Kamera. Möchte man hingegen die Menge der darstellbaren Lichtpfade eines Algorithmus beschreiben, so werden diese Quantoren hinzugenommen:

- | Oder
- * Beliebig viele
- + Beliebig viele, aber min. 1
- ? 0 oder 1

Ein Algorithmus, der die Rendergleichung (Gleichung (2.8)) löst, kann die Menge der Lichtpfade L(D|S)*E darstellen.

3.2 Erwartungstreue und Konsistenz

Bildsynthese Algorithmen machen bei der Berechnung des finalen Bildes Fehler. Die Art dieser Fehler entscheiden über Erwartungstreue und Konsistenz eines Algorithmus. Bildsynthese Algorithmen, welche mit größer werdender Sample-Größe gegen die richtige Lösung konvergieren, werden als konsistent bezeichnet. Bildsynthese Algorithmen, welche keine vereinfachenden Annahmen in die Rendergleichung (Gleichung (2.8)) einführen, werden als erwartungstreu bezeichnet. Bei ihnen kann der Fehler nur durch zu geringe Anzahl an Samples entstehen (Varianz). Rendering Algorithmen, welche die Rendergleichung verändern, werden als verzerrt (nicht erwartungstreu) bezeichnet. Eine Verzerrung kann z.B. sein, die Anzahl der maximalen Reflexionen zu begrenzen oder nur diffuse Reflexionen zu unterstützen. Auch verzerrte Algorithmen können konsistent sein[7].

3.3 Radiosity

Radiosity ist neben dem Raytracing eines der ältesten Verfahren zur Berechnung von globaler Beleuchtung[10]. Die Grundidee von Radiosity ist, dass Licht in der Umgebung von verschiedenen Objekten hin und her reflektiert und absorbiert wird. Diese Objekte bestehen aus einzelnen Flächen, welche im Algorithmus Patches genannt werden. Jede Fläche ist somit auch eine Lichtquelle, aufgrund des Lichtes, dass sie zurückwirft. Die Grundvereinfachungen von Radiosity sind:

1. Die eingehende und ausgehende Lichtintensität (auch genannt Radiosity) wird nur pro Patch berechnet, sie ist also auf allen Punkten innerhalb des Patches gleich.
2. Alle Flächen reflektieren perfekt diffus.

Diese beiden Vereinfachungen sorgen dafür, dass Einfallswinkel (der Winkel $\cos\theta_i$ aus Gleichung (2.6) findet pro Patch Beachtung, jedoch nicht die BRDF pro Punkt) und Einfallpunkt des Lichtes auf dem Patch vernachlässigt werden können. Die Rendergleichung (Gleichung (2.8)) besagt, dass die Beleuchtung jedes Punktes von jedem anderen Punkt rekursiv abhängt. Damit hängt auch jeder Patch von jedem anderen sichtbaren

Patch ab. Somit muss nur noch die Lichtwechselwirkung der einzelnen Patches bestimmt werden. Dafür wird für jedes Patch zu jedem anderen Patch ein sogenannter Formfaktor berechnet. Dieser gibt an wie viel Licht von einem Patch zum anderen gelangt. Der Formfaktor hängt davon ab:

- wie groß die Patches sind
- den Winkel der Patches zueinander
- dem Abstand der Patches zueinander
- ob die Patches gegenseitig teilweise oder ganz durch andere Patches verdeckt werden

Der Formfaktor muss für jedes Paar von Patches bestimmt werden. Für die Radiosity eines Patches ergibt sich folgende Formel:

$$B_i = E_i + p_i \sum_{j=0}^{n} B_j F_{ji} \tag{3.1}$$

B_i ist die Radiosity des Patches. E_i ist das Licht, das der Patch emittiert (für Lichtquellen). p_i ist der Reflexionsgrad des Patches. $\sum_{j=0}^{n} B_j F_{ji}$ ist die Summe des eintreffenden Lichtes von allen n andern Patches B_j, gewichtet nach dem entsprechenden Formfaktor F_{ji}. Da angenommen wird, dass das Licht unendlich schnell ist, stellt sich sofort ein Gleichgewicht innerhalb der Szene ein. Die oben stehende Formel lässt sich nun für jedes Patch anhand eines Gleichungssystems lösen.

$$\begin{bmatrix} 1 & -p_1F_{12} & \cdots & -p_1F_{1n} \\ -p_1F_{21} & 1 & \cdots & -p_1F_{2n} \\ \vdots & \vdots & \cdots & \vdots \\ -p_nF_{n1} & -p_1F_{n2} & \cdots & 1 \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{bmatrix} \tag{3.2}$$

Aufgrund der gemachten Vereinfachungen kann Radiosity nur **LD*E** Lichtpfade darstellen.

Lichtpfad	Erwartungstreu	Konsistent
LD*E	×	×

3.4 Voxel Cone Tracing

Voxel Cone Tracing global Illumination (VXGI)[8] basiert auf der Voxelisierung der zu beleuchtenden Szene. Diese Voxel werden dann in einem *Sparse Voxel Octree (SVO)* gespeichert. Dieser ermöglicht es, auf Voxel auch in Gruppen zuzugreifen (Daher die Baumstruktur), wobei die Szene hierbei immer in 8 Voxel geteilt wird.

In den Voxeln wird dann die Information über das von Oberflächen der eigentlichen Szene abgegebene Licht gespeichert. Dieses wird vorher mit einer anderen Methode direkt auf der Szene berechnet (z.B. Blinn-Phong-Modell). Dies geschieht in der Form, dass pro Seite des Voxels gespeichert wird, wie viel Licht er abgibt. Die indirekte Beleuchtung wird nun berechnet, indem ein Kegel ähnlich eines Lichtstrahls von der Oberfläche in die Szene geschossen wird und dabei das Licht aus den Voxeln, die er schneidet, sammelt. Dies hat im Gegensatz zu Strahlverfolgungverfahren (Abschnitt 3.6) den Vorteil, dass viele Strahlen in einem Kegel repräsentiert werden können. Da eine genaue Schnittberechnung mit den Voxeln zu aufwendig wäre, wird diese durch gezielte Lookups im SVO ersetzt (3.2). Wie alle volumetrischen Verfahren hat VXGI Probleme mit Bleeding. Dies bezeichnet den Effekt, dass Licht durch Objekte dringt, wo dies nicht beabsichtigt ist. Durch die Approximation der Voxel und Kegel gehen außerdem große Anteile der Richtungsinformationen des Lichtes verloren. Die so berechnete globale Beleuchtung wird dann in einem zusätzlichen Rasterungs-Schritt verwendet. Da Lichtanteile verloren gehen, ist VXGI weder erwartungstreu noch konsistent.

Lichtpfad	Erwartungstreu	Konsistent
LD*	×	×

3.5 Light Propagation Volumes

Die Idee von Light Propagation Volumes (LPV) ist es, die Szene in ein gleichmäßiges dreidimensionales Gitter aufzuteilen. In den einzelnen Zellen dieses Gitters wird dann die Strahldichte und Richtungsverteilung (Lichtverteilung) des Lichtes gespeichert. Die Lichtverteilung innerhalb der Zellen wird mithilfe von *Spherical Harmonic Functions (SHF)* gespeichert. Diese stellen eine 2D Funktion auf einer Kugel in der Frequenzdomäne dar (Ähnlich einem *Fourier transformierten* Bild auf einer Cubemap). Umso höher die Ordnung der SHF, umso genauer können hochfrequente Anteile (nicht im Sinne der

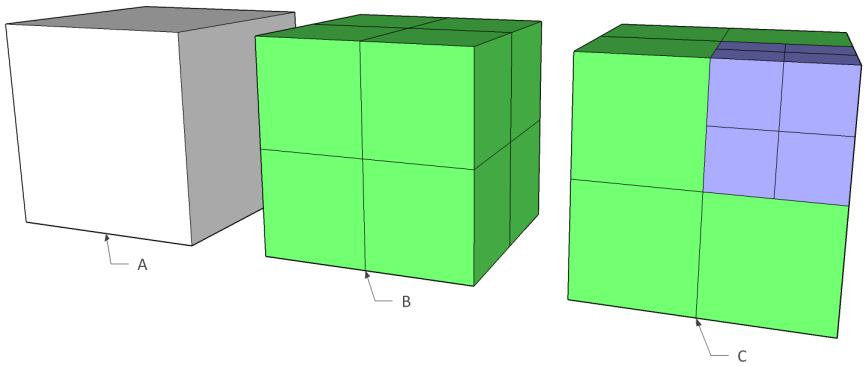


Abbildung 3.1: Sparse Voxel Octree - Speichert Voxel hierarchisch, indem ein Knoten (Voxel) immer zweifach geteilt wird. A ist dabei der Wurzelknoten, B die zweite Ebene (Kindknoten von A), C die dritte. Dies sind dann die Blattknoten. Eine Ebene muss nicht voll gefüllt sein. So wird das unnötige Speichern von leeren und hochaufgelösten Voxeln vermieden

Frequenz des Lichtes, sondern im Sinne der Frequenzen der Richtungsinformationen in der SHF) dargestellt werden (Umso mehr Koeffizienten hat die SHF). Umso mehr hochfrequente Anteile, umso genauer können nicht diffuse Lichtanteile dargestellt werden. Durch Erhöhung der Ordnung der SHF steigt jedoch auch der Rechenaufwand. Die Berechnungsschritte mit LPV sind:

1. Berechnung der direkten Beleuchtung der Szene
2. Speichern des Lichts, welches durch die direkte Beleuchtung reflektiert wird in die Zellen am Rand der beleuchteten Fläche
3. Propagation, die Zellen sammeln Licht von ihren umgebenen Zellen und modifizieren ihre eigenen SPH entsprechend des abgegebene Lichtes der umgebenen Zellen. Mehrere Iterationen dieses Schrittes sind notwendig, um das Licht weiter zu verteilen.
4. Rasterung

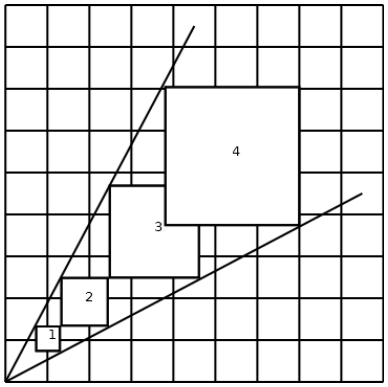


Abbildung 3.2: Tracing eines Kegels entlang seines Zentrums mithilfe von Lookups im SVO (vereinfachte Skizze in 2D). Der Lookup findet in den Octree Ebenen kleiner und größer des Lookup Voxels statt. Dazwischen wird interpoliert. So würde der Lookup für 1 in unteren Baumebenen stattfinden. Der für 4 in weiter oben gelegenen

Da die SHF nur begrenzt viele Dimensionen haben, ist VXGI nicht erwartungstreu und nicht konsistent.

Lichtpfad	Erwartungstreu	Konsistent
L(D|S){1,2}	×	×

3.6 Raytracing (Strahlverfolgung)

Raytracing Algorithmen versuchen zwischen der Lichtquelle und der Kamera einen Lichtpfad in Form eines oder mehrerer Strahlen zu konstruieren. Entlang der Strahlen des Pfades wird dann der Lichttransport von der Lichtquelle zu Kamera bestimmt. Die Strahlen werden entsprechend der Objekte, auf die sie treffen, reflektiert, gestreut oder gebrochen. Raytracing Algorithmen sind grundlegend in drei verschiedene Klassen gegliedert:

- **Kamera zur Lichtquelle:** Es wird ein Pfad von der Kamera zur Lichtquelle konstruiert.
- **Lichtquelle zur Kamera:** Es wird ein Pfad von der Lichtquelle zur Kamera konstruiert. Hier wird häufig der Begriff Particle Tracing verwendet[22].

- **Hybride**: Der Lichtpfad wird von beiden Seiten konstruiert und irgendwann verbunden.

Die Strahlverfolgung zur Lichtberechnung in der Computergrafik ist bereits lange bekannt[2]. Der Begriff Raytracing wird häufig als Überbegriff für alle auf Strahlverfolgung basierenden Algorithmen verwendet, kann jedoch auch einen bestimmten Algorithmus beschreiben (Abschnitt 3.6.2).

3.6.1 Ray Casting

Ray Casting (Begriff geprägt durch[25]) ist der Ursprung aller strahlbasierten Bildsynthese Algorithmen. Hierbei handelt es sich nicht um eine Methode zu Berechnung von globaler Beleuchtung, sondern nur um eine Methode zur Lösung des Sichtbarkeitsproblems.

3.6.2 Raytracing

Raytracing fügt dem Ray Casting die Berechnung von Schatten hinzu. Vom dem Punkt, an dem der Strahl auf eine Oberfläche trifft, werden Strahlen zu allen Lichtquellen ausgesendet. Trifft der Strahl dabei erneut auf ein Objekt, so ist der Punkt für diese Lichtquelle verdeckt.

3.6.3 Rekursives Raytracing

Rekursives Raytracing[33] fügt dem Raytracing die Fähigkeit indirekter Beleuchtung hinzu. Dies geschieht dadurch, dass nach dem Auftreffen eines Strahls auf ein Objekt dieser gespiegelt wird. Ein Strahl wird so lange gespiegelt, bis er auf eine Lichtquelle trifft (oder ein anderes Abbruchkriterium wie z.B. eine maximale Rekursionstiefe erreicht). Der Algorithmus ist durch die Limitation, dass pro Strahl, der auf eine Oberfläche trifft, nur ein neuer Strahl generiert wird, nur in der Lage perfekt spiegelnde oder brechende Oberflächen darzustellen. Der $L_i(x, \omega)$ Term der Rendergleichung (Gleichung (2.8) wird also pro Punkt auf eine Richtung limitiert. Somit kann keine diffuse Reflexion berechnet werden. Der Algorithmus ist also nicht erwartungstreu.

Lichtpfad	Erwartungstreu	Konsistent
LS*E	✗	✓

3.6.4 Distributed Raytracing

Distributed Raytracing[6] (auch *stochastisches* oder *diffuses Raytracing*) führt erstmals die Idee ein, die Strahlen anhand der BRDF der Oberfläche, auf die sie treffen, zu verteilen. Jedoch wird beim Auftreffen eines Strahls nicht mehr als ein Sekundärstrahl generiert. Die Verteilung anhand der BRDF erfolgt nur auf den Strahlen, welche für ein Antialiasing und/oder Überabtastung des Bildes sowieso benötigt werden. Dies schränkt die Möglichkeit, perfekt diffuse Oberflächen darzustellen, welche in der Mitte des Lichtpfades liegen, stark ein. Nur gerichtete Reflexionen mit leicht diffusen Anteil sind so darstellbar.

Lichtpfad	Erwartungstreu	Konsistent
LDS*E	×	✓

3.6.5 Path Tracing

Path Tracing wurde zusammen mit der Rendering Gleichung vorgestellt[15]. Sie bildet die formale mathematische Grundlage zu Distributed Raytracing und führt zusätzliche Sekundärstrahlen ein. Die Crude Monte-Carlo-Integration (MCI) wird verwendet, um das Integral der Rendering Gleichung Gleichung (2.8) zu lösen. Hierbei wird Russian Roulett (RR) verwendet, um die Laufzeit zu verbessern. Sowohl MCI als auch RR erhöhen zwar die Varianz, vereinfachen die Rendergleichung aber nicht.

Lichtpfad	Erwartungstreu	Konsistent
L(D|S)*E	✓	✓

3.6.5.1 Monte Carlo Integration

MCI (deutsch *direkte Monte-Carlo-Integration* oder *randomisierte Quadratur*) ist ein Verfahren zur Lösung von Integralen, welche sich nicht oder nur schwer analytisch oder durch herkömmliche numerische Verfahren lösen lassen. Dies ist beim Integral der Rendergleichung der Fall, da die Funktion, die für einen Punkt die Strahlungsdichte pro Winkel angibt, unbekannt ist. Bei der MCI werden n zufällige Werte w der zu

integrierenden Funktion gewählt. Dann wird die Fläche A des Integrals mit der Formel berechnet:

$$A = \frac{l}{n} * \sum_{i=0}^{n} w_i \tag{3.3}$$

Wobei l die länge des zu integrierenden Funktionsabschnitts angibt.

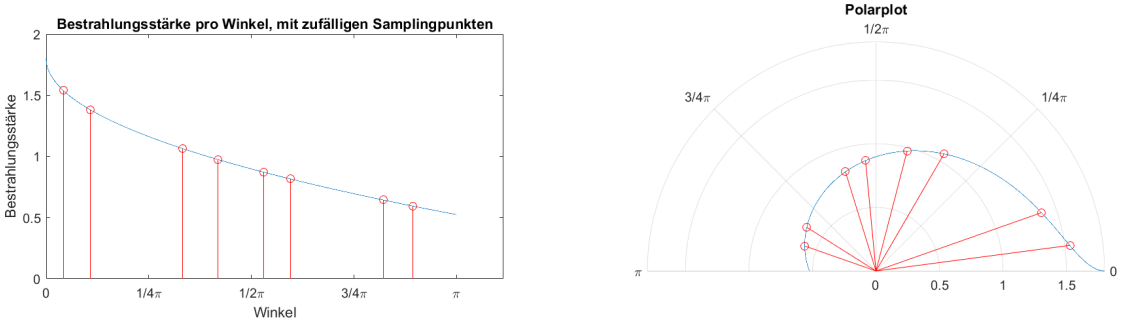


Abbildung 3.3: Beispiel einer Monte-Carlo-Integration. Hier wird angenommen, dass die zu integrierenden Funktion bekannt ist. Die analytisch bestimmte Fläche des Integrals ist hier 2.9818, die Summe der 8 Sampling punkte 7.8868. Mit Gleichung (3.3) mit $l = \pi$ ergibt sich 3.0971

3.6.5.2 Russian Roulett

RR beschreibt eine Technik, bei der Sampling Richtungen ausgelassen werden, welche nur einen geringen Anteil an der Bestrahlungsstärke des Punktes vermuten lassen. Bei der MCI kann dies z.B. eine Richtung sein, in der die BRDF (Abb. 2.8) sehr gering ist, oder ein sehr flacher Einfallswinkel sein, welcher nach dem Lambertschen Gesetz Abschnitt 2.2.2 kaum Beitrag zur Bestrahlungsstärke des Punktes liefert.

3.6.6 Bidirektionales Path Tracing

Die Idee des *Bidirektionalen Path Tracing* (BPT)[18] ist es, statt Pfade nur von Kamera zum Auge oder umgekehrt zu konstruieren, beide Richtungen zu verwenden. Grund dafür dies zu tun, ist, dass das Treffen einer Lichtquelle, welche verdeckt ist, aber dennoch viel Licht in die Szene abgibt, von der Kamera aus sehr schwierig sein kann(Abb. 3.4). Dies hat den Vorteil, dass das die MCI des Integral aus Gleichung (2.8) schneller konvergiert. Die Schwierigkeit beim BPT ist, die Pfade miteinander zu verbinden.

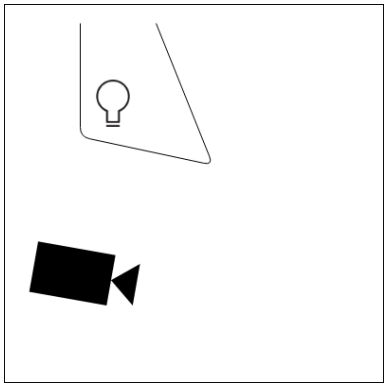


Abbildung 3.4: Skizze einer Szene die für einen von der Kamera zur Lichtquelle operierenden Raytracer schwierig zu lösen ist, da die Lichtquelle schwer zu treffen ist

3.6.7 Metropolis Light Transport

Alle bisher vorgestellten Algorithmen erstellen voneinander unabhängige Samples. Bei *Metropolis Light Transport*[30] werden jedoch die Ergebnisse vorhergegangener Samples berücksichtigt. Samples sind also statistisch korreliert. Hierbei werden vorher gegangene Pfade, welche einen großen Einfluss auf das Bildsynthese Ergebnis haben, mutiert. So können leichter Pfade gefunden werden, welche ebenfalls einen großen Einfluss auf das Bildsynthese Ergebnis haben sollten. Es verbessert also das MCI des Path Tracing. Konsistenz und Erwartungstreue ändern sich dadurch nicht.

Lichtpfad	Erwartungstreu	Konsistent
L(D|S)*E	✓	✓

3.7 Particle Tracing

Particle Tracer werden häufig damit erklärt, dass sie Pakete von Energie (Photonen) von den Lichtquellen basierend auf ihren Eigenschaften (Abschnitt 2.4.2) verteilt in die Szene schießen. Diese werden in der Szene reflektiert, bis sie irgendwann von der Szene absorbiert werden oder auf die Kamera treffen. Particle Tracer genügen Gleichung (2.8)[29]. Verzerrung wird häufig nur durch das Speichern der Partikel auf den Oberflächen (also der Strahldichte der Oberfläche) der Szene erzeugt.

3.7.1 Backward Raytracing

Auch *Light Raytracing*[3] genannt, ist ein Particle Tracing Algorithmus, welcher in 2 Phasen operiert. Erst werden von jeder Lichtquelle Strahlen (Photonen) in die Szene geschossen. Auf den Oberflächen, auf denen diese Photonen auftreffen, wird nun gespeichert, mit wie viel Energie sie reflektiert werden. Dies geschieht mithilfe von *Illumination maps*, einer Art von Textur, welche die Energie des auftreffenden Photons auf vier umliegende Punkte verteilt. Diese Illumination Maps werden dann in der zweiten Phase mit von rekursiven Raytracing ausgewertet. Das Speichern der Photonen in der Illumination Map erzeugt eine Verzerrung.

Lichtpfad	Erwartungstreu	Konsistent
L(D|S)*E	×	✓

3.7.2 Photon Mapping

Photon Mapping[13] funktioniert ähnlich dem Backward Raytracing. Auch beim Photon Mapping werden Photonen auf Oberflächen gespeichert. Dies geschieht in der namensgebenden *Photon Map*. Die Interpolation wir mithilfe einer *density Extimation* durchgeführt. Dabei wird zwischen den gespeicherten Phontonen in der Photon Map anhand ihrer Position und Energie interpoliert. Durch das Interpolieren spart man die Konstruktion weiterer Lichtpfade. Jedoch wird dadurch auch ein *Blur* in den Algorithmus eingefügt. Photon Mapping ist also nicht erwartungstreu. Mit einer hoch genug gewählten Anzahl an Photonen konvergiert es jedoch gegen die Lösung der Rendergleichung.

Lichtpfad	Erwartungstreu	Konsistent
L(D|S)*E	×	✓

3.8 Beschleunigungsdatenstrukturen

Der Großteil der Rechenleistung beim Raytracing wird für die Schnittpunktberechnung mit Objekten benötigt[26]. Um überflüssige Schnittpunktberechnungen zu minimieren, verwenden Raytracing Algorithmen Beschleunigungsdatenstrukturen.

3.8.1 Hüllkörper

Hüllkörper (*engl. bounding volumes*) umschließen ein Objekt um die Schnittpunkt Berechnung zu vereinfachen. Dies geschieht dadurch, dass die Berechnung des Schnittpunktes mit dem Hüllkörper einfacher ist, als die Schnittberechnung mit dem eigentlichen Objekt. Wird ein Hüllkörper von z.B. einem Strahl nicht geschnitten, so wird auch das Objekt nicht geschnitten. Jedoch wird bei einem Schnitt mit dem Hüllkörper, nicht unbedingt das Objekt geschnitten. In diesem Fall muss eine erneute Schnittberechnung mit dem Objekt ausgeführt werden. Mögliche Hüllkörper sind unter anderem:

- **Axis Aligned Bounding Box (AABB)**: Bildet einen Quader um das Objekt. Axis Aligned Bounding Box zeichnen sich dadurch aus, dass ihre Seiten parallel zum Weltkoordinatensystem sind.
- **Oriented Bounding Boxes (OBB)**: Bilden ebenfalls einen Quader um das Objekt. Orientieren sich jedoch an der Ausrichtung des Objektes und nicht an den Achsen des Weltkoordinatensystems.
- **Kugel**: Die Objekte werden von einer Kugel umschlossen.
- **k-Discretely Oriented Polytopes (k-DOP)**: Erlaubt die Umschließung des Objektes mit k verschiedenen sich paarweise gegenüber liegenden parallelen Linien.

Die einzelnen Körper haben Vor- und Nachteile. Der Schnittpunkt mit einer Kugel ist am einfachsten zu berechnen, sie umschließt jedoch die Körper sehr selten sehr genau, es entsteht viel Freiraum. AABB sind meist besser, OBB noch besser, jedoch ist die Schnittpunktberechnung noch schwieriger. Gleiches gilt für k-DOPs im Vergleich zur OBB[17][32].

3.8.2 Slabs Methode

Die Slabs (deutsch. Kachel) Methode ist ein effizienter Algorithmus um den Schnittpunkt eines Strahls mit einer AABB zu berechnen[16]. Zur besseren Veranschaulichung und mit Rücksicht auf die Verwendung im 2D Raytracer, wird der Algorithmus in 2D erklärt. Man begrenzt die AABB durch zwei sogenannte Kacheln. Diese sind durch parallele Linien zu den AABB definiert. Eine Kachel begrenzt dabei die AABB jeweils in Y-Richtung und eine in X-Richtung. Zuerst werden die Entfernungen tx_{min}, tx_{max}, ty_{min} und ty_{max}

berechnet, bei welchen der Strahl die Kacheln schneidet (3.5). Anschließend werden die gefundenen Entfernungen sortiert:

$$t_{min} = max(tx_{min}, ty_{min}) \tag{3.4}$$
$$t_{max} = min(tx_{max}, ty_{max})$$

Ist nun $tmax < tmin$ so trifft der Strahl das Rechteck nicht (Abb. 3.6).

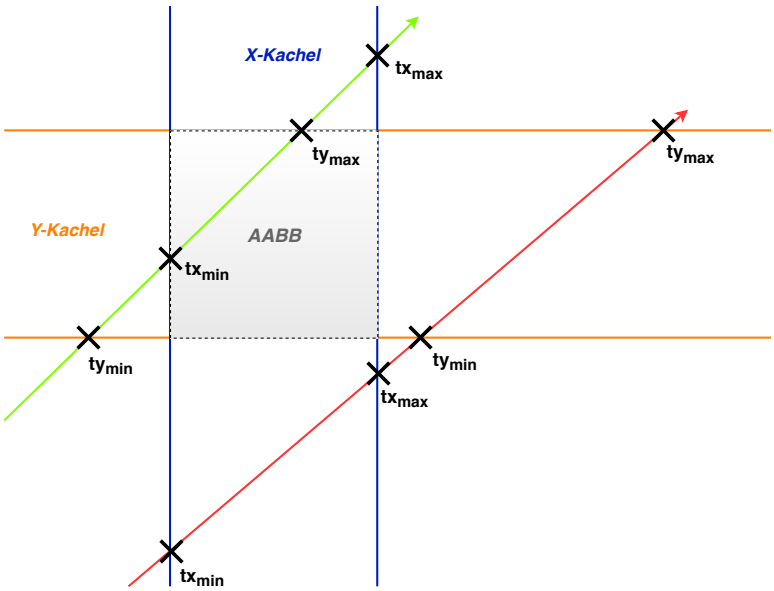


Abbildung 3.5: Slabs Methode - Erster Schritt - zwei verschiedene Strahlen (grün und rot). Der grüne Strahl schneidet die AABB, der rote nicht. Zur besseren Veranschaulichung werden die gefundenen Entfernungen auf den Strahlen dargestellt.

3.8.3 Hüllkörperhierarchie

Bei einer Hüllkörperhierarchie (*engl. Bounding Volume Hierachy, (BVH)*), zu deutsch Hüllkörperhierarchie, werden Hüllkörper hierarchisch in einem Baum angeordnet. In den Blättern des Baumes sitzen die Objekte welche minimal von ihren entsprechenden Hüllkörper umschlossen werden. An der Wurzel des Baumes liegt ein Hüllkörper, welcher alle Objekte erfasst. Die dazwischen liegenden Knoten des Baumes enthalten dann wieder minimal umschlossen die Hüllkörper der nachfolgenden Knoten bzw. der Blätter. Hierdurch kann garantiert werden, dass wenn ein Strahl einen übergeordneten Knoten

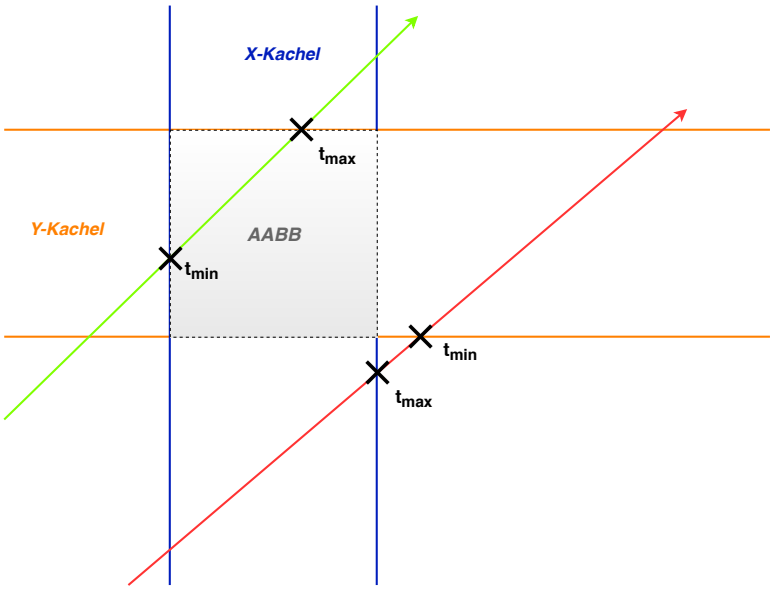


Abbildung 3.6: Slabs Methode - Zweiter Schritt - sortieren der Schnittpunkte

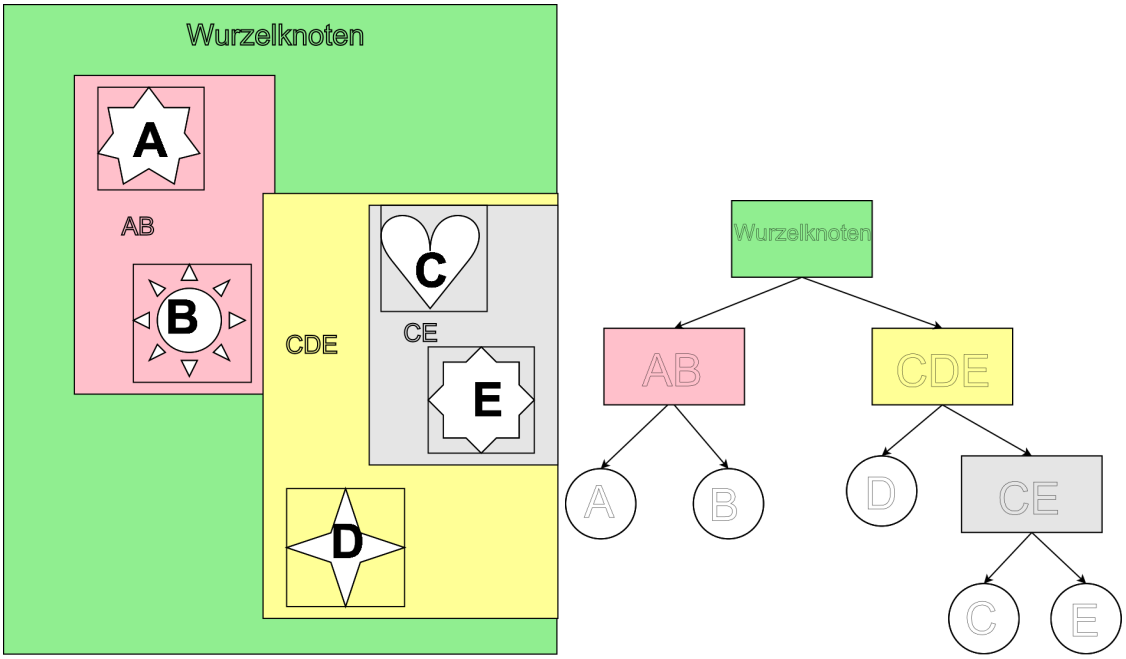
nicht trifft, er auch keinen der untergeordneten treffen kann, und somit auch keine Objekte welche darunter in den Blättern sitzen.

3.8.3.1 Knoten Teilen

Bei BVHs muss immer entschieden werden, wie ein Knoten weiter aufgeteilt werden soll. Die einfachste Methode ist es, die Objekte nach ihrem Mittelpunk zu sortieren und den Teilung so zu wählen, dass die eine Hälfte der Objekte auf der einen Seite ist und die andere Hälfte auf der anderen(*Median Split*[16]). Die Nutzung einer *Surface Area Heuristic* (SAH)[20] kann die Qualität einer BVH erhöhen[28][1]. Ziel einer SAH ist es, möglichst viele Objekte pro Fläche des Knoten zu haben. Dies verringert die Anzahl der unnötig durchgeführte Schnittberechnungen, da sich die Wahrscheinlichkeit, dass ein Strahl den Hüllkörper des Knoten trifft aber nichts darin, verringert. Für die Minimierung dieser Wahrscheinlichkeit werden als Heuristik die Kosten einer Teilung mit folgender Formel bestimmt:

$$c(A,B) = t_{trav} + p_A \sum_{i=0}^{N_A} t_{intrsec}(a_i) + p_B \sum_{i=0}^{N_B} t_{intrsec}(b_i) \tag{3.5}$$

[1] Aus Online Kapitel: Online chapter: Real-Time Ray Tracing, v1.4, 05.11.2018



(a) Hüllkörper um Objekte und Hüllkörper um Hüllkörper

(b) Eine mögliche aus (a) entstandene Hüllkörperhierarchie mit Knoten entsprechend der Boundving Volumes von (a)

Hierbei ist:

- $c(A, B)$ Kosten der frei gewählten Teilung der Kindknoten in A und B.
- t_{trav} Kosten für das Traversieren des zu teilenden Knoten N.
- p_X Wahrscheinlichkeit, dass der potentielle Kindknoten (A oder B) getroffen wird. Sie ist äquivalent Fläche zu F von X durch die Fläche von N. $p_X = \frac{F_X}{F_N}$.
- $t_{intersect}$ Kosten der Schnittprüfung mit einem Objekt im Knoten.
- $\sum_{i=0}^{N_X} t_{intersect}(x_i)$ Summe über die Schnittkosten für alle Objekten in einem Kindknoten.

Nun werden zufällige Teilungen gewählt. Die Teilung mit der geringsten SAH wird verwendet.

3.8.4 Binary Space Partitioning Trees

Binary Space Partitioning Trees (BSP Baum) teilen die Szene mithilfe von Ebenen in Segmente auf. Die Bounding Box um die gesamte Szene wird dabei immer wieder in kleinere Räume aufgeteilt. Dies geschieht, bis eine maximale Tiefe erreicht ist und/oder bis die Anzahl der Kanten/Objekte in den einzelnen Abschnitten unter einer bestimmten Grenze liegt.

3.8.5 kd-Trees

k-dimensional-Trees (kd-Trees) sind ein Spezialfall des BSP Baumes[5]. Hierbei können die Ebenen, welche den Raum unterteilen, immer nur parallel zu einer der Achse des Koordinatensystems gewählt werden. Dies vereinfacht die Konstruktion und das Traversieren des Baumes.

3.8.6 Octrees

Ein Octree kann als weitere Spezialisierung eines kd-Trees gesehen werden, bei dem der Raum immer in 8 gleich große Teile geteilt wird. Das Prinzip wurde bereits beim VXGI Algorithmus erläutert Abschnitt 3.4.

3.8.7 Reguläre Gitter

Reguläre Gitter unterteilen die Szene in gleich große Zellen(Voxel)[9]. Diese bilden dann das reguläre Gitter. Probleme bei dieser Technik sind, das viele leere Zellen entstehen können und dass bei Objekten mit sehr unterschiedlichen Auflösungen der Objekte (zb. Meshes mit sehr hoher Netzdichte und solche mit geringer) die Auflösung des Grids schwer zu wählen ist. Dieses Problem bezeichnet man allgemein als *Teapot in a Stadium Problem*[11]. Der Vorteil ist die sehr einfache Datenstruktur.

4 Problemstellung

In diesem Kapitel wird die Problemstellung der globalen Beleuchtung in 2D dargestellt. Dazu wird das Modell der globalen Beleuchtung in 2D übertragen und dadurch auftretende Änderungen herausgestellt. Danach wird dieses 2D Modell formalisiert.

4.1 Modell der globalen Beleuchtung in 2D

Während manche der in 2.4 beschriebenen Phänomene und Objekte in 2D bis auf eine Reduktion der Dimensionen sich gleich verhalten, ergeben sich für andere aufgrund dessen große Änderungen.

4.1.1 Kamera

Abbildung 4.1 zeigt das Kameramodell der 2D Grafik. Da die Weltkoordinaten nun zweidimensional sind, wird aus dem Frustum in Abbildung 2.6 eine Fläche. Die Kamera U sitzt dabei selbst nicht in der Szene Z, sondern betrachtet sie von oben. Da es keine Tiefeninformationen gibt, ist eine hintere Clippingebene unnötig. Die vordere Clippingebene AB, auf die die Szene abgebildet wird, liegt in der 2D Grafik in der Szenen-Ebene selbst und begrenzt so auch gleichzeitig die Sichtbarkeit von Objekten. Es handelt sich also um eine dreidimensionale Sicht auf eine zweidimensionale Szene. Dies hat zur Folge, dass, im Gegensatz zur 3D Grafik, nicht die Ränder bzw. Oberflächen von Objekten betrachtet werden, sondern das Innere der Objekte sichtbar wird. Da der sichtbare Bereich keine Tiefe mehr besitzt, wird dieser nachfolgend statt als View Frustum als Anzeigebereich bezeichnet. Da sich die komplette Lichtausbreitung in der Szenenebene vollzieht, die Kamera jedoch nicht Teil der Szene ist, trifft auch kein Licht der Szene direkt auf die Kamera. Deswegen kann dieses Licht im Vergleich zum Kameramodell in einer 3D Szene auch nicht zur Bildsynthese genutzt werden.

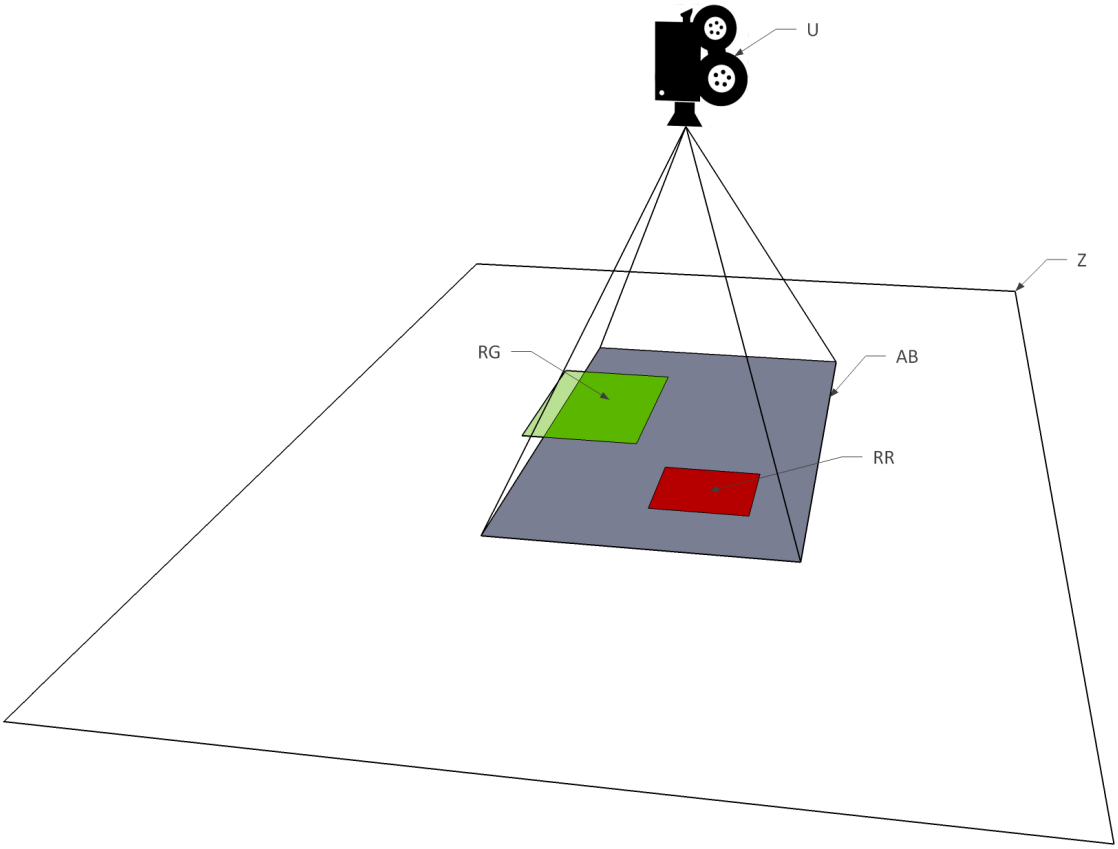


Abbildung 4.1: Kameramodell in der 2D Bildsynthese

4.1.2 Oberflächenstreuung

Die Oberflächen von Objekten sind nicht mehr sichtbar. Hierdurch können evtl. einfachere Modelle zur Beschreibung der Oberflächenstreuung ohne signifikante Qualitätsverluste gewählt werden.

4.1.3 Transmission

Da das Innere der Objekte sichtbar ist, muss für die Transmission der Weg durch das Objekt bestimmt werden, damit diese auch von Innen beleuchtet werden können.

4.2 Darstellung von Licht in 2D

Da die Kamera nicht in der Szene sitzt, trifft auch kein Licht auf sie. Angenommen bei normaler 2D Grafik ohne Beleuchtungsmodell erfasst die Kamera die gesamte Szene einfach mit einem konstanten rein ambienten Licht. D.h., die Kamera erfasst die Lichtausbreitung der gesamten Szene. Dies muss dann auch für die berechnete globale Beleuchtung in 2D gelten. Diese Darstellung des Lichtes muss in einer Form vorliegen, in der sie auf die vorhandene Szene angewendet werden kann. In der 2D Grafik sind das normalerweise Rastergrafiken. Daraus folgt, dass das Licht auch in Form einer Rastergrafik vorliegen muss. Diesen Prozess nennt man Rasterung [28].

4.3 Formale Beschreibung des Lichttransportes in 2D

Aufgrund der Änderungen in der globalen Beleuchtung wird sich auch die Rendergleichung ändern. Diese wird als Grundlage für die Entwicklung des 2D Raytracing Algorithmus benötigt.

4.3.1 Rendergleichung

Das grundlegende Problem der globalen Beleuchtung bleibt bei der Übertragung in 2D erst einmal gleich, und zwar die Reflexion des Lichtes an verschiedenen Oberflächen zu bestimmen. Die Rendergleichung mit Transmission aus 2.5 muss auch nur geringfügig angepasst werden:

$$L_r(x, \omega_o) = \int_C f'(x, \omega_o, \omega_i) L_i(x, \omega_i) \cos\theta_i d\omega_i \tag{4.1}$$

Die Sphäre S um den Punk x wird durch den Kreis C um Punk x ersetzt. Die Richtungen ω_o und ω_i, der Punk x und der Winkel θ_i verlieren jeweils eine Dimension.

4.3.2 Lichttransport zur Kamera

Wie in Abschnitt 4.1.1 erläutert, funktioniert der Lichttransport zur Kamera durch Gleichung (4.1) nicht. Um die Beleuchtung in die Form einer Rastergrafik zu bringen,

muss bestimmt werden, wie viel Strahlungsfluss durch die Pixel geht. Der Strahlungsfluss im Inneren des Pixels hängt davon ab, wie viel Licht in den Pixel P einfällt (Abb. 4.2).

Um den Strahlungsfluss durch P zu bestimmen, kann man den Pixel P einfach als quadratisches Objekt betrachten. Um nicht berücksichtigen zu müssen, wie das Licht den Pixel verlässt oder in ihm absorbiert wird, betrachtet man nur das in den Pixel einfallende Licht. Dies kann mithilfe einer modifizierten Gleichung (4.1) geschehen. Da man nur das einfallende Licht behandeln möchte, ändert man den Kreis C zum bedeckenden Halbkreis hC über den Oberflächenpunkten X_P. Da das Licht hier nicht reflektiert wird, wird die BSDF $f'(x, \omega_o, \omega_i)$ und das Lambertsche Gesetz $\cos\theta_i d\omega_i$ nicht benötigt. Dies führt man nun einfach für die Menge X_P der Punkte der Oberfläche des Pixel P durch.

$$L_f(P) = \sum_{k=0}^{|X_P|} \int_{hC} L_i(x_k, \omega_i) t(P, x_k, \omega_i) d\omega_i \tag{4.2}$$

Die Funktion $t(P, x_k, \omega_i)$ gewichtet das Licht aus dem Winkel ω_i hierbei nach der Strecke, welche es in P zurücklegt. Dies ist wichtig, da ein Strahl welcher mehr Strecke in einem Pixel zurücklegt, auch einen größeren Anteil am Strahlungsfluss in diesem Pixel hat (Abb. 4.2).

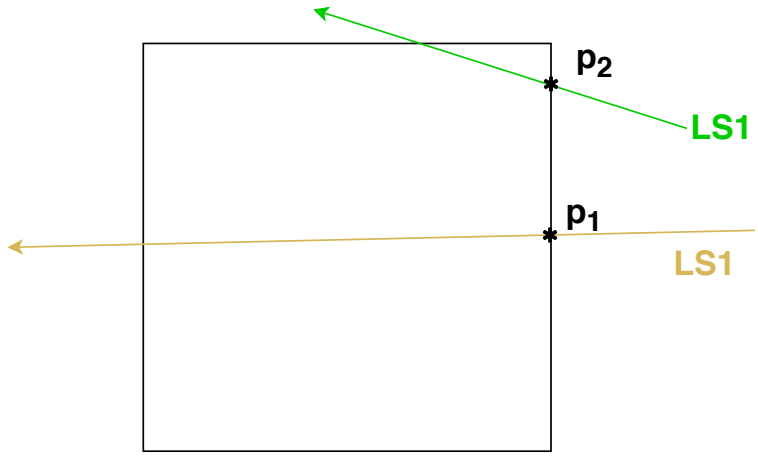


Abbildung 4.2: Pixel Strahlungsfluss: Der Strahlungsfluss im inneren ist abhängig von den Punkten p_x. Der zurückgelegte Weg im Pixel muss berücksichtigt werden, da z.B. LS1 mehr Einfluss auf den Strahlungsfluss hat als LS2

5 Konzept

Nachfolgend werden Lösungsansätze basierend auf den in Kapitel 3 vorgestellten Methoden für die im vorhergegangenen Kapitel erfassten Probleme vorgestellt. Danach werden die Anforderungen für eine Implementierung dieser Lösungsansätze formuliert.

5.1 Algorithmus für 2D globale Beleuchtung

Da wie in Abschnitt 4.3.2 beschrieben das Licht am Ende in einer gerasterten Form vorliegen muss, wäre das Nächstliegende einen volumetrischen Ansatz wie VXGI oder LPV zu wählen. Diese haben zwar den Vorteil, dass sie das Licht gleich in der richtigen Form berechnen, allerdings haben sie den Nachteil nicht erwartungstreu oder konsistent zu sein und damit nicht alle Lichtpfade darstellen zu können. Da der Algorithmus aber das Problem der globalen Beleuchtung vollständig lösen soll, darf dies nicht der Fall sein. Diese Eigenschaften werden durch die Strahlverfolgungsalgorithmen in Abschnitt 3.6 erfüllt.

Da die Kamera nicht in der Szenenebene sitzt, ist eine Verwendung von *Hybriden* oder *Kamera zur Lichtquelle* basierten Algorithmen nicht möglich. Auch reine *Lichtquelle zur Kamera* basierte Algorithmen gestalten sich schwierig, da der Algorithmus sowohl Gleichung (4.1) als auch Gleichung (4.2) lösen müsste, welche verschiedene Probleme beschreiben. Es ist also logisch, den Algorithmus in zwei Phasen aufzuteilen. In der ersten Phase wird Gleichung (4.1) angenähert. Dafür wird ein *Particle Tracing* basierte Algorithmus gewählt, welcher die Lichtausbreitung in der Szene bestimmt. Die Lichtausbreitung wird in Form von Pfaden der Strahlen in der Szene gespeichert.

In der zweiten Phase wird der Lichttransport zur Kamera bestimmt. Die Pfade werden in einen Strahlungsfluss pro Pixel umgewandelt. Dazu wird der Strahlungsfluss aller Pfade für jeden Pixel nach Gleichung (4.2) bestimmt. Effektiv handelt es sich dabei um eine Rasterung und Aufsummierung der Pfade.

5.2 Objekte

Für die Berechnung der globalen Beleuchtung werden Polygone als Objekte verwendet. Da diese nur gerade Kanten haben können, wäre die Darstellung von gerundeten Objekten unmöglich. Dieser Effekt wird durch die Verwendung von gewichteten Punktnormalen abgemildert.

5.2.1 Oberflächenmodell

Da die eigentliche Oberfläche der Objekte nicht sichtbar ist, ist, mit Blick auf die Echtzeitfähigkeit, eine Verwendung komplexer Reflexionsfunktionen überflüssig. Stattdessen wird eine vereinfachte BSDF verwendet. Pro Objekt wird hierbei eine Vereinfachte BSDF zugewiesen. Die BSDF besteht dabei aus BRDF und BTDF. Sowohl BTDF und BRDF werden einfach gehalten. Die BTDF bestimmt nur die Transmissionstiefe. Die BRDF ist in der Lage, diffuse und gerichtete Reflexionen sowie deren Mischformen darzustellen. Die BRDF wird im Raytracer mit Hilfe von MCI und RR gesampelt.

5.3 Beschleunigungsdatenstruktur

Als Beschleunigungsdatenstruktur dient eine Hüllkörperhierarchie, da diese für die Szene im 2D Raytracing im Vergleich zu den anderen Ansätzen die bessere Laufzeiten bieten sollte [31]. Zur Laufzeitverbesserung der BVH wird eine *Surface Area Heuristic* verwendet. Als gute Mischung aus Leistung und Genauigkeit werden Axis Aligned Bounding Box (AABB) als Hüllkörper für die BVH gewählt. Der Schnitttest wird mit Hilfe der *Slabs Methode* durchgeführt.

5.4 Lichtquellen

Die Szene benötigt initiale Lichtquellen welche das Licht in die Szene senden. Diese sollen die Eigenschaften aus Abschnitt 2.4.2 erfüllen. Dies kann dadurch erreicht werden, dass von den Lichtquellen initial Strahlen entsprechend der Eigenschaften in die Szene platziert werden, wie in Abb. 2.7) gezeigt.

5.5 Bildsynthese

Da Ziel dieser Arbeit die Berechnung der Beleuchtung in Echtzeit ist, wird auf eine volle Rendering Pipeline verzichtet. Dargestellt wird lediglich das berechnete Licht.

5.5.1 Lightmap

Die gerasterten Pfade werden in einer Textur namens *Lightmap* gespeichert. Die Lightmap hat die Auflösung des Anzeigebereiches und enthält Pixel mit den Werten von 0 bis 1. Diese geben jeweils die Beleuchtungsstärke der einzelnen Pixel der Szene an.

5.5.2 Rasterung

Die Rasterung der im Raytracing bestimmten Pfade in die Lightmap soll auf der Grafikkarte mit Hilfe von OpenGL geschehen. Dies hat den Grund, dass die GPU auf das Rastern ausgelegt ist und es hardwareseitig unterstützt. Dementsprechend läuft dies schneller als auf der CPU.

5.6 Anforderungen

Die Szenen und Beschleunigungsdatenstruktur werden vor der Raytracing Phase statisch initiiert. Der Algorithmus berechnet daraufhin einmalig die Lightmap der Szene. Dieser Schritt soll in Echtzeit möglich sein. Als Echtzeitgrenze werden 25 FPS (40 m s) bei der häufig verwendeten Auflösung von FullHD ($1920 * 1080$ Pixel)[1] festgelegt.

Objekte werden zur besseren Wiederverwendbarkeit aus einer Datei geladen. Zum Laden der Objekte, Debuggen der Szenen und des Raytracers wird eine Kontrollsoftware für den Raytracer mit grafischer Oberfläche erstellt. Diese zeigt BVH, Objekte und Normale sowie die Strahlen in vereinfachter Form an.

Um Objekte leichter erstellen zu können, wird eine zusätzliche Software geschrieben, welche Polygone aus Rastergrafiken generiert (Anhang A.5).

[1] *https://de.trendcounter.com/research/*, aufgerufen 14.08.2019

Zum Testen des Systems werden mehrere Testszenen erstellt, um unterschiedliche Eigenschaften des Systems zu testen.

5.7 Software, Bibliotheken und Werkzeuge

Für die Erstellung des Raytracers und der Kontrollsoftware wird als IDE *QtCreator 4.8.2* mit *Qt 5.12.0* für die UI verwendet. Als Compiler wird der bundeled *MinGW-W64-4.3.4* von Qt mit *gcc 7.2.0* verwendet[2]. C++ wird in der version C++14 benutzt. Fußnoten zum C++ standard beziehen sich stets auf C++14. Für das Einlesen und Exportieren von CSV Dateien werden *Fast C++ CSV Parser*[3] und *CSVWriter*[4] verwendet.

Für die Kommunikation mit der Grafikkarte und das Rastern wird *OpenGL 4.6*[5] mit *Glew 2.1.0*[6]verwendet. Der OpenGl Kontext wird mit *GLFW 3.3*[7] erstellt.

Für die Umwandlung von Rastergrafiken in Objekte wird *OpenCV V. 4.0.0-alpha*[8] verwendet. Das Programm wurde mit der IDE *VisualStudio Community 2017 15.5.2* mit *Windows SDK 10.0.16299.0* entwickelt. Als compiler dient *Microsoft Visual C++ 2017* [9]

Zum manuellen Erstellen der gerenderten Bilder wird *Gimp 2.8.18*[10] verwendet.

[2] *https://download.qt.io/official_releases/qtcreator/4.8/4.8.2/*, abgerufen am 05.07.2019
[3] *https://github.com/ben-strasser/fast-cpp-csv-parser*, abgerufen am 05.07.2019
[4] *https://github.com/al-eax/CSVWriter*, abgerufen am 05.07.2019
[5] *https://www.opengl.org/*, abgerufen am 05.07.2019
[6] *http://glew.sourceforge.net/*, abgerufen am 05.07.2019
[7] *https://www.glfw.org/*, abgerufen am 05.07.2019
[8] *https://opencv.org/opencv-4-0-alpha/*, abgerufen am 05.07.2019
[9] *https://docs.microsoft.com/de-de/visualstudio/releasenotes/vs2017-relnotes-v15.5*, abgerufen am 05.07.2019
[10] *https://download.gimp.org/mirror/pub/gimp/v2.8/windows/*, abgerufen am 05.07.2019

6 Umsetzung

Hier wird die Umsetzung des Konzeptes erläutert und es werden wichtige Implementierungsdetails aufgezeigt.

Eine vollständige Dokumentation der Software sowie eine Anleitung zur Nutzung findet sich auf der beigefügten CD.

6.1 Architektur

Die Software besteht aus 3 Hauptkomponenten. Der Kontrollsoftware, dem Raytracer und der Rasterung. In der Kontrollsoftware wird die Szene definiert und vereinfacht dargestellt. Das Raytracer bestimmt die Lichtausbreitung in der Szene. Die Rasterung erzeugt die Lightmap und zeigt diese an. Die Kommunikation zwischen den Komponenten erfolgt über einen statisch angelegten Strahlbuffer und einen statisch angelegten Pfadbuffer, welche als Referenz übergeben werden. Dies spart das Kopieren der Buffer zwischen den Komponenten, was für die Laufzeit von Vorteil ist.

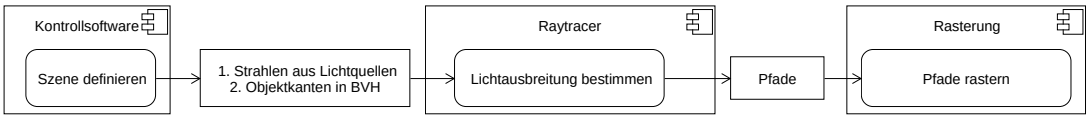


Abbildung 6.1: Architektur der Software. Komponenten in Verbindung mit Aktivitätsdiagramm der Komponenten.

6.2 Szene definieren

Die Szene definiert die Lichtquellen und die Objekte in der Szene.

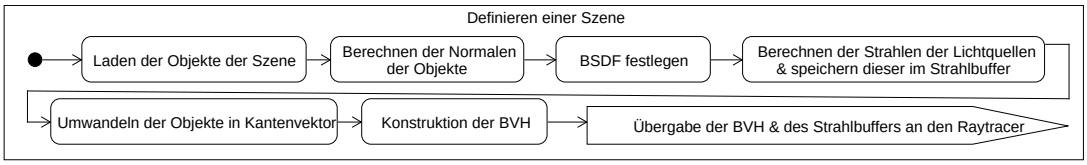


Abbildung 6.2: Ablauf der Definition einer Szene

6.2.1 Objekte

Objekte bestehen aus einem Polygon und Kanten- und Punktnormalen, sowie einer BSDF. Sie stellen außerdem eine Funktion zur Translation des Objektes mit Hilfe eines Vektors bereit. Die Polygone werden aus einer CSV Datei geladen, welche die Punkte des Polygons enthält. Für die Erstellung von Polygonen aus Rastergrafiken wurde ein extra Programm geschrieben (Anhang A.5).

6.2.1.1 Berechnung der Normalen

Die Oberflächennormale werden mit Gleichung (2.1) berechnet. Die Punktnormalen werden mit der normierten Summe der Kantennormalen gebildet. Zwischen den Punktnormalen einer Kante wird linear interpoliert (Abb. 6.3). Um weiterhin auch harte Kanten darstellen zu können, gibt es zwei Punktnormalen pro Punkt.

6.2.1.2 BSDF

Pro Objekt wird eine BSDF verwendet. Die BSDF besteht aus einer BRDF und einer BTDF als Delegates. Sie selbst entscheidet nur, wie viel Licht jeweils die BTDF und die BRDF erhalten. Zuerst wird die Transmission mit der BTDF bestimmt, dann die Reflexion mit der BRDF. Die BSDF wird folgender Maßen parametrisiert:

- **Transmissionsgrad (t):** Prozentangabe des Lichtes, welches von der BTDF behandelt wird. Das übrige wird von der BRDF behandelt.
- **Undurchsichtigkeit (opaqueness o):** Gibt an, wie leicht das Licht in das Objekt eindringen kann.
- **Streuung (diffuse d):** Faktor, wie stark die BRDF streut

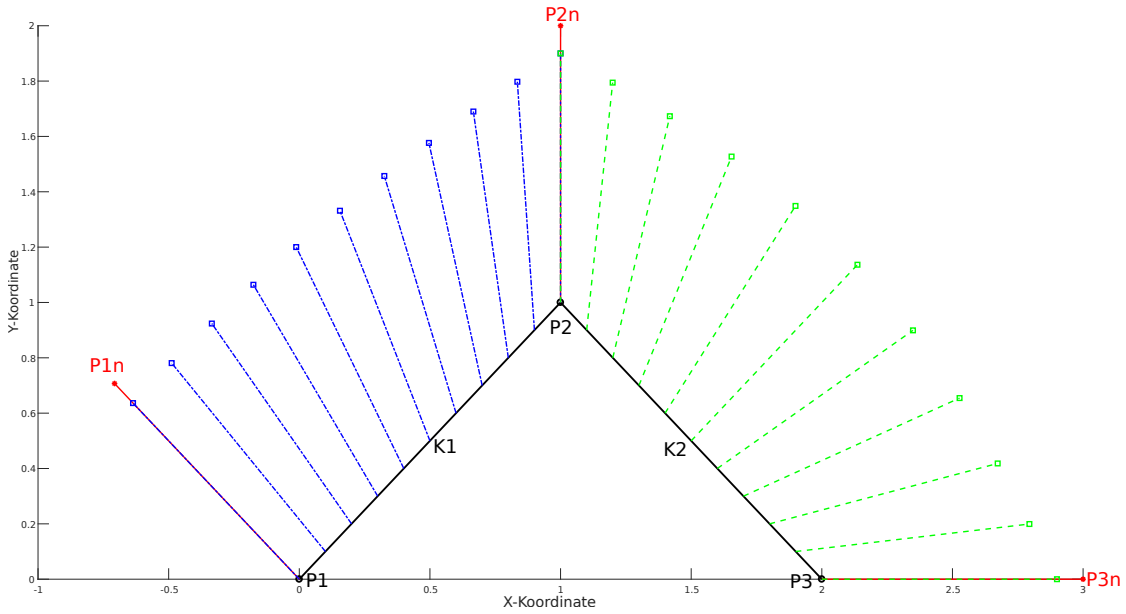


Abbildung 6.3: Interpolation der Punktnormalen P1n bis P2n (rot) der Punkte P1 bis P3 eines Ausschnitts eines Polygons. Die interpolierten Normalen der Kanten K1 sind blau dargestellt und die von K2 grün

- **Absorption (absorption a):** Prozentangabe, wie viel Licht die BRDF absorbiert und wie viel sie reflektiert.

6.2.1.2.1 BRDF Die BRDF verwendet eine einfache Funktion, welche immer $1 - a$ oder 0 ist, abhängig von d.

$$f(\omega_o, \omega_i) = \begin{cases} 0 & \omega_o > r(\omega_i) + d_h \\ 0 & \omega_o < r(\omega_i) - d_h \\ 1 - a & \text{sonst} \end{cases} \tag{6.1}$$

Da die BRDF für das ganze Objekt gleich ist, kann die Position auf dem Objekt und damit der Punkt x ignoriert werden. $r(\omega_i)$ beschreibt den Ausfallwinkel zu ω_i nach Gleichung (2.2). d_h wird durch den Term $d_h = (d * \pi)/2$ bestimmt.

Ein Problem kann sein, das $r(\omega_i) \pm d_h$ weiter als $pi/2$ von der Kantennormalen entfernt liegt (also in das Objekt zeigt, Abb. 6.6). Um dies zu verhindern, wird $r(\omega_i)$ auf den Winkel limitiert, bei dem dies nicht auftritt.

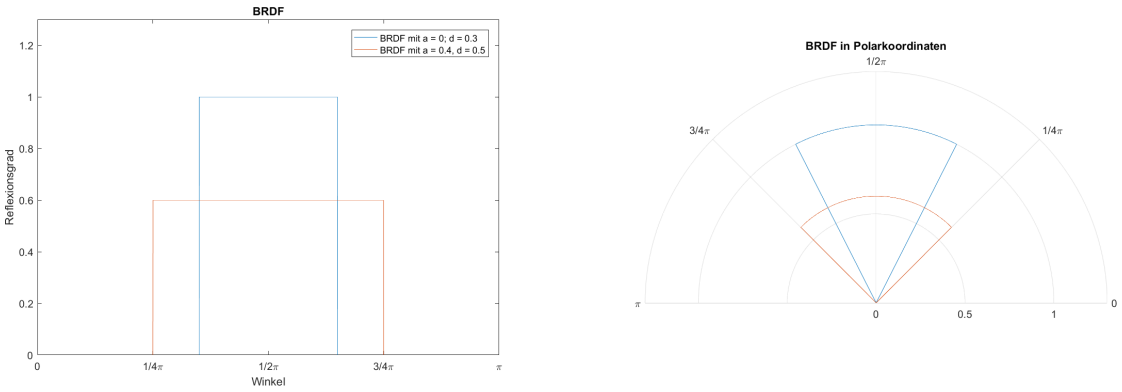


Abbildung 6.4: Zwei BRDF nach Gleichung (6.1) mit $\omega_i = \pi$

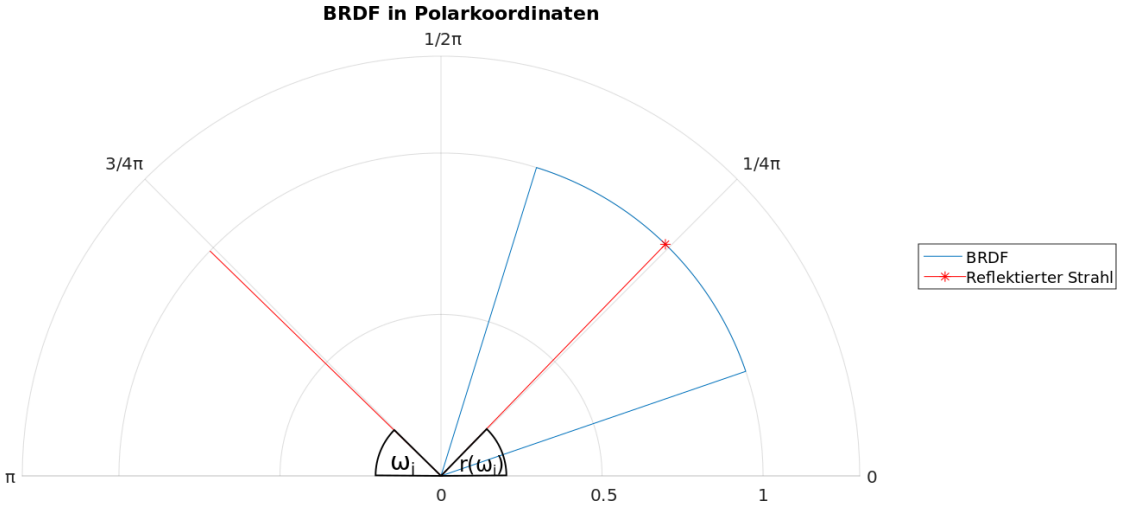


Abbildung 6.5: BRDF, welche die Richtungsabhängigkeit der BRDF von $\omega_i = \pi$ zeigt

6.2.1.2.2 BSDF Die BSDF behält die Richtung des Strahls bei und bestimmt lediglich die Transmission (Abschnitt 6.4.2.2).

6.2.2 Erstellen des Kantenvektors

Für den späteren Raytracing Schritt und das Erstellen der BVH, ist es sowohl von der Geschwindigkeit als auch von der Verarbeitung der Datenstruktur her einfacher, wenn alle Kanten und Normalen in der Szene in einem Vektor vorliegen statt in den einzelnen Objekten. Dafür werden alle Eigenschaften der Kante in einem `Edge` Objekt zusammengefasst. Dies wird für alle Objekte gemacht und die `Edges` dann im Kantenvektor gespeichert.

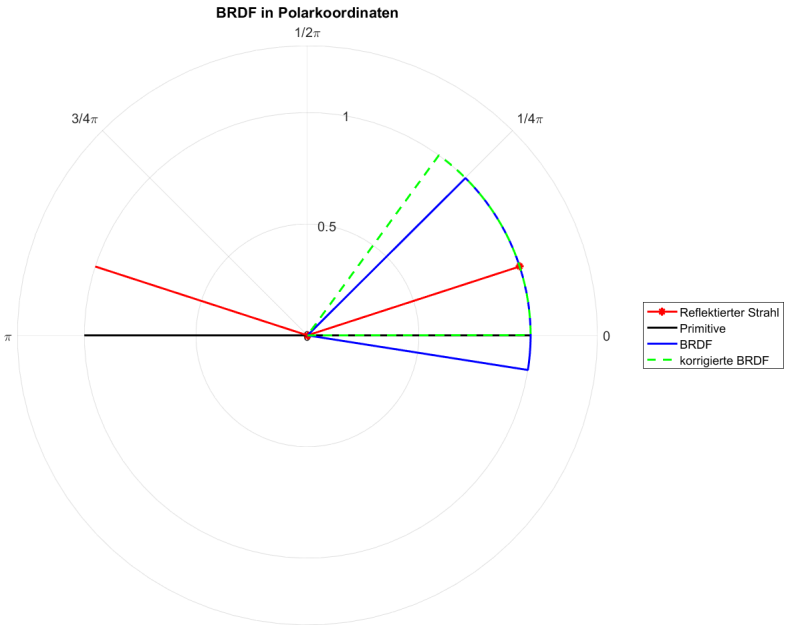


Abbildung 6.6: $r(\omega_i) + d_h$ liegt unter der Primitiven, um dies zu korrigieren, wird die BRDF gedreht, bis dies nicht mehr der Fall ist

Edge
-Vec2Df p1 //Punkt 1 -Vec2Df p2 //Punkt 2 -Vec2Df n1 //Punktnormale 1 -Vec2Df n2 //Punktnormale 2 -Vec2Df ns //Kantennormale -Vec2Df centroid //Mittelpunkt -Object* obj //Objekt von dem die Kante stammt

Abbildung 6.7: Edge Klasse

6.2.3 Lichtquellen

Lichtquellen generieren Strahlen, welche dann frei in der Szene platziert werden. Jede Lichtquelle leitet sich von einer gemeinsamen Basisklasse `Lightsource` ab. Die enthält nur die Position und die Gesamthelligkeit für die Lichtquelle sowie eine Liste von Strahlen, welche in der Szene platziert werden sollen. Wie diese Liste gefüllt wird, obliegt alleine den Subklassen von `Lightsource`. Es sind eine Punktlichtquelle und eine Stablichtquelle implementiert. Ein Beispiel wie diese Strahlen platziert werden findet sich in Anhang A.4.

6.3 BVH

6.3.1 Konstruktion

Die Anzahl der der `Edges` p_a in den Blattknoten ist festgelegt auf $p_a < n_{max}$ (Abschnitt 7.3.2). `Edges` sind jeweils höchstens in einem Knoten enthalten. Die Knoten enthalten statt den `Edges` selbst nur Indices auf den Kantenvektor, um Speicheroverhead zu vermeiden. Die BVH ist ein binärer Baum, dies erleichtert das Traversieren, da ein Knoten immer zwei Nachfolger hat oder keinen. Die BVH enthält einen Vektor, welcher alle Knoten enthält. Die Konstruktion der BVH aus den `Edges` erfolgt rekursiv. Zuerst wird Algorithmus 1 mit dem ersten und dem letzten Index des Kantenvektors aufgerufen. Der erste Knoten bildet den Wurzelknoten. Die AABB eines Knoten wird gebildet, indem die minimale AABB um die Punkte der `Edges` eines Knoten gebildet wird.

Algorithmus 1 : BVH_Konstruieren

Data : unterer Index, Oberer Index

Result : Index des aktuellen Knoten

```
1  SAH mit Indices initialisieren;
2  Knoten mit Indices initialisieren;
3  Größe der AABB des Knoten bestimmen;
4  if p_a < n_max then // Prüfung, ob Blattknoten
5  |  Knoten speichern;
6  |  return Knoten Index;
7  end
8  Index Teilung mit SAH bestimmen;
9  linker Kindknoten ← BVH_Konstruieren(unterer Index, Index Teilung);
10 rechter Kindknoten ← BVH_Konstruieren(Index Teilung+1, oberer Index);
11 Knoten speichern;
12 return Knoten Index;
```

6.3.1.1 SAH

Die Gleichung (3.5) wird vereinfacht. Durch die feste Anzahl der `Edges` im Knoten muss in der SAH nicht mehr berücksichtigt werden, ob eine Teilung des Knoten überhaupt Sinn ergibt. Dadurch fällt t_{trav} weg, da das Verhältnis zwischen t_{trav} und $t_{intrsec}$ nicht

mehr beachtet werden muss. $t_{intrsec}$ tritt dann nur noch als Gesamtfaktor in den Kosten auf und kann deshalb ebenfalls entfernt werden.

$$c(A,B) = p_A \sum_{i=0}^{N_A} a_i + p_B \sum_{i=0}^{N_B} b_i \tag{6.2}$$

p_X wird im Gegensatz zu Gleichung (3.5) nicht mit der Fläche der AABB berechnet, sondern mit dem dem Umfang (Abb. 6.8).

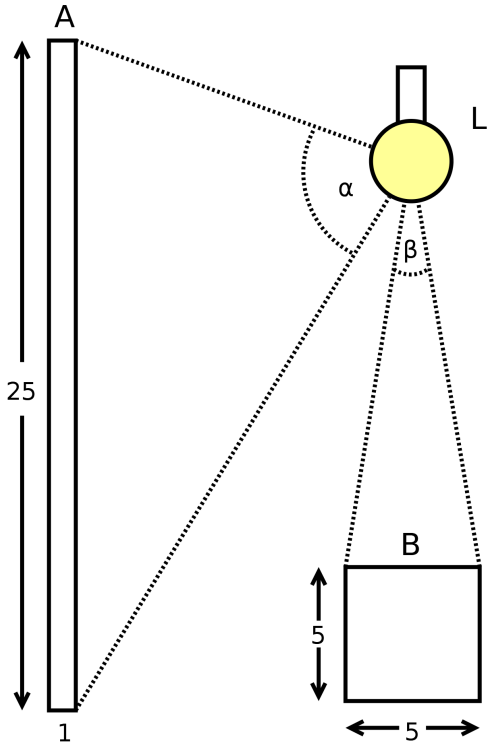


Abbildung 6.8: SAH - Verwendung des Umfanges. Zu erkennen ist, das die Fläche als Heuristik für die Wahrscheinlichkeit bei der SAH in 2D nicht geeignet ist. Die Knoten A und B haben zwar die gleiche Fläche (25), jedoch hat A den viel größeren Umfang ($Umfang_A = 2 * (25 + 1) = 52$, $Umfang_B = 2 * (5 + 5) = 20$. Dies spiegelt sich auch in dem Winkel wieder, in denen die Knoten von einem Strahl von der Lichtquelle L getroffen werden könnten. Da $\beta < \alpha$, ist auch die Wahrscheinlichkeit, B zu treffen, geringer.

Da die BVH statisch vor dem Raytracing berechnet wird, wird jede mögliche Teilung des Knoten entlang der X- und Y-Achse geprüft. Dafür werden die Kanten entlang ihrer

Mittelpunkte sortiert und dann jeder mögliche Index für die Teilung getestet.

Algorithmus 2 : SAH

Data : unterer Index, oberer Index

Result : Index Teilung

```
1  entlang aktueller Achse sortieren;
2  for alle Indecies zwischen unterer und oberer Index do
3  |  AABB U für Edges unterer Index bis aktueller Index;
4  |  AABB V für Edges akteuller Index bis oberer Index;
5  |  kosten ← c(U,V);
6  |  if kosten < kosten Index Teilung then
7  |  |  Index Teilung ← aktueller Index;
8  |  end
9  end
10 return Index Teilung
```

6.4 Raytracing

Der Raytracing Algorithmus verfolgt die Strahlen durch die Szene und wandelt diese in Pfade für die Rasterung um. Der Raytracer nimmt BVH und initiale Strahlen der Lichtquellen im Strahlbuffer aus der Szene entgegen.

Algorithmus 3 : Raytracer Algorithmus

Result : gefüllter Pfadbuffer

```
1  while Strahlbuffer hat noch Strahlen do // wird nebenläufig ausgeführt
2  |  nächsten Strahl aus Strahlbuffer hohlen;
3  |  Strahlverfolgung(Strahl);
4  end
5  return Index Teilung
```

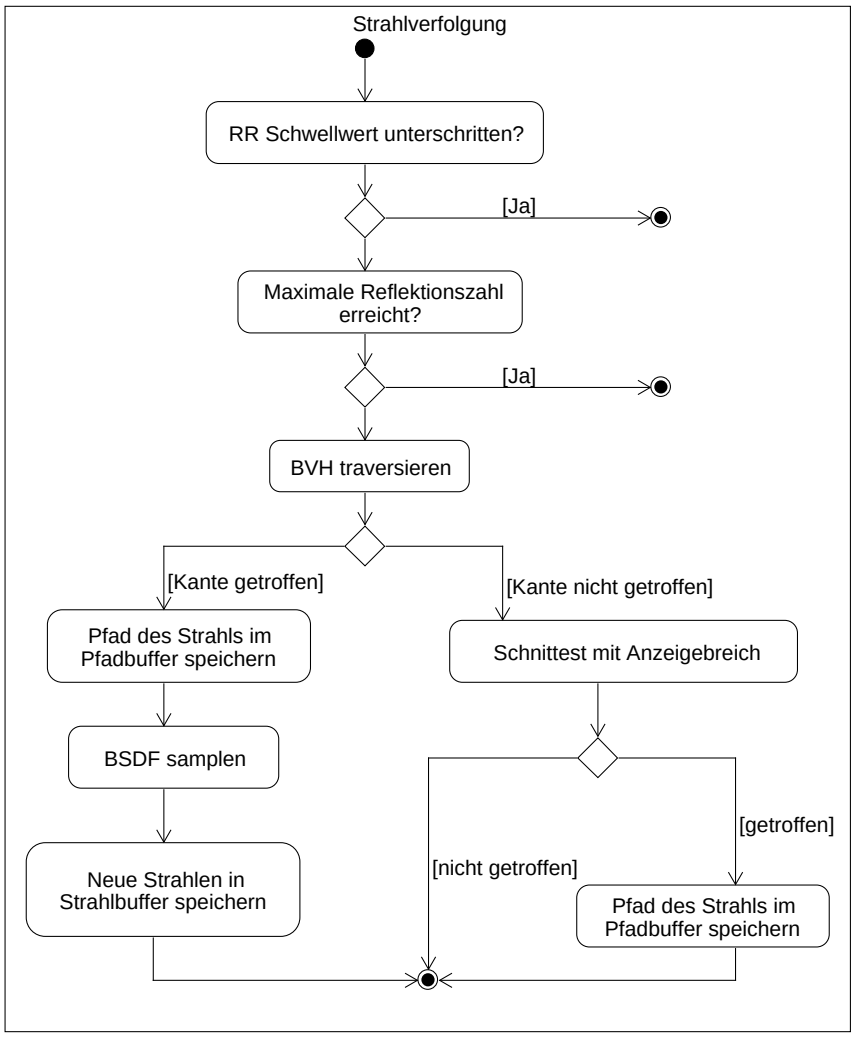


Abbildung 6.9: Ablauf der Strahlverfolgung eines einzelnen Strahls

6.4.1 Schnittpunkte bestimmen

Mithilfe der Schnittpunkte der Strahlen mit der Szene werden die Pfade berechnet. Der Pfad geht dabei immer vom Ursprungpunkt des Strahls zum Schnittpunkt. Der Schnittpunkt mit einer Kante wird mithilfe von Gleichung (2.3) durchgeführt.

6.4.1.1 AABB

Für den Schnittest mit einer AABB wird mithilfe einer *Verzweigungsfreien Slabs Methode*[34] nach Abschnitt 3.8.2 durchgeführt. Besonderheit hier ist, dass sowohl t_{min} als auch t_{max} zurückgegeben werden und t_{min} auf 0 gesetzt wird, wenn der Strahl Ursprung r_o in der AABB liegt.

6.4.1.2 BVH Traversieren

Die BVH wird mit einem Stack basierten Algorithmus traversiert(Algorithmus 4). Es werden die Schnittpunkte mit den Kindknoten des Aktuellen Knoten gebildet. Der dichtere wird zum aktuellen Knoten, der andere wird im Stack gespeichert. Wurde ein Schnittpunkt mit einer Kante gefunden, so muss noch kontrolliert werden, ob der Stack noch einen Knoten enthält, welcher einen dichteren Schnittpunkt hat, da dieser auch dichtere Kanten enthalten könnte. Das Gleiche gilt, wenn keine Kante getroffen wurde, dann müssen die Knoten im Stack ebenfalls überprüft werden. Von dort beginnt der Algorithmus dann erneut. Der Algorithmus beginnt initial mit dem Wurzelknoten der

BVH.

Algorithmus 4 : BVH Traversieren

Data : Strahl

Result : dichtester Schittpunkt und die Kante die geschnitten wurde

```
1  nächster Knoten ← Wurzelknoten BVH;
2  dichtester Treffer gefunden ← false;
3  while ¬dichtester Treffer gefunden do
4      Knoten ← nächster Knoten;
5      if ¬Blattknoten then
6          Schnittpunkt mit den beiden Kindknoten;
7          if getroffen then
8              nächster Knoten ← der dichtere Kindknoten;
               // nur wenn beide Kindknoten getroffen
9              Stack ← der andere Knoten mit dem Schnittpunkt;
10             continue;
11         end
12     else
13         for jede Kante im Knoten do
14             Schnittpunkt bestimmen;
15             if Schittpunkt entfernung < dichtester Schnittpunkt then
16                 dichtester Schittpunkt = Schittpunkt;
17             end
18         end
19     end
20     dichtester Treffer gefunden ← true;
21     if Gibt es einen dichteren Knoten auf dem Stack then
22         nächster Knoten ← Knoten vom Stack;
23         dichtester Treffer gefunden ← false;
24     end
25 end
26 return Schnittpunkt und Kante die geschnitten wurde
```

6.4.1.3 Anzeigebereich

Definiert den sichtbaren Bereich der Szene wie in Abb. 4.1 dargestellt. Für alle Strahlen, welche in der BVH keine Kanten treffen, werden zunächst keine Pfade berechnet. Zum Bestimmen der gesamten Lichtausbreitung in der Szene müssen aber auch diese Pfade bestimmt werden. Da der Anzeigebereich rechteckig ist, wird er als AABB implementiert.

1. der Strahl hat seinen Ursprung außerhalb des Anzeigebereich
2. der Strahl hat seinen Ursprung innerhalb des Anzeigebereich

Der Pfad (von Punkt $p1$ zu $p2$) eines Strahls berechnet sich dann entsprechend $r_o + t_{min} * r_d = p1$ und $r_o + t_{max} * r_d = p2$. Dadurch, dass im inneren der AABB $t_{min} = 0$ ist müssen die oben beschriebenen Fälle nicht unterschieden werden.

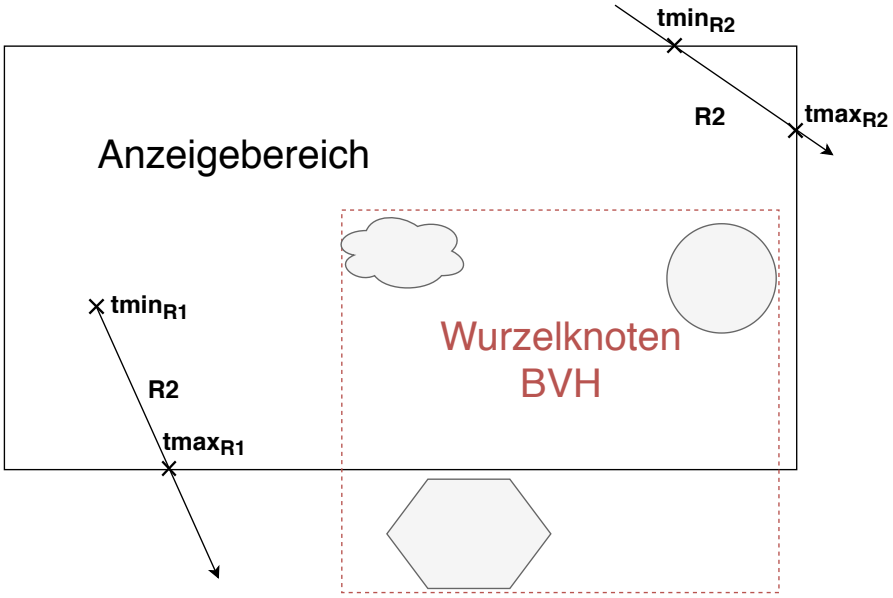


Abbildung 6.10: Schnittberechnung mit dem Anzeigebereich. Grau dargestellt sind Objekte. R1 und R2 sind Sonderfälle von Rays, bei denen die Pfade nicht mit Hilfe der Kollision mit Objekten berechnet werden können. R1 hat den Ursprung im Anzeigebereich, R2 außerhalb

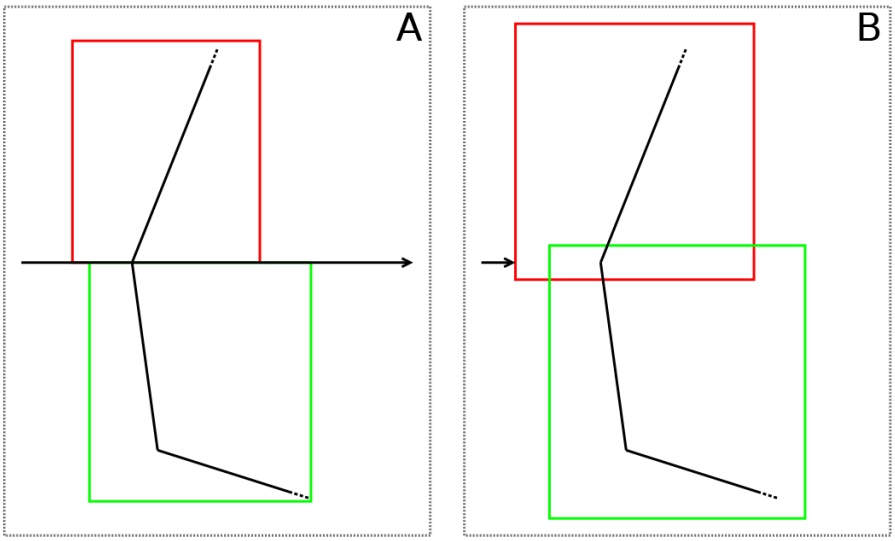


Abbildung 6.11: AABB Vergrößerung: Während bei (A) der Strahl zwischen den AABB hindurchdringt, ist dies bei (B) nicht der Fall

6.4.1.4 Beheben von Gleitkommafehlern

Gleitkommazahlen sind keine exakte Darstellung der reellen Zahlen $\mathbb{R}$[1]. Somit kommt es bei der Berechnung von Schnittpunkten zu Rundungsfehlern.

6.4.1.4.1 AABB Bei parallelen und fast parallelen Strahlen zu einer AABB Kante kann der Effekt auftreten, dass der Strahl die AABB nicht trifft. Dies führt zu Fehlern, wenn zwei AABB genau an zwei aufeinander folgenden `Edges` eine Seite haben, da der Strahl zwischen den AABB hindurchdringen kann. Dies wird verhindert, indem die AABB minimal, abhängig von der Gleitkommagenauigkeit, vergrößert werden (Abb. 6.11).

6.4.1.4.2 Kante Durch Rundungsfehler bei der Schnittpunktberechnung kann es passieren, dass der Schnittpunkt auf der falschen Seite der Kante liegt. Das Skalarprodukt v_p zweier Vektoren, die gleich orientiert sind, ist immer > 0 [1]. Bei der Berechnung der Reflexion kann so geprüft werden, ob der Strahl aus dem Inneren des Objektes stammt (Abb. 6.12). Ist v_p also > 0, so hat vorher ein Rundungsfehler stattgefunden. Statt nun die Reflexion zu berechnen, wird die Strahlrichtung einfach beibehalten.

[1] *http://www.mvps.org/DirectX/articles/math/dot/index.htm*, abgerufen am 21.07.19

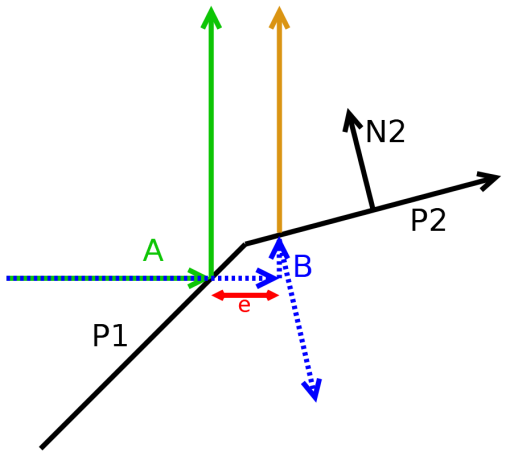


Abbildung 6.12: Gleitkomma Fehlerkorrektur bei Kanten mithilfe des Skalarproduktes. e ist die Abweichung durch Gleitkommafehler vom eigentlichen Treffer. Grün (A) ist der Korrekte Pfad, gelb der korrigierte, blau der falsche (B), schwarz Kanten und Normale

6.4.2 BSDF Samplen

Durch das Samplen der BSDF werden die reflektierten Strahlen und die innere Beleuchtung eines Objektes bestimmt. Zuerst wird die gerichtete Reflexion des Strahls an der Kante mithilfe der interpolierten Punktnormalen bestimmt. Diese wird dann an die BRDF weitergegeben.

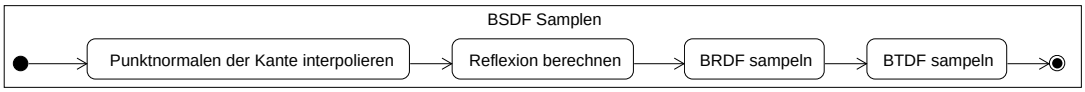


Abbildung 6.13: Ablauf des Sampeln einer BSDF

6.4.2.1 BRDF Samplen

Um eine BRDF MCI zu sampeln, wird eine `std::uniform_real_distribution` Gleichverteilung verwendet[2]. Die Gleichverteilung verwendet eine `std::mt19937` *Mersenne Twister Engine*[3] um Zufallszahlen zu erzeugen. Die Anzahl der Samples pro Radian

[2] *https://en.cppreference.com/w/cpp/numeric/random/uniform_real_distribution*, abgerufen am 19.07.2019
[3] *https://en.cppreference.com/w/cpp/numeric/random/mersenne_twister_engine*, abgerufen am 09.07.2019

wird statisch in der BRDF festgelegt. Die Gleichverteilung bestimmt einen zufälligen Winkel, welcher gesampelt werden soll. In diese Richtung werden dann neue Strahlen ausgesendet.

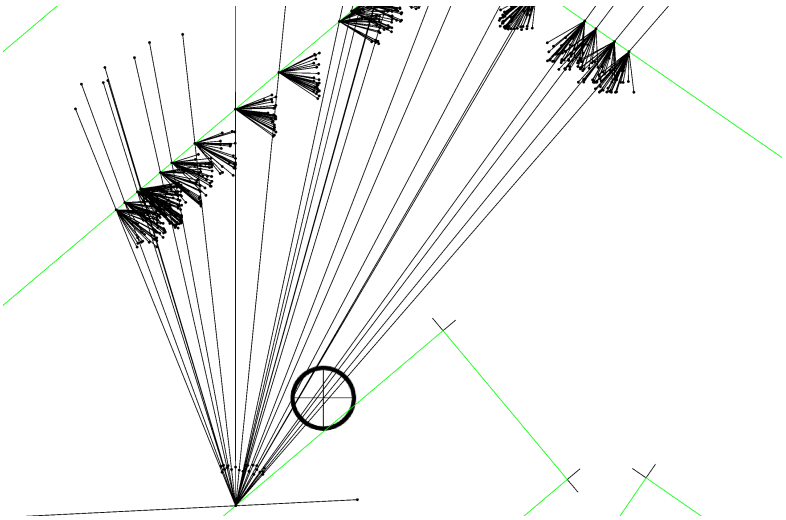


Abbildung 6.14: Beispiel der zufälligen Strahl Generierung in der BRDF. Ein Strahl trifft auf Kanten und wird mehrfach diffus reflektiert.

6.4.2.2 BTDF sampeln

Die BTDF berechnet die Ausleuchtung der Objekte. Dazu wird ein Strahl im inneren des Objektes generiert. Da es keine interne Reflexion o.ä. gibt, kann für diesen Strahl sofort vom Raytracer ein Pfad berechnet werden. Dieser Strahl wird nie in den Strahlbuffer geschrieben. Dies hat der Vorteil, das in Raytracer keine Fallunterscheidungen zwischen transmissions und normalen Strahlen gemacht werden muss. Wurde der Objekt interne Pfad berechnet, so kann anhand dessen Länge l_{path} die Farbe c_{end} für den Endpunkt des Pfades berechnet werden. Die Farbe wird zwischen Start- und Endpunkt des Pfads linear interpoliert [4]:

$$c_{end} = c_{Strahl} * 1 - l_{path} * o \tag{6.3}$$

Der Term $1 - l_{path} * o$ wird dabei auf ≥ 0 beschränkt.

[4] OpenGL interpoliert die Linienfarbe linear zwischen den Vertex Farben der Linie (Abschnitt 6.5.1)

6.4.3 Nebenläufigkeit der Strahlverfolgung

Da die Verfolgung der Strahlen unabhängig voneinander stattfindet, kann dies nebenläufig ausgeführt werden. Dies ist von Vorteil für die Echtzeitfähigkeit des Algorithmus. Die Nebenläufigkeit wird mit `std::threads` realisiert[5]. Es werden so viele Threads erzeugt, wie hardwareseitig nebenläufig unterstützt werden[6]. Um während der Strahlverfolgung Strahl- und Pfadbuffer nicht zwischen den Threads synchronisieren zu müssen, werden diese in entsprechend viele Teile geteilt. Nachdem alle Threads terminiert haben, werden die Buffer wieder vereinigt. Zum Aufteilen und Vereinigen der Buffer wird `std::memcpy` verwendet, da es die schnellste Kopierfunktion ist[7].

6.5 Rasterung

Die Rasterung wird Großteils mit OpenGL ausgeführt. Die Rasterung verwendet die Pfade welche im Raytracer berechnet wurden. Bevor diese an die Grafikkarte übergeben werden können, müssen die Pfade in die richtige Datenstruktur gebracht werden. Dann können diese auf der Grafikkarte gerastert werden. Teile des OpenGL spezifischen Code wurden abgewandelt aus Beispielen von *http://www.opengl-tutorial.org/*, abgerufen am 01.06.2019.

6.5.1 Pfade in OpenGL Datenstruktur bringen

Für das Rastern der Linien in OpenGL wird GL_LINES verwendet[8]. Diese werden in einem Vertexbuffer und einem Farbbuffer an die Grafikkarte übergeben. Die Punkte der Pfade werden in den Vertexbuffer geschrieben, die Farben der Pfade in den Farbbuffer. Jede Linie hat zwei Farbwerte (für je einen Vertex). Auf der Linie wird dann linear interpoliert.

[5] *https://en.cppreference.com/w/cpp/thread/thread*, abgerufen am 14.07.2019
[6] *https://en.cppreference.com/w/cpp/thread/thread/hardware_concurrency*, abgerufen am 14.07.2019
[7] *https://en.cppreference.com/w/cpp/string/byte/memcpy*, abgerufen am 11.07.2019
[8] *https://www.khronos.org/opengl/wiki/Primitive#Line_primitives*, abgerufen am 05.07.2019

6.5.2 Rasterungskorrekturen

Das Rastern mit OpenGL hat den Nachteil, das Gleichung (4.2) nicht voll gelöst werden kann. Die hat zwei Gründe:

1. OpenGL rastert Linien aliased
2. OpenGL beachtet die Steigung einer Linie beim Rastern nicht

6.5.2.1 Korrektur Aliasing

OpenGL rastert Linien aliased. Das Einschalten von Antialiasing führt nicht zur Reduktion von Aliasing Effekten (Anhang A.2). Statt dessen wird das Bild aufgehellt. Auch die fehlenden Unterschiede zwischen 4xMSAA und 16xMSAA lassen darauf schließen, dass MSAA hier nicht besonders effektiv ist. Die Verwendung von GL_LINE_SMOOTH[9] ist nicht geeignet dieses Problem zu lösen, da das Verhalten stark von der OpenGL Implementierung abhängt und die Aliasing Effekte ebenfalls nicht reduziert werden. Hier wird das Bild auch stark aufgehellt. Die einzige Möglichkeit Aliasing zu reduzieren, ist also die Anzahl der Samples (Nicht im Sinne von MSAA pro Pixel, sondern im Sinne der Strahlen aus den Lichtquellen) zu erhöhen.

6.5.2.2 Strahlungsflusskorrektur in Abhängigkeit der Liniensteigung

OpenGL beachtet auch nicht die Entfernung, welches das Licht in in einem Pixel zurück legt (Gleichung (4.2)), sondern nur ob es durch einen Pixel fließt. Dies führt dazu, dass diagonale Linien dunkler dargestellt werden(Abb. 6.15). Dieser Effekt muss dadurch korrigiert werden, das man den Strahlungsfluss eines Pfades abhängig davon, wie viel Strecke er in einem Pixel zurück legt, nach oben korrigiert. Um diesen Korrekturfaktor c_k zu berechnen, wird die Gesamtlänge des Pfad p durch die Anzahl der Pixel geteilt, die er schneidet. Die Anzahl der Pixel, die der Pfad schneidet, ist äquivalent zum größeren der Beträge seiner x und y Komponenten.

$$c_k = \frac{|\vec{p}|}{max(|p_y|, |p_x|)} \tag{6.4}$$

[9] *ttps://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glEnable.xhtml*, abgerufen am 05.07.2019

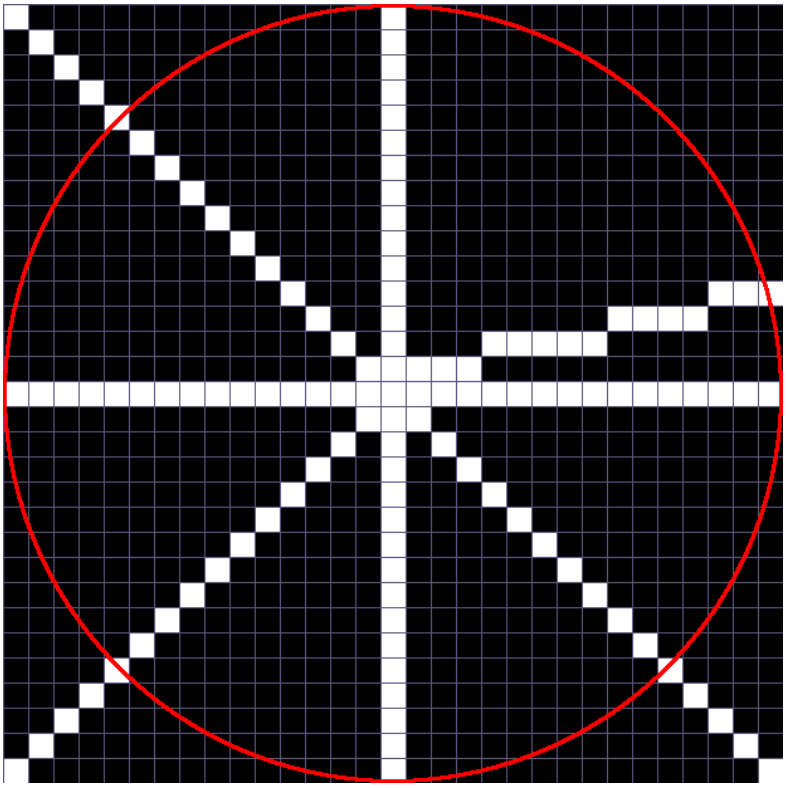
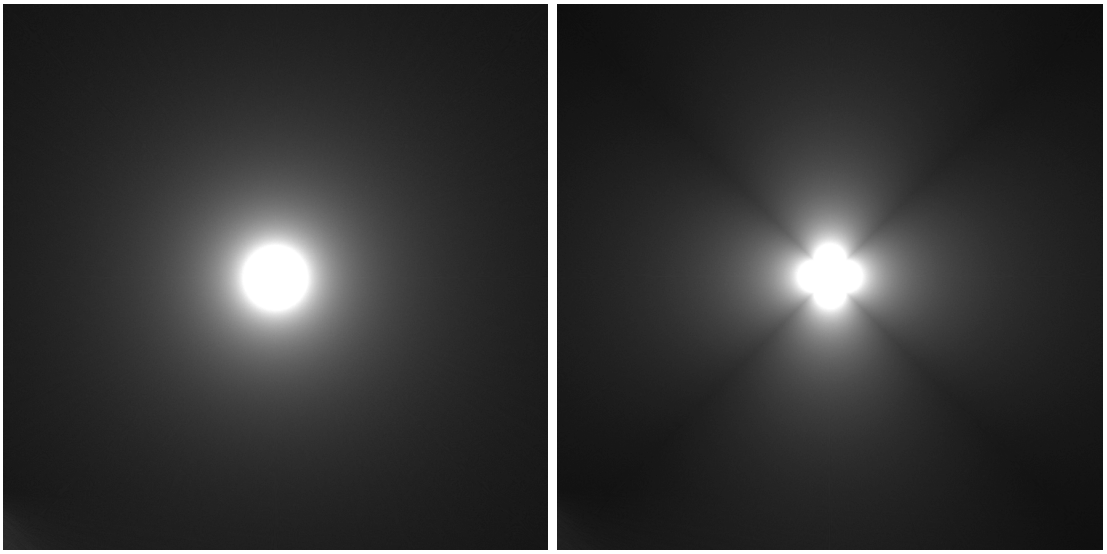


Abbildung 6.15: 8 Linien vom Mittelpunkt des Bildes gerendert. Der Rote Kreis Markiert eine Entfernung von 16 Pixelbreiten zum Mittelpunkt des Bildes. Umso weiter ein Winkel der Linien von einer der Bild Achsen abweicht, umso weniger Pixel werden für die Darstellung der Linie verwendet. Während senkrechte und Horizontale Linien mit vollen 16 Pixeln gerastert werden, werden diagonale nur mit ≈ 11 gerastert. Etwas weniger diagonale (oben Rechts) mit etwas mehr (≈ 15).

6.5.2.3 Rastern mit Hilfe einer Textur

Ein Problem beim Rastern der Szene mit Hilfe vieler Pfade ist, dass die Farbwerte für einzelne Linien sehr klein werden können. Da die Colorbuffer des Framebuffer in OpenGL standardmäßig meist nur 8 oder 10bpp Farbtiefe pro Kanal haben, führt dies in Kombination mit den kleinen Farbwerten der Pfade zu Ganzzahlarithmetik Rundungsfehlern (Abb. 6.17). Deswegen wird die Lightmap statt in den Standard Framebuffer in eine Textur gerendert, welche dann im Framebuffer dargestellt wird. Diese hat 32bpp Farbtiefe RGB (GL_RGB32F, GL_RGB) und als Gleitkomma Datentypen (GL_FLOAT)[10]. Dies

[10] *https://www.khronos.org/registry/OpenGL-Refpages/es3.0/html/glTexImage2D.xhtml*, abgerufen am 05.07.2019



(a) Mit Korrekturfaktor c_k (b) Ohne Korrekturfaktor c_k

Abbildung 6.16: Vergleich von Bildern mit Pfad Flux Korrektur und ohne

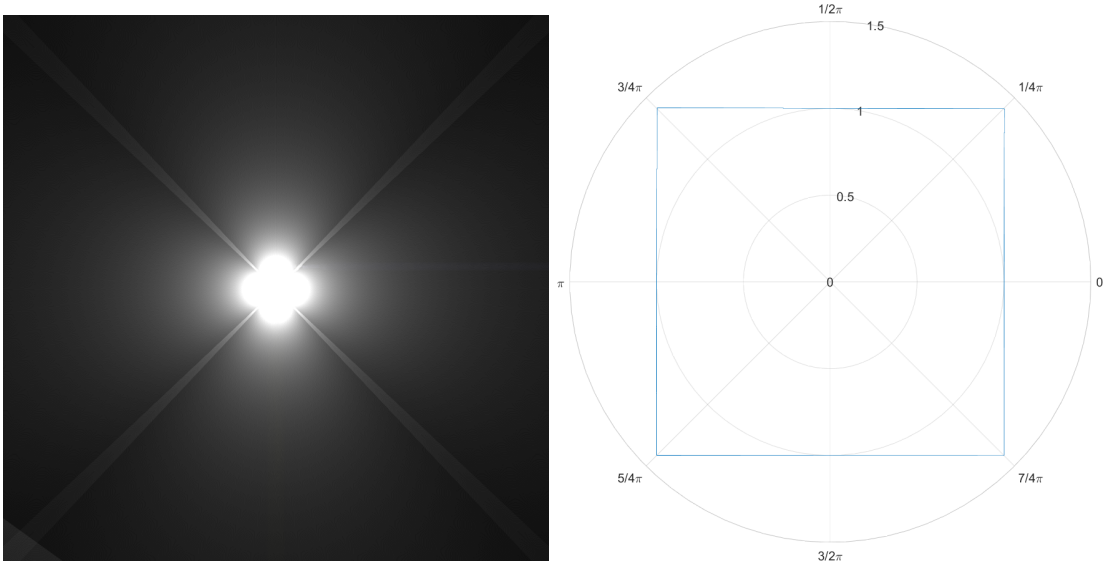
ist von der Genauigkeit her ausreichend.

```
glTexImage2D(GL_TEXTURE_2D, 0,GL_RGB32F,
             Anzeigebereich.x, Anzeigebereich.y, 0,GL_RGB, GL_RGB, 0);
```

6.5.3 Additiver Modus

Im Normalfall wird im Framebuffer die Farbe eines Pixels überschrieben, sobald ein neues Primitiv an dieser Stelle gerastert wurde. Da aber nach Gleichung (4.2) alle Paths, die ein Pixel kreuzen, aufsummiert werden sollen, muss der Wert im Framebuffer $\vec{c_{fb}}$ mit dem neuen Wert $\vec{c_{neu}}$ addiert werden. Dies kann man mit dem Additiven Modus (engl. Blend Mode) von OpenGL realisieren.[11] Wie die Pixel addiert werden, bestimmt hierbei die Blend Funktion (glBlendFunc[12]). Möchte man einfach nur aufsummieren wie in Gleichung (4.2), so wählt man `glBlendFunc(GL_ONE, GL_ONE)`. Der alte im Framebuffer und der neue Wert werden dabei eins zu eins addiert. Dies hat jedoch den Nachteil, dass in der Lightmap die Lichtquellen extrem hell erscheinen. Pixel, welche weiter weg von der

[11] *https://www.khronos.org/opengl/wiki/Blending*, abgerufen am 05.07.2019
[12] *https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glBlendFunc.xhtml*, abgerufen am 05.07.2019



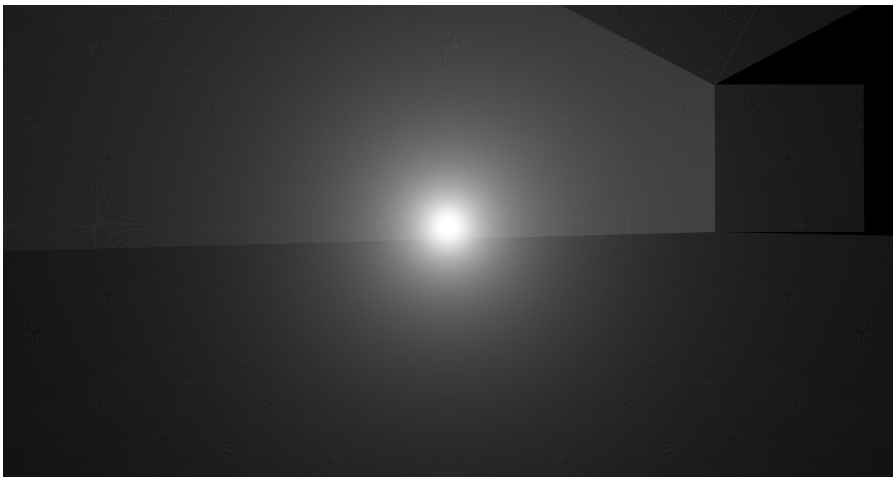
(a) Fehler c_k (b) c_k abhängig von der Richtung des Pfades

Abbildung 6.17: Fehler durch 8bpp Farbtiefe im Framebuffer in Kombination mit der Pfad Strahlungsflusskorrektur. Dort wo c_k groß wird, kippt der Rundungsfehler. Dies äußert sich durch die Diagonalen Streifen im Bild

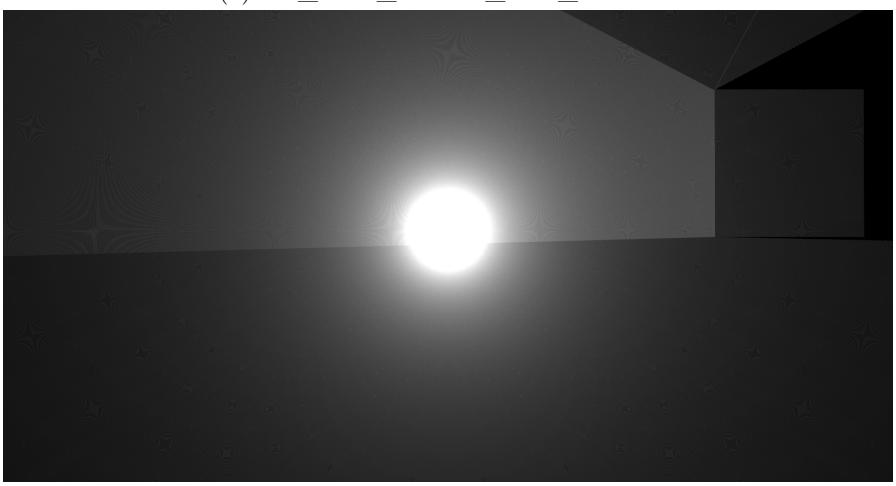
Lichtquelle liegen, erscheinen hingegen verhältnismäßig dunkel. Um die Optik zu verbessern, wird die Blend Funktion auf `glBlendFunc(GL_ONE, GL_ONE_MINUS_SRC_COLOR)` geändert (Gleichung (6.5)):

$$\vec{c_{nfb}} = 1 * \vec{c_{fb}} + (\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} - \vec{c_{fb}}) * \vec{c_{neu}} \tag{6.5}$$

Dies hat zur Folge, dass ein neuer Farbwert immer weniger Einfluss hat, umso größer der Farbwert im Framebuffer bereits ist.



(a) GL_ONE_MINUS_SRC_COLOR



(b) GL_ONE

Abbildung 6.18: Vergleich der Blend Funktionen. Bei GL_ONE_MINUS_SRC_COLOR werden helle Regionen abgedunkelt, während dunkle Regionen weitesgehend gleich bleiben. Rechts befindet sich ein Quader im Bild, der Licht reflektiert

7 Evaluation

Hier werden die Ergebnisse des Systems mit Hinblick auf ihre Optik, Erfüllung der Eigenschaften der globalen Beleuchtung und der Laufzeit evaluiert.

7.1 Eigenschaften Globaler Beleuchtung

Der Algorithmus erfüllt die Eigenschaften, welche in Abschnitt 2.4 aufgestellt worden sind. Die Kamera wird definiert durch den Anzeigebreich, die Bildsynthese erfolgt über OpenGL.

7.1.1 Lichtquellen

Da Strahlen der Lichtquellen frei platzierbar sind, ist jede Form von Lichtquelle realisierbar.

7.1.2 Oberflächenstreuung

Während die rein gerichtete Reflexion sehr gut funktioniert, hat die diffuse Reflexion einige Probleme. Sie erzeugt sehr viel Bildrauschen in der Lightmap, wenn die Anzahl der Samples beim MCI Sampling der BRDF nicht sehr hoch gewählt wird. Dies sieht man z.B. an der hinteren Kiste in Abb. A.15. Sie ist jedoch funktional.

7.1.3 Transmission

Die lineare Interpolation in der BTDF erzeugt leider Artefakte in der Lightmap, da sie abhängig von der Geometrie des Objektes und des Strahls ist. Weil vom Eintritts bis zum Austrittspunkt interpoliert wird, nimmt z.B. bei dünneren Objekten die Helligkeit des Pfads schneller ab.

7.1.4 Indirekte Beleuchtung

Die indirekte Beleuchtung funktioniert. Objekte, welche von der Lichtquelle durch ein anderes Objekt verdeckt sind, werden von Objekten, welche Licht reflektieren, beleuchtet (Abb. A.15).

7.2 Laufzeit Messungen

Während das Rendern von Lightmaps mit rein gerichteten Reflexionen in Echtzeit problemlos möglich ist (Abb. A.16), ist das Rendern von Lightmaps mit diffusen Lichtern und Reflexionen dieser Form nicht in Echtzeit möglich (Abb. A.14, A.13 und A.15). Um einigermaßen rauschfreie Lightmaps zu erhalten, ist die Anzahl der benötigten Rays, je nach Szene, zu groß, um die Strahlverfolgung auf der CPU mit der implementierten Technik auszuführen.

Alle Performance Messungen wurden auf einem *Inter Core i5-2400* und einer *AMD R9 290* durchgeführt. Für den GCC wird Optimierungsstufe *O3* eingestellt.

7.3 Raytracing Algorithmus

Das Profiling von der Raytracing Phase zeigt, dass die Kosten für das Raytracing zu $> 90\%$ aus der BVH stammen (Abschnitt 7.3). Deswegen werden die anderen Teile hier für die Performance Messung vernachlässigt. Das Profiling wurde mit Hilfe der Szene Abb. A.14 und dem Visual Studio Profiler erstellt, da für mit dem GCC erstellte

Programme kein äquivalent gutes Profiling Tool auf Windows existiert [1]. Hierbei sieht man, dass die eigentlichen Schnitttests mit den Primitiven sehr erfolgreich vermieden werden, sie tragen fast nichts zur Gesamtlaufzeit bei (siehe Abschnitt 7.3, dort werden sie sogar unter *Andere* geführt). Dies deckt sich mit Abb. 7.3.

Abbildung 7.1: Profiling der Raytracing Phase. Angaben in % der Gesamtlaufzeit der Phase

7.3.1 Performance BVH

Zur Bestimmung der Leistung der BVH wurde die Rasterungs Phase abgeschaltet. Um die BVH zu testen wird die Szene aus Abb. A.17 verwendet. In der Mitte der Szene sitzt eine Punktlichtquelle, welche 2×10^5 Rays in einem Abstrahlwinkel von 2π rad in die Szene strahlt. Die maximale Reflexionszahl der Strahlen wurde auf Null gesetzt, sodass die Anzahl der Strahlen mit der Abnahme der Kanten immer gleich bleibt und sich nicht durch veränderte Reflexionen verändert. Die Szenen wurden so entworfen, dass die Wahrscheinlichkeit, eine Kante zu treffen von der Lichtquelle aus möglichst gleich bleibt. Für die Reduktion der Anzahl der Kanten wurde das selbe Polygon stufenweise immer weiter vergröbert.

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Primitve	2263	1795	1379	1041	859	755	547	417
Laufzeit [ms]	380,341	389,375	357,264	346,388	355,646	350,496	344,658	334,096

Tabelle 7.1: BVH Performance, zu Abb. 7.3

Abb. 7.3 zeigt, das die Leistung mit steigender Kanten Anzahl nicht stark einbricht. Dies zeigt die Effektivität der BVH. Selbst bei sehr viel mehr Kanten müssen nicht viel mehr Schnitttests durchgeführt werden. Die Inkonsistenz der Steigung lässt sich mit den Qualitätsproblemen durch die Blattknotengröße erklären (Abschnitt 7.3.3).

[1] Entsprechend wurde der Sourcecode hier auch mit dem MVC 2017 kompiliert. Da Laufzeiten weitesgehend identisch zu dem mit dem GCC erstellten Programm sind, lassen sich die Profiling Ergebnisse weitestgehend übertragen

Abbildung 7.2: Profiling der Schnittpunkt Methode der BVH. Angaben in % der Gesamtlaufzeit der Phase. `BBox::intersect` ist hierbei die Slabs Methode der AABB

Für 2×10^5 Rays benötigt der Raytracer im schlechtesten Fall $389.375\,\mathrm{m\,s} \approx 400\,\mathrm{m\,s}$. Dies entspricht einer Leistung von $5 \times 10^5 \frac{Rays}{\mathrm{s}}$

7.3.2 Bestimmung der Blattknotengröße

Um eine optimale Größe für die Knoten zu bestimmen, wird die Szene in Abb. A.17 mit der höchsten Polygonanzahl mit verschiedenen Blattknotengrößen ausgeführt. Als beste Knotengröße ergibt sich 3 Kanten (Abb. 7.4).

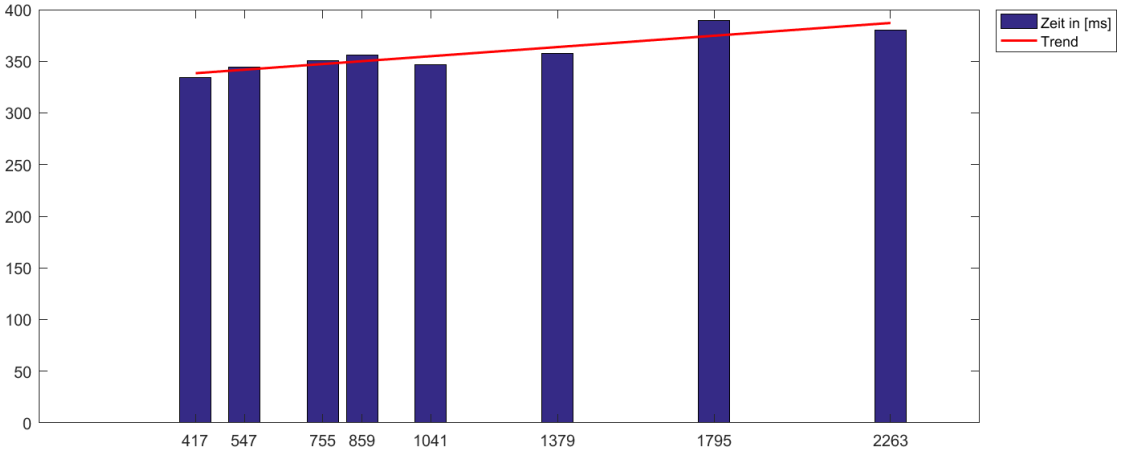


Abbildung 7.3: Performance der BVH: Abhängig von den Kanten Anzahl in der Szene. Der Trend ist eine Ausgleichgrade. Durchschnitt von 240 Messdurchläufen pro Szene

7.3.3 Qualität der BVH

Durch die feste Größe der BVH Kindknoten kann es in der BVH zu ungünstigen Platzierungen der Knoten kommen. Dies zeigt Abb. A.5. Die Knoten auf der rechten unteren Seite erstrecken sich über zwei Objekte, da das untere Objekte nicht genug Kanten besitzt.

Auf der gleichen Grafik ist links zu sehen, das eine starke Ungleichverteilung der Kanten (Hoch aufgelöste Wolke gegenüber dem niedrig aufgelösten Quadrat) zu einer falschen Wahl der Achse für die Teilung führen kann.

7.4 Rastern

Auch wenn das Rastern von Linien eines der Dinge ist, für die Grafikkarten entworfen werden, so ist das massenhafte Rendern dieser mit Blending ein Randfall. Abbildung 7.5 zeigt, dass die Zeitkomplexität linear steigt.

Die Rasterung ist dabei in der Lage $6.6242 \times 10^3 \frac{Paths}{\mathrm{ms}}$ zu zeichnen. Damit wird die Echtzeitgrenze bei 2.6497×10^5 Pfaden erreicht.

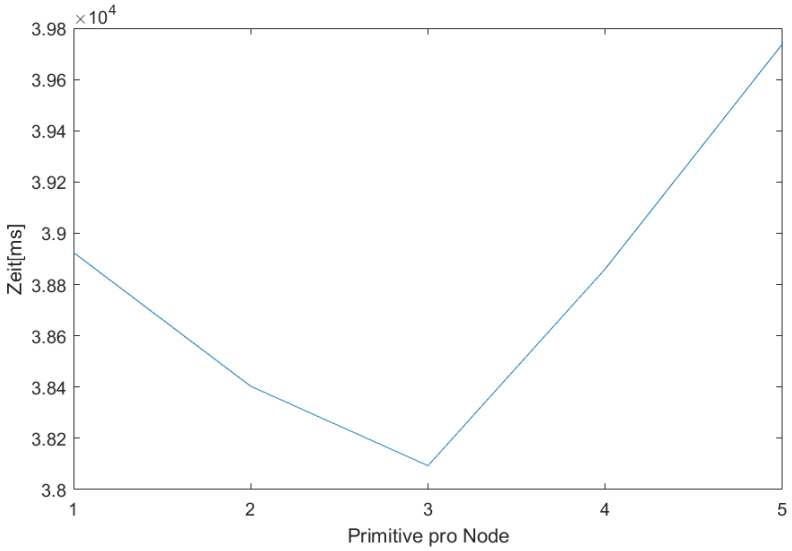


Abbildung 7.4: BVH Performance: Abhängig davon, wie viele Kanten die Kindknoten der BVH enthalten. Gemessen wurde 240 mal die Szene mit den meisten Kanten aus Abb. A.17

7.5 Optik

Die Ergebnisse in Anhang A.3 zeigen, dass die Anwendung der globalen Beleuchtung auf 2D Grafik mit Hilfe von Raytracing möglich ist, und auch glaubhafte Resultate erzielen kann.

Das Verbinden von Lightmaps mit den Texturen verringert ebenfalls die Sichtbarkeit von Rauschen in den Lightmaps.

7.6 Zusammenhang mit der Echtzeitfähigkeit

Um bei FullHD ein weitestgehend aliasingfreies Bild zu erhalten, muss man Punktlichtquellen bei ca. $7957 \frac{Strahlen}{\text{rad}}$ initialisieren. Das sind pro Punktlichtquelle mit Abstrahlwinkel von 2π rad rund 2.5×10^4 Rays.

Für eine diffuse Reflexion mit nur geringem Rauschen werden rund 20 Samples pro Radian Abstrahlwinkel in der BRDF benötigt, wobei dies auch stark von der Szene abhängt. Das gleiche gilt auch für diffuse Volumenlichtquellen (eine Objekt, welches Strahlen reflektiert, ist ja nichts anderes als eine Volumenlichtquelle).

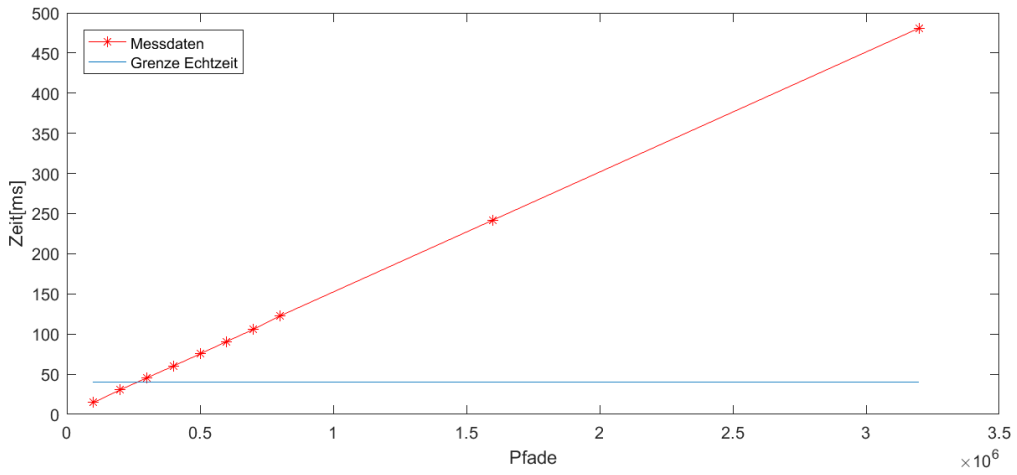


Abbildung 7.5: Performance Rastern. Gemessen wurde mit Hilfe eines einzigen Punktlichts, dessen Rayanzahl immer weiter erhöht wurde.

	Pfade	**Zeit [ms]**
1	100	14,695
2	200	30,470
3	300	45,387
4	400	60,183
5	500	75,208
6	600	90,687
7	700	105,800
8	800	122,495
9	1600	242,004
10	3200	481,141

Tabelle 7.2: Performance Rastern: Messwerte, (Abb. 7.5)

Diese Werte zeigen, warum eine diffuse Szene in Echtzeit zu rendern schwer möglich ist. Selbst mit nur einer diffusen Reflexion pro Strahl ist man bei $2.5 \times 10^4 * 20 = 5 \times 10^5$. Strahlen und Pfaden. Dies übersteigt die Echtzeitgrenze des Raytracer und die der Rasterung schon alleine. Dieser Effekt lässt sich gut in Abb. 6.14 beobachten.

8 Fazit

Die Arbeit zeigt, dass globale Beleuchtung in 2D möglich und durchaus sinnvoll ist.

Während das Echtzeitkriterium von 25 FPS nur unter sehr speziellen Randbedingungen erfüllt werden kann (Abb. A.16), ist die Berechnung immer noch sehr viel schneller als die Berechnung von Raytracing in 3D. Dies ermöglicht die Verwendung für Lichteffekte, welche nicht mit jedem Frame aktualisiert werden müssen, wie z.B. die Beleuchtung der Szene durch die Sonne, wo sich abhängig von der Tageszeit Winkel und Farbe langsam ändern.

Auch ist eine Verwendung zum Erstellen von Lightmaps im klassischen Sinne möglich, wobei das Licht nicht mehr in Echtzeit berechnet wird, sondern statisch direkt in die Texturen der Szene gepackt.

8.1 Ausblick

Eine erste sinnvolle Erweiterung wäre die Verwendung von eindimensionalen Texturen auf den Polygonen, um so in der BRDF die unterschiedliche Behandlung der Farbkanäle zu ermöglichen. Des Weiteren wäre eine Vervollständigung der Rendering Pipeline oder das Einbinden in eine vorhandene Engine sinnvoll, um eine tatsächliche Bildsynthese zu haben.

Die Raytracing Phase ist viel langsamer als die Rasterungs Phase. Da es die Limitierung auf 2D nicht soweit vereinfacht hat einen echtzeitfähigen Raytracer zu scheiben, wie gedacht, ist hier ist evtl. der Austausch des Raytracing Algorithmus durch vorhandene Algorithmen sinnvoll. Beispiele dafür sind Intel Embree[1] oder eine Auslagerung auf die Grafikkarte , z.B. mit den Raytracing Feature von DirectX 12 [2].

[1] *https://www.embree.org/*, abgerufen am 14.07.2019
[2] *https://docs.microsoft.com/de-de/windows/win32/direct3d12/direct3d-12-graphics*, abgerufen am 14.07.2019

Da das Linien Rasterung der Grafikkarte nicht schnell genug ist, um die Anzahl von Strahlen zu zeichnen, um rauschfreie Bilder zu erhalten, könnte ein Denoising Algorithmus eingestzt werden, um bei weniger Samples dennoch ein rauschfreies Bild zu erhalten. Diese Technik wird beim 3D Raytracing bereits angewendet[8][3].

[3] *https://openimagedenoise.github.io/*, abgerufen am 14.07.2019

Literaturverzeichnis

[1] IEEE Standard for Floating-Point Arithmetic. In: *IEEE Std 754-2008* (2008), Aug, S. 1–70

[2] Appel, Arthur: Some Techniques for Shading Machine Renderings of Solids. In: *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*. New York, NY, USA : ACM, 1968 (AFIPS '68 (Spring)), S. 37–45. – URL *http://doi.acm.org/10.1145/1468075.1468082*

[3] Arvo, James: Backward Ray Tracing. In: *In ACM SIGGRAPH '86 Course Notes - Developments in Ray Tracing*, 1986, S. 259–263

[4] Asmail, Clara: Bidirectional Scattering Distribution Function (BSDF): A Systematized Bibliography. In: *Journal of research of the National Institute of Standards and Technolog* (1991)

[5] Bentley, Jon L.: Multidimensional Binary Search Trees Used for Associative Searching. In: *Commun. ACM* 18 (1975), September, Nr. 9, S. 509–517. – URL *http://doi.acm.org/10.1145/361002.361007*. – ISSN 0001-0782

[6] Cook, Robert L. ; Porter, Thomas ; Carpenter, Loren: Distributed Ray Tracing. In: *SIGGRAPH Comput. Graph.* 18 (1984), Januar, Nr. 3, S. 137–145. – URL *http://doi.acm.org/10.1145/964965.808590*. – ISSN 0097-8930

[7] Crane, Keebab: *Bias in Rendering*. 2006

[8] Crassin, Cyril ; Neyret, Fabrice ; Sainz, Miguel ; Green, Simon ; Eisemann, Elmar: Interactive Indirect Illumination Using Voxel Cone Tracing: A Preview. In: *Symposium on Interactive 3D Graphics and Games*. New York, NY, USA : ACM, 2011 (I3D '11), S. 207–207. – URL *http://doi.acm.org/10.1145/1944745.1944787*. – ISBN 978-1-4503-0565-5

[9] Fujimoto, A. ; Tanaka, Takayuki ; Iwata, K.: Tutorial: Computer Graphics; Image Synthesis. New York, NY, USA : Computer Science Press, Inc., 1988, Kap. ARTS: Accelerated Ray-tracing System, S. 148–159. – URL *http://dl.acm.org/citation.cfm?id=95075.95111*. – ISBN 0-8186-8854-4

[10] Goral, Cindy M. ; Torrance, Kenneth E. ; Greenberg, Donald P. ; Battaile, Bennett: Modeling the Interaction of Light Between Diffuse Surfaces. In: *SIGGRAPH Comput. Graph.* 18 (1984), Januar, Nr. 3, S. 213–222. – URL *http://doi.acm.org/10.1145/964965.808601*. – ISSN 0097-8930

[11] Haines, Eric: Spline Surface Rendering, and Whatś Wrong with Octrees. In: *Ray Tracing News* (1988)

[12] Heckbert, Paul S.: Adaptive Radiosity Textures for Bidirectional Ray Tracing. In: *SIGGRAPH Comput. Graph.* 24 (1990), September, Nr. 4, S. 145–154. – URL *http://doi.acm.org/10.1145/97880.97895*. – ISSN 0097-8930

[13] Jensen, Henrik W.: *Realistic Image Synthesis Using Photon Mapping*. Natick, MA, USA : A. K. Peters, Ltd., 2001. – ISBN 1-56881-147-0

[14] Jensen, Henrik W. ; Marschner, Stephen R. ; Levoy, Marc ; Hanrahan, Pat: A Practical Model for Subsurface Light Transport. In: *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA : ACM, 2001 (SIGGRAPH '01), S. 511–518. – URL *http://doi.acm.org/10.1145/383259.383319*. – ISBN 1-58113-374-X

[15] Kajiya, James T.: The Rendering Equation. In: *SIGGRAPH Comput. Graph.* 20 (1986), August, Nr. 4, S. 143–150. – URL *http://doi.acm.org/10.1145/15886.15902*. – ISSN 0097-8930

[16] Kay, Timothy L. ; Kajiya, James T.: Ray Tracing Complex Scenes. In: *SIGGRAPH Comput. Graph.* 20 (1986), August, Nr. 4, S. 269–278. – URL *http://doi.acm.org/10.1145/15886.15916*. – ISSN 0097-8930

[17] Klosowski, James T. ; Held, Martin ; Mitchell, Joseph S. B. ; Sowizral, Henry ; Zikan, Karel: Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs. In: *IEEE Transactions on Visualization and Computer Graphics* 4 (1998), Januar, Nr. 1, S. 21–36. – URL *http://dx.doi.org/10.1109/2945.675649*. – ISSN 1077-2626

[18] Lafortune, Eric P. ; Willems, Yves D.: Bi-directional path tracing. In: *Proceedings of Third International Conference on Computational Graphics and Visualization Techniques (Compugraphics '93)*. Alvor, Portugal, December 1993, S. 145–153

[19] Lambert, J.H. ; Anding, E.: *Lamberts Photometrie: Photometria, sive De mensura et gradibus luminus, colorum et umbrae (1760)*. W. Engelmann, 1892 (Lamberts Photometrie: Photometria, sive De mensura et gradibus luminus, colorum et umbrae). – URL *https://books.google.de/books?id=jPk4AAAAMAAJ*

[20] MacDonald, J. D. ; Booth, Kellogg S.: Heuristics for ray tracing using space subdivision. In: *The Visual Computer* 6 (1990), May, Nr. 3, S. 153–166. – URL *https://doi.org/10.1007/BF01911006*. – ISSN 1432-2315

[21] O. Bartell, F ; Dereniak, Eustace ; L. Wolfe, W: The Theory And Measurement Of Bidirectional Reflectance Distribution Function (Brdf) And Bidirectional Transmittance Distribution Function (BTDF). In: *Proceedings of SPIE - The International Society for Optical Engineering* 257 (1980), 01, S. 154–160

[22] Pharr, Matt ; Jakob, Wenzel ; Humphreys, Greg: *Physically based rendering: From theory to implementation*. Morgan Kaufmann, 2016

[23] Phong, Bui T.: Illumination for Computer Generated Pictures. In: *Commun. ACM* 18 (1975), Juni, Nr. 6, S. 311–317. – URL *http://doi.acm.org/10.1145/360825.360839*. – ISSN 0001-0782

[24] Pöpsel, Josef ; Claussen, Ute ; Klein, Rolf-Dieter ; Plate, Jürgen: *Computergrafik: Algorithmen und Implementierung*. Springer-Verlag, 2013

[25] Roth, Scott D.: Ray casting for modeling solids. In: *Computer Graphics and Image Processing* 18 (1982), S. 109–144

[26] Rubin, Steven M. ; Whitted, Turner: A 3-dimensional Representation for Fast Rendering of Complex Scenes. In: *SIGGRAPH Comput. Graph.* 14 (1980), Juli, Nr. 3, S. 110–116. – URL *http://doi.acm.org/10.1145/965105.807479*. – ISSN 0097-8930

[27] Shirley, Peter: *Ray Tracing in One Weekend*. 1.54. 2018

[28] Tomas Akenine-Möller, Naty H.: *Real-time Rendering*. 2018

[29] Veach, Eric: *Robust Monte Carlo Methods for Light Transport Simulation*. Stanford, CA, USA, Dissertation, 1998. – AAI9837162

[30] Veach, Eric ; Guibas, Leonidas J.: Metropolis Light Transport. In: *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA : ACM Press/Addison-Wesley Publishing Co., 1997 (SIGGRAPH '97), S. 65–76. – URL *https://doi.org/10.1145/258734.258775*. – ISBN 0-89791-896-7

[31] Vinkler, Marek ; Havran, Vlastimil ; Bittner, Jiří: Bounding Volume Hierarchies Versus Kd-trees on Contemporary Many-core Architectures. In: *Proceedings of the 30th Spring Conference on Computer Graphics*. New York, NY, USA : ACM, 2014 (SCCG '14), S. 29–36. – URL *http://doi.acm.org/10.1145/2643188.2643196*. – ISBN 978-1-4503-3070-1

[32] Weghorst, Hank ; Hooper, Gary ; Greenberg, Donald P.: Improved Computational Methods for Ray Tracing. In: *ACM Trans. Graph.* 3 (1984), Januar, Nr. 1, S. 52–69. – URL *http://doi.acm.org/10.1145/357332.357335*. – ISSN 0730-0301

[33] Whitted, Turner: An Improved Illumination Model for Shaded Display. In: *Commun. ACM* 23 (1980), Juni, Nr. 6, S. 343–349. – URL *http://doi.acm.org/10.1145/358876.358882*. – ISSN 0001-0782

[34] Williams, Amy ; Barrus, Steve ; Morley, R. K. ; Shirley, Peter: An Efficient and Robust Ray-box Intersection Algorithm. In: *ACM SIGGRAPH 2005 Courses*. New York, NY, USA : ACM, 2005 (SIGGRAPH '05). – URL *http://doi.acm.org/10.1145/1198555.1198748*

A Anhang

A.1 BVH mit allen Baum Levels

Darstellung aller Level eine BVH mit 8 Levels insgesamt. Grün sind die Kanten einer der Szene welche sich nicht momentan nicht in einem ausgewählten BVH Knoten befinden. Kanten, welche sich in einem Knoten im aktuellen BVH Level befinden werden blau dargestellt. Schwarz sind die Punktnormalen der Primitiven gekennzeichnet. Der Anzeigebereich ist durch die Ecken in der Seite und das Kreuz in der Mitte gekennzeichnet.

Links neben den Darstellungen der BVH in der Szene befindet sich jeweils die Hierarchie der BVH. Die Nodes der jeweiligen Levels sind rot markiert. Die Node darstellung ist nach dem Schema [C:*Index Linker Kindknoten*|*Index Rechte Kindknoten*] [P:*Unterer Kanten Index*|*Oberer Kanten Index*] N:*Blattknoten* aufgebaut.

Die Blattknotengröße n_{max} beträgt hier 5.

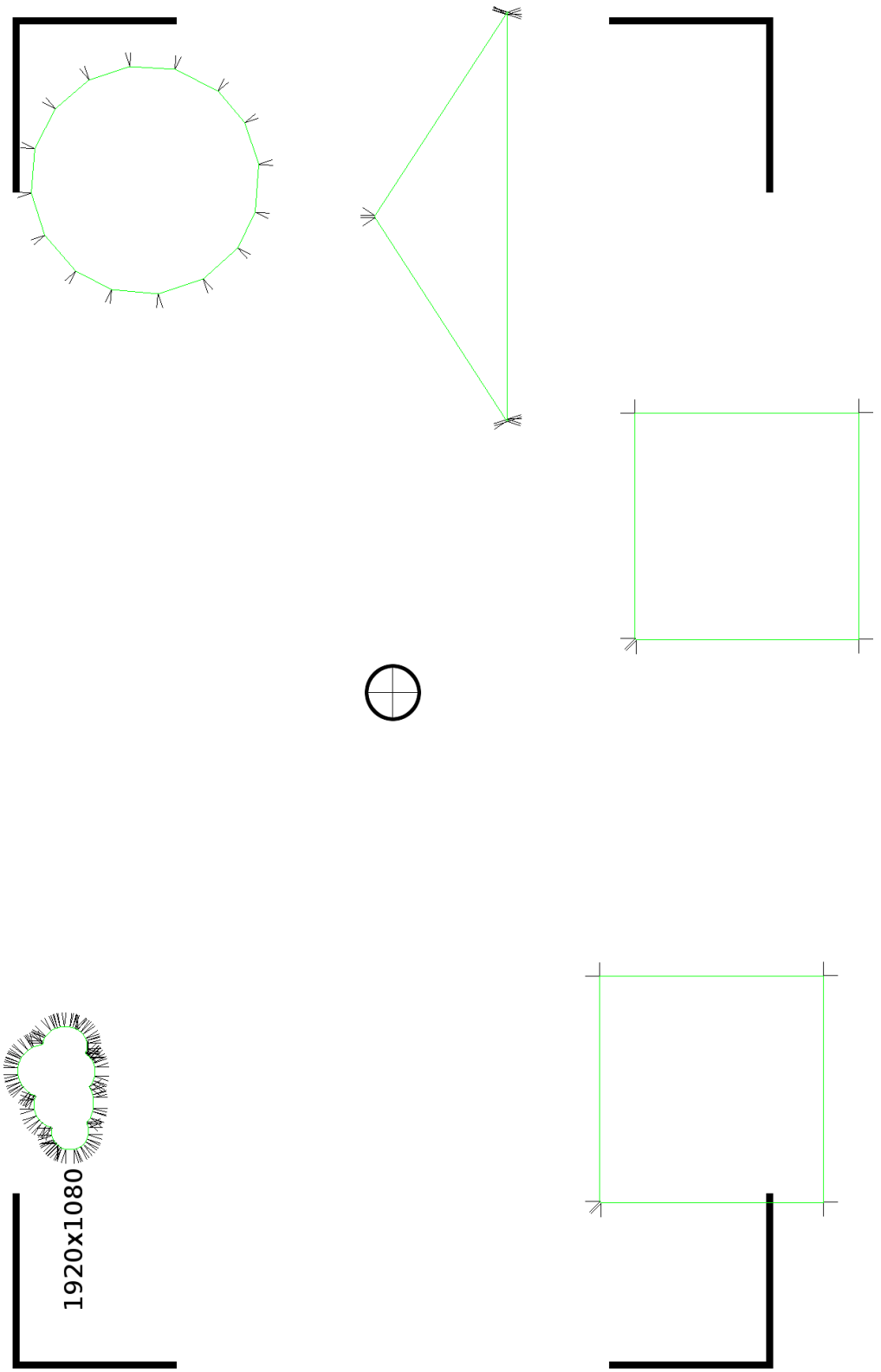


Abbildung A.1: Die Szene der BVH

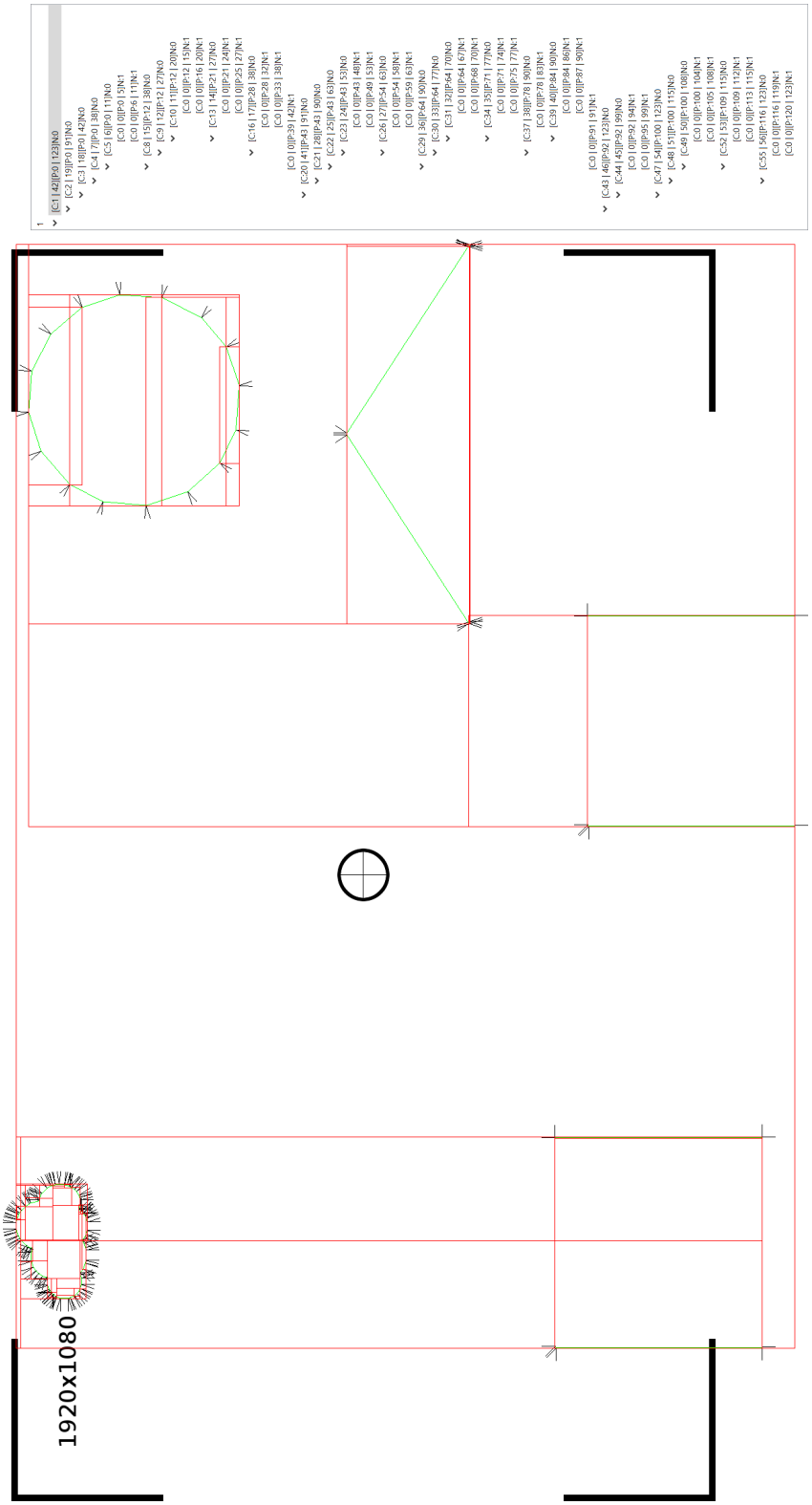


Abbildung A.2: Komplette BVH mit allen Nodes in Rot

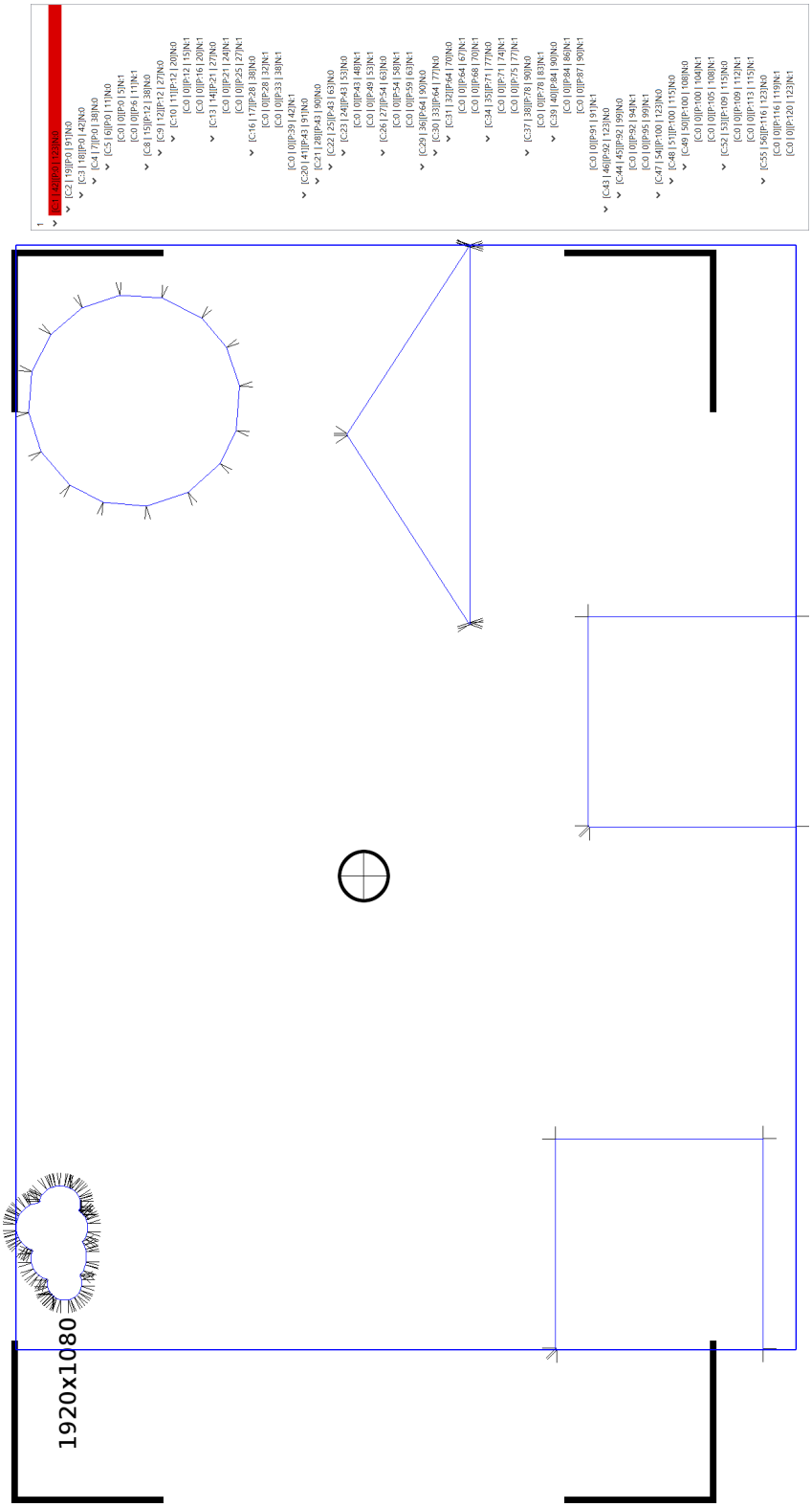


Abbildung A.3: Level 1

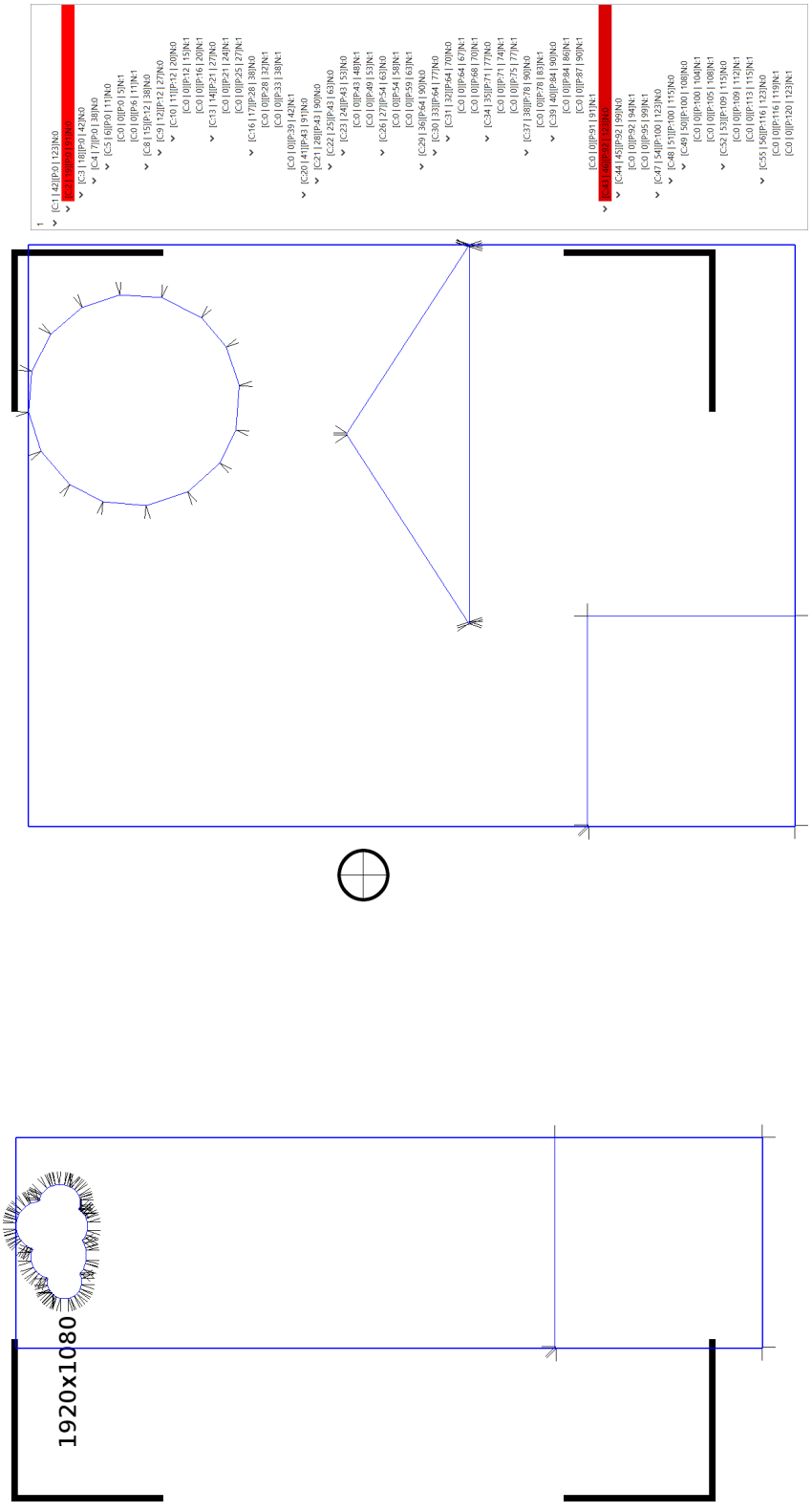


Abbildung A.4: Level 2

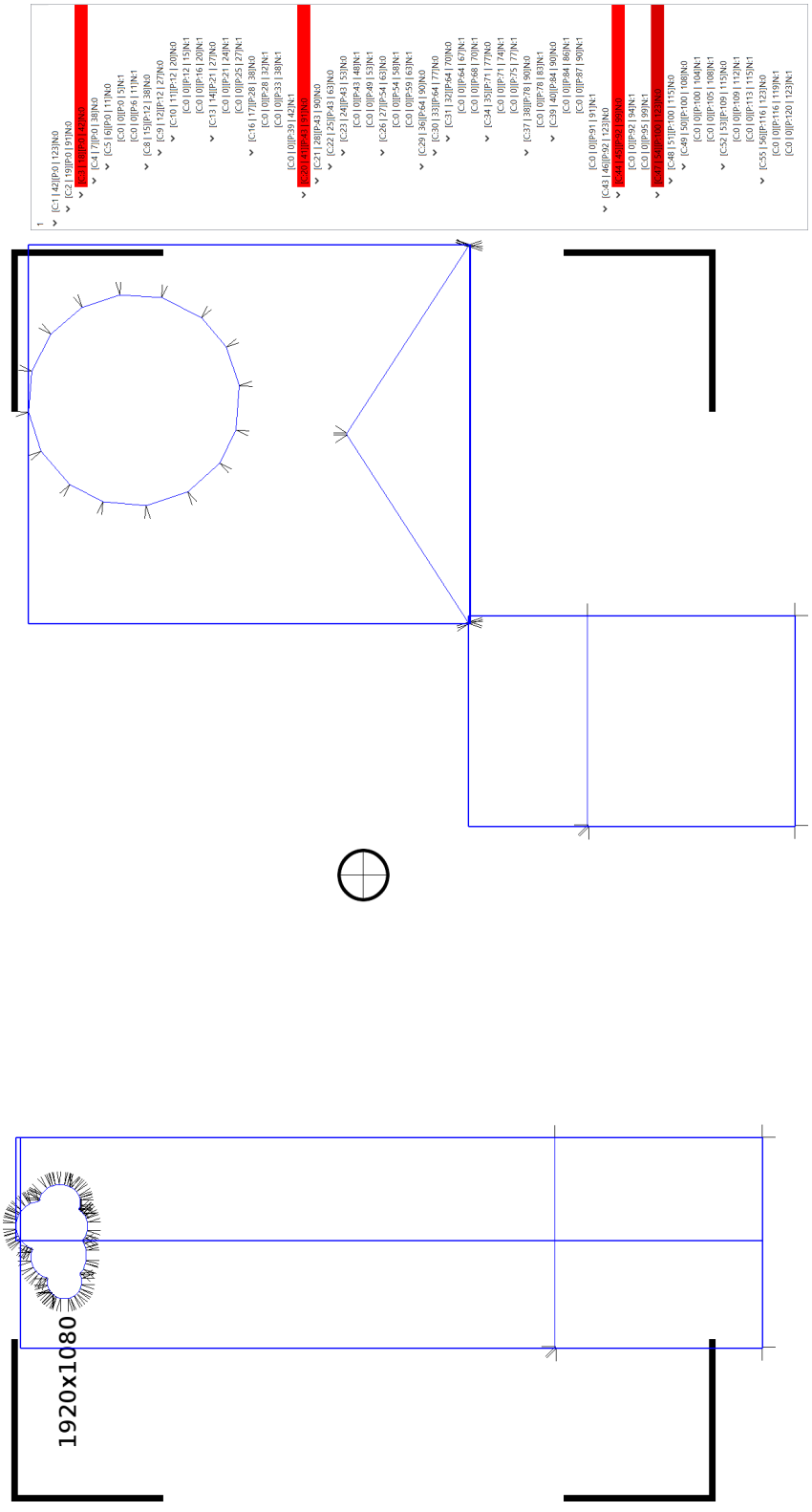


Abbildung A.5: Level 3

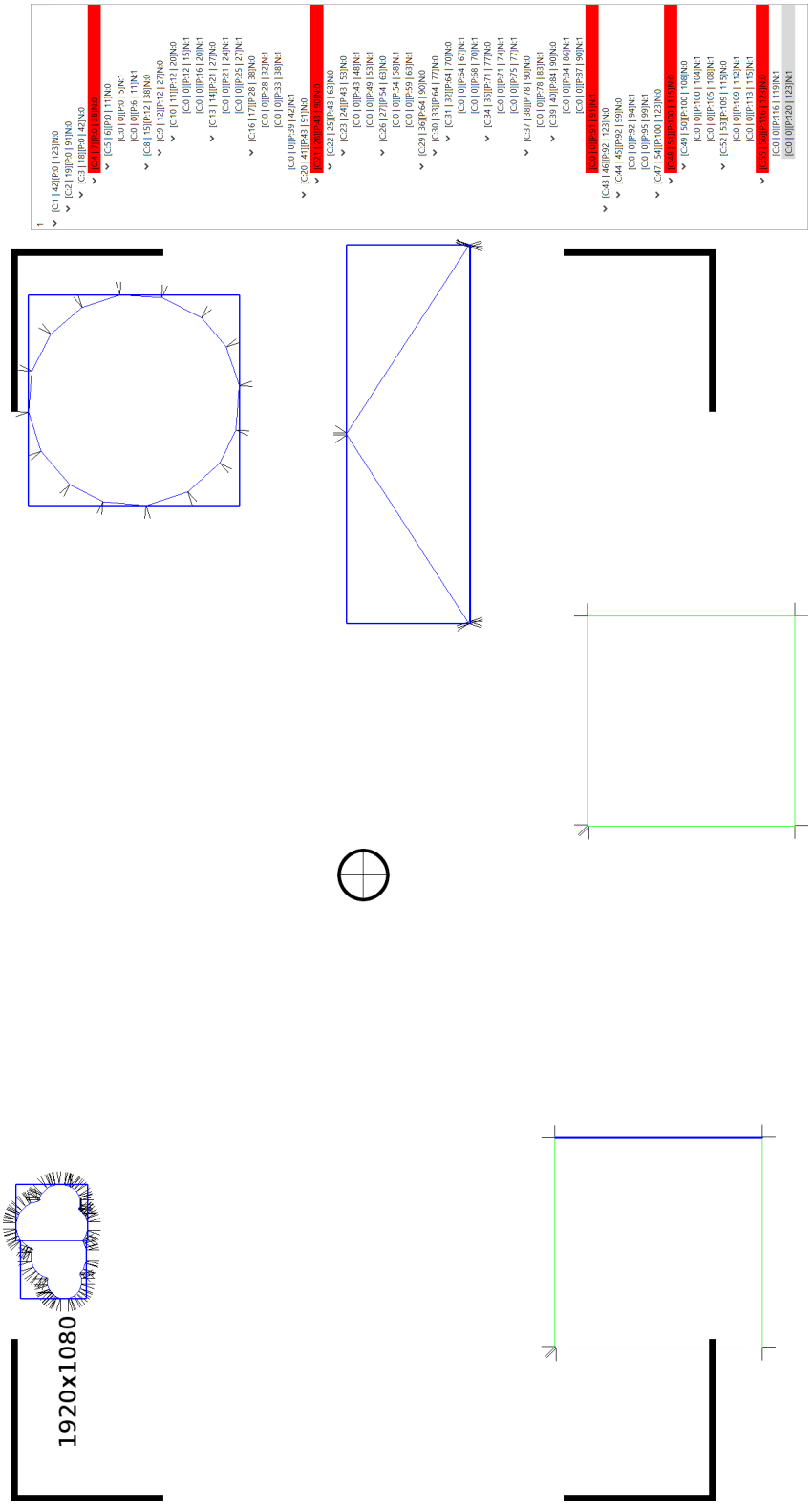


Abbildung A.6: Level 4

Abbildung A.7: Level 5

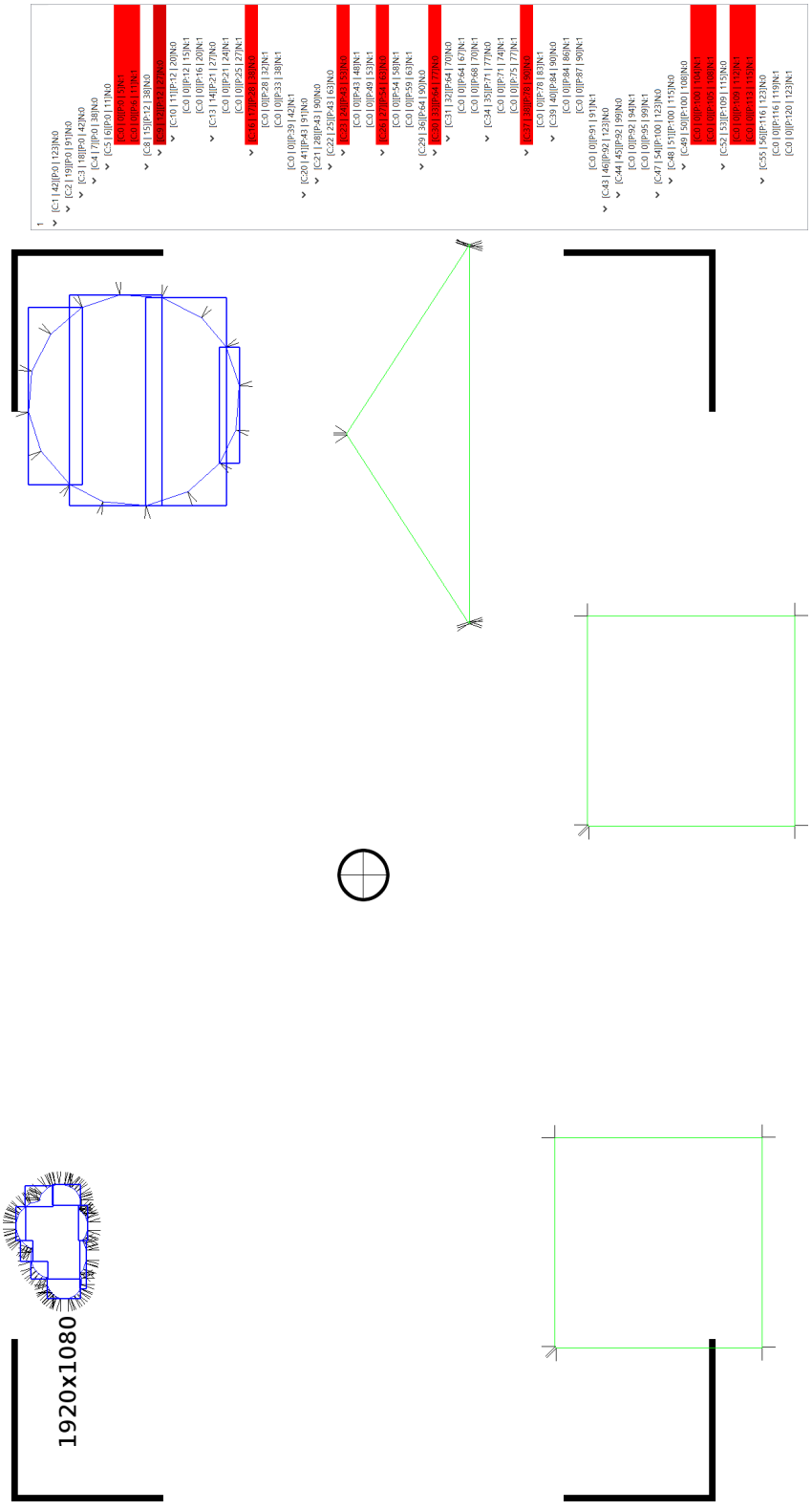


Abbildung A.8: Level 6

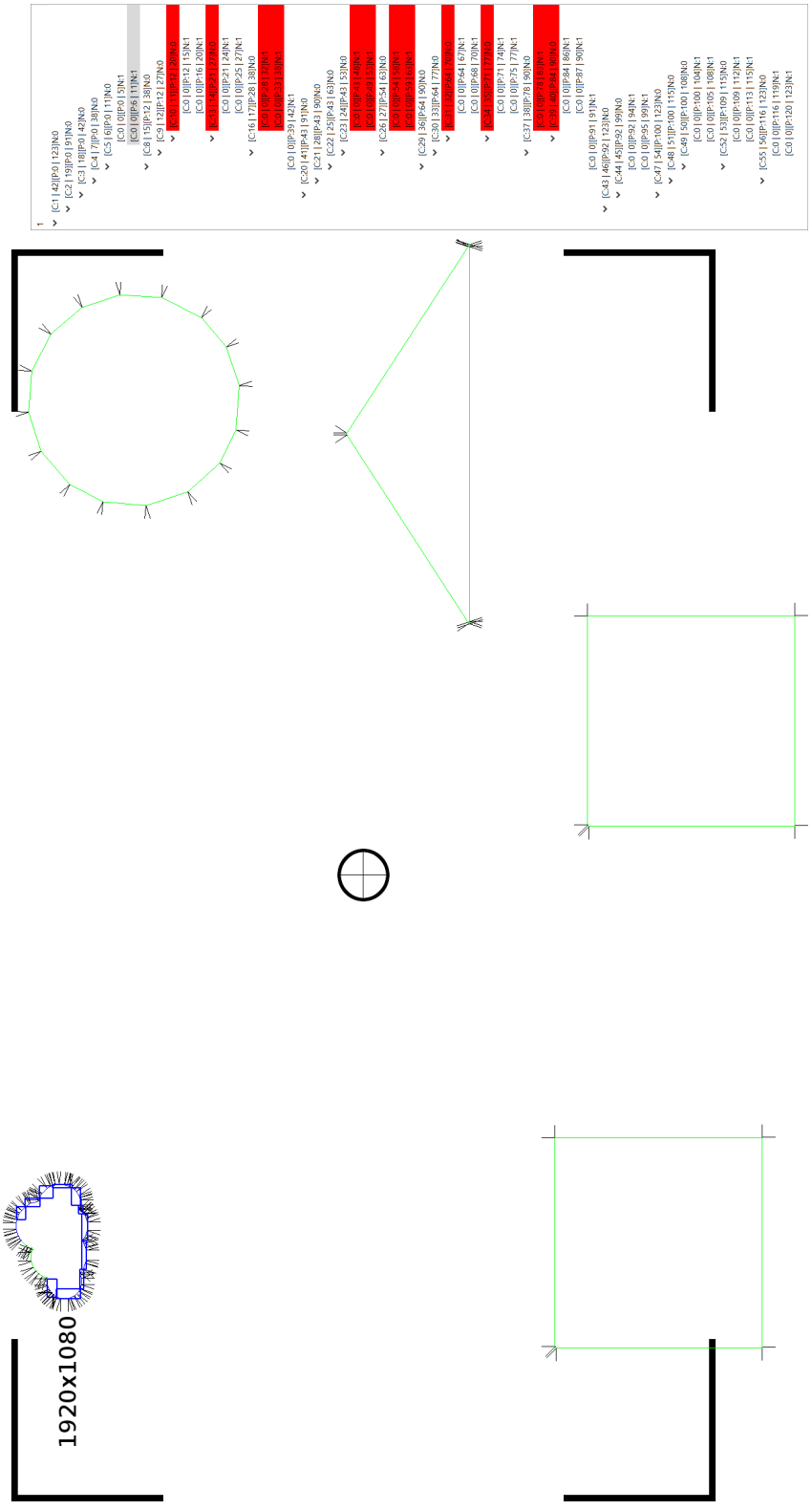


Abbildung A.9: Level 7

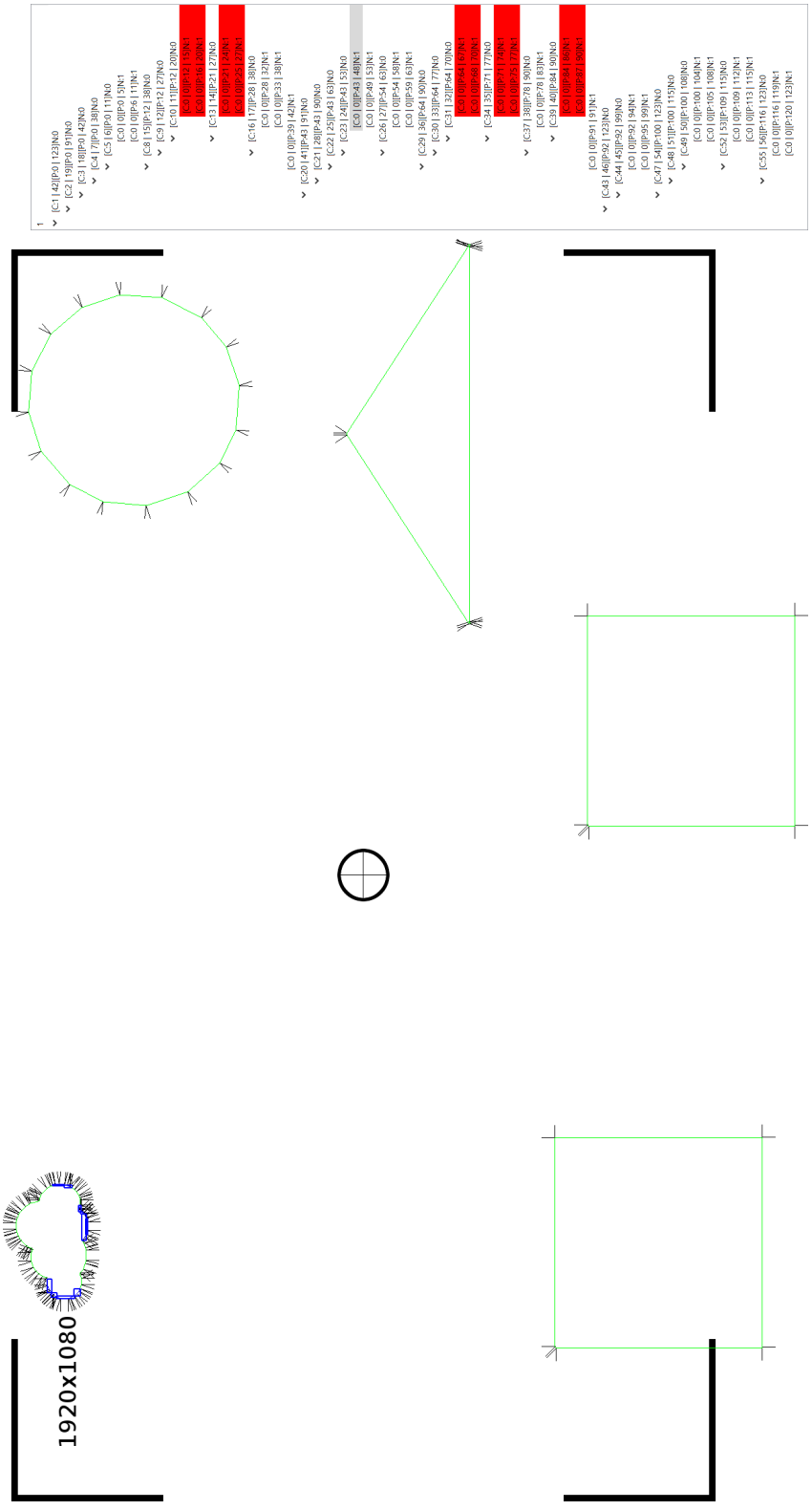


Abbildung A.10: Level 8

A.2 OpenGL Anti Aliasing Probleme

Die Testszenen sind hier absichtlich ohne diffuses Licht und nur mit einer Punktlichtquelle (ohne Monte-Carlo-Sampling) und vielen gerichteten Reflexionen entworfen, um möglichst viele Aliasing Effekte zu verstärken und zu provozieren. Regionen, in denen Aliasing Effekte gut zu beobachten sind, sind rot markiert.

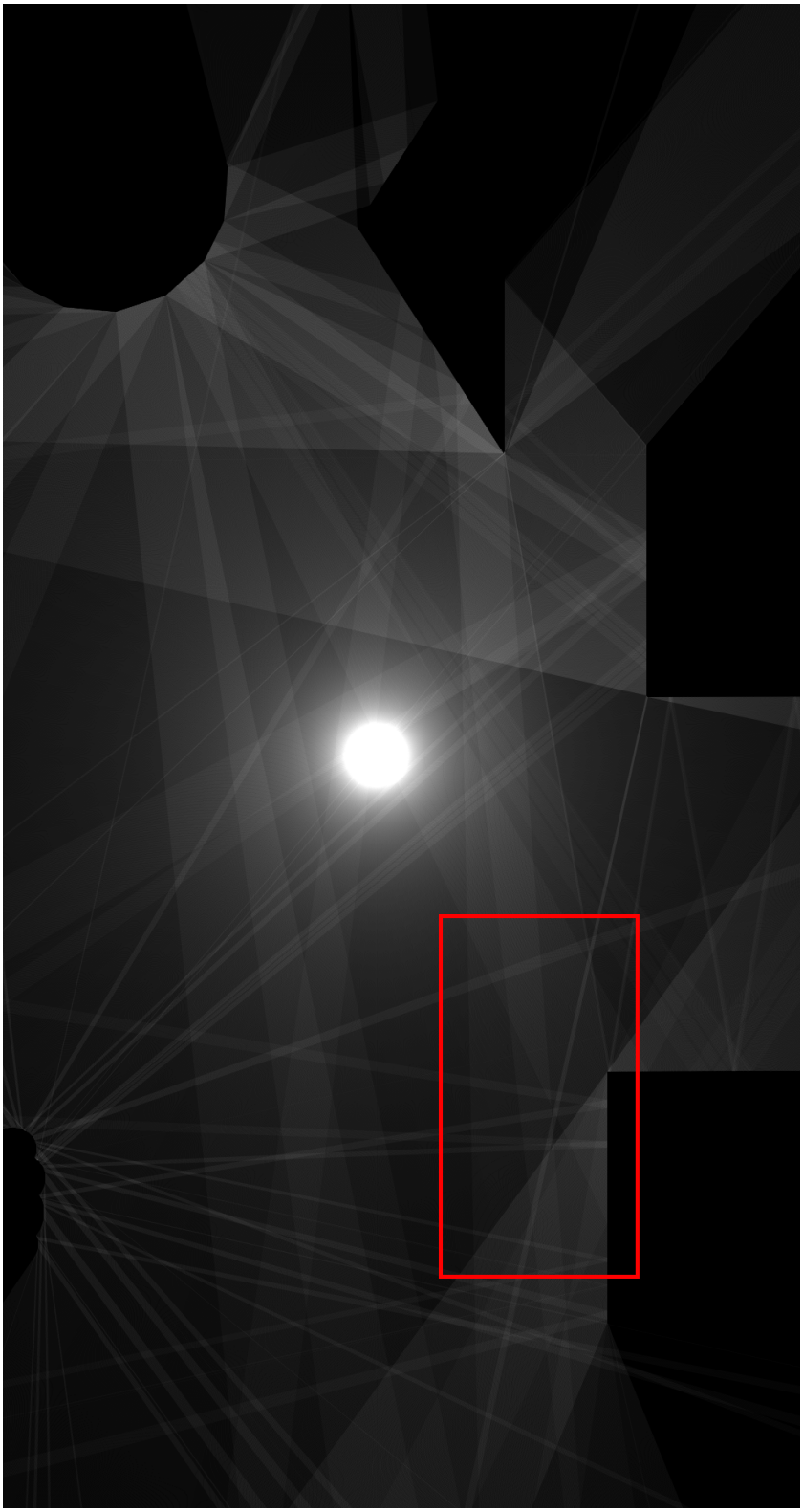
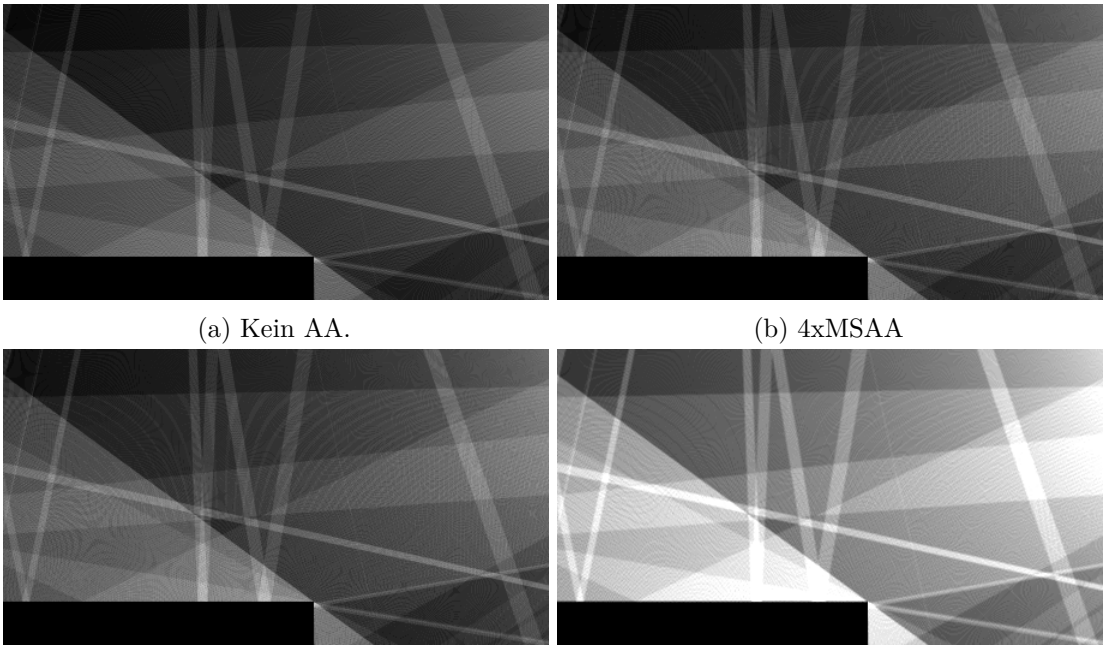


Abbildung A.11: Ohne AA, roter Ausschnitt s.u.



(a) Kein AA. (b) 4xMSAA

(c) 16xMSAA (d) GL_LINE_SMOOTH

Abbildung A.12: Verschiede Antialiasing Methoden, Kontrast und Helligkeit bei allen um den selben Faktor stark erhöht

A.3 Render Ergebnisse und Szenen

Beispiel wie die Lightmaps verwendet werden können. Das verwendeten Texturen stammen von *https://pixabay.com/*[1], wurden aber teilweise verändert.

A.3.1 Manuelle Bildsynthese

Es werden zuerst Vorder- und Mittelgrundtexturen gelayert und in eine Textur vereinigt. Auf diese wird dann mit der Lightmap multipliziert. Der Hintergrund wird dann dahinter gelegt.

[1] *https://pixabay.com/service/license/*

Abbildung A.13: Diffuse Volumenlichtquelle erzeugt diffusen Schatten hinter Objekt, Rendern der Lightmap ca. 4632 ms, da die diffuse Lichtquelle mit sehr hohem Sampling gerendert wurde.

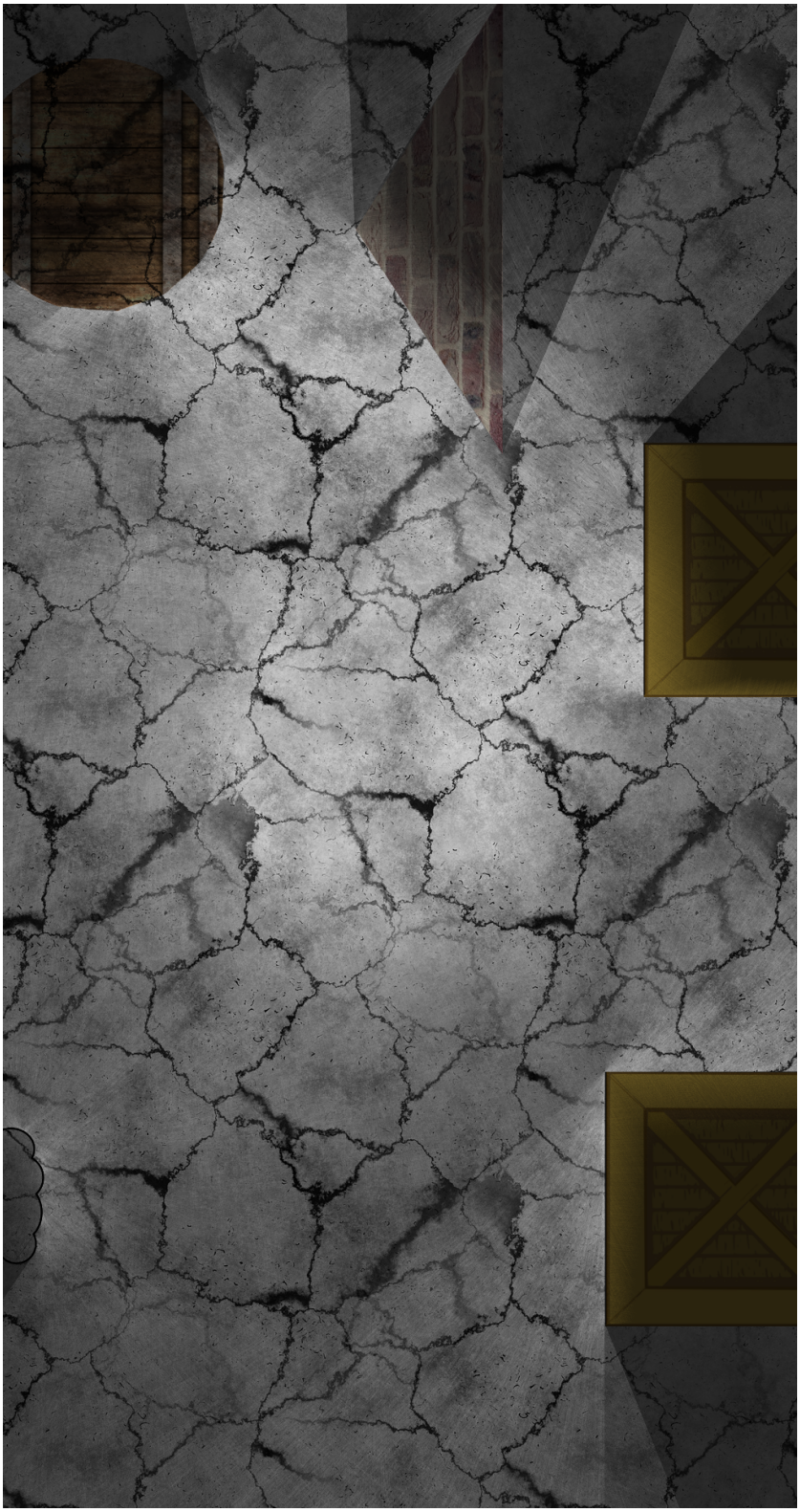


Abbildung A.14: Punktlichtquellen umgeben von Objekten welche mit einem diffusen Faktor von 0.3 reflektieren. Raytracing der Szene 415 m s, Rastern der Lightmap: 22 m s, Gesamt 437 m s

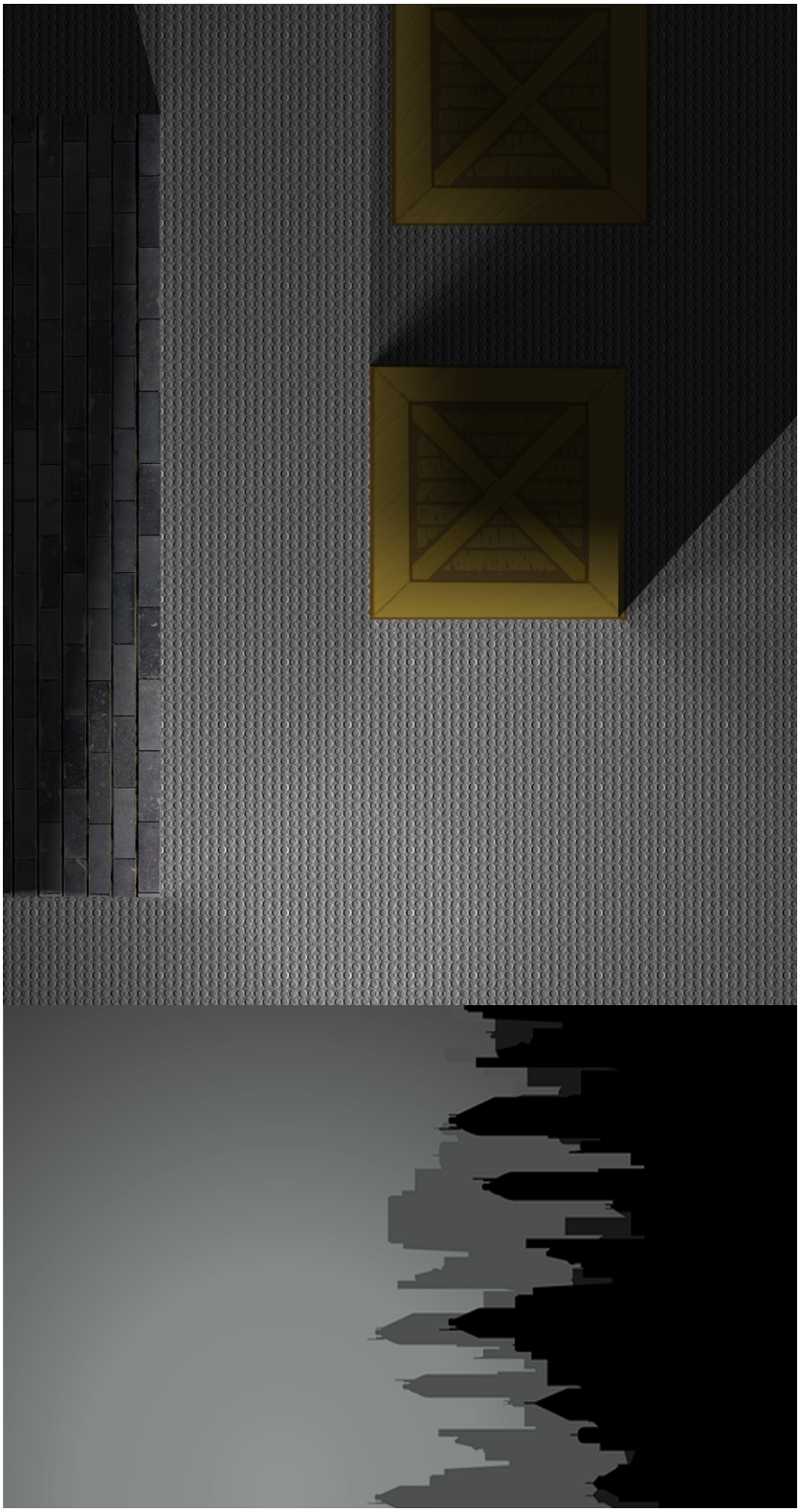


Abbildung A.15: Indirekte Beleuchtung. Die hintere Kiste wird durch die diffuse Reflexion des Lichtes an der Decke beleuchtet. Der Hintergrund (die Stadt), wird nicht beleuchtet. Raytracing der Szene 382 m s, Rastern der Lightmap: 21 m s, Gesamt 403 m s

Abbildung A.16: Szene mit rein gerichteten Reflexionen. Lichtstrahl wird zwischen den ersten 3 Wänden reflektiert und an der vierten absorbiert. Raytracing der Szene 11 m s, Rastern der Lightmap: 4 m s, Gesamt 15 m s

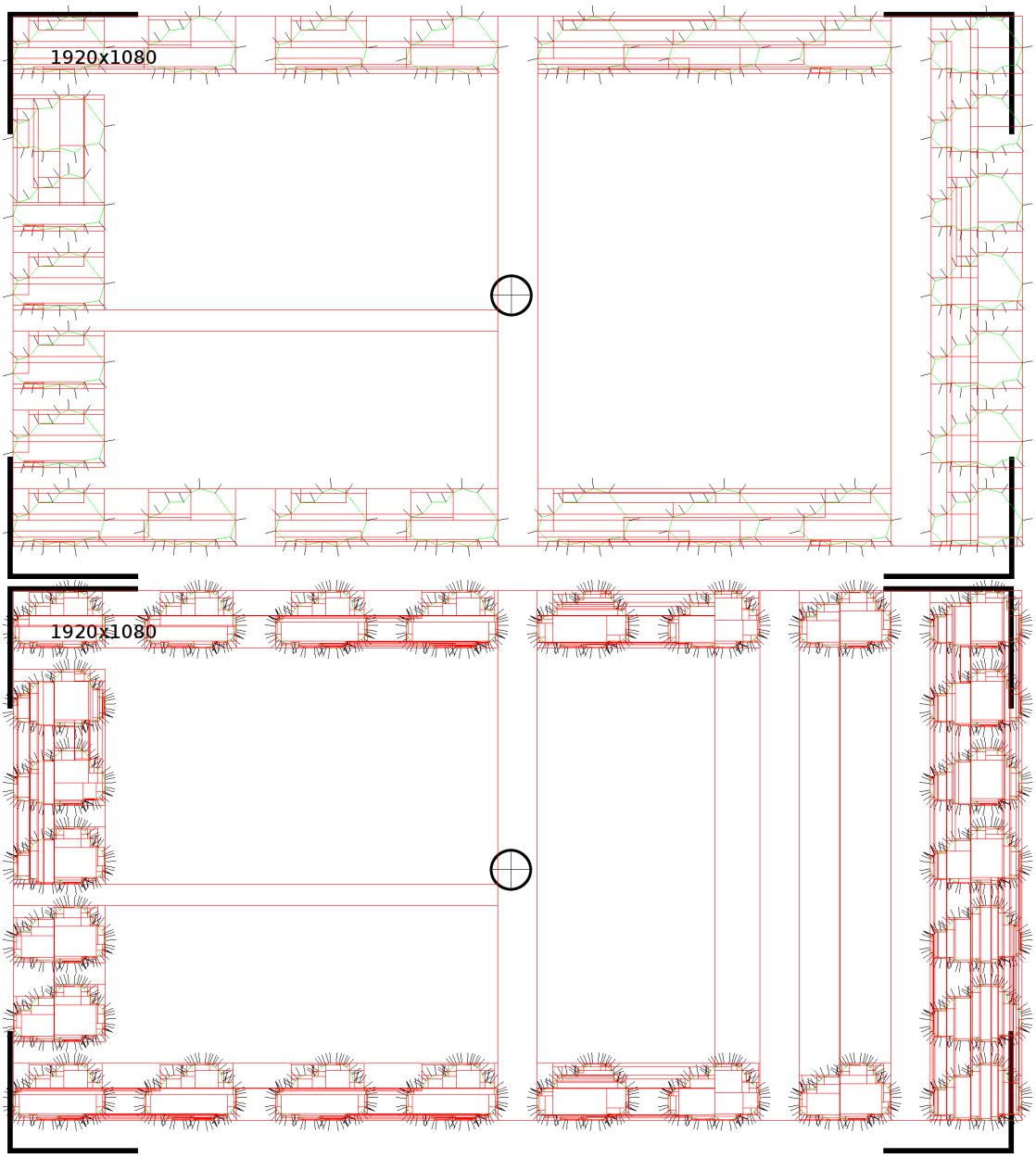
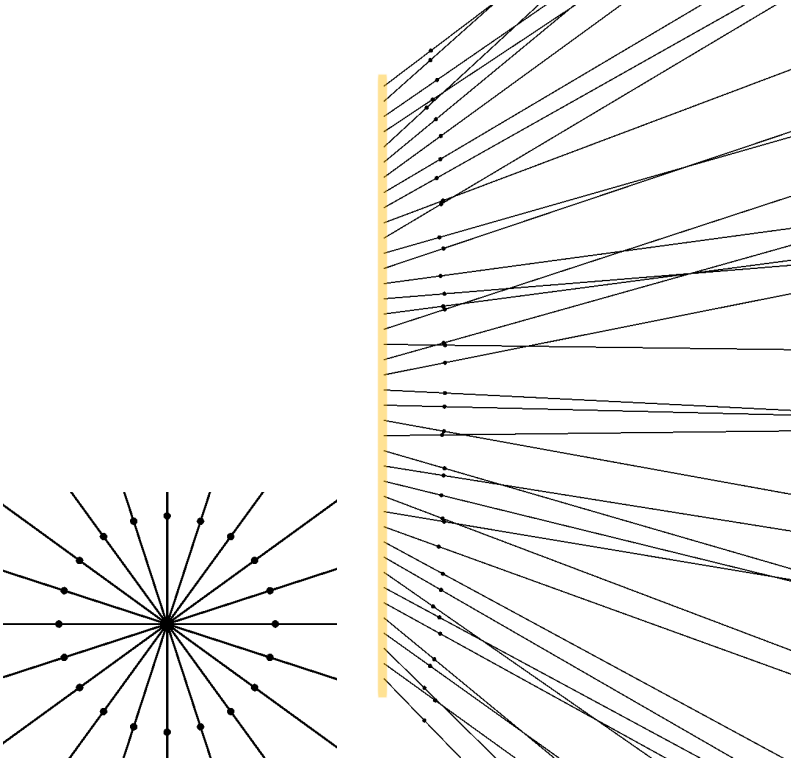


Abbildung A.17: BVH Benchmark: Szenen welche zur Performance Messung der BVH verwendet wurden

A.4 Platzierung von Strahlen durch Lichtquellen



(a) Verteilung der Strahlen einer Punktlichtquelle. Alle Strahlen kommen von einem gemeinsamen Ursprung.

(b) Verteilung der Strahlen einer Stablichtquelle. Der Ursprung wurde nachträglich gelb Markiert. Der Abstrahlwinkel äußert sich darin, das der Winkel der Strahlen nach außen größer wird. Die Diffusion in der zufälligen Richtung der Strahlen.

Abbildung A.18: Strahlen von Lichtquellen

A.5 OpenCV

Generieren von Objekt CSV Dateien mithilfe von OpenCV.

Extraktion des Alphakanal Der Alphakanal gibt an, wie durchsichtig ein Bild an einer Stelle ist. Bereiche, die komplett durchsichtig sind, können also entfernt werden.

Deswegen wird der Algorithmus für das ganze Bild auf dem Alphakanal durchgeführt.

Anwenden des Canny Filters Säubern der Kanten mir dem Canny Algorithmus.

Extraktion der Kanten Die gesäuberte Kannte wird nun extrahiert. Dann wird sie vergröbert. Dies ist aus zwei Gründen von Vorteil. Erstens, das gröbere Polygone die Leistung steigern, weil sich dadurch weniger Kanten in der BVH befinden. Zweitens, da so teilweise sinnvollere Geometrie Informationen gewinnen lassen. Die Kontur besteht zuerst nur aus Pixeln, welche alle nur einen Winkel von ± 0, $\pm\frac{1}{2}\pi$ oder $\pm\frac{1}{4}pi$ zueinander haben. Erst durch Vergröberung entstehen Winkel zwischen den Konturpunkten, welche die Gesamtform des Objektes besser beschreiben.

Export als CSV Das so aus dem Umriss generierte Polygon wird als CSV exportiert.

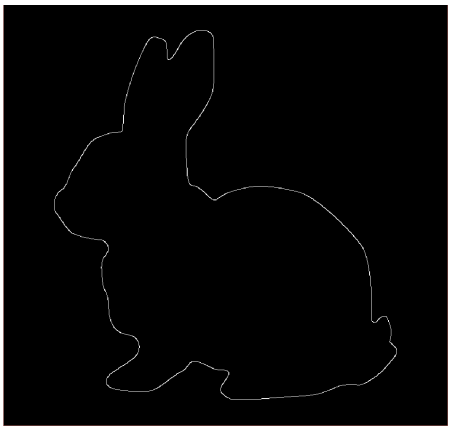
Zurechtschneiden der Rastergrafik Die Vergröberung ändert Form und Fläche des Polygons im Vergleich zur ursprünglichen Rastergrafik. Diese muss nun manuell angepasst werden. Um dies zu erleichtern, wird das gefüllte Polygon als Rastergrafik ausgegeben.

Tabelle A.1: OpenCV Verarbeitungschritte

Nr	Ergebnis	Schritt
	C++ OpenCV code	
1		Extrahieren des Alphakanal aus dem Bild .

```
1 Mat channels[4]; //create image for each channel
2 split(img, channels); //split each channel into image
3 Mat alpha = channels[3]; //get alpha channel
```

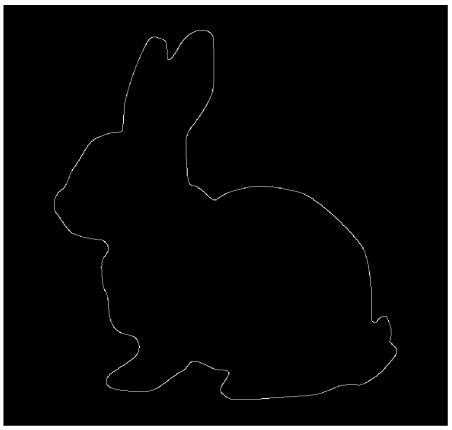
2



Anwenden des Canny Filters auf den Alphakanal und Extrahieren der Kanten. Verhältnis zwischen Upper und Lower Threshold beim Canny ist wie empfohlen 2:1. [2]

```
1 Mat alphaCanny; //output
2 //treshold values aus Slider
3 Canny(alpha, alphaCanny, slider, slider*2);
```

3



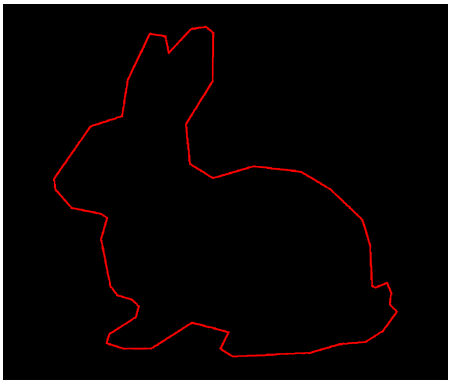
Anwenden des The-Chin chain aproximation Algorithmus (CHAIN_APPROX_TC89_KCOS, Zeile 5) [3]. OpenCV findet immer alle Konturen in einem Bild und ordnet diese von innen nach außen. Deswegen wird RETR_EXTERNAL (Zeile 5) als Parameter übergeben. Dies bewirkt, das nur äußere Kannten gewählt werden.

[2] *https://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/canny_detector/canny_detector.html*, abgerufen am 06.07.2019

[3] *https://docs.opencv.org/2.4/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html#tehchin89*, abgerufen am abgerufen 21.07.2019

```
1 vector<vector<Point> > contours;
2 Mat contourImg; //output
3 findContours(alphaCanny, contours,
4               RETR_EXTERNAL, CHAIN_APPROX_TC89_KCOS);
```

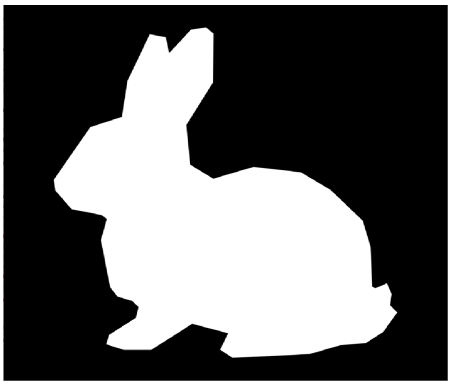
4



Vergröberung der Linie mit dem Ramer-Douglas-Peucker Algorithmus, welche so durchgeführt wird, das dabei ε (Zeile 5) höchstens die maximale Abweichung von der Ursprungskontur ist. ε kann über einen Regler eingestellt werden. 0.001 ist Skalierung für den Regler (Werte 0 bis 100) und limitiert ε auf 10% der Kontur Länge (Zeile 3). Die Konturlänge macht den einstellbaren Regler unabhängig von der Größe der zu bearbeitenden Kontur.

```
1 vector<Point> approx; //approximation of the original contour
2 Mat polyImg; //output
3 double contourLength = arcLength(cnt, true); //calc cnt length
4 double epsilon = slider * 0.0001 * contourLength;
5 approxPolyDP(cnt, approx, epsilon, true);
6 polylines(polyImg, approx, true, { 0,0,255 }); //draw approx
```

5



Zeichnen der Kontur als weiß gefülltes Polygon mit Antialiasing.

```
1 Mat cutOut; //output
2 fillPoly(cutOut, vector<vector<Point>>{approx}, WHITE, LINE_AA);
```

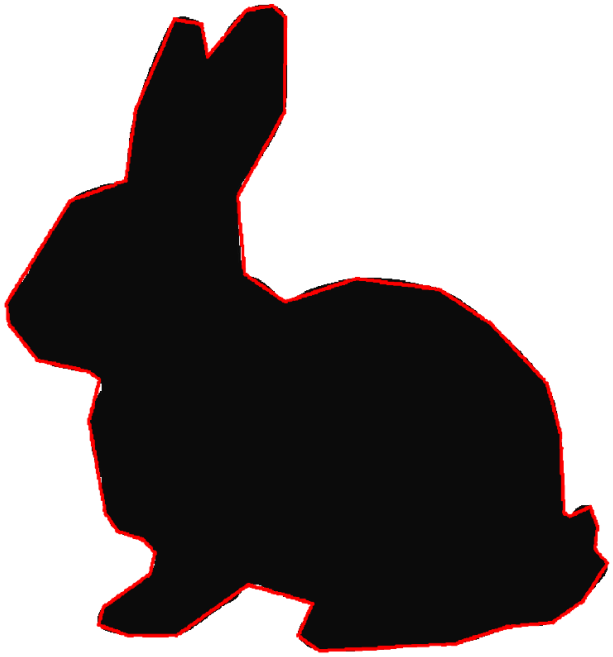


Abbildung A.19: Vergleich der gefundenen Kontur (Schwarz) mit der vergröberten Kontur (Rot) (mit $\varepsilon = 19 * 0.001 * l_{Kontur}$), weclche als CSV exportiert wrird.

Glossar

Bestrahlungsstaerke Ist die gesamte Leistung an Strahlung/Licht, welche auf eine Oberfläche trifft, bezogen auf deren Fläche..

Blattknoten Knoten, welche keine nachfolgenden Knoten in einer Baum Datenstruktur hat.

Blinn-Phong-Modell lokales Beleuchtungsmodell.

Cubemap Textur, welche aus 6 Texturen auf den Seiten eines Würfels besteht.

Delegate Werden Funktionen von einer Klasse zu einer Memberklasse durchgeleitet, so spricht man von Delegation.

Fourier-Transformation Umwandlung in den Frequenzbereich.

Integral Ordnet einer Funktion eine Zahl zu.

Kanal Trägt die Farbinformationen einer Farbe.

Kindknoten Knoten, welcher auf einen anderen Knoten in einer Baum Datenstruktur folgt.

Mobile Games Spiele, welche auf Smartphones und Tablets laufen.

Normale Abb. 2.3.

normieren Vektor auf Länge 1 bringen ohne dabei die Richtung zu ändern, in die er zeigt.

Pfad engl. Path, besteht aus Ursprung, Endpunkt und Farbwerten.

Polarkoordinaten Koordinatensystem mit Winkel und Abstand vom Ursprung.

Primitive Einfachstes Grafikelement, in der 3D Grafik meist Dreiecke, in 2D Linien, bzw. Polygon Kanten.

Profiling Analyse des Laufzeitverhaltens einer Software.

Quantor Gibt in einem regulären Ausdruck an, wie häufig ein Zeichen/Muster wiederholt werden muss/kann.

Rasterisierung Umwandlung von Primitiven in Pixel.

Rendering Pipeline auch Grafikpipeline, beschreibt die Schritte die notwendig sind, um aus einer Szene ein Bild auf Bildschirm zu erzeugen.

Root Node Wurzelknoten.

Schwellwert Wert, ab dessen überschreitung sich die Verarbeitung der Daten ändert.

Strahl engl. Ray, besteht aus Ursprung, normierter Richtung und Farbwerten.

Strahlungsdichte Die Strahldichte beschreibt die orts- und richtungsabhängigkeit einer von einer Fläche abgegebenen Strahlungsfluss..

Strahlungsfluss Beschreibt die Energie pro Zeit (Leistung) der Strahlung/Lichts..

Varianz Bezeichnet die Streuung von Werten.

Vertexbuffer Speichert Vertecies.

Volumenstreuung Streuung des Lichtes, welche nicht nur an der Oberfläche stattfindet, sondern auch im Inneren des Objektes.

Wurzelknoten Knoten am Begin einer Baum Datenstruktur.

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Ort	Datum	Unterschrift im Original