# Computergrafik

## RayTracing-Algorithmus

Benotete Abgabe

von

8978851

9482203

und 9741250

Abgabedatum:10.01.2021

| | |
|---|---|
| Dozent | Johannes Riester |
| Bearbeitungszeitraum | 6 Wochen |
| Sourcecode | www.github.com/FreshMax9000/RayTrace |

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# 1 Theoretische Prinzipien

## 1.1 Grundlagen Raytracing

Beim Raytracing handelt es sich um eine Alternative zur Rasterung, der vorherrschenden Methode zum Darstellen von Objekten im Sichtfeld des Betrachters. Dabei wird durch Operationen der Linearen Algebra eine dreidimensionale Szenerie auf einen zweidimensionalen Bildschirm geworfen, wobei die Positionierung des Beobachters im Verhältnis zum Bildschirm und den Objekten in der Szenerie eine Rolle spielt.[1]
Bei dem Raytracing-Verfahren spielt vor allem die Berechnung von Licht, Reflektionen und Schatten eine Rolle. Hier werden vom Beobachter aus Strahlen durch den Bildschirm in die Szenerie geworfen und weiterverfolgt. Abhängig davon, wo diese auftreffen, inwiefern diese reflektieret werden und, wie die Fläche auf die sie auftreffen im Verhältnis zum Licht steht wird dann bestimmt, wie der erste getroffene Punkt beleuchtet wird/welche Farben dieser hat etc. Das Raytracing Verfahren steht dabei konträr zur Realität. Denn tatsächlich treffen Strahlen von Lichtquellen, wie der Sonne oder Lampen zuerst auf Gegenständen auf und fallen dann in unser Auge. Sie gehen also natürlicherweise von der Lichtquelle aus. Beim Raytracing werden allerdings nur die relevanten Strahlen betrachtet, da aus der Perspektive des Betrachters aus nachvollzogen wird, welche Strahlen ins Auge fallen. Das spart Rechenleistung, ist aber offensichtlich auch nicht so realitätsnah, wie die wirkliche Welt.[2]
In der Praxis kann man sich die Realisation von Raytracing grob wie folgt vorstellen:
Aus der Kamera heraus, die eine bestimmte Position im Raum einnimmt, werden Strahlen durch jeden Pixel auf einem Bildschirm geschossen. Dadurch entsteht eine bestimmte Perspektive. Auf der anderen Seite des Bildschirms ist die Szenerie aufgebaut, die man nachvollziehen möchte. Für jeden einzelnen Strahl wird berechnet wo dieser als erstes auftrifft und wie er z.B. potenziell weiterreflektiert wird. Da manche Effekte von Licht aus der Realität viel zu aufwändig zu berechnen sind wird auf Annäherung und Vereinfachungen zurückgegriffen. So z.B. mit dem Phong-Modell, welches in 1.2 näher beschrieben ist. Dort wird z.B. die grundlegende Umgebungsbeleuchtung

[1] (Wegscheider, 2009/2010, S. 3)

[2] (Wegscheider, 2009/2010, S. 3-5)

festgelegt. Aus verschiedenen Zahlenwerten, die für die getroffene Oberfläche und die Lichtwerte gelten und weitere Berechnungen für Reflektion und z.B. das Verhältnis zwischen Lichtquelle und Oberfläche etc. werden einzelne Werte von Farben errechnet, welche auf den anfangs im Bildschirm getroffenen Pixel angewendet werden.
Das Verfahren an sich ist sehr aufwändig, da je nach Pixel sehr viele Strahlen, Kollisionen etc. berechnet werden, vor allem, wenn auch noch hochdetaillierte Schatten ins Spiel kommen. Genau deswegen hat es bisher nur Anwendung im Bereich von z.B. Animationen, wie Filmen, gefunden. Eben in Anwendungen, die keine Echtzeitanforderungen haben. Mit dem Fortschritt der Technik schafft es Raytracing aber auch langsam in die Welt der Computerspiele.

## 1.2 Phong

Beim Phong-Beleuchtungsmodell wird die Beleuchtung eines Punktes im Raum aus den drei Werten, “Ambient”, “Diffuse” und “Specular, zusammengesetzt. Dabei wird nicht das Ziel verfolgt, die Realität möglichst realistisch abzubilden, sondern ein einfaches und schnell zu berechnendes Verfahren zu verwenden.

### 1.2.1 Ambient Beleuchtungswert

Der “Umgebungswert” eines Punkts ist nur abhängig von dem Ambient Wert der Oberfläche und dem Ambient Wert der Lichtquelle, bzw. der Lichtquellen. Er stellt somit eine konstante Beleuchtung des gesamten Raums dar, unabhängig von der Position der Lichtquelle oder des Beobachters.

$$Beleuchtung_{ambient} = Objekt_{ambient} * Lichtquelle_{ambient}$$

In Abbildung 1 sieht man ein Bild was ausschließlich mit ambienter Beleuchtung erzeugt wurde. Alle Wände sind gleichmäßig beleuchtet.

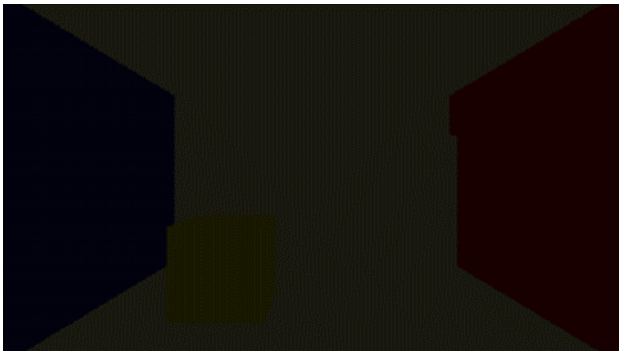


**Abbildung 1: Bild mit Umgebungsbeleuchtung**

### 1.2.2 Diffuse-Beleuchtungswert

Der diffuse Beleuchtungswert ist abhängig von den „Diffuse“-Werten der Lichtquelle , dem Objekt, sowie dem Winkel der Oberfläche zur Lichtquelle. Steht der Normalenvektor der Oberfläche des Objekts auf dem Vektor zur Lichtquelle ist der diffuse Wert maximal.

$$Beleuchtung_{diffuse} = Objekt_{diffuse} * Lichtquelle_{diffuse} * (Richtung_{licht} \cdot Normalenvektor_{oberfläche})$$

In Abbildung 2 ist ein Bild zu sehen, welches nur durch diffuses Licht beleuchtet wurde. Die Flächen sind jetzt unterschiedlich beleuchtet und man sieht auf der dem Licht abgewandten Seite der Quader kein Licht, da das Skalarprodukt aus der Richtung des Lichts und des Normalenvektors der Oberfläche <= 0 ist.

Da der „Ambient“- und der „Diffuse“-Wert nicht von der Position des Betrachters abhängen, kann man diese Werte bei einer Echtzeitanwendung im Voraus berechnen.

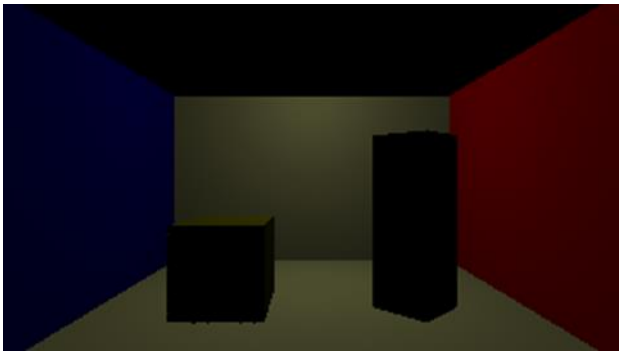


**Abbildung 2: Bild mit diffuser Beleuchtung**

### 1.2.3 Specular Beleuchtungswert

Der „Specular“-Beleuchtungswert soll spiegelnde, besonders helle Punkte einer Oberfläche betonen, in denen der Winkel zum Licht und zum Betrachter von der Oberflächennormale relativ ähnlich sind. Zur Berechnung wird die optimale Reflektionsachse bestimmt und mit der realen Reflektionsachse (der Oberflächennormale) verglichen.

$$Reflektionsachse_{optimal} = Richtung_{licht} + Richtung_{betrachter}$$

Je näher sich diese Werte sind (also je größer das Skalarprodukt zwischen den beiden Werten), desto höher ist der Specular-Beleuchtungswert für diesen Punkt. Dieses Skalarprodukt wird dann noch mit einem Wert „shinyness“ potenziert. Durch einen höheren „shinyness“-Wert erhält man eine Fläche die glänzender, bzw. Spiegelnder erscheint, während ein niedriger Wert zu einer eher matten Oberfläche führt.

$$Beleuchtung_{specular} = Objekt_{specular} * Lichtquelle_{specular} * \left(Reflektionsachse_{optimal} \cdot Reflektionsachse_{real}\right)^{shinyness}$$

Abbildung 3 zeigt das Bild nur mit „Specular“-Beleuchtung. Besonders an der Lichtquelle kann man die „harten“ Reflektionen gut sehen.

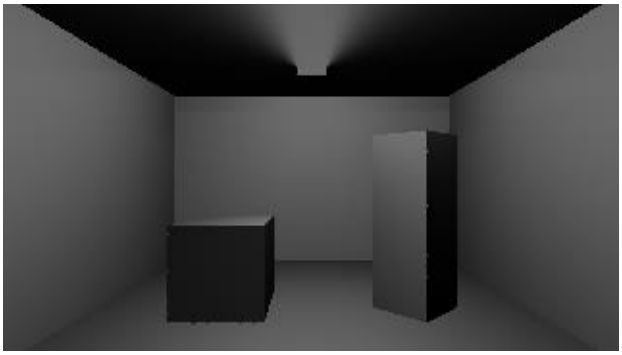


**Abbildung 3: Bild mit „Specular“-Beleuchtung**

### 1.2.4 Zusammensetzung der Beleuchtungswerte

Für jeden Punkt werden die drei Werte addiert. Dadurch ergibt sich das Bild in Abbildung 4. Durch das Anpassen von den Parametern kann das Bild entsprechend verändert werden.[3]

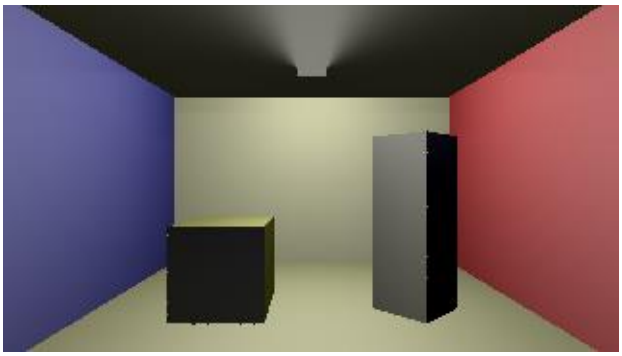


**Abbildung 4: Zusammengesetztes Bild**

[3] (Phong, 1975); (Aflak, 2020)

# 2 Algorithmische Umsetzung

## 2.1 Kollisionserkennung von Rays mit Objekten

Bei einem der zentralsten Funktionen in einem Raytracing-Algorithmus handelt es sich um die Kollisionserkennung. Diese ist notwendig um bei den verschiedenen Strahlen deren Weg nachvollzogen werden soll, nachzuvollziehen, wo diese als erstes auftreffen. Eine Grundlage für die Anwendung einer solchen Funktion ist das Verständnis von linearer Algebra und das Verständnis für den dreidimensionalen Raum und vor allem die Implementierung der einzelnen Konzepte, wie die eines Strahles oder die Darstellung von Objekten im Raum. Näher wird hierauf in Abschnitt 3.1 eingegangen. An dieser Stelle wird auf die Implementierung von Strahlen und die von Objekten bzw. Flächen eingegangen.

Ein Strahl besteht immer aus einem Ursprung und einem Richtungsvektor. Wandert man vom Ursprung aus entlang des Richtungsvektors erhält man alle Punkt, die vom Strahl erreicht werden können. Da der Richtungsvektor eben nur die Richtung anzeigt, sollte er normalisiert werden. Das bedeutet, dass die Länge des Vektors eins ergibt, er aber trotzdem weiterhin in die gleiche Richtung zeigt. So lässt sich auch später die Distanz, die der Strahl zurücklegen muss, ganz einfach und ohne weitere Berechnung bestimmen.

Je nach Verwendungen von darzustellenden Objekten müssen auch Möglichkeiten implementiert werden, Schnittpunkte mit diesen auszurechnen. In unserem Beispiel lassen sich alle Objekte durch Rechtecke darstellen. Egal ob Wände des Raums oder die einzelnen Flächen eines Würfels. Alle Objekte sind eigentlich durch zweidimensionale Flächen im Raum implementiert. Bei einem Würfel liegen dann sechs einzeln definierte Flächen im Raum, die aneinander anschließen. Vereint sind sie durch einzelne Eigenschaften, die für das gesamte Objekt gelten, vgl. Phong Modell aus Abschnitt 1.2. Die einzelnen Flächen wiederrum sind definiert durch einen Stützvektor, also einen Punkt im Raum und zwei direktionale Vektoren, die die Fläche aufspannen. Dabei haben in diesem Fall die direktionalen Vektoren eine bestimmte Länge. Diese entspricht der Seitenlänge des Rechtecks. Verbindet man jetzt die zwei direktionalen Vektoren miteinander, die vom Stützvektor ausgehen, indem man für die jeweiligen Vektoren auf

der Spitze des anderen direktionalen Vektors parallel den gleichen Vektor zeichnet erhält man eine rechteckige Fläche, wie zu sehen in Abbildung 5.

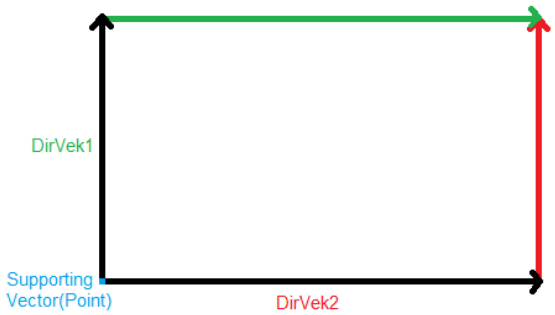



**Abbildung 5: Skizze von Rechteck im Raum**

Nun soll anhand dieser Gegebenheiten im Raum errechnet werden, wo mögliche Schnittpunkte zwischen Strahlen und den Flächen liegen und welcher der möglichen Schnittpunkte der erste ist, also welche Fläche als erstes geschnitten wird.
Hierzu ist ein lineares Gleichungssystem notwendig.

$$S_1 + xD_1 + yE_1 = O_1 + zN_1$$
$$S_2 + xD_2 + yE_2 = O_2 + zN_2$$
$$S_3 + xD_3 + yE_3 = O_3 + zN_3$$

$mit\ S_{tützvektor}, D_{irektionalvektor1}, E_{direktionalvektor2}, O_{Ursprung\ Strahl}, N_{direktionalvekor\ Strahl}$

Mit dem obigen Gleichungssystem lassen sich die einzelnen Koordinatenteile, also x/y/z senkrecht und jeweils für die Fläche links und den Strahl rechts berechnen. So erhält man ein lineares Gleichungssystem dritter Ordnung. X steht hierbei für die Ausdehnung des ersten direktionalen Vektors und Y für die des zweiten. Mit Y lässt sich die Ausdehnung des normalisierten Vektors des Strahls berechnen. Ist das Gleichungssystem lösbar so gibt es einen Schnittpunkt. Mit X und Y und dem Rest der Flächengleichung könnte man den Schnittpunkt errechnen. Aber auch mit Z und dem Rest der Gleichung für einen Strahl lässt sich dieser Punkt errechnen. Eine Besonderheit hier: Aufgrund der Normalisierung des Richtungsvektor des Strahls gibt Z zudem die Distanz zwischen Ursprung des Strahls und der Fläche an.
Wie genau diese Formel implementiert wurde und zu welchen Problemen die Kollisionserkennung führt, ist näher in Abschnitt 2.1 beschrieben.

## 2.2 Quadratische Lichtquelle

Da eine quadratische Lichtquelle gefordert ist, aber die in Kapitel 1 beschriebene Phong-Methode nur mit eindimensionalen Lichtquellen arbeitet, stellt das Berechnen der Beleuchtung eines Punktes ein besonderes Problem dar. Zusätzlich ist der einfache Ansatz zur Schattenbestimmung eines Punktes durch Werfen eines Strahls zur (punktförmigen) Lichtquelle, bedingt durch die zweidimensionale Ausdehnung der Lichtquelle, nicht möglich. Um weiche Schatten zu erhalten muss also ein anderer Ansatz gewählt werden.

### 2.2.1 Berechnung der Schatten

Zur Berechnung der Schatten wird ein primitiver Algorithmus verwendet, der die Schattenbestimmung von einer eindimensionalen Lichtquelle auf eine mehrdimensionale Lichtquelle portiert. Statt zu überprüfen, ob zwischen zwei Punkten ein Objekt liegt, wird überprüft wie viele Strahlen ungestört von einem Punkt auf dem Objekt zu mehreren anderen Punkten auf dem Licht reichen. Es werden also von dem zu prüfenden Punkt mehrere „Schattenfühler“ (Shadow Rays) zur Lichtquelle geschickt und aus dem Anteil der angekommen Fühler ein Schattenwert für diesen Punkt berechnet.[4]

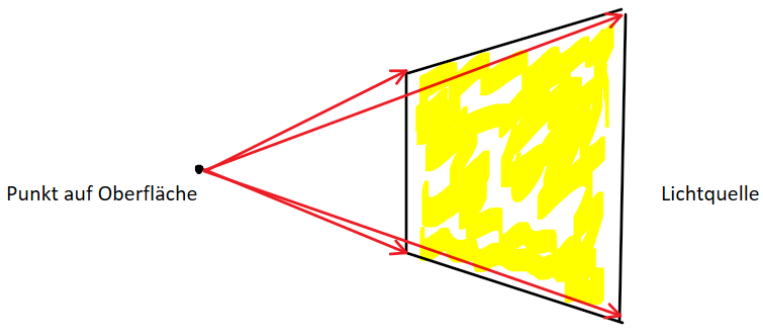



**Abbildung 6: Darstellung von 4 Shadow Rays (in rot)**

Diese Methode approximiert die Lichtfläche nur durch mehrere Punkte und stellt daher kein perfektes Ergebnis dar und nur stufenförmige Schatten (siehe Abbildung 8). Die Punkte, durch die die Lichtquelle repräsentiert wird, können dabei systematisch(siehe

[4] Die Idee dafür kam von folgendem Eintrag bei Stackoverflow: https://stackoverflow.com/questions/31709332/ray-tracing-soft-shadow

Abbildung 7 bis Abbildung 9), zufällig(siehe Abbildung 10) oder gemischt(siehe Abbildung 11) gewählt werden. Für den finalen Render wurden jedoch nur systematische Shadow Rays benutzt, da diese für ein schöneres Bild sorgen und zufällige Shadow Rays bei einer zu niedrigen Anzahl zu einem Rauschen führen (Siehe Abbildung 10 und Abbildung 11).

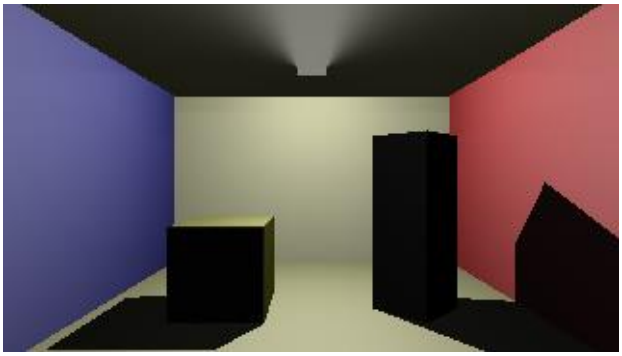


**Abbildung 7: Bild mit 1 systematischen Shadow Ray**

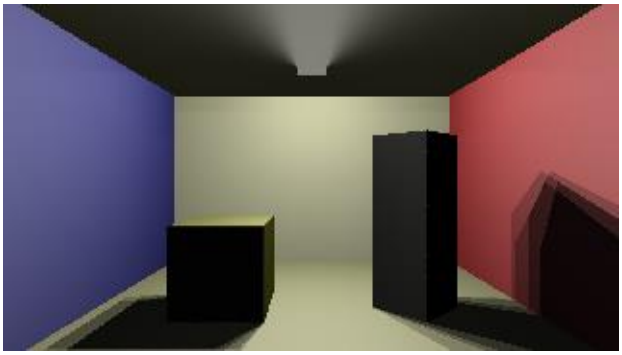


**Abbildung 8: Bild mit 4 systematischen Shadow Rays**

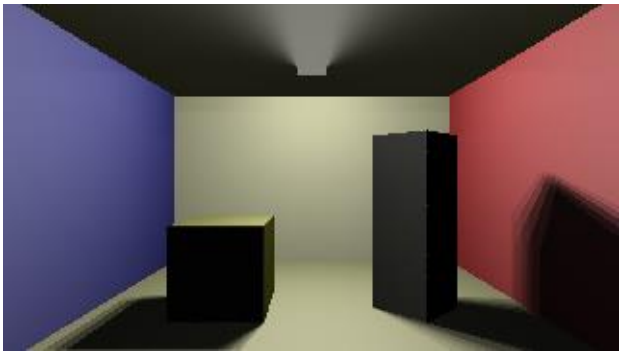


**Abbildung 9: Bild mit 9 systematischen Shadow Rays**

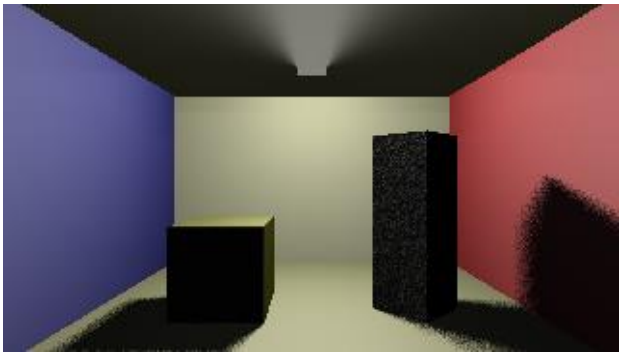


**Abbildung 10: Bild mit 8 zufälligen Shadow Rays**

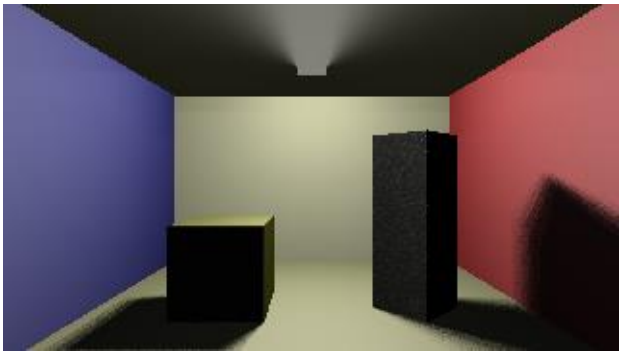


**Abbildung 11: Bild mit 9 systematischen und 8 zufälligen Shadow Rays**

### 2.2.2 Berechnung der Beleuchtung

Bei der Berechnung der Beleuchtung für eine quadratische Lichtquelle, nach dem Phong-Modell, ist nur der „Diffuse“- und „Specular“-Wert wichtig, da nur diese von der Position des Lichts abhängen. Um die in Kapitel 1.2 vorgestellten Formeln zu verwenden, muss daher ein Punkt auf der Lichtquelle als Lichtquelle angenommen werden. Der dafür entwickelte Algorithmus basiert darauf, für einen bestimmten Strahl (Ray) den optimalen Punkt auf der Lichtfläche zu finden.

Dieser initiale Ray ist bei der Berechnung des „Diffuse“-Werts der Normalenvektor der Oberfläche und bei der Berechnung des „Specular“-Werts ein von der Oberfläche reflektierter Ray, der ursprünglich der Kamera entspringt. Die Skizzen in diesem Kapitel sind, wenn möglich, zur Vereinfachung im zweidimensionalen Raum dargestellt.

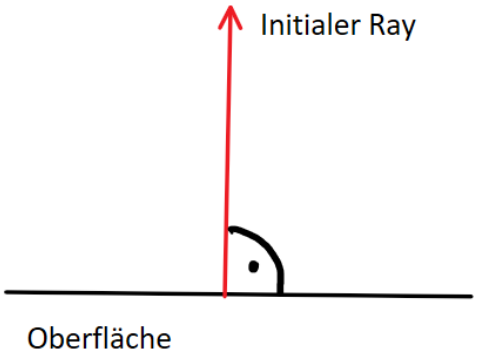



**Abbildung 12: Initialer Ray bei „Diffuse“-Berechnung (2D)**

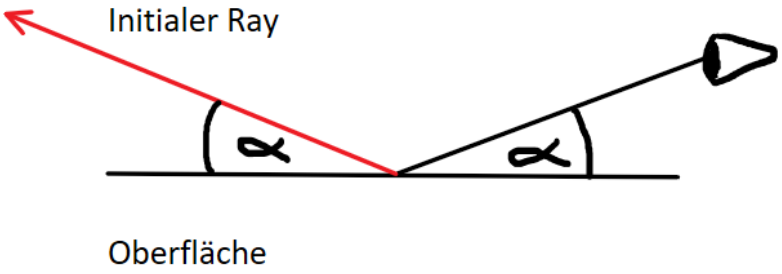



**Abbildung 13: Initialer Ray bei „Specular“-Berechnung (2D)**

Die anschließende Berechnung des besten Punkts auf der Lichtfläche erfolgt über folgenden Algorithmus:

1. Berechnung der Schnittstelle des Rays mit der Lichtebene. Ist der Ray parallel zur Lichtebene, wird ein sehr kleiner Offset hinzugefügt, um einen Schnittpunkt zu erzwingen.

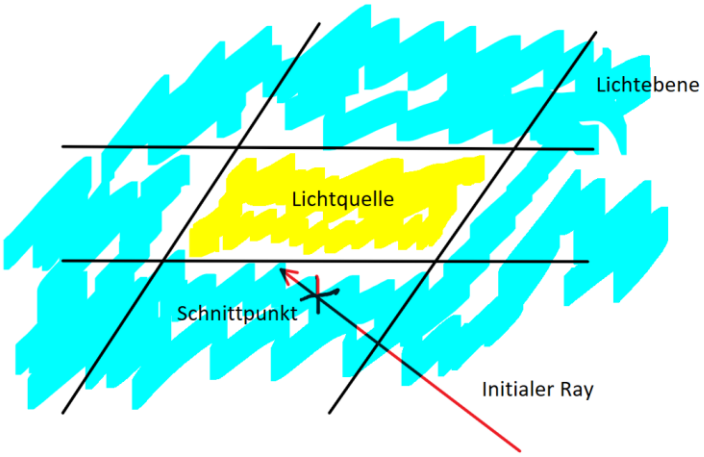



**Abbildung 14: (1) Schnittpunkt von initialem Ray und Lichtebene (3D)**

2. Übertragung des Schnittpunktes in den zweidimensionalen Raum. Als Basisvektoren der x- und y-Achse werden dabei die aufspannenden Vektoren der Lichtquelle verwendet.

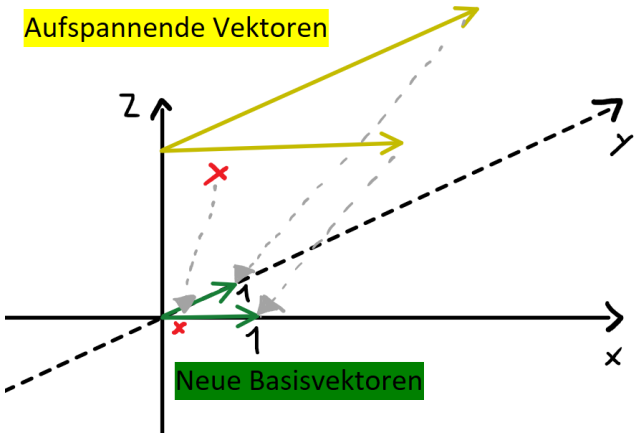



**Abbildung 15: (2) Übertragung des Schnittpunkts in den zweidimensionalen Raum**

3. Bestimmung des nächsten Punktes im zweidimensionalen Raum, der die Eigenschaft $0 \leq x, y \leq 1$ besitzt. Ist der in Punkt 1 berechnete Schnittpunkt „hinter" dem Ausgangspunkt des Rays, wird der entfernteste Punkt bestimmt. Dadurch, dass die Basisvektoren dieses kartesischen Raums sich aus den aufspannenden Vektoren der Lichtquelle befinden, ergibt sich, dass sich der ermittelte Punkt auf der Lichtquelle befindet.

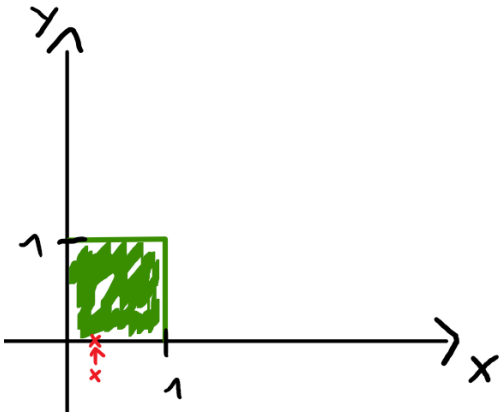



**Abbildung 16: (3) Ermitteln des nächsten Punktes innerhalb der Lichtquelle**

4. Rückübertragung des gefundenen Punktes in den dreidimensionalen Raum

Der so ermittelte Punkt wird dann als Position der eindimensionalen Lichtquelle angenommen.

# 3 Implementierung

## 3.1 Implementierung des 3-dimensionalen Raums

Um einen 3-dimensionalen Raum darzustellen bedarf es grundlegend einer Strategie, um Flächen darstellen zu können. Der Raum selbst, als auch die Objekte, die sich darin befinden, bestehen alle aus rechteckigen Flächen. Es bot sich daher an für eine Fläche einen Aufbau mit einem Stützvektor und zwei Richtungsvektoren zu wählen. Dabei wird mit dem Stützvektor der Punkt im Raum definiert. Die beiden Richtungsvektoren spannen von diesem Punkt aus eine Fläche auf (näheres dazu in Abschnitt 2.1). Damit kann zum Beispiel ein Quader jeglicher Größe mit 6 Flächen, welche jeweils mit einen Stütz- und 2 Richtungsvektoren beschrieben werden, dargestellt werden. Für die Positionierung des Raumes muss die Position der Kamera und die Positionierung der Bildebene in Betracht gezogen werden. Die Kamera wird auf die Position **(0 0 1)** gelegt. Die Bildebene wird in den Ursprung gelegt und erstreckt sich entlang der X- und Y-Achse (siehe Abbildung 17, Abbildung 18). Das Sichtfeld wird durch den Abstand zwischen Kamera und Bildebene bestimmt. Um sicher zu stellen, dass auf dem zu rendernden Bild nur das Raum Innere abgebildet wird, werden die Flächen die den Raum darstellen, sowohl um die Kamera als auch um die Bildebene gelegt.

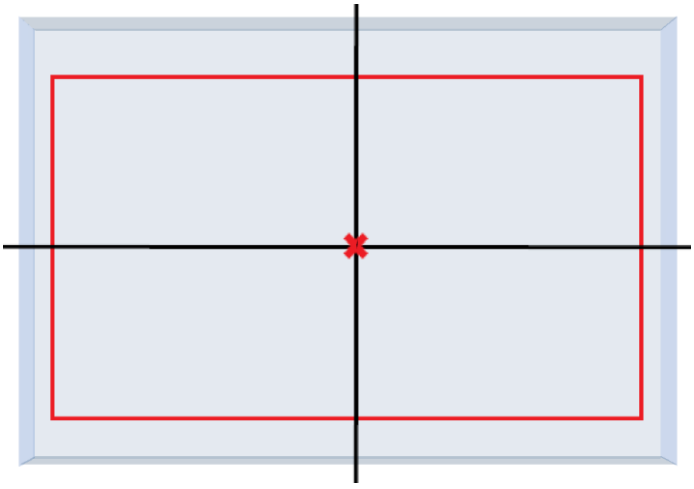


**Abbildung 17: Skizze der Frontansicht entlang der Z-Achse**

In Abbildung 17 ist die X- und die Y-Achse, sowie der Raum (mit hellblauen Flächen) skizziert. Das rote Kreuz stellt die Kamera und das rote Rechteck stellt die Bildebene dar. Die Bildebene hat auf der X-Achse zwei fixe Punkte an den Stellen 1 und -1. Die

zwei Punkte für die Y-Achse werden dynamisch zur Laufzeit des Programmes berechnet (Klasse „Camera“). Dafür wird zunächst das Verhältnis (ratio) von einer im Programm festgelegten Auflösung (zum Beispiel 1280x720 Pixel) berechnet:

$$ratio = \frac{width_{px}}{height_{px}}$$

Für die Berechnung der oberen und unteren Grenze der Bildebene gilt:

$$screen_{top} = \frac{1}{ratio}$$

$$screen_{bottom} = -\frac{1}{ratio}$$

Die hintere und vordere Wand (Betrachtung entlang der Z-Achse) werden von ihrer Breite mit 2.5 LE gewählt, also etwas größer als die Breite der Bildebene (2 LE). Für die Höhe wird eine Länge von 1.5 LE gewählt. Für die Tiefe des Raumes Beträgt die Länge vom Ursprung aus in Z-Richtung ebenfalls 2.5 LE. Die zwei Köper werden circa in der Mitte des Raumes platziert. Dafür wird angenommen das die Seitenlängen des Würfels jeweils 0.5 LE betragen. Bei dem Quader werden die Seitenlängen der Bodenfläche mit einer Länge von 0.5 LE und einer Höhe von 1 LE gewählt.

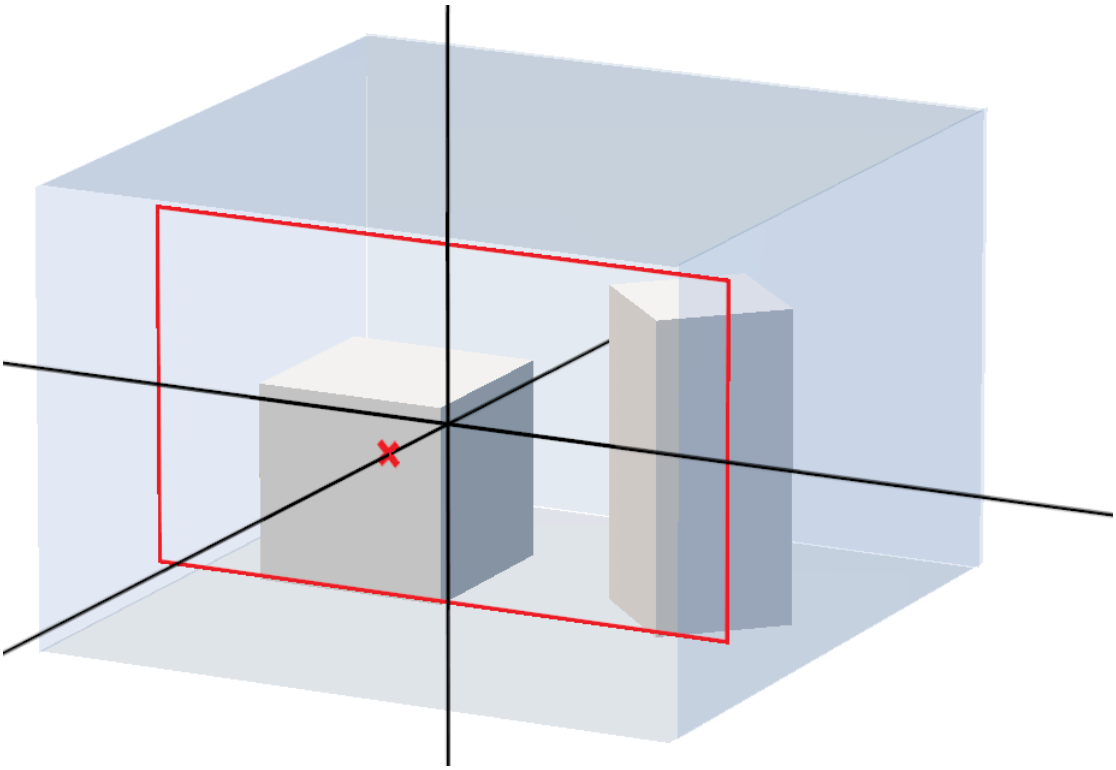


**Abbildung 18: Skizze mit den platzierten Körpern**

Wie in Abbildung 18 zu sehen ist der Quader um 45° gedreht. Der Würfel hingegen hat die identische Ausrichtung wie der Raum, von dem dieser umschlossen wird. Um nun noch die Kamera mitsamt der Bildebene im Raum mit einzuschließen, wird vom Ursprung aus die Tiefe des Raumes um 1.5 LE entlang der Z-Achse in den positiven Bereich gestreckt. Für die Seitenwände ergibt sich somit jeweils eine Länge von insgesamt 3.5 LE und eine Höhe von 1.5 LE. Bei dem Festlegen der Werte für die Stütz- und Richtungsvektoren stellte sich beim Testen des Programmes heraus das die Y-Achse im Raum invertiert ist. Es mussten also alle Y-Werte des Raumes und der Körper invertiert werden um ein Bild zu erhalten, welches nicht auf dem Kopf steht.

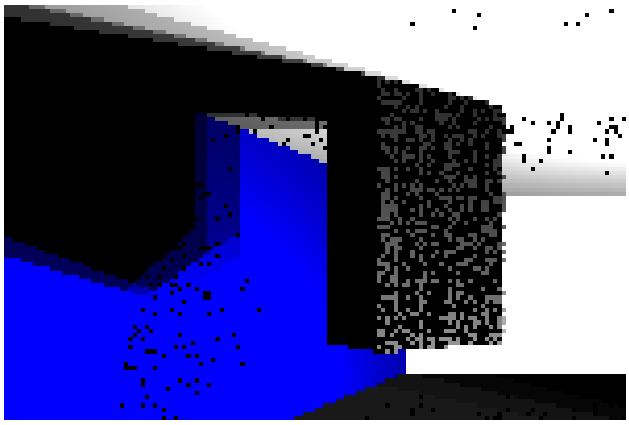


**Abbildung 19: Testbild mit Quader mit noch falschen Y-Werten**

Was in den beiden Skizzen nicht ersichtlich ist, sind die Farben. Diese können jedoch leicht implementiert werden indem pro Fläche ein Array mit 3 Elementen (für die Farbwerte rot, grün und blau) definiert wird. Die einzelnen Flächen werden durch die Klasse „Surface“ verkörpert, sprich jede definierte Fläche ist vom Typ „Surface“.

## 3.2 Funktionen

An dieser Stelle sollen interessante Funktionen der einzelnen Klassen beleuchtet werden. Ein Überblick über alle Klassen und deren Funktionen wird dabei in Abschnitt 3.3 gegeben.

### 3.2.1 Funktion zur Schnittpunktberechnung

Wie schon in Abschnitt 2.1 beschrieben ist für die Implementierung des Raytracers eine Kollisionserkennung notwendig. Im Programm existiert eine Klasse „surfaces“.

Diese wird genutzt um alle Oberflächen, die im Modell existieren, unter einem Objekt zusammenzufassen. Auf diese Oberflächen kann dann eine Methode der Klasse „surfaces“ mit dem Namen „getCollisionObject()“ angewandt werden. Die Funktion erwartet einen „ray“, also einen Strahl. Dieser verfügt auch in der implementierten Klasse nur über die Eigenschaften „origin“, also Ursprung und die Eigenschaft „normDirection“, also den normalisierten Richtungsvektor des Strahls.
Die Funktion aus Abschnitt 2.1 lässt sich umstellen, sodass eine Seite abhängig ist von den Koeffizienten X/Y/Z und eine Seite einen Konstanten Wert hat.

$$xD_1 + yE_1 - zN_1 = O_1 - S_1$$
$$xD_2 + yE_2 - zN_2 = O_2 - S_2$$
$$xD_3 + yE_3 - zN_3 = O_3 - zN_3$$

So lassen sich im Programm die folgenden Matrizen bilden:

$$(x \quad y \quad z) * \begin{pmatrix} D_1 & E_1 & -N_1 \\ D_2 & E_2 & -N_2 \\ D_3 & E_3 & -N_3 \end{pmatrix} = \begin{pmatrix} O_1 - S_1 \\ O_2 - S_2 \\ O_3 - S_3 \end{pmatrix}$$

Die Werte werden aus dem „ray“-Objekt und einer einzelnen Fläche, die Teile einer Liste, eines „surfaces“-Objekts ist, herausgelesen. Angenommen man überprüft nur eine Fläche auf einen Schnittpunkt so wird versucht das Gleichungssystem zu lösen. Dazu wird die linalg.solve()-Funktion von NumPy genutzt. Diese empfängt als Parameter die 3x3 Koeffizientenmatrix und die 3x1 Ergebnismatrix. Bei Nicht-Lösbarkeit wird eine Exception geworfen, die abgefangen werden kann. Bei Lösbarkeit wird als Rückgabewert die Variablenmatrix mit X/Y/Z zurückgegeben.
Ist das lineare Gleichungssystem lösbar, so werden noch die Werte X und Y aus der zurückgegebenen Variablenmatrix überprüft. Wenn X und Y bestimmte Werte eingenommen haben existiert kein Schnittpunkt. Orientiert man sich an unserem vorgestellten Modell der Flächen aus Abschnitt 2.1, dürfen die errechneten Werte von X und Y aus der Variablenmatrix nur zwischen 0 und 1 liegen(Zahl jeweils miteingeschlossen). Ist z.B. der Wert von X größer als 1, so wird zu lange entlang des ersten direktionalen Vektors gewandert und der Schnittpunkt ist nicht mehr Teil der Fläche. Ist X kleiner als null passiert etwas ähnliches, man verlässt den Bereich der Fläche im negativen Teil des direktionalen Vektors. In beiden Fällen befindet sich der errechnete Punkt nicht mehr auf der gewollten Fläche. Auch wird überprüft ob Z positiv ist, sodass der Strahl nicht in die falsche Richtung geworfen wird.

Da wir nun aber mehrere Flächen haben und es möglich ist, dass ein Strahl Schnittpunkte mit mehreren dieser Flächen hat, muss auch noch überprüft werden, welche Fläche als erstes getroffen wird. Dazu wird die erste Fläche, die einen Schnittpunkt hat, zusammen mit der Variablenmatrix zwischengespeichert und später der Z-Wert der Variablenmatrix mit dem Z-Wert des neuen Schnittpunktes mit einer anderen Fläche verglichen. Ist das neue Z kleiner so wird die neue Fläche zusammen mit der neuen Variablenmatrix zwischengespeichert. Die Fläche, die am Ende übrigbleibt, wird zusammen mit der Strecke, die der Strahl zurücklegt, also Z, zurückgegeben.

### 3.2.2 Probleme mit Schnittpunktberechnung

Bei der Berechnung der Schnittpunkte, wie in 3.2.1 treten einige Fehler bzw. ein zentraler Fehler auf. Dabei handelt es sich immer um Grenzwert-Fälle und Floats. Konkret geht es darum, wenn ein Strahl auf eine Kante auftrifft. Dabei sind die Fehler meist abhängig von der Auflösung. Ein Fehler, der bei einer 400x300 Auflösung auftritt, muss nicht bei einer 300x200 Auflösung auftreten. Auch zeigen sich an bestimmten Kanten einige Fehler nur oder besonders intensiv bei niedrigen Auflösungen. Je nach Implementierung der Flächen und der Funktion wird so z.B. für die Kante zwischen der oberen und hinteren Seite des linken Würfels als Kante des hinteren Würfels erkannt, obwohl sie aus Sicht des Betrachters eigentlich zur Oberseite gehören sollte. Durch verschiedene Winkel zum Licht sollte die obere Seite eigentlich viel heller als die hintere Seite des Würfels sein. Durch den Fehler erscheint allerdings eine dunklere Linie am hinteren Ende der Oberseite, wie in Abbildung 20 zu sehen.

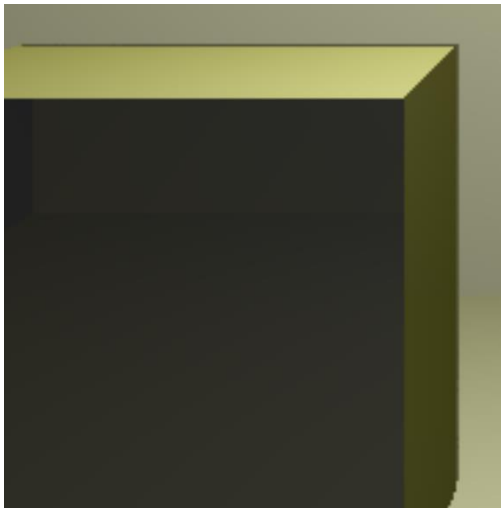


**Abbildung 20: Würfel, Kantenfehler, gesamte Kante**

Des Weiteren können z.B. durch falsche Berechnung von floats auch einzelne Pixelfehler entstehen, die ihren Ursprung darin haben, dass ein Float der eigentlich 0 sein sollte, als z.B. – 7.01351..e-16 berechnet wird, sodass eigentlich ein Schnittpunkt mit der Fläche entsteht, er aber nicht registriert wird, weil der Wert ja marginal kleiner ist als 0. Werte minimal kleiner als null lassen sich allerdings abfangen und verbessern, bevor sie auf die Bedingung geprüft werden.
Durch falsche Berechnungen an Grenzwerten können aber auch zu einzelnen Pixelfehlern führen, sodass nicht gleich die gesamte Kante die falsche Farbe annimmt.
Beide Arten von einzelnen Pixelfehlern stellen sich ähnlich dar, wie in Abbildung 21 zu sehen.

**Abbildung 21: Würfel, einzelne Pixelfehler an Kante**

Generell ist das Lösen dieser Grenzwertprobleme sehr mühselig, da es sich als eine große Herausforderung darstellt, für jede mögliche Kante, die dieses Problem betrifft, einen funktionalen Ansatz zu bieten, der nicht an anderer Stelle wieder dazu führt, dass Pixelfehler auftreten. So ist z.B. auf Verhältnisse und Abstände zum Ursprung des Strahls kein Verlass, da eine Anwendung von Operationen an der einen Stelle dazu führen könnten, dass an anderer Stelle wiederrum die falsche der beiden Flächen erkannt wird.
So wäre es z.B. möglich eine einzelne Lösung für jede einzelne Problemkante zu schreiben. Also eine Funktion, die mit dem Kantenvergleich zwischen oberer und hinterer Würfelfläche umgeht, oder zwischen oberer und vorderer Würfelfläche. So könnte

man dann über fünf Funktionen schreiben, um das Problem zu lösen. Allerdings handelt an sich dabei um einen relativ stupiden Ansatz.
Ein ähnlich stupider Ansatz ist das Verändern der direktionalen Vektoren. Verkleinert man z.B. den direktionalen Vektor, der die vordere Würfelfläche nach oben hin ausdehnt auf dezimaler Ebene minimal, sodass es keinen Schnittpunkt mehr zwischen der vorderen und oberen Fläche mehr gibt, so verschwinden auch alle Pixelfehler. Dabei handelt es sich mathematisch aber nicht um die sauberste Lösung.
Nach langem herumprobieren und nachdenken ist es leider immer noch nicht möglich eine Lösung zu finden, die simpel, z.B. durch bestimmte Operationen der linearen Algebra eine allgemeine Lösung für alle Kantenprobleme liefert. So wird auf die Lösung durch Veränderung der direktionalen Vektoren zurückgegriffen. Ob dabei andere Fehler entstehen, wie z.B., dass ein Strahl in den Würfel hineinfällt musste auch ausprobiert werden.
Nach Überprüfung des Bilds in 4K lässt sich feststellen, dass sämtliche Pixelfehler behoben sind und auf der anderen Seite keine weiteren Fehler mehr auftreten. So wurde sich dazu entschieden die Verkleinerung der direktionalen Vektoren beizubehalten, da sie als einzige Lösung funktioniert und dabei auch noch so einfach ist.

### 3.2.3 Funktion zur Berechnung der Farbwerte von Pixeln

Die Funktion „traceRays“ berechnet die Farbwerte für jedes einzelne Pixel. Dafür kommen die Funktionen, welche in Abschnitt 3.2.1 und in Abschnitt 2.2 erläutert wurden zum Einsatz. Aufgerufen wird die Klasse in der main()-Funktion. Diese iteriert dabei über alle Pixel. Die Anzahl der Pixel ist durch die vorab definierte Auflösung festgelegt. Für jedes Pixel wird die Funktion „traceRays()“ aufgerufen. Die Funktion ist in der Klasse „RayTracer“ definiert und erwartet als Parameter einen Pixel, welcher mit seiner Position in Höhe und Breite beschrieben wird. In der Funktion selbst wird ein Ray erzeugt und der Funktion „getCollisionObject()“ (Abschnitt 3.2.1) mitgegeben. Diese liefert ein Tupel mit der Fläche, die bei diesem Pixel am nächstem ist, sowie dem Distanzwert zu dieser Fläche in Z-Richtung zurück (minDistance). Daraufhin wird die Position auf der Fläche berechnet:

$$pos_{Surface} = ori\ gin + minDistance * normDirection$$

Hier entspricht „origin“ der Kamera-Position und „normDirection“ ist der normalisierte Vektor, der die Richtung des Strahls angibt. Anschließend wird von diesem Punkt aus ein kleiner Offset von 1e-5 in Richtung des Normalenvektors der Oberfläche addiert. Dies ist ein kleiner Hack um bei späteren Berechnungen eine Kollision mit der „Ursprungs-Surface“ in jedem Fall zu verhindern. Nachdem die Berechnung der Beleuchtung für diesen Punkt abgeschlossen ist, wird ein reflektierter Ray berechnet und der Einfluss dieses reflektierten Rays auf den Farbwert durch einen von der Oberfläche abhängigen Multiplikator („reflection“) angepasst. Ist „reflection“ 0 oder wurde die maximale Tiefe („RayTracer._max_depth“) erreicht, werden die Farbwerte auf einen Bereich zwischen 0 und 1 eingeschränkt (mit „numpy.clip()“) und auf die Bildfläche geschrieben.

## 3.3 Gerendertes Bild in 4k

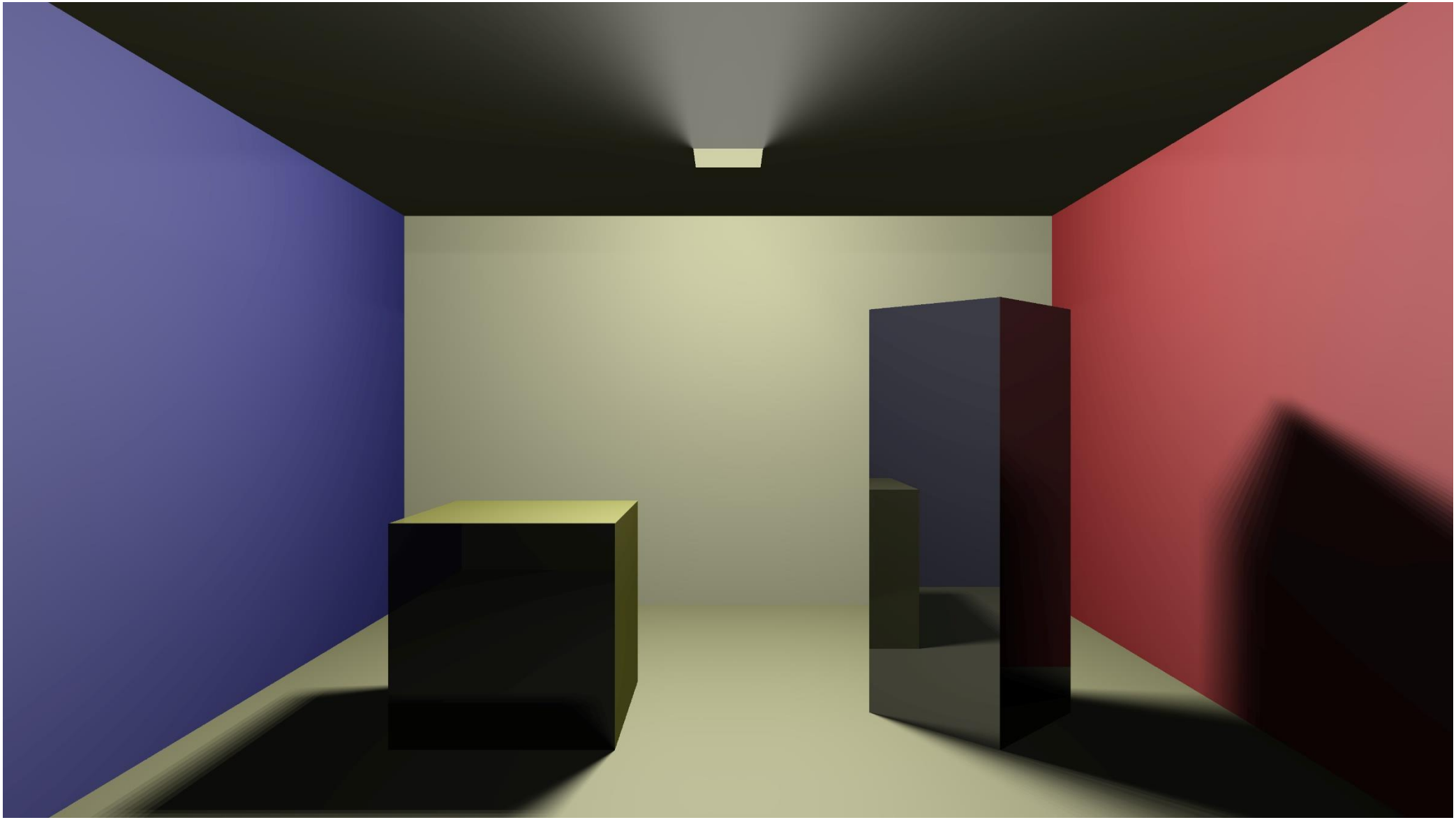
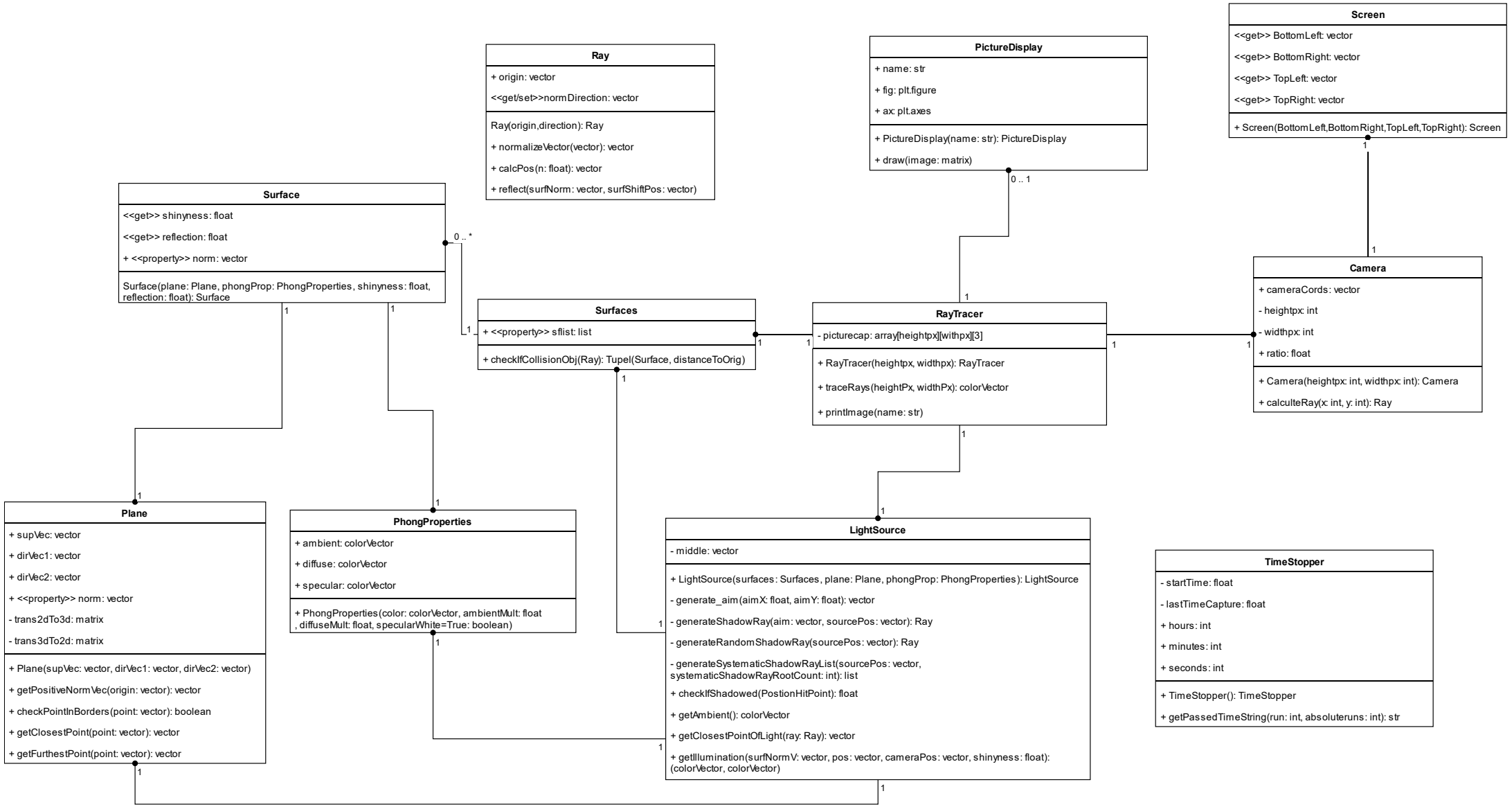


**Abbildung 22: Ergebnis der Implementierung, Auflösung 4k, Schatten etc. hoch**

## 3.4 Klassenmodell



## 3.5 Performance-Optimierungen

Neben der leichten Erweiterbarkeit des Programms durch seinen objektorientierten Aufbau, kann man das Programm auch noch abwandeln, um eine schnellere Berechnung zu erreichen. Zusätzlich kann man das bilderzeugende Verfahren anpassen, um realistischere oder interessantere Bilder zu bekommen.

Das implementierte Programm ist denkbar ineffizient. So braucht die Berechnung eines Bildes in Full-HD (1080*1920px) mit 64 Shadow Rays und einer maximalen Raytracing-Tiefe von 4 (also maximal 256 Shadow Rays pro Pixel) etwa 15 Stunden auf einer modernen CPU. Dies liegt daran, dass Python als interpretierte Sprache langsamer ist als eine Sprache, die als Maschinencode ausgeführt wird, wie etwa C.

Trotzdem kann man durch Nutzen von in C implementieren Modulen von Python, wie etwa Numpy, und das Verwenden von in C umgesetzten builtins von Python, wie der map() Befehl, die Ausführungsgeschwindigkeit beschleunigen. Durch die Ersetzung von einigen Schleifen durch map() konnte die Laufzeit um fast ein Zehntel reduziert werden.

Neben der effizienteren Ausnutzung der vorhandenen Ressourcen kann man die vorhandenen Ressourcen erhöhen, indem man mehr Prozessorkerne zur Berechnung einsetzt. Durch den Einsatz des Python Moduls „multiprocessing“ (Standard Python Modul) konnte die Berechnungszeit auf einem System mit vier Kernen ca. auf ein Drittel reduziert werden.

| Berechnungsverfahren | Zeit |
|---|---|
| Single Process | ~15:00 |
| Multi Process | ~6:20 |

**Tabelle 1: Vergleich der Rechenzeiten eines 480*853px Bildes mit 16 Shadow Rays**

Da multiprocessing kein erlaubtes Modul ist, gibt es im Github einen extra Branch der dieses Modul verwendet.

Auch wäre es möglich den ganzen Python-Code z.B. mit Cython in C-Code umzuwandeln. Dadurch könnte sich das Programm ähnlich schnell verhalten, wie eine Raytracing Implementierung in C. Allerdings müssten hierzu statische Typdeklarationen für Variablen im Code angewandt werden und eine weitere Einarbeitung in Cython müsste

stattfinden. Der Arbeitsaufwand hierbei würde den Rahmen des Projektes sprengen, sodass es bei einer Bemerkung an dieser Stelle bleibt.

## 3.6 Animation

Durch Erzeugen von mehreren Bildern mit leichten Veränderungen kann eine Animation erzeugt werden. In dem Branch „animation“ wurde dafür eine Lösung implementiert. Diese Lösung verwendet neben den erlaubten Modulen das Modul „multiprocessing“.
Die erzeugten Bilder wurden mit einer Videobearbeitungssoftware zusammengeschnitten und sind auf YouTube einsehbar[5].

[5] https://youtu.be/bupGiTj3trM

# Quellenverzeichnis

Aflak, O. (26. Juli 2020). *Ray Tracing From Scratch in Python: TheStartup-Website*. Abgerufen am 28. Dezember 2020 von TheStartup-Website: https://medium.com/swlh/ray-tracing-from-scratch-in-python-41670e6a96f9

Phong, B. T. (1975). *Illumination for Computer Generated Pictures.*

Wegscheider, A. (2009/2010). *Facharbeiten: Technische-Universität-München-Website.* Abgerufen am 28. Dezember 2020 von Technische-Universität-München-Website: https://www.edu.tum.de/fileadmin/tuedz01/www/Sch%C3%BClerkonferenz/Facharbeiten_2010/andreas_wegscheider_2010.pdf