Fachbereich Mathematik / Informatik
Studiengang Informatik

Rendering von Lichtbrechung in Echtzeit

Felix Unger

Erstprüfer: Prof. Dr. Gabriel Zachmann

Zweitprüfer: Prof. Dr. Frieder Nake

Abgabedatum: 01.12.2015

Kurzfassung

Mein Ziel bei dieser Bachelorarbeit war es, ein interaktives Programm zu schreiben, das in Echtzeit zweiseitige Lichtbrechung bei transparenten Objekten berechnen und rendern kann. In dieser Arbeit stelle ich dieses Programm, den benutzten Algorithmus und verschiedene alternative Methoden für das Rendering von Lichtbrechung vor. Außerdem gehe ich auf die physikalischen Gesetze in diesem Kontext ein.

Das Programm kann zweiseitige Lichtbrechung in interaktiver Geschwindigkeit für diverse Objekte realistisch rendern und auf Wunsch die gebrochenen Lichtstrahlen visualisieren. Sehr konkave Objekte oder Objekte mit hoher Depth Complexity lassen sich mit dieser Methode allerdings nicht zufriedenstellend darstellen.

Abstract

My goal for this bachelor thesis was to create an interactive application that calculates and renders two sided refraction for transparent objects in real time. In this thesis I present this application, the used algorithm and various alternative methods for rendering refraction. Additionaly I go into the physical laws in this context.

The application can realistically render two sided refraction in interactive speeds for various objects and visualize the refracted light rays on demand. Very concave objects or objects with a high depth complexity can't be rendered satisfyingly with this method.

I Inhaltsverzeichnis

II Abbildungsverzeichnis

III Tabellenverzeichnis

IV Listing-Verzeichnis

1 Einleitung

In vielen 3D-Applikationen, wie z.B. Simulationen, Modellierungsprogrammen oder Videospielen, ist Interaktivität sehr wichtig. Um diese zu erreichen, braucht man Software, die in Echtzeit 3D-Szenen darstellen (*rendern*) kann, die vom Nutzer oder dem Programm manipuliert werden. In dieser Arbeit geht es um die Darstellung von transparenten Objekten, beispielsweise Objekten aus Glas, in interaktiver Geschwindigkeit.

Es gibt bereits Techniken, die Lichtbrechung korrekt darstellen, die aber meistens nicht in Echtzeit umsetzbar sind, wie z.B. *Raytracing* (siehe Kapitel 3.1). Deshalb muss man für Echtzeitanwendungen auf die Technik des *Scanline-Renderings* zurückgreifen. Dieses ist wesentlich schneller als Raytracing, aber es ist damit nicht ohne Weiteres möglich, physikalisch korrekte Lichtbrechung darzustellen. Man kann nur eine Annäherung an korrekte physikalische Effekte erreichen. Ich habe ein Programm geschrieben mit dem ich versucht habe, diese Effekte so realistisch wie möglich zu halten und dabei trotzdem eine hohe Framerate zu behalten. Das ist mir auch relativ gut gelungen. Die Ergebnisse reichen bei einer Auflösung von 1400 x 583 Pixeln und mit einer dynamischen Environment Map von 160 FPS bei 540.000 Polygonen bis zu 300 FPS bei 65.000 Polygonen oder mehr bei niedrigeren Polygonzahlen.

Zuerst gehe ich in Kapitel 2 auf die physikalischen und technischen Grundlagen ein. In Kapitel 3 stelle ich diverse alternative Methoden vor. Dann folgt in Kapitel 4 eine ausführliche Beschreibung der von mir genutzten Methode und die Ergebnisse davon. Anschließend gebe ich in Kapitel 5 eine Zusammenfassung und einen Ausblick.

2 Grundlagen

2.1 Physikalischer Hintergrund

Hier gehe ich auf die physikalischen Aspekte von Lichtbrechung ein.

2.1.1 Brechungsgesetz

Lichtbrechung ist die Richtungsänderung eines Lichtstrahls beim Wechsel von einem in ein anderes Medium. Das lässt sich zum Beispiel sehr gut erkennen, wenn man einen Stift in ein Glas mit Wasser hält. An der Oberfläche scheint der Stift einen Knick zu haben. Das liegt daran, dass die Lichtstrahlen, die aus dem Wasser kommen und auf das Auge fallen, vom Medium Wasser zum Medium Luft wechseln und so gebrochen werden. Diese Lichtbrechung wird vom *Snelliusschen Brechungsgesetz*[BBE+03] beschrieben. Dieses lautet:

$$\frac{sin(\alpha)}{sin(\beta)} = \frac{n_2}{n_1} \tag{1}$$

Dabei ist α der Eingangswinkel, β der Brechungswinkel und n_1 und $\vec{n_2}$ sind die *Brechungsindizes* der beiden Materialien. Der *Brechungsindex* ist eine physikalische Konstante, die vom Material abhängt. Der Brechungsindex von Luft ist ungefähr 1, der von Glas ungefähr 1.5. In Abbildung 1 ist der Sachverhalt noch einmal dargestellt.

Für unsere Zwecke stellen wir die Formel folgendermaßen um:

$$asin\left(\frac{sin(\alpha) \cdot n_1}{n_2}\right) = \beta \tag{2}$$

Wir brauchen also den Eingangswinkel und die beiden Brechungsindizes um den Ausgangswinkel zu bestimmen.

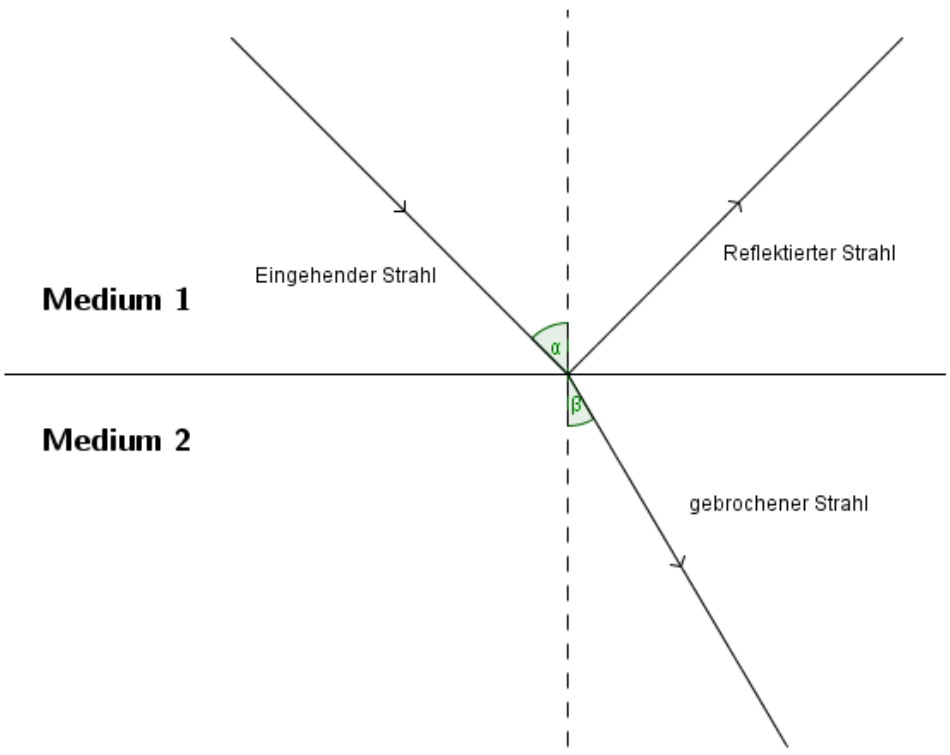

Abbildung 1: Lichtbrechung zwischen zwei Medien

Viele Menschen denken bei dem Wort „Lichtbrechung“ an ein Prisma in dem weißes Licht in die verschiedenen Farben aufgespalten wird. Dieser Effekt wird „Dispersion“[1] genannt. Dispersion ist in der Optik die Abhängigkeit des Brechungsindex von der Wellenlänge

des Lichts. D.h. Licht wird unterschiedlich stark gebrochen abhängig von der Farbe des Lichts. Bei einem Prisma wird das besonders deutlich, da dies geometrisch genau darauf ausgelegt ist, das Licht aufzuspalten. Bei den meisten anderen Objekten aus Glas ist der Effekt aber vernachlässigbar, weshalb ich hier nicht weiter darauf eingehe.

2.1.2 Reflexion

Alle Materialien, die transparent sind, reflektieren auch zu einem gewissen Grad. Für eine möglichst realistische Darstellung muss also auch die Reflexion bedacht werden. Die Gleichung zum Ausrechnen des Winkels des reflektierten Vektors ist sehr einfach. Der Ausgangswinkel ist gleich dem Eingangswinkel. Es gibt aber noch eine andere Formel, die in diesem Zusammenhang wichtig ist. Und das ist *Schlicks Approximation* [Sch94]. Diese Formel bestimmt, wie viel des eingehenden Lichtes bei Auftritt auf eine transparente Oberfläche reflektiert und wie viel gebrochen wird.

$$R(\theta) = R_0 + (1 - R_0)(1 - cos(\theta))^5 \tag{3}$$

$$R_0 = \left(\frac{n_1 - n_2}{n_1 + n_2}\right)^2 \tag{4}$$

$R(\theta)$ ist der *Reflexionskoeffizient*, θ ist der Eingangswinkel und n_1 und $\vec{n_2}$ sind die Brechungsindizes der beiden Materialien.

Der Reflexionskoeffizient ist eine Zahl zwischen 0 und 1, die angibt, wie hoch der Anteil des reflektierten Lichts ist im Vergleich zum gebrochenen Licht. Die sichtbare Farbe an der Oberfläche eines transparenten Objektes errechnet sich also folgendermaßen:

$$C = R(\theta) \cdot C_{reflekt} + (1 - R(\theta)) \cdot C_{gebrochen} \tag{5}$$

2.1.3 Totale interne Reflexion

Totale interne Reflexion ist ein Phänomen, das auftreten kann, wenn ein Lichtstrahl von einem dichten auf ein weniger dichtes Medium trifft. Dabei wird das gesamte Licht an der Grenzfläche reflektiert und erreicht somit nicht das andere Medium. Dies tritt auf, wenn der Eintrittswinkel des Lichtes größer ist als ein bestimmter Winkel θ_c, der *kritischer Winkel* genannt wird. Das ist der Winkel, in dem der Strahl parallel zur Grenzfläche gebrochen würde. Der kritische Winkel, lässt sich folgendermaßen berechnen:

[1] Quelle: `http://www.mikomma.de/optik/disp/dispaket.htm`

$$\theta_c = arcsin\left(\frac{n_2}{n_1}\right) \tag{6}$$

Dabei sind n_1 und $\vec{n_2}$ die Brechungsindizes des ersten bzw. zweiten Mediums.

Beweis:

Wenn ein Strahl im kritischen Winkel auf eine Oberfläche trifft, muss der gebrochene Strahl parallel zur Grenzfläche verlaufen, muss also senkrecht zur Normalen stehen. Das bedeutet, dass diese Gleichung gelten muss:

$$asin\left(\frac{sin(\theta_c)\cdot n_1}{n_2}\right) = \frac{\pi}{2} \tag{7}$$

Jetzt setze ich für θ_c den Term für den kritischen Winkel ein:

$$\begin{aligned} & asin\left(\frac{sin\left(arcsin\left(\frac{n_2}{n_1}\right)\right)\cdot n_1}{n_2}\right) \\ = \quad & asin\left(\frac{\frac{n_2}{n_1}\cdot n_1}{n_2}\right) \\ = \quad & asin(1) \\ = \quad & \tfrac{\pi}{2} \end{aligned}$$

□

2.2 Computergrafik Basis

Hier erkläre ich einige Basiskonzepte der Computergrafik(CG), die für das Verständnis der später vorgestellten Algorithmen sehr wichtig sind.

2.2.1 Shader

Shader sind essentiell für die Echtzeitberechnung von Lichtbrechung. Shader sind Programme, die an bestimmten Stellen in der Grafik-Pipeline „eingeschoben“ werden (Abb. 2). Es folgen Beschreibungen der drei wichtigsten Shader-Arten, die ich auch in meinem Programm verwendet habe.

Der *Vertex Shader* bekommt als Eingabe die Vertices und Normalen eines Objektes und gibt genau so viele Vertices und Normalen wieder aus. Dabei kann er beliebige Transformationen ausführen. Im Vertex Shader werden normalerweise die Objektkoordinaten in Welt- und Kamerakoordinaten verwandelt.

Nach dem sogenannten „primitive assembly“, also dem Zusammensetzen von Vertices zu Primitiven, werden diese zum *Geometry Shader* gegeben. Dabei muss man vorher festlegen, welche Art von Primitiven dieser erhält. Also zum Beispiel Dreiecke oder Vierecke. Der Geometry Shader kann beliebige Transformationen ausführen und sogar neue Vertices erzeugen, die er dann in der Pipeline weitergibt.

Nach dem Rasterisieren der Primitive, werden die „Fragmente“, also die Informationen zum Füllen eines Pixels, zum *Fragment Shader* gegeben. Der Fragment Shader kann auf beliebige Art die Farbe des Pixels berechnen und diese weitergeben an den Framebuffer. Hier wird normalerweise die Beleuchtung berechnet.

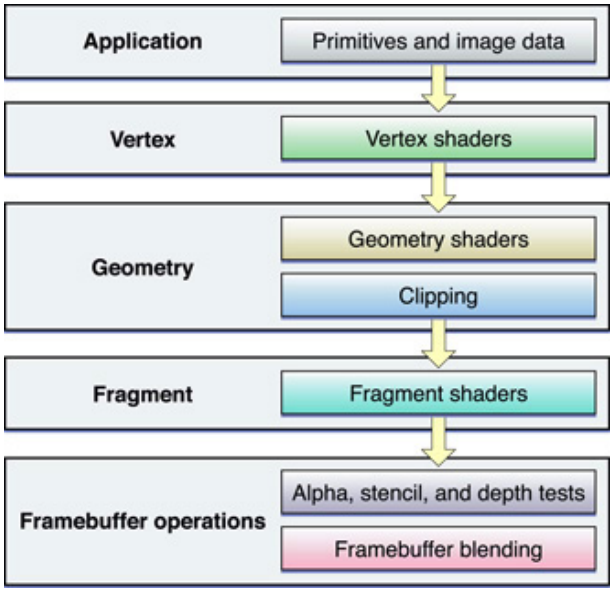

Abbildung 2: Vereinfachte Pipeline des Scanline Renderings[2]

2.2.2 Cubemap-Texturen

Eine Cubemap-Textur ist eine Textur, die aus sechs quadratischen 2D-Texturen besteht, die wie ein Würfel angeordnet sind. Indexiert wird sie mithilfe eines 3D-Vektors, dessen Ursprung in der Mitte des Würfels liegt. Dieser Vektor „zeigt“ auf ein bestimmtes Texel in der Cubemap (Abb. 3).

Es muss also der 3D-Vektor (r_x, r_y, r_z) in Texturkoordinaten s, t umgewandelt werden. Dafür werden diese Formeln benutzt [Zim99]:

[2]Quelle: `https://developer.apple.com/library/mac/documentation/GraphicsImaging/Conceptual/OpenGL-MacProgGuide/art/driver_graphics_pipeline.jpg`

$$s = \frac{\frac{sc}{|ma|} + 1}{2} \tag{8}$$

$$t = \frac{\frac{tc}{|ma|} + 1}{2} \tag{9}$$

Die Werte sc, tc und ma werden aus der Tabelle 1 entnommen und hängen von der „major axis direction“ ab. Das ist die Komponente des Vektors r, die am größten ist. Damit wird angegeben, welche Seite des Würfels von dem Vektor geschnitten wird, also von welcher Seite der Cubemap die Texturkoordinaten berechnet werden müssen.

major axis direction	**sc**	**tc**	**ma**
$+r_x$	$-r_z$	$-r_y$	r_x
$-r_x$	$+r_z$	$-r_y$	r_x
$+r_y$	$+r_x$	$+r_z$	r_y
$-r_y$	$+r_x$	$-r_z$	r_y
$+r_z$	$+r_x$	$-r_y$	r_z
$-r_z$	$-r_x$	$-r_y$	r_z

Tabelle 1: Die Variablenzuweisung für die Berechnung der Texturkoordinaten bei Cubemaps [Zim99]

Mit den berechneten Texturkoordinaten wird dann die entsprechende Seite der Cubemap indexiert und die Farbe wird zurückgegeben. Um selbst eine Cubemap zu erstellen, indem z.B. die Umgebung gerendert wird, muss jeweils ein Bild in positiver und negativer Richtung der drei Koordinatenachsen gerendert werden, bei einem Field of View von 90 Grad.

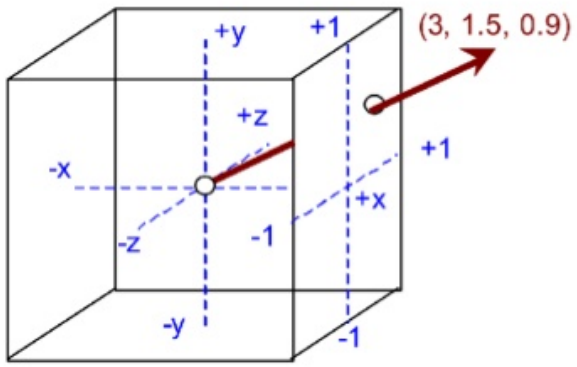

Abbildung 3: Skizze zum Indexieren einer Cubemap[3]

Eine Environment-Textur ist eine Cubemap-Textur, die die Umgebung in einer Szene darstellt und in jeder Richtung von der Kamera unendlich weit entfernt ist. Dies wird üblicherweise so realisiert, dass die Cubemap-Textur auf die Innenseite eines Würfels gemappt wird, in dem die Kamera liegt. Dieser Würfel macht die gleichen Translationen wie die Kamera, dreht sich aber nicht mit. Außerdem wird er immer als erstes gezeichnet und dabei der *Depth Test* ausgeschaltet, sodass der Würfel von der Kamera aus gesehen immer hinter allem anderen liegt. Eine Environment-Textur wird auch oft jeden Frame aktualisiert, um eine dynamische Umgebung darstellen zu können. Meistens sind das Bilder des Himmels und des Horizonts.

3 Alternative Methoden

In diesem Kapitel stelle ich verschiedene Methoden für Rendering von Lichtbrechung vor, die es bisher gibt, die ich aber nicht benutzt habe und ich erkläre, warum ich sie nicht benutzt habe.

3.1 Raytracing

Es ist heute bereits möglich, physikalisch nahezu exakte Darstellungen von transparenten Szenen oder Objekten zu erreichen mit dem sogenannten *Raytracing*. Dies ist eine Technik, bei der Strahlen von der Kamera aus auf die Szene „geschossen“ werden (Abb. 4). Dabei werden die Pixel in der gewünschten Auflösung auf die Bildebene projiziert und für jedes ein Strahl erzeugt, der an der Kamera startet und durch den Pixel verläuft. Wenn ein Strahl auf etwas in der Szene trifft, wird ein *Shadow Ray*, ein reflektierter, ein gebrochener und bei Bedarf weitere Strahlen generiert und rekursiv weiterverfolgt, solange bis ein bestimmtes Abbruchkriterium erreicht ist. Dann werden die Farben der von den Strahlen getroffenen Objekte als gewichtete Summe für das Pixel benutzt.

[3] Quelle: `http://de.slideshare.net/Mark_Kilgard/11texture`

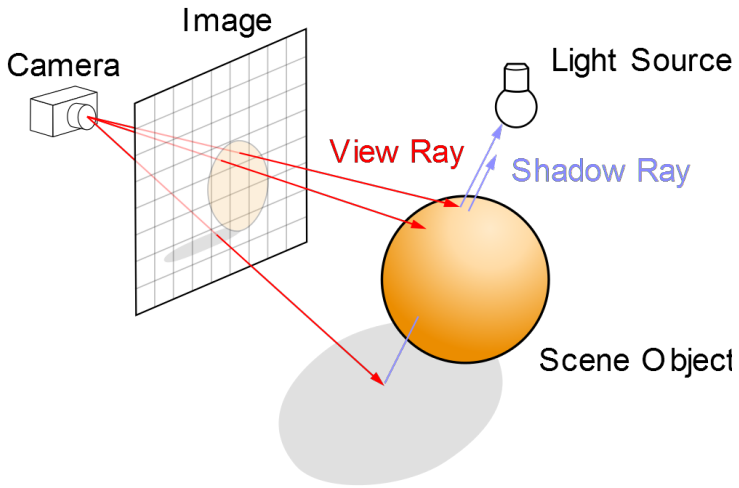

Abbildung 4: Beispielbild für die Generierung von Strahlen beim Raytracing [4]

Für diese Technik gibt es viele Erweiterungen wie Multisampling [AK] und auch Versuche des Raytracings in Echtzeit [ORM08], aber der große Nachteil dieser Technik bleibt trotzdem bestehen. Sie ist sehr rechenaufwändig und deshalb mit heutiger Hardware noch nicht praktikabel für den Einsatz in Echtzeitanwendungen.

3.2 Methode in Game Engines

In Computerspielen ist ein schneller Rendering-Loop besonders wichtig, um ein flüssiges Spielgeschehen zu gewährleisten. Um dort also transparente Objekte darzustellen, wird oft Lichtbrechung nur vorgetäuscht. Bei *Unity* zum Beispiel wird im ersten Schritt die Geometrie hinter dem transparenten Objekt in eine Textur gerendert und diese wird dann mithilfe einer Noise-Funktion oder auch den Normalen des Objektes verzerrt [Pra]. Das ist zwar keine korrekte Lichtbrechung, aber es erzeugt einen Eindruck von z.B. Glas und das reicht in diesem Kontext aus. Ich suche aber nach einer besseren Annäherung an reale Effekte.

3.3 Lichtbrechung an einer Oberfläche

Eine weitere einfache Lösung ist die Lichtbrechung an nur einer Oberfläche. Dabei wird das Licht nur an der Oberfläche gebrochen, die direkt von der Kamera aus sichtbar ist. Das heißt, es wird ein Strahl berechnet, der von der Kamera aus zur vordersten Oberfläche des transparenten Objektes geht. An diesem Punkt wird mithilfe der *Normalen* (also dem Vektor, der Senkrecht auf dem Polygon steht) und den Brechungsindizes der gebrochene Vektor berechnet (Glei. 2). In allen gängigen Shader-Sprachen gibt es bereits

[4] Quelle: `https://upload.wikimedia.org/wikipedia/commons/8/83/Ray_trace_diagram.svg`

eine Lichtbrechungsmethode. Mit dem gebrochenen Strahl indexiert man ein Pixel in einer Cubemap, die den Hintergrund darstellt, um die Farbe an diesem Punkt herauszufinden.

Diese Methode ist sehr einfach umzusetzen. Sie ist außerdem sehr ressourcenschonend, wenn man für die Berechnungen und die Texturen Shader verwendet. Sie produziert allerdings auch nicht sonderlich realistische Ergebnisse, da der Lichtstrahl nur an einer Oberfläche gebrochen wird, obwohl er in der Realität mindestens zweimal gebrochen werden müsste, wenn man durch ein Objekt hindurchschaut.

3.4 Lichtbrechung an zwei Oberflächen im Image-Space

Es gibt auch Ansätze, die Lichtbrechung an zwei Oberflächen nur mit Berechnungen innerhalb des Image-Space darstellen, also nur mit der Geometrie innerhalb des Viewing Frustum. Hier stelle ich zwei dieser Methoden vor.

3.4.1 Wyman

In dieser Methode von Wyman [Wym05] wird von der Kamera als Startpunkt der Lichtbrechung ausgegangen, die Lichtbrechung wird also praktisch „rückwärts“ berechnet. Das funktioniert aber genau so wie andersherum, wenn man die Brechungsindizes vertauscht.

Zuerst wird der erste gebrochene Lichtstrahl $\vec{T_1}$ berechnet (Abb. 5). Das wird mithilfe des Snelliusschen Brechungsgesetzes (Glei. 1) gemacht. Alles, was zur Berechnung des gebrochenen Strahls benötigt wird, ist der Kameravektor $\vec{V}$, der Schnittpunkt P_1 und die Normale des getroffenen Polygons $\vec{N_1}$. Der Fragment Shader kann also sehr einfach $\vec{T_1}$ berechnen.

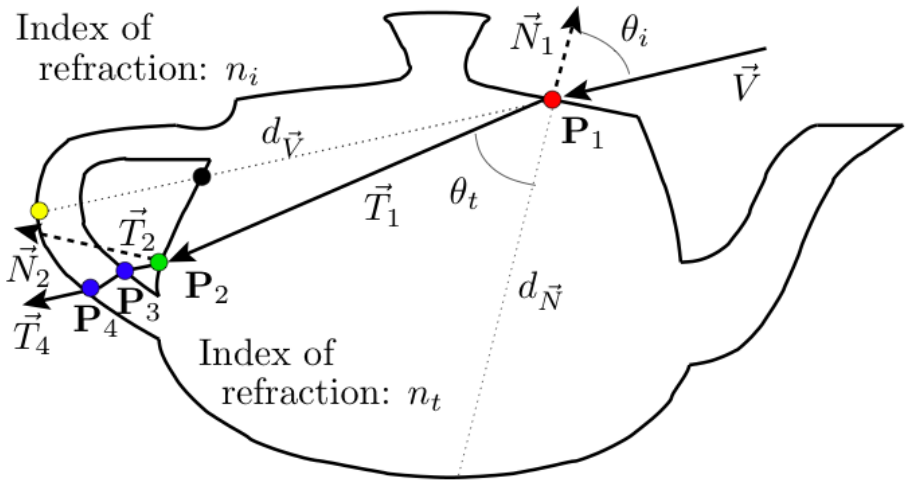

Abbildung 5: Beispiel zur Veranschaulichung von Wymans Algorithmus [Wym05]

Um den zweiten gebrochenen Vektor $\vec{T_2}$ zu berechnen, braucht man den zweiten Schnittpunkt P_2 und die Normale $\vec{N_2}$ an diesem Punkt. Die Geradengleichung des Vektors $\vec{T_1}$ sieht folgendermaßen aus: $P_1 + t \cdot \vec{T_1}$. Man kann allerdings nicht einfach wie beim Raytracing den Strahl $\vec{T_1}$ mit dem Objekt schneiden um den zweiten Schnittpunkt zu erhalten, da die Rechenzeit dazu viel zu hoch wäre und im Fragment Shader nicht auf andere Polygone der Szene zugegriffen werden kann. Um also den zweiten Punkt P_2 zu finden, muss der Faktor t möglichst genau approximiert werden. Dazu wird mithilfe der Winkel θ_i (Eingangswinkel) und θ_t (Ausgangswinkel) zwischen den beiden Werten $d_{\vec{N}}$ und $d_{\vec{V}}$ interpoliert mit folgender Formel:

$$t = \frac{\theta_t}{\theta_i} \cdot d_{\vec{V}} + \left(1 - \frac{\theta_t}{\theta_i}\right) \cdot d_{\vec{N}} \tag{10}$$

Dabei ist $d_{\vec{V}}$ die Entfernung vom vordersten zum hintersten Polygon in diesem Pixel aus der Sicht der Kamera und $d_{\vec{N}}$ die Entfernung des vorderen Polygons zum Polygon in Richtung der inversen Normalen (Abb. 5). Die Entfernung $d_{\vec{V}}$ wird in jedem Frame berechnet indem das Objekt einmal mit umgedrehtem Depth-Test und einmal normal gerendert wird und jeweils die Z-Werte davon in einer Textur gespeichert werden. Von den Z-Werten der beiden Texturen wird dann die Differenz ausgerechnet. $d_{\vec{N}}$ wird in einer Vorberechnung für jedes Polygon bestimmt. Damit kann t berechnet und somit P_2 approximiert werden.

Um die Normale $\vec{N_2}$ zu finden, werden in einem vorherigen Rendering Pass die Normalen der nach hinten gerichteten Polygone in einer Textur als Farbwert gespeichert. Dann wird der approximierte hintere Schnittpunkt P_2 in die Bildebene projiziert um mit ihm die Textur zu indexieren und die Normale zu erhalten. Jetzt kann man $\vec{T_1}$, P_2 und $\vec{N_2}$ benutzen, um mit dem Brechungsgesetz den zweiten gebrochenen Vektor $\vec{T_2}$ zu berechnen. Mit diesem wird dann eine Environment-Textur indexiert und die zurückgegebene Farbe wird als Farbe für den Punkt P_1 benutzt.

Dieser Algorithmus hat allerdings auch einige Problemfälle. Totale interne Reflexion (Glei. 6) kann nicht abgebildet werden. Stattdessen wird hier einfach jeder Winkel, der größer ist als der kritische Winkel, „abgeklemmt“, sodass totale interne Reflexion nicht auftritt.

Außerdem kann es passieren, dass die Projektion des zweiten Schnittpunkts P_2 im Image-Space außerhalb der Silhouette des Objektes liegt, oder sogar außerhalb des Sichtfeldes. Das passiert, wenn der gebrochene Strahl die Seite des Objektes verlässt anstatt die

Hinterseite. Dann kann man aus der Textur mit den Normalen keinen Wert erhalten, da diese nur Werte innerhalb der Silhouette enthält. Wymans Lösung ist, in solch einem Fall von einer Normalen auszugehen, die senkrecht zum Kamera-Vektor steht.

Ein letztes Problem ist die Auswahl des korrekten Polygons bei einer hohen *Depth Complexity*. Wie in Abbildung 5 zu sehen, sollte der gebrochene Strahl das Objekt in den Punkten P_2, P_3 und P_4 schneiden. Mit seiner Methode sind allerdings nur ein eingehender und ein ausgehender Schnittpunkt möglich. Für die Schnittpunkt- und Normalenberechnung wird immer das hinterste Polygon benutzt, da dies die plausibelsten Ergebnisse erschafft.

3.4.2 Oliveira et al.

Hier stelle ich eine alternative Methode von Oliveira und Brauwers[OB07] vor. Sie besteht aus 3 Rendering Passes.

Im ersten Rendering Pass werden, ähnlich wie bei Wyman, die Normalen und Z-Werte der hintersten Polygone in einer Textur gespeichert.

Im zweiten Rendering Pass wird der Bereich der Z-Werte der nach hinten gerichteten Polygone, die im Viewing Frustum sind, berechnet. Dabei wird in Z_{min} der kleinste Z-Wert gespeichert und in Z_{max} der größte.

Im dritten Rendering Pass wird das tatsächliche Rendern durchgeführt. Zuerst wird wie bei Wyman (siehe 3.4.1) der erste gebrochene Vektor $\vec{T_1}$ an dem ersten Schnittpunkt P_1 berechnet. Um den zweiten gebrochenen Vektor $\vec{T_2}$ berechnen zu können, wird die Normale an dem Schnittpunkt von $\vec{T_1}$ mit den nach hinten gerichteten Polygonen benötigt.

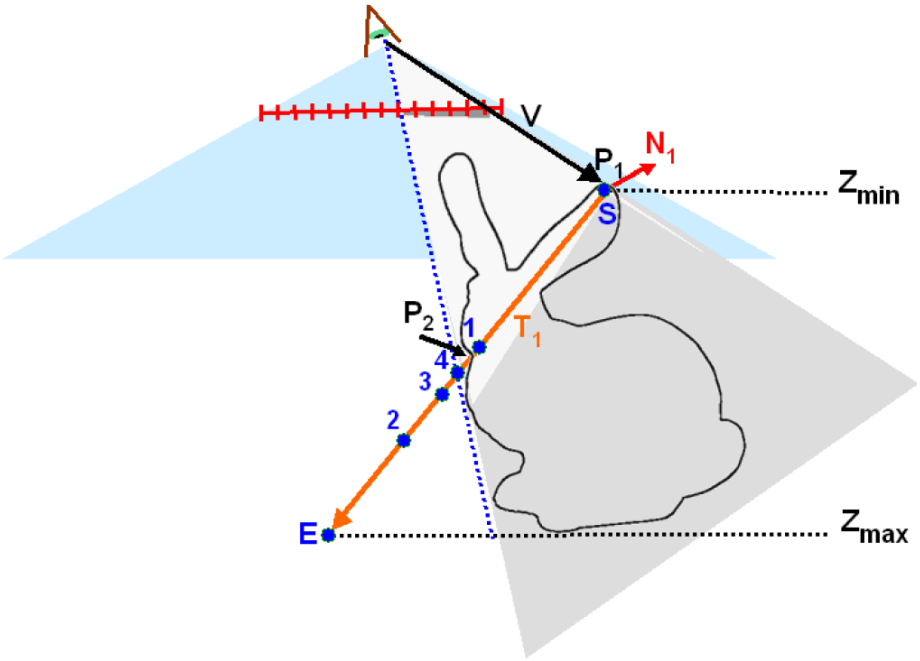

Abbildung 6: Beispiel zur Veranschaulichung von Oliveiras und Brauwers Algorithmus [OB07]

Um den zweiten Schnittpunkt zu approximieren, wird die sogenannte *Conservative Binary Search* benutzt. Dazu braucht man $\vec{T_1}$, den Startpunkt S, der am Anfang gleich P_1 ist und den Endpunkt E, der gleich dem Schnittpunkt von $\vec{T_1}$ und der Ebene $Z = Z_{max}$ ist (Abb. 6). Nun wird in einer Schleife der Mittelpunkt zwischen S und E berechnet und dessen Z-Wert wird mit dem Z-Wert verglichen, der in der Tiefen-Textur im ersten Rendering Pass gespeichert wurde. Falls der Z-Wert des berechneten Punktes kleiner ist, wird S gleich dem berechneten Punkt gesetzt (Punkt 1 in Abb. 6). Falls der Z-Wert größer ist, wird die Schrittweite zwischen S und E halbiert. In beiden Fällen wird danach die Schleife fortgesetzt, solange bis eine bestimmte Zahl an Durchläufen erreicht ist (siehe Pseudocode in Listing 1).

```
1  float3 P2 = S
2  float3 d = E–S
3
4  for i=0 to search_steps
5    d *= 0.5
6    float3 P = P2 + d
7    float2 texcoords = texture coordinates of P
8    storedDepth = indexTexture(depthTex,texcoords)
9    storedDepth *= Zfar
10
11   if P.z < storedDepth
12     P2 += d
13
14 return P2
```

Listing 1: Pseudocode für Conservative Binary Search

Ein Ausnahmefall ist, wenn der berechnete Mittelpunkt außerhalb der Silhouette des Objektes liegt, denn dann würde die Projektion des Punktes in Texturkoordinaten auf einen nicht validen Texel gemappt. Das sieht man in Abbildung 6 an Punkt 4. Wenn man diesen Punkt in Texturkoordinaten projiziert, wird kein Polygon aus der Tiefen-Textur getroffen. Um dem vorzubeugen, haben Oliveira und Brauwers ein Kriterium eingefügt, dass wenn der berechnete Mittelpunkt nicht auf einen validen Texel der Tiefen-Textur gemappt werden würde, der Punkt S nicht verändert, sondern stattdessen die Schrittweite halbiert wird.

Mit den in diesem Algorithmus berechneten Texturkoordinaten indexiert man die Normalentextur aus dem ersten Rendering Pass und holt sich die Normale an dem approximierten Schnittpunkt. Mit dieser und dem Vektor $\vec{T_1}$ kann man nun den zweiten gebrochenen Vektor berechnen, mit dem die Environment Map indexiert wird. So erhält man die Farbe an dem Punkt P_1.

Dieser Algorithmus hat ähnliche Schwachstellen wie der von Wyman. Totale interne Reflexion wird nicht simuliert. Wie auch bei Wyman wird hier ein Eintrittswinkel über dem kritischen Winkel abgeklemmt, sodass interne Reflexion nicht auftritt. Außerdem wird die Normale an dem zweiten Schnittpunkt immer von einem nach hinten gerichteten Polygon geholt. Wenn der Lichtstrahl durch ein nach vorne gerichtetes Polygon austritt, wird die falsche Normale benutzt.

3.5 Sonstige Methoden

Hier stelle ich noch ein paar andere, neuere Methoden vor, die etwas mit Lichtbrechung zu tun haben, aber andere Probleme zu lösen versuchen als ich.

Es gibt Lösungen für das Echtzeitrendering von Lichtbrechung bei Objekten mit rauer Oberfläche. Die Schwierigkeit ist hier, dass die Lichtstrahlen sehr viel mehr verbreitet werden bei der Brechung. Dadurch entsteht ein diffuseres Aussehen, wie z.B. bei Milchglas. Es gibt beispielsweise einen Algorithmus von de Rousiers et. al. [DRBS+11]. Dieser Algorithmus arbeitet wie auch der von Wyman und Oliveira et. al. im Image Space.

Eine anderer Aspekt, wenn man transparente Objekte realistisch rendern möchte, ist die sogenannte „Kaustik “. Kaustik ist ein Bereich in dem Lichtstrahlen gebündelt werden. Für das Rendering bedeutet dies, dass Lichtstrahlen an bestimmten Punkten auf umgebender Geometrie gebündelt werden, wenn sie vorher ein transparentes Objekt durchqueren. Die Position der Punkte, an denen das Licht gebündelt wird, hängt von der Form und dem Material des Objektes und der Position der Lichtquelle ab. Liktor und Dachsbacher [LD11] haben sich beispielsweise damit beschäftigt.

4 Benutzte Methode

Ich habe mich für mein Programm für die Methode von Li et al. [LGHW07] entschieden, die ich mit einstellbaren Parametern für den Benutzer versehen habe.

4.1 Beschreibung des Algorithmus

In einer Vorberechnung rendere ich eine Cubemap-Textur (siehe Kap. 2.2.2) vom Mittelpunkt des Objektes aus, die die Polygone des Objektes enthält, um später den Schnittpunkt des ersten gebrochenen Strahls mit dem Objekt finden zu können. Die Normalen werden in dem RGB-Kanal gespeichert und die Entfernung zum Mittelpunkt im Alpha-Kanal. Dabei wird der Depth-Test auf „Größer“, anstatt „Kleiner“ gestellt, um das Polygon zu rendern, welches am weitesten weg ist. Dieses muss bei einem korrekten Objekt immer ein Polygon sein, dessen Normale vom Mittelpunkt weg zeigt. Solch ein Polygon würde normalerweise nicht gerendert werden. Damit es trotzdem gerendert wird, wird das Backface Culling ausgeschaltet.

Da die Werte in einer Textur nur im Bereich $[0, 1]$ liegen können, die Normalen aber Werte zwischen -1 und 1 haben, werden sie zum Speichern in der Textur mit folgender Gleichung in den richtigen Wertebereich umgewandelt:

$$texNormal = normal + vec3(1, 1, 1))/2.0 \tag{11}$$

Die Entfernung zum Mittelpunkt teile ich vor dem Schreiben in die Textur durch die maximale Entfernung aller Polygone zum Mittelpunkt dieses Objektes um in den Bereich $[0, 1]$ zu kommen. Ich skaliere alle Objekte so, dass die maximale Entfernung zum Mittelpunkt genau 100 beträgt, so kann ich unabhängig vom Objekt immer durch 100 teilen, um in den korrekten Wertebereich umzuwandeln. Das Rendern der Normal-Cubemap muss nur einmal und nicht in jedem Frame geschehen.

In Abbildung 7 sieht man ein Beispiel für eine Normal-Cubemap, die nach dem ersten Schritt gerendert wurde. Sie hat 6 quadratische Seiten, also für jede Seite des Würfels eine.

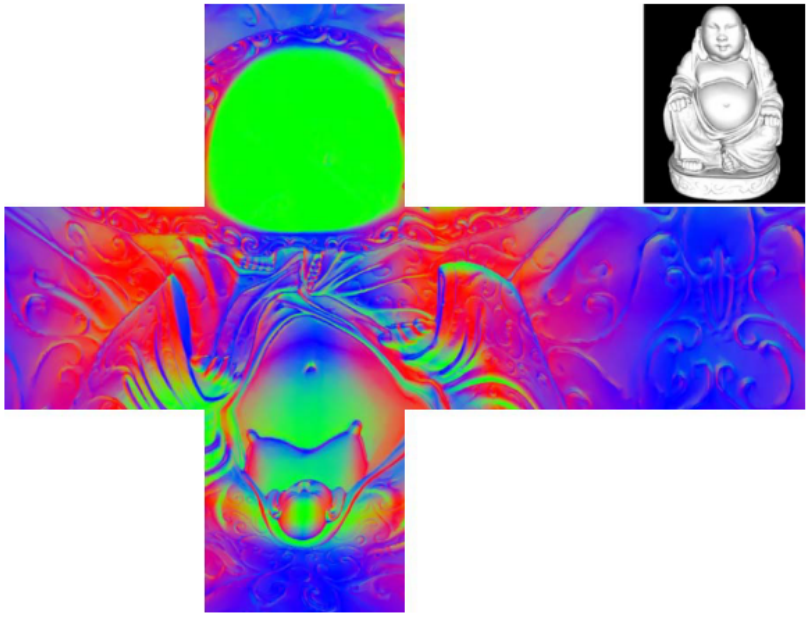

Abbildung 7: Beispiel für eine Normalen-Textur nach dem ersten Renderschritt [LGHW07]

Den Mittelpunkt berechne ich in einer Vorberechnung als Durchschnitt aller Vertices des Objektes. Das ergibt in den meisten Fällen einen für meine Zwecke sinnvollen Mittelpunkt des Objektes. Wichtig ist vor allem, dass der Mittelpunkt innerhalb des Objektes liegt (für mögliche Probleme siehe Kapitel 4.4).

Der eigentliche Renderloop beginnt, wie bei den bisher vorgestellten Methoden (siehe 3.4) mit dem Berechnen des ersten gebrochenen Vektors $\vec{T_1}$ an dem Schnittpunkt P_1. Dann muss ich den korrekten Parameter t für die Geradengleichung $P_2 = P_1 + t \cdot \vec{T_1}$ finden, um den zweiten Schnittpunkt P_2 berechnen zu können (Abb. 8). Das geht wie bereits gesagt nicht mit einer einfachen Schnittpunktberechnung wie beim Raytracing, da diese viel zu teuer wäre. Stattdessen benutze ich eine binäre Suche, um den zweiten Schnittpunkt zu

approximieren, die ich hier beschreibe.

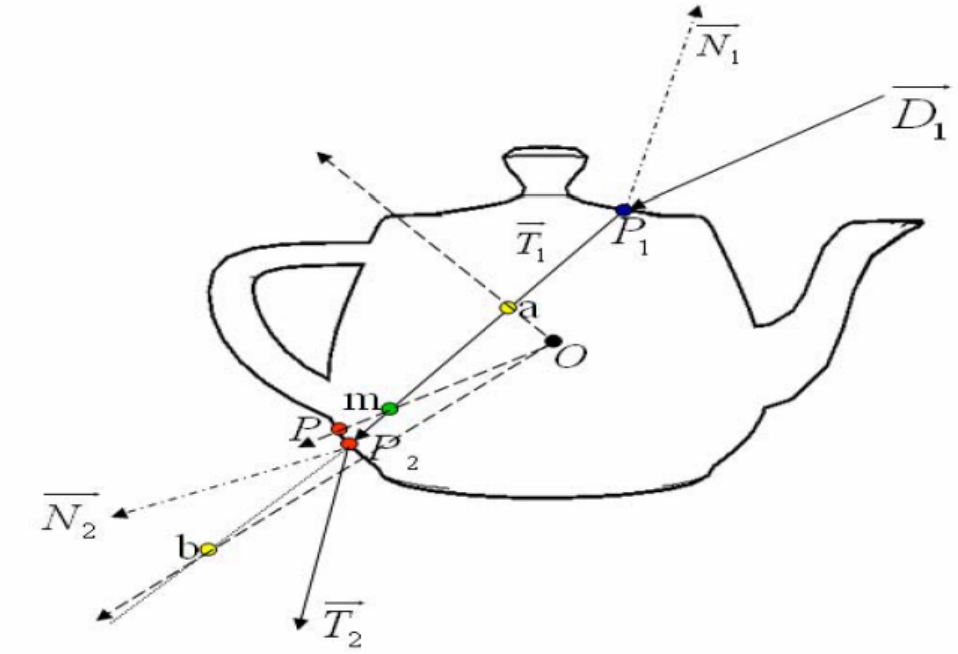

Abbildung 8: Beispiel zur Veranschaulichung des Algorithmus von Li et al. [LGHW07]

Für die binäre Suche setze ich zuerst die Grenzen des Geradenparameters t für die binäre Suche als $a = 0.01$ und $b = 250$, also als Punkte $A = P_1 + a \cdot \vec{T_1}$ und $B = P_1 + b \cdot \vec{T_1}$ auf der Geraden. Wichtig ist, dass die eine Grenze immer innerhalb des Objektes liegt und die andere immer außerhalb. 0.01 ist ein relativ geringer Wert und daher liegt dieser Punkt nur knapp neben P_1 innerhalb des Objektes. Die 250 kommt daher, dass das Objekt vorher so skaliert wurde, dass es maximal den Durchmesser 200 haben kann. Ein Punkt $B = P_1 + 250 \cdot \vec{T_1}$ liegt also immer außerhalb des Objektes.

In Abb. 9 wird das nochmal verdeutlicht. Alle Punkte auf der Geraden $\vec{T_1}$, inklusive P_2, können durch die Gleichung $P = P_1 + t \cdot \vec{T_1}$ dargestellt werden. Punkt A mit $t = 0.01$ liegt immer knapp innerhalb des Objektes. Da ich sicherstelle, dass der maximale Durchmesser aller benutzten Objekte 200 beträgt, liegt Punkt B mit $t = 250$ (, also 250 von P_1 entfernt) garantiert außerhalb des Objektes.

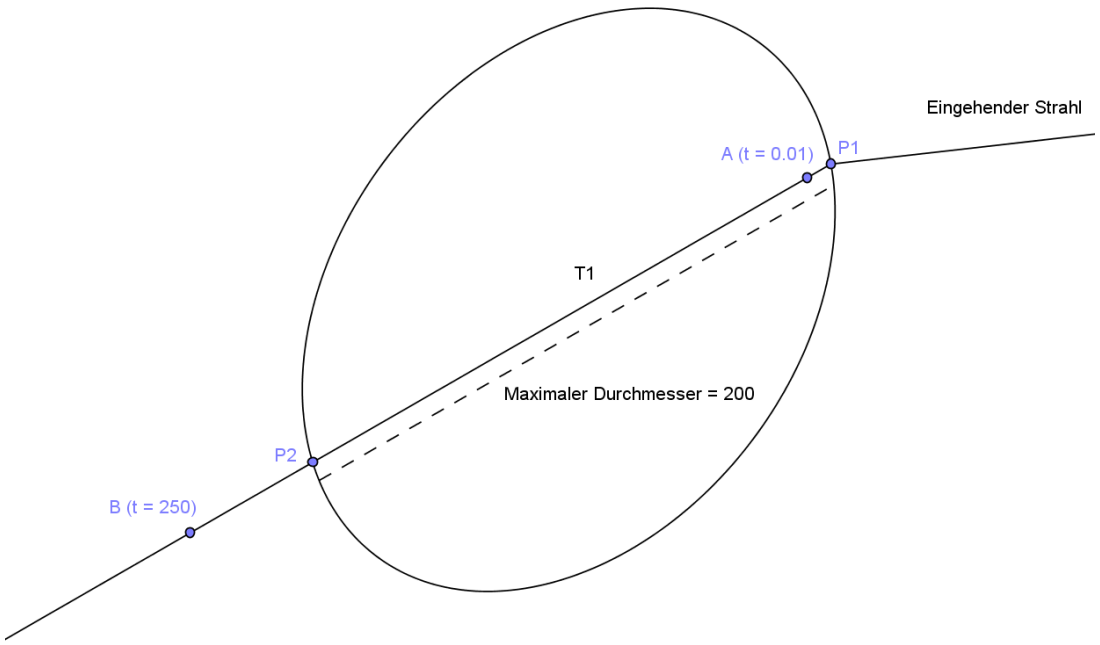

Abbildung 9: Veranschaulichung der Parameterwahl für die binäre Suche

Danach wird der Mittelpunkt M zwischen A und B berechnet (Punkt m in Abb. 8). Dann wird die Entfernung zwischen dem Mittelpunkt des Objektes O und M mit dem Tiefen-Wert verglichen, der am Anfang in der Normal/Depth-Cubemap im Alpha-Kanal gespeichert wurde. Wenn die Entfernung zwischen O und M kleiner ist, gehe ich davon aus, dass M innerhalb des Objektes liegt. In diesem Fall wird als nächster Schritt $A = M$ gesetzt. Wenn die Entfernung zwischen O und M größer ist als der Wert in der Normal/Depth-Cubemap, liegt M außerhalb des Objektes und ich setze $B = M$.

Dann beginnt die Schleife ab dem Berechnen des neuen Mittelpunkts von vorne. So nähert sich der Punkt M immer mehr dem Rand des Objektes an. Nach einer vorher festgelegten Anzahl an Schleifendurchläufen wird die Schleife beendet und M ist die berechnete Approximation des Schnittpunktes P_2 (siehe Pseudocode in Listing 2).

```
1   float a = 0.01
2   float b = 250
3   float3 m
4
5   for i=0 to binarySearchIterations
6     m = p1 + (a+b)/2 * t
7
8     //m is inside object
9     if indexTexture(normalCubeMap, m-O).a > length(m-O)
10      a = (a+b)/2
11
12    //m is outside object
13    else
14      b = (a+b)/2
```

Listing 2: Pseudocode für binäre Suche des zweiten Schnittpunktes

Um die Normale an Punkt M zu erhalten, indexiert man mit dem Vektor $\overrightarrow{OM}$ die Normal/Depth-Map. Mit der erhaltenen Normalen $\vec{N_2}$ und dem Vektor $\vec{T_1}$ berechnet man den zweiten gebrochenen Vektor $\vec{T_2}$ und indexiert damit die Environment-Map um die Farbe zu erhalten. Zusätzlich berechnet man noch den reflektierten Vektor indem man den Kameravektor an P_1 reflektiert. Mit diesem indexiert man auch die Environment-Map und mischt die beiden Farben mithilfe Schlicks Approximation (siehe Gleichung 3).

Simulation von totaler interner Reflexion ist auch möglich. Wenn der Winkel von $\vec{T_1}$ zu der Normalen $\vec{N_2}$ größer ist als der kritische Winkel (siehe Gleichung 6), wird $\vec{T_1}$ an der negativen Normalen reflektiert anstatt gebrochen zu werden. Für den neuen reflektierten Vektor $\vec{T_2}$ wird dann auf die gleiche Weise per binärer Suche der neue Schnittpunkt und der nächste gebrochene oder reflektierte Vektor $\vec{T_3}$ berechnet. Das geht solange weiter, bis entweder keine interne Reflexion mehr auftritt oder die vorher festgelegte maximale Anzahl an Schnittpunktberechnungen erreicht ist. Bei Li et. al. ist nur bis zu eine interne Reflexion möglich.

Anders als bei Li et. al. gibt es in meiner Implementierung einige zur Laufzeit einstellbare Parameter. Diese Parameter sind: Die Anzahl der Iterationen für die binäre Suche, der Brechungsindex und die Anzahl der möglichen internen Reflexionen.

Die für die Reflexion und Lichtbrechung benutzte Environment-Map wird jeden Frame neu gerendert indem vom Mittelpunkt aus eine Cube-Map mit dem Hintergrund und allen anderen Objekten gerendert wird. Damit können auch dynamische Szenen mit einem transparenten Objekt gerendert werden. Diese dynamische Environment-Map hat eine

geringere Auflösung als die originale Environment-Map, damit das Rendern jeden Frame nicht zu sehr die Performanz einschränkt. Man sieht aber keinen qualitativen Unterschied zu der originalen Skybox-Textur.

Ich habe mich für diese Methode entschieden, da ich hier nicht auf den Image-Space angewiesen bin. D.h. ich kann auch Polygone für meine Berechnungen benutzen, die außerhalb des Viewing Frustums liegen. Es ist mit diesem Algorithmus auch garantiert, dass jedem Polygon eine Normale zugeordnet werden kann, vorausgesetzt der Mittelpunkt liegt innerhalb des Objektes. Bei vielen Image-Space-Algorithmen ist das nicht so. Außerdem sind sehr einfach mehrere interne Reflexionen möglich.

4.2 Implementierung

4.2.1 Beschreibung der GUI

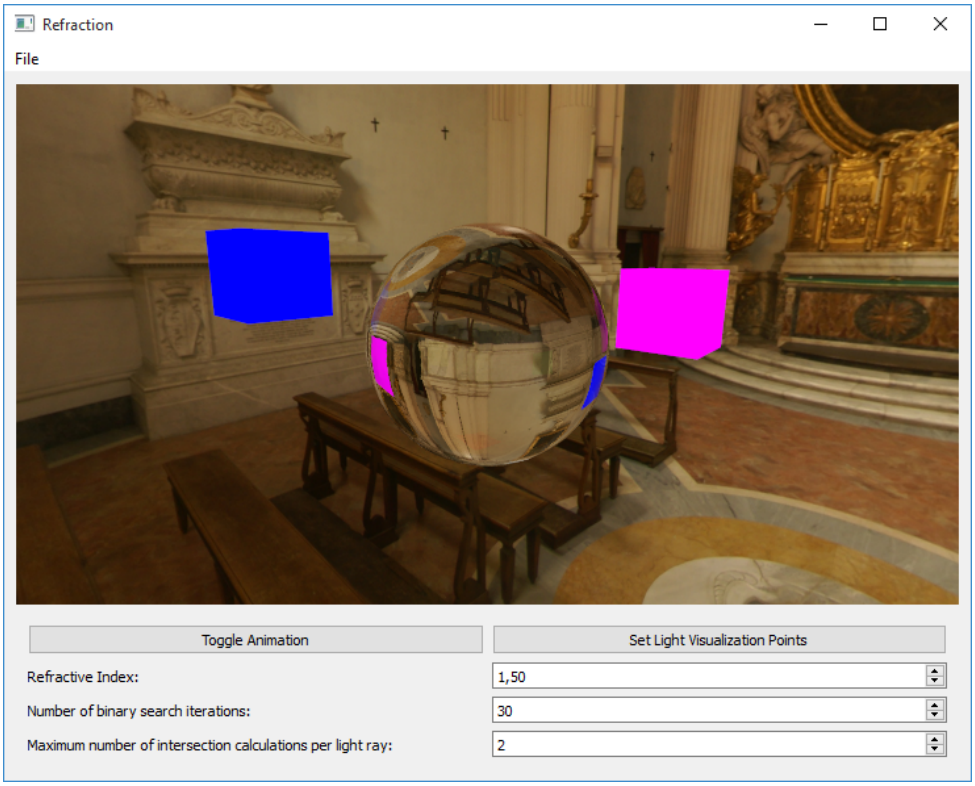

Abbildung 10: Screenshot der GUI

Die benutzte Beispielszene besteht aus einer Skybox [Per], dem transparenten Objekt, das im Raum schwebt und 6 verschieden farbigen Würfeln, die auf verschiedenen Bahnen um das Objekt kreisen. In Abbildung 10 sieht man diese Szene und die GUI des Programms.

Nach dem Starten des Programms kann man durch Klicken oben links auf den Button

„File → Open“ eine Objekt-Datei öffnen. Erlaubt sind alle OBJ-Dateien, die aus Dreiecken und Vierecken bestehen und Normalen enthalten. Man kann die Kamera um das Objekt drehen, indem man die Maus bewegt und dabei die linke Maustaste gedrückt hält. Mit der mittleren Maustaste oder ctrl verschiebt man die Kamera in z-Richtung. Mit der rechten Maustaste verschiebt man die Kamera in der xy-Ebene.

Mit dem Button „Toggle Animation“ hält man die Animation der rotierenden Würfel an oder startet sie wieder. In dem Feld „Refractive Index“ kann man den Brechungsindex einstellen. In dem Feld „Binary Search Iterations“ stellt man die Anzahl der Schleifendurchläufe für die binäre Suche des zweiten Lichtbrechungspunktes ein. Im Feld „Maximum number of intersection calculations for one light ray“ kann man die Anzahl der Lichtbrechungen bzw. Reflexionen bestimmen, die für einen Lichtstrahl berechnet werden sollen. Ein Wert von 1 entspricht dabei der Lichtbrechung an einer Oberfläche.

Das Programm hat außerdem einen Modus zum Visualisieren der Lichtstrahlen (siehe Kapitel 4.2.2). Dafür klickt man rechts auf den Button „Set Light Visualization Points“. Die Kamerasteuerung wird blockiert und man kann an beliebig vielen Punkten auf das Objekt klicken. Bei jedem Klick erscheint ein Lichtstrahl, der die Brechung des Lichtes von der Kamera auf den angeklickten Punkt darstellt. Wenn man nun noch einmal auf den gleichen Button klickt, kann man die Kamera wieder bewegen, wobei die Lichtstrahlen auf ihrem alten Platz bleiben und man kann sie sich aus jeder gewünschten Perspektive ansehen. In Abbildung 11 sind zwei solcher Lichtstrahlen dargestellt. Der Punkt oben rechts, an dem die Strahlen zusammenlaufen ist der Kamerapunkt.

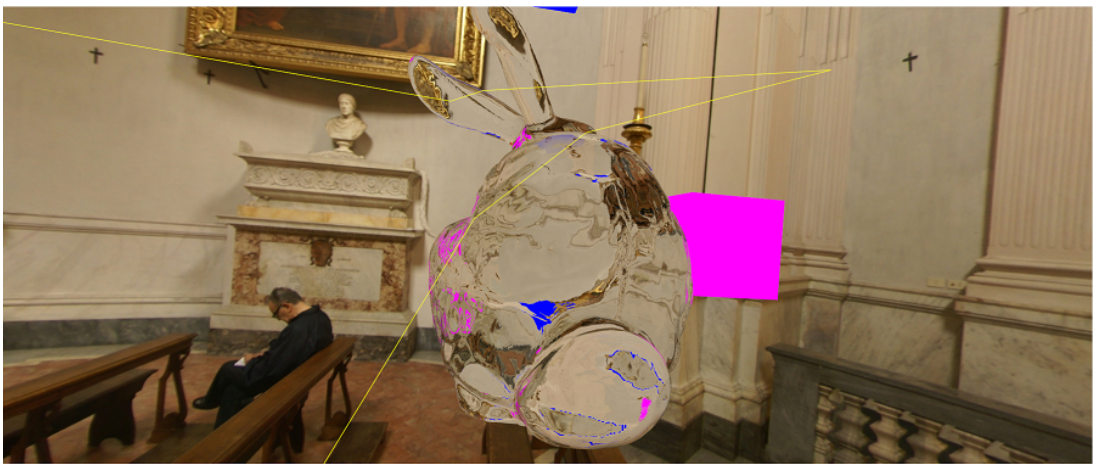

Abbildung 11: Visualisierung der Lichtstrahlen

4.2.2 Die Lichtstrahlvisualisierung

Wenn im Lichtvisualisierungsmodus auf das Fenster geklickt wird, wird die Position der Maus im 2D-Raum auf die Viewing Plane im 3D-Raum projiziert und es wird ein Strahl von der Kamera aus durch diesen Punkt auf die Szene geschossen. Dann berechne ich den ersten Schnittpunkt mit dem transparenten Objekt. Für die Berechnung benutze ich die Methode von Möller und Trumbore [MT97]. Das getroffene Polygon sowie die aktuelle Kameraposition wird gespeichert.

Das wird für alle Klicks gemacht, die im Lichtvisualisierungsmodus gemacht werden. Dann werden für die gespeicherten Punkte die korrekten Normalen ermittelt, indem die Normalen der Vertices der Polygonen interpoliert werden. Die Punkte mit den zugehörigen Normalen werden dann an die Graphik-Pipeline übergeben. Für diesen Modus benutze ich eigene Vertex-, Geometry- und Fragment Shader. Am wichtigsten ist hier der Geometry Shader. Dieser bekommt einen Punkt und eine Normale als Eingabe und gibt einen *linestrip* mit vier Vertices aus. Zuerst zeichnet er eine Linie zwischen der vorher gespeicherten Kameraposition und dem ersten Schnittpunkt. Dann wird der zweite Schnittpunkt mit dem in Kapitel 4.1 vorgestellten Algorithmus berechnet, der dann als dritter Punkt des linestrips fungiert. Von dort wird der zweite Brechungsvektor berechnet und als vierter Punkt des linestrips wird ein Punkt in weiter Entfernung in Richtung dieses Vektors verwendet.

4.2.3 Benutzte Techniken

Das Programm ist in C++ geschrieben. Für die Grafikbefehle wurde OpenGL 3.3 mit GLSL verwendet. Ich habe mich für OpenGL als Grafik-API entschieden, da es sehr verbreitet ist und mit vielen Betriebssystemen kompatibel ist. Die Version 3.3 habe ich gewählt, da viele Rechner damit kompatibel sind, aber trotzdem bereits der Geometry Shader existiert, durch den es möglich wird, alle sechs Seiten einer Cubemap in einem Renderingpass zu rendern.

Es gibt eine Windows- und eine Mac-Version des Programms. Der Sourcecode dieser beiden unterscheidet sich in einigen wenigen Punkten, um auf beiden Plattformen lauffähig zu sein, sie sind aber inhaltlich komplett identisch. Für das Rendern der Cubemap in Windows wurde die Library GLEW[5]verwendet.

Es hängt von der benutzten Grafikkarte und dem Treiber ab, wie gut das Anti-Aliasing ist. Das Programm versucht, 16 Samples pro Pixel zu benutzen. Wenn das von der Grafikkarte nicht unterstützt wird, werden weniger benutzt. Wie viele tatsächlich benutzt werden, sieht man nach dem Start des Programms in der Konsolenausgabe unter „Samples“.

[5]Quelle: `http://sourceforge.net/projects/glew/`

Die Seiten der Normal-Map haben die Auflösung 800 x 800 Pixel, die der dynamischen Environment-Map 1000 x 1000 und die der Environment-Map für die Umgebung 2048 x 2048 Pixel. Die GUI sowie die Methode für das Laden des Hintergrundbildes wurde mit QT5[6]erstellt.

Um alle Ausgaben, wie die FPS und andere Informationen sehen zu können, empfehle ich, das Programm von der Konsole aus zu starten.

4.2.4 Architektur

Hier gehe ich auf alle Klassen ein, die ich implementiert und benutzt habe.

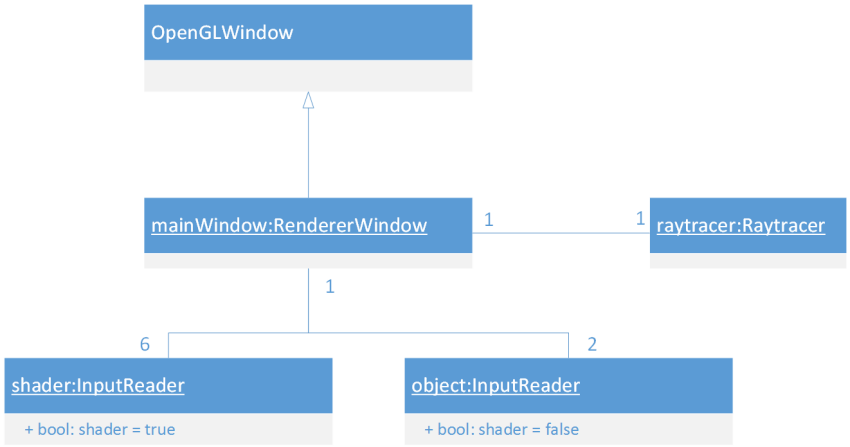

Abbildung 12: Ein Objektdiagramm des Programms

Die Klasse *OpenGLWindow* ist die Basisklasse für die Verwaltung des Renderingloops und des OpenGL-Kontextes. Diese Klasse basiert zu großen Teilen auf einem Beispiel der offiziellen QT-Website [Qt].

RendererWindow ist die Hauptklasse meines Programms. Sie erbt von *OpenGLWindow* und kümmert sich um den Inhalt des Renderingloops, die Generierung der Cubemaps und die Initialisierung und Anwendung der Shader. In der gleichen Klasse, aber in einer anderen Datei als der Rest, ist die Implementierung der Interaktionen mit der GUI, wie der Steuerung der Kamera und den Einstellungsmöglichkeiten der Renderingparameter. (Quelle für Code für Übergeben der Objekte an OpenGL: [OT])

Der *InputReader* kümmert sich um das Einlesen von Dateien. Es gibt eine static-Methode zum allgemeinen Einlesen von Dateien. Diese wird für die Shader-Dateien benutzt. Wenn man einen *InputReader* für das Einlesen einer Objekt-Datei verwendet, repräsentiert er dieses Objekt, d.h. er speichert die Vertices und die Normalen in einem *Indexed Face Set*.

[6]Quelle: `http://www.qt.io/`

Hier findet nach dem Einlesen auch die Berechnung des Mittelpunktes und die Skalierung des Objektes statt.

Der *Raytracer* berechnet den näherstehen Schnittpunkt eines Strahls mit der Szene, wenn für die Visualisierung der Lichtstrahlen auf das Objekt geklickt wird. Hier wird auch die interpolierte Normale berechnet.

In der Main-Methode werden die Buttons für die GUI erstellt und natürlich das Programm gestartet.

In Abbildung 12 sieht man ein Objektdiagramm, das die Beziehungen der Programmobjekte zur Laufzeit darstellt. Es gibt ein *RendererWindow*, das von *OpenGLWindow* erbt. Das *RendererWindow* kennt insgesamt 6 Shader, die vom *InputReader* eingelesen werden. Das sind jeweils 2 Vertex-, Geometry- und Fragment Shader. Drei für das Rendern der Objekte und der Umgebung und drei für das Rendern der Lichtstrahlen für die Visualisierung. Außerdem gibt es noch 2 *InputReader*, die Objekte einlesen. Es gibt einen, der den Würfel einliest für die Objekte, die um das transparente Objekt kreisen und für die Umgebungstextur, die auch auf einen Würfel gemappt wird. Der andere *InputReader* ist für das Einlesen des transparenten Objektes zuständig. Zuletzt gibt es noch einen *Raytracer*, der für das Berechnen des Schnittpunktes eines Strahls mit der Szene zuständig ist.

In Abbildung 13 ist ein beispielhaftes Sequenzdiagramm für den Anwendungsfall, in dem der Nutzer die Brechung der Lichtstrahlen visualisieren möchte. Nach dem Start des Programms öffnet der Nutzer eine obj-Datei. Das *RendererWindow* erstellt daraufhin einen *InputReader*, der die Datei liest und das Objekt zurückgibt. Das *RendererWindow* rendert dann das Objekt. Danach klickt der Nutzer auf den Button „Set Light Visualization Points“ um den Lichtvisualisierungsmodus zu aktivieren. Ohne eine Antwort abzuwarten, klickt der Nutzer auf das Objekt. Das *RendererWindow* schickt die Koordinaten der Maus zum *Raytracer*. Dieser berechnet das getroffene Polygon und gibt den Schnittpunkt zurück. Damit rendert das *RendererWindow* den Lichtstrahl. Zum Schluss beendet der Nutzer das Progamm.

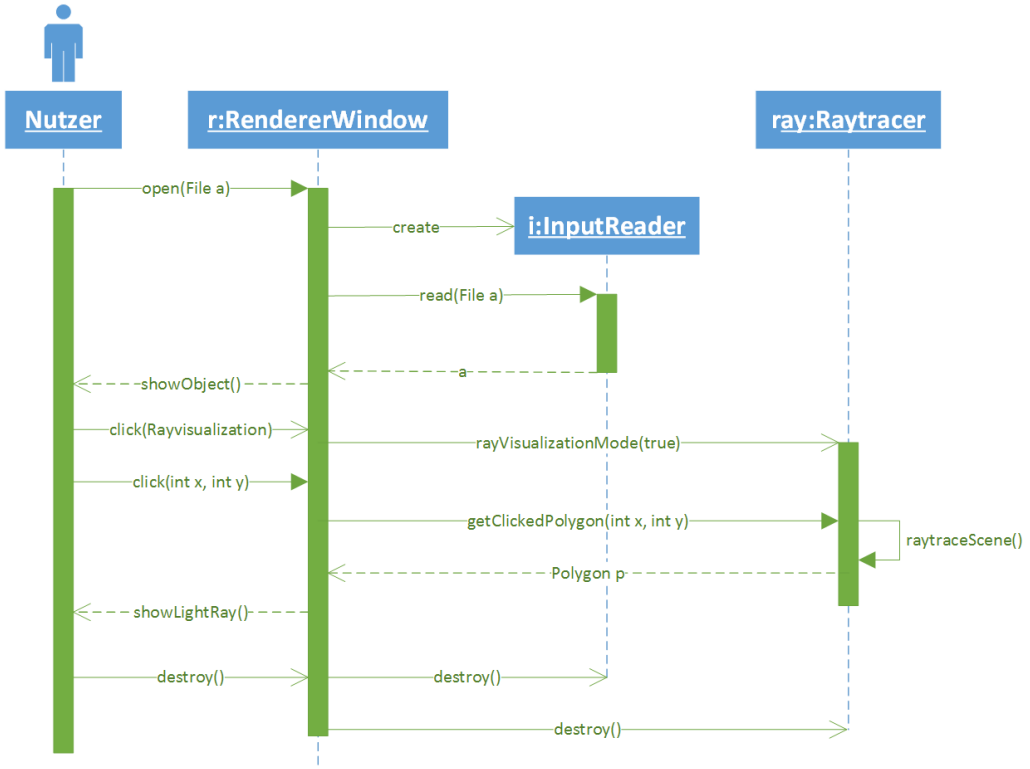

Abbildung 13: Ein Sequenzdiagramm für den Anwendungsfall des Visualisierens der Lichtstrahlen

4.3 Ergebnisse

Die Performanztests in Tabelle 2 sowie die dazugehörigen Screenshots wurden auf einem PC mit einem Intel Core i7-4770K Prozessor mit 3.5 GHZ und einer Nvidia Geforce GTX 780 bei einer Auflösung von 1400 x 583 Pixeln erstellt. Der Brechungsindex ist 1.5, die Anzahl der Iterationen für die binäre Suche ist 20 und die Anzahl der rekursiven Schnittpunktberechnungen ist 2. Die Vergleichstests wurden mit Lichtbrechung an einer Oberfläche (Single Surface Refraction, SSR) durchgeführt. Alle Tests wurden mit einer dynamischen Environment Map durchgeführt, die jeden Frame aktualisiert wird. Alle Objekte bis auf die Sphere sind aus der INRIA Gamma models database [INR]. Das Sphere-Modell kommt aus einem OpenGL-Forum [Din].

Modell	**# Polygone**	**FPS(Mein Programm)**	**FPS(SSR)**
Sphere (Abb. 14)	720	812	962
Armadillo (Abb. 15)	345.934	190	334
Bunny (Abb. 16)	69.664	294	553
JXKR (Abb. 17)	538.768	165	260

Tabelle 2: Performanztest mit verschiedenen Modellen als Vergleich von meinem Programm mit einseitiger Lichtbrechung(SSR)

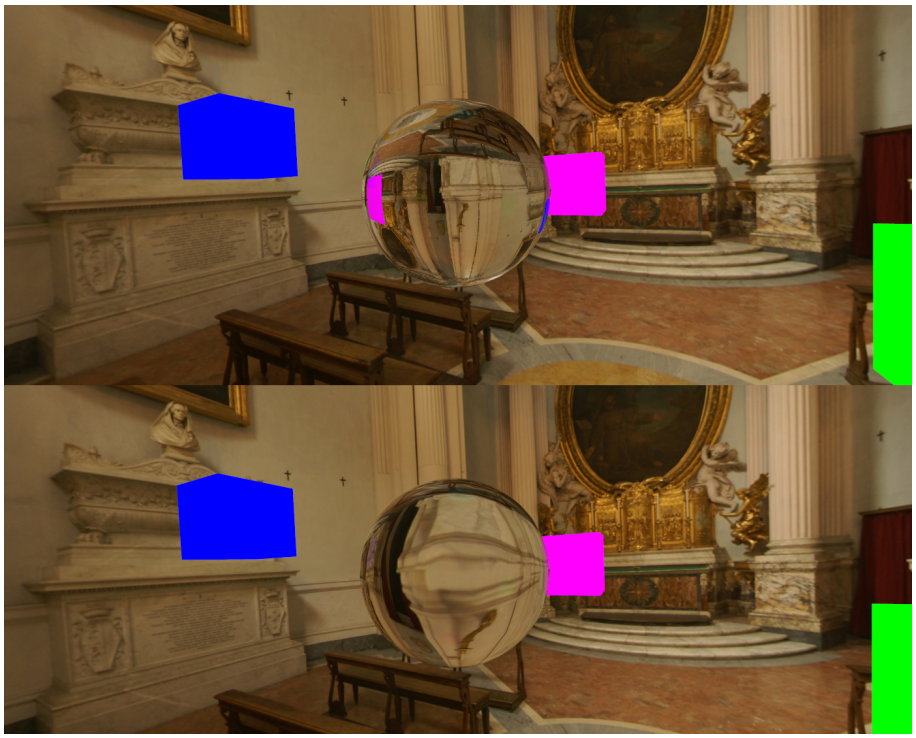

Abbildung 14: Sphere-Modell
Mein Programm oben, einseitige Lichtbrechung unten

Abbildung 15: Armadillo-Modell
Mein Programm oben, einseitige Lichtbrechung unten

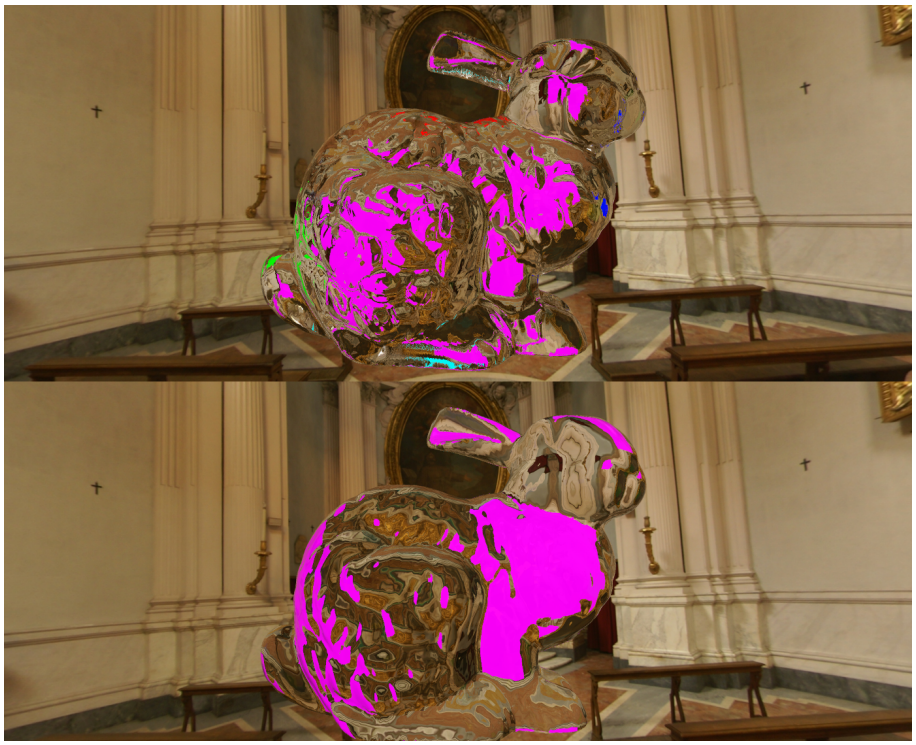

Abbildung 16: Bunny-Modell
Mein Programm oben, einseitige Lichtbrechung unten

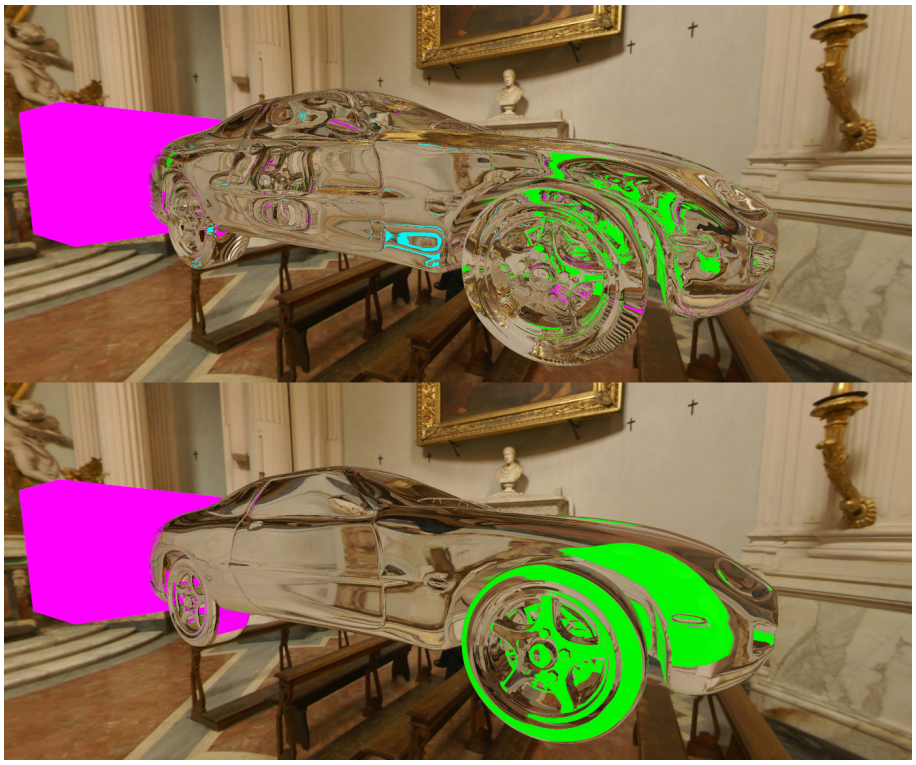

Abbildung 17: JXKR-Modell
Mein Programm oben, einseitige Lichtbrechung unten

Einseitige Lichtbrechung ist immer ca. 1.5 bis 2 Mal so schnell wie das Rendern bei meinem Programm. Mein Programm erzeugt allerdings viel realistischere Ergebnisse. Das sieht man beispielsweise in der Abbildung 16. Das Bild wird bei meiner Methode mehr verzerrt. Außerdem erkennt man auch interne Reflexionen, die bei der anderen Methode nicht möglich sind. Das sieht man sehr gut bei Abbildung 14 in der Reflexion des pinken Würfels in der Kugel, die man bei einseitiger Lichtbrechung nicht sieht. Bei komplexeren Modellen wie dem JXKR-Modell wird bei einem hohen Brechungsindex bei meinem Programm das Hintergrundbild sehr stark verzerrt. Das ist durchaus realistisch, da bei einem Modell mit vielen Polygonen selbst eine kleine Änderung des Einfallswinkels einen großen Einfluss auf den ausgehenden Vektor hat. Außerdem treten bei hohen Brechungsindizes öfter interne Reflexionen auf. Es lässt sich ein etwas „saubereres“ Bild erzeugen, wenn man einen niedrigeren Brechungsindex benutzt. Das habe ich in Abbildung 18 verglichen. Wie man sieht, lässt sich im unteren Bild der linke Hinterreifen besser erkennen als im oberen Bild, da die Lichtstrahlen da weniger stark gebrochen werden.

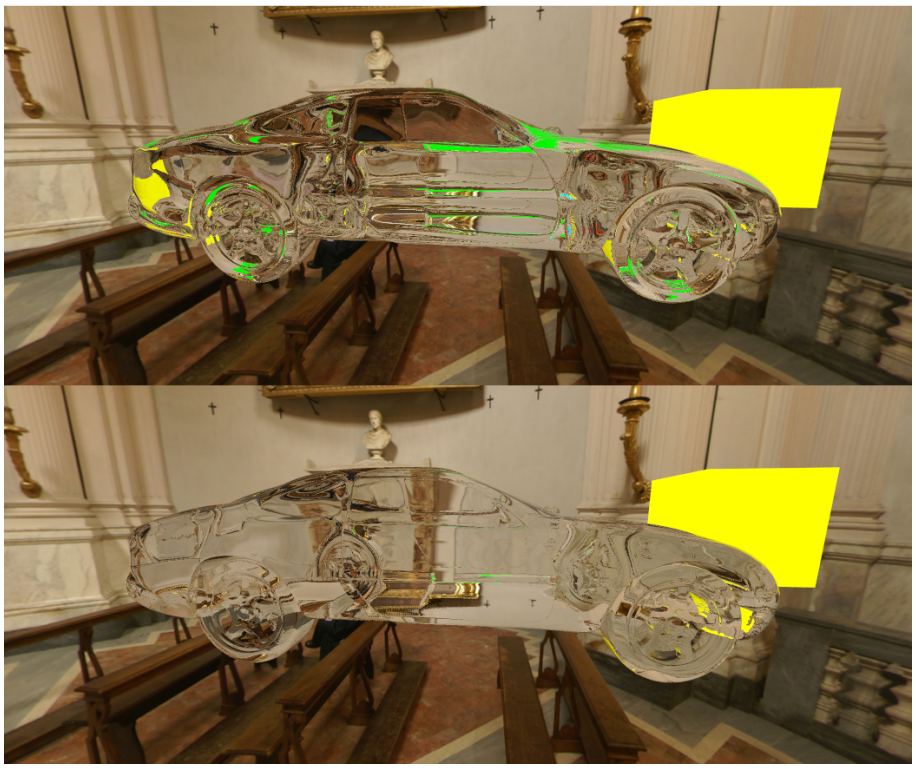

Abbildung 18: Vergleich zweier Brechungsindizes
Oben: 1.5, Unten: 1.1

In Abbildung 19 erkennt man nochmal sehr gut den Unterschied zwischen meiner zweiseitigen, und einseitiger Lichtbrechung. Bei zweiseitiger Lichtbrechung lässt sich bspw. der Kopf des Kaninchens auch von unten erkennen, bei einseitiger nicht.

Abbildung 19: Vergleich von ein- zu zweiseitiger Lichtbrechung beim Bunny-Modell
Oben: zweiseitig, Unten: einseitig

In Abb. 20 sind die Frames pro Sekunde für das Rendern von verschiedenen Levels of Detail beim Bunny-Model angegeben. Wie man sieht, wird das Gefälle des Graphen bei steigenden Polygonzahlen niedriger. Es lässt sich annehmen, dass dieser Trend bei höheren Polygonzahlen weiter besteht.

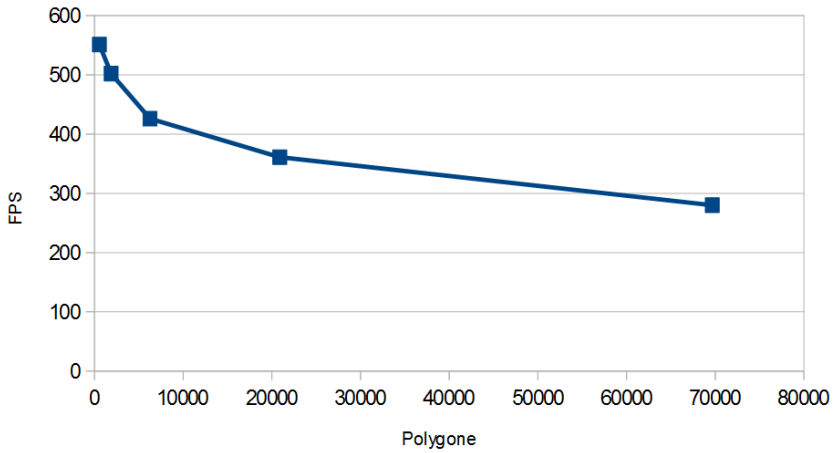

Abbildung 20: Performanz für verschiedene Levels of Detail bei dem Bunny-Model

In Abbildung 21 sieht man einen Vergleich von meinem Programm mit Raytracing. Die Bilder wurden beide mit 800 x 800 Pixeln gerendert mit einem Brechungsindex von 1.5. Die Anzahl der Schnittpunktberechnungen wurde bei der von mir benutzten Methode auf

3 gesetzt. Das Raytracing wurde mit 18 Samples pro Pixel und mit einer Rekursionstiefe von 10 berechnet. Mein Programm hat eine Performanz von 180 FPS, das Raytracing hat 2:52 Minuten zum Rendern gebraucht. Der Hintergrund für die beiden Bilder ist leider nicht exakt gleich, da ich für das Raytracing ein anderes Programm benutzt habe, aber man kann erkennen, dass sich die beiden Methoden von der Qualität her auf den ersten Blick nicht wirklich unterscheiden. Erst bei sehr genauem Hinsehen kann man bei dem linken Bild an dem Ohr ein paar Artefakte erkennen.

Abbildung 21: Vergleich von meinem Programm (links) zu Raytracing (rechts) beim Bunny-Modell

Mir ist außerdem noch aufgefallen, dass die Performanz relativ stark davon abhängt, wie viele Pixel auf dem Bild durch das Objekt abgedeckt sind. D.h. wenn ein Objekt näher an der Kamera ist, verringert das die FPS, bei Vergrößerung des Abstandes werden die FPS wieder höher. Das liegt daran, dass die meisten Berechnungen zur Lichtbrechung im Fragment Shader stattfinden und je mehr Pixel des Bildschirms vom Objekt abgedeckt sind, umso häufiger wird der Fragment Shader eingesetzt.

4.4 Limitierungen

Für relativ konvexe Objekte, die vom Mittelpunkt aus keine zu hohe Depth Complexity haben, erzeugt dieser Algorithmus sehr gute Ergebnisse bei einer guten Performanz, die weit besser ist, als z.B. bei Raytracing-Techniken. Er hat allerdings auch einige Nachteile.

In der Normal-/Tiefen-Map werden nur die hintersten Polygone gespeichert, d.h. das Licht wird immer nur am hintersten Rand gebrochen, selbst wenn dazwischen noch andere Übergänge zwischen zwei Medien liegen. Das sieht man sehr gut an Abbildung 18 unten. Dieses Modell hat tatsächlich auch einen modellierten Innenraum, auf dem Bild sieht es

aber so aus, als wäre nichts in dem Auto.

Bei sehr konkaven Objekten kann es vorkommen, dass der berechnete Mittelpunkt M nicht innerhalb des Objektes liegt. Das sieht man in Abbildung 22. Das Problem in diesem Beispiel ist, dass nach dem Rendern der Normal-Map der gestrichelte Bereich keine Normalen enthält, sodass einem Punkt in diesem Bereich keine Normale zugeordnet werden kann. Außerdem wird in dem Beispiel dem Punkt G inkorrekt die Normale des Punktes H zugeordnet. Solche Objekte können für diese Methode leider nicht benutzt werden.

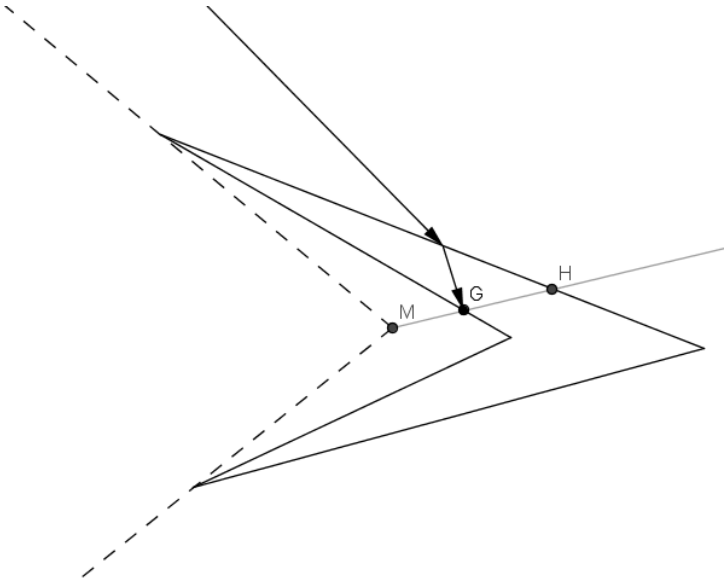

Abbildung 22: Beispiel für das Problem der Mittelpunktberechnung bei konkaven Objekten

Ein anderes Problem tritt auf, wenn die Depth Complexity vom Mittelpunkt aus an einer Stelle des Objektes sehr hoch ist. In Abbildung 23 sieht man ein Objekt mit einer hohen Depth Complexity an einer Stelle. Die Gerade oben ist eine Darstellung der Normalen-Textur. Die Vektoren schneiden die Gerade genau an den Grenzen eines Pixels. Allen Punkten auf den rot markierten Linien werden die gleiche Normale zugeordnet. Dabei entstehen unschöne Artefakte, wie z.B. Linien (Abb. 24). Dieses Problem lässt sich mit einer höher aufgelösten Normalen-Textur minimieren, aber es lässt sich mit diesem Algorithmus nie völlig beseitigen.

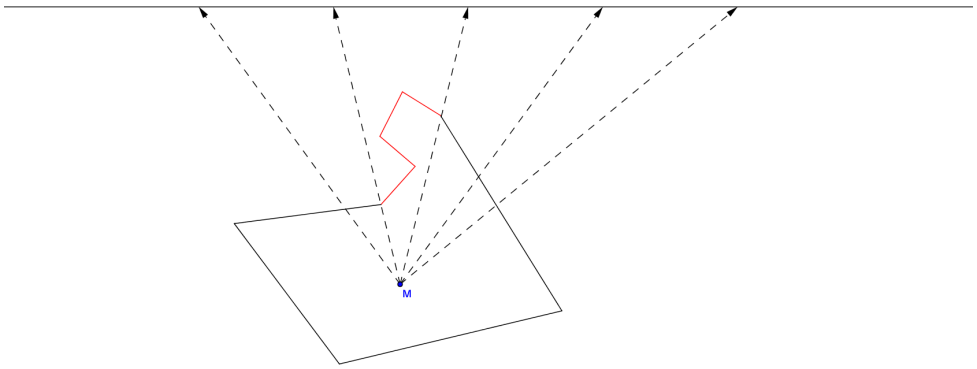

Abbildung 23: Beispiel für das Problem der falschen Zuordnung von Normalen

Abbildung 24: Beispiel für Artefakte bei falscher Zuordnung von Normalen

Ab ca. 3 bis 4 internen Reflexionen und einem hohen Brechungsindex kommt es bei etwas komplexeren Objekten zu Artefakten in Form von Noise bei der Lichtbrechung (Abb. 25). Dieses Bild wurde mit 5 Rekursionen für interne Reflexion und einem Brechungsindex von 1.5 aufgenommen. Wie man sieht, kommt es in dem Bild zu unschönem Noise der verschiedenen gebrochenen Farben. Das kommt daher, dass bei sehr vielen Reflexionen selbst ein nur wenig veränderter Eingangsvektor am Ende einen völlig anderen Ausgangsvektor erzeugt. Ein anderer Grund könnte sein, dass es bei jeder Berechnung eines Schnittpunkts zu kleinen Rundungsfehlern kommt, die sich bei wiederholter Reflexion zu einer größeren Ungenauigkeit propagieren. Das sieht man auch daran, dass sich bei der Lichtstrahlvisualisierung bei vielen Reflexionen der ausgehende Strahl bei Bewegen der Kamera verändert. Da die Rechnung aber jedes mal die gleiche ist, kann das nur an Rechenungenauigkeiten liegen. Die Performanz verschlechtert sich bei zu vielen Rekursionen auch deutlich. Deshalb empfehle ich, je nach Komplexität des Objektes nur 2 bis 4 Lichtbrechungen zuzulassen.

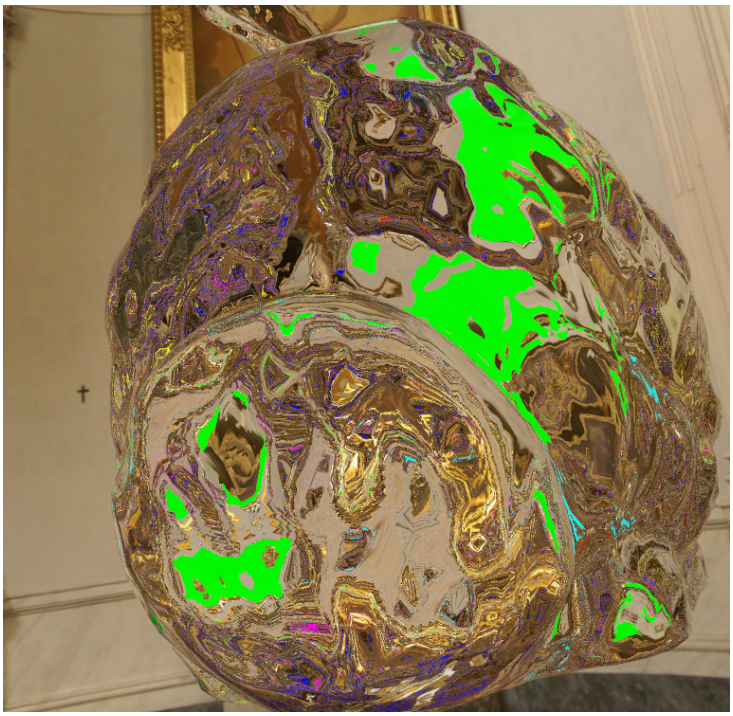

Abbildung 25: Beispiel für Artefakte bei vielen internen Reflexionen

Des weiteren sind Objekte, die man durch das transparente Objekt sieht, genau genommen nicht an der korrekten Position. Das liegt daran, dass für die Farbe an einem Pixel nur die Richtung des ausgehenden Vektors entscheidend ist, nicht aber der Punkt, an dem dieser austritt. Es wird für die Berechnung der Farbe eine Environment-Map auf einem Würfel benutzt, der den gleichen Mittelpunkt hat, wie das Objekt selbst. Für das Indexieren der Environment-Map „zeigt“ ein Vektor, der dem Mittelpunkt entspringt, auf den gewünschten Pixel. Für das Programm heißt das nun, dass es sozusagen immer davon ausgeht, dass der ausgehende Vektor seinen Ursprung im Mittelpunkt hat, unabhängig davon, wo der Vektor tatsächlich aus dem Objekt austritt. Das kann dazu führen, dass die Position und die Entfernung zur Kamera von Objekten hinter dem transparenten Objekt verfälscht werden.

Das fällt allerdings nur bei sehr flachen Objekten, wie Würfeln auf, die das Licht nicht stark brechen. Außerdem sieht man diesen Effekt auch, wenn man den Brechungsindex auf 1 setzt. Da das in der Realität aber dem Brechungsindex von Luft entsprechen würde und somit gar kein Übergang zwischen zwei Materialien auftreten würde, ist das vernachlässigbar.

5 Zusammenfassung und zukünftige Arbeiten

Es ist möglich, in einer Echtzeitanwendung Lichtbrechungseffekte zu erzeugen, die sich auf den ersten Blick nicht von von Raytracing erzeugten Bildern unterscheiden, aber dabei eine sehr viel bessere Performanz aufweisen. Die Methoden zur Berechnung im Image-Space scheinen auch vielversprechend zu sein, da viele der in Kapitel 4.4 beschriebenen Probleme dort nicht auftreten, dafür haben sie aber andere. Z.B. benötigen sie meist mehr Rendering Passes und es können nur Polygone in die Berechnungen einfließen, die innerhalb des Sichtfeldes der Kamera liegen.

In der Zukunft könnte man eine Methode entwickeln, mehr als zwei Lichtbrechungen zu berechnen, wobei dann eben nicht nur die hintersten, sondern mehr/alle Polygone berücksichtigt werden. Außerdem könnte man versuchen, einen besseren Algorithmus zu finden, um den Mittelpunkt des Objektes zu berechnen, sodass der Mittelpunkt garantiert innerhalb des Objektes liegt oder man benutzt direkt mehrere Punkte von denen eine Normal-Map gerendert wird. So könnte man auch das Problem lösen, dass vielen Punkten die gleiche Normale zugeordnet wird. Die leichte Verfälschung der Position eines Objektes hinter einem transparenten Objekt ließe sich unter Umständen mit einer zusätzlichen binären Suche lösen, die den genauen Punkt findet, an dem der ausgehende Strahl das andere Objekt trifft.

Bleibt noch die Frage, wie hilfreich dieser Algorithmus ist. In den meisten Echtzeitanwendungen, wie Spielen, werden transparente Objekte oft nur durch Lichtbrechung an einer Oberfläche oder sogar nur durch zufällige Verzerrung der hinter dem Objekt liegenden Szene dargestellt. Das reicht bei nicht sehr auffälligen transparenten Objekten oft auch aus und ist sogar noch performanter, als die hier vorgestellte Methode. Aber ich denke, dass Applikationen, in denen transparente Objekte auffällig eingebaut sind, sehr von diesem Algorithmus profitieren können.

6 Quellenverzeichnis

[AK] Atulit Kumar, Harry G.: *Multisampling and Motion Blur in a Ray Tracer*. http://www.andrew.cmu.edu/user/hgifford/projects/msaa.pdf. – Abgerufen am: 3.10.2015

[BBE+03] Becker, Frank-Michael ; Bossek, Dr. H. ; Engelmann, Lutz ; Ernst, Dr. C. ; Fanghänel, Dr. G. ; Höhne, Heinz ; Kalenberg, Dr. A.: *Formelsammlung - Formeln, Tabellen, Daten ; Mathematik, Physik, Astronomie, Chemie, Biologie, Informatik ; [bis zum Abitur ; mit CD-ROM und Internetportal]*. Berlin : Duden Schulbuch, 2003. – ISBN 978–3–898–18700–8

[Din] Dinev, Ilian: *Sphere*. https://www.opengl.org/discussion_boards/showthread.php/176762-looking-for-a-simple-sphere-obj-file. – Abgerufen am: 7.10.2015

[DRBS+11] De Rousiers, Charles ; Bousseau, Adrien ; Subr, Kartic ; Holzschuch, Nicolas ; Ramamoorthi, Ravi: Real-Time Rough Refraction. In: *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*, 2011

[INR] INRIA: *Gamma models database*. https://www.rocq.inria.fr/gamma/gamma/download/download.php. – Abgerufen am: 7.10.2015

[LD11] Liktor, Gábor ; Dachsbacher, Carsten: Real-time Volume Caustics with Adaptive Beam Tracing. In: *Symposium on Interactive 3D Graphics and Games*. New York, NY, USA : ACM, 2011 (I3D '11). – ISBN 978–1–4503–0565–5, 47–54

[LGHW07] Li, Shuai ; Gao, Yujian ; Hao, Aimin ; Wang, Lili: Interactive Two-sided Refraction for Dynamic Object on GPU. In: *Proceedings of the Fourth International Conference on Image and Graphics*, 2007 (Proceedings of the Fourth International Conference on Image and Graphics), S. 997–1003

[MT97] Möller, Tomas ; Trumbore, Ben: Fast, Minimum Storage Ray-triangle Intersection. In: *J. Graph. Tools* 2 (1997), Oktober, Nr. 1, 21–28. http://dx.doi.org/10.1080/10867651.1997.10487468. – DOI 10.1080/10867651.1997.10487468. – ISSN 1086–7651

[OB07] Oliveira, Manuel M. ; Brauwers, Maicon: Real-time refraction through deformable objects. In: *Proceedings of the 2007 Symposium on Interactive 3D*

graphics and games, 2007 (Proceedings of the 2007 Symposium on Interactive 3D graphics and games), S. 89–96

[ORM08] Overbeck, R. ; Ramamoorthi, R. ; Mark, W.R.: Large ray packets for real-time Whitted ray tracing. In: *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*, 2008, S. 41–48

[OT] OpenGL-Tutorial: *Tutorial 2*. http://www.opengl-tutorial.org/beginners-tutorials/tutorial-2-the-first-triangle/. – Abgerufen am: 3.10.2015

[Per] Persson, Emil: *Lancellotti Chapel*. http://www.humus.name. – Abgerufen am: 3.10.2015

[Pra] Pranckevicius, Aras: *Refraction*. http://wiki.unity3d.com/index.php?title=Refraction. – Abgerufen am: 14.9.2015

[Qt] Qt Company Ltd.: *OpenGL Window Example*. http://doc.qt.io/qt-5/qtgui-openglwindow-example.html. – Abgerufen am: 3.10.2015

[Sch94] Schlick, Christophe: An Inexpensive BRDF Model for Physically-based Rendering. In: *Computer Graphics Forum* 13 (1994), S. 233–246

[Wym05] Wyman, Chris: An Approximate Image-space Approach for Interactive Refraction. In: *ACM SIGGRAPH 2005 Papers*. New York, NY, USA : ACM, 2005 (SIGGRAPH '05), 1050–1053

[Zim99] Zimmons, Paul: Final Project: Spherical, Cubic, and Parabolic Environment Mappings. In: *COMP238: Raster Graphics* (1999). http://cgvr.cs.uni-bremen.de/teaching/cg_literatur/Spherical,%20Cubic,%20and%20Parabolic%20Environment%20Mappings.pdf

Anhang

Die beiliegende CD enthält:

- Diese Arbeit als PDF
- Das Programm als ausführbare Binaries für Mac und Windows
- Die hier gezeigten Screenshots in höherer Auflösung
- Der Sourcecode des Programms für Mac und Windows
- Der Sourcecode der benutzten Shader
- Die benutzen OBJ-Modelle sowie das für die Environment Map benutze Bild

Erklärung

Hiermit versichere ich, dass ich meine Abschlussarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen der Arbeit, die wörtlich oder sinngemäß aus fremden Quellen entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt oder an anderer Stelle veröffentlicht.

Datum: .....

(Unterschrift)