
Leopold-Franzens-Universität Innsbruck
Institut für Informatik
Forschungsgruppe Infmath Imaging

Bakkalaureatsarbeit

# Implementierung eines Raytracers

Markus Trenkwalder

4. Februar 2005

Betreuer: Univ.-Prof. Dr. Otmar Scherzer

# Inhaltsverzeichnis

# 1 Mathematische Notation

In dieser Bakkalaureatsarbeit werden mehrere mathematische Formeln beschrieben, welche auch Vektorausdrücke enthalten. Da man in der Literatur unterschiedliche Schreibweisen für die mathematischen Operationen bei Vektoren und Matrizen finden kann und dies zur Konfusion führen könnte wird in diesem Abschnitt auf die in dieser Arbeit verwendete Notation eingegangen.

Die folgende Tabelle fasst die Schreibweise für verschiedene Parameter zusammen:

| **Typ** | **Notation** | **Beispiel** |
|---|---|---|
| Winkel | Kleiner griechischer Buchstabe | $\alpha, \phi, \gamma$ |
| Skalar | Klein und kursiv | $a, b, t, v_x, v_y, v_z$ |
| Vektor oder Punkt | Klein und fett | **x, v, p** |
| Matrix | Groß und fett | **M, T(t), X** |
| Ebene | $\pi$ : Vektor und Skalar | $\pi : \mathbf{n} \cdot \mathbf{x} + d = 0$ |
| Funktion | Groß und kursiv | *I, R, F* |

Vektoren und Matrizen werden folgendermaßen geschrieben:

$$\mathbf{v} = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix}, \ \mathbf{M} = \begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{pmatrix}$$

eine Matrix kann auch durch ihre Spaltenvektoren beschrieben werden:

$$\mathbf{M} = (\mathbf{u} \quad \mathbf{v} \quad \mathbf{w}) = \begin{pmatrix} u_x & v_x & w_x \\ u_y & v_y & w_y \\ u_z & v_z & w_z \end{pmatrix}$$

Die folgende Tabelle fasst die verwendeten Operatoren zusammen:

| **Operator** | **Beschreibung** |
|---|---|
| $\cdot$ | Skalarprodukt von Vektoren ($\mathbf{a} \cdot \mathbf{b}$) |
| $\times$ | Kreuzprodukt von Vektoren ($\mathbf{a} \times \mathbf{b}$) |
| $\otimes$ | Komponentenweises Vektorprodukt |
| $\mathbf{v}^T$ | Transponierter Vektor |
| $\lVert \cdot \rVert$ | Länge des Vektors |
| $\lvert \cdot \rvert$ | Determinante einer Matrix oder Spatprodukt |

# 2 Was ist Ray tracing?

Ray tracing steht für „Strahlverfolgung“ und ist ein populäres Verfahren der 3D Computergrafik um (meist fotorealistische) Darstellungen von dreidimensionalen Szenen zu erstellen. Bilder werden erzeugt indem die Pfade von Lichtstrahlen in einer Szene verfolgt und die Interaktionen der Strahlen mit den Objekten der Szene berechnet werden. Durch dieses Verfahren können somit auf relativ einfache Weise Effekte wie Reflexion, Refraktion (Lichtbrechung) und Schatten berechnet werden.

Um eine Darstellung einer Szene (z.B. Raum mit Tisch, Stühlen und einer Lampe) zu erstellen wird beim so genannten *Backward Ray tracing* eine Bildebene vor die virtuelle Kamera gelegt. Anschließend werden Strahlen ausgehend vom Augpunkt durch jedes Pixel der Bildebene gelegt und zurückverfolgt um zu bestimmen welches Licht entlang dieses Strahles auf die Bildebene projiziert wird und wie das entsprechende Pixel gefärbt werden muss.

Trifft man beim Zurückverfolgen des Strahls auf spiegelnde oder transparente Objekte wird der Algorithmus rekursiv aufgerufen um den reflektierten bzw. gebrochenen Strahl zu verfolgen.

Anders ist dies beim *Forward Ray tracing*; hier werden die Strahlen von den Lichtquellen in der Szene emittiert und bestimmt, wie sich die Energie der Strahlen in der Szene verteilt. Dieses Verfahren, auch unter dem Begriff *Photon Mapping* bekannt, kann zusammen mit dem Backward Ray tracing kombiniert werden um auch diffuse Beleuchtungseffekte oder Linsen, welche das Licht bündeln, simulieren zu können.

Ein anderes Verfahren, welches auch globale Beleuchtungseffekte und diffuse Oberflächen simulieren kann nennt sich *Monte Carlo Ray tracing*. Der Unterschied zum normalen Backward Ray tracing besteht darin, dass die Rekursion nicht bei diffusen Oberflächen stoppt, sondern mehrere diffus reflektierte Strahlen generiert und weiterverfolgt werden. Die berechneten Bilder sind realistischer, jedoch ist der Aufwand der Berechnung um ein Vielfaches höher.

Das Ray tracing Verfahren ist recht einfach zu implementieren und erlaubt es besonders realistische Bilder zu erstellen, hat jedoch einen entscheidenden Nachteil. Ray tracing ist wegen der großen Anzahl an Strahlen, welche berechnet werden müssen, äußerst langsam. Aus diesem Grund wird es nicht in Interaktiven Anwendungen verwendet, sehr wohl aber in der Filmindustrie.

Ein großer Vorteil des Ray tracings ist, dass es sich gut parallelisieren lässt und dabei linear skaliert was heißt, dass bei doppelter zur Rechenleistung auch die doppelte Geschwindigkeit erzielt wird. Ein deutsches Forschungsprojekt [6] nutzt diese Tatsache um Ray tracing in Echtzeit auf einem Cluster mit 48 Athlon MP Prozessoren zu realisieren.

# 3 Der Ray tracing Algorithmus

Nachdem wir nun einiges über das Ray tracing erfahren haben können wir den zugrunde liegenden Algorithmus des Backward Ray tracings genauer betrachten. Der folgende Pseudocode beschreibt Kernalgorithmus eines jeden Raytracers.

```
1:     for each pixel (x,y) in image {
2:       Ray = CalculatePrimaryRay(x,y)
3:       NearestT = INFINITY
4:       NearestObject = NULL
5:
6:       for every object O in scene {
7:         T = Intersect(Ray, O)
8:         if T != INFINITY and T < NearestT {
9:           NearestT = T
10:          NearestObject = O
11:        }
12:      }
13:
14:      if NearestObject == NULL {
15:        PixelColor = BackgroundColor;
16:      } else {
17:        PixelColor = Shade(NearestObject, NearestT, Ray);
18:      }
19:    }
```

Für jedes Pixel des zu berechnenden Bildes wird ein Strahl erzeugt, der ausgehend vom Augpunkt durch das Zentrum des Pixels verläuft. Dieser Strahl wird Primärstrahl genannt und im Folgenden verwendet um Schnittpunkte mit Objekten zu berechnen.

Als nächstes muss das Objekt und der Schnittpunkt, der dem Ausgangspunkt des Strahls am nächsten gelegen ist gefunden werden. Dies wird in den Zeilen 3 bis 12 realisiert, indem für jedes Objekt der Szene überprüft wird ob es einen Schnittpunkt gibt, der näher als der bereits bekannte Schnittpunkt liegt.

Sind alle Objekte überprüft muss bestimmt werden mit welcher Farbe das Pixel auf der Bildebene eingefärbt werden soll. Gibt es keinen Schnittpunkt mit einem Objekt in der Szene wird die Hintergrundfarbe zugewiesen. Andernfalls berechnet die so genannte *„Shading Funktion“* anhand der Informationen über den Schnittpunkt mit dem Objekt den Farbwert für das Pixel.

Das Resultat der Shading Funktion kann von verschiedenen Materialeigenschaften des Objektes beeinflusst werden. Es ist auch möglich den Ray tracing Algorithmus rekursiv aufzurufen um z.B. Reflexionen zu berechnen, wobei man jedoch bedenken muss, dass es zu keiner endlosen Rekursion kommen darf. Dieses Problem kann durch einen Parameter, der die maximale Rekursionstiefe angibt, gelöst werden.

# 4 Die Shading Funktion

Die Shading Funktion ist zentraler Bestandteil des Ray tracing Algorithmus, denn sie ist (abgesehen von der Form) für das Aussehen der Objekte und somit des gesamten Bildes verantwortlich.

Die Shading Funktion hat die Aufgabe den Farbwert, der einem Pixel zugeordnet werden soll, zu ermitteln und muss dazu die Materialeigenschaften der Objekte, sowie Lichtquellen und sogar die Orientierung andere Objekte im Raum (z.B. für Schatten) berücksichtigen.

Ziel der Computergrafik und damit Aufgabe der Shading Funktion ist es die so genannte *„Rendering Equation"* zu lösen bzw. eine möglichst gute Approximation der Lösung zu bestimmen. Die Lösung der Rendering Equation beschreibt wie Licht auf die Objekte einer Szene fällt und bestimmt damit wie die einzelnen Pixel der Bildebene einzufärben sind.

Die Intensität des reflektierten Lichts an einem Punkt einer Oberfläche kann durch ein Integral über die Hemisphäre oberhalb der Oberfläche von einer Beleuchtungsfunktion *I* und einer Reflexionsfunktion *R* beschrieben werden.

$$I(\phi_r, \theta_r) = \int_{\phi_i} \int_{\theta_i} L(\phi_i, \theta_i)\, R(\phi_i, \theta_i, \phi_r, \theta_r)\, \mathrm{d}\phi_i \mathrm{d}\theta_i$$

wobei ($\phi_i$, $\theta_i$) den Eingangwinkel des Strahl und
($\phi_r$, $\theta_r$) den Ausgangwinkel des reflektierten Strahls beschreibt.

Die Beleuchtungsfunktion *L* bestimmt welches Licht aus einer bestimmten Richtung auf das Objekt fällt. Dabei werden Schattenwurf, diffuse Reflexionen und andere physikalische Effekte berücksichtigt. Die Reflexionsfunktion *R* ist abhängig vom Material des Objektes und beschreibt wie stark und in welche Richtung Licht, das auf das Objekt strahlt, reflektiert wird.

Dieses Integral ist im Allgemeinen zu komplex um analytisch gelöst zu werden. Selbst eine numerische Lösung dieses Integrals ist extrem aufwändig und deshalb wird in der Computergrafik die Realität durch ein so genanntes *Shading Model* modelliert und zusätzlich einige vereinfachende Annahmen getroffen:

- In der Computergrafik werden meist nur punktförmige Lichtquellen verwendet. Durch diese Annahme ist *L* immer null außer für bestimmte diskrete Richtungen, in welchen sich die Punktlichtquellen befinden. Dadurch lässt sich das Integral über L durch eine Summe über diese diskreten Richtungen ersetzten. Diese Vereinfachung verursacht *harte Schatten*.

- Die Beleuchtungsfunktion wird durch die Annahme einer *ambienten Lichtquelle* weiter vereinfacht. Hier nimmt man an, dass man für alle Richtungen, die nicht zu einer Lichtquelle gehören, das Licht aus jener Richtung durch einen konstanten, richtungsunabhängigen Term, beschreiben kann.
- Auch die Reflexionsfunktion wird vereinfacht. Meist werden in der Computergrafik nur diffuse Oberflächen beleuchtet von Punktlichtquellen und die perfekte Reflexion, wie sie bei Spiegeln zu finden sind berechnet. Für diffuse Oberflächen hat diese Annahme zur Folge, dass die Reflexionsfunktion unabhängig von der reflektierten Richtung ($\phi_r$, $\theta_r$) gemacht wird. Bei Spiegeln ist $R$ immer null außer für die Richtung des reflektierten Strahls.
- Eine weitere Vereinfachung der Computergrafik ist, dass diese meist nur mit den RGB Farbwerten rechnet und damit vernachlässigt, dass es Licht mit den verschiedensten Wellenlängen gibt. Phänomene der Natur, welche von der Wellenlänge des Lichts abhängig sind werden dabei durch eine wellenlängenunabhängige Approximation modelliert.

Wir wissen nun, dass die Shading Funktion für die Berechnung der verschiedenen darzustellenden Effekte verantwortlich ist. Wie diese Effekte nun tatsächlich berechnet werden können, wird in den nächsten Abschnitten erläutert.

## 4.1 Lambertsches Kosinusgesetz

Perfekt diffuse Oberflächen, so genannte *lambertsche Oberflächen*, sind komplett matt und reflektieren Licht, welches auf diese scheint, gleichmäßig in alle Richtungen. Nach dem lambertschen Kosinusgesetz ist die Intensität des reflektierten Lichtes einer solchen diffusen Oberfläche proportional zum Kosinus des Winkels zwischen der Oberflächennormalen (Vektor, der senkrecht auf der Oberfläche steht) und dem Vektor zur Lichtquelle. Der Kosinus des Winkels zwischen diesen beiden Vektoren kann mit der folgenden Formel bestimmt werden

$$\cos\phi = \mathbf{n} \cdot \mathbf{l}$$

wobei $\mathbf{n}$ der Normalenvektor und $\mathbf{l}$ der Vektor zur Lichtquelle ist und beide Vektoren normalisiert sind.

Für Winkel $\phi > \pi/2$ ist die diffuse Beleuchtung null, da die Oberfläche der Lichtquelle abgewandt ist. Weiters müssen noch die Oberflächeneigenschaften sowie die Farbe des Lichts berücksichtigt werden. Wenn ein Lichtstrahl auf eine Oberfläche trifft wird ein Teil des Spektrums des Lichtstrahls absorbiert und der andere Teil reflektiert. So absorbiert beispielsweise eine rote Oberfläche alle Wellenlängen des Lichtspektrums außer jenen, welche das Objekt rot erscheinen lassen.

In der Computergrafik wird im RGB Farbmodell gerechnet. Man kann der Lichtquelle und dem Material somit jeweils einen dreiwertigen Vektor $\mathbf{m}_{diff}$ bzw. $\mathbf{s}_{diff}$ zuweisen, welcher die Farbe des Lichts bzw. Materials kodiert. Die Intensität des von der diffusen Oberfläche reflektierten Lichts kann nun, unter der Berücksichtigung aller genannten Aspekte, als dreiwertiger Vektor für die RGB Farben mit folgender Formel ausgedrückt werden:

$$\mathbf{i}_{diff} = \max((\mathbf{n} \cdot \mathbf{l}),0)\mathbf{m}_{diff} \otimes \mathbf{s}_{diff}$$

Der Operator $\otimes$ steht dabei für die komponentenweise Multiplikation zweier Vektoren.

## 4.2 Reflexion

Die meisten Oberflächen in der Natur weisen neben diffusen Eigenschaften zusätzlich auch Spiegelungen auf. Die meisten Objekte reflektieren Licht nicht gleichmäßig in alle Richtungen, sondern reflektieren mehr Licht in Richtung des Reflexionsvektors. Durch diesen Umstand ergeben sich so genannte Glanzpunkte auf Oberflächen verschiedenster Materialien wie Plastik, Metall oder Glas. Ein perfekter Spiegel ist hierbei das Extrem. Er weist überhaupt keine diffusen Eigenschaften auf und reflektiert Licht ausschließlich in die Reflexionsrichtung.

Um die Intensität der Glanzpunkte (engl. *specular highlights*) zu bestimmen wird in der Computergrafik die Formel von Blinn verwendet.

$$i_{spec} = (\mathbf{n} \cdot \mathbf{h})^{m_{shi}} = (\cos\theta)^{m_{shi}}$$

$$\mathbf{h} = \frac{\mathbf{l} + \mathbf{v}}{\|\mathbf{l} + \mathbf{v}\|}$$

Diese Formel nimmt an, dass alle Vektoren normalisiert sind. Der Vektor **n** beschreibt die Oberflächennormale, **l** den Vektor von der Oberfläche zur Lichtquelle und **v** den Vektor von der Oberfläche zum Betrachter des Objektes. Der Vektor **h** wird als Halbvektor bezeichnet, weil er den Winkel zwischen den Vektoren **l** und **v** halbiert. Geometrisch interpretiert beschreibt der Vektor **h** die Normale jener Ebene, welche das Licht der Lichtquelle direkt in das Auge reflektieren würde. Ist der Winkel zwischen **h** und **n** also klein oder sogar null, wird mehr Licht von der Lichtquelle in das Auge reflektiert und der entsprechende Punkt der Oberfläche erscheint heller. Der Parameter $m_{shi}$ ist ein Parameter des Materials und beschreibt die Stärke des Glanzes und die Größe des Glanzpunktes. Der Farbwert für den Glanzpunkt wird ermittelt indem der Term $i_{spec}$ mit der Glanzfarbe des Materials verrechnet wird.

Diese Formel kommt auch in der Echtzeitgrafik zum Einsatz. Sie wird von OpenGL verwendet, da diese Formel die etwas aufwändigere, explizite Berechnung des Reflexionsvektors vermeidet.

Um beim Ray tracing perfekt reflektierende Oberflächen zu simulieren muss man den Reflexionsvektor explizit berechnen. Nachdem man den Reflexionsvektor berechnet hat wird der Lichtstrahl in der berechneten Richtung weiterverfolgt um den reflektierten Gegenstand zu ermitteln.

Der Reflexionsvektor **r** kann mit der folgenden Formel berechnet werden, wobei wieder angenommen wird, dass alle Vektoren in der Formel normalisiert sind.

$$\mathbf{r} = \mathbf{i} - 2(\mathbf{n} \cdot \mathbf{i})\mathbf{n}$$

In dieser Formel steht der Vektor **n** wieder für die Oberflächennormale, an welcher der Vektor gespiegelt wird. Der Vektor **i** steht für den Vektor, der aus der Betrachtungsrichtung kommt und auf die Oberfläche trifft.

Will man Oberflächen simulieren, welche beispielsweise teilweise diffus und teilweise reflektierend sind, müssen beide Komponenten berechnet und anschließend entsprechend gewichtetet werden (z.B. 20% reflektierend und 80% diffus). Dabei ist darauf zu achten, dass insgesamt maximal 100% verrechnet werden dürfen, da die Oberflächen sonst zu hell aussehen würden.

## 4.3 Refraktion (Lichtbrechung)

Neben der Reflexion gibt es in der Natur auch die Lichtbrechung, auch Refraktion genannt, welche bei Transparenten Objekte wie Wasser oder Glas auftritt. Dieser Effekt kann beobachtet werden, wenn man in ein mit Wasser gefülltes Glas einen Stift hält und von der Seite in das Glas sieht. Der Stift scheint verbogen zu sein.

Trifft ein Lichtstrahl auf ein Objekt mit einer höheren optischen Dichte (z.B. Lichtstrahl in der Luft trifft auf Glas), so wird der Lichtstrahl gebrochen und abhängig vom so genannten *Refraktionsindex* der beiden Materialien (Luft und Glas) etwas abgelenkt. Wandert ein Strahl von einem Material mit höherem Refraktionsindex wieder zu einem Material mit einem niedrigeren (vom Glas wieder in die Luft) wird der Strahl erneut abgelenkt. Hier kann es aber auch zur so genannten *Totalreflexion* kommen, welche auftritt, wenn der Austrittswinkel zu flach ist. In einem solchen Fall wird der Lichtstrahl in das Objekt zurückreflektiert. Dieser Effekt wird bei Glasfaserkabeln angewandt, wodurch das Licht im Kabel bleibt.

Eine in der Computergrafik vernachlässigte Tatsache ist, dass die Stärke der Lichtbrechung abhängig von der Wellenlänge des Lichtes ist. Durch diese Vernachlässigung können Prismeneffekte, bei denen ein Lichtstrahl in die Spektralfarben aufgeteilt wird nicht korrekt simuliert werden.

Der Zusammenhang zwischen dem eintreffenden und dem abgelenkten Vektor wird durch das snelliussche Brechungsgesetz beschrieben.

$$n_1 \sin(\theta_1) = n_2 \sin(\theta_2)$$

Wobei $n_i$ den Refraktionsindex des jeweiligen Mediums und $\theta_i$ den Winkel gemessen zur Oberflächennormalen beschreibt.

In praktischen Anwendungen hat man den Richtungsvektor des auf das Material auftreffenden Lichtstrahls, den Normalvektor auf die Oberfläche und den Refraktionsindex der beiden beteiligten Materialien gegeben. In „An Introduction to Ray tracing“ [3] wird Heckberts Methode, den refraktierten Strahl zu bestimmen, besprochen. Eine leicht effizientere Methode von Xavier Bec [5] ist folgende:

$$\begin{aligned} r &= \frac{n_1}{n_2} \\ w &= -(\mathbf{i} \cdot \mathbf{n})r \\ k &= 1 + (w - r)(w + r) \\ \mathbf{t} &= r\mathbf{i} + (w - \sqrt{k})\mathbf{n} \end{aligned}$$

Der Vektor **i** ist der Eingangsvektor und **n** die Normale. Die Formel nimmt an, dass diese bereits normalisiert sind. Das Ergebnis der Formel ist der gebrochene Vektor **t**. Hier ist zu beachten, dass es nur eine Lösung gibt, wenn $k \geq 0$ ist. Ist $k < 0$ tritt der Fall der totalen internen Reflexion auf.

## 4.4 Fresnelsches Reflexionsgesetz

Trifft ein Lichtstrahl auf eine Glaskugel, so wird dieser gebrochen und ein abgelenkter Strahl wandert durch die Glaskugel. Ein anderer Teil des Lichtes wird aber auch von der Kugel reflektiert und wandert nicht durch diese hindurch. Das fresnelsche Gesetzt beschreibt wie viel des Lichts reflektiert und wie viel refraktiert wird. Der fresnelsche Term ist vom Winkel des auftreffenden Lichtstrahls, vom Refraktionsindex des Materials sowie der Wellenlänge und Polarisierung des Lichts abhängig. Der Fresnel Effekt kann besonders gut bei elektrischen Nichtleitern wie Glas aber auch bei nichttransparenten Materialien wie Papier beobachtet werden, indem man in einem sehr flachen Winkel auf ein Objekt schaut. Je flacher der Winkel wird, desto reflektierender wird das Material.

Haines und Möller [2] präsentieren eine von der Wellenlänge und Polarisierung des Lichts unabhängige Formel zur Berechnung des fresnelschen Terms:

$$c = \mathbf{v} \cdot \mathbf{h}$$
$$g = \sqrt{n^2 + c^2 - 1}$$
$$F = \frac{1}{2}\frac{(g-c)^2}{(g+c)^2}\left(1 + \frac{[c(g+c)-1]^2}{[c(g-c)+1]^2}\right)$$

Für die Vektoren **v** und **h** gilt wieder die Annahme, dass sie normalisiert sind. Der Vektor **v** ist der Vektor zum Betrachter, es hier aber auch der Vektor zur Lichtquelle verwendet werden, da dies am Resultat nichts ändert. Der Vektor **h** ist der Halbvektor zwischen der Lichtquelle und dem Betrachter. Der Parameter *n* gibt den Refraktionsindex des Materials an. Will man beispielsweise eine glatte Glasoberfläche simulieren, kann man **h** der Oberflächennormalen gleichsetzen, da das reflektierte Licht ausschließlich aus der Reflexionsrichtung kommt.

## 4.5 Schatten

Bis jetzt wurde besprochen wie Oberflächen mit dem Licht interagieren. Es gibt jedoch auch Stellen in einer Szene, welche nicht von allen Lichtquellen beeinflusst werden. Von solchen Stellen sagt man, dass sie im Schatten liegen.

Die Berechnung und Darstellung von Schatteneffekten verleiht einer Szene mehr Tiefe und war bis vor einigen Jahren noch nicht im Echtzeitcomputergrafikbereich vertreten. Dort haben sich mehrere Techniken entwickelt um Schatten darzustellen, welche aber alle als Spezialeffekte zu sehen sind.

Beim Ray tracing ist es einfach und intuitiv Schatten hinzuzufügen und bedarf keiner besonderen Spezialbehandlung. Bevor man die diffuse Komponente oder Glanzpunkte, welche von einer Lichtquelle auf einer Oberfläche erzeugt werden, berechnet muss ermittelt werden ob die Lichtquelle den entsprechenden Punkt auf der Oberfläche eines Objektes überhaupt „sehen“ kann und demnach den Punkt beleuchtet. Gibt es kein freies Blickfeld vom Punkt auf der Oberfläche zur Lichtquelle, so liegt der Punkt im Schatten und die Lichtquelle hat keine Auswirkung auf die Beleuchtung dieses Punktes. Um zu bestimmen ob sich ein Punkt im Schatten einer Lichtquelle befindet muss beim Ray tracing Verfahren lediglich ein Strahl, ein so genannter *Schattenfühler*, vom Punkt auf der Oberfläche zur Lichtquelle verfolgt werden um zu sehen ob es auf dem Weg dorthin ein Objekt gibt, das den Weg blockiert. Ist der Weg blockiert liegt das Objekt im Schatten.

Diese Vorgehensweise ist einfach hat aber einige Probleme, da sie nur Punktlichtquellen berücksichtigt und somit *harte Schatten* produziert. Auch die Tatsache, dass zwischen der Lichtquelle und dem zu berechnenden Punkt spiegelnde

oder transparente Objekte liegen könnten, welche die Färbung des Lichts oder den Weg des Strahls beeinflussen wird vernachlässigt.

Halbschatten (*Penumbra*) können durch Verwendung von flächenförmigen Lichtquellen simuliert werden, indem man genügend viele Strahlen vom Oberflächenpunkt zu zufällig gewählten Punkten auf der Lichtfläche verfolgt und bestimmt welcher Anteil der Lichtquelle vom Ausgangspunkt sichtbar sind. Wie viele Strahlen es effektiv benötigt um eine gute Annäherung an die tatsächliche Realität zu erstellen hängt dabei von einigen Faktoren wie der Größe und Entfernung der Lichtquelle ab und kann nicht einfach beantwortet werden. Im Allgemeinen erzielt man bei der Verwendung von mehr Strahlen bessere Ergebnisse auf Kosten der Geschwindigkeit.

Sollen auch transparente Objekte, Linsen oder Spiegel, welche sich zwischen den Lichtquellen und den zu beleuchtenden Punkten befinden können korrekt berücksichtigt werden kann man auf Techniken wie *Photon mapping* oder das aufwändige *Monte Carlo Ray tracing* zurückgreifen.

## 4.6 Weitere Effekte

Es gibt eine Reihe von weiteren interessanten grafischen Effekten, die mit dem Ray tracing Verfahren erzeugt werden können, von denen einige bereits den Sprung in die Echtzeitgrafik geschafft haben. Diese Effekte werden im Folgenden kurz angesprochen.

### Texturen

Texturen sind nicht wirklich ein „Effekt“. Sie werden bereits seit sehr langem auch im Echtzeitbereich verwendet und werden verwendet um einem Objekt grafische Details zu geben ohne dieses geometrisch komplexer zu machen. So kann beispielsweise auf eine flache Ebene ein sich wiederholendes Bild von Fliesen gelegt werden um den Effekt eines Fliesenbodens zu simulieren.

### Prozedurale Texturen

Prozedurale Texturen, erst seit einiger Zeit auch auf Grafikbeschleunigern über so genannte *Pixelshader* realisierbar, sind Texturen, welche nicht als Bild gespeichert sind, sondern zur Laufzeit dynamisch berechnet werden können. So ist es möglich das Muster eines Schachbretts auf einer Ebene darzustellen, indem für jedes Pixel der darzustellende Farbwert durch eine Funktion berechnet wird.

### Bump/Normal mapping

Ein weiterer Effekt, der es in den Echtzeitbereich geschafft hat ist das Bump mapping. Beim Bump mapping oder auch *Normal mapping* werden Informationen

über Rillen, Einkerbungen, Rissen oder Kratzern auf Oberflächen in einer Textur kodiert und in die Lichtberechnung miteinbezogen indem die geometrische Normale der Oberfläche über die Informationen der Textur verändert wird. Die veränderte Normale wird dann für die Lichtberechnung verwendet.

## Verwischte, unscharfe Reflexionen

Unscharfe Reflexionen, wie sie auf rauen, aber dennoch reflektierenden Oberflächen auftreten, können realisiert werden indem die gesamte Reflexionsfunktion aus der Shading Funktion berücksichtigt wird und mehrere Strahlen entsprechend der Reflexionsfunktion in die Szene zurückverfolgt werden. Dieses Verfahren ersetzt die Berechnung der „Specular Highlights“.

## Depth of Field

Diesen Effekt, auf Deutsch Schärfentiefe genannt, können wir jeden Tag beobachten, wenn wir ein Objekt betrachten. Man sagt, das betrachtete Objekt liegt in der Fokusebene. Objekte, welche näher am Auge oder weiter entfernt liegen wirken dabei unscharf. Dieser Effekt kann beim Ray tracing simuliert werden indem man ein komplexeres Kameramodell verwendet. Anstatt eines Lochkameramodells, welches der Standard ist, wird eine Kamera mit einer Linse simuliert und für jedes Pixel mehrere Strahlen berechnet, welche sich alle in der Fokusebene schneiden, jedoch von unterschiedlichen Punkten der Linse ausgehen.

# 5 Aliasing

Ein Problem der digitalen Signalverarbeitung und somit auch des Bereichs der Computergrafik und des besprochenen Ray tracing Algorithmus ist das so genannte Aliasing. Bei computergenerierten Bildern hat es den bekannten Treppeneffekt zur Folge, welcher besonders bei nahezu horizontalen oder vertikalen Kanten auffällt. Das Problem in der Signalverarbeitung ist, dass ein analoges Signal durch eine Reihe von Zahlen angenähert wird. Den Prozess, der das analoge Signal in diskreten Zeitabschnitten abtastet und dabei digitalisiert, nenn man *Sampling*. In der Computergrafik werden geometrische Gebilde wie zum Beispiel Dreiecke durch eine Menge von gefärbten Pixeln angenähert. Auch dies ist ein Samplingprozess.

## 5.1 Räumliches Aliasing

Beim besprochenen Ray tracing Algorithmus tritt das Phänomen des *räumlichen Aliasings* auf. Es wird für jedes Pixel genau ein *Sample* (Stichprobe) berechnet, indem ein einzelner *unendlich dünner* Strahl durch das Pixel in die Szene geschossen und ein Farbwert berechnet wird. Der berechnete Farbwert wird anschließend repräsentativ für das gesamte Pixel verwendet. Aufgrund dieses Vorgehens führen wir einen Fehler ein, welcher sich als Treppeneffekt im Endergebnis manifestiert. Denselben Effekt kann man auch bei Computerspielen beobachten. Verwendet man eine geringere Bildschirmauflösung wird dieselbe Bildinformation durch wesentlich weniger Pixel beschrieben, was zur Folge hat, dass das Bild „pixeliger“ erscheint.

## 5.2 Zeitliches Aliasing

Zeitliches Aliasing tritt bei der Computeranimation und im Film auf, kann aber auch in der Natur beobachtet werden. Fährt ein Auto hinter einem Zaun vorbei und schaut man durch die freien Stellen beim Zaum auf die Reifen des Wagens, scheint es manchmal als würden sie sich sehr langsam oder sogar rückwärts bewegen. In einem solchen Fall tritt das zeitliche Aliasing auf, da es dem Auge nur möglich ist die Stellung des Reifen in diskreten Zeitabständen (wenn er direkt hinter einer Lücke steht) zu sehen. Dieselben Effekte treten auch bei der Computeranimation und im Film auf, da hier kontinuierliche Bewegungen durch eine vorgegebene Anzahl von Bildern pro Sekunde repräsentiert werden.

# 6 Anti-Aliasing

Um die soeben beschriebenen Aliasing Artefakte in computergenerierten Bildern zu reduzieren und somit die Bildqualität deutlich zu erhöhen werden in der Computergrafik verschiedene Techniken verwendet. Diese Techniken können auch beim besprochenen Ray tracing Algorithmus angewendet werden. In den folgenden Abschnitten werden diese Techniken besprochen.

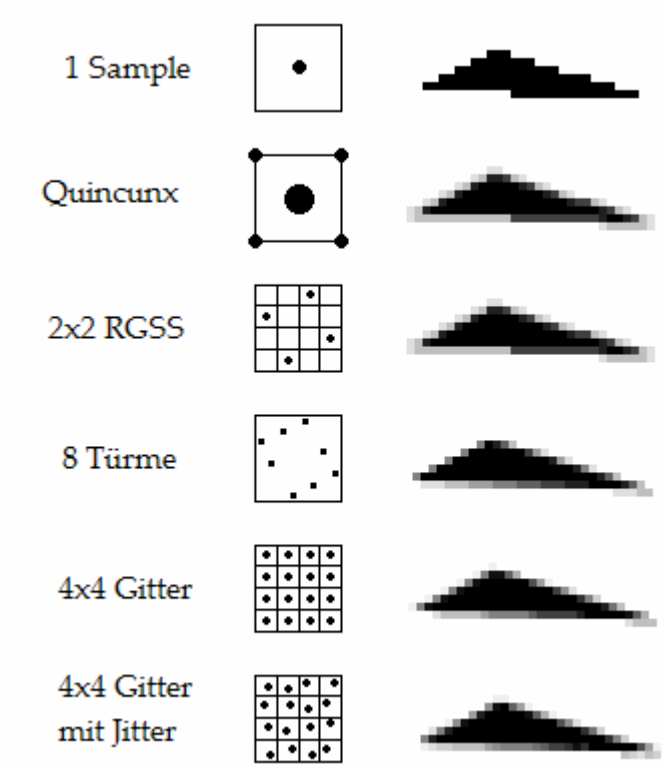
## 6.1 Supersampling

Ein einfacher Weg die durch das räumliche Aliasing hervorgerufenen Treppeneffekte zu verringern ist, anstatt eines einzelnen Strahles, viele Strahlen pro Pixel zu berechnen, einen Mittelwert der berechneten Farben zu bestimmen und diesen als Pixelfarbe zu verwenden. Diese Technik wird als *Supersampling* bezeichnet und kann die Bildqualität erheblich steigern. Das Supersampling ist allerdings recht aufwändig da der Zeitaufwand zum Berechnen eines Bildes mit der Anzahl der verwendeten Samples linear ansteigt.

Es gibt mehrere verschiedene Supersampling Verfahren, die sich in der Anordnung der Samples innerhalb der Pixelfläche unterscheiden. Tatsächlich hat die Verteilung der Samples auf der Pixelfläche eine große Auswirkung auf die erzielbare Bildqualität. Einfache Varianten wie ein 4x4 Raster oder ein 4x4 Schachbrettmuster liefern gute Ergebnisse bei 16 bzw. 8 Samples pro Pixel. In Bildern sind nahezu horizontale oder vertikale Kanten besonders stark vom Treppeneffekt betroffen. Solche Kanten können mit dem so genannten *Rotated Grid Supersampling* (RGSS) mit vier Samples oder dem „8 rooks“ (8 Türme) Muster mit acht Samples besonders gut geglättet werden und liefern in der Regel bei niedrigerem Aufwand bessere Ergebnisse als normale Raster.

Ein anderes Interessantes Supersampling Schema welches von NVIDIA Grafikkarten verwendet ist das *Quincunx* Verfahren oder auch *High Resolution Antialiasing (HRAA)* genannt. Dieses Verfahren verwendet vier Samples an den Ecken des Pixels und eines im Pixelzentrum. Die Samples an den Ecken des Pixels tragen jeweils zu einem achtel und das Sample im Zentrum zur Hälfte am resultierenden Pixelwert bei. Dieses Verfahren ist deshalb besonders interessant, da die Samples in den Ecken von mehreren Pixeln geteilt werden können und somit nicht extra berechnet werden müssen. Dies führt dazu, dass im Durchschnitt lediglich zwei Samples pro Pixel verwendet werden aber die Bildqualität im Vergleich zum normalen Supersampling mit zwei Samples erheblich besser ist.

Einige in der Computergrafik verwendete Supersampling Varianten werden in folgen den Bild dargestellt:




## 6.2 Adaptives Supersampling

Die bisher besprochenen Schemas zum Supersampling verwendeten eine vorgegebene Anzahl von Samples für jedes Pixel. In den meisten Fällen ist es jedoch so, dass nur bestimmte Stellen im Bild besonders viele Samples benötigen. Solche Stellen sind unter anderem Ränder von Objekten und Schatten. Diesen Umstand nutzt das adaptive Supersampling aus.

Das adaptive Supersampling startet wie das Quincunx Verfahren mit fünf Samples, davor vier in den Ecken und eines im Pixelzentrum. Im nächsten Schritt wird der Unterschied zwischen jedem Sample in einer Ecke und dem Sample in der Mitte betrachtet. Falls dieser einen gewissen vorher festgelegten Schwellenwert überschreitet wird in der Region zwischen Zentrum und entsprechender Ecke der Algorithmus rekursiv aufgerufen um dort mehr Samples zu berechnen.

Diese Technik arbeitet in den meisten Fällen recht gut und effizient, hat aber auch ihre Probleme. Diese Methode startet mit einer fixen Anzahl von Samples und kann so kleine Details übersehen. Dass diese Details keine besonderen Auswirkungen auf das Endergebnis haben, kann nicht einfach behauptet werden.

## 6.3 Stochastisches Supersampling

Alle bisher besprochenen Antialiasing Schemas verwenden eine bestimmte Anzahl von Samples, welche in einem regulären Gitter innerhalb der Pixelfläche angeordnet sind. Durch diese reguläre Anordnung können Aliasing Artefakte lediglich vermindert, aber nie beseitigt werden.

Beim stochastischen Supersampling werden die Positionen der einzelnen Samples zufällig und für jedes Pixel unterschiedlich gewählt. Dies hat zur Folge, dass das aufgrund des regulären Samplings hervorgerufene Aliasing verschwindet. Aliasing Effekte werden hierbei durch leichtes, weniger störend wirkendes, Rauschen ersetzt.

Ohne Aliasing Effekte wieder einzuführen kann das Rauschen vermindert werden, indem die Positionen der einzelnen Samples nicht vollständig zufällig gewählt werden. Beim so genannten *Jittered Grid Supersampling* wird das Pixel zwar durch ein reguläres Gitter in gleich große Zellen aufgeteilt, jedes Sample wird aber nur zufällig innerhalb einer solchen Zelle positioniert und nicht zufällig innerhalb des gesamten Pixels. Das Rauschen wird reduziert da der maximal mögliche Abstand einer Gruppe von Samples reduziert wird.

## 6.4 Statistisches Supersampling

Bei stochastischen Supersampling ist die Anzahl der Samples, welche für ein Pixel zu berechnen sind fix vorgegeben. Mit dem statistischen Supersampling ist es möglich, die gerade richtige Anzahl an Samples, welche zur Berechnung des Farbwertes eines Pixels benötigt werden, zu bestimmen. Nachdem man einige zufällig verteilte Samples berechnet hat vergleicht man diese und bestimmt ob der Farbwert, welcher sich aus ihnen ergeben würde, eine „ausreichend gute“ Annäherung darstellt.

Um zu bestimmen ob eine Annäherung ausreichend gut ist werden verschiedene statistische Analysen auf die Werte der bisher berechneten Samples angewendet. Solange festgestellt wird, dass die Annäherung nicht gut genug ist, werden weitere Samples zufällig auf der Pixelfläche verteilt und berechnet. Ist der Farbwert gut genug kann mit der Berechnung des nächsten Pixels fortgefahren werden.

# 7 Homogene Notation

## 7.1 Punkte, Vektoren, Strahlen und Ebenen

Bevor wir fortfahren können müssen einige wichtige Aspekte der Computergrafik angesprochen werden, welche uns im folgenden Verlauf nützlich sein werden.

In der Computergrafik wird zur Repräsentation von Punkten, Vektoren und Ebenen die so genannte *homogene Notation* des projektiven Raumes verwendet. Im Unterschied zum euklidischen dreidimensionalen Raum werden hier Punkte durch Vektoren mit vier anstatt drei Komponenten beschrieben.

$$\mathbf{p} = \begin{pmatrix} p_x \\ p_y \\ p_z \\ p_w \end{pmatrix}$$

Der entsprechende Punkt im euklidischen dreidimensionalen Raum hat folgende Koordinaten.

$$\mathbf{p}_e = \begin{pmatrix} \frac{p_x}{p_w} & \frac{p_y}{p_w} & \frac{p_z}{p_w} \end{pmatrix}^T$$

Für Punkte ist die vierte Komponente $p_w = 1$ wohingegen bei Vektoren, welche eine Richtung beschreiben, selbst aber keine Position haben $p_w = 0$ ist. Wenn $p_w \neq 0$ und $p_w \neq 1$ kann der tatsächliche Punkt durch *Homogenisierung* bestimmt werden indem alle Komponenten durch $p_w$ dividiert werden.

Eine Ebene $\pi : \mathbf{n} \cdot \mathbf{x} + d = 0$ kann ebenfalls durch einen vierwertigen Vektor $\mathbf{e} = (a, b, c, d)^T$ beschrieben werden wobei $a = n_x$, $b = n_y$ und $c = n_z$ sind. Eine Ebene ist in normalisierter Form, wenn $\|\mathbf{n}\| = 1$ ist.

Ein Strahl hat einen Ursprung und eine Richtung. Alle Punkte eines Strahls können durch zwei Vektoren, welche den Ursprung und die Richtung angeben und durch einen Parameter $t$, beschrieben werden. In Vektorform kann ein Strahl wie folgt geschrieben werden:

$$\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$$

Wobei der Vektor **o** den Ursprung und **d** den Richtungsvektor beschreibt. Der Vektor **d** ist normalerweise normalisiert. Der skalare Parameter $t$ wird verwendet um

auf dem Strahl liegende Punkte zu berechnen wobei man für Punkte mit $t \leq 0$ sagt, dass sie hinter dem Strahl liegen und nicht Teil desselben sind.

## 7.2 Transformationen

Lineare Transformationen sind ein wichtiges Werkzeug der Computergrafik. Transformationen von Punkten, Vektoren und Ebenen wie etwa Verschiebung, Rotation und Skalierung können dabei durch 4x4 Matrizen ausgedrückt werden.

Die Oberflächen dreidimensionaler Objekte werden oft durch Polygone beschrieben deren Koordinaten sich auf ein lokales Objektkoordinatensystem beziehen. Um diese Objekte im Raum zu positionieren und zu orientieren müssen so genannte Objekttransformationen auf die einzelnen Eckpunkte der Polygone angewendet werden. Auch die Oberflächennormalen von Objekten müssen korrekt transformiert werden.

Eine Verschiebung (Translation) um einen Vektor $\mathbf{t} = (t_x, t_y, t_z)$ kann beispielsweise durch die folgende Transformationsmatrix ausgedrückt werden:

$$\mathbf{T}(\mathbf{t}) = \mathbf{T}(t_x, t_y, t_z) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Ein Punkt oder Vektor kann anhand dieser Matrix transformiert werden, indem er mit der Matrix von rechts multipliziert wird:

$$\mathbf{v}' = \mathbf{T}\mathbf{v} = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} v_x \\ v_y \\ v_z \\ v_w \end{pmatrix} = \begin{pmatrix} v_x + t_x v_w \\ v_y + t_y v_w \\ v_z + t_z v_w \\ v_w \end{pmatrix}$$

Diese Transformationsmatrix hat auf Vektoren, welche nur eine Richtung angeben, aber selbst keine Position haben keinen Einfluss, da für diese $v_w = 0$ ist.

Um Ebenen und Oberflächennormalen korrekt zu transformieren kann nicht immer dieselbe Matrix wie für Punkte und Vektoren verwendet werden. Ebenen und Normalen müssen mit der transponierten der inversen jener Matrix transponiert werde, welche für Punkte und Vektoren verwendet wird. In unserem Fall bedeutet dies, dass wir eine Oberflächennormale **n** folgendermaßen transformieren müssen:

$$\mathbf{n}' = (\mathbf{T}^{-1})^T \mathbf{n}$$

Für einen Raytracer sind Transformationen beim Berechnen von Schnittpunkten mit einem Strahl interessant. Es ist hierdurch möglich lediglich einen Spezialfall (z.B. Kugel mit Radius 1 im Zentrum) zu betrachten und anhand von Transformationen jeden anderen Fall wie beispielsweise Ellipsoide zu berechnen. Hierbei wird die Transformation nicht direkt auf das Objekt, sondern auf den Strahl angewendet um diesen in das Koordinatensystem des Objektes zu transformieren. Erst dann wird der Schnittpunkt berechnet.

Formeln zur Berechnung anderer Transformationen können in der entsprechenden Literatur gefunden werden [2, 8].

# 8 Schnittpunkte mit Objekten

Um überhaupt die Shading Funktion anwenden und Farbwerte für Pixel berechnen zu können ist es notwendig den Schnittpunkt eines Strahls mit einem Objekt zu berechnen. Beim Ray tracing stellen meist diese Schnittpunktberechnungen den Flaschenhals dar, weshalb es wichtig ist, diese effizient zu implementieren.

## 8.1 Schnittpunkt von Strahl mit Ebene

Um den gesuchten Schnittpunkt berechnen zu können betrachten wir zunächst noch einmal die Gleichungen für den Strahl:

$$\mathbf{r}(t) = \mathbf{o} + t\mathbf{d} \quad \text{mit } t > 0$$

Und die Gleichung der Ebene wobei **n** normalisiert ist:

$$\pi : \mathbf{n} \cdot \mathbf{x} + d = 0$$

Den Abstand eines Punktes **x** zur Ebene kann berechnet werden, indem dieser in die Gleichung der Ebene eingesetzt wird. Punkte, welche auf dem Strahl liegen können erzeugt werden, indem der Parameter t der Strahlgleichung variiert wird.

Gesucht ist jener Wert für $t$, der einen Punkt auf dem Strahl erzeugt, der gleichzeitig Punkt der Ebene ist. Um diesen Wert zu finden wird zunächst die Gleichung des Strahls in die Gleichung der Ebene eingesetzt:

$$\mathbf{n} \cdot (\mathbf{o} + t\mathbf{d}) + d = 0$$

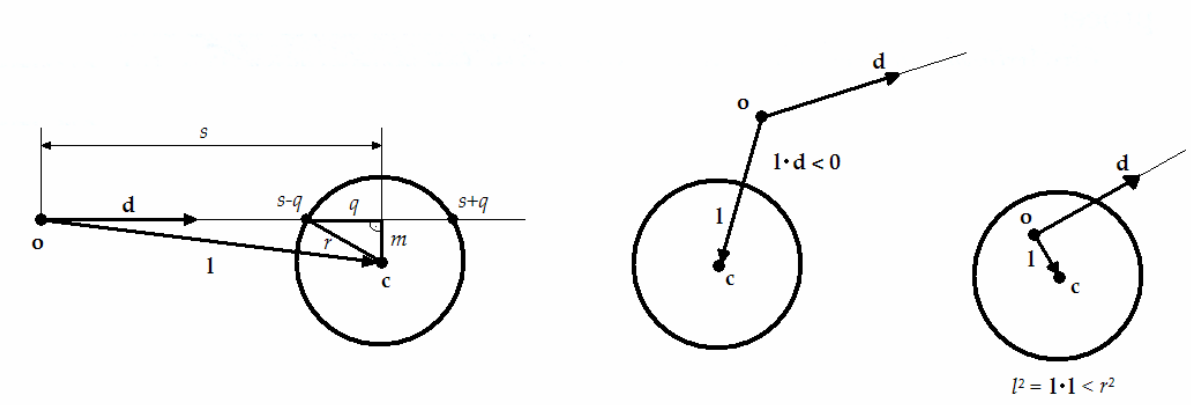
Die einzige unbekannte Variable in dieser Gleichung ist der Parameter $t$, nach welchem die Gleichung umgeformt werden kann. Nach Umformung erhält man folgende Gleichung für $t$:

$$t = \frac{-(\mathbf{n} \cdot \mathbf{o} + d)}{\mathbf{n} \cdot \mathbf{d}}$$

Verläuft der Strahl parallel zur Ebene ist $\mathbf{n} \cdot \mathbf{d} = 0$ und es gibt keinen Schnittpunkt. Ist $t \leq 0$ liegt der Schnittpunkt mit der Ebene hinter dem Strahl und wird somit nicht als tatsächlicher Schnittpunkt gezählt. Die Lösung für $t$ gibt hierbei die Entfernung des Schnittpunktes vom Ursprungs **o** des Strahls in Längeneinheiten relativ zur Länge des Richtungsvektors **d** an. Um die Koordinaten des Schnittpunkts zu berechnen kann die Lösung für den Parameter $t$ in die Gleichung des Strahls eingesetzt werden.

## 8.2 Schnittpunkt von Strahl mit Kugel

Um die Schnittpunkte eines Strahls mit einer Kugel zu berechnen, kann analog zur Schnittpunktberechnung mit der Ebene verfahren werden, indem die Gleichung des Strahls in jene der Kugel eingesetzt und nach dem Parameter $t$ aufgelöst wird. Effizienter ist jedoch die geometrische Lösung. Hier ist es möglich frühzeitig zu entscheiden ob überhaupt ein Schnittpunkt existieren kann was es erlaubt unnötige Berechnungen einzusparen und den Algorithmus zu beschleunigen. Im Folgenden wird die geometrische Lösung von Eric Haines [2, 4] besprochen.




Der Strahl wird wieder durch $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$ beschrieben wobei $\mathbf{d}$ normalisiert ist. Eine Kugel kann durch ihr Zentrum $\mathbf{c}$ und den Radius $r$ beschrieben werden. Als erstes wird der Vektor $\mathbf{l} = \mathbf{c} - \mathbf{o}$ vom Ursprung des Strahls zum Zentrum der Kugel berechnet. Weiters wird noch $s = \mathbf{l} \cdot \mathbf{d}$ (die Länge der Projektion von $\mathbf{l}$ auf den Richtungsvektor $\mathbf{d}$ des Strahls) sowie $l^2 = \mathbf{l} \cdot \mathbf{l}$ (die quadratische Länge von $\mathbf{l}$) berechnet. Nach diesen Berechnungen ist es möglich zu entscheiden ob der Strahl die Kugel mit Sicherheit verfehlt oder weitere Berechnungen notwendig sind. Ist $s < 0$ liegt das Zentrum der Kugel hinter dem Ursprung des Strahls. Wenn nun der Ursprung des Strahls auch noch außerhalb der Kugel liegt ($l^2 > r^2$) gibt es keinen Schnittpunkt und es bedarf keiner weiteren Berechnungen. Im anderen Fall wird der quadratische Abstand des Kugelzentrums von seiner Projektion auf dem Strahl berechnet: $m^2 = l^2 - s^2$ ($m$ ist der kürzeste Abstand des Kugelzentrums vom Strahl). Ist $m^2 > r^2$ gibt es keinen Schnittpunkt, da der Strahl an der Kugel vorbeizieht. Im anderen Fall gibt es zwei Schnittpunkte: $q = \sqrt{r^2 - m^2}$, $t = s \pm q$. Ist der Ursprung des Strahls außerhalb der Kugel wird $t = s - q$ verwendet, ansonsten $t = s + q$.

Alle durchzuführenden Schritte zusammengefasst:

1. $\mathbf{l} = \mathbf{c} - \mathbf{o}$
2. $s = \mathbf{l} \cdot \mathbf{d}$
3. $l^2 = \mathbf{l} \cdot \mathbf{l}$
4. Wenn $s < 0$ und $l^2 > r^2$ gibt es keinen Schnittpunkt
5. $m^2 = l^2 - s^2$
6. Wenn $m^2 > r^2$ gibt es keinen Schnittpunkt

7. $q = \sqrt{r^2 - m^2}$
8. Wenn $l^2 > r^2$ dann ist $t = s - q$
9. sonst ist $t = s + q$

## 8.3 Schnittpunkt von Strahl mit Quader

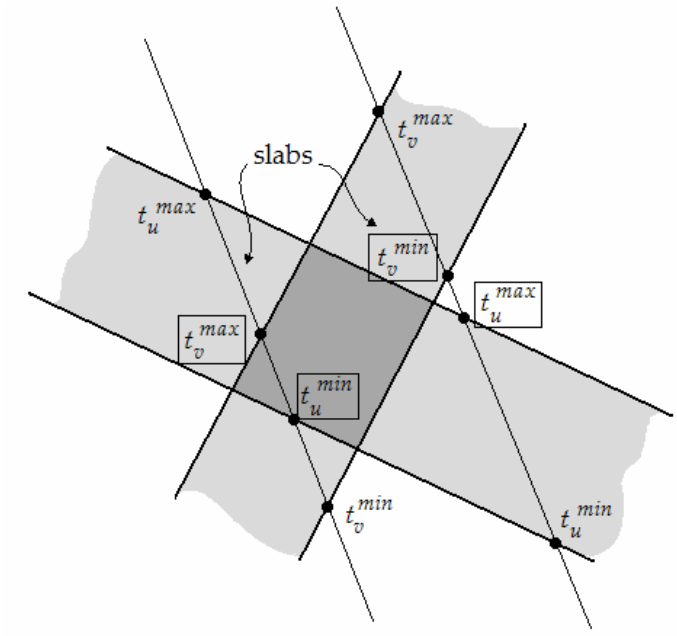
Eine einfache Methode zur Schnittpunktberechnung zwischen Strahl und Quader ist die von Eric Haines [2, 4] präsentierte „Slabs“ Methode.

Ein Slab ist dabei der Raum zwischen zwei parallel verlaufenden Ebenen. Der von einem Quader eingenommene Raum kann über die Schnittmenge von drei, jeweils senkrecht aufeinander stehenden, Slabs beschrieben werden. Um den Schnittpunkt des Strahls mit dem Quader zu berechen werden zunächst alle *t*-Werte der, zu den Slabs gehörenden, Ebenen auf die bereits beschriebene Weise berechnet. Für jeden Slab gibt es einen maximalen Wert $t_i^{max}$ und einen minimalen Wert $t_i^{min}$ wobei $\forall i \in \{u,v,w\}$ gilt. Um bestimmen zu können ob der Strahl den Quader überhaupt trifft müssen das Maximum der $t_i^{min}$ Werte und das Minimum der $t_i^{max}$ Werte bestimmt werden:

$$t^{min} = \max(t_u^{min}, t_v^{min}, t_w^{min})$$
$$t^{max} = \min(t_u^{max}, t_v^{max}, t_w^{max})$$

Ist $t^{min} > t^{max}$ gibt es keinen Schnittpunkt mit dem Strahl. Im anderen Fall ist der *t*-Wert des Schnittpunkts der kleinere jener Werte die größer als Null sind. Sind beide *t*-Werte kleiner Null so ist der Schnittpunkt hinter dem Strahl und wird nicht als wirklicher Schnittpunkt betrachtet.

Die folgende Grafik veranschaulicht den Vorgang in zwei Dimensionen:



## 8.4 Schnittpunkt von Strahl mit Zylinder

Die implizite Gleichung $x^2 + y^2 = 1$ beschreibt einen unendlichen Einheitszylinder. Um den Schnittpunkt desselben mit dem Strahl zu berechnen muss die Gleichung des Strahls für $x$ bzw. $y$ in die Formel eingesetzt werden:

$$(o_x + td_x)^2 + (o_y + td_y)^2 = 1$$

Nach dem Ausmultiplizieren der Quadrate erhält man eine einfache quadratische Gleichung:

$$\begin{aligned}
&{o_x}^2 + 2td_x o_x + t^2 {d_x}^2 + {o_y}^2 + 2td_y o_y + t^2 {d_y}^2 - 1 = 0 \\
&({d_x}^2 + {d_y}^2)t^2 + 2(d_x o_x + d_y o_y)t + {o_x}^2 + {o_y}^2 - 1 = 0 \\
&at^2 + bt + c = 0 \\
&t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}
\end{aligned}$$

Zur Darstellung in computergenerierten Bildern ist ein unendlicher Zylinder meist ungeeignet. Ein offener, abgeschnittener Zylinder kann realisiert werden, indem man die tatsächlichen Schnittpunkte durch einsetzten der $t$-Werte in die Gleichung des Strahls berechnet und nur jene Schnittpunkte akzeptiert für welche der $z$-Wert im gewünschten Bereich liegt. Soll der Zylinder geschlossen sein müssen zusätzliche Schnittpunkte mit einer oberen und unteren Ebene berechnet werden. In den meisten Fällen werden Zylinder mit unterschiedlichen Radien und Orientierungen im Raum benötigt. Dies kann realisiert werden, indem der Strahl durch eine 4x4 Matrix in das Objektkoordinatensystem des Zylinders transformiert wird und erst dann der Schnittpunkt berechnet wird. Um die tatsächlichen Schnittpunktkoordinaten zu ermitteln wird der errechnete $t$-Wert in die nicht transformierte Strahlgleichung eingesetzt.

## 8.5 Schnittpunkt von Strahl mit Dreieck

Dreiecke sind die in der Computergrafik am häufigsten verwendeten Objekte, da es mit ihnen möglich ist die Oberflächen beliebiger andere Objekte anzunähern. Da in der Regel eine sehr große Anzahl an Dreiecken benötigt wird muss der Algorithmus zum bestimmen des Schnittpunktes schnell sein. Haines und Möller [2] beschreiben einen solchen Algorithmus.

Ein Dreieck kann durch drei Eckpunkte $\mathbf{v}_0$, $\mathbf{v}_1$ und $\mathbf{v}_2$, die so genannten *Vertices*, beschrieben werden. Die Punkte des Dreiecks können anhand von *baryzentrischen Koordinaten* anhand der folgenden expliziten Formel ausgedrückt werden:

$$\mathbf{t}(u,v) = (1 - u - v)\mathbf{v}_0 + u\mathbf{v}_1 + v\mathbf{v}_2$$

Die Parameter ($u$, $v$) sind dabei die baryzentrischen Koordinaten für welche $u \geq 0$, $v \geq 0$ und $u + v \leq 1$ gelten muss. Der Schnittpunkt mit einem Strahl kann berechnet werden indem dessen Gleichung für $\mathbf{t}(u, v)$ eingesetzt wird:

$$\mathbf{o} + t\mathbf{d} = (1 + u + v)\mathbf{v}_0 + u\mathbf{v}_1 + v\mathbf{v}_2$$

Die Terme der obigen Gleichung können umgeordnet werden und formen somit ein lineares Gleichungssystem in drei Unbekannten (geschrieben in Vektornotation):

$$\begin{pmatrix} -\mathbf{d} & \mathbf{v}_1 - \mathbf{v}_0 & \mathbf{v}_2 - \mathbf{v}_0 \end{pmatrix} \begin{pmatrix} t \\ u \\ v \end{pmatrix} = \mathbf{o} - \mathbf{v_0}$$

Eine Lösung für $t$, $u$ und $v$ kann gefunden werden, indem man die Matrix $(-\mathbf{d} \quad \mathbf{v}_1 - \mathbf{v}_0 \quad \mathbf{v}_2 - \mathbf{v}_0)$ explizit invertiert, was aber nicht besonders effizient ist.

Die Lösung des Gleichungssystems kann unter Verwendung der *Cramerschen Regel* effizienter berechnet werden, da hier die inverse der Matrix nicht explizit bestimmt werden muss.

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{\det(-\mathbf{d}, \mathbf{e}_1, \mathbf{e}_2)} \begin{pmatrix} \det(\mathbf{s}, \mathbf{e}_1, \mathbf{e}_2) \\ \det(-\mathbf{d}, \mathbf{s}, \mathbf{e}_2) \\ \det(-\mathbf{d}, \mathbf{e}_1, \mathbf{s}) \end{pmatrix}$$

Hier entspricht $\mathbf{e}_1 = \mathbf{v}_1 - \mathbf{v}_0$, $\mathbf{e}_2 = \mathbf{v}_2 - \mathbf{v}_0$ und $\mathbf{s} = \mathbf{o} - \mathbf{v}_0$.

Noch effizienter geht es indem man die die Determinanten der 3x3 Matrizen durch eine Kombination von Kreuz- und Skalarprodukt schreibt ($\det(\mathbf{a},\mathbf{b},\mathbf{c}) = (\mathbf{a} \times \mathbf{b}) \cdot \mathbf{c} = -(\mathbf{a} \times \mathbf{c}) \cdot \mathbf{b}$) und effizient umformt um so wenige verschiedene Faktoren wie möglich zu haben:

$$\mathbf{p} = \mathbf{d} \times \mathbf{e}_2$$
$$\mathbf{q} = \mathbf{s} \times \mathbf{e}_1$$

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{\mathbf{p} \cdot \mathbf{e}_1} \begin{pmatrix} \mathbf{q} \cdot \mathbf{e}_2 \\ \mathbf{p} \cdot \mathbf{s} \\ \mathbf{q} \cdot \mathbf{d} \end{pmatrix}$$

# 9 Präzisionsprobleme bei Gleitkommazahlen

Programmiert man einen Raytracer wird man spätestens bei der Implementierung von Schatten auf die Probleme der Gleitkommazahlen bei Computern stoßen. In den Bildern manifestiert sich das Problem durch Oberflächen mit Eigenschatten an Stellen wo eigentlich kein Schatten sein sollte.

Um zu bestimmen ob ein bestimmter Punkt der Oberfläche eines Objektes im Schatten liegt wird von genau diesem Punkt aus ein Strahl in Richtung der Lichtquelle gerichtet und bestimmt ob es ein Objekt gibt, welches diesen Strahl schneidet. Gibt es ein Objekt, so liegt der Punkt im Schatten.

Bei der Schnittpunktüberprüfung wird beim vorgestellten Algorithmus jedes Objekte getestet und damit auch jenes Objekte auf welchem sich der Oberflächenpunkt befindet. Theoretisch sollte dies kein Problem darstellen, da für jenen Punkt $t = 0$ ist und er somit nicht Teil des Strahls ist. Da es dem Computer aber nicht möglich ist Gleitkommazahlen in unendlicher Genauigkeit darzustellen schleichen sich bei jeder durchgeführten Rechenoperation kleine Fehler ein, welche dazu führen, dass für $t$ ein Wert ungleich Null berechnet werden kann. Es wird dann fälschlicherweise angenommen, dass ein blockierendes Objekt existiert und der Punkt im Schatten liegt.

Eine Möglichkeit dieses Problem zu vermeiden ist indem man explizit angibt, ob sich der Ursprung des zu testenden Strahls auf der Oberfläche des Objektes befindet. Hier kann dieser Umstand bei der Schnittpunktberechnung berücksichtigt werden. Hier muss man allerdings bedenken, dass es nicht nur Strahlen zu Schattenberechnung, sondern auch jene für Reflexionen und auch für die Refraktion gibt, welche auch in die Kugel eindringen können.

Eine weitere einfache Möglichkeit ist den Ursprung eines neuen Strahls etwas zu versetzen, so dass es bei den Berechnungen nicht zu den numerischen Problemen kommt. Dabei werden Schatten und Reflexionsstrahlen in Richtung der Oberflächennormalen versetzt und Refraktionsstrahlen in die Gegenrichtung.

# 10 Implementierung eines Raytracers

Im Rahmen der vorliegenden Bakkalaureatsarbeit habe ich selbst einen Raytracer in der Programmiersprache C++ implementiert um einige Bilder selbst rendern zu können und die Möglichkeiten des Ray tracings zu demonstrieren. Für meine Implementierung hatte ich mir zu Beginn einige Ziele gesetzt:

- Erweiterbarkeit
- Unterstützung verschiedener Objekttypen (Kugel, Ebene, Quader, Zylinder und Dreieck) und das einfache Hinzufügen neuer Objekttypen ermöglichen
- Ausführen beliebiger linearer Transformationen auf Objekte
- Unterstützung verschiedener Kameramodelle
- Unterstützung der verschiedenen vorgestellten Samplingverfahren und das einfache Hinzufügen neuer Verfahren ermöglichen
- Entkoppelung der Shading Funktion vom Kernalgorithmus und es erlauben beliebige Shading Funktionen zu implementieren

Das Hauptziel bei der Implementierung war für mich die Erweiterbarkeit, sodass es mir möglich war verschiedene Objekte, Samplingverfahren und auch Shading Funktionen zu testen.

## 10.1 Die math3d++ Bibliothek

Eine große Hilfe bei der Implementierung des Raytracers war die von mir selbst geschriebene C++ Bibliothek *math3d++*. Sie enthält C++ Klassen für Vektoren und Matrizen sowie zusätzliche Funktionen, welche in der 3D Computergrafik oft benötigt werden und vereinfacht dadurch die Programmierung entsprechender Anwendungen erheblich. Um ein Beispiel der einfachen Handhabung zu geben wird hier die folgende Gleichung zur Schnittpunktberechnung von Strahl und Ebene als Codefragment dargestellt:

$$t = \frac{-(\mathbf{n} \cdot \mathbf{o} + d)}{\mathbf{n} \cdot \mathbf{d}}$$

```
vec3 normal, origin, direction;
float d;

float t = -(dot(normal, origin) + d) / dot(normal, direction);
```

Aus dem Beispiel ist ersichtlich, dass durch die Bibliothek die Handhabung von Vektoren ähnlich einfach wie die Handhabung normaler Variablen gemacht wird.

## 10.2 Objektorientiertes Programmieren

Das Hauptziel für meine Implementierung war die Erweiterbarkeit des Raytracers. Es sollte einfach sein neue Typen von Objekten, neue Samplingverfahren und neue Shading Funktionen hinzuzufügen ohne direkt in den Kern des Raytracers eingreifen zu müssen. Beispielsweise sollte es bei der Implementierung eines neuen Objektes nicht notwendig sein bestehenden Code zu ändern, sondern lediglich neuen Code hinzuzufügen.

Um dieser Anforderung gerecht zu werden hat sich das objektorientierte Programmieren in C++ gut geeignet. Der Raytracer kann durch die Verwendung von so genannten abstrakten Klassen, welche eine Schnittstelle zu den konkreten Objekten darstellen, vollständig unabhängig von diesen gemacht werden. Will man den Raytracer nun um ein neues geometrisches Objekt erweitern ist es lediglich notwendig eine, das Objekt repräsentierende, Klasse zu programmieren, welche die abstrakte Schnittstelle implementiert. Auf dieselbe Weise ist es möglich die Samplingverfahren und die Shading Funktion zu abstrahieren.

## 10.3 Kernklassen des Projekts

Der Raytracer ist modular aufgebaut und die Funktionalität ist in mehrere Klassen und Interfaces aufgeteilt. Jede Klasse hat dabei ihre eigene Aufgabe für welche sie alleine zuständig ist. Die wichtigsten Klassen und ihre Aufgaben werden im Folgenden kurz besprochen. Für mehr Details wird auf den Quellcode und die dazugehörige Dokumentation verwiesen.

### Basis

Die Klasse kapselt das Koordinatensystem eines Objektes und erlaubt es Vektoren, Punkte und Strahlen vom globalen Koordinatensystem in jenes des Objektes bzw. vom Objektkoordinatensystem ins globale Koordinatensystem zu transformieren.

### ICamera

Dieses Interface wird von Kameraklassen implementiert welche die Aufgabe haben für ein gegebenes Pixel auf der Bildfläche einen Strahl zu generieren, welcher dann vom Ray tracing Algorithmus verwendet wird. Durch diese Klasse ist es zum Beispiel möglich verschiedenen Projektionen wie die Normalprojektion und die Perspektive zu realisieren.

### IObject

Will man den Raytracer um ein neues Objekt erweitern, ist es lediglich notwendig eine neue Klasse zu definieren, die die Methoden dieses Interfaces implementiert,

was es dem Ray tracing Algorithmus ermöglicht einen Schnittpunkt mit dem Objekt zu berechnen.

### IPixelSampler

Um verschiedene Samplingverfahren zu realisieren ist es lediglich notwendig dieses Interface zu implementieren.

### IMaterialShader

Jedem geometrischen Objekt ist ein Shader Objekt zugewiesen, dessen Aufgabe es ist die Interaktion der Lichtstrahlen mit der Oberfläche des Objektes zu implementieren. Durch Implementierung dieses Interfaces ist es somit möglich die verschiedensten Shading Funktionen zu realisieren. Da jedem Objekt ein eigenes Shader Objekt zugewiesen wird kann die Shading Funktion für bestimmte Objekte spezialisiert werden wodurch bessere Leistung bzw. Spezialeffekte ermöglicht werden.

### IBackgroundShader

Es kann der Fall auftreten, dass ein berechneter Strahl kein Objekt trifft. Will man in einem solchen Fall nicht einfach eine schwarze Hintergrundfarbe kann eine Klasse dieses Interface implementieren und abhängig vom Strahl eine Hintergrundfarbe zuweisen.

### Scene

Diese Klasse stellt eine Szene dar, welche aus mehreren Objekten bestehen kann. Sie enthält den Kernalgorithmus des Ray tracings und kann für einen Strahl den Farbwert berechnen indem er rekursiv durch die Szene verfolgt wird.

### Renderer

Die Renderer Klasse kann mit einer Kamera, einem PixelSampler, einer Szene und einem Framebuffer konfiguriert werden und hat die Aufgabe die Komplette Szene in den Framebuffer zu rendern, sodass dieser anschließend dargestellt werden kann.

# 11 Ergebnisse

## 11.1 Beispielszenen

Um die Fähigkeiten meines Raytracers aufzuzeigen habe ich einige Beispielszenen erstellt und gerendert. Die gerenderten Bilder der einzelnen Szenen können auf der nächsten Seite betrachtet werden.

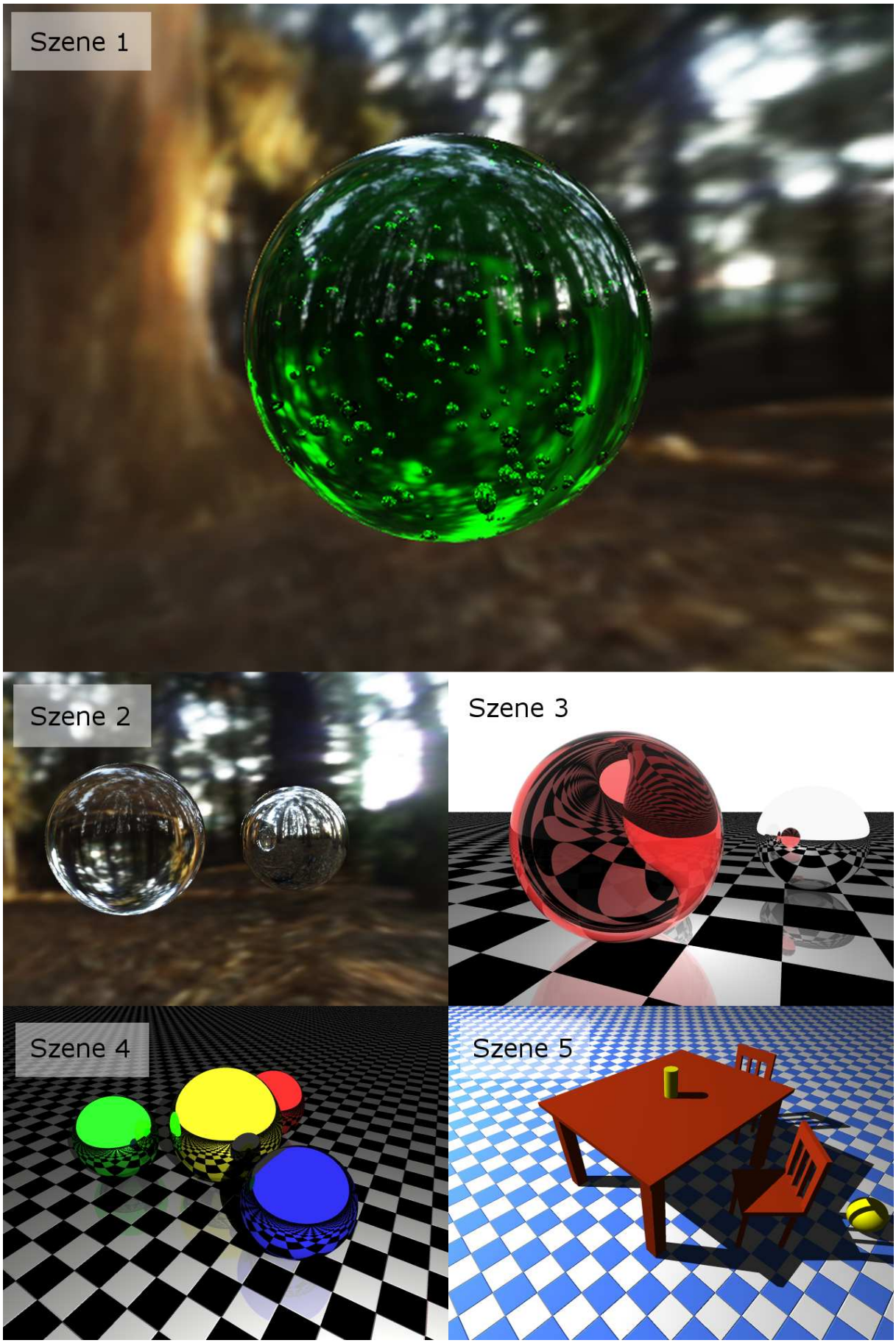
In Szene 1 wird eine grüne Glaskugel, welche Luftblasen enthält, dargestellt. Wie unschwer zu erkennen ist wird das Licht von der Kugel refraktiert und auch zum Teil reflektiert. Das Verhältnis zwischen Reflexion und Refraktion wird über den fresnelschen Term bestimmt. Das durch die Kugel wandernde Licht wird des weiteren auch noch gefiltert, was zur Folge hat, dass die Kugel grün aussieht. Für den Hintergrund der Szene wurde eine von Paul Debevec's *Light Probes* [7] verwendet, welche Informationen über die Lichtverhältnisse der Szene enthält. In dieser Szene wird auch das so genannte *High Dynamic Range (HDR) Rendering* angewendet, was realistischere Beleuchtungsberechungen ermöglicht.

In Szene 2 werden zwei Kugeln dargestellt, denen unterschiedliche MaterialShader zugewiesen wurden. Dadurch erscheint eine Kugel metallisch, glatt und stark reflektierend wohingegen die andere wie eine Glaskugel aussieht, in welcher das Licht gebrochen wird.

Szene 3 ist eine einfach gestaltete Szene, welche kein HDR Rendering verwendet. Auf einer unendlich großen Ebene mit einer prozedural generierten Schachbretttextur befinden sich drei Kugeln. Der Effekt, der in der roten Kugel zu beobachten ist, ergibt sich durch die Refraktion der Kugel und dadurch, dass sich hinter dieser eine reflektierende Kugel befindet.

Szene 4 stellt verschiedenfarbige reflektierende Kugeln dar. Hier sind mehrfache Reflexionen erkennbar.

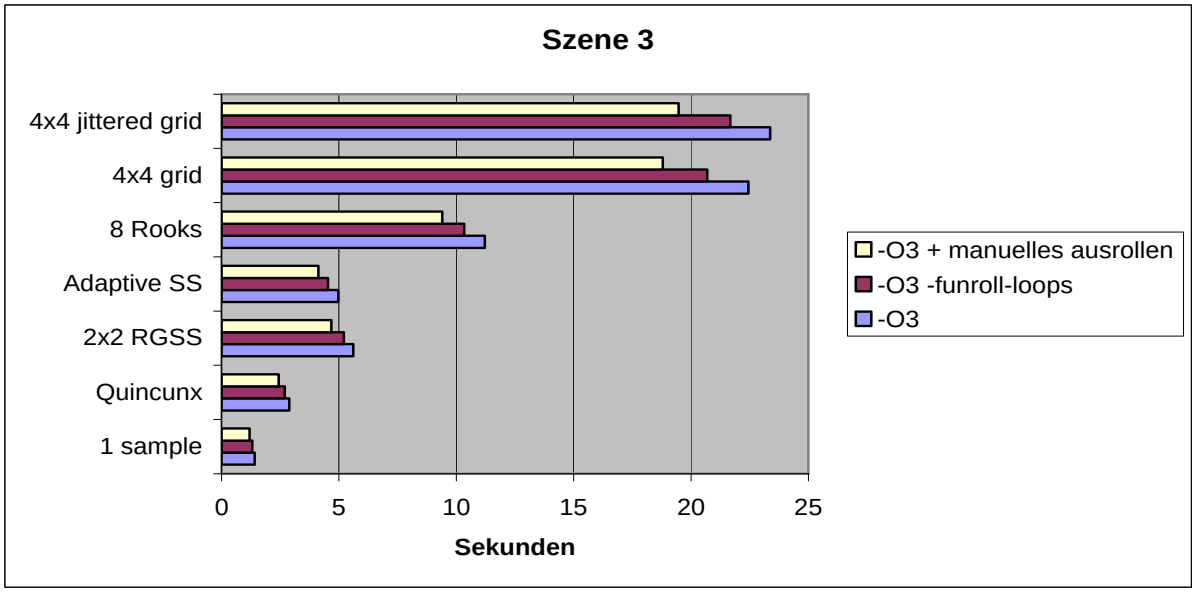
In Szene 5 werden ein Tisch und zwei Stühle dargestellt. Der Tisch und die Stühle wurden durch mehrere unterschiedlich transformierte Quader modelliert. Besonders gut zu erkennen ist der Schattenwurf des Tisches und der Stühle. Der Fliesenboden hat wieder ein Schachbrettmuster, diesmal aber auch noch mit Bumpmaps, was dem Boden mehr Tiefe verleiht ohne die geometrische Komplexität der Szene zu erhöhen.


Szene 1
Szene 2
Szene 3
Szene 4
Szene 5

## 11.2 Performancemessungen

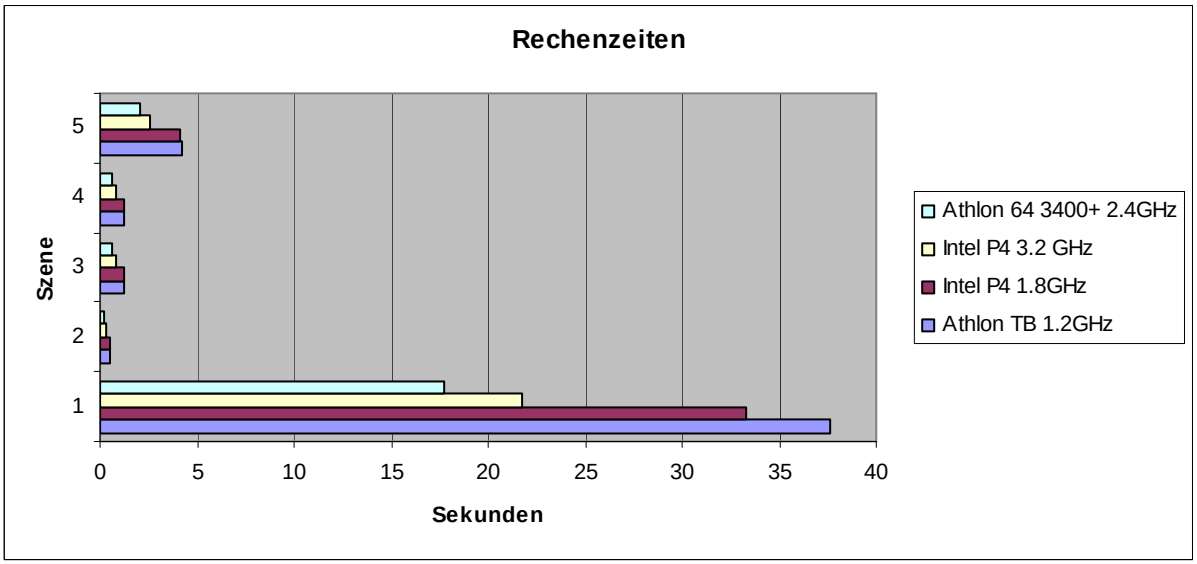
Im folgenden Abschnitt findet sich eine Zusammenfassung der durchgeführten Performancemessungen. Anzumerken ist hier, dass der Raytracer nicht optimiert wurde, d.h. es wurden keine beschleunigenden Datenstrukturen und auch keine speziellen Prozessorbefehlssätze wie beispielsweise SSE verwendet.

Das folgende Diagramm zeigt, welche Performanceunterschiede sich abhängig von den Kompilieroptionen ergeben können. Getestet wurde hierbei auf einem AMD Athlon Thunderbird mit 1.2 GHz unter Windows. Kompiliert wurde mit dem GNU g++ Compiler der Version 3.2.3 aus dem MinGW Paket.




Kompiliert wurde der Raytracer immer mit Optimierung auf Stufe 3 (ohne ist er viel zu langsam). Durch das Hinzunehmen der Compileroption *–funroll-loops* konnte der Raytracer etwa 10% schneller gemacht werden. Anfänglich wurden in der math3d++ Bibliothek for-Schleifen über die einzelnen Elemente von Vektoren und Matrizen für deren Operationen verwendet. Durch manuelles ausrollen der Schleifen in der math3d++ Bibliothek konnte eine Geschwindigkeitssteigerung von etwa 20% erzielt werden. Dies zeigt, dass manuelles ausrollen von kurzen Schleifen in performancekritischen Codeabschnitten durchaus Sinn macht. Die erzielte Leistung ist höher als jene, welche durch das Setzen der Compileroption erzielt wird. Anzumerken ist hier, dass nur die Schleifen in der math3d++ Bibliothek ausgerollt wurden. Jene im Raytracer selbst könnten durch *–funroll-loops* zusätzlich ausgerollt werden, was noch etwas mehr Geschwindigkeit bringen sollte. Zudem könnte man auch noch weitere Compileroptionen testen.

Das folgende Diagramm stellt die benötigten Rechenzeiten der einzelnen Szenen bei einem Sample pro Pixel und einer Auflösung von 640 mal 480 Bildpunkten dar.




Ein AMD Athlon mit 1.2 GHz ist in dem meisten Fällen etwa gleich schnell wie ein Intel Pentium 4 Prozessor mit 1.8 GHz. Das kommt daher, dass AMD Rechner im Allgemeinen bei Floating Point Berechnungen schneller sind. In Szene 1 überholt der Intel Rechner den Athlon jedoch deutlich. Die Ursache dafür liegt in der größeren Anzahl von Objekten in der Szene, was zur Folge hat, dass ein gewisser Codeabschnitt, welcher lediglich für die Verwaltung zuständig ist häufiger aufgerufen wird. Dieser besagte Codeabschnitt enthält keine Floating Point Berechnungen.

# Referenzen


[1] Andrew S. Glassner. „An Introduction To Ray Tracing“. Academic Press Inc., 1989

[2] Tomas Akenine-Möller und Eric Haines. „Real-Time Rendering” 2. Ausgabe. A.K. Peters, Ltd. 2002

[3] Paul S. Heckbert. „Writing a ray tracer“. In An Introduction to Ray Tracing von Andrew S. Glassner. Academic Press Inc., 1989

[4] Eric Haines. “Essential Ray Tracing Algorithms”. In An Introduction to Ray tracing von Andrew S. Glassner. Academic Press Inc., 1989

[5] Xavier Bec. „Faster Refraction Formula, and Transmission Color Filtering” in Ray Tracin News, Volume 10 Number 1, 1997. http://www.acm.org/tog/resources/RTNews/html/rtnv10n1.html

[6] The OpenRT Real-Time Ray-Tracing Project. http://www.openrt.de/

[7] Paul Debevec. Light Probe Image Gallery. http://www.debevec.org/Probes/

[8] The OpenGL® Graphics System A Specification. Version 2.0. September 2004 http://www.opengl.org/documentation/specs/version2.0/glspec20.pdf

[9] 3D Object Intersections. http://www.realtimerendering.com/int/

[10] Recursive Ray Tracing. http://www.cs.fit.edu/wds/classes/adv-graphics/raytrace/raytrace.html

[11] The Recursice Ray Tracing Algorithm http://www.geocities.com/jamisbuck/raytracing.html

[12] Some Mathematics for Advanced Graphics http://www.cl.cam.ac.uk/Teaching/1999/AGraphHCI/SMAG/

[13] Introduction to Computer Graphics http://medialab.di.unipi.it/web/IUM/Waterloo/html.html

[14] Exposure Control http://freespace.virgin.net/hugo.elias/graphics/x_posure.htm

[15] Ray Tracing: Graphics for the Masses http://www.cs.unc.edu/~rademach/xroads-RT/RTarticle.html