Johannes-Heidenhain-Gymnasium Traunreut Kollegiatenjahrgang 2009/2010

FACHARBEIT

aus dem Fach

Mathematik

Computergrafik mittels Raytracing

Verfasser: **Andreas Wegscheider**

Leistungskurs: **Mathematik**

Kursleiter: **Herr Weich**

Abgabetermin: .....

Erzielte Note: ..... in Worten: .....

Erzielte Punkte: ..... in Worten: .....

Abgabe der korrigierten Arbeit beim Kollegstufenbetreuer: .....

Inhalt

1 Einführung

1.1 Einleitung

Ziel der Computergrafik ist es, die Wirklichkeit so gut wie möglich darzustellen, was jedoch nur näherungsweise erreicht werden kann. Zu diesem Zweck gibt es heutzutage im Wesentlichen zwei Verfahren: Projektionsalgorithmen (Rasterisierung) und Raytracing.
Bei der *Rasterisierung* werden die Objekte in der 3D-Szene auf eine 2D-Fläche mittels spezieller Matrixrechnungen projiziert, diese Methode wird am häufigsten, vor allem für Echtzeitgrafiken (z.B. bei Computerspielen und einigen 3D-Visualisierungsprogrammen) verwendet, da sich für die einzelnen Algorithmen spezialisierte Hardwarebauteile konstruieren lassen. Diese dedizierte Hardware liefert eine enorme Rechenleistung, bietet dafür aber nur eine stark eingeschränkte Flexibilität bezüglich der Programmierung. Referenzierung und Rekursion sind beispielsweise nicht möglich.

Beim *Raytracing* werden die Farben für jeden einzelnen Bildpunkt berechnet, indem man den Weg des Lichts zu diesem Punkt zurückverfolgt (daher der Name: Strahlenverfolgung). Raytracing erreicht dadurch eine sehr hohe Wirklichkeitsnähe, erfordert aber auch hohe Rechenzeiten und ist deshalb (noch) nicht für Echtzeitberechnungen zu gebrauchen. Die Arbeit an Echtzeitraytracing ist derzeit in vollem Gang, die Chancen, dass es die Rasterisierung als dominantes Verfahren ablösen werden sind gering; die drei großen Hardwareentwickler Intel, Nvidia und ATI/AMD arbeiten jeweils an eigenen Plattformen und Programmierschnittstellen für ihre Grafikkarten bzw. spezieller Hardware.
Zurzeit wird die Strahlenverfolgung vor allem für die Animation bei Filmen verwendet. Einige Beispiele sind die Animationsfilme *Ice Age*, *Horton hört ein Hu!* und *Robots*[1], die komplett mit Raytracing gerendert wurden. Ein weiteres Anwendungsgebiet sind Animations- und 3D-

[1] Aufgeführt sind die deutschen Titel der Kinofilme, vgl. http://www.blueskystudios.com, v.a. http://www.blueskystudios.com/content/features.php, aufgerufen 14.7.2009

Modellierungsprogramme wie z.B. Cinema 4D[2]. Raytracing wird auch in der Physik und in der Medizin verwendet, um die Ausbreitung von elektromagnetischen Strahlen zu simulieren.
Meine Facharbeit wird sich mit dem Raytracingverfahren beschäftigen und eine kurze Einführung in das Thema bieten.

1.2 Kurzer Überblick über das Verfahren

In der Wirklichkeit entsteht ein Bild (in der Kamera oder im Auge) dadurch, dass die Lichtstrahlen von der Sonne oder anderen Lichtquellen auf ein Objekt treffen und absorbiert, reflektiert oder gebrochen wird. Ein Teil der gebrochenen oder reflektierten Strahlen landet dann im Auge bzw. auf dem Sensor der Kamera und erzeugt so das Bild, das wir sehen.

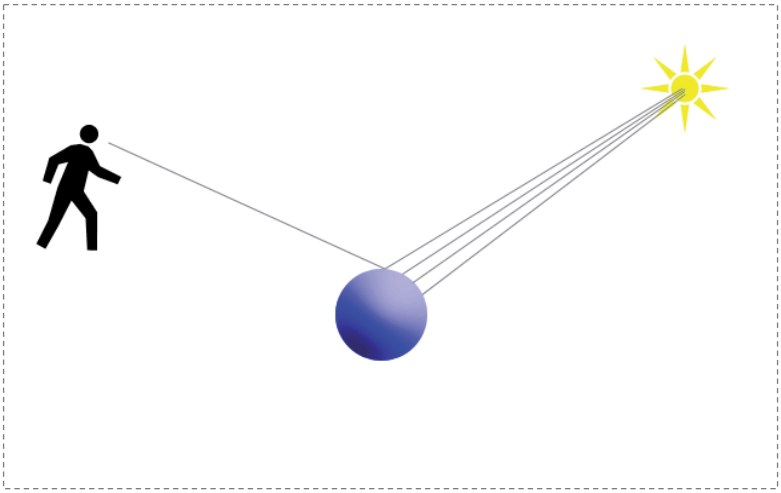

Abbildung 1: Verlauf der Lichtstrahlen von der Lichtquelle (Sonne) aus.

Hier gehen viele Strahlen von der Sonne aus, die nicht im Auge des Betrachters landen. Beim Raytracing wird der tatsächliche Vorgang rückwärts durchlaufen und nur die Strahlen getestet, die auch im Auge ankommen.

Abbildung 2 veranschaulicht das Verfahren: Vom Auge des Betrachters aus werden verschiedene Strahlen durch die Bildebene (in Abb. 2 grün dargestellt) geschickt. Zur Wahrung der Übersichtlichkeit wurde nur ein Teil der Strahlen eingezeichnet. Vom ersten Schnittpunkt des Strahls (Kamerastrahl, schwarz) mit einem Objekt werden mehrere verschiedenartige Sekundärstrahlen[3] ausgesandt.

---

[2] „MAXON – The makers of CINEMA 4D and BodyPaint 3D" http://www.maxon.net/de/home.html, aufgerufen am 17.9.2009

[3] In den meisten meiner Quellen tauchen die Begriffe „Primary Ray" und „Secondary Ray" auf. Laut Peter Shirley (vgl. [6]) sollten diese aber vermieden und stattdessen zwischen Camera Ray (Kamerastrahl), Shadow Ray

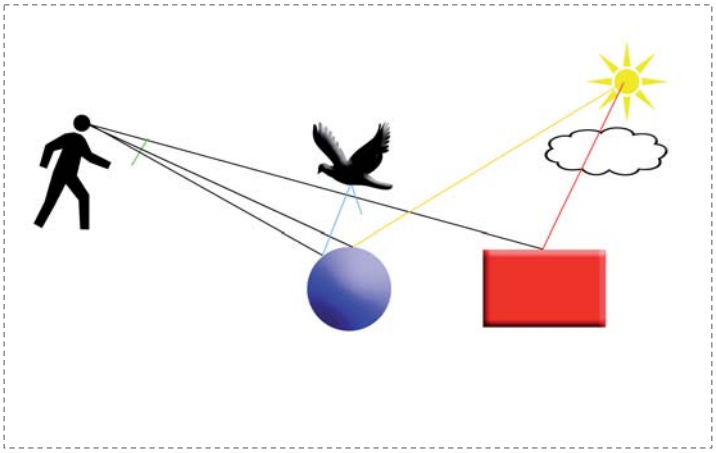

Abbildung 2: Strahlenverfolgung vom Auge des Betrachters aus

Ein Strahl (gelb) zur Ermittlung des Einfallswinkels des Lichts. Dieser wird verwendet, um die reflektierte Strahlung zu berechnen.

Der blaue Strahl ist zur Ermittlung der indirekten Beleuchtung (z.B. mittels *Ambient Occlusion*). Die indirekte Beleuchtung kommt in der Realität aus allen Richtungen von unendlich vielen verschiedenen Quellen und ist unmöglich zu berechnen. Ambient Occlusion ist eine eher unschöne Annäherung. Eine bessere Lösung ist das sogenannte *Path Tracing*, das eines der rechenaufwendigsten Verfahren in der Computergrafik überhaupt ist.
Der rote Strahl testet, ob das Objekt im Schatten liegt.

Ein zentrales Element des Raytracingverfahrens ist die Schnittpunktberechnung, sie wird bei jedem einzelnen Strahl aufs Neue durchgeführt. Dies ist der Abschnitt im Algorithmus, für den die meiste Rechenarbeit verwendet wird, aber auch der, an dem die entscheidenden Vorteile des Raytracings gegenüber der Rasterisierung greifen. Im schlimmsten Fall ist die Renderzeit direkt proportional zur Anzahl der Elemente in der Szene, wenn man nur die Primärstrahlen nimmt. Es müssen nämlich $n_{Bildpunkte} * n_{Objekte}$ Schnittpunktberechnungen durchgeführt werden. Hinzu kommen noch das Antialising und die Sekundärstrahlen, die, je näher man der Wirklichkeit kommen will, den mit Abstand größten Teil der Strahlen ausmacht.

(Schattenstrahl), Radiance Ray, Occlusion Ray und Specular Reflection Ray unterschieden werden. Wenn ich hier von Sekundärstrahlen spreche, meine ich alle Strahlen, die nicht direkt von der Kamera ausgehen.

Ein entscheidender Vorteil des Raytracing ist nun die Möglichkeit, für diese Berechnungen Optimierungsalgorithmen einzuführen. Eine einfache Optimierung ist die Verwendung eines regelmäßigen Gitters, bei dem zuerst Schnittpunktberechnungen mit den einzelnen Zellen des Gitters durchgeführt werden, und dann erst mit den Objekten, die sich in der Zelle befinden. Dies bringt vor allem bei einer sehr großen Anzahl an Objekten eine enorme Beschleunigung. Abbildung 3 zeigt eine Veranschaulichung der Methode.

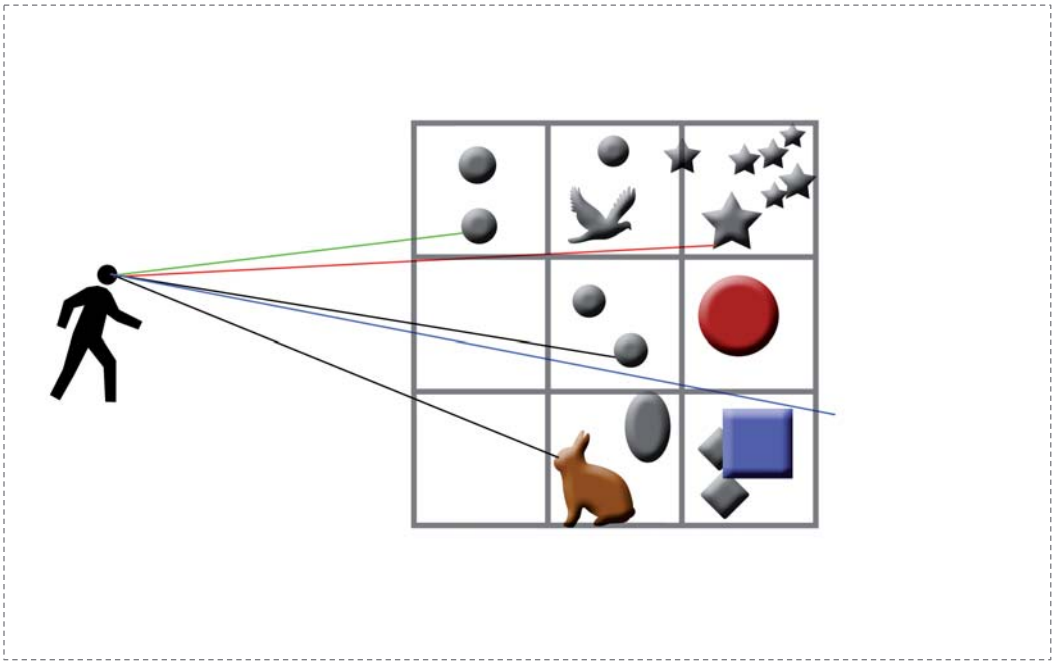

Abbildung 3: Ein regelmäßiges Gitter als Beschleunigungsmethode. Der grüne Strahl benötigt fast keine Zeit, er wird gegen zwei Kugeln getestet und schneidet eine Kugel schon in der ersten Zelle. Der rote Strahl braucht dagegen sehr viel Zeit, er durchquert zwei Zellen ohne etwas zu treffen, wird also vergeblich gegen die drei Kugeln, den Vogel und den einen Stern getestet und schneidet erst in der dritten Zelle, wo er auch noch gegen sieben Objekte getestet werden muss, den Stern. Der blaue Strahl durchquert das Gitter ganz, was in ungünstigen Fällen enorme Zeit in Anspruch nehmen kann.

2 Theoretische Grundlagen

2.1 Definitionen

Zuerst werden hier einige der wichtigsten Begriffe, die ich später verwenden werde, definiert.
Die **Bildebene** (`class Viewplane`) ist ein Rechteck, auf das die Szene projiziert wird. Sie ist vergleichbar mit der Netzhaut im Auge oder dem Sensor in einer Digitalkamera. Sie entspricht dem fertigen Bild und besteht aus den einzelnen Pixeln.

Die **Kamera** (`class Camera`) definiert Perspektive, Augpunkt, Blickrichtung, Depth of Field[4], Entfernung von der Bildebene.

Nebenstehende Skizze veranschaulicht die Verwendung von Bildebene und Kamera. Kamerastrahlen werden durch jeden Pixel auf der Bildebene geschossen. Das Sichtfeld (in der Skizze blau eingezeichnet) ist eine gerade Pyramide mit rechteckiger Grundfläche, die sich ins Unendliche erstreckt. Dadurch, dass die Bildebene eine ebene Fläche ist, und eigentlich eine Halbkugel sein müsste, entstehen an den Rändern des Bildes Verzerrungen. Diese Verzerrungen sind in der Mitte des Bildes aufgrund der Kleinwinkelnäherung nicht gravierend und können mit ausreichender Größe von d minimiert werden.

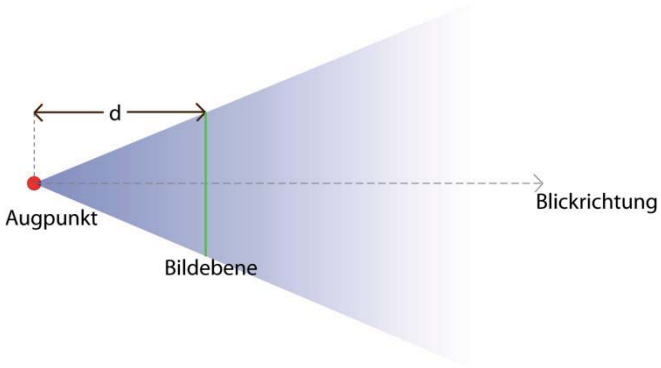

Abbildung 4: Kamera und Bildebene

Der **Strahl** (`class Ray`) ist gegeben durch

$$\vec{p} = \vec{o} + t\vec{d} \tag{1}$$

Wobei $\vec{p}$ ein Punkt auf dem Strahl,

$\vec{o}$ der Ursprungspunkt des Strahls,

$\vec{d}$ die Richtung des Strahls als Einheitsvektor und

t der Abstand des Punkts vom Ursprung entlang des Strahls ist. (siehe [1], S. 49)

[4] Tiefenunschärfe, ein Effekt, der hinzugefügt wird, um mehr Realismus zu erzeugen. Es wird eine Linse simuliert, die auf eine bestimmte Entfernung ein scharfes Bild erzeugt.

Der **Raumwinkel** dΩ ist die Fläche $dA^{\perp}$ auf der Kugel mit dem Radius 1, die vom Mittelpunkt aus gesehen von der Fläche dA verdeckt wird. Der Raumwinkel ist also das zweidimensionale Pendant zum normalen Winkel. Die Einheit ist ein Steradiant (1 sr).

Allgemein ist

$$d\Omega = \frac{\cos\theta \, dA}{{r_A}^2}$$

(siehe [1] S. 34f.)

Der Raumwinkel der kompletten Kugel (entspricht 2π beim 1D-Winkel) ist $d\Omega = \frac{4\pi r^2}{r^2} = 4\pi$.

Die **Strahldichte** L ist der Quotient aus der von einem Flächenelement dA in einer Richtung abgestrahlten Strahlungsleistung $d^2\Phi$ und dem Produkt aus der in gegebener Richtung gesehenen Fläche $dA^{\perp} = dA \cdot \cos\varphi$ und durchstrahltem Raumwinkel $d\Omega$. (aus [2], S. 1896)

$$L = \frac{d^2\Phi}{dA^{\perp} d\Omega} \tag{2}$$

Für folgende Rechnungen ist die Beziehung zwischen Strahldichte und Bestrahlungsstärke E noch besonders wichtig:

$$dE = L \cos\theta d\Omega \tag{3}$$

2.2 Schnittpunktberechnungen

Wie bereits erwähnt, sind die Schnittpunktberechnungen ein wesentlicher Aspekt des Verfahrens. Ein Beispiel, das ich hier aufführen und genauer betrachten will, ist die Kugel. Bei der Kugel zeigt sich ein weiterer erheblicher Unterschied zwischen Rasterisierung und Raytracing.

Was bei der Rasterisierung unmöglich zu rendern und nur durch Annäherung erreichbar ist, ist für einen Raytracer das am einfachste zu rendernde Objekt: eine perfekte Kugel. Für die Projektionsalgorithmen muss die Kugel in lauter Dreiecke zerlegt werden (Tessellation) und wird dadurch nie vollkommen rund; außerdem verbraucht sie dann enormen Speicherplatz. Raytracing braucht für die Darstellung einer perfekten Kugel nur die Koordinaten des Mittelpunkts und den Radius.

Für eine Kugel gilt

$$\text{Kugeloberfläche} = \{\vec{p}: |\vec{p} - \vec{c}| = r\}$$

Für die Berechnung von t setzt man Gleichung (1) in obige Beziehung ein und erhält die quadratische Gleichung

$$(\vec{d} \circ \vec{d})t^2 + 2[\vec{d} \circ (\vec{o} - \vec{c})]t + (\vec{o} - \vec{c}) \circ (\vec{o} - \vec{c}) - r^2 = 0$$

die einfach mithilfe der Mitternachtsformel zu lösen ist. Es gibt so viele Schnittpunkte, wie die Gleichung Lösungen besitzt[5]. Ist t negativ, liegt der Schnittpunkt hinter dem Augpunkt.

2.3 Die Rendering Equation

Grundlegend für das Raytracing ist die Rendergleichung. Sie beschreibt, wie sich das Licht verhält, wenn es auf eine Oberfläche trifft. In der *Hemisphärenform* wird über die Halbkugel oberhalb des entsprechenden Oberflächenpunkts integriert. Die Halbkugel wird in Raumwinkeln gemessen.

(4)

$$L_o(\vec{p}, \vec{\omega}_o) = L_e(\vec{p}, \vec{\omega}_o) + \int_{2\pi^+} f_r(\vec{p}, \vec{\omega}_i, \vec{\omega}_o)\, L_i(\vec{p}, \vec{\omega}_i) \cos\,\theta_i \, d\Omega_i$$

Wobei L_o die vom Punkt $\vec{p}$ in Richtung $\vec{\omega}_o$ ausgehende Strahldichte,

L_e die vom Objekt im Punkt $\vec{p}$ emittierte Strahlung,

L_i die am Punkt $\vec{p}$ eintreffende Strahlung,

f_r ein das Verhältnis von einfallendem und abgestrahltem Licht ist.

$2\pi^+$ bedeutet, dass die Halbkugel (Raumwinkel 2π) über dem Punkt gemeint ist.

Das Licht, das vom Material im Punkt $\vec{p}$ in Richtung des Augpunkts ($\vec{\omega}_o$) abgestrahlt wird, ist die Summe aus dem Licht, das vom Material reflektiert wird und dem, das emittiert wird.

Von der Rendergleichung gibt es noch eine *Flächenform*, in der über alle Oberflächen in der Szene integriert wird. Diese Form wird in meiner Facharbeit nicht verwendet[6], sie ist hier der Voll-

[5]Der Fall, dass die Gleichung eine Lösung hat, die Kugel also nur berührt wird, kommt praktisch nie vor.

ständigkeit halber aber noch aufgelistet. Bei ihr werden Sample Points auf der Oberfläche aller Objekte generiert und mithilfe einer Funktion geklärt, ob diese Punkte überhaupt vom Ausgangspunkt aus sichtbar sind, wohingegen bei der Hemisphärenform die Richtungen von ebendiesem aus generiert werden und sich dadurch erst die Punkte auf den Oberflächen ergeben. Die Flächenform lautet

$$L_o(\vec{p}, \vec{\omega}_o) = L_e(\vec{p}, \vec{\omega}_o) + \int_A f_r(\vec{p}, \vec{\omega}_i, \vec{\omega}_o)\, L_o(\vec{p}, -\vec{\omega}_i) V(\vec{p}, \vec{p}') G(\vec{p}, \vec{p}') dA$$

2.4 BRDF

Eine BRDF („bidirectional reflectance distribution function") beschreibt genau, wie eine Oberfläche das Licht reflektiert, das heißt wie aus der einfallenden Strahlung dL_i in einem Punkt $\vec{p}$ aus der Richtung $\vec{\omega}_i$, die austretende Strahlung dL_o in die Richtung $\vec{\omega}_o$ wird. Beim Raytracing wird dieser Vorgang, wie bereits erwähnt, rückwärts durchlaufen. Die BRDF ist im Grunde die Proportionalitätskonstante $\frac{dL_o}{dE_i}$.

Mit (3):

$$f_r(\vec{p}, \vec{\omega}_i, \vec{\omega}_o) = \frac{dL_o}{dE_i} = \frac{dL_o(\vec{p}, \vec{\omega}_o)}{L_i(\vec{p}, \vec{\omega}_i) \cos\theta_i \, d\Omega_i}$$

definiert. (aus [1], S. 224)

In der Computergrafik ist es üblich, die Materialeigenschaften (Texturen, Shader) in sogenannten *Materials* anzugeben. *Shader* sind Funktionen, die auf der Grafikkarte ausgeführt werden und ähnlich der BRDF die Farbe eines Objekts oder verschiedene Arten von Transformationen berechnen. Materials gibt es auch beim Raytracing und sie übernehmen hier im Grunde die selben Aufgaben.

---

[6] Die Flächenform kann beispielsweise bei Lichtquellen verwendet werden, die eine Ausdehnung besitzen, also geometrische Objekte mit einem Licht emittierendem Material.

2.5 Phong

Als Beispiel möchte ich hier kurz ein Material namens Phong erklären. Es besteht aus drei Teilen: *Ambient*, *Diffuse* und *Specular*. Jeder der drei Teile hat eine eigene BRDF und eine Konstante k, die angibt, welchen Anteil diese an der gesamten Strahlung ausmacht. Die Summe dieser Konstanten muss aufgrund des Energieerhaltungssatzes kleiner als 1 sein. Die Ambient-BRDF liefert lediglich $\boldsymbol{c}k_A$ ($\boldsymbol{c}$ ist die Farbe des Objekts am Punkt $\vec{p}$)[7]. Diffuse liefert einen Wert, der unabhängig von Ein- und Ausfallswinkel ist, da die diffuse Reflexion das Licht in alle Richtungen gleich verteilt. Specular ist nun abhängig vom Winkel zwischen ausfallender Strahlung und direkt reflektierter Strahlung.

Nebenstehende Skizze soll dies veranschaulichen. Je näher das Auge dem Bereich kommt, in den das Licht direkt reflektiert wird, desto heller erscheint der Punkt. Dies hat Phong[8] mit folgender Gleichung beschrieben:

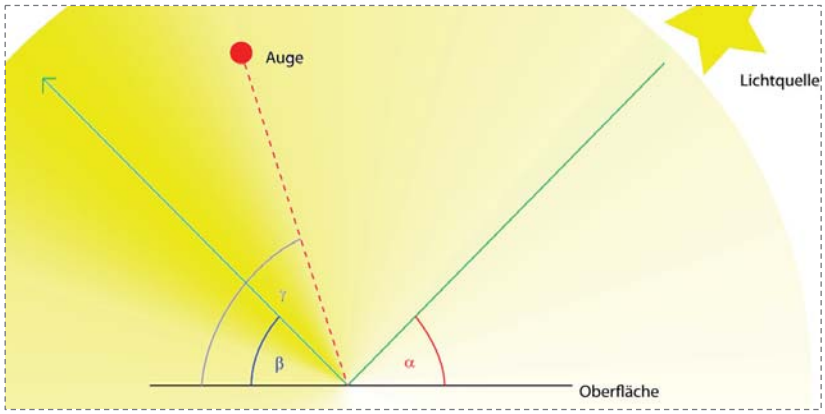

Abbildung 5: Specular-Anteil. Je näher $\vec{\omega}_o$ an $\vec{r}$ herankommt, desto heller erscheint der Punkt auf der Oberfläche. Der Winkel zwischen den beiden Vektoren ist in der 2D-Skizze $|\gamma - \alpha|$.

$$(\vec{r} \circ \vec{\omega}_o)^e$$

Wobei $\vec{r}$ die Richtung des reflektierten Lichtstrahls ist und der Exponent e ein Maß für die Rauheit des Materials ist. Ist $e \in [0; 10]$ ist noch fast kein Glanz zu sehen, wächst e sehr stark an, wird der Glanz immer stärker und kleiner. Für $e \to \infty$ wird es ein extrem kleiner Lichtpunkt.

[7] Die Farbe eines Objekts kann abhängig vom Schnittpunkt sein

[8] Phong T. Bui: Illumination for computer generated Pictures. Communications of the ACM, 18(6), 311 - 317

3 Implementierung

Als praktische Anwendung des Raytracingverfahrens habe ich einen Raytracer programmiert. Als Grundlage wurde wieder Kevin Sufferns Arbeit verwendet. Auf seiner Webseite ([2]) hat er den Quellcode für einen Raytracer in C++ bereitgestellt. Ich habe hier eine grundlegende Änderung vorgenommen: Mein Programm rendert das Bild nicht auf den Bildschirm, sondern speichert es in eine BMP-Datei und benötigt deshalb keine grafische Schnittstelle. Suffern bietet zwar eine solche an, doch um den Prozess besser überwachen zu können, eignet sich ein Konsolenprogramm besser; außerdem habe ich so das fertige Bild gleich als Datei auf meinem Computer gespeichert.

3.1 Struktur des Programms

Folgende Skizze stellt dar, wie mein Raytracer grundlegend arbeitet:

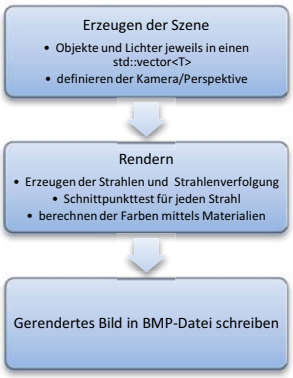

Abbildung 6: Funktionsweise des Raytracers

3.1.1 Erzeugen der Szene

Die geometrischen Objekte und Lichter werden in Instanzen der Klasse `std::vector<T>` untergebracht. Suffern hat für seinen Raytracer eine Funktion zum Erstellen der Szene in den Code einprogrammiert, weil es einem viel Arbeit erspart. Der einzige Nachteil dieses Vorgehens ist, dass man das Programm nur innerhalb der Entwicklungsumgebung ausführen kann, da eine Veränderung der Szene gleich eine Umgestaltung des Programms selbst bedeutet. Ich habe es erst mal dabei belassen, obwohl ich für einige Fälle die andere, mehr verbreitete Variante bevorzugen würde, die Szene in einer eigenen Datei zu erstellen und dann vom Raytracer einlesen zu lassen.

Alle geometrischen Objekte erben von der abstrakten Klasse GeometricObject. Sie überschreiben die Funktionen `hit(const Ray& ray, double& t, ShadeRec& s)`, `shadow_hit(const Ray& ray, double& t)` und `get_bounding_box()`.

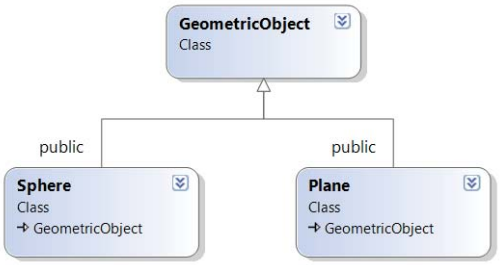

Abbildung 7: Klassendiagramm zu einigen geometrischen Objekten.

Hier ist ein Ausschnitt aus dem Quellcode der Klasse:

```
class GeometricObject {

    public:

        virtual bool hit(const Ray& ray, double& t, ShadeRec& s) const = 0;

        virtual bool shadow_hit(const Ray& ray, double& t) const = 0;

        Material* get_material(void) const;

        virtual void set_material(Material* mPtr);

        virtual BBox get_bounding_box(void) = 0;

    protected:

        Material* material_ptr;
};
```

GeometricObject.h

Die Funktionen `hit` und `shadow_hit` führen die Schnittpunktberechnung aus. Die Klasse `ShadeRec` enthält alle Informationen, die das Material für die Berechnung von L_o braucht. Die Funktion `shadow_hit` liefert kein `ShadeRec`. In diesem Fall zählt nämlich nur, ob es einen Schnittpunkt gibt oder nicht.

Die Funktion `get_bounding_box()` wird zur Erstellung einer Beschleunigungsstruktur[9] verwendet.

[9] Siehe Kapitel 5.1

3.1.2 Rendern der Szene

Für das Rendern ist die Klasse Tracer zuständig. Suffern hat diese Klasse eingeführt, um mehrere Arten von Raytracern verwenden zu können, ohne jedes Mal größere Änderungen im Quellcode vorzunehmen. Alle Tracer überschreiben die Funktion `trace_ray(const Ray ray, const int depth)`, die einen Strahl verfolgt und dann die entsprechende Farbe zurückgibt.

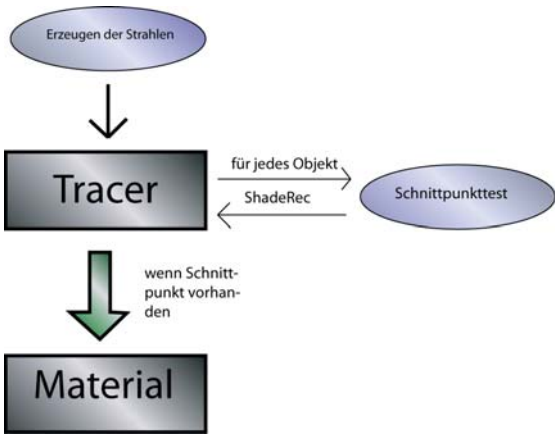

Abbildung 8: Skizze des Renderprozesses (vereinfacht)

Die Berechnung der Farbe geschieht nun bei jedem Tracer auf eine andere Art und Weise. Ich verwende RayCast, der den Strahl zuerst gegen jedes Objekt testet und, wenn ein Schnittpunkt existiert, die Funktion `Material::shade(ShadeRec& sr)` aufruft. Diese berechnet dann mithilfe der BRDF des Materials die Farbe.

Für jeden Pixel auf der Bildebene werden mehrere Strahlen ausgesendet, um Aliasingeffekte zu minimieren. Zur Optimierung wird Sampling angewandt, bei dem die Richtung der Strahlen in einem kleinen Bereich zufällig gewählt wird. Eine gute Samplingtechnik muss bestimmte Anforderungen erfüllen. Meine Facharbeit beinhaltet keine genauere theoretische Betrachtung der Sampling- und Antialiasingtechniken, es würde zu weit führen[10].

3.2 Quellcode

3.2.1 Darstellung von Zahlen, Vektoren und Farben

Eine wichtige Entscheidung, die am Anfang zu treffen ist, wenn man einen Raytracer programmiert, ist, welche Typen für die Darstellung der Zahlen verwendet werden soll, Gleitkommazahlen mit einfacher Genauigkeit (`float`) oder mit doppelter Genauigkeit (`double`). Sufferns Vorschlag ist, für geometrische Berechnungen Doubles und für Strahlungs- und Farbberechnungen Floats zu nehmen.

---

[10] Suffern behandelt das Thema Sampling sehr ausführlich (siehe [1], Kapitel 4 – 7).

Die Klasse `Vector3D` enthält neben den Komponenten viele Operationen. Zuerst einmal die grundlegenden mathematischen Operatoren (Addition, Subtraktion, Multiplikation/Division mit einem Skalar) und Skalar- und Kreuzprodukt. Zusätzlich gibt es Funktionen, die Länge und quadratische Länge des Vektors zurückgeben. Die Funktion `hat()` liefert den Einheitsvektor $\vec{v}^o$, `normalize()` normiert den Vektor selbst.

Farben sind mit der Klasse `RGBColor` vertreten. Sie enthält die Rot-, Grün- und Blauanteile als Floats und ein paar einfache Operatoren.

3.2.2 Sphere, Ray und PointLight

Die Klasse `Sphere` enthält Mittelpunkt und Radius. Folgender Codeausschnitt zeigt die Schnittpunktberechnung:

```
bool Sphere::hit(const Ray& ray, double& tmin, ShadeRec& sr) const {
    double      t;
    Vector3D    temp        = ray.o - center;
    double      a           = ray.d * ray.d;
    double      b           = 2.0 * temp * ray.d;
    double      c           = temp * temp - radius * radius;
    double      disc = b * b - 4.0 * a * c;

    if (disc < 0.0)
        return false;
    else {
        double e = sqrt(disc);
        double denom = 2.0 * a;
        t = (-b - e) / denom;

        if (t > kEpsilon) {
            tmin = t;
            sr.normal        = (temp + t * ray.d) / radius;
            sr.local_hit_point = ray.o + t * ray.d;
            return true;
        }

        t = (-b + e) / denom;

        if (t > kEpsilon) {
            tmin = t;
            sr.normal   = (temp + t * ray.d) / radius;
            sr.local_hit_point = ray.o + t * ray.d;
            return true;
        }
    }

    return false;
}
```

Sphere.cpp

Die Klasse `Ray` beinhaltet lediglich den Ursprungspunkt und den Richtungsvektor. Ist der Richtungsvektor kein Einheitsvektor, so wird er normiert.

Als Beispiel für ein Licht habe ich hier eine punktförmige Lichtquelle aufgeführt (`class PointLight`). Jedes Licht enthält eine Farbe, eine Helligkeit und eine Funktion, die die Richtung berechnet, aus der die Lichtstrahlen den angegebenen Punkt erreichen. Bei der punktförmi-

gen Lichtquelle ist das einfach $\vec{p}_L - \vec{p}$ ($\vec{p}_L$ ist der Ort des Lichts, nicht alle Lichter haben diese Eigenschaft). Ansonsten enthält jedes Licht eine Farbe und einen Wert, der die Helligkeit angibt.

Eine Möglichkeit, volumetrische Lichter umzusetzen, bei der man eigentlich etwas mogelt, ist, den Ort eines punktförmigen Lichts im Raum zu sampeln, ihm also jedes Mal eine zufällige Position zuzuordnen.

4 Weiterführende Themen und Zukunftsaussichten

4.1 Beschleunigungsalgorithmen

Besonders entscheidend für das Raytracingverfahren ist die Verwendung von Methoden die den Raytracingprozess extrem beschleunigen. Um das zu erreichen gibt es zwei Ansätze:

1. Weniger Strahlen aussenden
2. Die Zahl der Schnittpunkttests minimieren.

Um Punkt 2 zu erfüllen gibt es eine Vielzahl von geometrischen Beschleunigungsmethoden, das einfachste ist ein simples regelmäßiges Gitter. Für spezielle Zwecke eignen sich bestimmte Methoden besser als andere. So gelten k-D-Bäume als beste Methode für statische Szenen und als beste Beschleunigungsstruktur überhaupt, wohingegen bei interaktiven Szenen BVHs (Bounding Volume Hierarchies) bevorzugt werden. Langetepe und Zachmann (2006) bieten eine gute Übersicht über solche Strukturen.

4.2 Ausbaumöglichkeiten

Die Computergrafik ist ein weites Gebiet, und allein schon, um ein einigermaßen „schönes“ Bild zu erzeugen, ist viel nötig. Im Laufe des vergangenen Jahres habe ich meinem Raytracer noch einiges an Funktionalität hinzugefügt: spiegelnde Materialien, volumetrische Lichter, Dreiecke und Ebenen als geometrische Objekte und 3D-Modelle, die aus einer riesigen Anzahl von Dreiecken bestehen (also, nur ein Objekt, das diese Modelle einliest). Ein paar, mit diesen Methoden gerenderte Bilder befinden sich mit dem Quellcode und einer kompilierten Version des Programms auf der beiliegenden CD.

Um die Bilder fotorealistisch aussehen zu lassen sind zum Einen extrem komplexe und aufwendige Verfahren (Path Tracing, Photon Mapping) und zum Anderen enorme Samplingraten und damit enormer Rechenaufwand nötig. Die freie Software POV-Ray[11] ist einer der bekanntesten

---

[11] Siehe http://www.povray.org/ [Stand: 2.12.2009, aufgerufen 28.1.2010]
Ein Bild, das mit Pov-Ray erzeugt wurde:
http://upload.wikimedia.org/wikipedia/commons/e/ec/Glasses_800_edit.png

Raytracer, mit dem realistische Bilder erzeugt werden können und mit dem auch bei der Internet Ray Tracing Competition[12] erfolgreich teilgenommen wird.

4.3 Raytracing auf der Grafikkarte

In den letzten Jahren und vor allem in den letzten Monaten zeigen sich immer mehr Projekte, die sich damit beschäftigen, das Raytracing auf der GPU ausführen zu lassen. Da Grafikkarten ultra-parallel arbeiten und beim Raytracing jeder Strahl unabhängig von den anderen bearbeitet werden kann, bietet sich das an. Einen großen Schritt vorwärts brachte die Entwicklung spezieller Programmiertechnologien wie CUDA von Nvidia und Stream von ATI. Entscheidende Nachteile und Schwierigkeiten bereitet die geringe Flexibilität des Grafikprozessors. Vor allem die Rekursivität der Rendering Equation ist auf der Grafikkarte schwer zu realisieren. Es haben sich jedoch auch einige Bemühungen als erfolgreich erwiesen und lassen darauf hoffen, dass in den nächsten Jahren Raytracing in Echtzeit auf der Grafikkarte möglich sein sollte.

[12] Siehe http://www.irtc.org/irtc/ [Stand: Januar 2010, aufgerufen 28.1.2010]

5 Quellen- und Literaturangaben

(1) Kevin Suffern: *Ray Tracing from the Ground Up*. A K Peters, Wellesley, Massachusetts, 2007

(2) Kevin Suffern: *Ray Tracing from the Ground Up*. URL: http://www.raytracegroundup.com [Stand: 31.12.2009, zuletzt zugegriffen: 26.1.2010]

(3) *Der Brockhaus. Naturwissenschaften und Technik. Band 3: Ph bis Z.* F. A. Brockhaus, Mannheim; Spektrum Akademischer Verlag, Heidelberg, 2003

(4) Jacco Bikker: *Raytracing: Theory and Implementation.* URL: http://www.devmaster.net/articles/raytracing_series/part1.php (bis */part4.php), [Stand: 6.10.2005, zuletzt zugegriffen: 26.1.2010]

(5) Lexikonartikel *Raytracing* auf Computerbase.de. URL: http://www.computerbase.de/lexikon/Raytracing [Stand: 23.12.2009, aufgerufen 26.12.2009]

(6) Peter Shirley: *Terminology for Ray Tracing*, URL: http://tog.acm.org/resources/RTNews/html/rtnv21n1.html#art5

(7) E. Langetepe, G. Zachmann: *Geometric data structures for computer graphics.* A K Peters,Wellesley, Massachusetts, 2006

(8) Nvidia: CUDA Zone URL: http://www.nvidia.de/object/cuda_home_de.html#

(9) ATI Stream URL http://www.amd.com/US/PRODUCTS/TECHNOLOGIES/STREAM-TECHNOLOGY/Pages/stream-technology.aspx

Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich die Facharbeit ohne fremde Hilfe angefertigt und nur die im Literaturverzeichnis angeführten Quellen und Hilfsmittel benützt habe.

.....,	den ……	……
Ort	Datum	Unterschrift des Schülers