TECHNISCHE UNIVERSITÄT DRESDEN

FAKULTÄT INFORMATIK
INSTITUT FÜR SOFTWARE- UND MULTIMEDIATECHNIK
PROFESSUR FÜR COMPUTERGRAPHIK UND VISUALISIERUNG
PROF. DR. STEFAN GUMHOLD

Proseminar “Computergraphik”

Raytracing

Paul Elsner
(Mat.-Nr.: 3298486)

Betreuer: Dr. W. Mascolus

Dresden, 10. Juni 2008

Inhaltsverzeichnis

1 Einleitung

In der Computergrafik wird Raytracing als Verfahren verwendet um hoch realistische Bilder von dreidimensionalen Welten zu erzeugen, Beispielsweise für Kinofilme. Um den höchstmöglichen Grad an Realitätsnähe in den Bildern zu erreichen, wird versucht die Natur nachzuahmen.

Die Farben, die wir sehen, sind Lichtstrahlen die von Lichtquellen, wie beispielsweise der Sonne, ausgestrahlt und von verschiedenen Objekten reflektiert werden und schließlich unsere Augen treffen. Beim Raytracing werden jedoch nicht alle Strahlen, die von einer Lichtquelle ausgesandt werden, verfolgt, sondern nur diejenigen, welche die Bildebene treffen. Der Grund dafür ist einfach, dass alles Andere viel zu aufwändig wäre. Und um das zu ermöglichen werden die Strahlen, ausgehend von der virtuellen Kamera durch jeden einzelnen Pixel des zu erzeugenden Bildes, hinein in die Szene zurückverfolgt. Dies ist möglich, da das Verhalten der Strahlen unabhängig von deren Richtung ist.

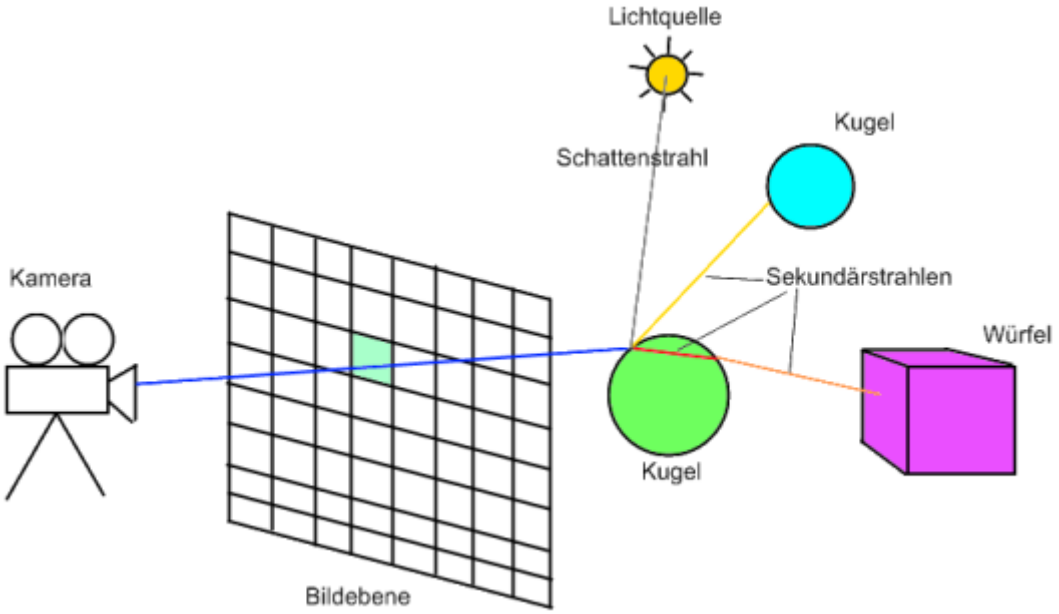

Abbildung 1: Raytracing Skizze

2 Grundlegende Konzepte

2.1 Strahl

Der Strahl ist die Grundlage des Raytracers. Alle Berechnungen beziehen sich unmittelbar auf den aktuell zu verfolgenden Strahl.

Im dreidimensionalen Raum lässt sich der Strahl einfach mittels zweier Vektoren darstellen: einem Ortsvektor, der den Ursprung des Strahls definiert und einem Richtungsvektor. Er ist also nichts anderes als eine Gerade im Raum mit der bekannten Formel:

$$\vec{x} = \begin{pmatrix} x_o \\ y_o \\ z_o \end{pmatrix} + d \begin{pmatrix} x_r \\ y_r \\ z_r \end{pmatrix}$$

Man unterscheidet zwischen verschiedenene Arten von Strahlen:

- Primärstrahlen – Strahlen, die ausgehend von der Kamera durch die Bildebene in die Szene hinein gehen
- Sekundärstrahlen – Strahlen, die die Farbe des Schnittpunktes genauer bestimmen
- Schattenstrahlen – Strahlen, die feststellen, ob ein Objekt im Schatten eines anderen liegt

Für jeden berechneten Bildpunkt muss mindestens ein Primärstrahl ausgesendet werden. Um jedoch realistisch wirkende Bilder zu erstellen, sind neben dem einen Primärstrahl noch mindestens ein Schattenstrahl und ein Sekundärstrahl erforderlich.

2.2 Verdeckung

Um eine 3D-Szene realistisch darzustellen, ist es nötig einen Eindruck von räumlicher Tiefe zu vermitteln. Dies erreicht man, indem man weiter entfernte Objekte kleiner darstellt als solche, die näher beim Betrachter liegen. Dieser Effekt wird implizit dadurch erreicht, dass sich die Gesamtmenge der Primärstrahlen von der Kamera pyramidenförmig ausbreitet. Somit werden entferntere Objekten von weniger Primärstrahlen getroffen, weshalb sie auf weniger Bildpunkten erscheinen.

Ein weiterer wichtiger Punkt, der wesentlich zur Vermittlung des räumlichen Eindruckes beiträgt, ist der, dass entfernte Objekte von näheren überdeckt werden. Um dies zu gewährleisten müssen alle Schnittpunkte mit der Szene ermittelt werden. Von diesen wird dann derjenige, der dem Betrachter am nächsten liegt, ausgewählt.

Dies kann ganz einfach mit folgendem Algorithmus umgesetzt werden:

```
for (i = 0; i < SceneManager->getNumSceneNodes(); ++i)
{
        CSceneNode *SN = SceneManager->getSceneNode(i);
        if (SN->checkForCollision(Strahl, tmpdist, tmpNormale))
        {
                if (tmpdist < dist)//nur wenn neuer Schnittpunkt näher ist
                {
                        dist = tmpdist; // neue Distanz festlegen
                        TrefferPrim = SN; //getroffenes Objekt speichern
                        Normale = tmpNormale; // Normalenvektor des getroffenen Objektes
                        hit = true; // es wurde ein Objekt getroffen
                }
        }
}
```

2.3 Beleuchtung

Eine der wichtigsten Aufgaben eines Raytracers ist die Berechnung der Beleuchtung eines Objektes. Dabei die Natur vollständig nachzuahmen wäre jedoch viel zu komplex, weshalb man sich lediglich auf Beleuchtungsmodelle beschränkt.

2.3.1 Diffuse Beleuchtung

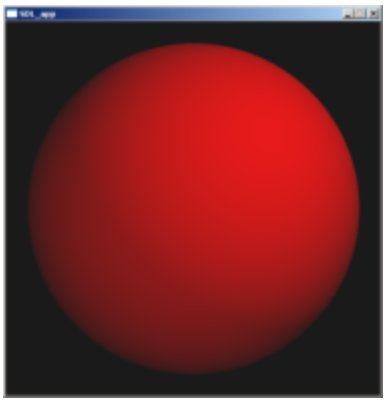

Abbildung 2: Diffuse Beleuchtung

Um die diffuse Beleuchtung eines Punktes zu berechnen, wird einfach das Skalarprodukt des Vektors von der Kamera zum Punkt und des Vektor vom Punkt zur Lichtquelle, mit dem Farbwert des Punktes und dem der Lichtquelle multipliziert.

```
//Diffuse Farbe berechnen
if (TrefferPrim->getMaterial()->GetDiffuse() > 0)
{
```

```
        const real dot = DOT(Normale,tmpL); //Skalarprodukt
        if (dot > 0.0f)
        {
                Farbe += (dot * TrefferPrim->getMaterial()->GetDiffuse())
                                        * TrefferPrim->getMaterial()->GetColor()
                                                //Farbe des Objektes
                                        * Light->getMaterial()->GetColor();
                                                //Farbe der Lichtquelle
        }
}
```

2.3.2 Spekulare Beleuchtung

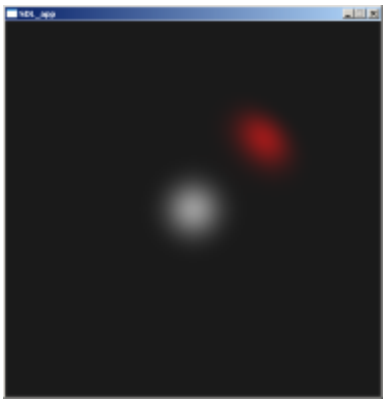

Abbildung 3: Spekulare Beleuchtung

Um glatte, glänzende Objekte darzustellen wird spekulare Beleuchtung verwendet. Die dabei entstehenden Spekulare, oder auch Highlights genannt, sollen die Reflektionen der Lichtquellen darstellen. Um diese zu berechnen, wird in den meißten Fällen, so zum Beispiel auch bei OpenGL, das Phongshading Verfahren angewandt. Um damit den Farbwert zu berechnen, potenziert man das Skalarprodukt des gespiegelten Vektors zwischen Punkt und Kamera und dem Richtungsvektor des Strahls mit einem Wert n und multipliziert diesen mit dem spekularen Wert des Objektes. Je größer dieser ist, desto stärker ist die Ausprägung der Spekulare und desto plastischer wirkt das Objekt. Der entstandene Wert wird nun mit dem Farbwert der Lichtquelle multipliziert.

```
//Speculare berechenen
if (TrefferPrim->getMaterial()->GetSpecular() > 0)
{
        const Vector3d R = tmpL - 2.0f * tmpL.Dot(Normale) * Normale;
                //Berechnung des gespiegelten Vektors
        const float dot = DOT(Strahl.direction,R);
                //Skalarprodukt
        if (dot > 0.0f)
```

```
        {
                const float spec = powf( dot, 20 ) *
                        TrefferPrim->getMaterial()->GetSpecular();
                Farbe += spec * SN->getMaterial()->GetColor();
        }
}
```

Anhand der Abbildungen 2 und 3 wird schnell deutlich, dass keines der beiden genannten Beleuchtungsverfahren alleine ausreicht, um ein Objekt wirkungsvoll zu beleuchten. Erst wenn man beide Verfahren gleichzeig anwendet, erhält man gut aussehende Bilder.

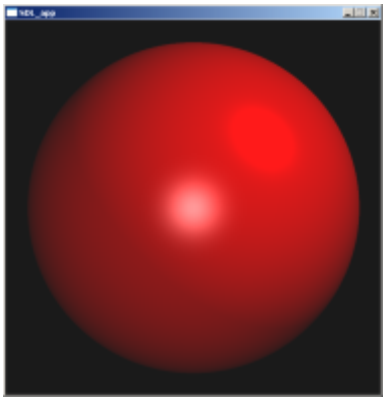

Abbildung 4: Beleuchtung Komplett

2.4 Schnittpunktberechnung

Es wurde bisher schon öfter davon gesprochen einen Schnittpunkt mit einem Objekt zu ermitteln. Nachfolgend soll gezeigt werden, wie man den Schnittpunkt eines Strahls mit einem Dreieck berechnet.

Das Dreieck spielt in der Computergrafik eine wesentliche Rolle, da damit Szenen beliebiger Komplexität modelliert werden können. Es ist durch drei linear unabhängige Vektoren $P_0, P_1, P_2 \quad (P_i \in \mathbb{R}^3)$ definiert, welche man hier als Vertices, also Eckpunkte, bezeichnet. Eine vorteilhafte Parametrisierung des Dreiecks ist durch

$$P(s,t) = (1-s-t)P_0 + sP_1 + tP_2$$

gegeben. Hierbei werden die Koordinaten $(1-s-t, s, t)$ als Gewichte oder barizentrische Koordinaten des Punktes $P(s,t)$ bezüglich P_0, P_1 und P_2 bezeichnet. Dies impliziert die Gültigkeit folgender Parametrisierung:

$$P(0,0) = P_0, P(1,0) = P_1, P(0,1) = P_2$$

Damit ein Punkt $P(s,t)$ innerhalb eines Dreieckes liegt, gibt es sechs Bedingungen:

$$0 \leq s \leq 1, \quad 0 \leq t \leq 1, \quad 0 \leq s+t \leq 1$$

Hinreichend sind aber schon diese drei Bedingungen:

$$s \geq 0, \quad t \geq 0, \quad s+t \leq 1$$

Die Richtungsableitungen

$$P_s(s,t) = P_1 - P_0, \quad P_t(s,t) = P_2 - P_0$$

und die Normale

$$N(s,t) = P_s \times P_t$$

sind für alle Ebenenpunkte konstant, also unabhängig von den Parameterwerten s und t.
Die (x,y,z)-Koordinaten eines Schnittpunktes P können durch Einsetzen der parametrischen Strahlengleichung

$$\vec{P} = \vec{O} + d\vec{D}$$

in die implizite Dreiecksebenengleichung

$$E(P) = \vec{N}\vec{P} - a = \vec{N}(\vec{O} + d\vec{D}) - a = 0$$

und der anschließenden Berechnung der Nullstelle d leicht berechnet werden. Die (s,t) Parameter eines solchen Schnittpunktes sind

$$s = \frac{area(P_0, P, P_2)}{area(P_0, P_1, P_2)}, \quad t = \frac{area(P_0, P_1, P)}{area(P_0, P_1, P_2)}$$

Diese Flächenverhältnisse ändern sich jedoch nicht, wenn man die Punkte P, P_1, P_2 und P_3 auf eine der Koordinatenachsenebenen (x,y), (y,z) oder (z,x) projiziert. Um dabei Ungenauigkeiten zu reduzieren, sollte die Ebene gewählt werden, auf der der projizierte Flächenanteil am größten ist. Hat zum Beispiel die Ebenennormale N in der z-Komponente ihren betragsmäßig größten Wert, so vereinfacht sich die Berechnung von s und t zu

$$\begin{aligned} s &= \frac{(x-x_0)(y_2-y_0) - (x_2-x_0)(y-y_0)}{(x_1-x_0)(y_2-y_0) - (x_2-x_0)(y_1-y_0)}, \\ t &= \frac{(x_1-x_0)(y-y_0) - (x-x_0)(y_1-y_0)}{(x_1-x_0)(y_2-y_0) - (x_2-x_0)(y_1-y_0)} \end{aligned}$$

Die explizite Berechnung von s und t bringt zwei Vorteile mit sich. Einerseits kann man mittels der drei hinreichenden Bedingungen schnell feststellen, ob ein Punkt innerhalb eines Dreieckes liegt und gegebenenfalls die Berechung vorzeitig abbrechen. Andererseits lassen sich s und t dazu verwenden, um die Normalen über dem Dreieck zu interpolieren.

Die Umsetztung des Algorithmus unterteilt sich eine Vorberechnung von Werten, die unabhängig von dem Strahl ist und in die Berechnung die zur Laufzeit ausgeführt wird.

```
Triangle(const CVertex a_p1, const CVertex a_p2, const CVertex a_p3) :
       p1(a_p1), p2(a_p2), p3(a_p3)
{
        const Vector3d c = p2.GetPos() - p1.GetPos();
        const Vector3d b = p3.GetPos() - p1.GetPos();

        this->Normale = b.Cross(c);

        //Ermitteln der Dominanten Achse
        if ( _fabs( this->Normale.x ) > _fabs( this->Normale.y ) )
        {
                if ( _fabs( this->Normale.x ) > _fabs( this->Normale.z ) )
                        //X ist dominante Achse
                        domAchse = 0;
                else
                        //Z ist dominate Achse
                        domAchse = 2;
        }
        else
        {
                if ( abs( this->Normale.y ) > abs( this->Normale.z ) )
                         //Y ist dominante Ache
                        domAchse = 1;
                else
                        //Z ist dominante Achse
                        domAchse = 2;
        }
        int U,V;
        U = (domAchse + 1) % 3;
        V = (domAchse + 2) % 3;

        const float krec = 1.0f / Normale.cell[domAchse];

        nu = Normale.cell[U] * krec;
        nv = Normale.cell[V] * krec;
        nd = Normale.Dot(p1.GetPos()) * krec;

        const float reci = 1.0f / (b.cell[U] * c.cell[V] - b.cell[V] * c.cell[U]);
        bnu = b.cell[U] * reci;
        bnv = -b.cell[V] * reci;

        cnu = c.cell[V] * reci;
        cnv = -c.cell[U] * reci;

        NORMALIZE( Normale );
}
```

```
bool checkForCollision(CRay& Strahl, float& dist, Vector3d& Normale)
        //gibt die Distanz zum Schnittpunkt und die Normale zurück
{
#define ku modulo[domAchse + 1]
#define kv modulo[domAchse + 2]

        const Vector3d O = Strahl.Origin, D = Strahl.direction, A = p1.GetPos();

        if ( DOT(D, this->Normale) < 0) return false; //Rückseite?

        const float lnd = 1.0f  / (D.cell[domAchse] + nu * D.cell[ku] + nv * D.cell[kv]);
        const float t = (nd - O.cell[domAchse] - nu * O.cell[ku] - nv * O.cell[kv]) * lnd;

        if ( !(dist > t && t > 0)) return false; //weiter weg als anderer Schnittpunkt?

        const float hu = O.cell[ku] + t * D.cell[ku] - A.cell[ku];
        const float hv = O.cell[kv] + t * D.cell[kv] - A.cell[kv];
        const float beta = m_U = hv * bnu + hu * bnv;
        if (beta < 0 ) return false;
        const float gamma = m_V = hu * cnu + hv * cnv;
        if (gamma < 0 ) return false;
        if ((beta + gamma) > 1) return false;

        dist = t;
        const Vector3d N1 = p1.GetNormal();
        const Vector3d N2 = p2.GetNormal();
        const Vector3d N3 = p3.GetNormal();

        Normale = N1 + m_U * (N2 - N1) + m_V * (N3 - N1);
        NORMALIZE( Normale );
        return true; //wurde getroffen
}
```

3 Erweiterte Konzepte

Die oben vorgestellten Konzepte reichen aus, um ein 3D-Szene darzustellen. Doch man wird recht schnell feststellen, dass die Bilder langweilig aussehen. Abhilfe schafft der folgende Abschnitt, in dem ich auf die Darstellung von Effekten wie Schatten, Reflexion und Refraktion eingehe.

3.1 Schatten

Ein Schatten auf einem Objekt entsteht, wenn von ihm aus die Lichtquelle nicht sichtbar ist, da sie von einem anderen Objekt verdeckt wird. Und genau das wird beim Raytracing geprüft.
Zuerst wird ein Strahl, ausgehend vom zu prüfenden Punkt, zur Lichtquelle berechnet. Dieser so genannte Schattenstrahl wird dann mit jedem Objekt der Szene auf Schnittpunkte geprüft. Ist dies das Fall liegt der Punkt im Schatten, andernfalls wird die Beleuchtung wie gewohnt berechnet.

```
//Schatten berechen
const Vector3d L = (Light->getPos() - pi); //Vektor zur Lichtquelle
float shade = 1.0f;
float tdist = L.Length();
const Vector3d tmpL = L * (1.0f / tdist); //Normalisieren
CRay tmpRay(pi + tmpL* EPSILON, tmpL); //Neuen Strahl anlegen
for ( int s = 0; s < SceneManager->getNumSceneNodes(); ++s)//für jedes Objekt
{
        CSceneNode *PR = SceneManager->getSceneNode(s);
        if( (PR != SN) && PR->checkForCollision(tmpRay, tdist, tmpNormale))
        {
                shade = 0; //Falls getroffen
                break; //ein Objekt reicht
        }
}
```

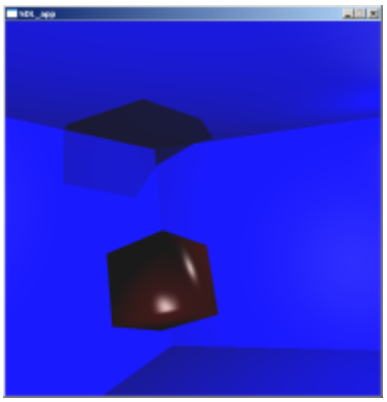

Abbildung 5: Schatten

3.2 Reflexion

Für die Erstellung von Reflexionen kommen das erste Mal die schon erwähnten Sekundärstahlen zum Einsatz. Der einzige Unterschied zu den Primärstrahlen liegt darin, dass sie nicht bei der Kamera, sondern am Punkt, dessen Farbwert ermittelt werden soll, beginnen.

Man unterscheidet zwischen zwei Arten von Reflexion, der totalen und der diffusen. Bei der Totalreflexion wird der Lichtstrahl an der Oberfläche eines Objektes vollständig reflektiert. Das heißt es gilt das physikalische Prinzip „Einfallswinkel ist gleich Aussfallswinkel“und die Spiegelung hat die Farbe des gespiegelten Objektes. Bei der diffusen Reflexion jedoch, dringt der Lichtstrahl ein Stück weit in das Objekt ein und wird erst dort von den Farbpigmenten reflektiert. Dabei nimmt der Strahl die Farbe des Objektes an, wodurch die Reflextion in dieser Farbe erscheint. Im folgenden möchte ich nur auf den ersten Fall eingehen.

Wie bereits erwähnt muss zuerst der Sekundärstrahl berechnet werden. Dessen Richtungsvektor errechnet sich aus der folgenden Formel:

$$\vec{D_s} = \vec{D_p} - 2 * (\vec{D_p} \times \vec{N}) * \vec{N}$$

Mit dem Sekundärstrahl wird dann die Funktion zur Strahlverfolgung rekursiv aufgerufen, wodurch auf diesen die gleichen Berechnungen wie auf den Primärstrahl angewandt werden.

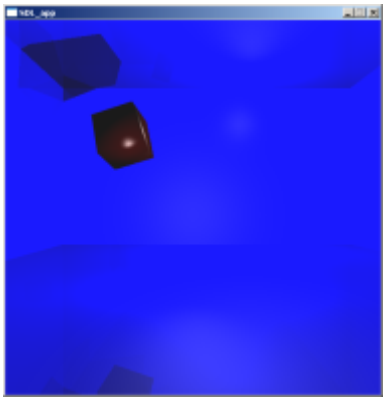

Abbildung 6: Reflektion

```
//Reflektion berechnen
if(TrefferPrim->getMaterial()->GetReflection() > 0.0f)
{
        if( depth < REFLECTION_DEPTH)
        {
                const Vector3d R = Strahl.direction - 2.0f
                        * DOT(Strahl.direction,Normale) * Normale; //Richtungsvektor
                Color rcol(0.0f, 0.0f, 0.0f);
```

```
                CRay tempRay(pi+R*EPSILON, R); //Sekundärstrahl erstellen
                TraceRay(tempRay, rcol, depth + 1);//Rekursiver Aufruf
                Farbe += TrefferPrim->getMaterial()->GetReflection() * rcol
                        * TrefferPrim->getMaterial()->GetColor();
        }
}
```

3.3 Refraktion

Refraktion ist die Brechung des Lichtes, die eintritt, wenn der Lichtstrahl von einem Medium zu einem anderen übergeht und die Medien unterschiedliche Lichtausbreitungsgeschwindigkeiten haben. Diesen

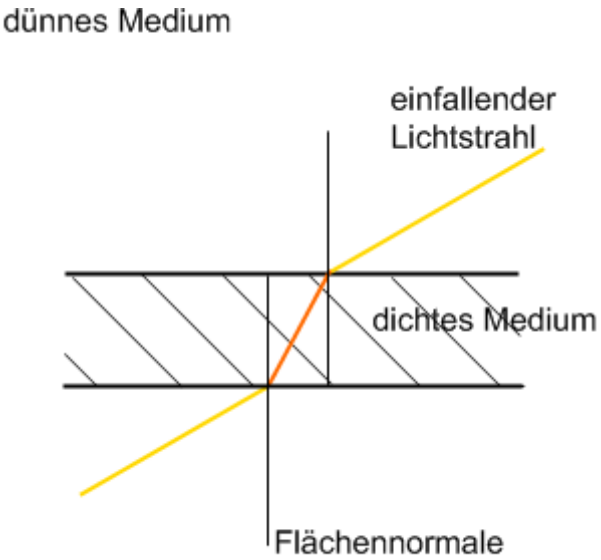

Abbildung 7: Refraktion Skizze

Effekt kann man beim Raytracing ermöglichen, indem wieder ein Sekundärstrahl benutzt wird. Dessen Richtungsvektor hängt vom Brechungsindex ab. Dieser ergibt sich folgendermaßen:

$$k = \frac{\text{Lichtgeschwindigkeit im äußeren Medium}}{\text{Lichtgeschwindigkeit im inneren Medium}}$$

Im Programm wird der Prozess der Refraktion so umgesetzt:

```
// Refraktion berechnen
float refr = TrefferPrim->GetMaterial()->GetRefraction();
if ((refr > 0) && (a_Depth < TRACEDEPTH))
{
        float rindex = TrefferPrim->GetMaterial()->GetRefrIndex();
        float k = a_RIndex / rindex;
        vector3 N = Normale * (float)result;
        //result gibt an, ob der Strahl von Innen nach Außen geht (-1) oder umgekehrt (1)
```

```
        float cosI = -DOT( N, Strahl.GetDirection() );
        float cosT2 = 1.0f - k * k * (1.0f - cosI * cosI);
        if (cosT2 > 0.0f)
        {
                Vector3d T = (n * Strahl.direction) + (k * cosI - sqrtf( cosT2 )) * N;
                Color rcol( 0, 0, 0 );
                float dist;
                TraceRay( Ray( pi + T * EPSILON, T ), rcol, depth + 1);
                Farbe += rcol;
        }
}
```

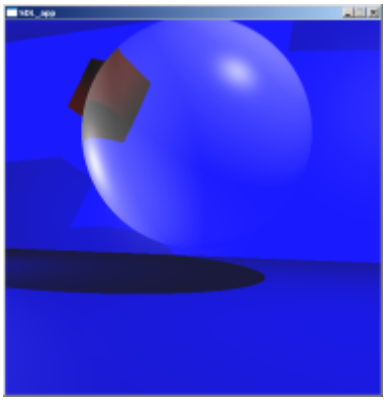

Abbildung 8: Refraktion

Ferner bleibt festzuhalten, dass eine Schwächung der Strahlungsintensität proportional zur Weglänge beim Durchgang durch eine absorbierende Substanz, in unserem Fall durch ein farbiges Medium, auftritt. Das gleiche Verhalten ist auch in Abhängigkeit von der Konzentration der absorbierenden Substanz zu beobachten. Die zu Grunde liegenden mathematischen Beziehungen sind im Lambert-Beerschen Gesetz formuliert.

4 Zusammenfassung

Zusammenfassend ist zu sagen, dass durch das Verfahren des Raytracing ausgesprochen realistische Bilder erzeugt werden können, welche der Natur sehr nahe kommen. Die beschriebenen Methoden sind jedoch nur ein kleiner Teil von dem, was man alles mit Raytracing bewerkstelligen kann. Beispielsweise wurden die globale Beleuchtung oder großflächige Lichtquellen, die weiche Schatten werfen, noch gar nicht erwähnt. Durch die Addition solcher Effekte erhöht sich jedoch die Rechenzeit drastisch, da sehr viele Strahlen berechnet werden müssen. Der Aufwand wird jedoch durch die Qualität der entstehenden Bilder gerechtfertigt. Diese ist zum Teil so hoch, dass es dem Betrachter schwerfällt, zwischen Natur und generierten Bild zu unterscheiden.

5 Quellennachweis

- „HIGHTLIGHT PC“Markus Rahlff / Hilmar Koch
- „Entwurf eines objektorientierten Visualisierungssystems auf der Basis von Raytracing“ Alwin Gröne
- http://www.devmaster.net/articles/raytracing_series/part1.php
- http://de.wikipedia.org/wiki/Raytracing
- http://en.wikipedia.org/wiki/Ray_tracing
- „Realtime Ray Tracing and Interactive Global Illumination“Ingo Wald

Erklärungen zum Urheberrecht

Hiermit erkläre ich das die von mir verfasste Arbeit selbstständig verfasst wurde und nur die im angegebenen Quellen Verwendung gefunden haben.