Benjamin Schnaidt

Raytracing

IRTC, Jaime Vives Piqueres, POVMan 0.7

Prolog

Im der folgenden Ausarbeitung zum Proseminar-Votrag geht es um das Thema Raytracing, das dahinter steckende Prinzip und Optimierungsmethoden. Hauptsächlich basiert der Inhalt auf dem Buch "Amiga reflections - Traumwelt und Realismus" (Fuchs, Carsten) - näheres dazu und die anderen verwendeten Quellen befinden im letzten Abschnitt.

Kurz ausgedrückt hat Raytracing das Ziel möglichst photorealistische Bilder durch Rendern zu erzeugen. Es basiert dabei auf dem Prinzip der Verfolgung von Lichtstrahlen, wie der englische Ausdruck schon sagt. Die Absicht ist dabei möglichst viele Eigenschaften der Realität naturgetreu zu simulieren, um nahezu photorealistische Resultate zu erzielen. Im Gegensatz zu anderen Verfahren ist daher der Zeitaufwand beim Raytracing um ein Vielfaches höher. Dafür sind aber die meisten grundlegenden Eigenschaften von Gegenständen wie Schatten, Spiegelung und Brechung relativ einfach zu implementieren. Die folgenden Abschnitte bauen zuerst schrittweise einen umrißhaften Raytracer auf und erklären dabei die nötigen Details.

Bis auf die Grafiken aus dem Internet Raytracing Competition (die Titelgrafik und die Grafiken am Ende) wurden die Bilder von mir mit CorelDRAW 9 und dem Raytracer CorelDREAM 3D 8 (auch bekannt als Raydream von Fractal Design) erstellt, wobei ich teilweise Cliparts aus den Bibliotheken verwendet habe.

Besonderen Dank geht an dieser Stelle noch an meinen Betreuer Michael Wand, der mir mit vielen Tipps und Anmerkungen geholfen hat, Fehler zu vermeiden oder sie rechtzeitig zu korrigieren. Außerdem hat er sich für die Korrektur des Vortrages und der Ausarbeitung immer viel Zeit genommen und das obwohl letztere ein gutes Stück länger als geplant ausgefallen ist.

Benjamin Schnaidt (BSC@swol.de)

Inhaltsverzeichnis

1	**RAYTRACING**	**4**
1.1	**"Lichtstrahlen verfolgen"**	**4**
1.1.1	Lichtteilchen	4
1.1.2	Vom Betrachter zur Lichtquelle	4
1.2	**Ein einfacher Raytracer**	**5**
1.2.1	Bilderzeugung	5
1.2.2	Raytrace_1	6
1.2.3	Schatten	6
1.2.4	Raytrace_2	7
1.2.5	Spiegelung	7
1.2.6	Raytrace_3	8
1.2.7	Brechung	8
1.2.8	Raytrace_4	9
1.3	**Die Beleuchtungsformel**	**10**
1.3.1	Farbberechnung mit 5 Vektoren	10
1.3.2	Farbmodell	10
1.3.3	Lichtanteile	11
1.3.4	Direktes Licht	12
1.3.5	Glanzlicht	13
1.3.6	Glanzlichtfunktion	14
1.3.7	Gesamtformel	14
1.3.8	Verschiedene Materialien	15
1.4	**Suche_Objekt**	**17**
1.4.1	Die Prozedur	17
1.4.2	Berechnung des Schnittpunktes	17
1.4.3	Aufwand	19
1.4.4	Reduzierung der Schnittests	20
1.4.5	Suche_Objekt_2	21
1.4.6	Effizienz	21
1.4.7	Umgrenzungskörper-Hierarchie	22
1.4.8	Suche_Objekt_3	23
1.4.9	Effizienz	23
1.4.10	Gittermethode	25
1.4.11	Vektorgenerator	25
1.4.12	Suche_Objekt_4	26
1.4.13	Octrees	27
1.4.14	Suche_Objekt_5	28
1.5	**Antialiasing**	**28**
1.6	**The Internet Ray Tracing Competition**	**29**
1.6.1	Compartmentalized Sea	29
1.6.2	Always running, never the same...	31
1.6.3	The wet bird	32
1.6.4	Lord of the Hunt	33
1.6.5	Spider project	33
1.7	**Verwendete Quellen**	**34**
2	**INDEX**	**35**

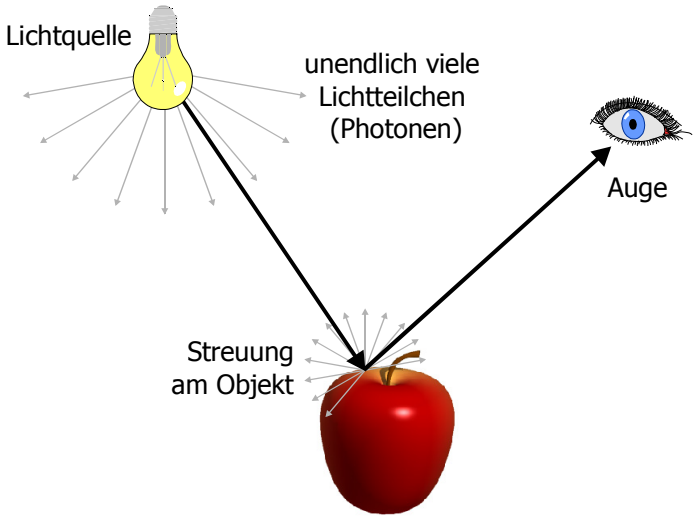
1 RAYTRACING

1.1 "Lichtstrahlen verfolgen"

1.1.1 Lichtteilchen

Wie der Name *Raytracing* schon sagt, geht es hier um die "Verfolgung von Lichtstrahlen", um möglichst photorealistisch wirkende Bilder zu erzeugen.

In der Realität werden von einer Lichtquelle aus unendliche viele Lichtteilchen bzw. Photonen in alle Richtungen abgestrahlt und treffen dabei auf verschiedene Objekte. Ein Teil des Lichtes wird dort absorbiert und in Wärme umgewandelt, der andere wird aber wieder abgestrahlt und dabei noch einmal in alle Richtungen gestreut. Die gestreuten Lichtstrahlen, die vom Objekt aus das Auge erreichen, sind das, was wir von dem Objekt bzw. allgemein von unserer Umgebung wahrnehmen.



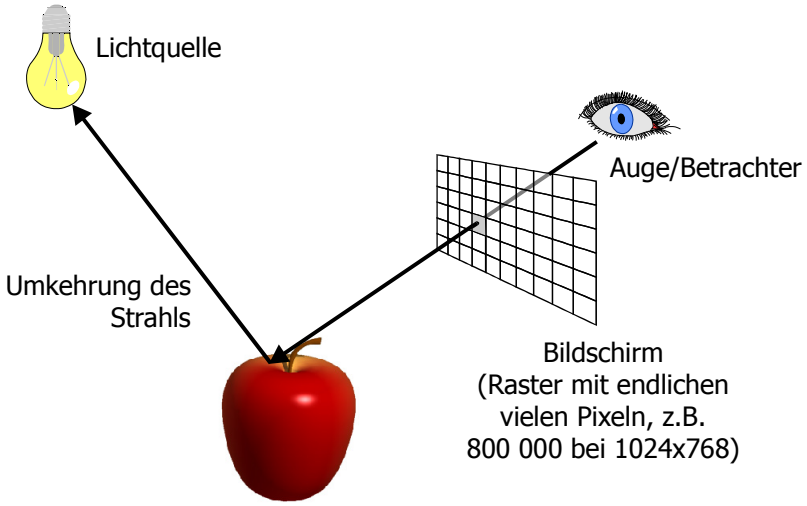
Sowohl von der Lichtquelle als auch bei der Streuung am Objekt gehen jeweils praktisch unendliche viele Lichtteilchen in jede Richtung. Auch wenn nur ein kleiner Teil schließlich in das Auge gelangt, handelt es sich dabei immer noch um unendlich viele. Daher benötigt man Wege, um die Anzahl der möglichen Strahlen auf die wichtigsten zu beschränken.

1.1.2 Vom Betrachter zur Lichtquelle

Zur Vereinfachung wird nun folgendes verändert ...

- **Umkehrung des Strahls:** Anstatt von der Lichtquelle zum Objekt und von da aus zum Betrachter zu gehen, kann man die Richtung des Strahls auch rückwärts berechnen, d.h. vom Betrachter aus (physikalische Gesetze der Strahlenoptik, mehr dazu weiter unten). Dadurch wird die Anzahl der nötigen Strahlen zwar eingeschränkt, ist aber ingesamt gesehen immer noch unendlich.

- **Bildschirmraster:** Da das Bild aber letztendlich auf einem Monitor dargestellt wird, der nur über ein Raster verfügt, das aus endlich vielen Pixeln besteht, kann man die obige Menge noch zusätzlich einschränken. Man betrachtet für jeden Rasterpunkt daher jeweils nur einen Strahl, der vom Auge aus auf das Objekt trifft.



Bei einer Standard-Auflösung von 1024×768 Pixeln sind das aber auch bereits fast 800 000 Strahlen, die berechnet werden müssen.

1.2 Ein einfacher Raytracer

1.2.1 Bilderzeugung

Mit dem obigen Wissen läßt sich bereits ein einfacher Raytracer schreiben. Um dies zumindest im groben zu Verdeutlichen, enthalten die folgenden Abschnitte kleine Stück an Pseudo-Code, die das Rahmenprogramm für einen einfachen Raytracer umschreiben. Die Details des Codes fehlen dabei natürlich, da man dazu noch zusätzliche Dinge wie z.B. eine Form der Objektbeschreibung benötigen würde.

```
Bilderzeugung()
{
    for (alle Pixel des Bildschirms)
        {
        Strahl = Strahl vom Augenpunkt zum Pixel;
        Farbe = Raytrace_1(Strahl);
        Pixelfarbe(Farbe);
        }
}
```

Wie bereits erwähnt, muß beim Raytracing jeder Pixel des Schirms einzeln berechnet werden. Daher sieht die Funktion **`Bilderzeugung`** auch recht einfach aus: sie muß nur für alle Pixel des Bildschirms jeweils den Raytracer aufrufen und die berechnete Farbinformation zum Bildschirm schicken (mit Hilfe von **`Pixelfarbe`**). Als Eingabe benötigt **`Raytrace_1`** nur den entsprechenden Strahl vom Augenpunkt zum Pixel, d.h. einen Vektor (mit Startpunkt). Man geht hier davon aus, daß die vorhandenen Objekte und die Lichtquellen in globalen Variablen gespeichert sind, die von **`Raytrace_1`** aus zugänglich sind.

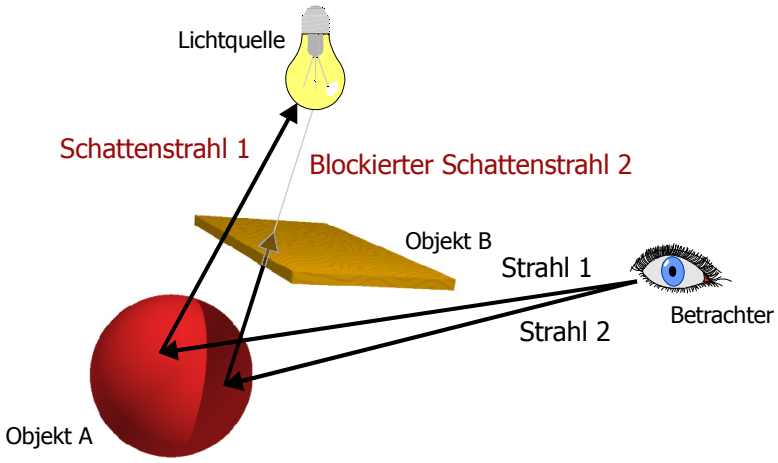
1.2.2 Raytrace_1

```
Raytrace_1(Strahl)
{
    Objekt = Suche_Objekt(Strahl);
    if (Objekt gefunden)                    Strahl Betrachter → Objekt
        Farbe = Berechne_Farbe(Strahl,Objekt,
          Schnittpunkt,Lichtquelle);
    else                                    Strahl Objekt → Lichtquelle
        Farbe = Hintergrund;
    return(Farbe);
}
```

Suche_Objekt bekommt als Eingabe den an **Raytrace_1** übergebenen **Strahl** und prüft nun, ob dieser vom Betrachter aus auf ein Objekt trifft mit Hilfe von **Suche_Objekt** (das später im Detail besprochen wird). Wir kein Objekt gefunden, kann einfach die Hintergrundfarbe zurückgegeben werden. Ansonsten muß aber noch zusätzlich der Strahl vom ersten getroffenen Objekt zur Lichtquelle überprüft werden - dies übernimmt hier **Berechne_Farbe**. Als Eingabe wir der vorherige **Strahl**, das **Objekt**, der **Schnittpunkt** mit dem **Objekt** und die entsprechende **Lichtquelle** benötigt (es könnte ja mehrere Lichtquellen geben). Aus diesen Informationen kann der Strahl Objekt→Lichtquelle berechnet werden und abhängig vom Winkel wird die entsprechende Farbinformation zurückgegeben.

1.2.3 Schatten

Nun fehlen aber noch eine Reihe an Spezialeffekten. Schatten sind am einfachsten in den obigen Raytracer einzufügen. Die Begriffe Schatten und Schattierung können an dieser Stelle leicht verwechselt werden. Die Schattierung wird in unserem Raytracer bereits von **Berechne_Farbe** übernommen - d.h. daß die Oberflächenfarbe der Kugel von hell nach dunkel abnimmt, je nachdem welche Seite der Lichtquelle zugewandt ist. Im Schatten liegt ein Objekt oder ein Teil des Objektes dann, wenn von dort aus die Lichtquelle nicht mehr zu sehen ist (wie im rechten Teil der unteren Kugel). Um dies festzustellen muß man nur vom Schnittpunkt aus zur Lichtquelle einen weiteren Strahl schicken. Dieser überprüft, ob wieder ein Objekt geschnitten wird oder nicht. Wenn ja, dann liegt das Objekt A im Schatten des zweiten Objektes B und erhält folglich kein Licht von der Lichtquelle.



1.2.4 Raytrace_2

In **Raytrace_2** wird nun das obige Prinzip implementiert.

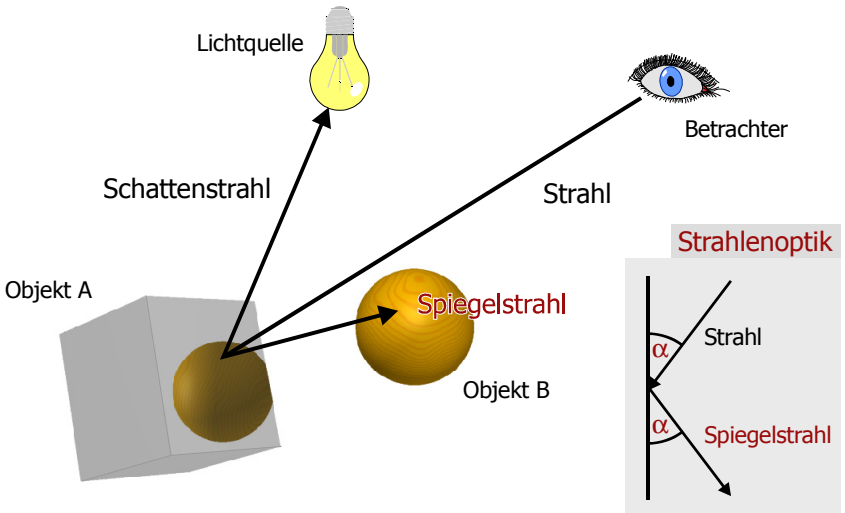
```
Raytrace_2(Strahl)
{
     Objekt = Suche_Objekt(Strahl);
     if (Objekt gefunden) {
          Schattenstrahl = Strahl von Schnittpunkt zur
            Lichtquelle;
          blockiert = Suche_Objekt(Schattenstrahl);
          if (blockiert = false)
               Farbe = Berechne_Farbe(Strahl,Objekt,
                 Schnittpunkt,Lichtquelle);
          else
               Farbe = Schwarz; // Oder Umgebungslicht }
     else
          Farbe = Hintergrund;
     return(Farbe);
}
```

Schattenstrahl

Suche_Objekt muß nun ein zweites mal aufgerufen werden, um den Schattenstrahl vom Objektschnittpunkt zur Lichtquelle zu überprüfen. Ist **blockiert** ≠ **false**, dann wurde ein Objekt gefunden und der Punkt liegt im Schatten und hat die Farbe **Schwarz**. Eventuell kann man hier noch eine gewisse Umgebungshelligkeit annehmen, dann wählt man statt Schwarz die Objektfarbe und multipliziert sie mit dem Faktor des Umgebungslichtes (daraus entsteht dann aber keine Schattierung, da das Umgebungslicht überall gleich stark ist). Ansonsten kann wieder **Berechne_Farbe** zur Berechnung der Farbe des Punktes benutzt werden, falls die Lichtquelle vom Objekt aus direkt sichtbar ist (**blockiert** = **false**).

1.2.5 Spiegelung

Spiegelung trägt wesentlich zur realistischen Abbildung von Gegenständen bei, da viele Gegenstände zumindest zum Teil Licht reflektieren (Kunststoff, Metall, ...). Da Raytracing auf der Verfolgung von Lichtstrahlen basiert, ist auch die Spiegelung sehr einfach zu implementieren. Nach den Gesetzen der Strahlenoptik wird der einfallende Strahl mit dem Winkel α zur Oberfläche des Objektes in genau demselben Winkel wieder abgestrahlt ...



Damit kann einfach ein Spiegelstrahl zum Sehstrahl berechnet werden. Schneidet dieser ein zweites Objekt, so spiegelt sich das im ersten.

1.2.6 Raytrace_3

Die Implementierung in den Raytracer erfolgt rekursiv.

```
Raytrace_3(Strahl)
{
     Objekt = Suche_Objekt(Strahl);
     if (Objekt gefunden) {
          Schattenstrahl = Strahl Schnittpunkt -> Lichtquelle;
          blockiert = Suche_Objekt(Schattenstrahl);
          if (blockiert = false)
               Farbe2 = Berechne_Farbe(Strahl,Objekt,
                 Schnittpunkt,Lichtquelle);
          else Farbe2 = Schwarz;
          Spiegelstrahl = Reflektions_Strahl(Strahl,
            Objekt,Schnittpunkt);
          Farbe = Raytrace_3(Spiegelstrahl) + Farbe2; }
     else
          Farbe = Hintergrund;
     return(Farbe);
}
```

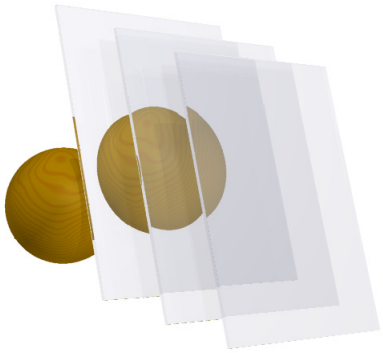
Spiegelstrahl

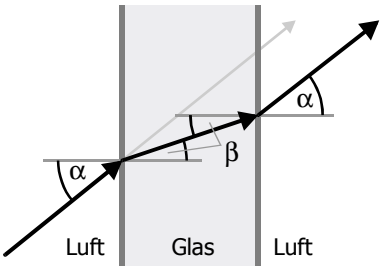
Die zusätzliche Subroutine **`Reflektions_Strahl`** berechnet aus dem Strahl, dem Objekt und dem Schnittpunkt den reflektierte Strahl bzw. **`Spiegelstrahl`** (nach dem obigem Gesetz). Dann ruft **`Raytrace_3`** sich selbst auf und überprüft sozusagen, was vom Punkt der Oberfläche des spiegelenden Objekts aus zu sehen ist. Dies ergibt dann die neue Farbe die zur Grundfarbe des Objektes hinzuaddiert wird - nicht alle Objekte spiegeln ja vollständig, sondern die meisten nur einen Teil des Lichtes.

An dieser Stelle wäre es natürlich möglich, daß bei zwei spiegelnden Objekten sich **`Raytrace_3`** in eine Endlosschleife verzweigt, falls der Spiegelstrahl ständig hin- und hergeschickt wird. Daher benötigt ein echter Raytracer hier noch eine zusätzliche Angabe: die Reflektions-Tiefe. Bei jedem Aufruf von **`Raytrace_3`** wird dann überprüft, wie oft es sich bereits rekursiv selbst aufgerufen hat und ab einem gewissen Wert wird abgebrochen. Im Normalfall genügt schon ein Wert von 2 oder 3, da der Beitrag der doppelt gespiegelten Objekte schnell sehr klein wird (dies wird später bei der Farbberechnung deutlich).

1.2.7 Brechung

Als letzter wichtiger Effekt fehlt nun noch die Lichtberechung an zum Beispiel Glas. Nach dem Brechungsgesetz der Physik, ändert sich der Winkel des Lichtes wenn es von einem optisch dünnerem Medium (wie Luft) und ein optisch dichteres Medium (wie Glas) übergeht. Der Winkel zum Lot wird beim Übergang von Luft in Glas etwas kleiner - dabei ist das Verhältnis des Sinus von beiden Winkeln immer konstant bei etwa 1,5. Diesen Wert bezeichnet mal allgemein als Brechungszahl.





Winkel zum Einfallslot

α Winkel im Vakuum (≈Luft)
β Winkel im angrenzenden Medium

Brechungszahl n

$\frac{\sin(\alpha)}{\sin(\beta)} = n$ (Glas n≈1,5)

Tritt das Licht auf der anderen Seite wieder aus dem Glas aus, ist der Effekt gerade umgekehrt. Daher erscheinen Gegenstände durch eine Glasscheibe etwas verschoben (oben im Bild der hellgrau und der schwarz austretende Strahl).

Natürlich ist Glas nicht nur durchsichtig, sondern spiegelt auch einen Teil des Lichts. In diesem Fall werden die Berechnungen bei mehreren übereinander liegenden Glasscheiben wie im Beispiel oben schnell sehr aufwendig.

1.2.8 Raytrace_4

Die Implementierung in den Raytracer ist fast diesselbe wie bei der Spiegelung.

```
Raytrace_4(Strahl)
{
     Objekt = Suche_Objekt(Strahl);
     if (Objekt gefunden) {
          Schattenstrahl = Strahl Schnittpunkt -> Lichtquelle;
          blockiert = Suche_Objekt(Schattenstrahl);
          if (blockiert = false)
               Farbe2 = Berechne_Farbe(Strahl,Objekt,...);
          else Farbe2 = Schwarz;
          Spiegelstrahl = Reflektions_Strahl(Strahl,Objekt,...);
          Farbe3 = Raytrace_4(Spiegelstrahl) + Farbe2;
          Brechungsstrahl = Brechungs_Strahl(Strahl,
            Objekt,Schnittpunkt);
          Farbe = Raytrace_4(Brechungsstrahl) + Farbe3; }
     else Farbe = Hintergrund;
     return(Farbe);
}
```

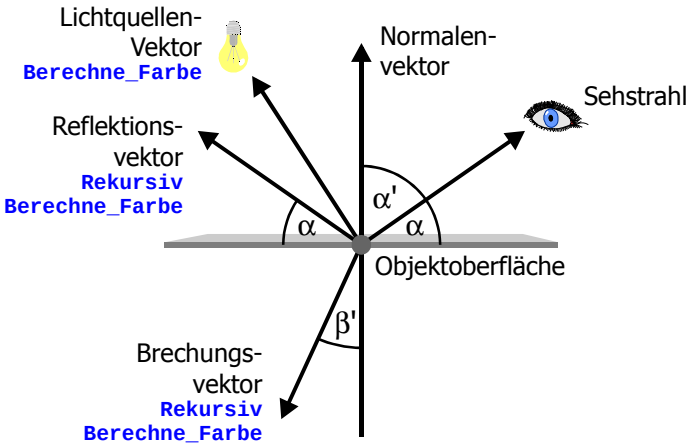
Brechungsstrahl

Nur wird hier der **Berechungsstrahl** mit Hilfe der Subroutine **Brechungs_Strahl** berechnet und damit erneut rekursiv **Raytrace_4** aufgerufen. Zusätzlich zum über die Spiegelung ermittelten Farbwert wird auch noch dieser dritte Farbwert zur endgültigen Farbe addiert.

1.3 Die Beleuchtungsformel

1.3.1 Farbberechnung mit 5 Vektoren

Die Prozedur **`Berechne_Farbe`** hat bisher die Farbberechnung des Objekts und rekursiv der Spiegelung und der Brechnung übernommen. Nun geht es darum, was diese Prozedur eigentlich macht - d.h. die Berechnung mit Hilfe der Beleuchtungsformel.



Insgesamt gibt es 5 Vektoren die an der Rechnung beteiligt sind ...

- **Sehstrahl:** Die Strahlrichtung vom Objektschnittpunkt zum Auge.
- **Normalenvektor:** Der Vektor senkrecht zur Objektoberfläche. Er wird in den folgenden Rechnungen an verschiedenen Stellen benutzt.
- **Lichtquellenvektor:** Die Strahlrichtung vom Objektschnittpunkt zur Lichtquelle. Eventuell gibt es hier auch mehrere Lichtquellen d.h. mehrere Vektoren, die untersucht werden müssen. Dieser Vektor wird direkt in **`Berechne_Farbe`** benutzt.
- **Reflektionsvektor:** Gemäß dem Reflektionsgesetz aus dem Sehstrahl berechneter Reflektionsvektor. Rekursiv wird damit **`Raytrace_4`** aufgerufen und dort dann über **`Berechne_Farbe`** der Farbwert des gespiegelten Objekts berrechnet.
- **Brechungsvektor:** Aus dem Sehstrahl mit dem Brechungsgesetz berechneter Brechungsvektor. Ebenso wie beim Reflektionsvektor ist hier **`Berechne_Farbe`** im rekursiven Aufruf enthalten.

Zur Farbberechnung muß zum Beispiel der Vektor zwischen dem Normalenvektor und dem Lichtquellenvektor betrachtet werden. Zuerst benötigt man aber noch eine rechnerinterne Farbdarstellung über Zahlenwerte.

1.3.2 Farbmodell

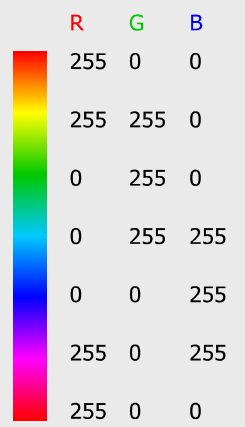
Ein typisches Modell hierfür ist das RGB-Modell. Um einen realistischen Farbeindruck zu gewährleisten ist eine genügend große Anzahl von Farben nötig. Als Minimum dafür wird normalerweise folgendes festgelegt, was den Bytes im Rechner entgegenkommt ...

- 256 Rotabstufungen (R)
- 256 Grünabstufungen (G)

- 256 Blauabstufungen (B)

Dies ergibt eine Menge von etwa $256^3 \approx 16{,}8$ Millionen verschiedenen Farben bzw. eine Farbtiefe von 24-Bit. Der Mensch kann normalerweise bis zu 7 Millionen verschiedene Farbabstufungen unterscheiden, daher werden auf diese Weise genügend Farbschattierungen abgedeckt, was beim Raytracing ja sehr wichtig ist (einige Farben, die der Mensch sieht, lassen sich aber dennoch mit den RGB-Modell nicht darstellen).

Zur Verdeutlichung hier nochmal einen kompletten Farbverlauf, der alle wichtigen Farbwerte abgedeckt (unabhängig von ihrer Helligkeit) ...



Im Raytracer selbst muß man daher für alle Rechnungen jeweils den Rot-/Grün-/Blauanteil seperat betrachten. In den nächsten Abschnitten gilt daher folgende Äquivalenz ...

$$\begin{aligned} \text{Farbe} &= \text{Farbe1} \cdot \text{Faktor} + \text{Farbe2} \\ &\Leftrightarrow \\ \text{Farbe.R} &= \text{Farbe1.R} \cdot \text{Faktor} + \text{Farbe2.R} \\ \text{Farbe.G} &= \text{Farbe1.G} \cdot \text{Faktor} + \text{Farbe2.G} \\ \text{Farbe.B} &= \text{Farbe1.B} \cdot \text{Faktor} + \text{Farbe2.B} \end{aligned}$$

1.3.3 Lichtanteile

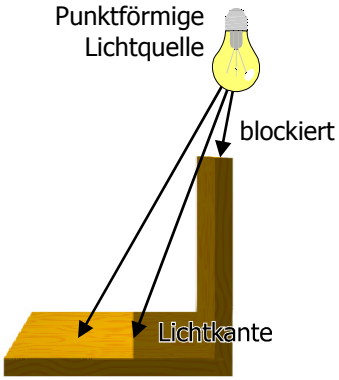
Als Modell für Licht selbst greift man auf 3 unterschiedliche Lichtarten zurück.

- **Direktes Licht:** Um zu berechnen, ob ein Objekt im Schatten liegt, wird beim Raytracing ja der Strahl vom Objekt der Lichtquelle benutzt (Lichtquellenvektor). Ist dieser nicht blockiert, so erhält der Gegenständ von der Lichtquelle direktes Licht.

- **Indirektes Licht:** Ein Gegenstand erhält zusätzlich durch Spiegelung und Brechung (also die rekursiven Raytrace-Aufrufe) indirektes Licht, das aufaddiert wird. Damit ist also das indirekte Licht eines Objekts A auf direktes Licht von Objekt B zurückzuführen (oder entsprechend rekursiv verschachtelt).

- **Umgebungslicht:** Ein großes Problem beim Raytracing ist das Streulicht. Da es hier nicht wirklich in diesem Modell simuliert werden kann, geht man von einer gewissen Grundhelligkeit aller Objekte bzw. der Szene aus. Sie ist überall gleich stark und erreicht auch im Schatten liegende Objekte.

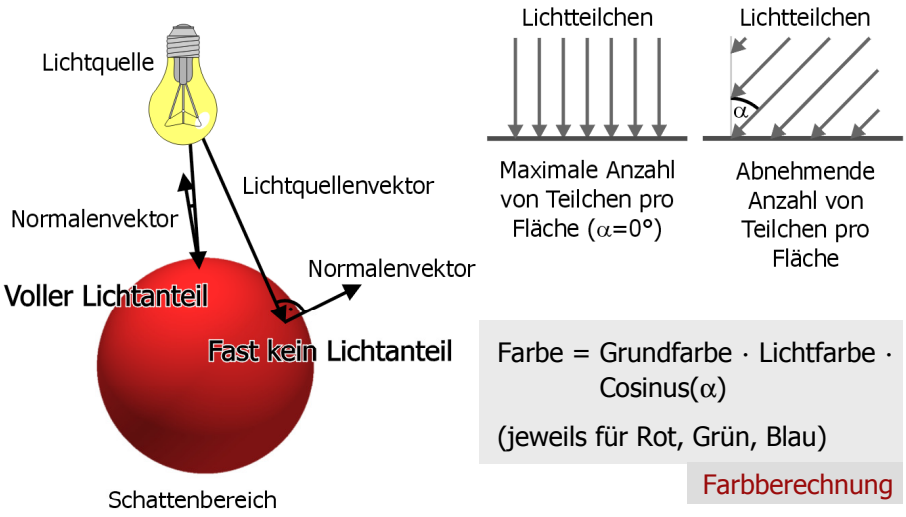
Beim Raytracing sind alle Lichtquellen praktisch punktförmig zu sehen. Entweder ist ein Teil eines Objekts definitiv im Schatten oder er erhält direktes Licht von der Lichtquelle. Daraus entsteht eine Lichtkante wie in der Grafik unten.



Bei ausgedehnter Lichtquelle
(z.B. Sonnenlicht) weicher
Übergang zum Schatten

In Realität sind aber die meisten Lichtquellen nicht punktförmig sondern ausgedehnt. Das einfachste Beispiel hierfür ist das Sonnenlicht in einem Raum: obwohl ein Raum ja nicht vollständig mit direktem Licht durch die Fenster erhellt wird, ist es dennoch im kompletten Raum hell. Die Sonne selbst ist zwar von der Erde aus auch fast eine punktförmige Lichtquelle, jedoch streuen die Gegenstände im Raum das Licht diffus in alle Richtungen und wirken daher wie ausgedehnte indirekte Lichtquellen. Dieses Licht füllt den ganzen Raum mit Streulicht und verursacht weiche Schattenübergänge und eine gewisse Grundhelligkeit an den meisten Orten. Dies kann mit dem Raytracing-Verfahren nicht wirklich simuliert werden, die Umgebungshelligkeit ist hierfür nur ein Ersatz. Ausgedehnte Lichtquellen lassen sich höchstens durch sehr viele punktförmige Lichtquellen in etwa nachahmen, was aber natürlich schnell viel zusätzliche Rechenzeit benötigt, falls die Übergänge fließend sein sollen.

1.3.4 Direktes Licht



Nun hat man das Vorwissen, das man für **`Berechne_Farbe`** benötigt. Dabei handelt es sich nämlich um direktes Licht und dieses ist abhängig vom einfallenden Winkel: Ist der Lichtquellenvektor parallel zum Normalenvektor, so erhält der jeweilige Punkt der Kugeloberfläche den vollen Lichtanteil. In diesem Fall treffen nämlich auf eine vorgegebene Fläche die maximale Anzahl von Lichtteilchen bzw. Photonen auf. Je weiter man nach außen auf der Kugel geht, desto mehr steigt der Winkel α zwischen Normalvektor und Lichtquellenvektor an. Da die Lichtteilchen nun in einem steileren Winkel einfallen, nimmt ihre Anzahl pro Fläche ab. Dies ist oben rechts in der Grafik illustriert, wobei hier einfachtsheitshalber angenommen wird, daß für die kleine Umgebung um den entsprechenden Punkt die Lichtstrahlen parallel einfallen. Genau dasselbe Prinzip ist der Grund dafür, daß es am Äquator am wärmsten ist und die Temperatur an den Polen stark abfällt.

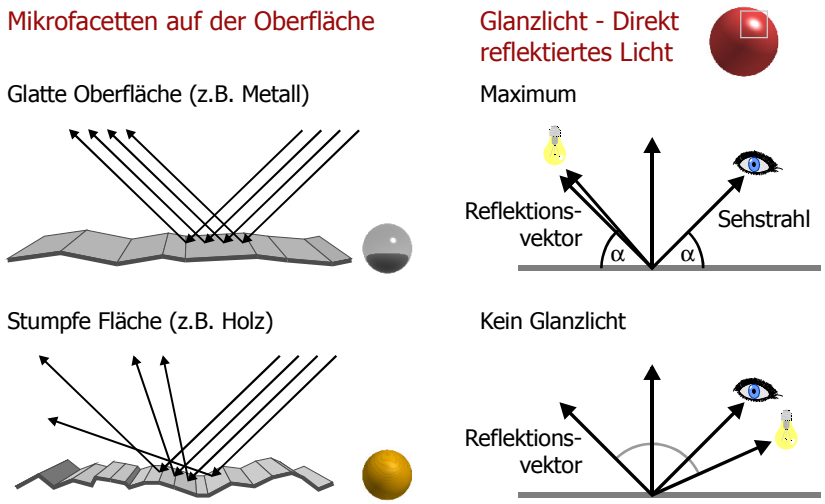
In Formeln ausgedrückt multipliziert man zuerst die Grundfarbe des Objektes mit der Lichtfarbe. Dabei ist die Lichtfarbe als ein Faktor zu sehen, der z.B. die Grundfarbe verdoppelt, um einen Gegenstand heller erscheinen zu lassen oder um nur den blauen Anteil eines Gegenstandes herauszufiltern (bei blauem Licht). Außerdem geht als zweiter Faktor der Cosinus des obigen Winkels α ein.

Auf diese Weise kann man einer punkförmigen Lichtquelle hier noch zusätzliche Eigenschaften geben, wie daß das Licht abhängig von der Entfernung zur Lichtquelle langsam dunkler wird. Auch kann ein Licht nur in eine bestimmte Richtung abstrahlen, wie z.B. eine Taschenlampe, d.h. innerhalb eines bestimmten Winkel. Dadurch ergibt sich dann ein runder Lichtkegel, den man auch am Rand abhängig vom Winkel von Schwarz zur Lichtfarbe abstufen kann. Dieser Effekt kann leicht mit Streulicht verwechselt werden, funktioniert aber nur bei der Lichtquelle selber und nicht bei den Schatten, die durch sie entstehen.

Wie im obigen Raytracer beschrieben, wird nun noch das indirekte Licht auf diesselbe Weise rekursiv berechnet und das Umgebungslicht einfach addiert.

1.3.5 Glanzlicht

Bisher fehlte noch eine wichtige Eigenschaft, die man für die Ermittlung der Oberflächenfarbe benötigt: das Glanzlicht. Bei einer punktförmigen Lichtquelle hat ein Gegenstand einen hellsten Punkt, an dem das Licht von der Lichtquelle praktisch direkt reflektiert wird. Nicht bei jeder Oberfläche hat das Glanzlicht diesselben Eigenschaften, da diese je nach Material im Mikrometer-Bereich varriert, auch wenn der Gegenstand mit dem menschlichen Auge glatt erscheint ...

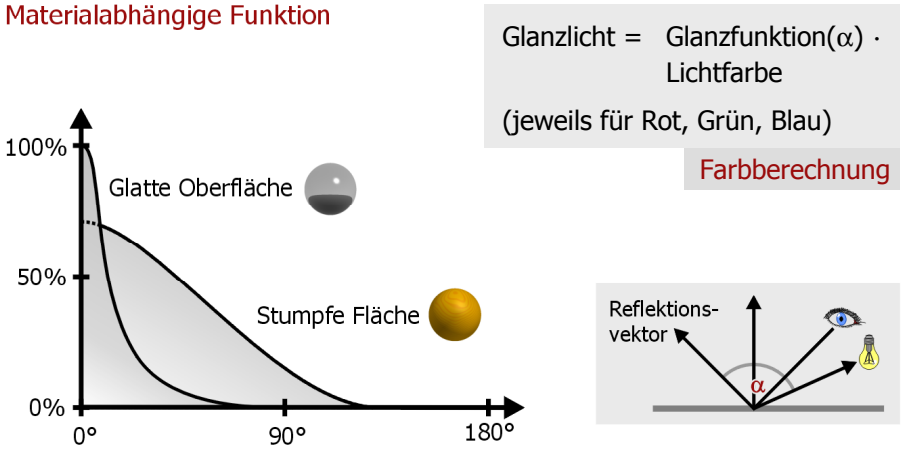


Bei z.B. metallischen Oberflächen sind diese Mikrofacetten über größere Bereiche eben und ausgeglichen. Dadurch werden die meisten Lichtstrahlen der Quelle wie im Bild oben direkt reflektiert. Damit erscheint das Glanzlicht - also die gespiegelte Lichtquelle - fast punkförmig und praktisch weiß. Bei stumpfen Gegenständen dagegen wird das einfallende Licht in viele unterschiedliche Richtungen gestreut. Dadurch ist das Glanzlicht weniger ausgeprägt, aber dafür über eine größere Fläche verteilt.

Im Rechner kann dies natürlich nicht simuliert werden, da viel zu viele Facetten nötig wären. Dafür läßt sich das Glanzlicht in Abhängigkeit zwischen dem Reflektionsvektor und dem Lichtquellenvektor betrachten. Ist der Reflektionsvektor direkt auf die Lichtquelle gerichtet, ist das Glanzlicht am stärksten. Bei einem Winkel über 90° zwischen den Vektoren ist der Glanzlichtanteil normalerweise 0.

1.3.6 Glanzlichtfunktion

Auf Grund des gegebenen Winkels könnte man natürlich auch das Glanzlicht mit Hilfe des Cosinus berechnen. Allerdings würde das nur für ein sehr weiches Glanzlicht funktionieren. Daher ist es am sinnvollsten, man definiert für jede Oberfläche eine eigene materialabhängige Glanzlichtfunktion.



Bei glatten Oberflächen ist diese zu Beginn maximal und fällt dann schnell ab. Bei stumpfen Oberflächen beginnt die Kurve bei einem niedrigeren Wert, aber fällt dafür viel langsamer ab. Spätestens bei 180° ist das Glanzlicht aller Funktionen 0. D.h. es wird vom Benutzer eine Glanzlichtfunktion zu jedem Material vorgegeben.

Das Glanzlicht berechnet sich dann einfach über Glanzfunktion(α)·Lichtfarbe und da die meisten Lichtquellen weiß sind, erscheint auch das Glanzlicht normalerweise als weißer Punkt. Da es ein Teil des direkten Licht eines Gegenstandes ist, wird es zur letzten Farbberechnungsformel addiert.

1.3.7 Gesamtformel

Nun hat man alle Bestandteile, die man für die Berechnung der eigentlichen Farbe des Gegenstandes benötigt. Daher hier noch einmal die Zusammenstellung der einzelnen Anteile.

Lichtbestandteile		
Direktes Licht	DFarbe =	Grundfarbe · Lichtfarbe · Cosinus + Glanzfunktion · Lichtfarbe
Reflektiertes Licht	SFarbe =	`Raytrace_4(Spiegelstrahl);`
Gebrochenes Licht	BFarbe =	`Raytrace_4(Brechungsstrahl);`
Umgebungslicht	UFarbe =	Grundfarbe · Umgebungslicht
Aufteilung		
Gesamtfarbe	Farbe =	DFaktor · DFarbe + SFaktor · SFarbe + BFaktor · BFarbe + UFarbe

Zur näheren Erläuterung ...

- **Direktes Licht:** Wie auf den letzten Abschnitten erläutert, setzt sich das direkte Licht aus der Grundfarbe in Abhängig von der Lichtquelle und dem Glanzlicht zusammen.
- **Reflektiertes Licht:** Reflektiertes Licht wird rekursiv berechnet, da es ja nur direktes Licht von anderen Objekten ist.
- **Gebrochenes Licht:** Genauso verhält es sich hier, ein rekursiver Aufruf erledigt die nötige Arbeit.
- **Umgebungslicht:** Das Umgebungslicht ist überall gleich stark, daher muß es nur mit der Grundfarbe des Gegenstandes multipliziert werden. Ein Gegenstand behält ja auch im Schatten seine Farbe, auch wenn sie nur noch schwach sichtbar ist.

Alle einzelne Berechnungen werden jeweils für den Rot/Grün/Blau-Anteil durchgeführt und können nun zu der Gesamtfarbe addiert werden. Da je nach Material direktes Licht und Licht von Spiegelung und Brechung unter stark zur Geltung kommen, kann man hier noch zusätzlich Faktoren festlegen, die die einzelnen Anteile verschieden gewichten.

Manche Gegenstände reflektieren überhaupt nicht oder sind nicht durchsichtig. Der Raytracer berücksichtigt dies nicht nur bei der obigen Rechnunge, sondern erspart sich natürlich, wenn möglich, schon die Berechnung des entsprechenden Spiegel- und Brechungsstrahls.

Ebenso kann man bei einem Raytracer noch zusätzlich einen eigenleuchtenden Anteil eines Gegenstandes angeben, wodurch er wie eine Lichtquelle auch im Schatten leuchtet. Allerdings betrifft dies nur den Gegenstand selbst, da der Raytracer ja keine flächigen Lichtquellen berechnen könnte, wie bereits erklärt wurde.

1.3.8 Verschiedene Materialien

In jedem Raytracer werden die Oberflächeneigenschaften etwas anders definiert oder bezeichnet, auch wenn das Prinzip natürlich gleich bleibt. Im Beispiel von CorelDREAM 3D sind hier einige Materialien dargestellt.

	Metall	Holz	Stein	Glas	Plastik
					
Grundfarbe					
Glanzlicht Stärke	85%	15%	50%	85%	75%
Glanzlicht Fokus	90%	10%	50%	90%	50%
Reflexion	60%	0%	0%	30%	0%
Transparenz	0%	0%	0%	85%	0%
Brechung	0%	0%	0%	24%	0%

Zuerst zu den verwendeten Begriffen ...

- **Glanzlicht Stärke/Fokus:** Die Glanzlicht Stärke und der Fokus definieren hier die Glanzlichtfunktion. Die Glanzlicht Stärke bestimmt die Intensität des Glanzlichtes und der Glanzlicht Fokus entscheidet darüber, ob das Glanzlicht über einen größeren Bereich verteilt ist oder an einer Stelle konzentriert.

- **Reflexion:** Dies entspricht dem Spiegel-Faktor (SFaktor) aus dem letzten Abschnitt. Je höher der Reflexions-Prozentsatz ist, desto mehr wird die Farbe des Gegenstandes durch die reflektierte Farbe bestimmt.

- **Transparenz:** Die Transparenz oder der Brechungs-Faktor geben an, wie durchsichtig ein Gegenstand ist.

- **Brechung:** Dies ersetzt die Angabe der materialabhängigen Brechungszahl der Physik - d.h. wie stark der Winkel bei einer Brechung verändert wird.

Nun zu den einzelnen Materialien ...

- **Metall:** Wie in Realität hat Metall hier ein sehr ausgeprägtes Glanzlicht. In diesem Fall handelt es sich um geschliffenes Metall, da der Reflektionsfaktor relativ hoch ist.

- **Holz:** Holz reflektiert nicht und ist auch nicht durchsichtig - auch das Glanzlicht ist relativ matt. Im wesentlichen wird es durch seine Textur bestimmt, d.h. statt einer Oberflächenfarbe wird ein Bitmap oder eine mathematische Formel verwendet, die die Beschaffenheit der Oberfläche nachgeahmt.

- **Stein:** Ebenso ist es mit geschliffenem Stein, allerdings ist hier das Glanzlicht wieder ausgeprägter. Je nach Steinart kann man hier natürlich auch einen Reflektionsanteil setzen.

- **Glas:** Glas ist am komplexesten in einem Raytracer darzustellen. Große Mengen davon vergrößern die benötige Rechenzeit sehr schnell. Neben einem ausgesprägten Glanzlicht reflektiert Glas einen guten Teil des Lichtes und ist natürlich auch transparent. Daher müssen hier alle möglichen Strahlen überprüft werden.

- **Plastik:** Plastik ist dagegen am einfachsten im Raytracer darzustellen. Normalerweise ist es einfarbig und benötigt daher keine Textur und außerdem kann man evtl. noch auf

den Reflektionsanteil verzichten, je nach dem Art des Materials. Außer dem direkten Licht muß hier nichts berechnet werden.

Auf diese Weise lassen sich fast alle realen Materialen gut simulieren. Ein Nachteil bleibt aber: Da normalerweise auf sehr kleine Objekte und Abweichungen der Oberfläche im Raytracer verzichtet wird, um die nötigen Objektanzahl überschaulich zu halten, erscheinen die meisten Flächen relativ künstlich, da ihr reale Unebenheiten fehlen. Hier muß daher nach einer möglichst guten Textur gesucht werden, die ein Material wesentlich realistischer erscheinen lassen kann.

1.4 Suche_Objekt

1.4.1 Die Prozedur

Ein wichtiger Teil des Raytracer wurde aber bisher noch nicht im Detail besprochen: die Prozedur **`Suche_Objekt`**. Sie muß unter Angabe eines Strahls feststellen, ob dieser ein Objekt schneidet und wenn ja, welches davon zuerst geschnitten wird. D.h. es geht hier um den eigentlichen Kern des Raytracers, dem auch die meiste Rechenzeit zufällt.

```
Suche_Objekt(Strahl)
{
    Schnittpunkt = Strahl verlängert ins Unendliche;
    Objekt = false;
    for (alle Objekte i=0...n) {
        s2 = Strahl_trifft_Objekt(Strahl,Objekte[i]);
        if (s2 existiert und s2 näher am Startpunkt
            als Schnittpunkt) {
            Objekt = Objekte[i];
            Schnittpunkt = s2; } }
    return(Objekt,Schnittpunkt);
}
```

Schnittpunkt-Test

Auch wie der Raytracer selbst ist die Ausformlierung zunächst relativ einfach. Ermittelt werden soll am Ende ein getroffenes **`Objekt`** und der **`Schnittpunkt`**. Für den Fall daß kein Objekt getroffen wird, wird der **`Schnittpunkt`** zu Beginn ins Unendliche verschoben, damit alle weiteren möglichen Schnittpunkte davor liegen (unendlich ist hier natürlich ein maximaler Wert, der angenommen werden kann), und das gefundene **`Objekt`** wird mit **`false`** initialisiert.

Danach müssen nun alle **`Objekte`** getestet werden. Die eigentliche Berechnung des neuen Schnittpunkts übernimmt hier die Funktion **`Strahl_trifft_Objekt`** (mehr dazu im nächsten Abschnitt). Sie testet, ob der Strahl das aktuelle Objekt **`Objekte[i]`** in der Schleife schneidet und gibt den **`Schnittpunkt`** zurück. Existiert dieser Schnittpunkt und liegt er näher am Startpunkt als der vorherige Wert, wird dieses **`Objekt`** und der neue **`Schnittpunkt`** zurückgegeben.

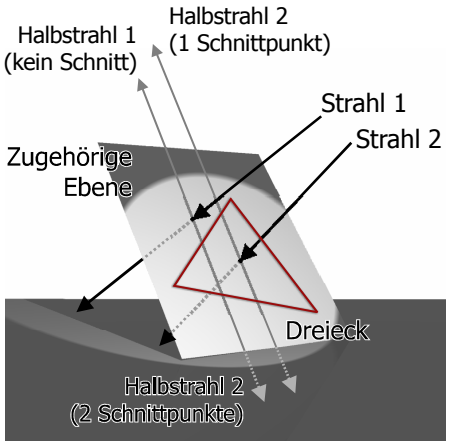
1.4.2 Berechnung des Schnittpunktes

Die Prozedur **`Strahl_trifft_Objekt`** deckt verschiedene Arten von Schnittpunktsberechnungen ab: ein Strahl kann mit einer Kugel geschnitten werden, mit einem Würfel

oder mit einem Objekt, das aus verschiedenen Formen zusammengesetzt ist. Daher kann die Berechnung auch mathematisch aufwendiger werden.

Im folgenden wird das einfachste Beispiel behandelt: ein Strahl schneidet ein Dreeick. Da aber alle Objekte durch Dreiecke angenähert werden können, ist damit auch die Berechnung von komplexeren Oberflächen möglich (auch wenn ein Vorteil des Raytracings ist, daß z.B. eine Kugel nur durch ihre mathematische Repräsentation im Rechner gespeichert werden kann und sie nicht unbedingt zerlegt werden muß, wie bei manchen anderen Verfahren).

Beispiel: Strahl schneidet Dreieck



Vektorrechnung

Strahl entspricht Geradengleichung

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} Start_x \\ Start_y \\ Start_z \end{pmatrix} + t \cdot \begin{pmatrix} Richtung_x \\ Richtung_y \\ Richtung_z \end{pmatrix}$$

Dreieck in Ebene

$$a \cdot x + b \cdot y + c \cdot z - d = 0$$

Einsetzen und nach t auflösen

$$t = \frac{d - a \cdot Start_x - b \cdot Start_y - c \cdot Start_z}{a \cdot Richtung_x + b \cdot Richtung_y + c \cdot Richtung_z}$$

$$\begin{pmatrix} Schnitt_x \\ Schnitt_y \\ Schnitt_z \end{pmatrix} = \begin{pmatrix} Start_x \\ Start_y \\ Start_z \end{pmatrix} + t \cdot \begin{pmatrix} Richtung_x \\ Richtung_y \\ Richtung_z \end{pmatrix}$$

Ein Strahl (also z.B. ein Sehstrahl) ist definiert durch einen Startpunkt und seine Richtung mit jeweils drei Koordinaten. Daher entspricht er im mathematischen Sinne einer Geradengleichung ...

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} Start_x \\ Start_y \\ Start_z \end{pmatrix} + t \cdot \begin{pmatrix} Richtung_x \\ Richtung_y \\ Richtung_z \end{pmatrix}$$

Ein Dreieck dagegen ist durch drei Punkte festgelegt, die in einer Ebene liegen ...

$$a \cdot x + b \cdot y + c \cdot z - d = 0$$

Für die Berechnung des Schnittpunktes muß man nun einfach die einzelnen Koordinaten der Geradengleichung in die Ebenengleichung einsetzen und nach t auflösen ...

$$a \cdot (Start_x + t \cdot Richtung_x) + b \cdot (Start_y + t \cdot Richtung_y) + c \cdot (Start_z + t \cdot Richtung_z) - d = 0$$

$$\Leftrightarrow t = \frac{- a \cdot Start_x - b \cdot Start_y - c \cdot Start_z + d}{a \cdot Richtung_x + b \cdot Richtung_y + c \cdot Richtung_z}$$

An dieser Stelle kann der Bruch des Terms auch 0 werden, falls die Ebene und die Gerade parallel verlaufen - dies wird zusätzlich abgeprüft. Ansonsten ergibt sich der Schnittpunkt nun direkt durch einsetzen von t ...

$$\begin{pmatrix} Schnitt_x \\ Schnitt_y \\ Schnitt_z \end{pmatrix} = \begin{pmatrix} Start_x \\ Start_y \\ Start_z \end{pmatrix} + \left(\frac{d - a \cdot Start_x - b \cdot Start_y - c \cdot Start_z}{a \cdot Richtung_x + b \cdot Richtung_y + c \cdot Richtung_z}\right) \cdot \begin{pmatrix} Richtung_x \\ Richtung_y \\ Richtung_z \end{pmatrix}$$

D.h. t legt fest, wie weit der Schnittpunkt vom Startpunkt entfernt ist. Damit kann auch gleich festgestellt werden, welches Objekt als erstes vom Startpunkt aus geschnitten wird

(dazu müßte von **Strahl_trifft_Objekt** oben noch t zurückgegeben werden). Allerdings, falls noch andere Arten von Schnittpunktsberechnungen verwendet werden, muß der Abstand Schnittpunkt/Startpunkt separat berechnet werden.

Um nun noch festzustellen, daß der Schnittpunkt auch innerhalb des Dreiecks liegt oder nicht, wird z.B. Halbstrahl-Verfahren angewendet: Vom Schnittpunkt aus wird innerhalb der Dreiecksebene ein Strahl in eine beliebige Richtung geschickt. Schneidet dieser keine oder zwei Kanten des Dreiecks (wie Strahl 1 im obigen Beispiel), liegt der Schnittpunkt außerhalb - bei nur einem Schnitt entsprechend innerhalb (Strahl 2). Bei Vielecken liegt ein Punkt innerhalb, wenn eine ungerade Anzahl von Kanten geschnitten wird.

1.4.3 Aufwand

Wie bereits erwähnt, fällt an **Suche_Objekt** die meiste Rechenzeit ab, da es die innere Schleife darstellt. Und diese Prozedur ist auch mit ein Grund dafür, weshalb Raytracing auch heute noch für Echtzeit-Berechnungen nicht wirklich geeignet ist und eher dann verwendet werden, wenn es um qualitativ gute und realistische Ergebnisse geht.

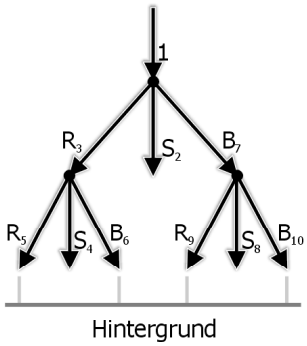
Von **Raytrace_1** bis **Raytrace_4** sind die Prozeduren des Raytracers jeweils komplexer geworden. Damit stieg auch kontinuierlich die Anzahl der Strahlen, die verfolgt werden müssen.

Anzahl der Aufrufe von Suche_Objekt

	Strahlen pro Pixel (bei getroffenem Objekt)
Raytrace_1	1
Raytrace_2	≥2 (Anzahl Lichtquellen)
Raytrace_3	≥3 (Anzahl Reflexionen)
Raytrace_4	≥4 (Anzahl Brechungen)

- Minimal: $1024 \cdot 768 \cdot 4 \approx 3$ Millionen
- Beispiel: $1024 \cdot 768 \cdot 10 \approx 8$ Millionen
- 1000 Objekte: $1024 \cdot 768 \cdot 10 \cdot 1000 \approx 8$ Milliarden
- Animation: $1024 \cdot 768 \cdot 10 \cdot 1000 \cdot 25 \approx 197$ Milliarden pro s

Aufrufbaum



S = Schattenstrahl
R = Reflexionsstrahl
B = Brechungsstrahl

Wie oben im linken Teil des Bildes dargestellt, benötigt **Raytrace_1** 1 Strahl pro Pixel (Sehstrahl), **Raytrace_2** mindestens 2 Strahlen (Sehstrahl und die Strahlen zu den einzelnen Lichtquellen, vorausgesetzt es wird nicht nur der Hintergrund getroffen), **Raytrace_3** mindestens 3 (die Anzahl der Reflexionen bei gesetztem Reflexionsfaktor) und **Raytrace_4** mindestens 4 (die Anzahl der Brechungen, bei gesetztem Brechungsfaktor). 4 Strahlen sind alleine natürlich schnell zu berechnen, allerdings zeigen die obigen Beispielewerte, daß die wenigen Strahlen noch mit mehreren Faktoren multipliziert werden müssen. Immerhin ergeben sich bereits 3 Millionen bei einer Standard-Auflösung von 1024·768 (wobei der wirkliche Minimalwert hier bei 1024·768·1 = 800 000 liegt, wenn die Szene nur aus dem Hintergrund besteht).

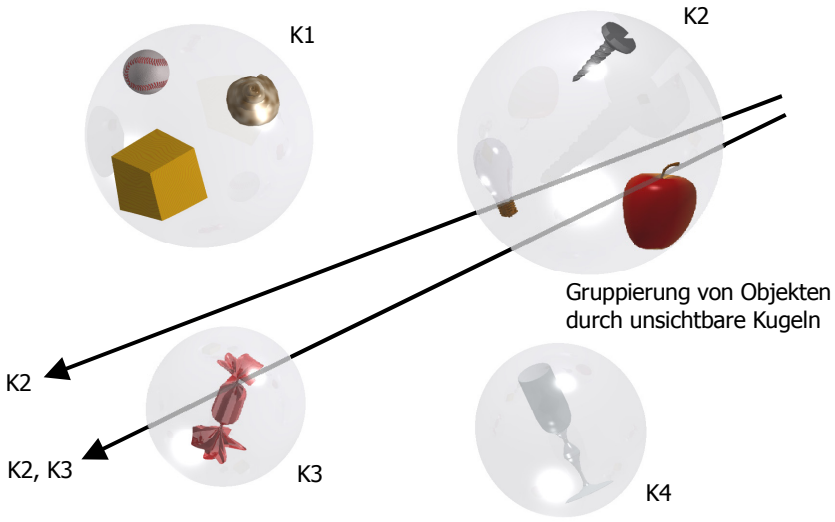
Angenommen es werden im Durchschnitt 10 Strahlen verfolgt, dann ergeben sich 8 Millionen Strahlen. Zum Beispiel bei einer Szene mit Reflexionen und Brechungen könnte dies

ein Mittelwert sein. Der Aufrufbaum ist im Bild oben rechts dargestellt, jeder Pfeil steht für einen entsprechenden Strahl. Es wird hier also angenommen, daß jeder Reflexions- und Brechungsstrahl noch auf ein Objekt trifft und dann auf den Hintergrund. Nun kommt aber noch die Anzahl der Objekte hinzu: im Beispiel sind es 1000 Objekte, was keine sehr große Anzahl ist, da komplizierte Objekte ja über viele einfachere mathematische Formen aufgebaut werden müssen. Dann ergeben sich bereits 8 Milliarden. Will man nun noch eine Animation der Szene erstellen, müssen pro Sekunde ca. 197 Milliarden Strahlen berechnet und auf Schnitte mit dem Objekten getestet werden. Auch wenn der entsprechende Programmcode sicherlich optimiert werden kann, ist dies auch heutzutage noch zuviel, da auch Szenen mit weit mehr als 1000 Objekten berechnet werden wollen und der Aufwand proportional damit ansteigen würde. Dies kann aber durch einige Veränderungen am Raytracer deutlich reduziert werden.

1.4.4 Reduzierung der Schnittests

Ob ein Reflexions- oder Brechungsstrahl losgeschickt werden muß, ist bereits vorher festgelegt durch den Reflexions- und Brechungsfaktor und die Rekursionstiefe, d.h. wie oft Reflexion/Brechung wiederholt werden. Damit ist die Anzahl der Strahlen pro Pixel fest, ebenso die Bildschirmauflösung und die Bilder pro Sekunde. Diese Werte kann der Benutzer je nach gewünschter Qualität einstellen. Variabel ist also nur noch die Anzahl der Objekte, die bei jedem Schritt geprüft werden müssen.

Anstatt für jeden Durchlauf jedes Objekt mit dem Strahl zu testen, kann man die Objekte zum Beispiel mit Kugeln zusammenfassen, die einen Teil der Objekte vollständig enthalten - wie im Bild unten.



Da Objekte ja normalerweise nicht beliebig im Raum verteilt sind, sondern sich an bestimmten Stellen konzentrieren, ist dies meist möglich (im Bild sind die Objekte nicht wirklich realistisch verteilt, aber das Prinzip läßt sich auch auf übliche Szenen anwenden). Der obere Strahl wird auf einen Schnitt mit K1 bis K4 geprüft und da nur K2 geschnitten wird, müssen nur hier die 3 inneren Objekte einzeln getestet werden. Beim zweiten Strahl entsprechend Kugel 2 und 3. Der Schnitt mit den Kugeln selbst ist natürlich viel einfacher zu testen, als mit den inneren Objekten die wieder aus mehreren Formen zusammengesetzt sind.

Die Kugeln selbst können auf diesselbe Weise gespeichert und getestet werden, wie normale Kugelobjekte. Da sie aber nicht dargestellt werden sollen und noch Unterobjekte enthalten,

benötigt man dafür eine eigene Datenstruktur und **Suche_Objekt** wird entsprechend angepaßt.

1.4.5 Suche_Objekt_2

Die Initialisierung der Daten und die inneren Schleife können unverändert übernommen werden. Zusätzlich wird eine äußere Schleife hinzugefügt ...

```
Suche_Objekt_2(Strahl)
{
     Schnittpunkt = Strahl verlängert ins Unendliche;
     Objekt = false;
     for (Teste Strahl mit allen Umgrenzungskugeln)
          {
          if (Strahl schneidet Umgrenzungskugel) {
               for (alle Objekte i innerhalb) {          K1...M
                    s2 = Strahl_trifft_Objekt(Strahl,Objekte[i]);
                    if (s2 existiert und s2 näher am Startpunkt
                         als Schnittpunkt) {
                         Objekt=Objekte[i]; Schnittpunkt=s2; } } }
          }
     return(Objekt,Schnittpunkt);
}
```

Anstatt mit allen Objekten wird der Strahl zuerst mit allen Umgrenzungskugeln getestet und nur falls eine Kugel geschnitten wurde, wird die innere Schleife ausgeführt und alle Objekte innerhalb getestet.

Natürlich muß sich dieses Verfahren nicht nur auf Umgrenzungskugeln beschränken. Da die Objekte wie die übrigen Objekte im Raytracer aufgebaut sein können, sind natürlich auch Umgrenzungswürfel oder sonstige Umgrenzungspolyeder möglich. Dabei muß natürlich darauf geachtet werden, daß die äußeren Schnittests nicht zu aufwendig werden, da sie für jedes Pixel ausgeführt werden müssen.

1.4.6 Effizienz

Dieses Verfahren ist aber noch nicht wirklich effizient. Hier einige Vor- und Nachteile ...

Vorteil

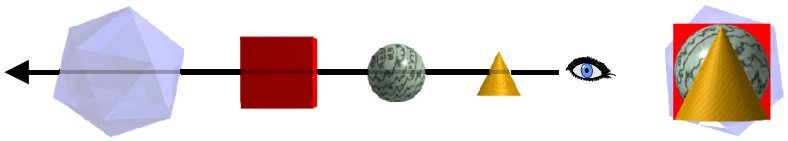
- Ein Objekt, das den Hintergrund trifft, muß nur noch mit allen Kugeln getestet werden und durchläuft daher sehr wenige Schnittests.

Nachteile

- Damit der obige Vorteil sich dagegen nicht ungünstig auswirkt, muß die Anzahl der Umgrenzungskörper gering bleiben.

- Außerdem müssen sich die Objekte überhaupt zusammenfassen lassen. Oft liegen die Objekte ungünstig, wie ein Szene innerhalb eines geschlossenen Raumes, der selbst von einer Kugel umschlossen werden müßte und damit im Inneren wieder alle Objekte getestet werden müssen. Oder auch wenn die Objekte wie im Beispiel unten hintereinander liegen. Egal wie man die Objekte hier zusammenfaßt (zum Beispiel jeweils zwei mit zwei Umgrenzungskugeln), da der Sehstrahl alle Objekte schneidet, fallen keine Schnit-

tests weg und anstatt Zeit zu sparen, machen die zusätzlichen Umgrenzungskugeln die Szene nur aufwendiger.

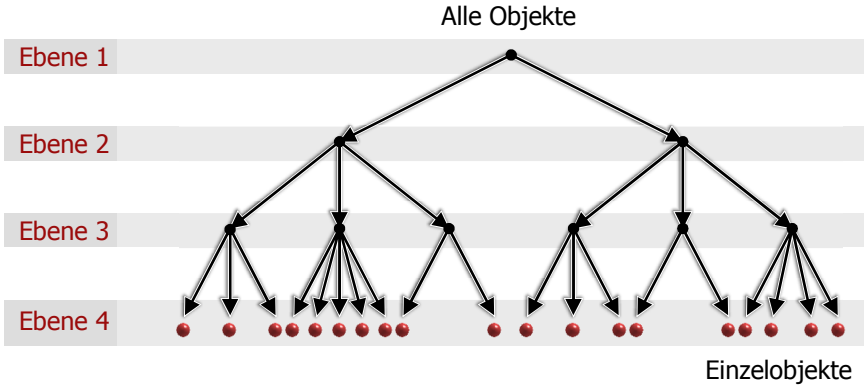


- Insgesamt gesehen funktioniert dieses Verfahren nur bei einer kleineren bis mittleren Objektanzahl. Bei zu vielen Objekten, benötigt man zuviele oder zu komplexe Umgrenzungskörper.

1.4.7 Umgrenzungskörper-Hierarchie

Diese Nachteile können zum Teil aber umgegangen werden. Anstatt nur eine Ebene von Umgrenzungskörpern zu erstellen, kann man mehrere Ebenen hierarchisch anordnen und daraus einen Baum erstellen ...

Erstellung eines Baums aus Umgrenzungskörpern



Ebene 1 besteht nur aus einem Objekt, das alle anderen Objekte enthält (damit wäre das Problem der geschlossenen Szenen von oben gelöst). Die einzelnen Ebenen werden nach unten immer feiner, bis die letzte Ebene wieder die einzelnen Objekte enthält.

Nun muß die Anzahl der Umgrenzungskörper nicht mehr insgesamt gering bleiben, sondern nur noch die Anzahl der Nachfolger eines jeden Knoten. Dadurch kann man auch viele Objekte sinnvoll aufteilen.

1.4.8 Suche_Objekt_3

```
Suche_Objekt_3(Strahl,Umgrenzungskörper)
{
    Schnittpunkt = Strahl verlängert ins Unendliche;
    Objekt = false;
    if (Strahl schneidet Umgrenzungskörper) {
        if (Enthält weitere Umgrenzungskörper) {
            for (alle enthaltenen Unterkörper) {
                (o,s)=Suche_Objekt_3(Strahl,Unterkörper);
                if (s existiert und näher am Startpunkt)
                    Objekt=o; Schnittpunkt=s;
        else {                                          Rekursion
            for (alle Objekte i innerhalb) {...} } }
    return(Objekt,Schnittpunkt);
}
```

Suche_Objekt_3 hat nun obige Form. Als Eingabeparameter benötigt es nun zusätzlich zum Strahl auch noch einen Umgrenzungskörper. Beim ersten Aufruf handelt es sich dabei um das Objekt der ersten Ebene, das alle anderen Objekte enthält. Falls der Strahl diesen Umgrenzungskörper schneidet, müssen nun zwei Fälle unterschieden werden: Entweder enthält er selbst wieder weitere Umgrenzungskörper einer niedrigeren Ebene (Unterkörper) oder nur normale Objekte - in letzterem Fall werden wieder alle Objekte innerhalb wie bei den Vorgängerversionen überprüft. Der erste Fall ist am einfachsten rekursiv zu lösen, d.h. daß für jeden Unterkörper bzw. für jeden Nachfolgerknoten wird **Suche_Objekt_3** erneut aufgerufen. Ist der so gefundene Schnittpunkt näher am Startpunkt als der vorherige, wird das neu gefundene Objekt und der neue Schnittpunkt zurückgegeben.

1.4.9 Effizienz

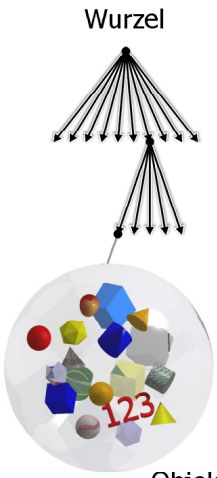
Nun zur Effizienz des neuen Verfahrens ...

Geschwindigkeitsgewinn

Beispiel

Ebene 1	1 Umgrenzungskörper mit 10 Unterkörpern
Ebene 2	10 Umgrenzungskörper mit 5 Unterkörpern
Ebene 3	50 Umgrenzungskörper mit 20 Objekten
Ebene 4	1 · 10 · 5 · 20 = 1000 Objekte

Schnittests : 1+10+5 = 16 mit Umgrenzungskörpern
20 mit Objekten ⇒ 36 statt 1000



Verbleibende Probleme

- Schnittests mit allen Objekten auf einer Ebene
- Erstellung eines optimalen Baums kompliziert und zeitaufwendig

Am einfachsten läßt Geschwindigkeitsgewinn an einem Beispiel betrachten, wie in der obigen Grafik rechts. Eine Szene wird hierarchisch in 4 Ebenen eingeteilt ...

- Ebene 1 besteht aus 1 Umgrenzungskörper (also alle Objekte) und dieser umschließt 10 weitere Unterkörper
- Ebene 2 besteht damit aus diesen 10 Umgrenzungskörpern, die jeweils 5 weitere Unterkörper enthalten
- Ebene 3 hat folglich $5 \cdot 10 = 50$ Umgrenzungskörper, die letztlich die einzelnen Objekte enthalten - jeweils 20
- Ebene 4 sind die Objekte selbst, wie im anderen Beispiel $1 \cdot 10 \cdot 5 \cdot 20 = 1000$ Stück

Am Baum daneben kann man erkennen, wie viele Schnittests nun letztendlich nötig sind (jeder Strahl und jedes Objekt steht jeweils für einen Schnittest): 1+10+5 = 16 mit den Umgrenzungskörpern der Ebene 1 bis 3 und noch 20 mit Objekten auf Ebene 4. Dann ergibt sich eine Ersparnis von immerhin 96% ($1-\frac{36}{1000} = 0{,}96$).

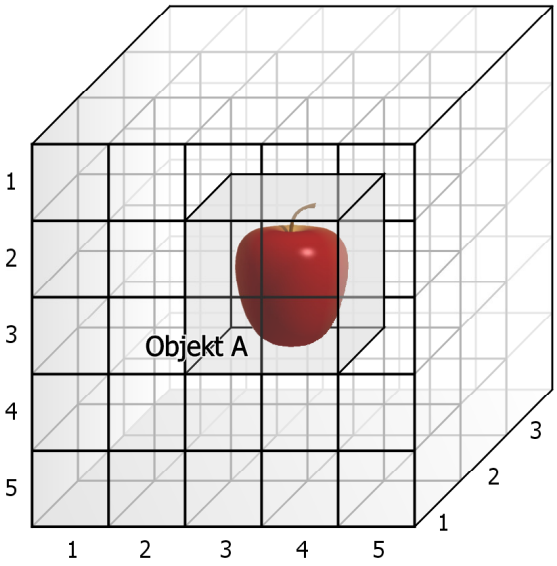
Allerdings ist dies auch der Maximalwert für die Einsparung (abgesehen von Strahlen, die auf den Hintergrund treffen). Schneidet der Strahl nämlich jeweils mehrere Umgrenzungskörpern auf einer Ebene - was ja relativ wahrscheinlich ist - müßten bei der obigen Version von **`Suche_Objekt_3`** auch jeweils diese getestet werden. Man kann dies aber noch optimieren, da sobald der Strahl innerhalb der Rekursion einen wirklichen Schnittpunkt findet, dann muß nicht mehr das Innere von Umgrenzungskugeln getestet werden, die weiter weg liegen, da dort der Strahl ja bereits blockiert wäre. Trifft der Strahl in der Umgrenzungskugel auf kein Objekt, müssen die anderen dennoch geprüft werden, da der Strahl z.B. durch alle Kugeln hindurch den Hintergrund treffen könnte.

Verbleibende Probleme

Doch auch dieses Verfahren hat noch Schwächen ...

- Auch mit der obigen Optimierung müssen auf jeder Ebene mindestens immer noch alle Umgrenzungskörper bzw. Objekte getestet werden. Man kann dies verbessern, in dem man die Zahl der Unterkörper pro Knotenpunkt klein hält - allerdings ergeben sich dann mehr Ebenen und der Suchbaum wird schnell größer und durch die vielen Ebenen müssen evtl. wieder sehr viele Schnittests durchgeführt werden.
- Außerdem bleibt das Problem, daß der geeigneter Baum erstellt werden muß. Dazu benötigt man auf der einen Seite einen guten Algorithmus und auf der anderen kann auch dieser selbst zeitaufwendig werden, was einen Teil der Einsparung wieder zu Nichte macht.

1.4.10 Gittermethode



- Raum mit Gitter in gleichmäßige Bereiche einteilen
- Liste von Gitterzellen erstellen

Koordinaten	Objekte
...	
[3][2][1]	A
[4][2][1]	A
...	
[3][3][1]	A
[4][3][1]	A
...	

Anstatt komplizierte Strukturen zu suchen, die sich in der Szene gruppieren lassen, geht man folgendermaßen vor ...

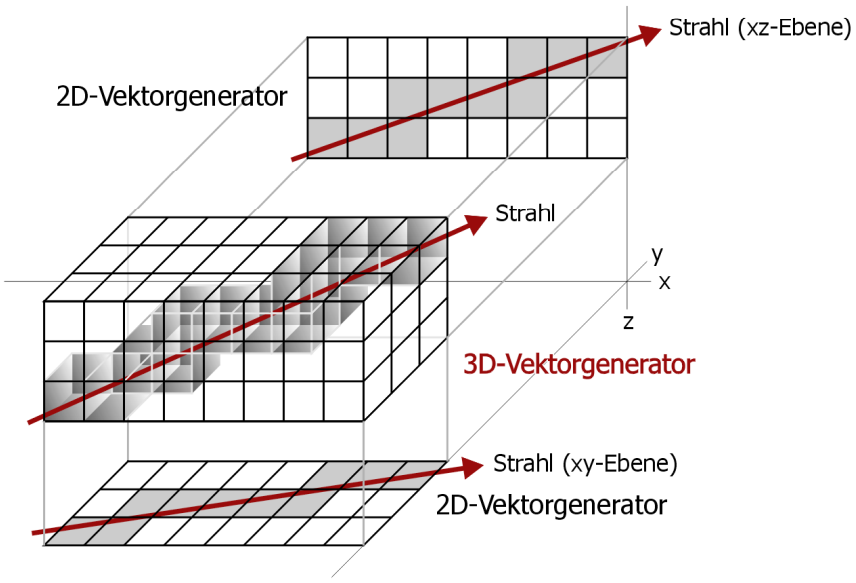
- Man legt ein gleichmäßiges 3-dimensionales Gitter über die komplette Szene und bekommt so viele einzelne Zellen
- Für jede dieser Zellen stellt man nun fest, welche Objekte in ihr liegen oder die Zelle schneiden. Damit ergibt sich dann eine Liste, die alle einzelnen Zellen enthält und ihnen die Menge ihrer Objekte zuweist (evtl. ist diese natürlich auch leer). Im Beispiel ist das Objekt A in insgesamt 4 Zellen enthalten.

Die Methode ist zwar zu den Umgrenzungskörpern nicht ganz unähnlich, allerdings ist das Gitter regelmäßig und alle Zellen sind gleich groß. Damit kann man nun vom Startpunkt die Richtung eines Strahls Gitterpunkt für Gitterpunkt verfolgen. Damit müssen nur noch die Gitterzellen getestet werden, die der Strahl auch wirklich schneidet.

1.4.11 Vektorgenerator

Die Bahn des Strahls entspricht einer Linie, die auf einem Computerbildschirm Pixel für Pixel angenähert wird. Dies berechnet hier ein Vektorgenerator. Da es sich um ein 3-dimensionales Gitter laufen hier einfach zwei 2-dimensionale Vektorgeneratoren parallel (*siehe Grafik nächste Seite)*.

Der Vorteil dieser Vektorgeneratoren ist, daß sie in inneren Schleife nur sehr wenige Operationen benötigen (lediglich zwei Subtraktionen und Vergleiche). Da heutzutage normalerweise auch sehr viel Speicher zur Verfügung steht, es ist damit kein Problem das Netz relativ fein zu machen - aber auch hier gibt es natürlich Grenzen, da ein Gitter mit jeweils 1000 Unterteilungen bereits 1 Milliarde Gitterpunkte hat (durch die vielen Zellen ist dann natürlich auch die Suche bedeutend langsamer). Dieses Problem kann man auch wieder lösen, indem man eine baumartige Struktur aufbaut.



1.4.12 Suche_Objekt_4

Zuerst aber einmal zu der angepaßten Version von **Suche_Objekt** ...

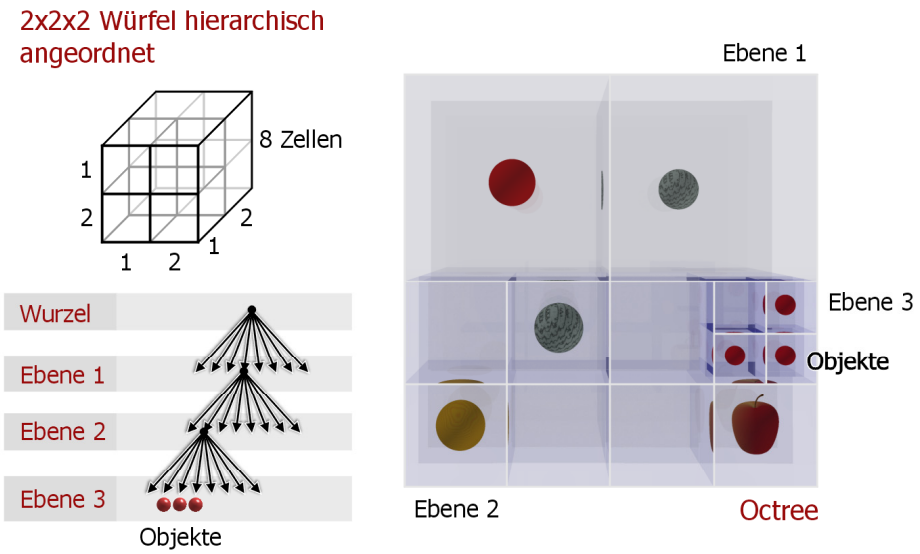
```
Suche_Objekt_4(Strahl)
{
    Schnittpunkt = Strahl verlängert ins Unendliche;
    Objekt = false;
    - Startzelle des Strahls berechnen
    for (Kein Objekt getroffen && Gitter nicht durchquert)
        {
        if (Objektliste der Zelle nicht leer) {
            - Strahl mit allen Objekten der Zelle testen
           if (Objekt getroffen)
                - Objekt merken, das zuerst getroffen wird
           else { - Nächste Zelle berechnen } }
        else { - Nächste Zelle berechnen }
        }
    return(Objekt,Schnittpunkt);
}
```

Zellen durchsuchen

Von der Startzelle aus wird das Gitter schrittweise durchquert, solange kein Objekt getroffen wurde und das Gitter noch nicht zu Ende ist. Für jede Zelle muß nun ihre Objektliste überprüft werden. Ist diese leer kann direkt zur nächste Zelle gesprungen werden. Ansonsten müssen alle Objekte innerhalb geprüft werden. Falls auch hier kein Objekt getroffen wird, springt **Suche_Objekt_4** ebenfalls gleich zur nächsten Zelle. Aber wenn wirklich ein Objekt getroffen wird und es ist das Objekt der Zelle, daß dem Startpunkt am nächsten liegt, dann kann bereits hier abgebrochen werden. Durch die Gittermethode ist ja sichergestellt, daß die Zellen vom Startpunkt aus durchgesucht werden und somit alle weiteren Objekte durch dieses Objekt verdeckt sein müssen. Damit ergibt sich ein weiterer Vorteil der Gittermethode.

1.4.13 Octrees

Ein einfaches Gitter wird z.B. dann problematisch, wenn die Objekt einer Szene z.B. weiter auseinander liegen. D.h. es kann sein, daß ein Großteil der Zellen leer ist und umsonst durchquert werden, wobei auf der anderen Seite sich viele Objekt in einer Zelle befinden und damit wieder zu viele Schnittests benötigt werden. Um das zu umgehen, baut man auch hier wieder eine baumartige Struktur auf. Ein einfaches Beispiel hierfür ist die Erstellung eines Octree ...



Die komplette Szene wird von einem 2×2×2 Würfel, also mit 8 Zellen, umschlossen. Sind zuviele Objekte in einer seiner Zellen, wie im Beispielbild links unten, wird die Würfelzelle wieder auf diesselbe Weise in 8 Zellen unterteilt (zur Übersichtlichkeit ist in jede Zelle nur ein Objekt gezeichnet, im Normalfall können es natürlich auch mehrere sein oder ebenso kann sich ein Objekt auf mehrere Zellen verteilen). Dies wird entsprechend oft wiederholt, bis man einen Baum hat, der an jedem Knoten 8 Nachfolger hat, wie im Bild oben links. Die Blätter des Baumes enthalten nun ein oder mehrere Objekte - oder können natürlich auch leer sein.

Das Ziel ist es, die Octree-Struktur so anzupassen, daß in Bereichen mit vielen Objekten die Szene fein in viele Zellen unterteilt wird und leere Bereiche mit wenigen Zellen, d.h. mit wenigen Schritten, durchquert werden können. Damit wird die Objektliste einer Zelle nie zu lang.

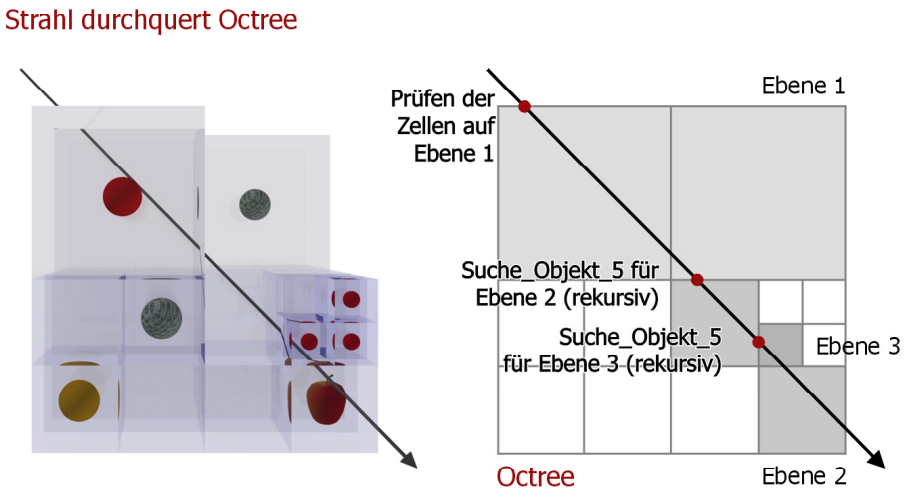
Ein wichtiger Vorteil ist hier, daß das Octree-Verfahren nicht von der Art der Szene abhängt, d.h. daß man praktisch immer einen bedeutenden Geschwindigkeitsvorteil dadurch erzielen kann. Außerdem ist der Octree einfach zu erzeugen, so daß auch hier nicht zuviel Rechenzeit im Vergleich zum Raytracing selbst abfällt, falls die Szene nicht wirklich sehr aufwendig ist. Der Speicherverbrauch im Vergleich zu den normalen Objektdaten dagegen kann sich dadurch natürlich vervielfachen, selbst wenn nur die belegten Zellen platzsparend abgespeichert werden (z.B. 1 Bit pro leerer Zelle, doppelte Listen werden nur einmal gespeichert).

Anstatt eines einfachen Würfels mit nur 8 Zellen, kann man z.B. auch ein Gitter mit jeweils 512 Zellen nehmen (also einem Würfel mit jeweils 8 Zellen an jeder Seite), dadurch erhält man natürlich viel schneller eine sehr feine Einteilung der Szene. Im Buch von Fuchs, auf

dem dieser Votrag basiert, wird z.B. ein Raytracer verwendet, der diese Einteilung benutzt und im Durchschnitt pro Strahl nur noch 2 bis 8 Objekte auf einen Schnitt testen muß - auch bei größerer Objektanzahl. Allerdings gibt es hier auch Ausnahmen, d.h. dies ist kein allgemein gültiger Durchschnittswert. Dies wird im letzten Absatz des nächsten Abschnitts noch einmal aufgegriffen.

1.4.14 Suche_Objekt_5

`Suche_Objekt_5` wäre an einem Stück Programmcode zu unübersichtlich darzustellen, ist aber auch nicht wirklich neu zu den vorherigen Versionen. Es wird die letzte Version für das Gitter mit der Version für die Baumstruktur verbunden ...



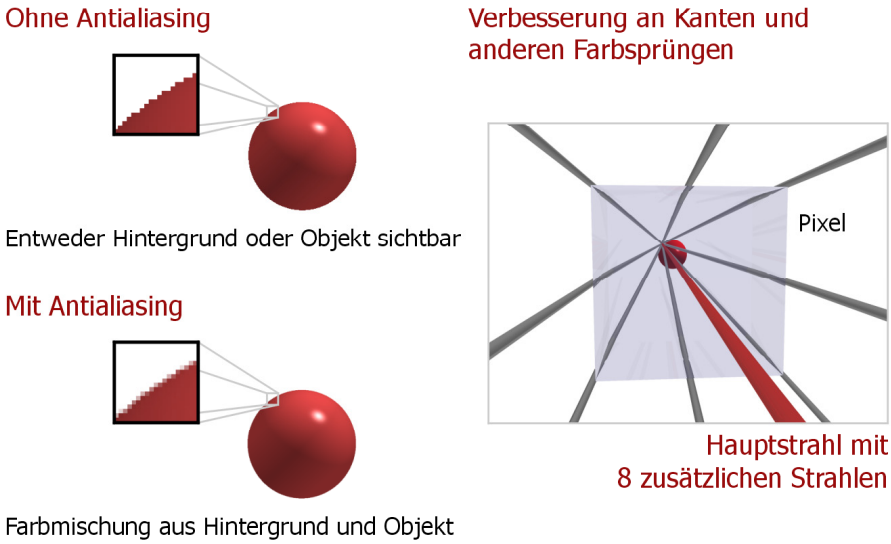
Ein Strahl durchquert nun schrittweise zuerst die Zellen auf der obersten Ebene. Trifft er auf eine Zelle die weiter unterteilt ist, wie die rechts unten, wird **`Suche_Objekt_5`** rekursiv für die zweite Ebene aufgerufen - ebenso für die 3. Ebene. Dies wird solange wiederholt, bis der Strahl auf ein Objekt trifft. Auch hier kann wieder das erste gefundene Objekt direkt zurückgegeben werden, auf das der Strahl trifft, da die Gitterzellen in der richtigen Reihenfolge durchlaufen werden.

Etwas aufwendiger ist diese Methode evtl. für Strahlen, die den Hintergrund treffen (oder allgemein für Strahlen die viele Objekte knapp verfehlen), wie der Strahl im Beispiel oben. Er schneidet keins der wirklichen Objekte, da die Glaswürfel ja eigentlich nicht sichtbar sind. Dies läßt sich aber kaum vermeiden, wenn der Hintergrundsstrahl Bereiche mit vielen Objekten durchquert. Hier können natürlich auch schnell weitaus mehr als 8 Objekte getestet werden müssen, denn im schlimmsten Fall sind immer noch alle Objekte der Szene zu prüfen.

1.5 Antialiasing

Da beim Raytracing die Strahlen durch das Bildschirmraster geschickt werden, wie am Anfang erwähnt, ist das Auflösungsvermögen des Bildes begrenzt, wodurch Kurven und schräge Geraden noch nur angenähert gezeichnet werden können. "Alias" ist die Bezeichnung für die dadurch entstehenden Treppenstufen und, wie der Begriff schon sagt, versucht das Antialiasing dem entgegenzuwirken. Wie im Beispielbild unten links ergeben sich

dann an den Rändern der Kugel leichte Abstufungen, die einen Mischwert aus dem Hintergrund und der Kugelfarbe an diesem Punkt darstellen.



(Anmerkung zum Bild: Auch wenn beim Raytracing der Kugel oben links kein Antialiasing verwendet wurden, erscheint sie auf der entsprechenden Folie im Powerpoint-Vortrag zu dieser Ausarbeitung dennoch wie mit Antiantiasing überarbeitet. Dies liegt daran, daß das Bild insgesamt zweimal skaliert wurde und dabei natürlich auch Antialiasing angewendet wurde.)

Anstatt aber das Bild nach dem Raytracing einfach nur zu glätten, wodurch weitere Detail-Informationen verloren gingen und das Bild unschärfer erscheinen würde, werden dafür mehrere Lichtstrahlen losgeschickt. Denn der im Bild rote Hauptstrahl geht beim Raytracing normalerweise durch die Mitte des Pixels (der komplette Block im Bild stellt 1 Pixel dar) - dadurch wird ein Objekt entweder getroffen oder nicht. Die Randpunkte kann man nun mit weiteren Punkten untersuchen, im Beispiel sind es dann insgesamt 9 Strahlen, die alle Richtungen abdecken und aus denen ein Mittelwert gebildet wird. Dadurch bekommt das Bild zwar nicht wirklich mehr Detail-Informationen, d.h. es sind sehr kleine Gegenstände nicht besser zu erkennen, aber zumindest wird sie damit nicht geringer - dafür erscheint das Gesamtbild wesentlich realistischer.

Da dies aber nur für einen kleineren Teil des Bildes nötig ist (z.B. am Übergang von Objekten und bei manchen Texturen), werden die zusätzlichen Strahlen nur wenn wirklich nötig berechnet. Im Raytracer heißt das, daß mehrere Pixel berechnet werden und falls ein Farbsprung relativ zu den umgebenden Pixel festgestellt wird, berechnet der Raytracer noch zusätzliche Strahlen.

1.6 The Internet Ray Tracing Competition

1.6.1 Compartmentalized Sea

Am Ende habe ich hier noch einige Beispiele aus dem Internet Ray Tracing Competition eingebunden (http://www.irtc.org). Die meisten der Bilder dort wurden mit großen Zeitaufwand erstellt, enthalten eine Vielzahl von Details und wirken zum Teil fast wie Photographien von realen Objekten (bei einigen Bildern ist natürlich auch das Gegenteil beabsichtigt).

Zugelassen sind dort zwar grundsätzliche alle Programme, die Bildern rendern, d.h. auch Nicht-Raytracer, aber die folgenden Bilder wurden alle mit POV-Ray erstellt. POV-Ray ist der bekannteste Raytracer, von dem es mittlerweile mehrere Abwandlungen gibt wie MegaPOV oder POVMan (POV-Ray steht als Open Source zur Verfügung).

IRTC, Ben Weston, POV-Ray 3.1

Beim Raytracing müssen im Normalfall verschiedene Tricks angewendet werden, wenn ein Bild naturgetreu wirken soll. In diesem Bild gibt es zum Beispiel 4 unterschiedliche Arten von Wasser ...

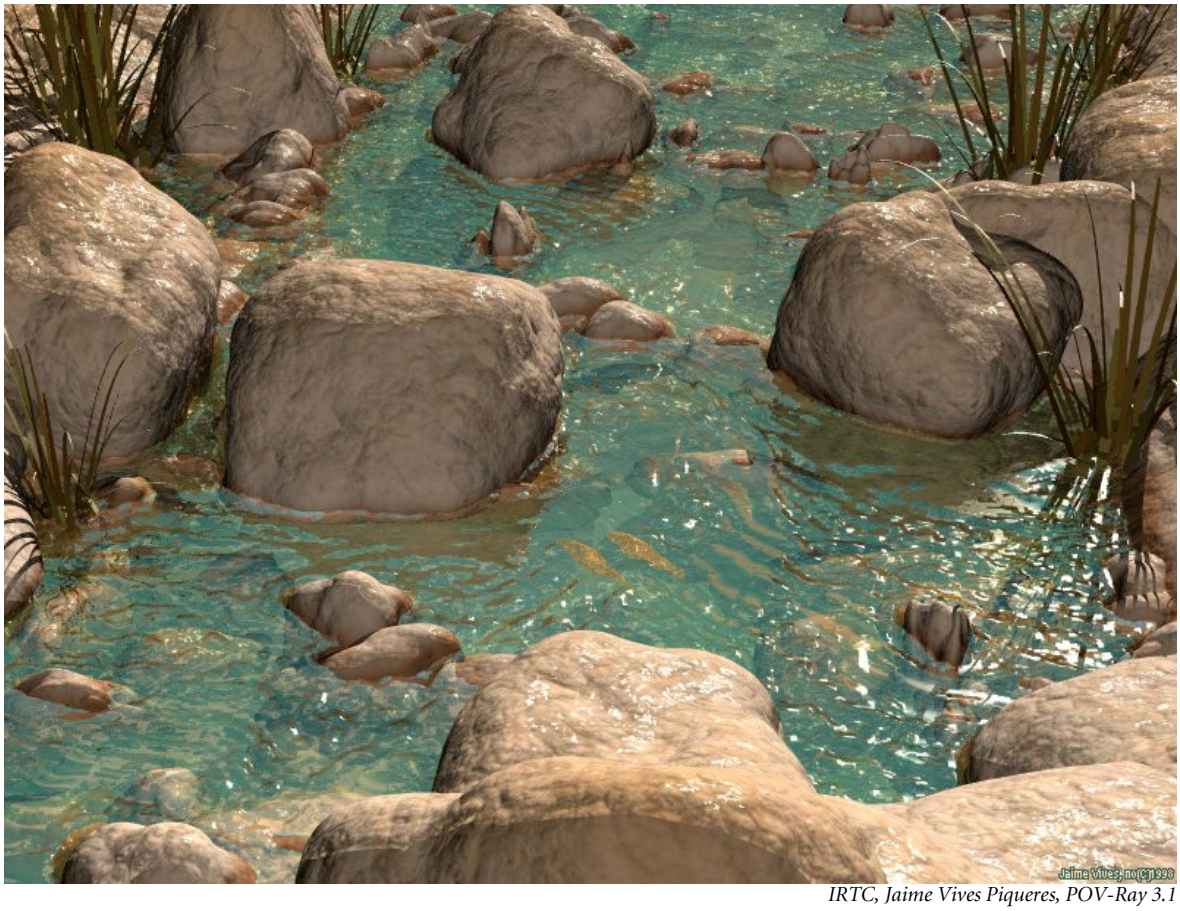
- Das Meer besteht aus einer Bump Map. D.h. es ist eigentlich eine komplett flache Ebene, bekommt aber durch eine Bump Map eine reliefartige Struktur, die in Abhängigkeit von der Lichtquelle die Oberfläche uneben erscheinen läßt. Die Bump Map verändert aber nichts am Objekt selber, d.h. der Querschnitt des Ozeans wäre hier immer noch eine Gerade.

- Das Wasser in den Boxen hat wirklich eine wellenförmige Oberfläche (im Bild schwer erkennbar).

- Der Wasserstrahls besteht aus einem Fraktal.

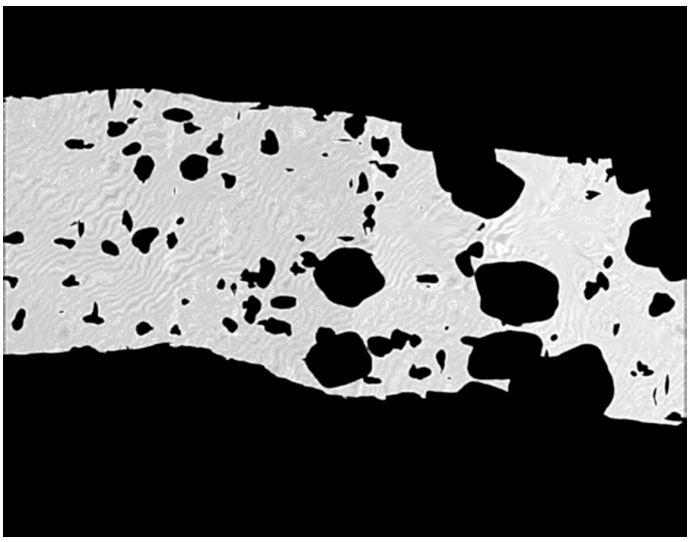
- Der Splash unten am Wasserstrahl ist aus vielen einzelnen Kugeln aufgebaut, die partikalartig angeordnet wurden.

1.6.2 Always running, never the same...



IRTC, Jaime Vives Piqueres, POV-Ray 3.1

Das Wasser in diesem Bild basiert ähnlich wie das Wasser in den Boxen des letzten Bildes auf einer wirklichen Verändungen der Wasserfläche. Die Flächennormale des Wassers wird einfach mit einer Wellenform überlagert. Im Bild scheinen aber die Steine das Wasser zu beeinflussen - hierfür wurde ein Trick angewendet ...



Ein Höhenfeld besteht aus einem Bitmap, das über eine Oberfläche gelegt wird und dessen Farbe Höhenabweichungen auf der Oberfläche angeben. Helle / dunkle Töne stehen dafür, daß die Oberfläche an dieser Stelle höher / tiefer erscheint oder umgekehrt. Dieses Höhen-

feld wurde dadurch erstellt, daß diesselbe Szene mit schwarzen Steinen, einem schwarzen Flußbett und Schwarz-Weiß gewellten Wasser von oben gerendert wurde.

1.6.3 The wet bird

IRTC, Gilles Tran, MegaPOV 0.4

Diese Szene besteht aus sehr vielen Objekten, die erscheinen, als würden sie in atmospärischem Nebel liegen. Dies wurde durch mehrere halb-transparente Ebenen erreicht, die an verschiedene Positionen des Bildes gelegt wurden. Durch eine Schattierungen von unten nach oben erscheint der Nebel unten dunkler und oben heller.

1.6.4 Lord of the Hunt

IRTC, Mick Hazelgrove, POV-Ray 3.5

Auch diese Szene enthält offensichtlich eine Vielzahl von Objekten. Das Gras wurde mit einem Makro in POV-Ray erstellt, wobei es von den der Szene abgewandten Oberflächen entfernt wurde, um die Objektanzahl zu verkleinern. Dennoch benötigt das Bild auf einem Athlon 800 etwa 10 Stunden Rechenzeit.

1.6.5 Spider project

IRTC, Jonathan Rafael Ghiglia, MegaPOV 0.7

Hier wurden, um die Lichtquelle flächig erscheinen zu lassen, viele Lichtquellen ein einer Reihe entlang der Lampe verwendet (ich habe deshalb noch einmal bei Autor des Bildes nachgefragt). Dieses Streulicht wäre aber auch durch Radiosity berechnbar (genauer, aber mit mehr Aufwand), was im anschließenden Vortrag behandelt wird.

Wer jetzt selbst einmal das Raytracing ausprobieren möchte, kann sich unter www.povray.com kostenlos die aktuelle Version von POV-Ray downloaden und selber einfache Szenen erstellen oder auch fertige Szenen/Objekte aus dem Internet auf dem eigenen PC rendern lassen.

1.7 Verwendete Quellen

Bücher (Inhalt)

- Amiga reflections - Traumwelt und Realismus (Fuchs, Carsten), Markt&Technik-Verlag 1989

Programme (Cliparts, Texturen)

- Corel CorelDRAW 9
- Corel CorelDREAM 3D 8 (auch bekannt als Raydream von Fractal Design)

Internet (Bilder)

- The Internet Ray Tracing Competition (http://www.irtc.org)

2 INDEX

- A -

Amiga reflections 35
Antialiasing 29
Aufwand 20

- B -

Beleuchtungsformel 10, 15
Berechne_Farbe 6, 10, 13
Berechungsstrahl 10
Bilderzeugung 5
Bildschirmraster 4
Brechung 9
Brechungsgesetz 9
Brechungsvektor 11
Brechungszahl 9
Bump Map 31

- C -

Corel CorelDRAW 9 35
Corel CorelDREAM 3D 8 35

- D -

Direktes Licht 12, 13

- E -

Eigenleuchten 16

- F -

Farbberechnung 10
Farbmodell 11

- G -

Gittermethode 25
Glanzlicht 14
Glanzlichtfunktion 14
Glas 9, 17
Gras 34

- H -

Halbstrahl-Verfahren 19
Hauptstrahl 30
Höhenfeld 32
Holz 17

- I -

Indirektes Licht 12
Internet Ray Tracing Competition 30

- L -

Lichtanteile 12
Lichtquelle (Eigenschaften) 13
Lichtquellenvektor 11
Lichtteilchen 4

- M -

Materialien 16
MegaPOV 31
Metall 17
Mikrofacetten 14

- N -

Nebel 33
Normalenvektor 10

- O -

Oberflächeneigenschaften 16
Objektbeschreibung 5
Octrees 27

- P -

Photonen 4
Plastik 17
POVMan 31
POV-Ray 30, 35

- R -

Radiosity 35
Raytrace_1 6
Raytrace_2 7
Raytrace_3 8
Raytrace_4 9
Raytracing 4
Reflektions-Tiefe 8
Reflektionsvektor 11
RGB-Modell 11

- S -

Schatten 6
Schattierung 6
Schnittpunkt 18
Sehstrahl 10
Sonnenlicht 12
Spezialeffekte 6
Spiegelstrahl 8
Spiegelung 7
Splash 31
Startzelle 27
Stein 17
Streulicht 12, 35
Suche_Objekt 6, 17
Suche_Objekt_2 21
Suche_Objekt_3 23
Suche_Objekt_4 26
Suche_Objekt_5 29

- T -

Taschenlampe 13
Textur 17

- U -

Umgebungslicht 7, 12
Umgrenzungskörper-Hierarchie 23
Effizienz 24
Umgrenzungskugeln 21
Effizienz 22

- V -

Vektorgenerator 26

- W -

Wasser 31, 32
Wasserstrahl 31