AUF DER HEFT-CD

Demo-Programmierung unter Windows 95/98/NT

Primitive in Perfektion

Zwei neue Klassen von Primitiven vervollständigen den bisher entwickelten Raytracer, den Sie zudem **gezielt optimieren**.

CARSTEN DACHSBACHER

Unsere Welt besteht aus einer Vielzahl von Formen. Daher verlangt die Gestaltung wirklichkeitsgetreuer Szenerien in einem Raytracer auch nach komplexeren Objekten. Diese bauen Sie am einfachsten aus Polygonen und sogenannten CSG-Primitiven (Constructed Solid Geometry) zusammen. Dadurch steigt der Anspruch an die Rechen-Performance – dem werden Sie durch gezielte Verbesserungen gerecht.

Polygon-Primitive

An dieser Stelle lernen Sie die Primitive kennen, die Sie wahrscheinlich schon am längsten vermißt haben: Polygone. Wir beschränken uns dabei auf ihre einfachsten Vertreter, die Dreiecke. Für Schnittpunktberechnungen mit Dreiecken existieren verschiedener Algorithmen. Wir stellen Ihnen das allgemeine Prinzip vor – eine besonders elegante Variante finden Sie im Quellcode. Sie stammt aus dem *Journal of Graphics Tools* (siehe Literaturtips am Ende).

Ein Dreieck definieren Sie durch seine drei Eckpunkte. Vielleicht erinnern Sie sich, daß Sie auf die gleiche Weise auch eine Ebene im Raum plaziert haben. Um einen Schnittpunkt mit einem Dreieck zu besitzen, muß eine Gerade notwendigerweise auch die Ebene schneiden, in der das Dreieck liegt – also die Ebene, die durch die drei Eckpunkte bestimmt wird.

Zudem muß der berechnete Schnittpunkt innerhalb des Dreiecks liegen. Betrachten Sie dazu das Koordinatensystem in der Abbildung unten.

Dort finden Sie die Funktion

```
y = -x + 1
```

eingezeichnet. Durch Umformen erhalten Sie daraus

```
x + y = 1
```

Außerdem sehen Sie im Bild ein Dreieck, dessen linker unterer Eckpunkt durch den Ortsvektor $\vec{a}$ festgelegt ist und dessen Kanten mit $\vec{b}$ bzw. $\vec{c}$ beschriftet sind.

Definieren Sie, daß der Einheitsvektor der x-Achse gleich $\vec{b}$ und der Einheitsvektor der y-Achse gleich $\vec{c}$ ist, können Sie in der obigen Formel x durch u, y durch v und das Gleichheitszeichen durch „kleiner gleich“ ersetzen. Daraus ergeben sich folgende Bedingungen für die Dreiecksfläche:

```
u + v <= 1
u >= 0
v >= 0
```

Die Werte u und v für einen berechneten Schnittpunkt $\vec{s}$ der Geraden mit der Ebene entnehmen Sie dem Skalarprodukt. Dieses berechnet, wie lang die Projektion eines Vektors auf einen anderen ist. Die Differenz zwischen $\vec{s}$ und $\vec{a}$ ergibt genau den Vektor, den Sie auf die Kanten des Dreiecks projizieren:

```
x⃗ = s⃗ - a⃗
u1 = x⃗ * b⃗
v1 = x⃗ * c⃗
```

Für das Skalarprodukt gilt ganz allgemein:

```
t * (x⃗ * y⃗) = (t * x⃗) * y⃗
```

Um u_1 und v_1 richtig zu skalieren, genügt es daher, sie durch die Länge der Kanten zu teilen:

```
u = u1 / |b⃗|
v = v1 / |c⃗|
```

Daraus ersehen Sie, ob der Schnittpunkt im Dreieck liegt:

```
if (u>0 && v>0 && (u+v)<=1)
  return true;
else return false;
```

Nachdem Sie jetzt Schnittpunkte mit Dreiecken berechnen können, bleibt die Frage der Beleuchtung. Natürlich haben Dreiecke eine gerade Oberfläche – genauso wie die Ebenen, in denen sie liegen. Die Normale bleibt also überall dieselbe. Daher bräuchten Sie nur die einmal berechnete Normale in der Beleuchtungsgleichung verwenden.

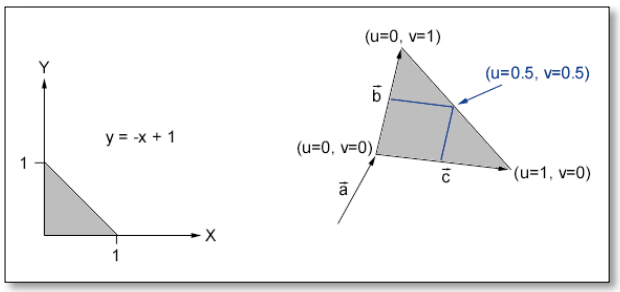
Allerdings dienen Dreiecke oft dazu, beliebig geformte Flächen anzunähern. Um zum Beispiel runde Flächen rund erscheinen zu lassen, benötigen Sie eigentlich Unmengen von Dreiecken. Dadurch steigt auch der Rechenaufwand immens. Möchten Sie mit wenigen Dreiecken auskommen, können Sie zumindest in der Beleuchtung die Fläche runder erscheinen lassen, als sie wirklich ist.

Die Phong-Schattierung täuscht gewölbte Flächen durch die Interpolation des Normalenvektors vor. Dazu weisen Sie nicht jedem Dreieck, sondern jedem Eckpunkt eine Normale zu. Die Normale an einem Schnittpunkt innerhalb des Dreiecks erhalten Sie dann durch die Interpolation der Normalen an den drei Eckpunkten. Zu jedem Dreieck berechnen Sie dazu die Normale an einem der Eckpunkte sowie die Differenzen zu den Normalen an den zwei anliegenden Kanten.

Sind also $\vec{n_1}$, $\vec{n_2}$ und $\vec{n_3}$ die Normalen an den Eckpunkten, dann berechnen Sie:

```
k⃗ = n⃗2 - n⃗1
t⃗ = n⃗3 - n⃗1
```

Aus der Schnittpunktberechnung besitzen Sie bereits die beiden Parameter u und v.



ANHAND DER WERTE u und v sehen Sie, ob ein Punkt im Dreieck liegt.

Für die Normale am Schnittpunkt gilt dann:

$\vec{n} = \vec{n}_1 + u * \vec{k} + v * \vec{t}$

Bevor Sie diese in die Berechnung der Beleuchtung einsetzen, normalisieren Sie sie noch: Auch wenn die ursprünglichen Normalen an den Eckpunkten bereits normalisiert vorliegen, ist durch die Interpolation nicht mehr gewährleistet, daß $\vec{n}$ die Länge *1* besitzt.

In der Skriptsprache des Parsers können Sie sowohl Dreiecke mit konstanter Normale als auch solche mit Phong-Schattierung definieren. Ein 3D-Objekt aus Dreiecken beginnen Sie zunächst mit

```
mesh {...}
```

Innerhalb der geschweiften Klammern geben Sie die zwei Dreiecksprimitive an:

```
triangle
{
 <x1,y1,z1>, <x2,y2,z2>,
 <x3,y3,z3>
}
```

definiert Dreiecke mit konstanter Normale, die dann berechnet wird. Für Dreiecke mit Phong-Schattierung geben Sie in

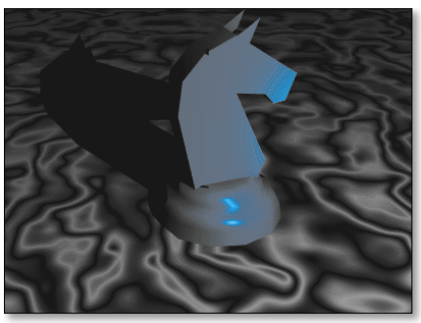
```
smooth_triangle
{
 <x1,y1,z1>, <x2,y2,z2>,
 <x3,y3,z3>, <nx1,ny1,nz1>,
 <nx2,ny2,nz2>, <nx3,ny3,nz3>
}
```

zusätzlich noch die Normalen der Eckpunkte an.

Ein solches Polygonobjekt sehen Sie im Bild unten. Die Polygondaten dafür stammen aus einer mit POV-Ray generierten Szenerie.

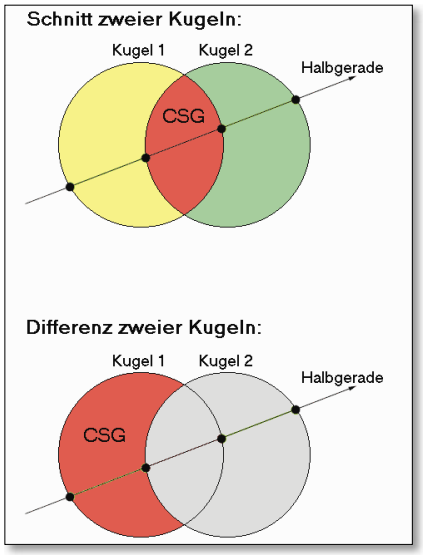
CSG-Primitive

Auch wenn sich durch Polygone prinzipiell alle Oberflächen annähern lassen, ist dies manchmal nicht die einfachste oder genaueste Lösung. Dann eignet sich vielleicht eher die Klasse der durch Constructed Solid Geometry (CSG) erzeugten Körper. Hinter dieser Bezeichnung, die sich nur schwer ins Deutsche übersetzen läßt, verbirgt sich ein Verfahren, mit dem Sie zwei oder mehrere einfache Primitive wie Kugel, Ebene oder Zylinder miteinander verknüpfen.



DIE PHONG-SCHATTIERUNG rundet die Kanten des Springers etwas ab.

Die möglichen Verknüpfungen sind dabei Vereinigung, Schnitt und Differenz. Diese Begriffe aus der Mengenlehre können Sie ohne weiteres auf die Primitive übertragen, da diese gewissermaßen Teilmengen des Raums darstellen. Anhand zweier Kugeln können Sie sich die erlaubten Operationen leicht vor Augen führen. Betrachten Sie dazu die rot schraffierte Schnittmenge zweier Kugeln im folgenden Bild.



DURCH SCHNITT und Differenz von Primitiven schaffen Sie CSG-Primitive.

Um die Schnittpunkte mit der Schnittmenge festzustellen, berechnen Sie zunächst alle Schnittpunkte der betrachteten Geraden mit den beiden Primitiven. Die Schnittpunkte mit der Schnittmenge finden Sie durch folgenden Algorithmus in Pseudocode:

```
Betrachte alle Schnittpunkte
der Primitive:
   Ist der Punkt von Primitiv 1
   und liegt in Primitiv 2
   oder
   ist der Punkt von Primitiv 2
   und liegt in Primitiv 1
dann Schnittpunkt gefunden
```

Sie sehen: Die Primitive – oder vielmehr die entsprechenden C++-Klassen – verlangen eine Methode, die angibt, ob ein Punkt im Inneren des Primitivs liegt. Im Falle der Kugel berechnet diese einfach den Abstand des Punkts vom Kugelmittelpunkt. Ist er kleiner oder gleich dem Radius, dann liegt der Punkt im Inneren.

Bei einer Ebene ist zunächst unklar, welche Seite den inneren bzw. äußeren Teil darstellen soll. Per Definition sei daher festgelegt, daß der Halbraum – eine Ebene teilt den Raum in zwei Hälften – außen ist, in den die Normale zeigt.

Dadurch reduziert sich der Aufwand für den Innen-/Außen-Test auf ein Skalarprodukt des zu prüfenden Punkts mit der Normalen, wovon Sie noch den (immer vorberechneten) Abstand der Ebene zum Ursprung subtrahieren. Ist das Resultat kleiner oder gleich Null, liegt der Punkt im Inneren.

Natürlich können Sie CSG-Objekte auch aus solchen Primitiven zusammensetzen, die ihrerseits CSG-Objekte sind. Auch diese haben alle für die Schnittpunktberechnung notwendigen Methoden implementiert.

Eine weitere mögliche Verknüpfung zweier Objekte ist die Vereinigung. Diese gestaltet sich besonders einfach, da Sie alle Schnittpunkte der Einzelprimitive auch als Schnittpunkte des CSG-Objekts verwenden können. Das ist deshalb erlaubt, da beim Raytracing sowieso der nächste Schnittpunkt gesucht wird. Alle weiter entfernten Schnittpunkte – auch die im Inneren des Vereinigungsobjekts – fallen nicht ins Gewicht.

Die letzte Verknüpfungsmethode ist die Differenz. Sie können sich das als das Herausschneiden eines Objekts aus einem anderen vorstellen.

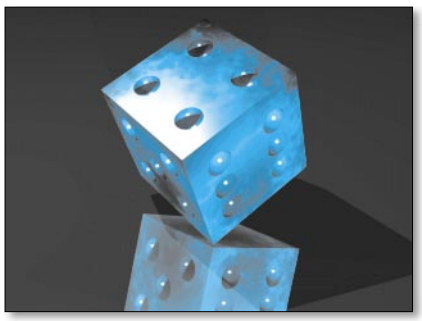
In diesem Fall gilt für die Schnittpunktklassifikation:

```
Betrachte alle Schnittpunkte
der Primitive:
   Ist der Punkt von Primitiv 1
   und liegt nicht in Primitiv 2
   oder
   ist der Punkt von Primitiv 2
   und liegt in Primitiv 1
dann Schnittpunkt gefunden
```

Damit haben Sie das Prinzip der CSG-Objekte erfaßt.
Ein Beispiel für ein komplexeres CSG-Objekt ist der Würfel in der Abbildung auf der nächsten Seite. Der Würfel ist die Schnittmenge aus sechs Ebenen. Die Vertiefungen der Augenzahlen entstehen durch herausgeschnittene Kugeln. Die Skriptdatei dazu finden Sie auf der Heft-CD bei den Quelltexten.

Schnittpunktberechnungen optimieren

Wollen Sie die Berechnung eines Bilds mit dem Raytracer beschleunigen, beginnen Sie bei den am häufigsten verwendeten Routinen. Ihr Augenmerk fällt dabei wohl zuerst auf die Schnittpunktberechnungen, die den größten Teil der Rechenzeit in Anspruch neh- ▶



DIE AUGEN DES WÜRFELS sind halbkugelförmige Vertiefungen.

men. An diesen mathematischen Problemen haben sich bereits viele versucht, und dementsprechend viele Algorithmen für Schnittpunktberechnungen für Primitive aller Art gibt es.

Im Quelltext *RTTriangle.cpp* des Polygonprimitivs finden Sie etwa einen eleganten Ansatz, um einen Schnittpunkt einer Gerade und eines Dreiecks zu berechnen. An dieser Stelle möchten wir Ihnen zeigen, wie Sie die Schnittpunkte einer Kugel anhand geometrischer Überlegungen schneller berechnen.

Vor einer Schnittpunktberechnung wissen Sie nicht, ob es überhaupt einen Schnittpunkt gibt. Eine oft verwendete Technik bei der Optimierung von Schnittests ist es, durch möglichst einfache Berechnungen bereits sehr früh festzustellen, ob der Strahl das Objekt sicher verfehlt. Diese Tests werden Rejection-Tests genannt – verläuft der Test negativ, gibt es keinen Schnittpunkt. Für den Fall der Kugel betrachten Sie am besten die drei Skizzen unten, die sich jeweils nur in der Lage von $\vec{o}$ unterscheiden. $\vec{o}$ ist dabei der Startpunkt der Halbgeraden, $\partial\vec{x}$ die Richtung der Geraden. Für die Halbgerade gilt:

$\vec{x} = \vec{o} + t * \partial\vec{x}$

Schließlich gibt es noch den Vektor $\vec{c}$ für den Kugelmittelpunkt. Der erste Rejection-Test berücksichtigt die Lage der Kugel bezüglich $\vec{o}$. Von Interesse sind nur Kugeln, die vor dem Startpunkt der Geraden liegen. Dazu berechnen Sie den Vektor vom Startpunkt zum Kugelmittelpunkt, also

$\vec{t} = \vec{c} - \vec{o}$

Daraus ermitteln Sie die quadrierte Länge von $\vec{l}$ – also das Quadrat des Abstandes – mit:

$l^2 = \vec{t} * \vec{t}$

Gleichzeitig haben Sie mit r^2 das Quadrat des Kugelradius gegeben, der bei der Initialisierung eines Kugelobjekts vorberechnet wird. Damit entscheiden Sie nun folgendes: Ist l^2 kleiner als r^2, befindet sich $\vec{o}$ in der Kugel, und es gibt (genau) einen Schnittpunkt. Wollen Sie nur feststellen, ob es überhaupt einen Schnittpunkt gibt, können Sie an dieser Stelle die Berechnung abbrechen.

Da Sie allerdings den Schnittpunkt bestimmen wollen, berechnen Sie als nächstes die Projektion von $\vec{l}$ auf $\vec{d}$. Dies geschieht mit dem Skalarprodukt

$d = \vec{t} * \partial\vec{x}$

Nun wenden Sie den ersten Rejection-Test an: Liegt $\vec{o}$ außerhalb der Kugel – also ist l^2 größer als r^2 –, und ist d negativ? Falls ja, gibt es keinen Schnittpunkt. Ansonsten fahren Sie mit der Berechnung fort.

Als nächstes interessiert Sie m^2, das Abstandsquadrat des Kugelmittelpunkts zu der Projektion von $\vec{l}$. Da es sich um ein rechtwinkliges Dreieck handelt, wenden Sie den Satz des Pythagoras an:

$m^2 = l^2 - d^2$

Nun sind Sie am zweiten Rejection-Test angelangt: Ist m^2 größer als r^2, dann wird der Strahl am Objekt vorbeischießen, ansonsten sicher treffen. Im letzteren Fall existieren also Schnittpunkte, die es zu berechnen gilt. Lösen Sie dazu die Gleichung

$q^2 = r^2 - m^2$

Da wegen des letzten Rejection-Tests $m^2 <= r^2$ gilt, ist q^2 größer oder gleich Null. Das bedeutet, daß Sie ohne Probleme die Wurzel daraus ziehen können:

$q = \sqrt{q^2}$

Um schließlich die Schnittpunkte zu bestimmen, berechnen Sie die Entfernungen zu den Schnittpunkten – also die Werte für t aus der Geradengleichung. Dabei gilt:

$t1 = d - q$
$t = d + q$

Die Routine sieht dann in etwa folgendermaßen aus:

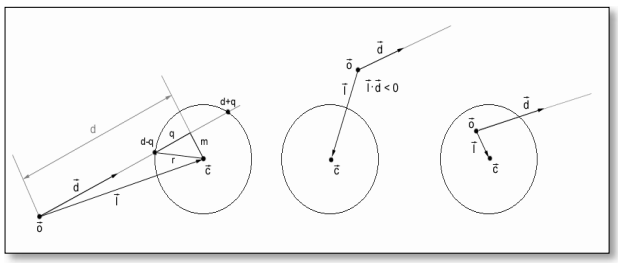
```
bool RaySphereIntersect(
  VERTEX3D o, d, c,
FLOAT r)
{
  VERTEX3D l = c-o;
  Float d = l*d;
  float l2 = l*l;
  float r2 = r*r;

  if (d<0 && l2>r2) return 0;

  float m2 = l2-d2;
  if (m2>r2) return 0;

  q = sqrt(r2-m2);
  t1 = d-q;
  t2 = d+q;
  return 1;
}
```

Wie Sie sehen, reduziert sich der Aufwand für eine Schnittpunktberechnung im Vergleich zur ursprünglichen Implementierung in Ausgabe 10/99 deutlich. Der damalige Ansatz verfolgt gewissermaßen eine analytische Lösung und dient vor allem als Einführung in die Schnittpunktberechnungen.



MIT REJECTION-TESTS bestimmen Sie frühzeitig, ob der Strahl ein Objekt verfehlt.

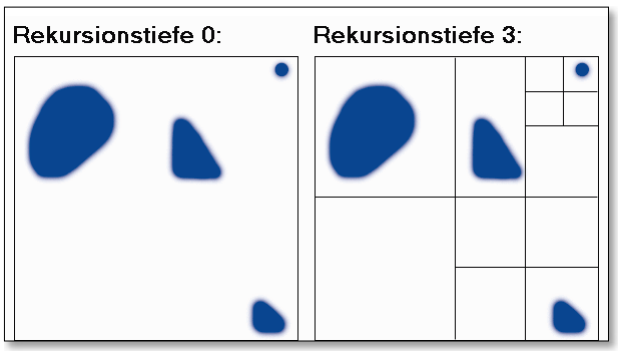
Schnittpunktberechnungen vermeiden

Schnelle Algorithmen für effiziente Schnittpunktberechnungen sind eine gute Grundlage für einen Raytracer. Bei Szenen mit Tausenden von Objekten wird Ihr Rechner trotzdem unerträglich lange arbeiten, da er für jedes Pixel des Bildes, für jede Rekursionstiefe und für jeden Schattenstrahl Schnittpunkttests mit allen Objekten durchführen muß. Deshalb erzielen Sie bei komplexen Szenen deutliche Geschwindigkeitssteigerungen, wenn Sie sich darüber Gedanken machen, wie Sie Schnitttests ganz vermeiden können.

Nutzen Sie für dreidimensionale Umgebungen das sogenannte Octrees-Verfahren: Octrees sind baumartige Speicherstrukturen mit jeweils acht Nachfolgern.

Im zweidimensionalen Raum läßt sich das Prinzip noch anschaulicher an den dort verwendeten Quadtrees verdeutlichen. Wie ihr Name besagt, besitzen diese jeweils vier Nachfolger.

Stellen Sie sich eine beliebige Anordnung mehrerer Objekte in einer Ebene vor, wie etwa in der Abbildung auf der nächsten Seite links oben. Nun ordnen Sie die Objekte in einer hierarchischen Struktur wie folgt an: Legen Sie ein möglichst kleines Quadrat so um die Objekte, daß diese vollständig darin enthalten sind. Vierteln Sie das Quadrat, und Sie erhalten vier kleinere, im ursprünglichen Quadrat enthaltene Quadrate. Diesen Vorgang wiederholen Sie so lange, bis jedes Quadrat entweder leer ist oder genau ein Objekt enthält. Sie erhalten dann ein Gittermuster.



REKURSIV BAUEN SIE einen Octree für eine Szene auf.

Wollen Sie einen Schnittest eines Strahls mit einem Objekt durchführen, testen Sie nur die Objekte, bei denen Sie auch das dazugehörige Quadrat treffen. Trifft ein Strahl ein Quadrat nicht, so können Sie auch alle Objekte, die in den darin enthaltenen Quadraten liegen, von dem Schnittest ausschließen.

Bei den dreidimensionalen Octrees teilen Sie einen Würfel rekursiv in seine acht Unterwürfel auf. Um herauszufinden, wieviel Platz ein Primitiv benötigt, geben Sie einen möglichst kleinen Würfel an, der ein Primitiv vollständig enthält. Das gestaltet sich relativ einfach, wenn Sie einen Würfel – eine sogenannte Bounding Box – verwenden, dessen Kanten parallel zu den Koordinatenachsen verlaufen. Somit sind seine Seitenflächen parallel zu den x/y- und x/z-Ebenen des Koordinatensystems.

Solche Würfel heißen Axis-Aligned Bounding Boxes (AABBs). Eine AABB legen Sie durch zwei Ortsvektoren fest: Diese geben jeweils die minimalen bzw. maximalen x-, y- und z-Koordinaten an, die das eingeschlossene Primitiv annimmt. Für eine Kugel bestimmen Sie eine AABB also wie folgt:

```
Kugelmittelpunkt  o⃗ = (x,y,z)
Kugelradius r
minV = (x-r, y-r, z-r)
maxV = (x+r, y+r, z+r)
```

Nicht für alle Primitive können Sie ohne weiteres eine AABB bestimmen: Das Ebenenprimitiv besitzt zum Beispiel keine endliche Ausdehnung. Diese speziellen Primitive behandeln Sie deshalb unabhängig vom Octree-Verfahren.

Um den Octree für eine 3D-Szenerie aufzubauen, berechnen Sie die AABBs für alle Primitive, bei denen dies möglich ist. Dann bestimmen Sie einen Würfel, dessen Mitte im Koordinatenursprung liegt und der alle AABBs enthält. Nun fügen Sie jedes Primitiv mit einer rekursiven Prozedur in den Octree ein. Die Speicherungsstruktur eines Octree-Würfels bezeichnet man – analog zur Namengebung bei strukturierten Bäumen – als Node (Knoten).

Folgender Pseudecode fügt ein Objekt in den Octree ein:

```
BOOL
InsertObject(OCTRE-
ENODE
  *node, RTObject
*o,
  const EXTEND *e)
{
  Liegt die AABB
des Objekts
    (EXTEND e) in dem Node ?
  Wenn nein, return False;
  if (noch nicht maximale
    Unterteilung)
  {
    Erzeuge 8 Unterwürfel
    Versuche, das Objekt dort
      einzufügen
  }
  Wenn noch nicht eingefügt,
    dann in den aktuellen
    Octreenode einfügen
  return True;
}
```

Den programmierten Code zeigt der Ausschnitt aus der Datei *RTOctree.cpp* (Listing 1) auf der nächsten Seite oben.

Bei einer Schnittpunktberechnung testen Sie einfach rekursiv den Octree mit den darin enthaltenen Objekten. Sie starten an der Wurzel, dem ersten Node des Octrees. Schneidet der Strahl die dazugehörige AABB, berechnen Sie die Schnittpunkte mit den Objekten in diesem Node.

Anschließend rufen Sie die Berechnungsroutine für jeden der acht Subnodes (Unterwürfel) auf. Dadurch schließen Sie bei den meisten Schnittpunktberechnungen viele Objekte durch wenige Schnittests mit AABBs aus.

Die Objekte, zu denen Sie keine AABBs berechnen konnten, speichern Sie in einer Liste. All diese Objekte unterziehen Sie wie bisher den Schnittpunkttests. Die gesamte Schnittpunktberechnung läuft schematisch wie im zweiten Listing auf der nächsten Seite.

Damit haben Sie fast alle Routinen für das Octree-Verfahren zusammen. Es fehlt noch der Schnittest für einen Strahl und eine AABB. Begrenzungsvolumen für Körper wie AABBs sind in der 3D-Grafik ein weitverbreitetes Mittel für Geschwindigkeitssteigerungen. Daher gibt es für solche Problemstellungen verschiedene Algorithmen.

Unser Raytracer verwendet die sogenannte Slab-Methode. Diese kommt auch mit Bounding Boxes zurecht, bei denen die Kanten nicht Axis-Aligned sind – also wie die AABBs an den Koordinatenachsen ausgerichtet. Der Begriff Slab bezeichnet hier zwei parallele Ebenen, die aus Geschwindigkeitsgründen bei der Berechnung gruppiert sind.

Da eine AABB durch drei parallele Ebenenpaare begrenzt wird, können Sie sie durch drei Slabs darstellen. Diese Slabs benennen Sie mit *u*, *v* und *w*. Für ein Slab können Sie – analog zu Ebenenprimitiven – Schnittpunkte berechnen. Da es sich immer um zwei Ebenen handelt, erhalten Sie jeweils einen maximalen und einen minimalen Wert für die Schnittpunkte.

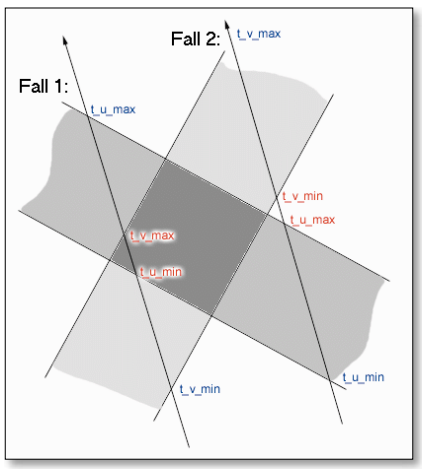
In diese Werte – zum Beispiel *t_u_min* oder *t_v_max* – fließt noch der Faktor *t* aus der Geradengleichung ein. Die nächste Berechnung ist dann der eigentliche Trick:

```
t_min = max(t_u_min, t_v_min,
  t_w_min);
t_max = min(t_u_max, t_v_max,
  t_w_max);
```

Bei *t_min* <= *t_max* schneidet der Strahl die Bounding Box, ansonsten verfehlt er sie. Den Sourcecode zu dieser Routine finden Sie in der Datei *RTOctree.cpp*. Der einfachere zweidimensionale Fall in der Skizze unten veranschaulicht die Arbeitsweise recht gut.

Auch Slabs sind hier eine Dimension kleiner, also zwei parallele Geraden. In der Grafik sehen Sie eine zweidimensionale Bounding Box aus zwei Slabs sowie zwei Geraden, an denen Schnittests durchgeführt werden sollen.

Im ersten Fall ist *t_min* das Maximum der Werte *t_u_min* und *t_v_min*, also *t_u_min*. Ebenfalls rot gekennzeichnet ist *t_v_max*, was dem Minimalwert *t_max* von *t_u_max* und *t_v_max*



DIE ROT MARKIERTEN WERTE sagen aus, ob der Strahl das Rechteck schneidet.

entspricht. Da *t_min* kleiner als *t_max* ist, schneidet die linke Gerade die Bounding Box.

Im zweiten Fall ist *t_min* = *t_v_min* größer als *t_max* = *t_u_max*. Daher zielt die rechte Gerade an der Bounding Box vorbei.

Mit dem fertigen Raytracing-Programm *OORT.exe* können Sie nun faszinierende Computerwelten entwerfen und beleuchten. Berechnen Sie schöne Bilder damit, und schicken Sie sie uns zu – wir freuen uns auf Ihre Zusendung.

Selbstverständlich können Sie das Programm um neue Primitive, Lichtquellen, Beleuchtungsmethoden oder Optimierungen erweitern.

Möchten Sie sich weiter über Raytracing informieren, empfehlen wir Ihnen die im Anschluß zitierte Literatur sowie einen Blick in den Sourcecode von POV-Ray. Dessen Webseite finden Sie unter

www.povray.org PEI

Die Quelltexte sowie den übersetzten Raytracer *OORT.exe* finden Sie zusammen mit der zugrundeliegenden Grafikbibliothek auf unserer Heft-CD 2 in der Rubrik *Praxis/Programmierung/PC Underground* und in unserem Internet-Angebot unter

www.pc-magazin.de/magazin/➥extras.htm

Klicken Sie unter *Online Extras* im Menü *Praxis* auf das entsprechende *Download*-Feld.

Literatur zum Thema Raytracing:

Journal of Graphics Tools (JGT): Online-Magazin, erhältlich unter
www.acm.org/jgt

Foley, van Dam, Feiner, Hughes, Phillips: Grundlagen der Computergrafik, Addison Wesley 1994, 100 Mark, ISBN 3-89319-647-1

Wilt: Object-Oriented Raytracing in C++, John Wiley & Sons 1993, etwa 130 Mark, ISBN 0-47130-415-8

Watkins, Coy, Finlay: Fotorealismus und Raytracing in C, Heise-Verlag 1993, 88 Mark, ISBN 3-88229-024-2

Müller, Haines: Real-Time Rendering, AK Peters Ltd., ISBN 1-56881-101-2

1 Ausschnitt aus RTOctree.cpp

```
1: //Ausdehnung eines Objekts
2: typedef struct
3: {
4:   //endliche Ausdehnung?
5:   BOOL finite;
6:   //wenn ja, dann hier die AABB-Daten
7:   VERTEX3D Amin, Amax;
8:   FLOAT size;
9: } EXTEND;
10:
11: typedef struct NODE
12: {
13:   //Objekte im Würfel
14:   List<RTObject*> Objects;
15:
16:   //Zeiger auf die Unterwürfel
17:   NODE *SubNode[8];
18:
19:   //Mittelpunkt und Größe des Würfels
20:   VERTEX3D m;
21:   FLOAT size;
22:
23:   //Rekursionstiefe des Würfels
24:   UINT32 Level;
25: } OCTREENODE;
26:
27: static OCTREENODE *Octree;
28:
29: BOOL InsertObject(OCTREENODE *node, RTObject *o,
30:   const EXTEND *e)
31: {
32:   //gehört das Objekt überhaupt in diesen Node ?
33:   FLOAT s = node->size*0.5f+0.1f;
34:
35:   if (node->Level>0)
36:   {
37:     if (e->Amin.x<(node->m.x-s)) return FALSE;
38:     if (e->Amin.y<(node->m.y-s)) return FALSE;
39:     if (e->Amin.z<(node->m.z-s)) return FALSE;
40:     if (e->Amax.x>(node->m.x+s)) return FALSE;
41:     if (e->Amax.y>(node->m.y+s)) return FALSE;
42:     if (e->Amax.z>(node->m.z+s)) return FALSE;
43:   }
44:
45:   if (node->Level<=MAXSUBDIVIDE)
46:   {
47:     //in Subnodes einfügen
48:     if (!node->nSubNodes)
49:     {
50:       //Subnodes erst anlegen
51:       FLOAT s = node->size*0.5f;
52:       FLOAT s2 = node->size*0.25f;
53:       for (SINT32 i = 0; i<8; i++)
54:       {
55:         node->SubNode[i] = new OCTREENODE;
56:         node->SubNode[i]->nSubNodes = 0;
57:         node->SubNode[i]->Level = node->Level+1;
58:         node->SubNode[i]->size = s;
59:         node->SubNode[i]->Objects = List<RTObject*>(0);
60:         node->SubNode[i]->ObjectIndex = List<UINT32>(0);
61:         node->SubNode[i]->m = node->m;
62:
63:         if (i & 1) node->SubNode[i]->m.x -= s2;
64:         else node->SubNode[i]->m.x += s2;
65:         if (i & 2) node->SubNode[i]->m.y -= s2;
66:         else node->SubNode[i]->m.y += s2;
67:         if (i & 4) node->SubNode[i]->m.z -= s2;
68:         else node->SubNode[i]->m.z += s2;
69:       }
70:       node->nSubNodes = 8;
71:     }
72:
73:     for (SINT32 i = 0; i<8; i++)
74:     {
75:       if (InsertObject(node->SubNode[i], o, index, e) ==
76:         TRUE) return TRUE;
77:     }
78:   }
79:
80:   {
81:     //Hier einfügen
82:     node->Objects.Add(o);
83:     node->ObjectIndex.Add(index);
84:     return TRUE;
85:   }
86: }
```

Das Octree-Verfahren in *RTOctree.cpp*

2 Prinzip der Schnittpunktberechnung

```
1: Void RecOctree(OCTREENODE *node)
2: {
3:   SINT32 i;
4:
5:   //Trifft der Strahl diesen Würfel überhaupt?
6:   if (RayAABBIntersect(...) == FALSE) return;
7:
8:   //alle Schnittpunkte mit Objekten im Node berechnen
9:   for (i = 0; i<node->Objects.num; i++)
10:  {
11:    SINT32 n =
12:      node->Objects[i]->GetClosestIntersection(*iray);
13:    if (n!=-1) Intersections++;
14:  }
15:
16:  //Nun die Subnodes testen
17:  for (UINT32 j = 0; j<node->nSubNodes; j++)
18:  {
19:    RecOctree(node->SubNode[j]);
20:  }
21: }
22:
23: int GetIntersections(...)
24: {
25:   Intersections=0;
26:
27:   //Objekte im Octree testen
28:   RecOctree(Octree);
29:
30:   //Objekte ohne AABBs
31:   for (i = 0; i<nInfinite; i++)
32:   {
33:     int n=RTScene[i]->GetClosestIntersection(pray);
34:     if (n!=-1) Intersections++;
35:   }
36:
37:   return Intersections;
38: }
39:
```

Pseudocode für die Schnittpunktberechnung