AUF DER HEFT-CD

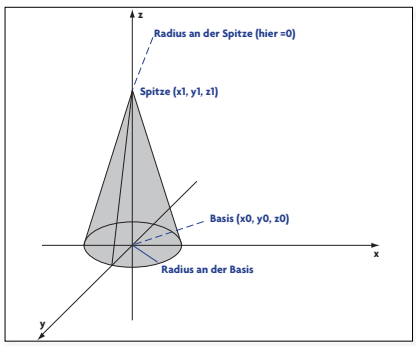
Demo-Programmierung unter Windows 95/98

Mit Kegel und Zylinder

Virtuelle Welten sehen oft künstlich aus. Das muß nicht sein: Dieser Beitrag erweitert Ihren Raytracer, damit sich die künstliche Welt der wirklichen nähert.

CARSTEN DACHSBACHER/
NILS PIPENBRINCK

Mit den Erweiterungen in diesem zweiten Teil unserer Raytracer-Serie zaubern Sie weitaus realistischere Landschaften auf Ihren Bildschirm. Doch bevor Sie die vielen Neuerungen in Ihren Raytracer einbauen und sehen, machen Sie sich an die mathematische Herleitung der Zylinder- und Kegelprimitive. Mit mathematischer Kleinarbeit verfolgen Sie die Lichtstrahlen und ihre verschiedenartigen Spiegelungen.



DIESER KEGEL hat seine Basis im Ursprung des Koordinatensystems und seine Spitze auf der positiven z-Achse.

Als erstes untersuchen Sie einen Kegel, wie ihn der Sourcecode zum Raytracing-Artikel (Heft 10, ab S. 226, auch auf der aktuellen Heft-CD) vorweggenommen hat. Es handelt sich hierbei um einen Spezialfall eines Kegels, bei dem die Basis, also der Mittelpunkt der unteren Kreisfläche, im Ursprung des Koordinatensystems liegt, und die Spitze sich auf der positiven *z*-Achse befindet. Sie können die Schnittpunktberechnungen aller Kegel auf Schnittpunkte mit solch einem Kegel zurückführen. Und so berechnen Sie diesen Kegel:

Wie bisher beschreiben Sie den verfolgten Lichtstrahl durch eine Halbgerade, hier gegeben durch ihren Startpunkt $\vec{g}$ und ihren Richtungsvektor $\partial\vec{g}$, wobei $t > 0$ gilt:

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

Stellen Sie sich nun den Kegel als einen Stapel von Kreisscheiben vor, deren Radius von der Höhe abhängt (Ihrer *z*-Koordinate), in der Sie liegen. So hätte die unterste Kreisscheibe den Radius des Basiskreises und die oberste den Radius *0*.

Da sich der Radius linear ändert, stellen Sie eine Formel auf, die den Radius abhängig von *z* beschreibt. In dieser Formel lassen Sie für die Spitze auch ein Radius ungleich Null zu, was einen Kegelstumpf beschreiben würde:

```
Radius(z)=Radius(Basis)+
➥(Radius(Spitze)-Radius(Basis))
➥* z / z1
```

oder:

```
Radius(z)=Radius(0)+dr*z
```

mit

```
dr=(Radius(z1)-Radius(z0))/z1
```

Den Schnittpunkt mit einer Kreisscheibe bestimmen Sie – fast analog zur Berechnung der Schnittpunkte mit Kugeln – über die Abstandsberechnung. Auf einem Kreis liegen alle Punkte *x*->, deren Abstand vom Mittelpunkt gleich dem Radius des Kreises ist, also:

$$| \vec{m} - \vec{x} | = \vec{d} = \text{radius}$$

Da Sie sich bei einem Kreis im Zweidimensionalen befinden, sind nur die *x*- und *y*-Komponenten von $\vec{m}$, $\vec{x}$ und $\vec{d}$ von Interesse:

$$\vec{d} * \vec{d} = \text{radius}^2$$
$$\vec{d}x*\vec{d}x+\vec{d}y*\vec{d}y=\text{radius}^2$$

Bei unserem Kegel ist der Radius jedoch nicht fest, sondern abhängig von *z*. Daraus ergibt sich:

```
d→x*d→x+d→y*d→y=
➥Radius(z)²
d→x*d→x+d→y*d→y=
➥(Radius(0)+dr*x→z)²
(m→x-x→x)*(m→x-x→x)+
➥(m→y-x→y)*(m→y-x→y)=
➥(Radius(0)+dr*x→z)²
```

Nun ersetzen Sie wieder alle $\vec{x}$ durch die Gleichung der Halbgeraden:

```
(m→x-(g→x+t*dg→x))*
➥(m→x-(g→x+t*dg→x))+
➥(m→y-(g→y+t*dg→y))*
➥(m→y-(g→y+t*dg→y))=
➥(Radius(0)+dr*(g→z+t*dg→z))²
```

Durch Ausmultiplizieren erhalten Sie dann:

```
(m→x-g→x)²-2*
➥(m→x-g→x)*t*dg→x+t²*
➥(dg→x)²+(m→y-g→y)²
➥-2*(m→y-g→y)*t*dg→y+t²
➥*(dg→y)²=
➥(Radius(0)+dr*g→z)²+t²
➥*(dr*dg→z)²+2*t*
➥(dr*dg→z)*(Radius(0)+dr*g→z)
```

Nun fassen Sie alle Terme so zusammen, damit Sie wieder eine quadratische Gleichung erhalten – wie bei der Schnittpunktberechnung mit Kugeln:

$$a*t^2+b*t+c=0$$

mit

```
a=(dg→x)²+(dg→y)²
➥-(dr*dg→z)²-(dr*dg→z)²
b=-2*(m→x-g→x)*dg→x-2*
➥(m→y-g→y)*dg→y-2*(dr*dg→z)*
➥(Radius(0)+dr*g→z)
c=(m→x-g→x)²+(m→y-g→y)²
➥-(Radius(0)+dr*g→z)²
```

a, *b* und *c* sind reelle Zahlen. Die möglichen Lösungen sind:

```
t1=(-b+sqrt(b²-4*a*c))/(2*a);
t2=(-b-sqrt(b²-4*a*c))/(2*a);
```

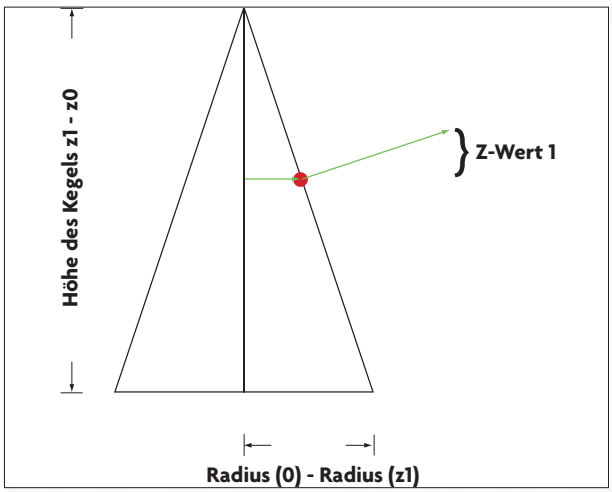
Die Zahl der Lösungen läßt sich durch die Diskriminante

$$D=b^2-4*a*c$$

bestimmen: Für $D < 0$ gibt es keine Lösung, $D = 0$ bringt eine Lösung, und $D > 0$ führt zu zwei Lösungen.

Setzen Sie *t1* und *t2* in die Halbgeradengleichung ein. Denken Sie daran, nur die Position *t1* und *t2* sind interessant. Sie erhalten dann die Schnittpunkte.

Die Berechnung der Oberflächennormalen gestaltet sich komplizierter als bei einer Kugel. Hier berechnen Sie zuerst den Differenzvektor des Schnittpunkts



UM DIE OBERFLÄCHENNORMALE im Kegel zu berechnen, brauchen Sie weitaus mehr Formeln als bei einer Kugel.

(rot eingezeichnet) mit dem Punkt auf der Mittelachse des Kegels, der auf gleicher Höhe liegt, d.h. den gleichen *z*-Wert besitzt:

$\vec{d} = \vec{s} - (0, 0, \vec{s}z)$

Normalisieren Sie diesen Vektor, und skalieren Sie ihn mit folgendem Term:

```
Skalierungsfaktor=
(z1-z0)/(Radius(0)-Radius(z1))
```

Der Skalierungsfaktor beschreibt, wie steil der Kegelmantel ist, also das Verhältnis seiner Höhe zu seiner Breite.

Da eine Normale immer senkrecht zu einer Oberfläche steht, muß die Normale des Kegels flacher sein, wenn der Kegel steiler wird. Das erreichen Sie, indem Sie den Vektor skalieren und die *z*-Komponente des resultierenden Vektors, die an dieser Stelle der Berechnung immer *0* ist, auf *1* setzen.

$\vec{d}=\vec{d}$*Skalierungsfaktor

$\vec{d}z = 1$

Der resultierende Vektor $\vec{d}$ ist die gesuchte Normale.

Am besten ist, Sie versuchen sich die Berechnung der Normalen anhand kleiner Skizzen verschiedener Kegel vorzustellen. Dabei bleibt die Frage, wie Sie alle möglichen Kegel auf den hier hergeleiteten Spezialfall zurückführen. Die Antwort liefert die Matrizenrechnung.

Dazu berechnen Sie eine Matrix, die die Transformation einer anderen Matrix rückgängig macht. Mathematiker bezeichnen dies als eine inverse Matrix.

In *PC Underground, Heft 10/99,* wurden alle Transformationen eines Objekts und die durch die Kamera resultierenden Transformationen in einer Matrix eines jeden Objekts zusammengefaßt (ab S. 212). Diese Matrix benutzen Sie, um zum Beispiel Koordinaten eines Objekts in den Raum der Kamera zu transformieren.

Umgekehrt können Sie eine Matrix zu einem Kegel konstruieren, die einen Punkt im Raum so abbildet, daß Sie den Spezialfall erhalten. Multiplizieren Sie zum Beispiel die Kegelspitze, die Sie frei im Raum plazieren können, mit dieser Matrix, erhalten Sie einen Punkt auf der positiven *z*-Achse; transformieren Sie die Basis des Kegels, erhalten Sie den Ursprung. Man spricht hierbei von einem Wechsel des Koordinatensystems.

Der große Vorteil von Matrizen ist, daß Sie solche Matrizen invertieren können. Sie berechnen also eine Matrix, die genau die umgekehrte Abbildung zur Folge hat. Haben Sie also einen Schnittpunkt im Koordinatensystem des Kegels – wie zuvor beschrieben – berechnet, können Sie ihn mit der invertierten Matrix in seine richtige Position im Raum zurücktransformieren.

Ohne Sie mit den Details der Matrixrechnung zu belasten, lernen Sie hier die Lösungsidee für das Aufstellen der Kegelmatrix kennen.

Die Matrix für die Abbildung in das gewünschte Koordinatensystem des Kegels erhalten Sie, indem Sie drei senkrecht aufeinander stehende Vektoren finden, bei denen im Kreuzungspunkt der Basismittelpunkt des Kegels und dessen Spitze auf der *z*-Achse liegt. Den ersten Vektor erhalten Sie durch Subtraktion des Ortsvektors der Spitze vom Ortsvektor des Basismittelpunkts.

Nun benötigen Sie einen Vektor, der senkrecht zu diesen steht. Eine einfache Methode besteht darin, zwei Komponenten des ersten Vektors zu vertauschen und eine der vertauschten zu negieren. Den dritten Vektor, der senkrecht auf den ersten beiden steht, erhalten Sie durch das Kreuzprodukt dieser Vektoren.

Damit haben Sie eine Matrix, die den Kegel so rotiert, daß er die richtige Orientierung für den Spezialfall besitzt. Jetzt fehlt noch eine Verschiebungsmatrix zur Lösung. Diese verschiebt alle Punkte um das Negative des Ortsvektors des Basismittelpunkts. Die Matrix für den Kegel erhalten Sie also aus der Matrixmultiplikation (Hintereinanderausführung) der Verschiebungs- und Rotationsmatrix.

Mit der Berechnung von Schnittpunkten mit Kegeln haben Sie den allgemeineren Fall kennengelernt. Ein Spezialfall des Kegels ist der Zylinder. Bei einem Zylinder ist der Radius unabhängig von der *z*-Komponente, das heißt die Formeln für die Schnittpunktberechnung vereinfachen sich deutlich. Sie finden das Resultat im Sourcecode. Eine weitere Herleitung ergibt keinerlei Neuerung gegenüber dem Kegel.

Die Implementation des Kegel- und Zylinderprimitivs finden Sie in im Quellcode von *RTCone.cpp* und *RTCylinder.cpp*.

Licht aus endlichen Quellen

In *PC Magazin 10/99* haben Sie ab S. 212 den einfachsten Fall von Lichtquellen beim Raytracing kennengelernt. Dabei handelte es sich um Lichtquellen, die von einem Punkt aus Licht in alle Richtungen aussenden, ohne selbst eine lichtemittierende Fläche zu besitzen.

Doch diese Vorstellung entspricht nicht den real vorhandenen Lichtquellen. Der deutlichste Unterschied, den punktförmige und sich ausdehnende Lichtquellen besitzen, ist, daß punktförmige immer einen harten Schatten werfen, während Sie in der Realität fast immer weiche Schatten vorfinden.

Wie Sie eine solche Lichtquelle in einen Raytracer implementieren, zeigt Ihnen dieser Teil. Dabei ändern Sie die rekursive Raytracing-Prozedur nur an zwei Stellen.

Die erste Stelle ist die Schattenberechnung. Es genügt nun nicht mehr, einen Schattenstrahl auszusenden, sondern Sie benötigen mehrere – besser gesagt viele.

Bei einem Schattentest prüfen Sie die Sichtbarkeit der Lichtquelle mit verschiedenen, möglichst gleichmäßig auf der Lichtquelle verteilten Punkten. Der Schattentest eines einzelnen Punkts funktioniert genauso wie bei punktförmigen Lichtquellen. Die resultierende Farbe ergibt sich durch Mittelung der Schattenfarben aller Schattenstrahlen.

Im Pseudocode sieht die Vorgehensweise wie folgt aus:

```
for (viele Schattenstrahlen)
{Wähle Punkt auf der Lichtquelle
Berechne Schattenfarbe für _
diesen Punkt per Schattentest
Kumuliere Schattenfarbe}
Resultierende Schattenfarbe =
Kumulierte Schattenfarbe /
➥Anzahl Schattenstrahlen
```

Die Auswahl der Zielpunkte auf der Lichtquelle erledigen Sie mit einem Zufallsgenerator, wobei die Punkte möglichst gut verteilt liegen sollten. Das erreichen Sie, indem Sie jeden bereits berechneten Punkt speichern.

Benötigen Sie dann einen neuen Zufallspunkt, generieren Sie mit einem Zufallszahlengenerator mehrere neue Kandidaten, nehmen aber nur denjenigen, der von allen Zufallspunkten den größten Abstand besitzt. Je mehr Kandidaten Sie zulassen, desto besser wird die zufällige Verteilung.

Da Sie aber nur im Halbschatten viele Schattenstrahlen benötigen, um ein gutes visuelles Ergebnis zu erhalten, definieren Sie ein Abbruchkriterium. Dieses stellt sicher, daß weitere Schattenstrahlen getestet werden, wenn die Helligkeit der bisherigen zu stark variiert. Umgekehrt lassen Sie zu, daß der Schattentest endet, wenn die Schattenstrahlen alle ähnliche Helligkeit besitzen.

Aus diesen Überlegungen ergibt sich als mögliche Formel:

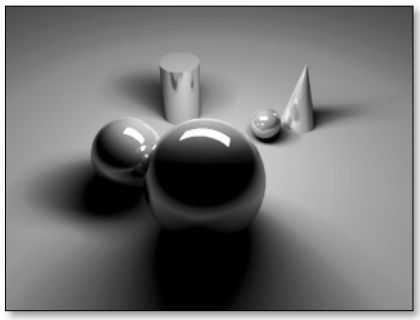
```
max = Max. aller Helligkeiten
min = Min. aller Helligkeiten
if ((max-min)/(max+min)>
Toleranzschwelle) mehr Strahlen;
 else Ende;
```

Die Funktionen dazu, die von den Schattenfarben auch die Rot-, Grün- und Blaukomponenten separat behandeln, finden Sie in der Datei *RTFunctions.h*. Zusätzlich zu diesem Abbruchkriterium legen Sie noch eine mindeste und eine maximale Anzahl von Schattenstrahlen fest.

Bedenken Sie, daß eine gute Darstellung von solchen Lichtquellen gewaltigen Rechenaufwand erfordern kann. Die Werte für die minimale Zahl von Schattenstrahlen liegen zwischen *4* und *8*, die der maximalen Zahl bei *100* oder mehr.

Der zweite Punkt, an dem Sie in der Raytracing-Prozedur Hand anlegen, ist die Berechnung der Glanzlichter. Bei punktförmigen Lichtquellen haben Sie das einfallende Licht an der Oberfläche eines Körpers gespiegelt. Die Intensität des Glanzlichts hing dann davon ab, wie genau der Betrachter von diesem gespiegelten Strahl getroffen wurde.

Nun gehen Sie den umgekehrten Weg. Sie spiegeln den einfallenden Lichtstrahl und testen, ob dieser Lichtstrahl einen Schnittpunkt mit der Lichtquelle besitzt. Wenn ja, ist die Intensität des Glanzlichts maximal und seine Farbe die der Lichtquelle. Gibt es keinen Schnittpunkt, existiert für den betrachteten Oberflächenpunkt kein Glanzlicht.



DIESE SZENE beleuchtet eine rechteckige Lichtquelle mit schön ausgedehntem Halbschatten.

An dieser Stelle ist interessant, welche Form die neue Lichtquelle besitzt. Als Beispiel für diesen Raytracer haben wir eine rechteckige Lichtquelle verwendet. Die Schnittpunktberechnung, die Sie für die Glanzlichter verwenden, heißt *BOOL IntersectTriangle(...)* und befindet sich in *RTFunctions.h*. Eine entsprechende Herleitung nimmt die nächste Ausgabe in Angriff, die auch polygonale Primitive und aus Polygonen zusammengesetzte Objekte behandelt.

Texturen & Bumpmapping

Nachdem Sie sich mit einer Reihe von geometrischen Primitiven beschäftigt und auch einiges an Aufwand mit den Lichtquellen getrieben haben, widmen Sie sich nun den Oberflächen der Objekte. Bisher haben Sie Oberflächen durch ihre physikalischen Parameter, wie den Reflektionskoeffizienten oder die Transparenz, und durch ihre Eigenfarbe definiert.

Damit konnten Sie nur die Eigenschaften für die ganze Oberfläche bestimmen. Viel interessanter gestaltet sich eine Szene, wenn Sie die Objekte mit Texturen belegen oder ihren Oberflächen eine aufgerauhte Struktur oder Beulen verpassen.

Prinzipiell können Sie Objekte mit zwei Arten von Texturen belegen:

- Das eine Verfahren projeziert eine zweidimensionale Bitmap auf einen Körper. So arbeiten 3D-Beschleuniger sowie die auf Polygonen basierenden 3D-Engines.
- Bei Raytracern verwenden Sie hingegen fast immer sogenannte prozedurale Texturen. Hierbei setzen Sie keine Bitmaps ein, sondern berechnen die Farbe von Punkten im Raum.

Im Zusammenhang mit prozeduralen Texturen und Raytracern fällt immer der Begriff Perlin Noise. Dieses Verfahren, das Ken Perlin entwickelt hat, setzen Experten häufig ein, um Texturen zu synthetisieren. Besuchen Sie Ken Perlin auf seiner Homepage:

`http://mrl.nyu.edu/perlin`

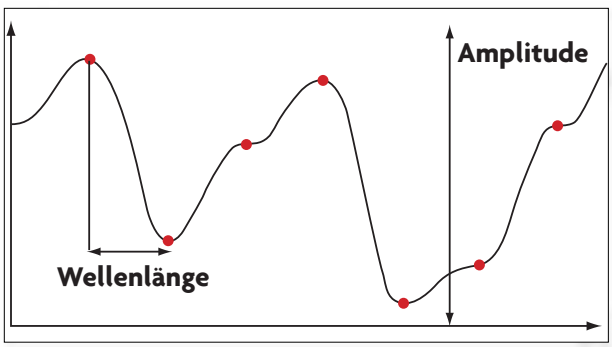
Perlin Noise

Wenn Sie sich Dinge in der Natur ansehen, stellen Sie fest, daß es verschiedene Detailstufen gibt, ähnlich wie bei Fraktalen. Ein anschauliches Beispiel ist ein Gebirge. Es enthält große Höhenunterschiede wie Berge, mittlere wie Hügel, kleine wie Felsbrocken und winzige Details wie Steine.

Noise bedeutet so viel wie Lärm oder Rauschen. Eine Noise-Funktion ist eine Funktion, die zu einem Parameter – in diesem Fall eine ganze Zahl – eine zufällige Zahl zurückliefert. Wenn Sie zweimal denselben Parameter übergeben, muß sie auch zweimal dasselbe Resultat erzeugen, sonst funktioniert das Perlin-Noise-Verfahren nicht.

Perlin Noise ahmt die in der Natur vorkommenden Detailstufen nach, indem es unterschiedlich skalierte Noise-Funktionen nach einem System addiert.

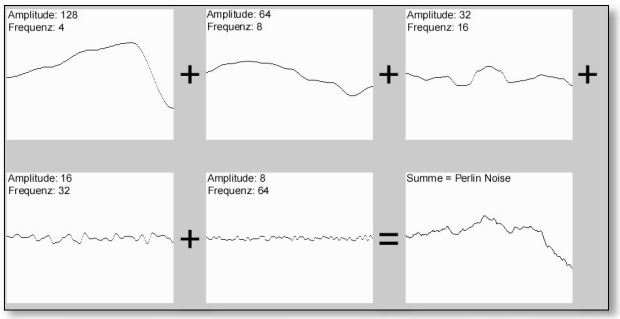
Beachten Sie bei der im Bild unten dargestellten Noise-Funktion, daß nur die roten Punkte generierte Zufallszahlen sind. Alle Zwischenwerte sind durch



BEI DER NOISE-FUNKTION sind die roten Punkte durch Zufallszahlen, die Zwischenwerte durch Interpolation entstanden.

Interpolation entstanden. Die Wellenlänge bezeichnet den Abstand zweier Zufallszahlen. Die Frequenz berechnet sich – analog zu Wellen in der Physik – als Kehrwert der Wellenlänge.

Addieren Sie mehrere Noise-Funktionen, wie im Bild unten zu sehen, erhalten Sie die Perlin-Noise-Funktion. Dasgleiche können Sie auch im zweidimensionalen Raum tun. Dazu benötigen Sie nur eine Noise-Funktion, die zu einem Zahlenpaar einen Zufallswert liefert. Das Ergebnis sehen Sie im Bild, das schattierte Grünflächen zeigt.



DIE SUMME aus diesen vier Noise-Funktionen ergibt im letzten Bild unten rechts die Perlin-Noise-Funktion

Die Amplitude und die Frequenz, die Sie für die einzelnen Noise-Funktionen verwenden, legen Sie durch die sogenannte Persistence fest. Sie bestimmen noch eine Amplitude und eine Frequenz für die erste Funktion. Für die jeweils nächste Funktion verdoppeln Sie die Frequenz und multiplizieren die Amplitude mit der Persistence.

Der Wert der Persistence sollte zwischen *0* und *1* liegen. Größere Werte bedeuten stärkere hohe Frequenzen, also mehr Rauschen. Bei der Anzahl der Funktionen, die überlagert werden, spricht man auch von Oktaven. Der Begriff wurde aus der Musik entliehen, da bei Klängen eine Verdopplung der Frequenz einem Sprung von einer Oktave entspricht.

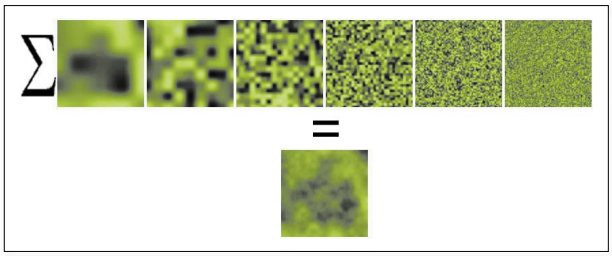
Wieviel Oktaven Sie wählen, ist Ihre Entscheidung. Berücksichtigen Sie nur, daß die Amplitude irgendwann so klein wird, daß die Funktion nicht mehr ins Gewicht fällt. In unserem Fall empfehlen sich etwa zwei bis acht Oktaven.

Wie erzeugen Sie Noise-Funktionen? Die herkömmlichen Zufallszahlengeneratoren, die Ihnen in C zur Verfügung stehen, liefern bei jedem Aufruf eine neue Zahl. Da das Ergebnis jedoch reproduzierbar sein muß, weil Sie eine Noise-Funktion eventuell mehrmals an derselben Stelle berechnen müssen, können Sie diese nicht verwenden.

Eine Möglichkeit besteht darin, eine Funktion zu finden, die relativ zufällig Werte liefert. Solche Funktionen enthalten meist sehr große Primzahlen. Ein Beispiel mit Zufallszahl zwischen *-1* und *1* zu *x* sieht so aus:

```
x = (x<13) ^ x;
return (1.0-
((x*(x*x*15731+
789221)+1376312589) &7fffffff)
/1073741824.0);
```

Ein anderer Lösungsweg: Legen Sie beim Start des Programms eine Tabelle mit Zufallszahlen mit Hilfe Ihres herkömmlichen Generators an. Es genügen 4096 verschiedene Zahlen. Als Funktion dient dann



DIE PERLIN-NOISE-FUNKTION im zweidimensionalen Raum erinnert hier an Flora.

```
return Zufallstabelle
[x % Größe der Tabelle ];
```

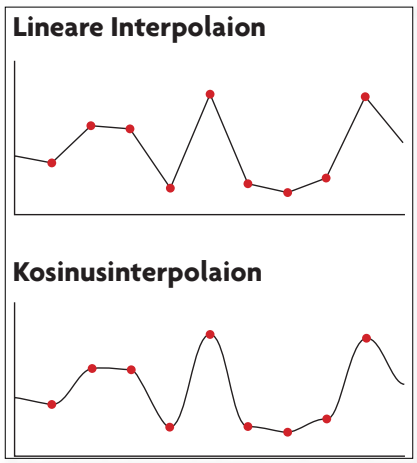
Wenn Sie nun einen Zufallswert zu nicht ganzzahligen *x*-Werten benötigen, erledigen Sie dies durch Interpolation.

Dazu werten Sie die Noise-Funktion an der von *x* aus gesehen nächstkleineren und nächstgrößeren ganzen Zahl aus. Mit dem Nachkommaanteil von *x* berechnen Sie den interpolierten Wert.

Die einfachste Methode ist es, linear zu interpolieren:

```
return  Wert1*(1-NachkommaX)
+ Wert1*NachkommaX
```

Bei der linearen Interpolation erhalten Sie jedoch keine sehr schönen Ergebnisse. Mit ein wenig mehr Rechenaufwand erzielen Sie mit der Kosinusinterpolation abgerundetere Ergebnisse:



DIE KOSINUSINTERPOLATION liefert realistischere Bilder als die lineare.

```
ft = NachkommaX * PI
f = (1 - cos( ft ) ) * 0.5
return  a*(1-f) + b*f
```

Der Unterschied ist, daß der Gewichtungsfaktor (hier *f*) bei der Kosinusinterpolation – durch die Kosinusfunktion an den Rändern (sprich bei Nachkommaanteilen nahe bei *0* oder *1* – langsamer steigt. Dadurch erhalten Sie in der Nähe der Zufallswerte abgerundete Verläufe.

Nehmen Sie nun alles zusammen, erhalten Sie eine Perlin-Noise-Routine wie im folgenden Pseudocode für eine Dimension:

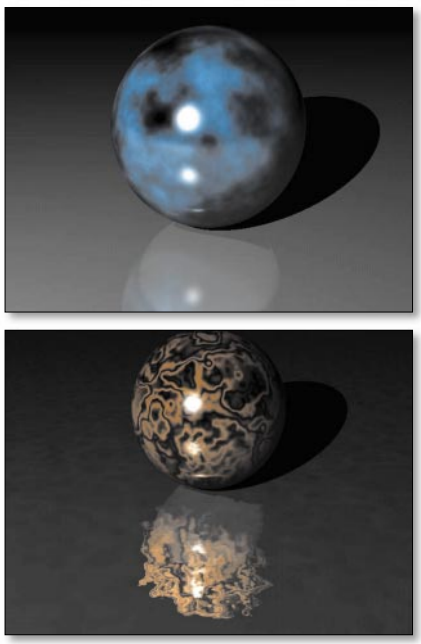
```
int Noise( int x )
{return Zufallstabelle
 [ x % Größe der Tabelle ];}
float Interpolate
 (float a,float b, float x )
{return  a*(1-x) + b*x;}
float InterpolatedNoise(float x)
{GanzzahlX = (int)x;
NachkommaX = x - GanzzahlX;
v1 = Noise( x );
v2 = Noise( x + 1 );
return Interpolate
 (v1,v2,NachkommaX);}
float PerlinNoise( float x )
{floattotal= (FLOAT)0;
floatPersistence= material.p;
int   Octaves= material.o;
floatFrequenz= material.f;
floatAmplitude=
 material.a * Persistence;
for (int i= 0;i<Octaves;i++)
{total += InterpolatedNoise3D
( x * Frequenz ) * Amplitude;
Frequenz  *= 2.0;
Amplitude *= Persistence;
 }
}....
```

Für dreidimensionale Noise-Funktionen berechnen Sie nicht zwei Punkte und interpolieren, sondern Sie berechnen acht Werte. Ein Punkt liegt im Dreidimensionalen innerhalb eines Würfels, dessen Kanten durch die jeweiligen nächstkleineren bzw. -größeren ganzzahligen Koordinaten bestimmt sind. Berechnen Sie die Werte für die Eck- ▶

punkte des Würfels, und interpolieren Sie anschließend.

Was haben Sie von einer 3D-Noise-Funktion? Sie können zu jedem Punkt im Raum einen Farb- oder Helligkeitswert bestimmen. Wollen Sie ein Objekt mit einer prozeduralen Textur versehen, speichern Sie in der Materialstruktur des Raytracers die Werte für die Noise-Funktionen und berechnen für jeden Schnittpunkt den Noise-Wert. Damit erzeugen Sie Texturen wie auf den unten abgebildeten Kugeln.

Mit den Noise-Werten berechnen Sie auch andere Texturen, zum Beispiel eine



MIT EINER 3D-NOISE-FUNKTION sehen diese Kugeln zum Greifen echt aus.

marmorähnliche Struktur. Hierzu berechnen Sie für jeden Raumpunkt zwei Noise-Werte: einen an seiner Originalposition und einen an einem Punkt, den Sie durch eine beliebig große Verschiebung erreichen.

Dann nehmen Sie die *x*-Komponente des Originalvektors und addieren den ersten Noise-Wert dazu. Genauso verfahren Sie mit der *y*-Komponente und dem zweiten Wert.

Die Nachkommastellen der resultierenden Werte bilden Sie mit der Funktion *cycloidal(...)* auf eine Sinusfunktion ab und multiplizieren das Ergebnis mit dem Turbulenzparameter, den Sie noch in der Materialbeschreibung einführen. Der Rückgabewert der Noise-Funktion entspricht jetzt nur der Länge des Ergebnisvektors, auf eine Dreiecksfunktion umgelegt:

```
{...}
floatcycloidal( float x )
{float temp = fmod( x, 1 );
  if (temp < 0) temp += 1;
 return sin(temp * 2 * PI );}
floattriangle_wave( float x )
{float offset = fmod( x, 1 );
if (offset < 0) offset += 1;
if (offset>0.5) offset=1-offset;
return offset + offset;}
{...}
// Marmor Wert am Punkt p
r = Noise( p.x, p.y, p.z );
r2 = Noise( p.x+1000, p.y,p.z);
p.x += cycloidal(p.x + r)
 * PTexture.Turbulenz;
p.y += cycloidal(p.y + r2)
 * PTexture.Turbulenz;
return triangle_wave
 (VLength(p));
{...}
```

Es gibt noch unzählige Wege, um die Noise-Werte zu anderen prozeduralen Texturen zu verknüpfen. Im Sourcecode (*RTNoise.cpp*) finden Sie eine Variante für holzähnliche Muster.

Wie Sie die Helligkeit der Farbe an einem Schnittpunkt mit prozeduralen Texturen verändern können, so modifizieren Sie auch die Oberflächennormale an einem Schnittpunkt, um die Beleuchtung, Spiegelung und Lichtbrechung zu beeinflussen. Hierzu berechnen Sie für einen Schnittpunkt drei Noise-Werte, einen an der Originalposition und zwei an verschobenen Stellen. Interpretieren Sie diese Werte als Vektor, skalieren Sie ihn, und addieren Sie den Vektor auf die Normale, die Sie im Zuge der Schnittpunktberechnungen erhalten haben. Als Ergebnis erhalten Sie die Kugeln rechts im Bild.

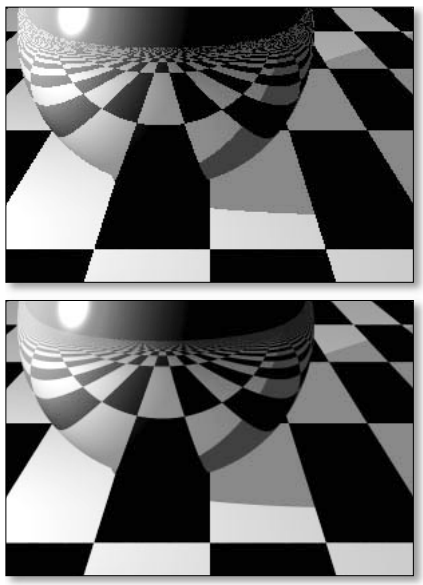
Anti-Aliasing

Ein Problem sind die Aliasing-Effekte – auch als Treppenstufen bezeichnet. Als Aliasing wird das irrtümliche Erscheinen von niederfrequenten Signalen bezeichnet, das aus fehlerhaftem Messen von hochfrequenten Signalen resultiert.

Anders ausgedrückt: Wenn Sie sich den Bildschirm als Fläche vorstellen, repräsentiert jeder Pixel einen kleinen Teil der Gesamtfläche. Verfolgen Sie nur einen Lichtstrahl pro Pixel zurück, können Sie durch zu geringes Abtasten Treppcheneffekte bekommen.

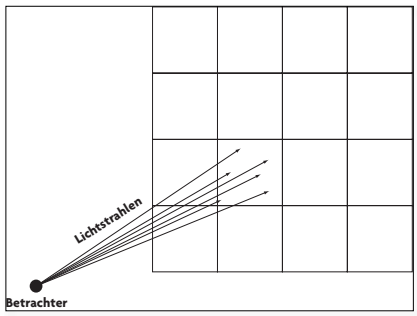
Sie lösen das Problem, indem Sie einen Pixel als Fläche behandeln und mehrere Strahlen durch diesen Pixel verfolgen. Ein Ansatz für das Anti-Aliasing ist das *statistische Super-Sampling*. Hierbei verfolgen Sie Strahlen, die vom Betrachter aus durch Punkte verlaufen, die zufällig auf der Fläche des Pixels verteilt sind.

Achten Sie dabei darauf, daß diese Punkte, wie schon bei den Schattenstrahlen der Halbschatten, möglichst gleich verteilt sind. Auch die Abbruchkriterien sind dieselben wie die bereits vorgestellten. Da das Verfolgen der Stahlen der rechenintensive Teil des Raytracing ist, ist der Preis für das Anti-Aliasing hoch, aber die deutlich bessere Darstellungsqualität rechtfertigt dies.



DAS ANTI-ALIASING-VERFAHREN beseitigt unschöne Treppcheneffekte.

Die vorgestellten Neuerungen bei den Texturen und dem Bump Mapping im Raytracing-Programm sind auch in den Parser integriert. Die Parameter



SIE BEHANDELN einen Pixel als Fläche und verfolgen mehrere Strahlen durch diesen Pixel.

für das Abtasten der Lichtquellen und das Anti-Aliasing finden Sie in den Quelldateien *RTPolylight.cpp* und *RTCamera.c*. ET

Die kompletten Quelltexte finden Sie auf der Heft-CD im Verzeichnis *Praxis\PC-Underground* und auf unserer Website unter
`www.pc-magazin.de/magazin/➥extras.htm`
Klicken Sie in der Tabelle *Online Extras* unter *Praxis* auf das entsprechende *Download*-Feld.