RAYSTAR

for Windows 95™

Dokumentation

Stand : 12. Juni 1995

Inhaltsverzeichnis

1 Einleitung

RayStar™ ist ein auf Windows 95™ / Windows NT™ basiertes Tool zum schnellen, einfachen und komfortablen Generieren von dreidimensionalen photorealistischen Bildern.

1.1 Systemvoraussetzungen

Folgende Minimalausstattung an Hard- und Software wird von **RayStar™** benötigt:

- 386DX/33 Prozessor (mathematischer Koprozessor dringend empfohlen!)
- 4 MB RAM
- 3 MB freier Festplattenplatz
- Graphikkarte mit einer Auflösung von 640x480 Punkten bei 256 Farben
- Maus
- MS-Windows 95

Zum optimalen Arbeiten ist folgende Ausstattung empfehlenswert:

- 486DX2/66 Prozessor oder höher
- 12 MB RAM
- High- oder True-Color Graphikkarte mit Graphikbeschleuniger
- MS-Windows 95™ oder MS-Windows NT 3.51™

1.2 Features

- Die **RayStar™**-Sprache ist eine echte Untermenge des Quasi-Standards **POV-Ray™**
- Integrierter Szenen-Editor
- Durchgehende 32-Bit Implementierung
- Echtes Multithreading beim Berechnen der Bilder
- Schnellvorschau
- Anzeige der Bilder während des Berechnens
- Export der Bilder in den Graphikformaten TIFF, GIF, PCX und BMP

2 Ray-Tracing Grundlagen

Die 3D-Computergraphik ist aus der Aufgabenstellung entstanden, wirklichkeitsgetreue, räumliche Bilder von Objekten zu erzeugen, ohne daß diese Objekte tatsächlich physikalisch vorhanden seien müssen. Zur Berechnung von photorealistischen Bildern hat sich das Ray-Tracing Verfahren und seine Weiterentwicklungen als eines der Standard-Verfahren etabliert.

Man steht bei der Erzeugung von Computergraphiken vor dem Problem, daß man eine Anzahl von geometrischen Objekten auf geeignete Weise im Computer repräsentiert hat und daraus nun eine ‘Photographie’ dieser Szene von einer gegebenen Kameraposition aus berechnen muß. Die gedachte ‘Photographie’ würde physikalisch dadurch entstehen, das an jedem Bildpunkt ein Lichtstrahl einer bestimmten Farbe eintrifft, der von irgendeiner Lichtquelle in der Szene ausgesendet worden ist und gegebenenfalls von einem oder mehreren Objekten in der Szene reflektiert und gebrochen worden ist.

Das Ray-Tracing Verfahren macht sich jetzt die Tatsache zunutze, daß der Weg der einzelnen Lichtstrahlen umkehrbar ist, d.h. aus der Richtung eines am Brennpunkt der gedachten Kamera eintreffenden Lichtstrahls läßt sich dessen Weg theoretisch bis zu seinem Ursprung in einer Lichtquelle zurückverfolgen (‘*back-tracing’*). Somit läßt sich die Farbe eines eintreffenden Lichtstrahls aus der Farbe der ihn aussendenen Lichtquelle und aus den Farben und Oberflächeneigenschaften der Objekte, die der Lichtstrahl auf seinem Weg zur Kamera passiert hat, berechnen.

Aus diesen Überlegungen heraus läßt sich sich sofort der grundlegende Ray-Tracing Algorithmus formulieren:

```
for ( each scan line in image )
        for  ( each pixel in scan line )
        {
                ray = determine ray from center of projection through pixel;
                pixel_color  = TraceRay(ray,1);
        }
```

Es wird dabei durch jedes Pixel in der Bildebene ein Strahl, meist ‘Sehstrahl’ genannt, ausgeschickt und dessen Farbe durch die Funktion *TraceRay* berechnet. Dazu ermittelt *TraceRay* dasjenige Objekt, welches der Strahl auf seinem Weg als erstes schneidet. Dann liefert *TraceRay* als Farbe des Strahls die Oberflächenfarbe diese Objekts an dem Punkt zurück, an dem der Sehstrahl das Objekt schneidet.

```
color TraceRay ( ray , depth )
{
        closest_object = determine closest intersection of ray with an object;
        if ( closest_object != NULL )
        {
                point = intersection point of ray with closest_object;
                normal = compute normal for closest_object at point;
                return CalcLighting ( closest_object, ray, point, normal, depth );
        }
        else
                return background_color;
}
```

Aus diesem Algorithmus wird sofort ersichtlich, daß die einzigen Informationen, die aus den Geometriedaten eines Objekts berechnet werden müssen, die Schnittpunkte des Objektes mit einem gegebenen Strahl und die Normalenvektoren des Objektes an den jeweiligen Schnittpunkten sind. Diese beiden Berechnungen lassen sich für viele geometrische Grundobjekte wie z.B. Kugeln, Quader, Ebenen sehr einfach und schnell berechnen. Daher werden üblicherweise die zu berechnenden Szenen aus solchen elementaren ‘Bausteinen’ zusammengebaut.

Die eigentliche Farbe des Sehstrahl wird durch die Funktion *CalcLighting* berechnet. Das dabei verwendete Beleuchtungsmodell ist das *Phong-Modell* nach PHONG BUI-TONG. Danach wird die Farbe des von einer Oberfläche abgestrahlten Lichts aus folgenden Komponenten zusammengesetzt:

- Ambiente Beleuchtung
- Diffuse Beleuchtung
- Spiegelnd reflektiertes Licht
- Schlaglichtern auf glänzenden Oberflächen
- Durchscheinendes Licht bei transparenten Objekten

Diese einzelnen Farbkomponenten werden in *CalcLighting* berechnet und gewichtet addiert als Oberflächenfarbe zurückgeliefert:

```
color  CalcLighting ( object, ray, point, normal, depth )
{
        color  =  ambient color of object;
        for ( each lightsource in scene )
        {
                shadow_ray  =  ray from point  to lightsource;
                if  (  dotproduct of direction of shadow_ray and normal is positiv )
                {
                        if  ( shadow_ray does not intersect with any object in the scene )
                                color  +=  compute diffuse and specular reflected light;
                }
                if  ( depth  < maxTracingDepth )
                {
                        if  (  object  ist reflective  )
                        {
                                reflected_ray  =  ray in reflection direction from point;
                                reflected_color =  TraceRay ( reflected_ray, depth + 1);
                                color  +=  scale reflected_color by object properties;
                        }
                        if (  object  ist transparent )
                        {
                                trans_ray  =  ray in refraction direction from point;
                                trans_color  =  TraceRay ( trans_ray, depth + 1);
                                color  +=  scale trans_color by  object properties;
                        }
                }
        }
        return  color;
}
```

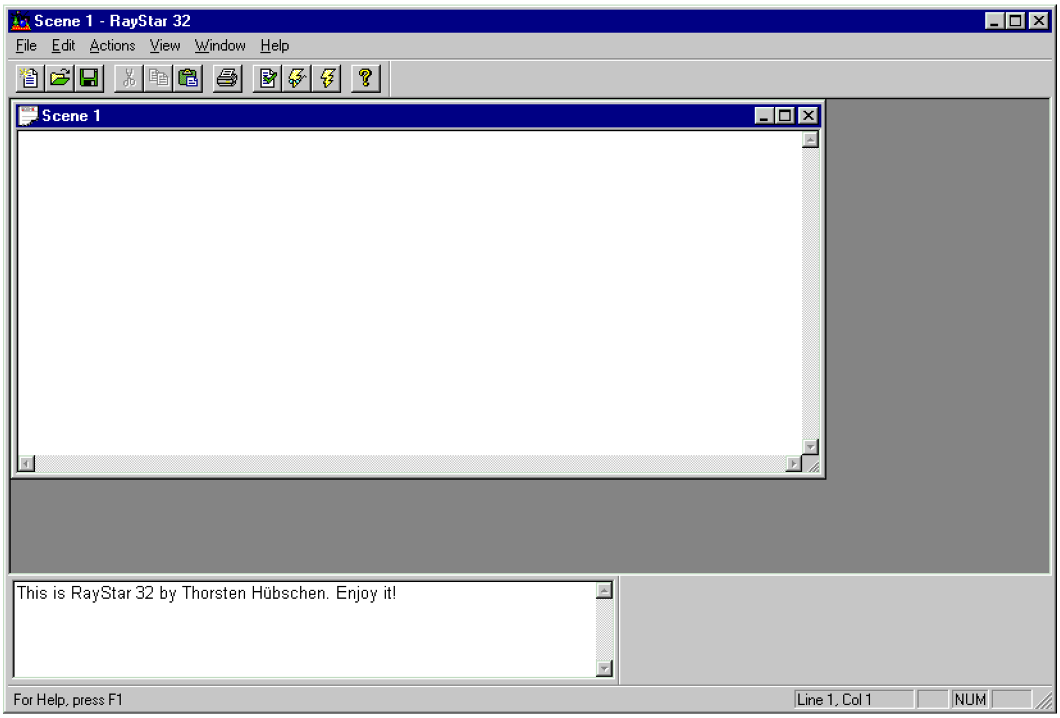
Wie dem Algorithmus zu entnehmen ist, müssen bei der Berechnung der Farbe eines Oberflächenpunktes bis zu zwei neue Strahlen ausgeschickt und deren Farben berechnet werden. Darüber hinaus muß auf jeden Fall für jede Lichtquelle überprüft werden, ob sie nicht von irgend einem anderen Objekt verdeckt wird.

Man sieht sofort, daß der Ray-Tracing Algorithmus zwar keine sonderlich komplizierten oder zeitaufwendigen Berechnungen beinhaltet, daß aber die einzelnen Routinen **sehr** häufig durchlaufen werden müssen. Schon bei einer noch recht geringen Anzahl von Bildpunkten und einer aus wenigen Objekten bestehenden Szene müssen u.U. schon mehrere Millionen Schnittpunkte und Normalenvektoren, ganz zu schweigen von Texturwerten (siehe 4.6), berechnet werden.

Das Ray-Tracing Verfahren ist in seiner Grundform sehr einfach zu implementieren und liefert fazinierend gute, photorealistische Bilder, ist aber bei größeren Szenen sehr rechenaufwendig. Man versucht daher durch ausgefeilte Optimierungsansätze die Anzahl der Berechnung zu minimieren. Methoden dazu findet man in [5],[19].

3 Quick-Tour

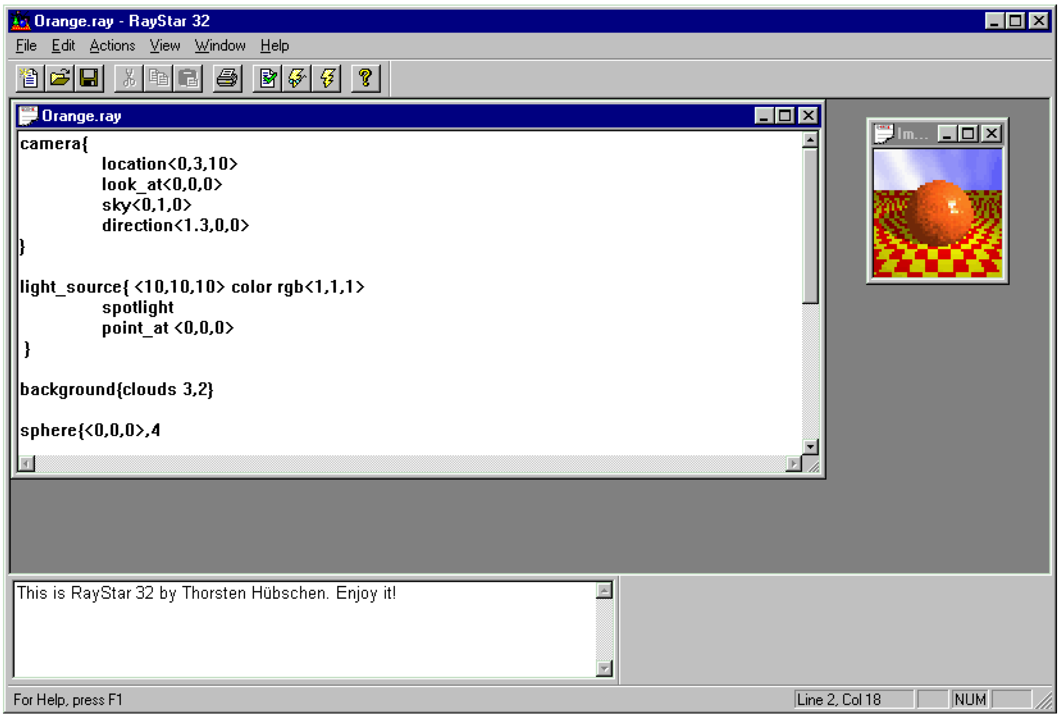
Nach dem Start von **RayStar™** erscheint das Applikationsfenster von **RayStar™** mit einem geöffneten neuen Szeneneditor-Fenster.



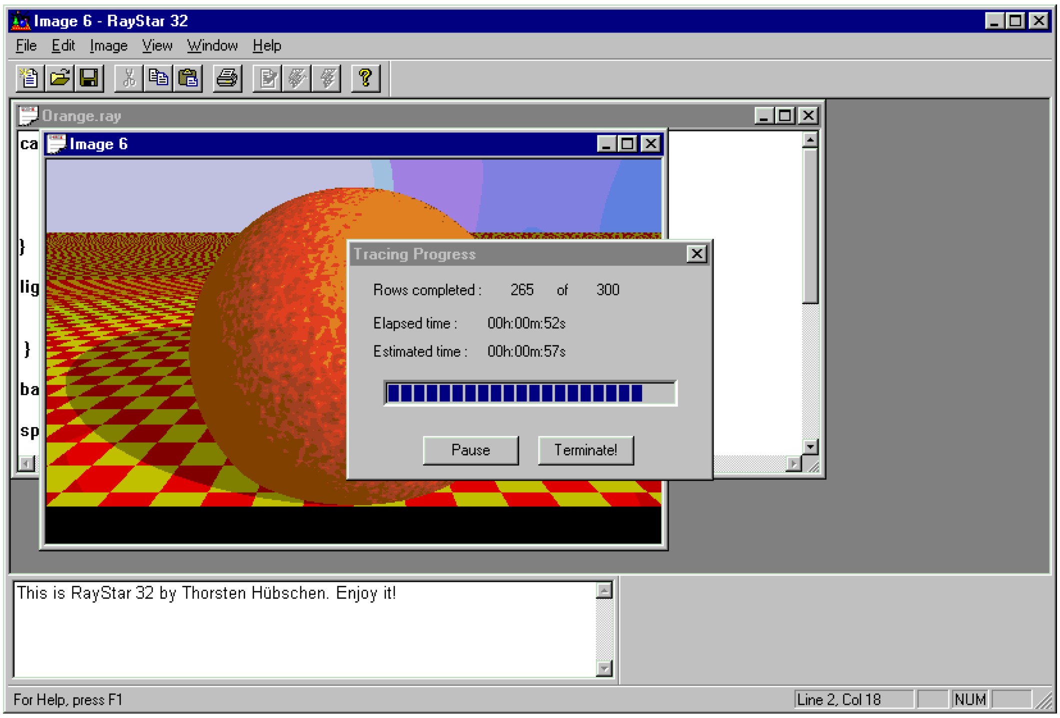
In diesem Editorfenster kann jetzt eine Szenenbeschreibung eingegeben werden. Wir starten mit folgendem einfachen Beispiel:

```
camera{
      location<0,3,10>
      look_at<0,0,0>
      sky<0,1,0>
      direction<1.3,0,0>
}

light_source{ <10,10,10> color rgb<1,1,1>
      spotlight
      point_at <0,0,0>
 }

background{clouds 3,2}

sphere{<0,0,0>,4
      texture{
            pigment{color rgb<0.9,0.35,0.1>}
            finish{ambient 0.5  diffuse 1 phong 1 phong_size 25}
            normal{bumps 0.3 scale 0.1}
      }
}
plane{<0,1,0>,-2.1
      texture{
            pigment{checker color red 1.0 color rgb<1,1,0>}
            finish{ambient 0.5 phong 1}
      }
}
```

Nachdem die Szenenbeschreibung eingegeben ist, kann schon direkt eine Voransicht der fertigen Szene mittels des Befehls ACTIONS-PREVIEW SCENE nahezu in Echtzeit berechnet werden.

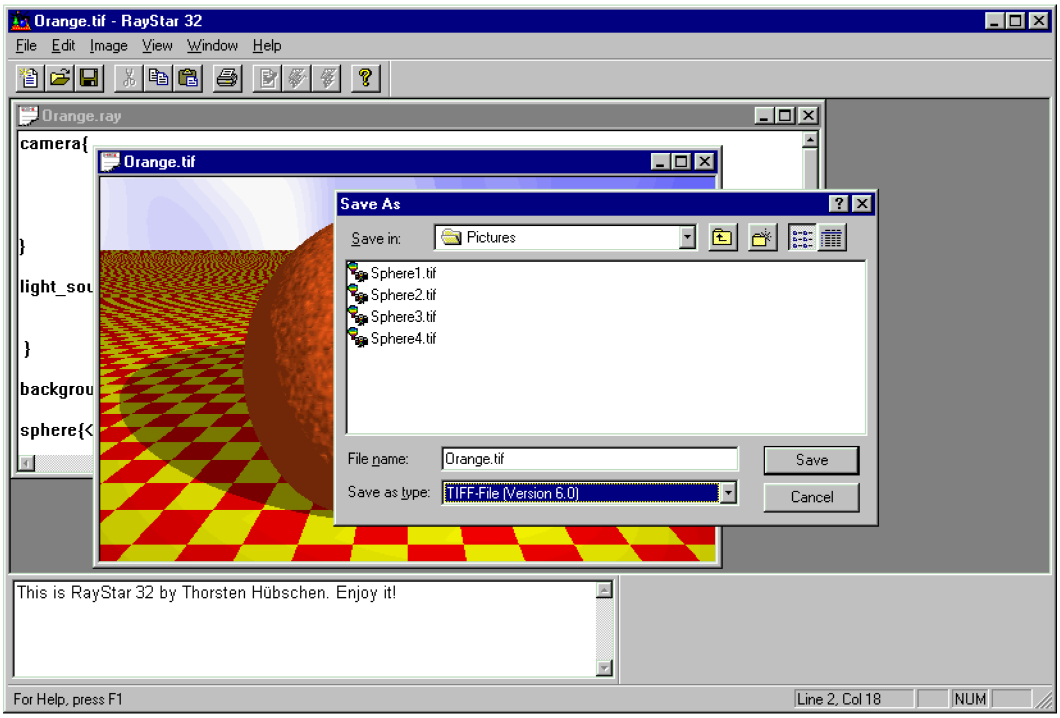


Ist man mit den Szeneneinstellungen, so wie sie in dem Vorschaufenster zu sehen sind, zufrieden, kann nun das endgültige Bild in der gewünschten Auflösung mit dem Befehl ACTIONS-START TRACING... berechnet werden.



Während des Berechnens werden in der Dialogbox TRACING PROGRESS einige Daten über den Fortschritt des Tracing-Prozesses angezeigt.

Ist das Bild fertig berechnet, kann es mit dem Befehl FILE-SAVE in einem gewünschten Graphikformat abgespeichert werden.



Die abgespeicherten Bilder können jederzeit wieder mit dem Befehl FILE-OPEN geladen und unter einem neuen Namen oder in einem anderen Graphikformat abgespeichert werden.

4 Die Szenen-Beschreibungssprache

4.1 Überblick

Eine Szenenbeschreibung besteht in **RayStar™** aus einer Textdatei, in der hintereinander die einzelnen Bestandteile und Parameter der gewünschten Szene definiert werden. Die Definition eines Elements hat dabei im Allgemeinen die Form:

element-name* { *element-parameter *options* **}**

Dabei ist ***element-name*** eines der in den nächsten Abschnitten beschriebenen Schlüsselwörter (z.B. **sphere** oder **pigment**), welches den Type des zu definierenden Element angibt. Die individuellen Parameter des neuen Elements (z.B. Radius und Mittelpunkt bei einer Kugel) werden durch ***element-parameter*** beschrieben, welches meist ein oder mehrere Vektoren oder Fließpunktzahlen sind. Danach können einer Elementdefinition i.a. noch zusätzliche Optionen, wie z.B. Transformation oder Oberflächeneigenschaften, zugewiesen werden.

4.1.1 Schlüsselwörter

Folgende Wörter werden von **RayStar™** als Schlüsselwörter verwendet:

sphere	**box**	**plane**	**camera**
location	**look_at**	**sky**	**direction**
up	**right**	**color**	**red**
green	**blue**	**rgb**	**rgbf**
light_source	**spotlight**	**point_at**	**radius**
falloff	**ambient**	**diffuse**	**reflection**
filter	**phong**	**phong_size**	**specular**
roughness	**clouds**	**pigment**	**texture**
finish	**metallic**	**checker**	**tightness**
background	**scale**	**translate**	**rotate**
fade	**cone**	**open**	

Wenn zwei Schlüsselwörtern direkt auseinander folgen, müssen sie durch mindestens ein Whitespace voneinander getrennt werden. Als Whitespaces werden in **RayStar™** folgende Zeichen interpretiert:

- Leerzeichen
- Horizantale Tabulatoren ('\t')
- Zeilenumbrüche ('\n')
- Wagenrücklauf ('\r')

Zwischen Schlüsselwörtern und Zahlen, Vektoren oder Klammern dürfen Whitespaces eingefügt werden, sind aber nicht notwendig.

4.1.2 Kommentare

In eine Szenenbeschreibung können Kommentare einfügt werden, welche von **RayStar™** beim Interpretieren der Szene ignoriert werden. Die Syntax für Kommentare ist identisch mit der der in C/C++ :

- Nach zwei Schrägstrichen '//' wird der Rest der Zeile bis zum nächsten Zeilenumbruch ignoriert.
- Nach '/*' werden alle Zeichen bis zum nächsten '*/' ignoriert.

4.1.3 Zahlen und Vektoren

Fließpunktzahlen werden in **RayStar™** genauso eingegeben, wie sie auch in C/C++ verlangt werden. Meistens wird aber eine Exponentialschreibweise gar nicht benötigt, da die Zahlen in **RayStar™** im Allgemeinen alle die selbe Größenordnung haben.

In den meisten Element-Definition werden als Parameter Vektoren verlangt, meist als Angabe eines Punktes oder einer Richtung im Raum. Die Syntax für einen Vektor lautet:

$$< x\ ,\ y\ ,\ z >$$

mit beliebigen Fließpunktzahlen *x,y* und *z*. Vektoren in **RayStar™** beziehen sich auf ein rechtshändig orientiertes Weltkoordinatensystem.

4.1.4 Farben

In vielen Element-Definitionen werden Farbangaben verlangt. In **RayStar™** werden Farbwerte im RGB-Farbraum spezifiert, als Angabe über die Intensität der einzelnen Farbkomponenten Rot, Grün und Blau. Dabei können die Intensitäten von 0.0 (kein Anteil) bis 1.0 (volle Sättigung) variieren. Also hat z.B. ein helles Gelb den RGB-Wert (1.0,1.0,0.0). Zur Angabe der RGB-Werte ist folgende Syntax möglich:

- **rgb <** *r-value* **,** *g-value* **,** *b-value* **>**

oder

- **color red** *r-value* **green** *g-value* **blue** *b-value*

Dabei sind *r-value, g-value* und *b-value* Fließpunktzahlen im Bereich von 0.0 bis 1.0. In der zweiten Schreibweise müssen nicht zwingend alle drei Farbkomponenten angeben werden, es genügt die Angabe einer einzigen, alle anderen werden dann defaultmäßig auf 0.0 gesetzt. So definiert z.B. **color green** *1.0* ein helles Grün und **color red** *0.7* **blue** *0.7* ein mittleres Magenta.

4.2 Transformationen

Es stehen in **RayStar™** folgende lineare Abbildungen zur Verfügung, mit denen man beliebige Objekte und Texturen frei im Raum transformieren kann:

4.2.1 Verschiebung

Ein Objekt oder eine Textur kann durch die Anweisung

translate <*x,y,z*>

relativ zu seiner momentanen Position im Raum um einen beliebigen Vektor <*x,y,z*> verschoben werden.

4.2.2 Skalierung

Durch die Anweisungen

scale *a* bzw. **scale** <*x,y,z*>

wird ein Objekt oder eine Textur um einen gemeinsamen Faktor *a* bzw. um die Faktoren *x,y,z* entlang den jeweiligen Raumrichtungen gestreckt oder gestaucht. Dies geschieht derart, daß jeder einzelne Punkt des Objektes oder der Textur mit dem entsprechenden Faktor multipliziert wird.

4.2.3 Rotation

Durch die Anweisung

rotate <*x,y,z*>

kann ein Objekt oder eine Texture nacheinander um den Winkel *x* um die x-, um den Winkel *y* um die y-und um den Winkel *z* um die z-Achse gedreht werden. Die Winkel *x,y,z* werden in Grad (0-360) ausgefaßt. Die Drehungen erfolgen im mathematisch positiven Drehsinn. D.h. durch eine Drehung z.B. um +90 Grad um die x-Achse wird die y-Achse auf die z-Achse und die z-Achse auf die -y-Achse abgebildet.

4.3 Kamera

In jeder **RayStar™**-Szene muß genau eine Kamera definiert werden. Die Kameraeinstellungen bestimmen den Blickwinkel und die Perspektive aus der die in der Szene beschriebenen Objekte betrachtet werden sollen. Die vollständige Definition einer Kamera lautet:

```
camera {
        location <x,y,z>
        look_at <x,y,z>
        sky <x,y,z>
        direction <x,y,z>
        right <x,y,z>
        up <x,y,z>
}
```

4.3.1 Position und Blickrichtung

Die Kamera wird an dem durch den **location**-Parameter angegebenen Punkt im Raum positioniert und so ausgerichtet, daß sie genau auf den durch den **look_at**-Parameter bestimmten Punkt zeigt. Der Horizont der Bildebene wird dann so ausgerichtet, daß er senkrecht auf der durch den **sky**-Vektor und durch die Blickrichtung der Kamera aufgespannten Ebene steht. Durch den **sky**-Vektors wird also eine Drehung der Kamera um die durch die Blickrichtung gegebene Achse angegeben. Standartmäßig hat der **sky**-Vektor den Wert <0,1,0>. Es ist darauf zu achten, daß der **sky**-Vektor und die Blickrichtung nicht linear abhängig seinen dürfen, da sie sonst ja keine Ebene aufspannen!

Man kann sich die Kamera am besten als ein kleines Flugzeug vorstellen, in dessen Cockpit der Betrachter sitzt. Das Flugzeug befindet sich an dem durch den **location**-Vektor gegebenen Punkt im Raum. Die Nase des Flugzeugs zeigt genau auf den durch **look_at** beschriebenen Punkt und die Flügel des Flugzeugs stehen genau senkrecht auf der durch den **sky**-Vektor und der Längsachse des Flugzeugs aufgespannten Ebene.

4.3.2 Perspektive und Bildausschnitt

Die Länge des Vektors **direction** gibt den Abstand der Bildebene von der Kamera an. Je kleiner dieser Abstand gewählt wird.umso mehr erscheint die Szene perspektivisch verzerrt. Die Längen der Vektoren **right** und **up** geben die Breite und die Höhe des Bildausschnittes in der Bildebene an. Die Richtungen der drei Vektoren dagegen spielen keine Rolle, da die Orientierung der Kamera und die Bildausschnitts schon durch die in 4.3.1 beschriebenen Vektoren eindeutig festgelegt sind. Die Vektorschreibweise ist nur aus Kompatibilitätsgründen zu POV-Ray gewählt worden.

4.4 Objekte

Zum Aufbau einer **RayStar™**-Szene stehen verschiedene einfache geometrische Objekte zur Verfügung. Diese Objekte können durch die in 4.2 beschriebenen Transformationen fast beliebig in ihrer Form und Position im Raum gestaltet werden. Neben der Angabe des Objekttyps, der nur die geometrische Form des Objektes bestimmt, müssen bei einer Objektdefinition auch Textureigenschaften (siehe 4.6) angegeben werden, welche das Erscheinungsbild des Objektes bestimmen. Allgemein hat eine Objektdefinition die Form:

***object-type* {**

shape-parameter

texture-definition

transformations

}

Stehen Transformationen in der Objektdefinition <u>vor</u> der Texturdefinition, dann wird die Textur nicht mittransformiert, stehen aber Transformationen <u>nach</u> der Texturdefinition, so wird die Transformation auch auf die Textur angewendet.

4.4.1 Kugeln

Eine Kugel um einen Mittelpunkt *<center>* mit dem Radius *radius* wird durch die Anweisung

sphere { *<center>* , *radius* }

beschrieben. Aus einer Kugel kann man durch Skalierungen mit verschiedene Faktoren in den jeweiligen Raumrichtungen und anschließende Drehungen leicht beliebige Ellipsoiden erhalten.

4.4.2 Ebenen

Eine Ebene wird durch die Anweisung

plane { *<normal>* , *distance* }

beschrieben. Dabei ist *<normal>* ein Normalenvektor auf der Ebene und *distance* der Abstand der Ebene vom Ursprung.

4.4.3 Quader

Durch die Anweisung

box { < *corner1* > , <*corner2*> }

wird ein Quader mit den beiden Eckpunkten *corner1* und *corner2* beschrieben. Dabei muß jede der drei Koordinaten von *corner2* echt größer als die entsprechende Koordinate von *corner1* sein. Die restlichen 6 Eckpunkte des Quaders werden so gesetzt, daß jede Kante des Quaders parallel zu einer der drei Raumachsen liegt.

4.4.4 Kegel

Die geometrischen Objekte Spitzkegel, Kegelstumpf und Zylinder können alle durch die Anweisung

cone { *<base>* , *base-radius* , *<top>* , *<top-radius>* [**open**] }

erzeugt werden. Dabei geben die beiden Vektoren *<base>* und *<top>* die beiden Endpunkte der Kegelachse an und die beiden Fließpunktzahlen *base-radius* und *top-radius* die Radien des Kegels und den beiden Endpunkten. Mindestens einer der beiden Radien muß von 0 verschieden sein. Ist einer der beiden Radien gleich 0, so erhält man einen Spitzkegel, sind beide Radius gleich, ergibt sich ein Zylinder und sonst ein Kegelstumpf. Durch das optionale Schlüsselwort **open** kann bestimmt werden, ob die beiden Enden des Kegels verschlossen sein sollen oder nicht.

4.5 Lichtquellen

In jeder Szene sollte mindestens eine Lichtquelle definiert sein, da sonst nur ambiente Beleuchtung auftritt, was nicht sonderlich realistisch aussieht.

4.5.1 Punktlichtquellen

Eine Punktlichtquelle ist ein beliebig kleiner Punkt im Raum, der Licht einer bestimmten Farbe gleichmäßig in alle Richtungen abstrahlt. Die Syntax für eine Punktlichtquelle lautet:

light_source { *<x,y,z> color-value* **}**

Die Intensität eine von einer Punktlichtquelle ausgestrahlten Lichtstrahls ist nicht von der Entfernung von der Lichtquelle abhängig, d.h. eine Punktlichtquelle erscheint immer in der gleichen Helligkeit, egal wie weit der Betrachter von ihr entfernt ist.

4.5.2 Spots

Spots sind Punktlichtquellen, die das Licht nur in eine bestimmte Richtung kegelförmig abstrahlen. Innerhalb dieses Kegels ist die Intensität des Lichts maximals, während sie zum Rande des Kegels langsam bis auf Null abfällt. Die vollständige Definition eines Spotlichts lautet :

light_source {

<x,y,z> color-value

spotlight

point_at *<x,y,z>*

radius *value*

falloff *value*

tightness *value*

fade *fade_from* **,** *fade_at*

}

Am besten stellt man sich ein Spotlicht als zwei ineinander verschachtelte Kegel vor. Die Spitzen der beiden Kegel befinden an der Position der Lichtquelle und die Achsen der Kegel zeigen in Richtung auf den durch den **point_at**-Parameter gegebenen Punkt. Innerhalb des inneren Kegels, dessen Scheitelwinkel in Grad durch den **radius**-Parameter bestimmt wird, ist die Lichtintensität maximal und außerhalb des äußeren Kegels, dessen Scheitelwinkel in Grad durch den **falloff**-Parameter angegeben wird, ist die Lichtintensität gleich Null.

Durch den **tightness**-Parameter, der zwischen 1.0 und 100.0 liegen sollte, kann gesteuert werden, wie schnell die Lichtstärke zwischen dem inneren und dem äußeren Kegel abfällt. Große Werte für **tightness** lassen die Lichtstärke sehr schnell, kleine Werte lassen sie langsamer zum Rand hin abfallen.

Im Gegensatz zu Punktlichtquellen erscheinen Spot dunkler, je weiter sie vom Betrachter entfernt sind. Durch den **fade**-Parameter kann angegeben werden, ab welchen Entfernungen dies geschehen soll. Ist die Entfernung von der Lichtquelle kleiner als *fade_from* , dann erscheint diese in voller Intensität, ist die Entfernung aber größer als *fade_at* , dann ist die Lichtquelle völlig dunkel, d.h. die Intensität ist gleich Null. Zwischen diesen beiden Entfernungen nimmt die Intensität der Lichtquelle linear ab. Standartmäßig ist *fade_from = fade_at = 1e100* eingestellt, so daß die Intensität der Lichtquelle praktisch gar nicht abnimmt.

4.6 Texturen

In **RayStar™** werden jedem Objekt einer Szene Texturen zugewiesen, welche bestimmen, wie die Oberfläche des Objekts dem Betrachter erscheint. Die Texturen sind als *solid-textures* realisiert, d.h. jedem Punkt eines Objektes wird über eine Funktion, welche die 3-dimensionalen Koordination des Punktes als Parameter übernimmt, ein bestimmter Textur-Wert zugeordnet, welcher dann die verschiedene Oberflächeneigenschaften des Objektes an diesem Punkt bestimmt. Der Name *solid-textures* ist dadurch zu verstehen, daß man sich die Objekte recht anschaulich als aus einem großen 'Block' dieser Textur herausgeschnitten vorstellen kann.

Eine Textur besteht im allgemeinen aus Beschreibungen der Pigmentierung, der Oberflächenbeschaffenheit, der Materialeigenschaften und einer beliebigen Anzahl von Transformationen (siehe 4.2):

texture {

  pigment { . . . **}**

  finish { . . . **}**

  normal { . . . **}**

  transformations ..

}

In einer Objektdefinition (siehe 4.4) können die **pigment**- und **finish**-Definition auch direkt, ohne die umgebene Texturdefiniton, angegeben werden.

4.6.1 Pigmentierung

Durch die Anweisung

pigment { *pigment-type pigment options* [*transformations...*] **}**

wird einem Objekt eine bestimmte Pigmentierung zugewiesen. Es sind folgende Pigmentierungsarten möglich:

4.6.1.1 Einheitliche Färbung

Mit **pigment {** *color-value* **}** wird einem Objekt einheitlich die durch *color-value* angegebene Farbe zugeordnet. Zusätzlich zu den in 4.1.4 angegeben Möglichkeiten für *color-value* steht noch ein weitere Parameter, **filter**, zu Verfügung:

- **rgbf <** *r-value* **,** *g-value* **,** *b-value* **,** *filter-value* **>**
- **color red** *r-value* **green** *g-value* **blue** *b-value* **filter** *filter-value*

Dieser Fließpunktwert zwischen 0.0 und 1.0 gibt die Transparenz des Objektes an, wobei ein Wert von 0.0 ein opakes, völlig undurchsichtiges Objekt definiert und ein Wert von 1.0 ein total durchscheinendes. Der Defaultwert für **filter** ist 0.0.

4.6.1.2 Schachbrett-Muster

Durch die Anweisung

pigment { checker *color-value1 color-value2* **}**

wird einem Objekt ein 3-dimensionales Schachbrettmuster zugeordnet. Man kann sich dazu vorstellen, daß der gesamte Raum mit Einheitswürfeln parkettiert wird, deren Eckpunkte jeweils ganzzahlige Koordinaten haben. Diese werden dann so mit den durch *color-value1* und *color-value2* gegebenen Farben eingefärbt, daß der Würfel mit den beiden Ecken (0,0,0) und (1,1,1) die Farbe *color-value1* hat und je zwei Würfel, die eine gemeinsame Seitenfläche haben, verschiedene Farben tragen. Aus dem so resultierenden räumlichen Schachbrettmuster wird dann das Objekt ‘herausgeschnitten’.

4.6.1.3 Wolkenmuster

Für die meisten Szene wird als Hintergrund ein möglichst realistischer blauer Himmel mit Wolken gewünscht. Dafür steht eine Wolken-Pigmentierung zur Verfügung, welche dem Hintergrund (siehe 4.7.1), aber auch jedem anderen Objekt, mit der Anweisung:

pigment { clouds *variation* **,** *thickness* **}**

zugeordnet werden kann. Mit den Fließpunktparametern *variation* und *thickness* kann das Aussehen des Wolkenmusters beeinflußt werden. Der Wert von *variation* , der etwa zwischen 0.1 und 5.0 liegen sollte, bestimmt, wie fein oder grob das Wolkenmuster erscheint. Kleine Werte von *variation* bewirken wenige, größere Wolken und höhere Werte bewirken viele kleinere Wolken. Der zweite Parameter, *thickness*, sollte etwa zwischen 0.0 und 10.0 liegen und bestimmt, wie ‘bewölkt’ der Himmel erscheinen soll. Kleine Werte erzeugen einen fast wolkenlosen, große Werte hingegen eine verhangenen, bewölkten Himmel. Defaultmäßig eingestellt sind die Werte *variation = 2.0* und *thickness = 3.0*.

4.6.2 Oberflächenbeschaffenheit

Die Oberflächenbeschaffenheit (oder *finish)* eines Objektes, bestimmt die Art und Weise, wie Lichtstrahlen von dem Objekt reflektiert werden. Eine Kugel aus Plastik wirkt z.B. wesentlich matter, als eine aus poliertem Metall. Es können insgesamt 7 Parameter eingestellt werden. Die vollständige Syntax hierzu lautet:

finish {

ambient *value*

diffuse *value*

reflection *value*

specular *value*

roughness *value*

phong value

phong_size *value*

metallic

}

Dieses Parameter bewirken im einzelnen:

4.6.2.1 Ambiente Helligkeit

Es wird in einer **RayStar™**-Szene angenommen, das an jedem Punkt im Raum eine Umgebungshelligkeit vorherscht, auch wenn dieser Punkt im Schatten von allem vorhandenen Lichtquellen liegt. Der **ambient**-Parameter eines Objektes bestimmt, wieviel von diesem Umgebungslicht von dem Objekt wieder abgestrahlt wird. Es sind Werte zwischen 0.0 und 1.0 möglich, wobei ein Wert von 0.0 das Objekt völlig dunkel erscheinen läßt, falls es sich im Totalschatten befindet, und ein Wert von 1.0 läßt das Objekt zu 100% in der Farbe gemäß seiner Pigmentierung erscheinen. Standardmäßig ist eine ambiente Helligkeit von 0.1 eingestellt.

4.6.2.2 Diffuse Helligkeit

Der **diffuse**-Parameter eines Objektes gibt an, wieviel an einem Punkt eingestrahltes Licht diffus reflektiert wird. Diffuse Reflexion ist die Art, wie Licht von völlig matten Oberflächen wieder abgestrahlt wird. Das eintreffende Licht wird in alle Richtungen mit der gleichen Intensität wieder abgestrahlt, die Position eines Betrachters zu dem Objekt ist also für das diffuse reflektierte Licht völlig egal, entscheidend ist für die Intensität des abgestrahlten Lichts nur der Winkel zwischen dem eintreffenden Lichtstrahl und der Oberfläche des Objekts. Mögliche **diffuse**-Werte liegen zwischen 0.0 und 1.0, voreingestellt ist ein Wert von 0.6. Ein Wert von 0.0 erzeugt eine sehr glatte, polierte Oberfläche und ein Wert von 1.0 eine völlig matte Oberfläche.

4.6.2.3 Reflexion

Neben der diffuse Reflexion tritt bei glatten, polierten Oberfläche meist auch Reflexion auf, so daß auf einem Objekt die Spiegelung von anderen Objekten sichtbar sind. Ein **reflection**-Wert von 1.0 erzeugt eine perfekt spiegelnde Oberfläche, während der voreingestellte Wert von 0.0 bewirkt, daß keine Reflexionen berechnet werden. Möglich sind **reflection**-Werte zwischen 0.0 und 1.0.

4.6.2.4 Schlaglichter

An glatten Oberflächen, z.B. an einem reifen Apfel, die von einer Lichtquelle beleuchtet werden, treten in der Regel Schlaglichter oder *Highlights* auf. In **RayStar™** können Schlaglichter nach zwei verschiedenen Beleuchtungsmodellen berechnet werden, die jeweils zu geringfügig anderen Ergebnissen führen. Die beiden Beleuchtungsmodelle schließen sich nicht gegenseitig aus, es ist aber normalerweise sinnvoll, nur entweder das eine oder das andere zu aktivieren. Standardmäßig sind sowohl *phong-* als auch *specular-highlights* ausgeschaltet.

4.6.2.4.1 Specular Highlights

Mit dem **specular**-Parameter wird die Helligkeit der *specular highlights* auf einem Objekt gesteuert. Die Werte können zwischen 0.0 und 1.0 liegen, wobei 0.0 überhaupt keine *specular highlights* erzeugt und 1.0 eine vollständige Sättigung des Schlaglichts an seiner hellsten Stelle in der Farbe der einstrahlenden Lichtquelle zur Folge hat. Mit dem **roughness**-Parameter kann kontrolliert werden, wie groß das Schlaglicht auf einem Objekt erscheint. Typische Werte liegen hierbei zwischen 0.0005 und 1.0. Ein kleine **roughness**-Wert führt zu einer sehr glatten Oberfläche mit einem fast punktförmigen Schlaglich, wogegen große Werte eine raue Fläche produzieren, auf der eine sehr großes, kreisförmiges Schlaglicht zu sehen ist. Voreingestellt ist ein Wert von 0.05.

4.6.2.4.2 Phong Highlights

Der **phong**-Parameter entspricht analog dem **specular**-Parameter. Mit dem **phong_size**-Parameter wird die Größe des Schlaglichts im Phong-Beleuchtungsmodell gesteuert. Typische Werte liegen zwischen 1.0 (sehr matt, großes Schlaglicht) und 250.0 (hochpoliert, fast punktförmiges Schlaglicht).

4.6.2.4.3 Metallische Oberflächen

Zusammen mit dem **phong**- oder **specular-**Parameter kann das Schlüsselwort **metallic** angegeben werden. Dieses bewirkt, daß die Farbe des Schlaglichts durch die Oberflächenfarbe des Objekts gefiltert wird, daß also z.B. auf einem roten Objekt von einer weißen Lichtquelle ein rotes Schlaglicht erzeugt wird und nicht ein weißes.

4.6.3 Materialeigenschaften

Die meisten Bilder, die von einfachen Raytracern erzeugt werden, sehen recht künstlich aus, da die dargestellten Gegenstände 'perfekte' geometrische Objekte sind. In der Wirklichkeit haben wir es aber meist mit Gegenständen zu, die eine unregelmäßig strukturierte Oberfläche aufweisen, man denke z.B. an eine Kugel aus Naturstein. Da es vom Rechen- und Speicheraufwand praktisch unmöglich ist solche 'natürlichen' Objekte exakt geometrisch zu modellieren, greift man zu anderen Methode, unregelmäßig strukturierte Oberflächen nachzubilden : Man variiert einfach den bei der Beleuchtungsberechnung benötigten Normalenvektor einer Oberfläche um einen gewissen kleinen Anteil. Dieser Anteil wird dabei so berechnet, daß die Oberfläche ein mehr oder minder strukturiertes Aussehen erhält. Die Geometrie des Objektes bleibt von der Veränderung der Normalenvektoren aber völlig unbeeinflußt. So sind die scheinbaren Strukturen einer Oberfläche auch nicht in ihrem Schattenwurf sichtbar.

Die Syntax für eine Normalentextur lautet:

normal { *normal-type* *value* [*transformations...*] **}**

Als ***normal-type*** kann dabei einer der im Folgenden beschriebenen Schlüsselwörter angegeben werden. Durch den Wert *value* wird bei allen Strukturen festgelegt, in welchem Ausmaß die Oberfläche verändert werden soll. Ein Wert von 0.0 bewirkt überhaupt keine Beeinflußung, ein Wert von 1.0 oder größer bewirkt eine starke Abänderung der Normalenvektoren.

4.6.3.1 Bumps

Mit der Normalen-Textur **bumps** wird einer Oberfläche ein unregelmäßig 'ausgebeultes' Aussehen gegeben. Mit einer entsprechend kleinen Skalierung läßt sich so z.B. die Oberfläche einer Orange sehr gut modellieren.

4.6.3.2 Dents

Die **dents**-Struktur ist der oben beschriebenen **bump**-Struktur sehr ähnlich. Die erzeugten Oberflächen zeigen aber deutlichere 'Beulen' und 'Einschläge'.

4.6.3.3 Wellen-Struktur

Durch die **waves**-Struktur kann einem Objekt das Aussehen einer Wasseroberfläche gegeben werden. Es erscheinen dabei typische Wellenberge und -täler, die von mehreren, zufällig innerhalb des Einheitswürfels verteilen Wellen-Zentren ausgehen, auf der entsprechenden Oberfläche.

4.7 Szenen-Einstellungen

4.7.1 Hintergrund

Trifft ein Sehstrahl auf kein Objekt in der Szene, so wird ihm ein Hintergrundfarbe zugeordnet. Diese Hintergrundfarbe kann in 4.6.1 beschriebene Pigmentierung sein.

background { *pigment-type pigment-options* **}**

Es wird dabei allerdings die Pigmentierungsfunktion für einen Punkt im Raum ausgewertet, sondern für den Richtungsvektor des jeweiligen Sehstrahls. Standartmäßig eingestellt ist ein einfacher schwarzer Hintergrund. In vielen Szenen wird mit **background { clouds** *2.0* **,** *3.0* **}** ein blauer Himmel mit weißen Wolken eingestellt.

5 Implementierungsübersicht

RayStar™ ist vollständig objektorientiert in Visual C++ 2.1 unter Windows 95/Windows NT implementiert. Die durchgängige Objektorientierung ermöglicht eine strikte Trennung auf Quellcode-Ebene zwischen der graphischen Oberfläche, dem ***Frontend***, und dem ***RayTracing-Kernel***, der für die Generierung der Graphiken verantwortlich ist.

5.1 Das Frontend

Das Frontend von **RayStar™** ist eine typische MFC-Applikation mit Multiple Document Interface (MDI) und Document-View-Architektur. Grundlagen zu diesem Applikations-Modell finden sich in [4],[11],[12],[13].

In der Applikationsklasse ***CRayStarApp*** werden folgende zwei Multidocument-Templates registriert:

- *m_pRayDocTemplate* verwaltet die Szenenbeschreibungen mittels der Klassen ***CRayStarDoc*** und ***CRayStarView***
- *m_pDIBDocTemplate* verwaltet die Bitmaps mittels der Klassen ***CDIBDoc*** und ***CDIBView***

5.1.1 Der Szenen-Editor

Die Klasse ***CRayStarView*** erbt die komplette Editor-Funktionalität von der Klasse ***CEditView*** und fügt dem u.a. noch das Anzeigen von Zeilen- und Spaltenposition in der Statuszeile hinzu. Darüber hinaus verwaltet ***CRayStarView*** die gesamten Benutzerinteraktionen, die daß Steuern der Raytracing-Jobs betreffen.

5.1.2 Der Bitmap-Viewer

Die Klasse ***CDIBView*** übernimmt das Anzeigen der berechneten Bitmaps. Die gesamte File-I/O Funktionalität ist in der Klasse ***CDIBDoc*** gekapselt. Innerhalb dieser Klasse werden Routinen aus der Library ***BITMAP.DLL*** aufgerufen, in welcher C-Funktionen zum Import/Export der verschieden Graphikformate und zum Handling der DIBs implementiert sind. Ferner verwaltet die Klasse ***CDIBDoc*** auch das Aktualisieren der Bitmaps während des Berechnens und die Anzeige der Progress-Dialogs.

5.2 Der Raytracing-Kernel

Der gesamte Raytracing-Kernel ist in der Klasse ***CRayEngine*** gekapselt.

```
class CRayEngine : public CObject
{
private:
        CRParser*     m_pParser;
        CRScene*      m_pScene;
public:
        WORD          InitScene(CString strScene);
        WORD          InitTracing(WORD nHeight, WORD nWidth,  WORD nDepth);
        COLORREF      CalcColor(WORD xPos, WORD yPos);
};
```

Zum Berechnen einer Szene sind nur folgende Schritte nötig:

1. Erzeugen eines ***CRayEngine***-Objektes (statisch oder dynamisch).
2. Aufrufen von *CRayEngine::InitScene* mit der zu berechnenden Szenebeschreibung als Parameter.
3. Aufrufen von *CRayEngine::InitTracing* mit den Werten für die Höhe, Breite und Berechnungstiefe des zu erzeugenden Bildes.
4. Aufrufen von *CRayEngine::CalcColor* für jedes Pixel des zu berechnenden Bildes.

5.2.1 CRParser

Innerhalb der Klasse ***CRayEngine*** übernimmt die Klasse ***CRParser*** die Aufgabe, aus einer gegebenen Szenenbeschreibung ein entsprechendes Objekt der Klasse ***CRScene*** zu erzeugen.

```
class CRParser  :  public CObject
{
public:
        CRParser();
        ~CRParser();
public:
        int           ParseScene(const CString strScene, CRScene* pScene);
};
```

Der Aufruf von *ParseScene(strScene,pScene)* baut das ***CRScene***-Object, auf das *pScene* zeigt, so zusammen, daß es 1:1 die Szene, die durch *strScene* beschrieben wird, repräsentiert.

5.2.2 CRScene

Die Klasse ***CRScene*** kapselt alle Daten und Algorithmen, die beim Raytracing-Prozeß benötigt werden.

```
class CRScene : public CObject
{
protected:
        CSize                                   m_ImageSize;
        CList<CRLightSource*,CRLightSource*>    m_pLights;
        CList<CRShape*,CRShape*>                m_pObjects;
        CRCamera                                m_Camera;
        CRPigment*                              m_pBackground;
        WORD                                    m_nMaxTracingLevel;
protected:
        CRShape*    FindClosestIntersection(const CRRay& Ray, CRVector& Intersection);
        CRColor&    TraceRay(WORD nLevel,const CRRay& ray, CRColor& color);
        void        CalcLight(const CRLightSource* pLight, const CRShape* pObject,
                            const CRVector& Q, const CRVector& N,
                            const CRRay& Ray, CRColor& color);
        BOOL        InShadow(const CRShape* pObject, const CRLightSource* pLight,
                            const CRVector& L);
public:
        COLORREF    CalcColor(const WORD xPos, const WORD yPos);
};
```

Eine Szene wird dabei durch folgende Daten vollständig repräsentiert:

- Alle in der Szene enthaltenen Objekte *(m_pObjects)*
- Alle in der Szene enthaltenen Lichtquellen *(m_pLights)*
- Die Kamera *(m_pCamera)*
- Hintergrundfärbung *(m_pBackground)*
- Auflösung des zu berechnenden Bildes in Pixel *(m_ImageSize)*
- Maximale Rekursionstiefe *(m_nMaxTracingLevel)*

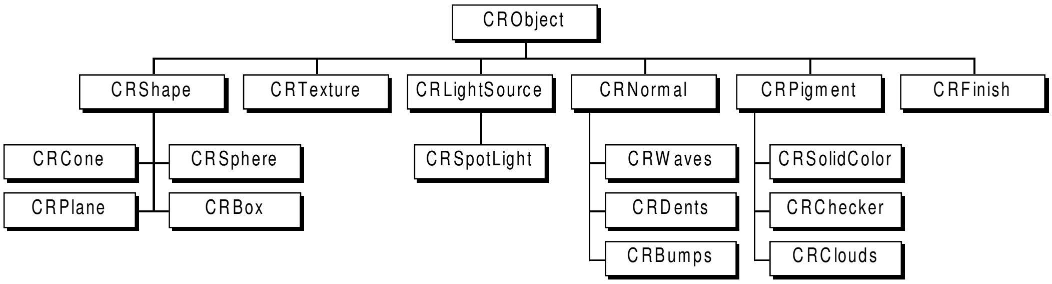
Die in Kapitel 2 in Pseudocode angegebenen Algorithmen sind in den folgenden vier Member-Funktionen implementiert:

- *FindClosestIntersection()*
- *TraceRay()*
- *CalcLight()*
- *InShadow()*

5.2.3 CRObject

Die Klasse ***CRObject*** ist die Basisklasse für fast alle Klassen, welche die in einer Szene enthaltenen Objekte repräsentieren. Es wird praktisch jedes in der **RayStar™-**

Sprache enthaltene Objekt (***pigment, sphere, texture,***...) durch ein entsprechendes,



von ***CRObject*** abgeleitetes C++-Objekt programmintern abgebildet.

Die grundlegende gemeinsame Eigenschaft von fast allen Objekten in einer Szene ist, daß sie initialisiert werden können und daß auf sie affine Transformationen angewendet werden können. Diese Funktionalität ist in virtuelle Funktionen in der Basisklasse ***CRObject*** implementiert:

```
class CRObject  :  public CObject
{
public:
        CRTransform*            m_pTransform;
public:
        CRObject();
        virtual ~CRObject();
public:
        virtual WORD            Init();
        virtual void            Translate(const CRVector& v);
        virtual void            Scale(const CRVector& v);
        virtual void            Scale(const double& Factor);
        virtual void            Rotate(const CRVector& v);
};
```

Diese Funktionen *Translate, Scale, Rotate* sind in der Basisklasse so implementiert, das die jeweiligen affinen Transformationen in *m_pTransform* akkumuliert werden.

5.2.4 CRTransform

Die Klasse ***CRTransform*** stellt Methoden zum Ausführen von affinen Transformationen bereit. Die gewünschte affine Transformation und deren inverse Abbildung werden in den 4x4-Matrizen *m_Matrix* und *m_Inverse* repräsentiert. Näheres zur Matrix-Repräsentation von 3D-Transformationen findet man in [5] S.180-191.

Der Konstruktor initialiert ein neues ***CRTransform***-Objekt auf die identische Abbildungung. Durch ggf. wiederholten Aufruf von *Translate, Scale, Rotate* , kann danach die gewünschte affine Abbildung zusammengesetzt werden.

Mittels der Member-Funktionen *TransPoint(),TransDir, TransRay()* , bzw. *Inv...* kann die so festgelegte affine Transformation, bzw. deren Inverse auf einen Ortsvektor, einen Richtungsvektor oder einen Strahl angewandt werden.

```
class  CRTransform  :  public CObject
{
private:
        CRMatrix            m_Matrix;
        CRMatrix            m_Inverse;
public:
        CRTransform();
public:
        void                Translate(const CRVector& v);
        void                Scale(const CRVector& v);
        void                Scale(const double& Factor);
        void                Rotate(const CRVector& v);
public:
        CRVector            TransPoint(const CRVector& point) const;
        CRVector            InvTransPoint(const CRVector& point) const;
        CRVector            TransDir(const CRVector& dir) const;
        CRVector            InvTransDir(const CRVector& dir) const;
        CRRay               TransRay(const CRRay& ray) const;
        CRRay               InvTransRay(const CRRay& ray) const;
};
```

5.2.5 CRShape

Die rein virtuelle Klasse ***CRShape*** bildet die Basis-Klasse, von der alle Klassen, welche die geometrischen Objekte in einer Szene (***sphere***, ***box***, ***cone***,...) repräsentieren, abgeleitet sind.

```
class CRShape  :  public CRObject
{
public:
        CRTexture           m_Texture;
public:
        CRShape();
        virtual ~CRShape();
public:
        void                Translate(const CRVector& v);
        void                Scale(const CRVector& v);
        void                Scale(const double& Factor);
        void                Rotate(const CRVector& v);
protected:
        virtual void        CalcNormal(const CRVector& Q, CRVector& N) const = 0;
public:
        virtual double      CalcIntersection(const CRRay& Ray,const CRShape** ppObject,
                                    CRIntersectionList* pIntersectionList = NULL) const = 0;
        void                GetNormal(const CRVector& Q, CRVector& N) const;
};
```

Innerhalb der Klasse ***CRShape*** ist das Objekt *m_Texture* für die Oberflächeneigenschaften des repräsentierten Objektes verantwortlich.

Die von ***CRObject*** virtuell geerbten Memberfunktionen *Translate(), Scale(), Rotate()*, sind in ***CRShape*** so implementiert, daß *CRObject::Translate(),...* und zusätzlich noch *m_Texture::Translate(),...* aufgerufen wird. Wie in Kapitel 2 schon erläutert, sind die einzigen beiden Berechnungen, die für ein geometrisches Objekt durchgeführten werden müssen, die Schnittpunktberechnung mit einem vorgegebenen Strahl und die Berechnung eines Normalenvektors an einem gegebenen Oberflächenpunkt. Diese beiden Aufgaben werden von den rein virtuellen Funktionen *CalcIntersection()* und *CalcNormal()* übernommen, die für jede abgeleitete Klasse implementiert werden müssen. Durch die Implementierung dieser beiden Funktionen ist die Geometrie eines Objektes vollständig bestimmt.

Die Funktion *CalcNormal()* ist protected, anstand ihrer wird nämlich die Funktion *GetNormal()* als public-Member bereitgestellt. Diese ruft intern *CalcNormal()* auf, verändert aber ggf. den so berechneten Normalenvektor entsprechend, falls in *m_Texture* eine Normalentextur (siehe Abschnitt 4.6.3) definiert worden ist, und gibt denn so erhaltenen endgültigen Normalvektor zurück.

5.2.6 CRTexture

In der Klasse ***CRTexture*** sind alle Daten und Algorithmen gekapselt, die die Oberflächeneigenschaften eines geometrischen Objektes bestimmen.

```
class CRTexture : public CRObject
{
public:
        CRPigment*        m_pPigment;
        CRNormal*         m_pNormal;
        CRFinish          m_Finish;
public:
        CRTexture();
        ~CRTexture();
public:
        void              Translate(const CRVector& v);
        void              Scale(const CRVector& v);
        void              Scale(const double& Factor);
        void              Rotate(const CRVector& v);
public:
        void              SetPigment(CRPigment* pPigment);
        void              SetNormal(CRNormal* pNormal);
        CRColor           GetAmbientColor(const CRVector& Position) const;
        CRColor           GetDiffuseColor(const CRVector& Position) const;
        CRColor           GetSpecularColor(const CRColor& LightColor, const CRVector& N,
                                 const CRVector& L, const CRVector& V,
                                 const CRVector& Position) const;
        CRColor           GetReflectedColor(const CRColor& color) const;
        CRColor           GetTransparentColor(const CRColor& color) const;
        BOOL              IsTransparent() const;
        BOOL              IsReflecting() const;
};
```

Die einzelnen Oberflächenparameter sind in den Membervariablen *m_pPigment, m_pNormal* und *m_Finish* gekapselt, analog zu den Sprachelementen ***pigment, normal, finish***. Die Klasse ***CRTexture*** stellt mit den fünf Memberfunktionen *Get...Color()* Algorithmen zu Verfügung, um die verschiedenen, in Kapitel 2 vorgestellten, Farbanteile zu berechenen. Näheres zu dem verwendeten Beleuchtungsmodell findet sich u.a. in [5] S.477-490.

5.3 Quellcode-Verzeichnis

Die einzelnen Klassen sind in folgenden Dateien implementiert:

Klasse	Implementierung	Headerdatei
CDIBDoc	DOCDIB.CPP	DOCDIB.H
CDIBView	VIEWDIB.CPP	VIEWDIB.H
CMainFrame	MAINFRM.CPP	MAINFRM.H
CProgressDlg	DOCDIB.CPP	DOCDIB.H
CProgressThread	DOCDIB.CPP	DOCDIB.H
CRayStarApp	RAYSTAR.CPP	RAYSTAR.H
CRayStarDoc	DOCRAY.CPP	DOCRAY.H
CRayStarView	VIEWRAY.CPP	VIEWRAY.H
CRBox	CRSHAPE.CPP	CRAYSTAR.H
CRBumps	CRTEXT.CPP	CRAYSTAR.H
CRCamera	CRAYSTAR.CPP	CRAYSTAR.H
CRChecker	CRAYSTAR.CPP	CRAYSTAR.H
CRClouds	CRAYSTAR.CPP	CRAYSTAR.H
CRColor	CRAYSTAR.CPP	CRAYSTAR.H
CRCone	CRSHAPE.CPP	CRAYSTAR.H
CRDents	CRTEXT.CPP	CRAYSTAR.H
CRFinish	CRAYSTAR.CPP	CRAYSTAR.H
CRLightsource	CRAYSTAR.CPP	CRAYSTAR.H
CRMatrix	CRVECTOR.CPP	CRVECTOR.H
CRNormal	CRAYSTAR.CPP	CRAYSTAR.H
CRObject	CRAYSTAR.CPP	CRAYSTAR.H
CRParser	CRPARSER.CPP	CRPARSER.H
CRPigment	CRAYSTAR.CPP	CRAYSTAR.H
CRPlane	CRSHAPE.CPP	CRAYSTAR.H
CRRay	CRAYSTAR.CPP	CRAYSTAR.H
CRRayEngine	CRAYSTAR.CPP	CRAYSTAR.H
CRScene	CRAYSTAR.CPP	CRAYSTAR.H
CRShape	CRSHAPE.CPP	CRAYSTAR.H
CRSolidColor	CRAYSTAR.CPP	CRAYSTAR.H
CRSphere	CRSHAPE.CPP	CRAYSTAR.H
CRSpotlight	CRAYSTAR.CPP	CRAYSTAR.H
CRTexture	CRTEXT.CPP	CRAYSTAR.H
CRTransform	CRAYSTAR.CPP	CRAYSTAR.H
CRVector	CRVECTOR.CPP	CRVECTOR.H
CRWaves	CRTEXT.CPP	CRAYSTAR.H
CStartTracingDlg	VIEWRAY.CPP	VIEWRAY.H
CTracingResults	VIEWRAY.CPP	VIEWRAY.H

6 Literaturverzeichnis

[1] BRONSVOORT, WILLEM F. and KLOK, FOPKE, *Ray Tracing Generalized Cylinders*, ACM Transactions on Graphics, Vol. 4, No. 4, October 1985, S.291-303

[2] CLAUSSEN, UTE und PÖPSEL, JOSEF, *Der Realität auf der Spur*, c't 6/91, S.204, c't 8/91, S.196, Heise Verlag

[3] CLAUSSEN, UTE und PÖPSEL, JOSEF, *Himmel und Hölle - Dreidimensionale Texturen und ihre Implementierung*, c't 1/93, S.160, Heise Verlag

[4] CUSTER, HELEN, *Inside Windows NT*, Microsoft Press 1992, ISBN 1-55615-481-X

[5] FOLEY, JAMES D. et al., *Grundlagen der Computergraphik*¸ Addison-Wesley 1994, ISBN 3-89319-647-1

[6] GLASSNER, ANDREW S.(Hrsg.), *An Introduction to Ray Tracing*, Academic Press 1989

[7] HEINY, LOREN, *Advanced Graphics Programming Using C/C++*,John Wiley & Sons, Inc. 1993, ISBN 0-471-57159-8

[8] HORNUNG, CHRISTOPH und PÖPSEL, JOSEF, *3-D à la carte*, c't 4/89, c't 5/89, c't 7/89, c't 8/89 und c't 10/89, Heise Verlag

[9] KAJIYA, JAMES T., *New Techniques for Ray Tracing Procedurally Defined Objects*, ACM Transactions on Graphics, Vol. 2, No. 3, July 1983, S.161-181

[10] KLASSEN, VICTOR R., *Modelling the Effect of the Atmosphere on Light*, ACM Transactions on Graphics, Vol. 6, No. 3, July 1987, S.215-237

[11] KRUGLINSKI, DAVID J., *Inside Visual C++,* Microsoft Press 1993, ISBN 1-55615-511-5

[12] MICROSOFT COPORATION, *Introducing Visual C++*,Document No. DB57154-0694

[13] MICROSOFT COPORATION, *Development Library*,Microsoft Developer Network

[14] PERLIN, KEN and HOFFERT, E., *Hypertexture*,Computer Graphics 23 (3) Juli 1989

[15] PERLIN, KEN, *An Image Synthesizer*, Computer Graphics 23 (3), Juli 1985

[16] POV-RAY TEAM, *POV-Ray Version 2.0 User's Documentation*, 1993

[17] PÖPSEL, JOSEF, et. al., *Neue Techniken der Computergrafik*, Franzis Verlag 1992

[18] VAN WIJK, JARKE J., *Ray Tracing Objects Defined by Sweeping Planar Cubic Splines*, ACM Transactions on Graphics, Vol. 3, No. 3, July 1984, S.232-237

[19] WEGHORST, HANK, HOOPER, GARY and GREENBERG, DONALD P., *Improved Computational Methods for Ray Tracing*, ACM Transactions on Graphics, Vol. 3, No. 1, January 1984, S.52-69

[20] WEGNER, TIM, *The Waite Group's Image Lab*, Waite Group Press 1992, ISBN 1-878739-11-5

[21] WELLS, DREW und YOUNG, CHRIS, *The Waite Group's Ray Tracing Creations*, Waite Group Press 1993, ISBN 1-878739-27-1