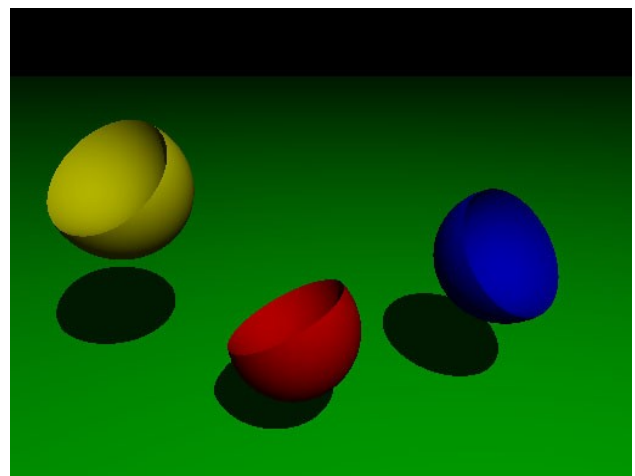
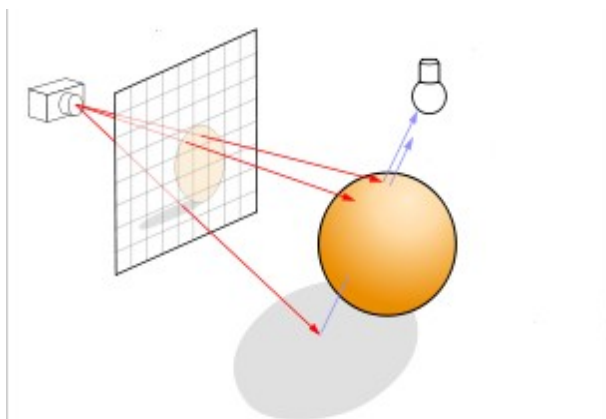
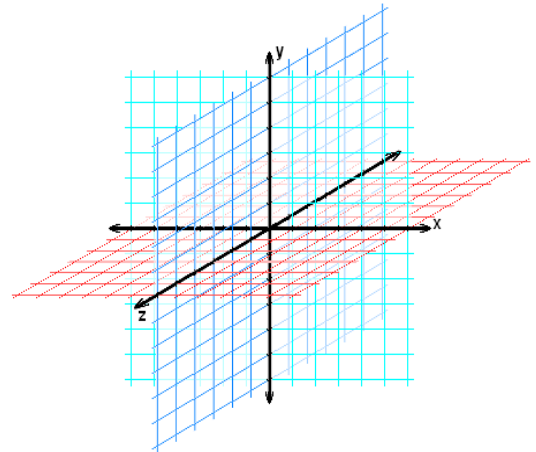


POV-Ray Guide





Inhaltsverzeichnis:

DIE ARBEITSOBERFLÄCHE VON POV-RAY.....	3
3D-KOORDINATENSYSTEM.....	4
SYNTAX.....	4
OBJEKTE.....	5
GERADE EBENE.....	6
VORDEFINIERTER HIMMEL.....	6
VORDEFINIERTE FARBEN.....	7
SELBSTGEMISCHTE FARBEN.....	8
FARBEN AUFHELLEN ODER ABDUNKELN.....	8
TRANSPARENZ.....	8
MUSTER.....	9
OBERFLÄCHEN.....	10
VORDEFINIERTER OBERFLÄCHEN.....	11
BEWEGUNG.....	12
VERSCHIEBEN.....	12
ROTIEREN / DREHEN.....	13
DREHRICHTUNG.....	13
EINE WEITERE ART DER MODIFIKATION VON OBJEKTEN: SKALIERUNG.....	14
BEWEGUNGEN SICHTBAR MACHEN.....	15
QUICKRES.INI.....	15
DER TAKT.....	15
BEISPIELE FÜR DIE VERWENDUNG VON CLOCK.....	16
KOMPLEXE OBJEKTE.....	17
VEREINIGUNG.....	17
DIFFERENZ.....	18
SCHNITT.....	19
ÜBERSICHT ZU DEN MENGEN-OPERATOREN.....	19
FORMATIERUNG.....	20
FEHLER.....	21
SYNTAXFEHLER.....	21
LOGISCHE FEHLER.....	21
VARIABLEN.....	22
KONTROLLSTRUKTUREN.....	23
EXKURS: ZUFALLSZAHLEN.....	24
LINKS.....	24

Quellenangaben zu den Titelbildern:

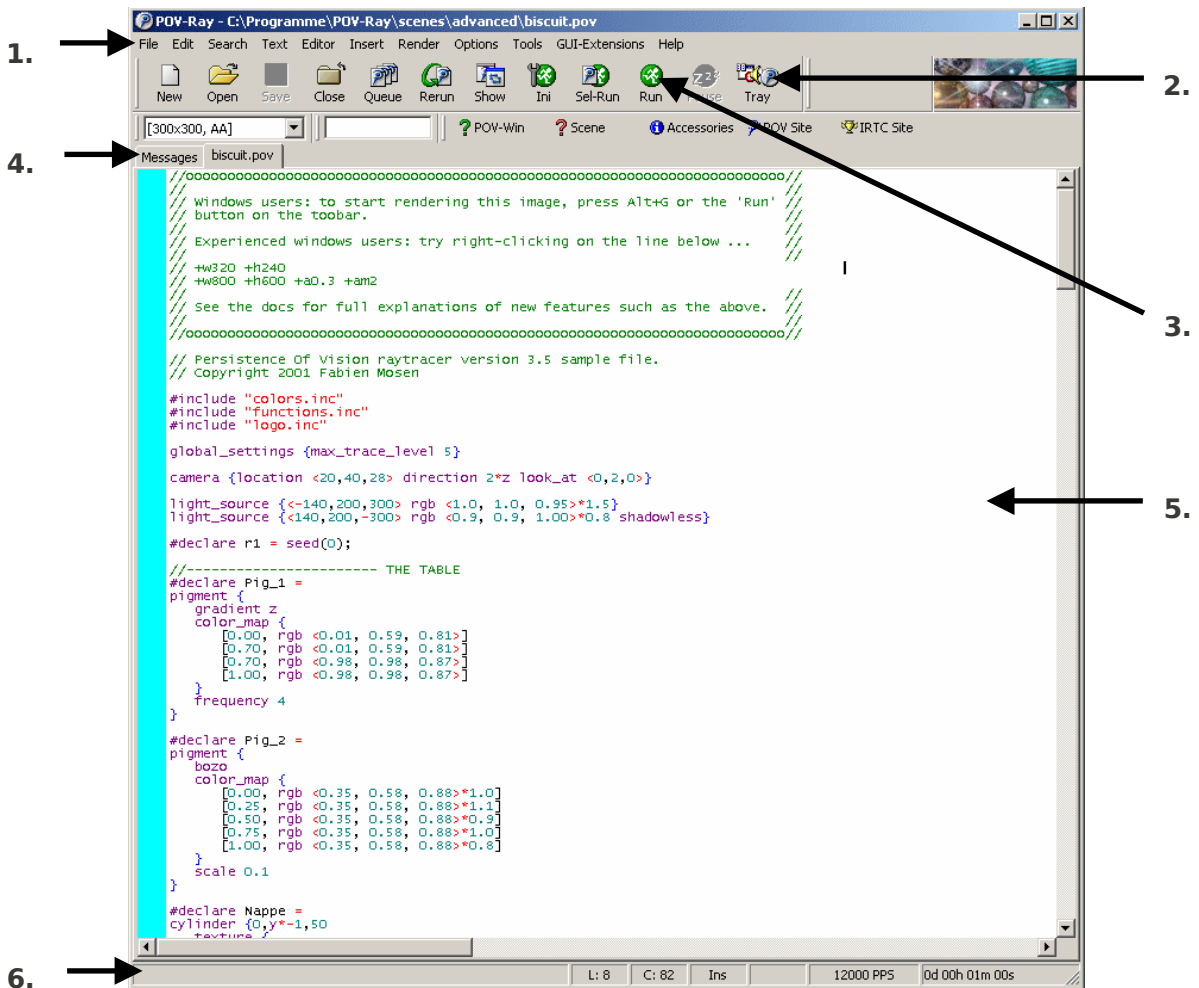
Oben links: Ein Bild aus dem Schnupperstudium des Dep. Informatik Uni Hamburg
Oben rechts: http://wikimediafoundation.org/wiki/File:3D_Cartesian_coordinates.PNG
Mitte: http://commons.wikimedia.org/wiki/File:Glasses_800.png
Unten links: angepasst von http://en.wikipedia.org/wiki/File:Ray_trace_diagram.svg
Unten rechts: www.haukemorisse.de

Version: September 2010

Die Arbeitsoberfläche von POV-Ray

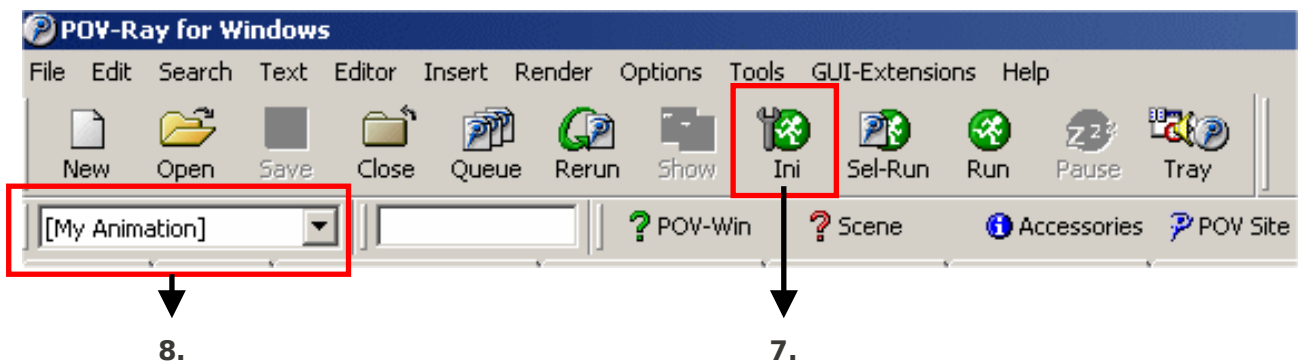
Wichtige Teile der Arbeitsoberfläche sind:

1. *Menüleiste* - enthält alle Möglichkeiten
2. *Symbolleiste* - enthält Buttons für die wichtigsten Befehle
3. *Run-Button* - damit lässt man die programmierten Szenen berechnen
4. *Datei-Reiter* - hier findet man die offenen Dateien
5. *Editor-Fenster* - hier schreibt man die Programme rein
6. *Statusleiste* - hier werden Meldungen aller Art angezeigt, z.B. Fehlermeldungen



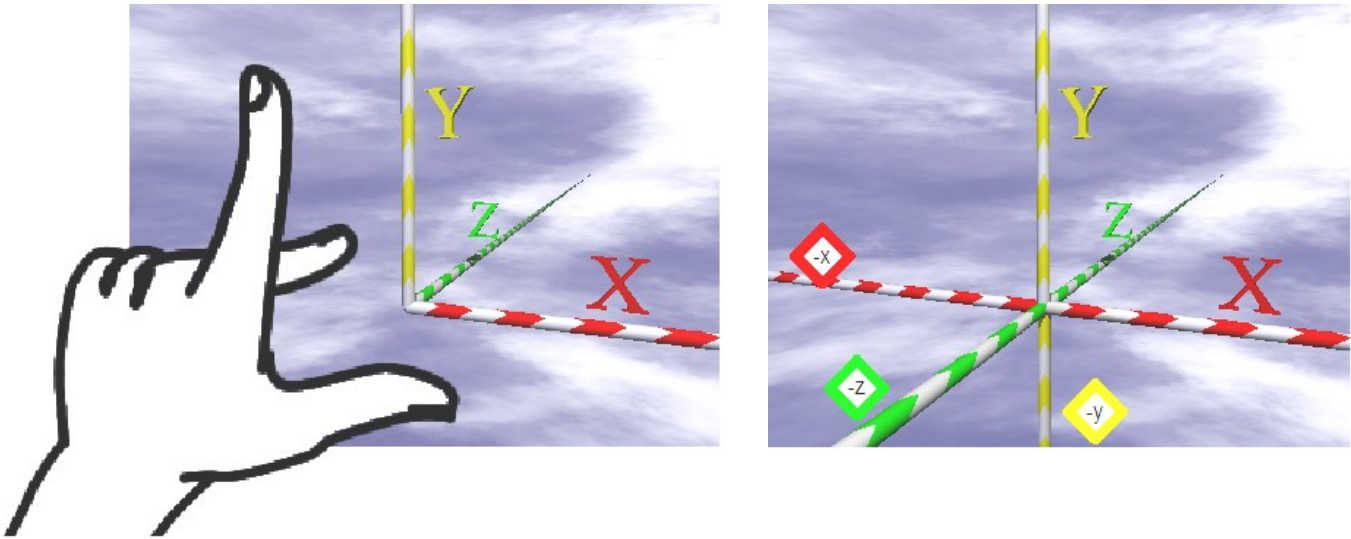
Außerdem sind 2 Teile der Arbeitsoberfläche wichtig, um **Bewegung sichtbar zu machen**:

7. *Ini-Button* - damit kann man u.a. die Quickres.ini auswählen
8. *Auswahlbox für Render-Optionen* - hier kann man zwischen verschiedenen Render-Optionen wählen, z.B. mit oder ohne Animation





3D-Koordinatensystem



Um ein Objekt zu positionieren, werden 3 Koordinaten angegeben:

- **rote Achse: X-Koordinate** gibt an, wie weit rechts oder links das Objekt steht
- **gelbe Achse: Y-Koordinate** gibt an, wie weit oben oder unten das Objekt steht
- **grüne Achse: Z-Koordinate** gibt an, wie weit vorne oder hinten das Objekt steht

Die Koordinaten stehen immer in spitzen Klammern, z.B. $\langle 1, 3.5, 2 \rangle$. Dies nennt man Vektorschreibweise.

POV-Ray verwendet ein linkshändiges Koordinatensystem. Das heißt, wenn man Daumen, Zeige- und Mittelfinger spreizt, weisen die drei Finger in x-, y- und z-Richtung. Wie die Hand lässt sich auch das Koordinatensystem drehen und von verschiedenen Seiten betrachten.

Hinweis:

In POV-Ray schreibt man Zahlen **mit Punkt** und nicht mit Komma, also zum Beispiel **0.5**. Das entspricht der amerikanischen Schreibweise.

Syntax

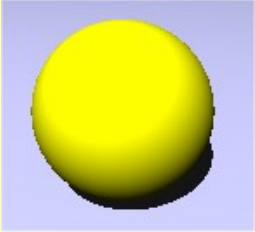
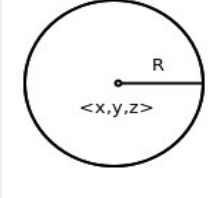
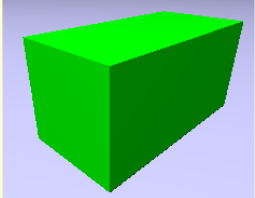
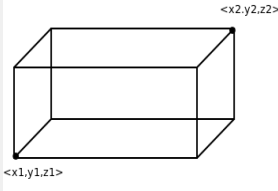
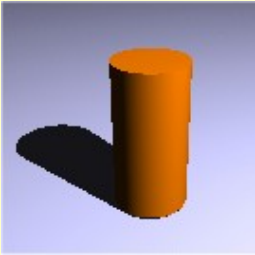
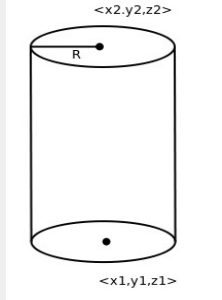
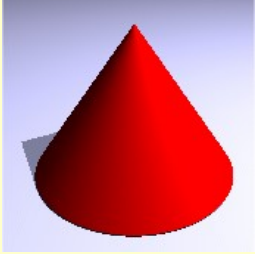
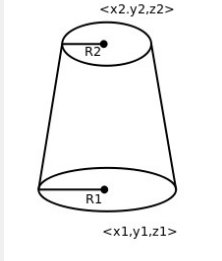
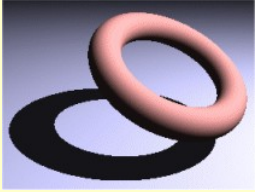
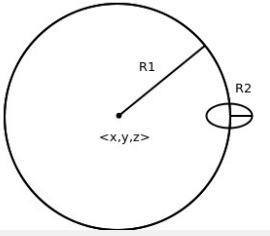
POV-Ray ist ein Programm, dem man genau beschreiben kann, was es tun soll. Da POV-Ray aber kein Deutsch versteht, muss man eine spezielle Beschreibungssprache benutzen. Wie bei Fremdsprachen muss man die Vokabeln und die Grammatik dieser Beschreibungssprache lernen.

Die Grammatik von formalen Sprachen nennt man auch ihre **Syntax**. Während Menschen einander auch dann gut verstehen, wenn sie grammatikalische Fehler machen (das passiert ständig), verstehen Computerprogramme syntaktisch fehlerhafte Befehle nicht. Das ist auch gut so, weil wir nicht wollen, dass Computer interpretieren, was wir von ihnen wollen, und es vielleicht falsch interpretieren (das passiert unter Menschen auch oft). Jeder syntaktisch richtige Ausdruck einer formalen Sprache hat genau eine Bedeutung. Die Bedeutung von formalen Sprachen nennt man ihre **Semantik**.

Beispiel:

Die Syntax für einen Vektor, der aus drei Koordinaten x, y und z besteht, ist oben beschrieben worden: $\langle x, y, z \rangle$. Die Semantik des Kommas ist, die Zahlen voneinander zu trennen. Deshalb kann das Komma nicht auch zur Darstellung von Dezimalzahlen verwendet werden. Dafür benutzt man den Punkt. (Siehe auch Seite 21: Fehler)

Objekte

Name	Bild	Beispiel	Schematische Darstellung	Syntax
Kugel (sphere)		<pre>Sphere { <0, 1, 0>, 1 pigment {color Yellow} }</pre>		<pre>sphere { <x, y, z> Radius pigment {color Farbe} }</pre>
Quader (box)		<pre>box { <0, 0, 0> <1, 1, 2> pigment {color Green} }</pre>		<pre>box { <x1, y1, z1> <x2, y2, z2> pigment {color Farbe} }</pre>
Zylinder (cylinder)		<pre>cylinder { <0, 0, 0> <0, 0.8, 0>, 0.2 pigment {color Orange} }</pre>		<pre>cylinder { <x1, y1, z1> <x2, y2, z2> Radius pigment {color Farbe} }</pre>
Kegel (cone)		<pre>cone { <0, 0, 0>, 0.7 <0, 1, 0>, 0 pigment {color Red} }</pre>		<pre>cone { <x1, y1, z1>, Radius1 <x2, y2, z2>, Radius2 pigment {color Farbe} }</pre>
Reifen (torus)*		<pre>torus { 1, 0.25 pigment { color Pink } }</pre>		<pre>torus { Hauptradius, Nebenradius pigment {color Farbe } }</pre>

* Der Torus hat keine Positionsangabe und liegt immer um den Nullpunkt $\langle 0,0,0 \rangle$ in der XZ-Ebene. Um die Position und Ausrichtung zu ändern muss verschoben oder gedreht werden (siehe Seite 12: Bewegung).

Gerade Ebene

Mit **plane** lässt sich eine unendlich große Ebene in die Szene legen. Deshalb ist die Syntax etwas anders als bei gewöhnlichen Körpern: es wird ein *Normalenvektor* – das ist ein Vektor, der senkrecht zu der Ebene steht – und zusätzlich der Abstand der Ebene vom Ursprung angegeben:

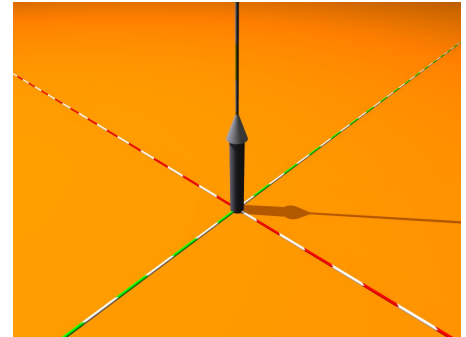
Syntax:

```
plane {
  Normalenvektor, Abstand vom Ursprung

  Farbe
}
```

Beispiel:

```
plane {
  <0, 1, 0>, -0.01
  pigment {color Orange}
}
```



Ebene mit Normalenvektor

Im Bild stellt der schwarze Pfeil den Normalenvektor dar. Man sieht, dass die Pfeilspitze senkrecht von der Ebene weg zeigt und zwar in Richtung der Y-Achse. Die Ebene hier liegt also senkrecht zur Y-Achse und damit horizontal. Die zweite Zahl ist -0.01, d.h. die Ebene ist um 0.01 nach unten verschoben.

Man könnte zur **Vereinfachung** auch angeben:

```
plane {
  y, -0.01
  pigment {color LightBlue}
}
```

Hinweis:

Bei einem negativen Abstand zum Ursprung wird die Ebene in die dem Normalenvektor entgegengesetzte Richtung verschoben. Dabei ist darauf zu achten, dass vor der Zahl ein Komma stehen muss. Sonst gibt es einen Syntaxfehler (siehe Seite 21).

Dies liegt daran, dass das **Minuszeichen zwei Bedeutungen** hat: einerseits kennzeichnet es eine negative Zahl, andererseits kann es zwischen zwei Zahlen oder zwei Vektoren stehen und kennzeichnet in diesem Falle eine Differenz!

Das Komma macht deutlich, dass wir hier keine Differenz meinen.

Vordefinierte Himmel

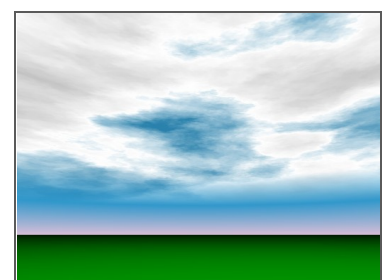
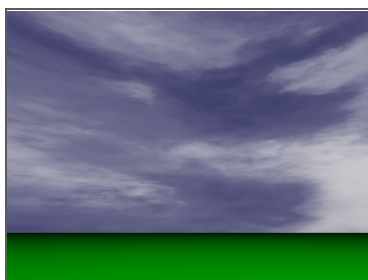
Um einen schönen Himmel in die Szene einzubauen, muss die Datei „skies.inc“ importiert werden. Dazu muss am Anfang der Szenenbeschreibung eine Zeile stehen, die so aussieht: **#include "skies.inc"**

Ein sehr realistisches Himmelsgewölbe bekommt man dann mit **sky_sphere**, z. B. mit:

```
sky_sphere { S_Cloud5 }
```

oder

```
sky_sphere{ S_Cloud2 }
```





Vordefinierte Farben

Um die vordefinierten englischen Namen für Farben benutzen zu können, z.B. pigment {color **Red**}, muss zuvor die Datei „colors.inc“ importiert werden. Das passiert mit der Zeile: **#include "colors.inc"**

In der "colors.inc" sind z.B. folgende Farben vordefiniert:

POV-Ray-Bezeichnung	Aussehen (ungefähr)	Bezeichnung deutsch
Red		rot
Green		grün
Blue		blau
Yellow		gelb
Cyan		türkis
Magenta		lila
Clear		durchsichtig
White		weiß
Black		schwarz
Orange		orange
Aquamarine		mintgrün
BlueViolet		violett
Brown		braun
IndianRed		dunkelrot
Khaki		khaki
ForestGreen		dunkelgrün
MidnightBlue		mitternachtsblau
SkyBlue		himmelblau

und viele mehr... !

Die Farbe, die wir festlegen, gibt genau an, wie ein Objekt aussehen soll, wenn es von einer hellen weißen Lichtquelle beleuchtet wird.

Wenn wir ein Bild von einer Kugel auf Papier malen würden, würden wir für die helle und dunkle Seite jeweils helle und dunkle Farben wählen, um so Licht und Schatten auf der Kugel darzustellen.

Bei POV-Ray sorgt das Raytracing dafür, dass Lichtreflexe und Schatten richtig dargestellt werden. Du suchst nur die Farbe aus und **POV-Ray kümmert sich um die realistische Darstellung** - und zwar in Abhängigkeit von allen Lichtquellen, Schatten und Reflexionen.



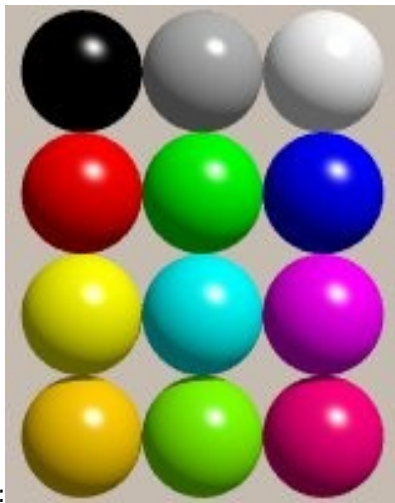
Selbstgemischte Farben

Man kann auch selbst jede beliebige Farbe aus Rot, Grün und Blau mischen. Dazu verwendet man die Anweisung:

```
pigment {  
  color rgb <Rot-Anteil, Grün-Anteil, Blau-Anteil>  
}
```

Die in der spitzen Klammern einzusetzenden 3 Zahlen für die Farbstärken müssen Werte zwischen 0.00 und 1.00 sein. Dabei bedeutet 1.00 = 100% und 0.00 = 0% der jeweiligen Farbe.

Beispiele:



color rgb<0,0,0> = Schwarz ("color Black")
color rgb<0.5,0.5,0.5> = Grau ("color Grey")
color rgb<1,1,1> = Weiß ("color White")

color rgb<1,0,0> = Rot ("color Red")
color rgb<0,1,0> = Grün ("color Green")
color rgb<0,0,1> = Blau ("color Blue")

color rgb<1,1,0> = Gelb ("color Yellow")
color rgb<0,1,1> = HimmelBlauGrün oder Cyan
color rgb<1,0,1> = RotViolett oder Magenta

color rgb<1,0.65,0> = OrangeGelb
color rgb<0.25,1,0> = Gelbgrün ("YellowGreen")
color rgb<1,0,0.25> = Pink

Farben aufhellen oder abdunkeln

color rgb <1, 0, 0> = **Rot**
color rgb <1, **0.5**, **0.5**> = **Hellrot**
color rgb <**0.5**, 0, 0> = **Dunkelrot**

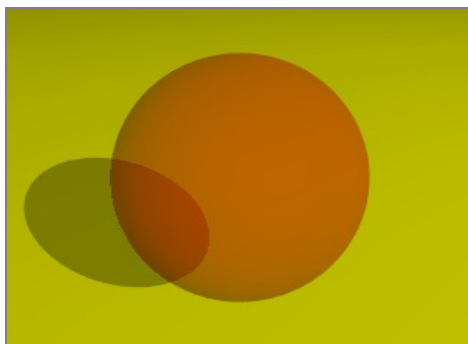
Transparenz

color rgbf <1, 0, 0, **0.5**> = **halb durchsichtiges Rot (filter)**

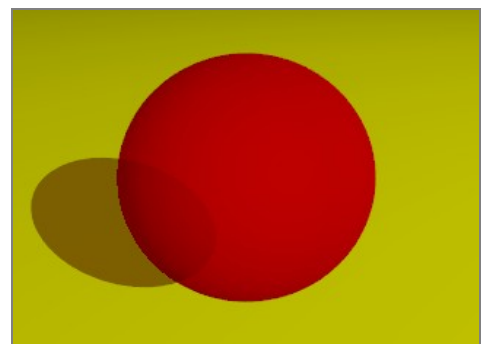
color rgbt <1, 0, 0, **0.5**> = **halb durchsichtiges Rot (transmit)**

Finde heraus, worin der Unterschied zwischen **rgbf** und **rgbt** besteht! Wie lässt sich das Aussehen beschreiben?

transmit

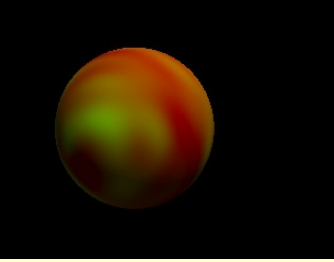
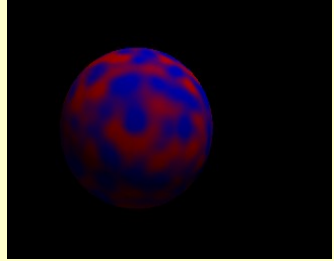
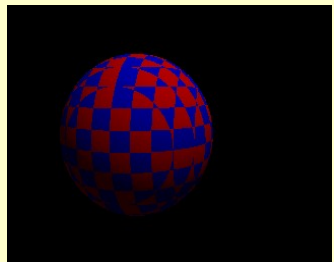


filter






Muster

Name	Bild	Beispiel	Syntax
Wellen (waves)		<pre>sphere {<0, 0, 2>, 5 pigment { waves color_map { [0.3, color Red] [0.7, color Yellow] } } }</pre>	<pre>pigment { waves color_map { [Anteil1, color Farbe1] [Anteil2, color Farbe2] } }</pre>
Flecken (bozo)		<pre>sphere {<0, 0, 2>, 5 pigment { bozo color_map { [0.3, color Red] [0.7, color Blue] } } }</pre>	<pre>pigment { bozo color_map { [Anteil1, color Farbe] [Anteil2, color Farbe] } }</pre>
Schachbrett (checker)		<pre>sphere {<0, 0, 2>, 5 pigment { checker color Red color Blue } }</pre>	<pre>pigment { checker color Farbe color Farbe }</pre>

Die Anteile der Farben in der **Color_Map** setzen sich folgendermaßen zusammen:
Stell Dir einen Zahlenstrahl vor, der **von 0.0 bis 1.0** geht.

Bei **Anteil1** ziehst Du einen Strich und malst den ersten Bereich mit **Farbe1** an. Bei **Anteil2** ziehst Du wieder einen Strich und malst alles zwischen dem Strich und dem Ende des Zahlenstrahls mit **Farbe2** an. In dem Bereich zwischen **Anteil1** und **Anteil2** geht **Farbe1** langsam in **Farbe2** über.

Das Muster (bozo, waves) gibt an, wie diese Farben durcheinander gewirbelt werden. Dabei sind auch **mehr als 2 Farben möglich**.

Bild	Beispiel	Definition
 <p>0.0 0.3 0.7 1</p>	<pre>pigment { bozo color_map { [0.3, color Red] [0.7, color Blue] } }</pre>	<pre>pigment { bozo color_map { [Anteil1, color Farbe1] [Anteil2, color Farbe2] } }</pre>



Oberflächen

Eine Oberfläche besteht im wesentlichen aus:

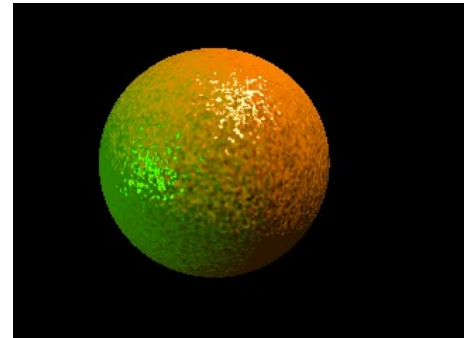
- Farbe und Durchsichtigkeit ("pigment")
- Oberflächen-Struktur ("normal")
- Oberflächen-Beschaffenheit ("finish")

Die letzten beiden sind allerdings optional, d.h. sie müssen nicht unbedingt angegeben werden.

Beispiel:

```

pigment { color Orange }
normal { bumps 0.5 scale 0.05 }
finish { ambient 0.15 diffuse 0.85 phong 1.0 }
    
```



Die **Oberflächen-Struktur** bezieht sich auf die Rauheit der Oberfläche. Wird diese nicht definiert, so bleibt die Oberfläche glatt. Einige interessante Effekte sind z.B.:

- **bumps** = Beulen
- **dents** = Zacken
- **waves** = Wellen
- **ripples** = gleichmäßige Wellen
- **wrinkles** = Knitter, Falten

Die **Oberflächen-Beschaffenheit** bezieht sich auf die Helligkeit und den Glanz, also z.B. auf Reflexionseigenschaften, Umgebungslichtstärke, Streulichtstärke, Lichtreflexgröße, Brechungseigenschaften bei Gläsern uvm. Im Einzelnen sind dies u.a.:

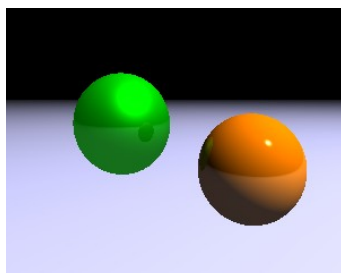
- **ambient** 0.10 = Schattenhelligkeit der Farbe (simuliert die durch indirekte Beleuchtung hervorgerufene Helligkeit)
- **diffuse** 0.90 = Anteil der durch direkte Beleuchtung erzeugten Helligkeit
- **reflection** 0.15 = Spiegelungs-Anteil der Helligkeit
- **phong** 1.0 = Erzeugung von Glanzlichtern

Sind einzelne Effekte nicht sofort zu erkennen, so liegt es meist daran, dass sie **zu groß** eingestellt wurden. Dies lässt sich durch Hinzufügen z.B. der Anweisung **scale 0.05** oder ähnlicher Werte ändern - man muss die richtige Größe probieren!

Für einige dieser Angaben ist es außerdem wichtig, dass die folgende Zeile am Anfang der Datei eingegeben wird:

```
global_settings { assumed_gamma 1.0 }
```

Auch für die Oberflächen-Beschaffenheit gibt es **vordefinierte Werte**, die man benutzen kann. Dafür muss man die Datei "*finish.inc*" bzw. "*metals.inc*" importieren:



Beispiele:

```
finish { Phong_Glossy }
```

```
finish { F_MetalA }
```

Einige vordefinierte Werte:

Dull, Shiny, Glossy, Phong_Dull, Phong_Shiny, Glass_Finish, Mirror

Metalle:

F_MetalB, ... F_MetalE

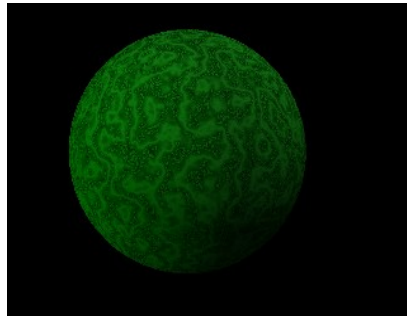
Vordefinierte Oberflächen

Es gibt aber auch schon **vordefinierte Oberflächen**, die man verwenden kann. Dazu muss man zunächst die Datei "textures.inc" importieren:

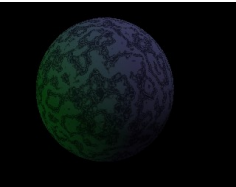
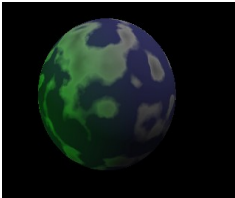
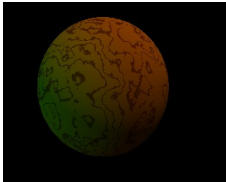
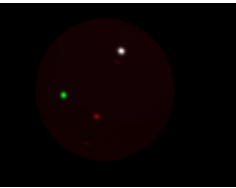
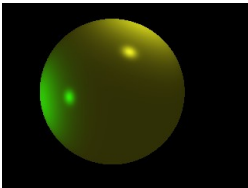
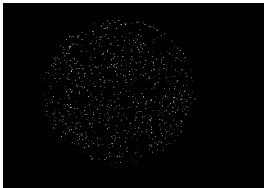
#include "textures.inc"

Anschließend kann man die gewünschte Oberfläche aufrufen - im folgenden Fall ist es 'Jade':

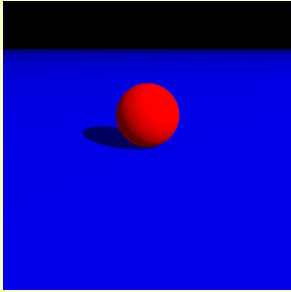
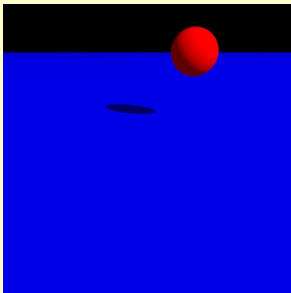
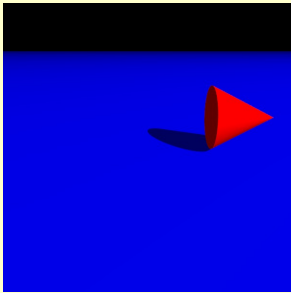
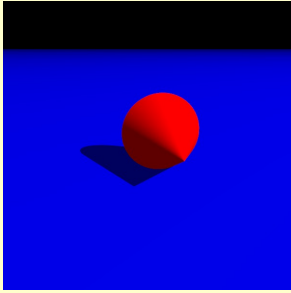
```
sphere {
  <-6, 1, 4>, 1
  texture {Jade}
}
```



Weitere **Beispiele:**

Steinoberflächen	Himmel	Holz
 <p>Jade Red_Marble Blue_Agate Sapphire_Agate PinkAlabaster</p>	 <p>Blue_Sky Clouds Blue_Sky3</p>	 <p>Cherry_Wood Tan_Wood Tom_Wood Rosewood Sandalwood</p>
Glas	Metall	Spezialeffekte
 <p>Glass Green_Glass Ruby_Glass</p>	 <p>Metal Soft_Silver Gold_Metal</p>	 <p>Water Peel Candy_Cane Cork Starfield</p>

Bewegung

Name	Bild	Beispiel	Syntax
Verschieben (translate)		<pre>sphere { <0, 0.5, 0>, 0.5 pigment { color Red } }</pre>	translate <x, y, z > x, y und z stellen den jeweiligen Abstand dar
		<pre>sphere { <0, 0.5, 0>, 0.5 pigment { color Red } translate <1, 1, 2> }</pre>	
Drehen (rotate)		<pre>cone { <1,0.5,0>,0.5, <2,0.5,0>,0 pigment { color Red } }</pre>	rotate <x, y, z> x, y und z stellen den jeweiligen Winkel dar
		<pre>cone { <1,0.5,0>,0.5, <2,0.5,0>,0 pigment { color Red} rotate <0, 80, 0> }</pre>	

Diese Abbildungen können sowohl auf die Körpergeometrie eines Objektes als auch z.B. auf eine Textur bzw. deren Bestandteile angewandt werden.



Verschieben

Das Verschieben mit **translate** bewirkt eine Verschiebung relativ zur jeweiligen Ausgangslage.

Beispiel:

translate $\langle -2, 1, 3 \rangle$ verschiebt den Körper um 2 Einheiten nach links (x-Richtung), um 1 Einheit nach oben (y-Richtung) und um 3 Einheiten nach hinten (z-Richtung).

Rotieren / Drehen

Das Drehen mit **rotate** bewirkt eine Drehung nach der "Linke-Daumen-Regel" um die jeweilige Koordinaten-Achse. Die Drehung bezieht sich also nicht auf das Objekt, sondern immer nur auf die Koordinatenachsen!

Beispiel:

rotate $\langle 0, 45, 0 \rangle$ dreht einen Körper um die Y-Achse um 45 Grad

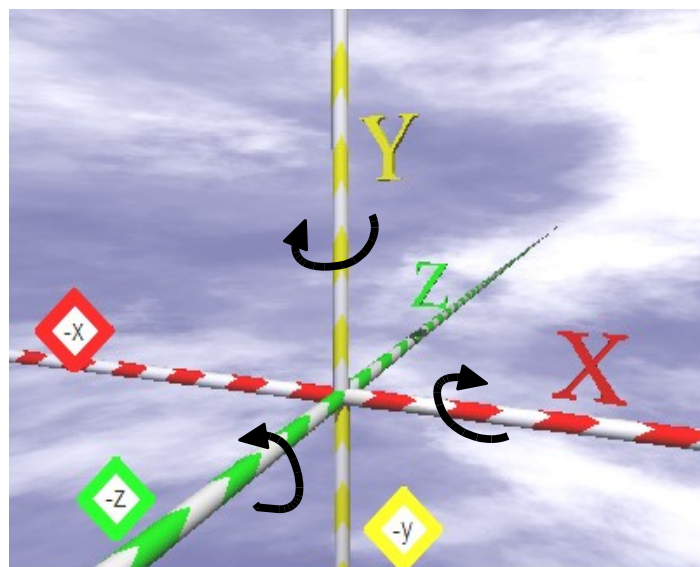


Achtung:

Falls mehrere Drehungen angegeben werden, werden diese in der aufgezählten Reihenfolge ausgeführt! Die Hintereinander-Ausführung von Drehungen um verschiedene Achsen ist nicht kommutativ, das heißt nicht beliebig vertauschbar!

Drehrichtung

Die Drehrichtung von Winkeln ist im "Gegenuhrzeigersinn" definiert, sofern man in die jeweilige positive Achsenrichtung blickt:





Eine weitere Art der Modifikation von Objekten: Skalierung

Name	Bild	Beispiel	Definition
Skalieren (scale)		<pre> box { <-1,0,-1> <1,1,1> pigment {color Red} } box { <-1,0,-1> <1,1,1> pigment {color Red} scale <1.5, 1, 0.2> } </pre>	<p>scale <x,y,z></p> <p>x, y und z stellen den jeweiligen Vergrößerungsfaktor dar</p>

Skalierung bedeutet: Vergrößern oder Verkleinern von Objekten. Dabei kann für jede Koordinaten-Richtung ein separater Skalierungsfaktor angegeben werden. Alle Koordinaten, die das skalierte Objekt beschreiben, sind einfach aus den Koordinaten des Ursprungsobjekts zu errechnen, indem man sie mit den Skalierungsfaktoren multipliziert.

$$\langle x_{neu}, y_{neu}, z_{neu} \rangle = \langle x * SkalierungX, y * SkalierungY, z * SkalierungZ \rangle$$

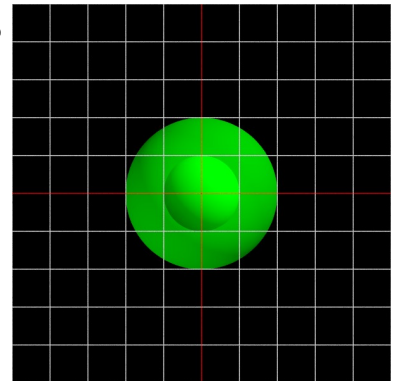
Häufig wird auch in alle Richtungen mit dem gleichen Faktor skaliert, weswegen es eine abkürzende Schreibweise gibt: *scale 2* entspricht *scale <2,2,2>*

Beispiel: Eine Kugel mit Mittelpunkt $\langle 0,0,0 \rangle$ und Radius 1, wird um den Faktor 2 skaliert.

```

sphere{<0,0,0>,1
  pigment{Green}
  scale <2,2,2>
}
                    
```

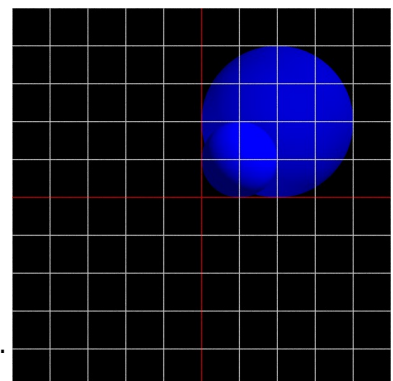
Daraus erhält man eine Kugel mit Mittelpunkt $\langle 0*2,0*2,0*2 \rangle = \langle 0,0,0 \rangle$ und dem Radius 2.



Achtung: Skaliert man etwas, dessen Mittelpunkt nicht im Ursprung $\langle 0,0,0 \rangle$ liegt, verschiebt sich die Position des Objekts. Hätte die Kugel aus unserem Beispiel den Mittelpunkt $\langle 1,1,0 \rangle$, würde sie nach der Skalierung mit Faktor 2 bei

$$\langle 1*2,1*2,0*2 \rangle = \langle 2,2,0 \rangle$$

liegen. Um dies zu vermeiden kann man das zu skalierende Objekt zunächst auf den Ursprung verschieben und nach der Skalierung zurückverschieben.





Bewegungen sichtbar machen

Verschieben und Drehen bewirken Bewegungen, wobei zunächst nur deren Endergebnis sichtbar ist. Will man die Bewegungen sichtbar machen, sind 2 Dinge wichtig: die Datei **Quickres.ini** und der **Takt** (clock).

Quickres.ini

Die Quickres.ini ist eine Datei, in der vor allem folgende Einstellungen vorgenommen werden:

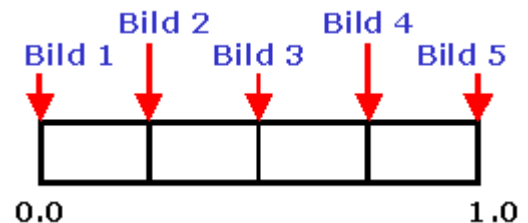
Name der Erzeugungskonfiguration	→	[myAnimation]
Größe des erzeugten Bildes	→	Width=300 Height=300
Nummerierung und Anzahl der Bilder	→	Initial_Frame = 0 Final_Frame = 4
Speicherplatz der Bilder	→	output_file_name=Film\
Anfangs- und Endwert des Taktes	→	Initial_Clock = 0.0 Final_Clock = 1.0

Der Takt

Der Takt wird durch die folgenden Angaben festgelegt:

Initial_Frame = 0
Final_Frame = 4

Initial_Clock = 0.0
Final_Clock = 1.0



Damit wird festgelegt, dass die innere Uhr von 0.0 bis 1.0 läuft und dass in dieser Zeit 5 Bilder (nämlich von 0 bis 4) erstellt werden.

Im Programm hat man dann Zugriff auf den Takt über die Variable **clock**.

Beispiel:

translate < -0.4, (**clock***10)-13, 1.3 >

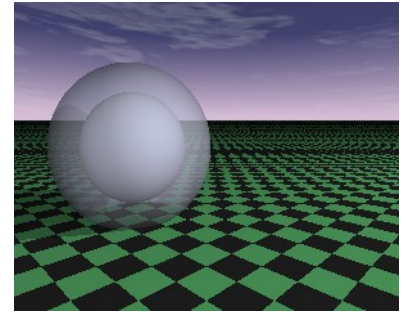
Im obigen Beispiel hat clock den Wert 0,25, wenn das zweite Bild erzeugt wird, d.h. der Befehl lautet in diesem Moment:

translate < -0.4, (**0.25***10)-13, 1.3 >

Beispiele für die Verwendung von clock

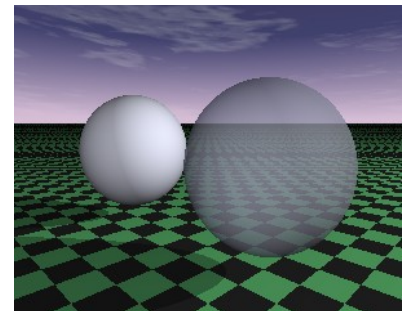
Der Radius der Kugel wird in Abhängigkeit von clock größer, d.h. die **Kugel wächst**:

```
sphere {
  <0, 2, -4>, 2+1*clock
  pigment {color Silver}
}
```



Der Z-Parameter von translate wird in Abhängigkeit von clock größer, d.h. die **Kugel wandert** die Z-Achse lang:

```
sphere {
  <0, 2, -4>, 2
  pigment {color Silver}
  translate <0, 0, 6*clock >
}
```

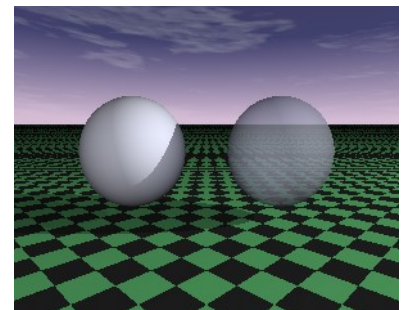


Hinweis:

Die Kugel scheint zwar größer zu werden, aber sie wird nicht größer, sondern kommt immer näher und erscheint deswegen größer!

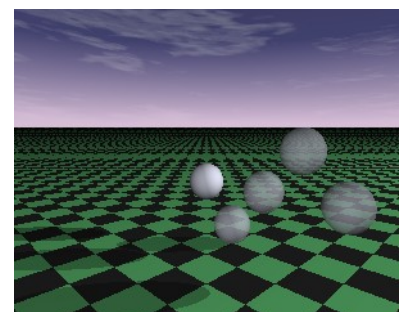
Der Y-Winkel wird in Abhängigkeit von clock größer, d.h. die **Kugel rotiert** um die Y-Achse:

```
sphere {
  <0, 2, -4>, 2
  pigment {color Silver}
  rotate <0, 90*clock, 0>
}
```



Die Y- und Z-Werte von translate verändern sich in Abhängigkeit von clock, wobei Y außerdem eine Sinus-Kurve beschreibt, d.h. die **Kugel hüpf** auf einer **Sinus-Kurve** zunächst runter, dann hoch und dann wieder runter:

```
sphere {
  <0, 2, -4>, 2
  pigment {color Silver}
  translate <0, 1+sin(-2*pi*clock), 4*clock>
}
```





Komplexe Objekte

Komplexe Objekte entstehen durch **Verknüpfung** mehrerer primitiver Objekte. Hierbei werden die primitiven Objekte durch **Mengen-Operationen** miteinander verknüpft. Diese Technik nennt man **CSG-Technik** (Constructive Solid Geometry).

Komplexe Objekte benutzt man, wenn man die Teile **mehrerer Objekte gemeinsam einfärben oder zusammen transformieren** (mittels scale, rotate oder translate) will.

Die **wichtigsten CSG-Operationen** sind:

- Vereinigung
- Differenz
- Schnitt

Vereinigung

Dafür stehen zwei Anweisungen zur Verfügung: **union** und **merge**

Hinweis:

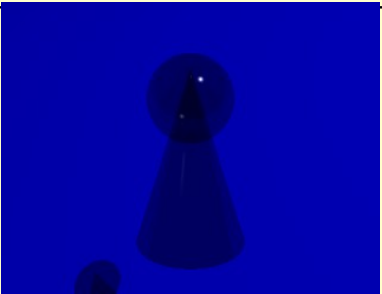
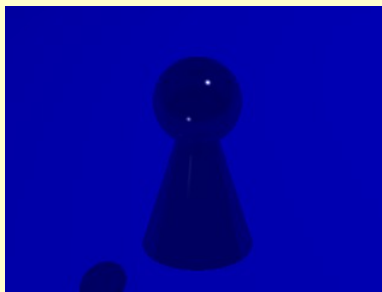
Beide Anweisungen werden zum Verbinden von Objekten benutzt, wobei die zu verbindenden Objekte **nicht notwendig räumlich zusammenhängen müssen!**

Der **Unterschied** zwischen union und merge wird nur sichtbar bei **transparenten Körpern**. merge löst nämlich die inneren Trennungsfächen der verbundenen Körper auf.




Achtung:

Bei nichttransparenten Körpern ist merge zu meiden, da es teilweise wesentlich längere Rechenzeiten benötigt!

Name	Bild	Beispiel	Definition
union		<pre> union { cone{ <0, 0, 0> 0.4 <0, 1.2, 0> 0 texture { Glass } } sphere{ <0, 1, 0> 0.3 texture { Glass } } } </pre>	<pre> union { 1.Objekt 2. Objekt } </pre>
merge		<pre> merge { cone{ <0, 0, 0> 0.4 <0, 1.2, 0> 0 texture { Glass } } sphere{ <0, 1, 0> 0.3 texture { Glass } } } </pre>	<pre> merge { 1. Objekt 2.Objekt } </pre>

Differenz

Mittels der Differenz lassen sich aus vorhandenen Körpern Löcher, Ausbuchtungen, Abschrägungen, Einkerbungen und dergleichen herausfräsen. Das "Loch" erhält jeweils die Textur der abgezogenen Körpers.

Name	Bild	Beispiel	Definition
Differenz (difference)		<pre> difference { sphere{<0, 0, 0> , 1 pigment{color Red} } sphere{<1, 0, 0> , 1 pigment{color Yellow} } } </pre>	<pre> difference { 1. Objekt 2. Objekt } </pre>



Achtung:

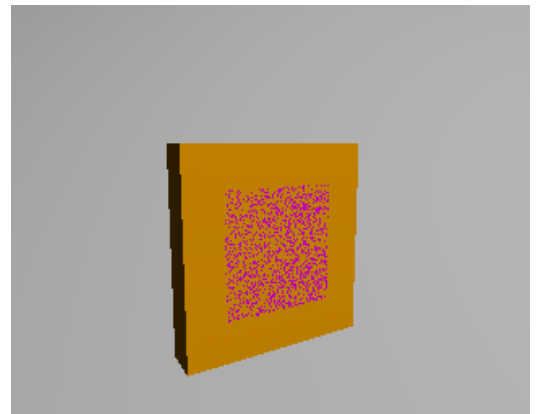
Bei der Differenz kann folgendes **Problem** eintreten:

Reicht der abzuziehende Körper nicht eindeutig über den Körper von dem er abgezogen wird hinaus, sondern liegen die Grenzflächen genau aufeinander, so kann der Raytracer nicht klar entscheiden, welche Fläche noch übrig bleiben soll. Als Folge bleibt eine mehr oder weniger perfekte Trennfläche stehen, d.h. die Differenz scheint nicht richtig zu funktionieren! Dies ist kein Fehler des Programms, sondern eine prinzipielle Beschränkung durch die Rundungsfehler, welche bei Berechnungen mit Computern auftreten.

Beispiel - das Problem:

```

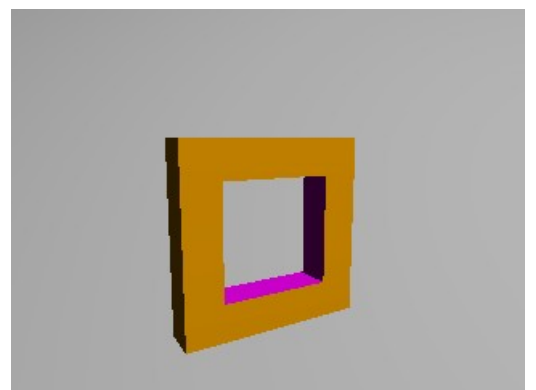
difference {
  box {<-0.5, 0, -0.1>, < 0.5, 1.0, 0.1>
    texture { pigment { color rgb <1, 0.65, 0> }
      finish { ambient 0.15 diffuse 0.85 }
    }
  }
  box {<-0.3, 0.2, -0.1>, < 0.3, 0.8, 0.1>
    texture { pigment { color Magenta }
      finish { ambient 0.15 diffuse 0.85 }
    }
  }
  rotate<0, -30, 0> translate<0, 0, 0>
}
    
```



Beispiel - die Problemlösung:

```

#declare D = 0.0001; // nur ein kleines bisschen!
difference {
  box {<-0.5, 0, -0.1>, < 0.5, 1.0, 0.1>
    texture { pigment { color rgb <1, 0.65, 0> }
      finish { ambient 0.15 diffuse 0.85 }
    }
  }
  box {<-0.3, 0.2, -0.1-D>, < 0.3, 0.8, 0.1+D>
    texture { pigment { color Magenta }
      finish { ambient 0.15 diffuse 0.85 }
    }
  }
  rotate<0, -30, 0> translate<0, 0, 0>
}
    
```

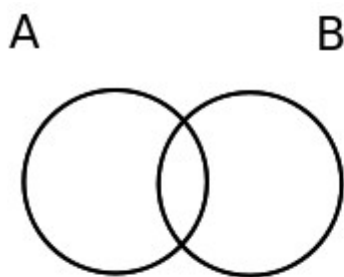


Schnitt

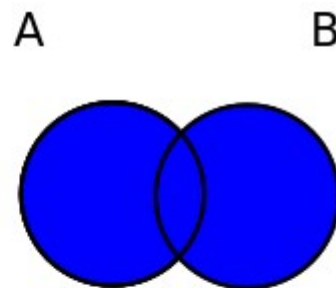
Die Schnittmenge zweier Körper enthält diejenigen Körperbereiche, die beiden Körpern gemeinsam sind, also die Bereiche in denen sich alle dabei verwendeten Objekte überlappen.

Name	Bild	Beispiel	Definition
Schnitt (intersection)		<pre> intersection { sphere{<0, 0, 0> , 1 pigment{color Red} } sphere{<1, 0, 0> , 1 pigment{color Yellow} } } </pre>	<pre> intersection { 1.Objekt 2. Objekt } </pre>

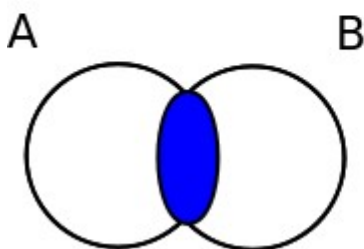
Übersicht zu den Mengen-Operatoren



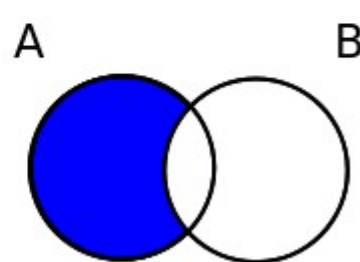
zwei Mengen A und B



die Vereinigungsmenge



die Schnittmenge



die Differenzmenge



Formatierung

Formatierung bedeutet einen Quellcode in ein bestimmtes Format zu bringen, um ihn besser lesbar zu machen und damit bestimmte Fehler beim Schreiben (und Denken) von Programmen zu verhindern. Es handelt sich dabei also um eine Prävention.

Beispiel:

Vergleiche die folgenden 2 Quelltexte - welcher ist besser zu lesen und zu verstehen?

Text 1:

```
#include "colors.inc" camera {location <2,16,5> sky <0,0,1> look_at <0,0,0>} light_source
{<10,15,20>color White} plane { <0,0,1>, -0.01 pigment {color Green}} sphere { <0,0,2>,
2 pigment {color Red}}
```

Text 2:

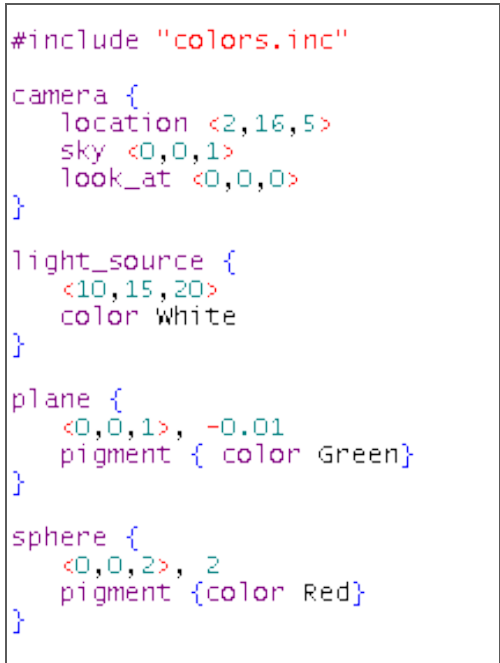
```
#include "colors.inc"

camera {
    location <2,16,5>
    sky <0,0,1>
    look_at <0,0,0>
}

light_source {
    <10,15,20>
    color White
}

plane {
    <0,0,1>, -0.01
    pigment { color Green}
}

sphere {
    <0,0,2>, 2
    pigment {color Red}
}
```



```
#include "colors.inc"

camera {
    location <2,16,5>
    sky <0,0,1>
    look_at <0,0,0>
}

light_source {
    <10,15,20>
    color White
}

plane {
    <0,0,1>, -0.01
    pigment { color Green}
}

sphere {
    <0,0,2>, 2
    pigment {color Red}
}
```

Syntax-Highlighting

Der Text-Editor von POV-Ray erhöht die Lesbarkeit zusätzlich, indem die unterschiedlichen Ausdrücke in verschiedenen Farben dargestellt werden. Dies nennt man Syntax-Highlighting: Schlüsselwörter, d.h. die bekannten POV-Ray-Befehle, erscheinen violett, Kommentare grün, Zeichenketten rot usw. Wenn also ein Teil rot dargestellt wird, der eigentlich andere Farben haben müsste, kann man leicht erkennen, dass ein Anführungszeichen vergessen wurde. Syntax-Highlighting hilft also dabei Fehler zu vermeiden.

Fehler

Kaum ein/e Programmierer/in erstellt auf Anhieb fehlerfreien Code. Deshalb ist es wichtig zu wissen, welche Fehlerarten auftreten können und wie darauf reagiert werden kann. Mögliche Fehlerarten sind z.B. Syntaxfehler und logische Fehler.

Syntaxfehler

Syntaxfehler sind Fehler, die am häufigsten auftreten, aber auch am schnellsten zu korrigieren sind.

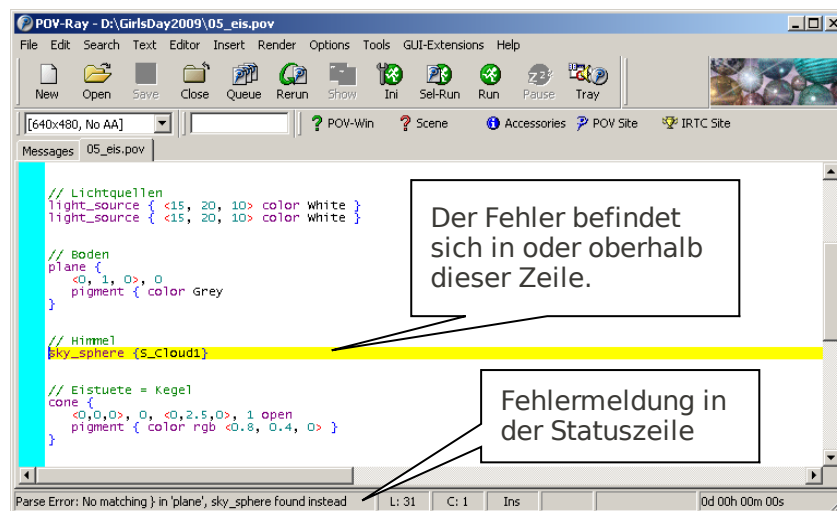
Typische Syntaxfehler:

- Tippfehler
- Fehler bei Zeichen- oder Klammersetzung
- falsche Groß-/Kleinschreibung

Ein Beispiel für einen Syntaxfehler in POV-Ray:

```
light_source {
  <10,15,20>
  color White
```

Der Fehler besteht darin, dass die schließende geschweifte Klammer vergessen wurde.



Ein typischer Fehler ist die fehlende schließende Klammer. Hier fehlt sie ein paar Zeilen über dem gelben Balken bei *pigment* in der *plane*-Anweisung.

Die entsprechende Fehlermeldung lautet: "No matching } in 'plane', sky_sphere found instead."

Logische Fehler

Logische Fehler sind oft nur schwer zu finden (wenn überhaupt!), da sie im Normalfall keine Fehlermeldungen erzeugen. Das Programm ist formal richtig, verhält sich aber anders als erwartet.

Logische Fehler können sein:

- Bei einer Berechnung wird ein Pluszeichen statt einem Minuszeichen verwendet.
- Es wird eine Formel benutzt, die eine falsche Zahl berechnet.

Ein Beispiel für einen logischen Fehler in POV-Ray:

Es wird vergessen, eine Lichtquelle anzugeben. Das Ergebnis ist, dass nichts zu sehen ist.

"Gut versteckte" logische Fehler können mit gut geplanten Tests aufgespürt werden, die zunächst Teilbereiche des Scripts (losgelöst vom Rest des Scripts) testen und die geforderte Funktionalität, sowie Normal- und Extremwerte testen und überprüfen.



Variablen

In der Informatik sind Variablen Behälter, in denen man Werte für Objekte speichern kann.

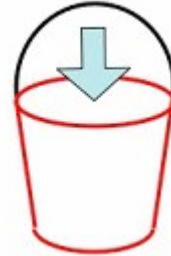
Wir definieren Variablen mit:

#declare Variablenname = Wert;

Und benutzen die Variable mit:

object { Variablenname }

oder einfach **Variablenname**



Beispiel:

```
#declare Kugel = sphere {
    < 0, 0, 0> , 1
    texture {Jade}
}; // Semikolon nicht vergessen!!
```

```
object { Kugel
    translate < 0, 4, 2>
}
```

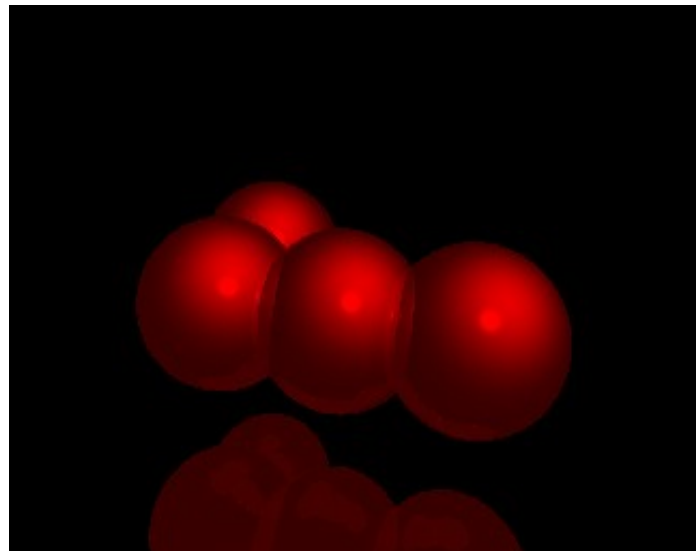
Variablen werden in der Programmierung dazu verwendet, sich viel Schreibarbeit zu ersparen und damit gleichzeitig den Quelltext lesbar zu halten.

Im obigen Beispiel wird eine komplexe Anweisung (Beschreibung einer Kugel) als Variable deklariert und später einfach nur aufgerufen (und verändert). Das ist vor allem dann wichtig, wenn die komplexe Anweisung mehrmals im Quelltext stehen soll, also z.B. wenn man eine Menge von Kugeln darstellen will.

Beispiel:

```
// Definition
#declare Kugel = sphere {
    <0, 0, 0>, 1
    pigment {color Red}
    finish{F_MetalE}
};
```

```
// Aufrufe
object { Kugel }
object { Kugel
    translate <-1.5, 0, 0>
}
object { Kugel
    translate <-1.5, 0, 1.5>
}
object { Kugel
    translate <1.5, 0, 0>
}
...
```





Kontrollstrukturen

Kontrollstrukturen werden verwendet, um den Ablauf eines Computerprogramms zu steuern. Eine Kontrollstruktur gehört entweder zur Gruppe der *Schleifen* oder der *Verzweigungen*.

Eine Schleifen-Art ist die **while-Schleife**:

```
#while (Bedingung)
    Rumpf
#end
```

Sie dient dazu, eine Abfolge von **Anweisungen mehrfach auszuführen**, solange eine Bedingung erfüllt ist. Diese Bedingung wird geprüft, **bevor** die Anweisungsfolge abgearbeitet wird. Es kann also auch sein, dass die Abfolge gar nicht ausgeführt wird.

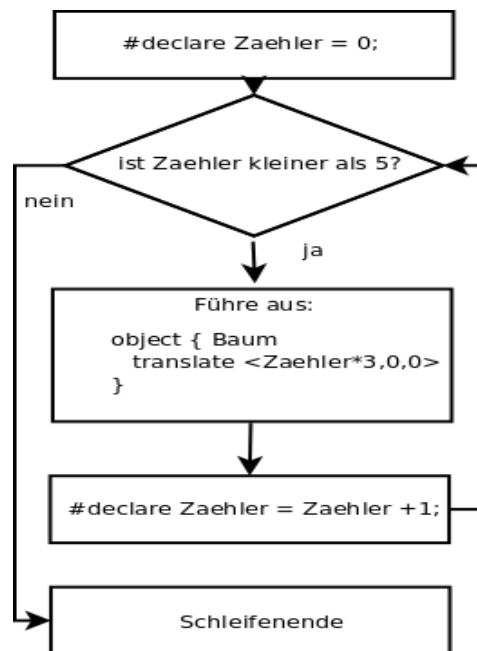
Beispiel:

```
// Laufvariable initialisieren
#declare Zaehler = 0;

// while-Block
#while ( Zaehler < 5 )
    object { Baum
        translate <Zaehler*3, 0, 0>
    }
}

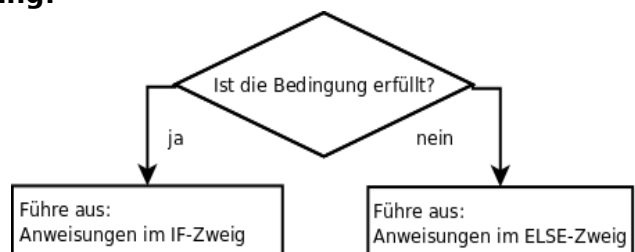
// Laufvariable erhöhen!!!
#declare Zaehler = Zaehler + 1;

#end
```



Eine Verzweigungsart ist die **Fallunterscheidung**:

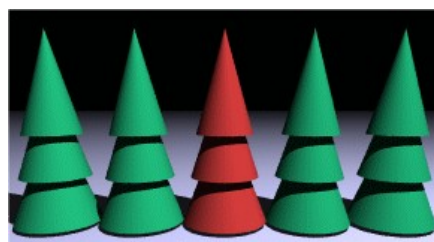
```
#if (Bedingung)
    IF-Zweig
#else
    ELSE-Zweig
#end
```



Sie dient dazu mehrere **Alternativen** zu ermöglichen. Aufgrund einer Bedingung wird der Programmfluss (die Abfolge der Ausführung der Befehle) verzweigt. Im Gegensatz zu Schleifen, die den Programmablauf nach oben zurückführen, geht der Ablauf bei einer Verzweigung immer über einen von mehreren Wegen weiter nach unten.

Beispiel:

```
#if (Zaehler != 0)
    pigment { rgb< 0.1, 0.6, 0.4 > }
#else
    pigment { rgb< 0.7, 0.2, 0.2 > }
#end
```



Exkurs: Zufallszahlen

POV-Ray besitzt eine Funktion zur Erzeugung verschiedener Folgen von Zufallszahlen. Bei Verwendung dieser Zufallszahlenfunktion in While-Schleife lassen sich in POV-Ray sehr realistische zufällige Verteilungen simulieren.

Beispiel:

Um einen möglichst natürlichen Wald zu erstellen, sollten die Bäume unregelmäßig verteilt werden. Das kann man erreichen, indem man Zufallszahlen generieren lässt und diese dann benutzt, um die Positionen für die Bäume festzulegen.

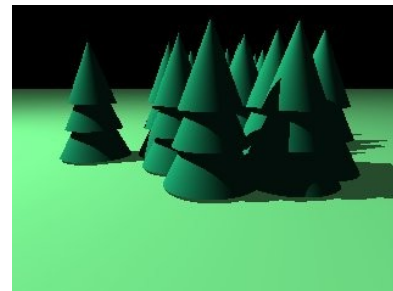
```
// Initialisierung der Zufallszahlenfolge
#declare Wald_Folge = seed (123);

// Laufvariable initialisieren
#declare Zaehler=0;

//while-Block
#while ( Zaehler < 15 )
  object { Baum
    translate < rand (Wald_Folge)*6, 0, 0 >
    rotate < 0, rand (Wald_Folge)*360, 0 >
  }

  // Laufvariable erhöhen
  #declare Zaehler=Zaehler+1;
#end // von while
```

Die Variable Wald_Folge sorgt dafür, dass die 15 Bäume zufällig um den Nullpunkt verteilt werden.



Mittels **#declare Wald_Folge = seed (123)**; definiert man die zu verwendende Zufallszahlenfolge. Der Aufruf dieser Werte erfolgt durch **rand (Wald_Folge)**. Jeder Aufruf dieser Art erzeugt eine andere Zufallszahl mit einem Wert zwischen 0 und 1.

Hinweis:

Der Computer macht nichts wirklich zufällig. Durch die Verwendung der Variable seed(Wert) werden mit rand(Variable) immer die selben „Zufallszahlen“ erzeugt, so dass das berechnete Bild bei jedem Aufruf gleich aussieht.

Links

POV-Ray ist ein frei erhältliches Programm, das auf der POV-Ray-Homepage herunter geladen werden kann:

<http://www.povray.org>

Auf der gleichen Seite findest Du auch eine **Dokumentation** zur POV-Ray-Beschreibungssprache (auf Englisch).

Eine **sehr empfehlenswerte Webseite**, von der auch viele unserer Erklärungen, Beispiele und Bilder stammen ist unter folgender Adresse zu finden:

http://www.f-lohmueller.de/pov_tut/pov_ger.htm

Anleitungen in **Deutsch** und jede Menge Tipps findest Du außerdem unter:

<http://www.mbork.de/frame.htm>