Grundlagen der 3D-Modellierung

POV-Ray

- Szenenbeschreibung

Include-Anweisungen:

```
#include "colors.inc"
#include "textures.inc"
#include "shapes.inc"
```

Körper und Grundelemente:

Kamera:

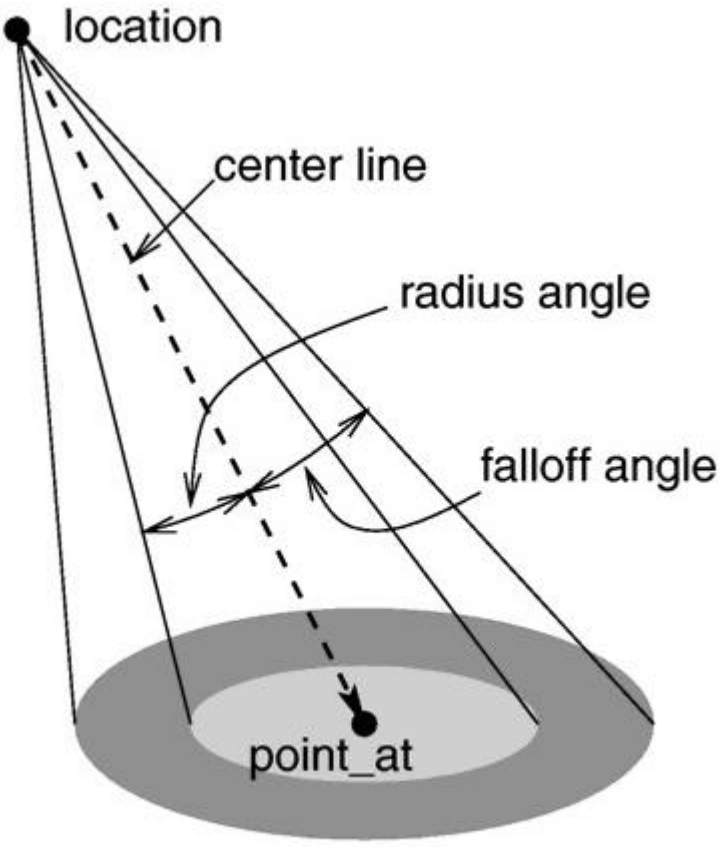
```
camera {
    location <0, 2, -3>
    look_at  <0, 1,  2>
}
```

```
camera {
    [ perspective | orthographic | fisheye |
      ultra_wide_angle | omnimax | panoramic |
      cylinder FLOAT ]
    location <VECTOR>
    look_at <VECTOR>
    right <VECTOR>
    up <VECTOR>
    direction <VECTOR>
    sky <VECTOR>
    right <VECTOR>
    angle FLOAT
    blur_samples FLOAT
    aperture FLOAT
    focal_point <VECTOR>
    normal { NORMAL }
}
```

Licht:

```
light_source { <2, 4, -3> color White}
```

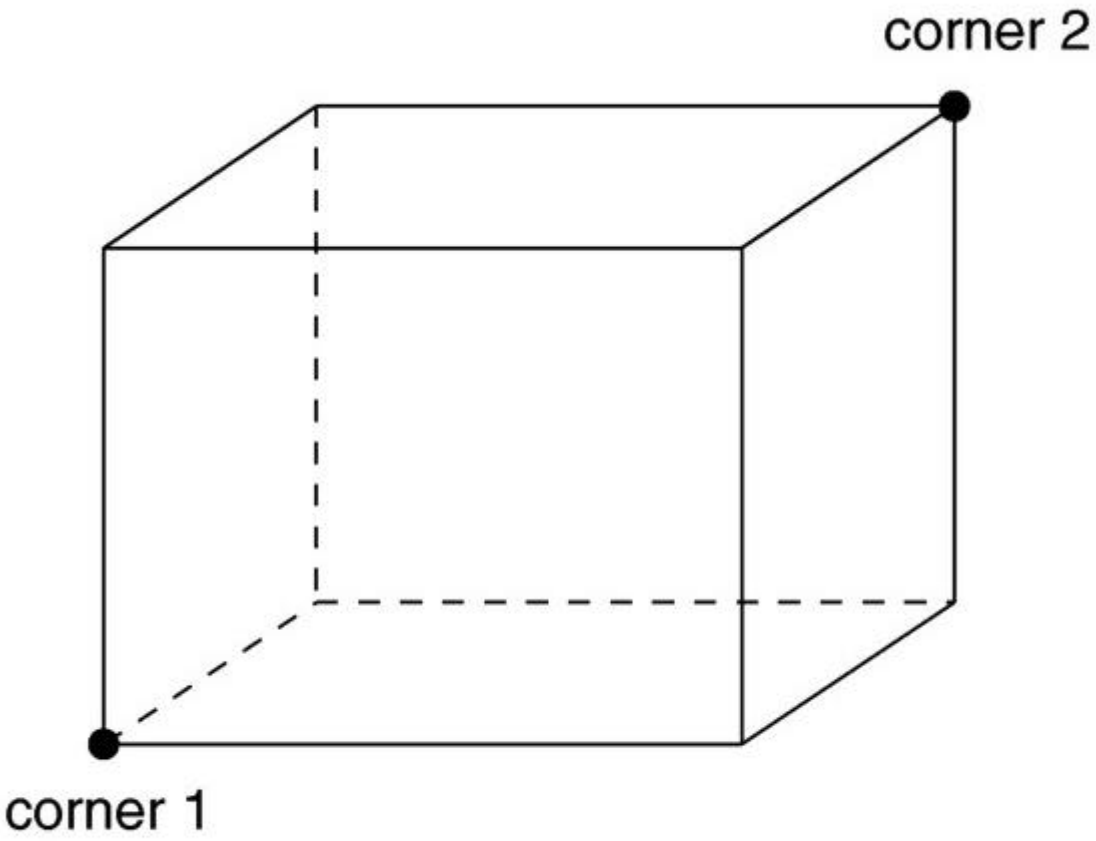
```
light_source {
    <LOCATION>
    color <COLOUR>
    [ spotlight ]
    [ point_at <POINT_AT> ]
    [ radius RADIUS ]
    [ falloff FALLOFF ]
    [ tightness TIGHTNESS ]
    [ area_light <AXIS1>, <AXIS2>, SIZE1, SIZE2 ]
    [ adaptive ADAPTIVE ]
    [ jitter JITTER ]
    [ looks_like { OBJECT } ]
    [ fade_distance FADE_DISTANCE ]
    [ fade_power FADE_POWER ]
    [ atmospheric_attenuation BOOL ]
}
```



The geometry of a spotlight.

Quader:

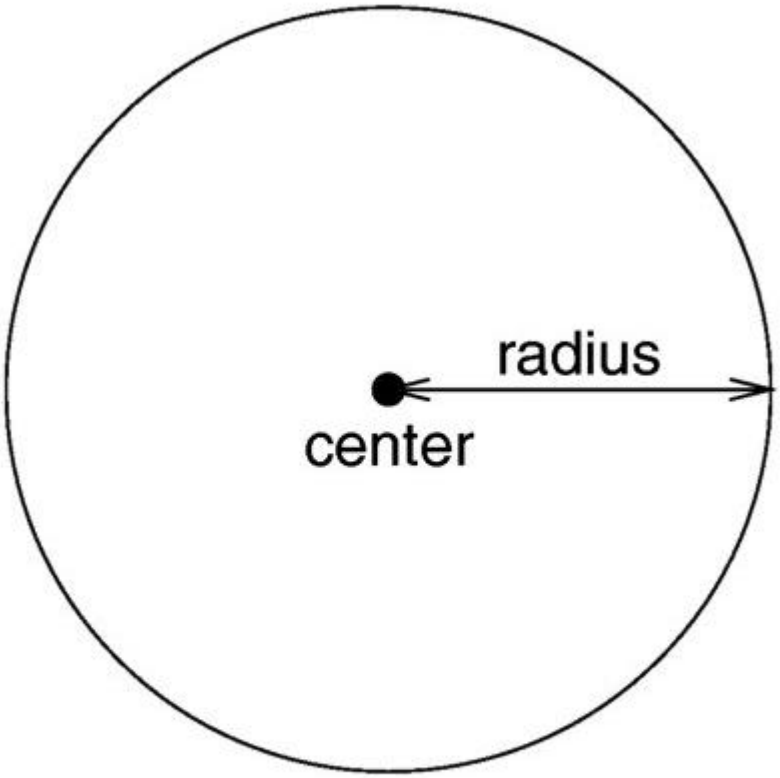
```
box { <CORNER1>, <CORNER2> }
```



The geometry of a box.

Kugel:

```
sphere {
    <CENTER>, RADIUS
}
```



The geometry of a sphere.

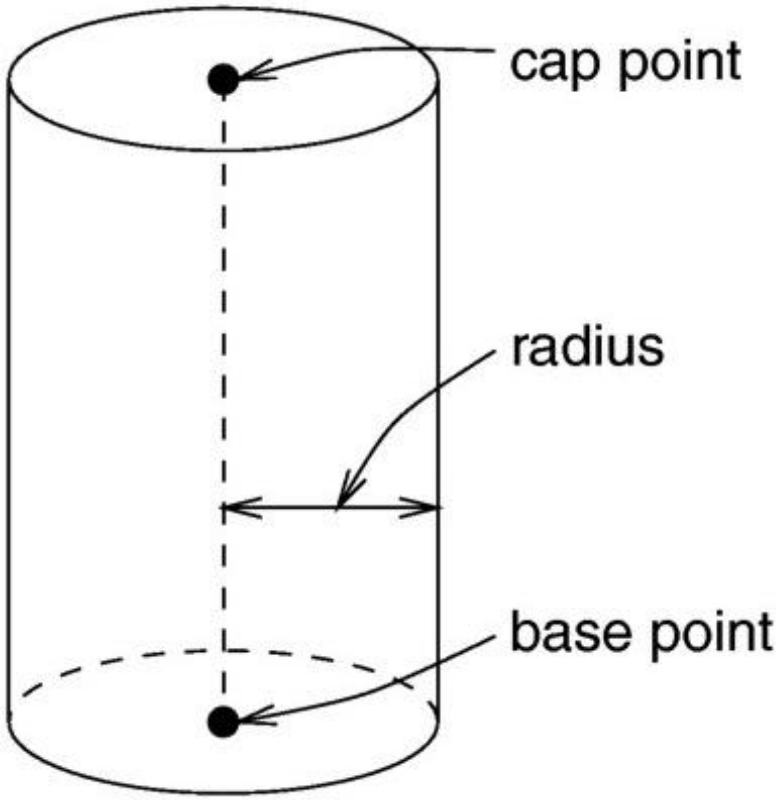
Ebene:

```
plane { <NORMAL>, DISTANCE }
```

Normalen-Vektor bestimmt Ober- bzw. Unterseite für CSG !!!

Zylinder (gerade, kreisförmig):

```
cylinder {
    <BASE_POINT>, <CAP_POINT>, RADIUS
    [ open ]
}
```



The geometry of a cylinder.

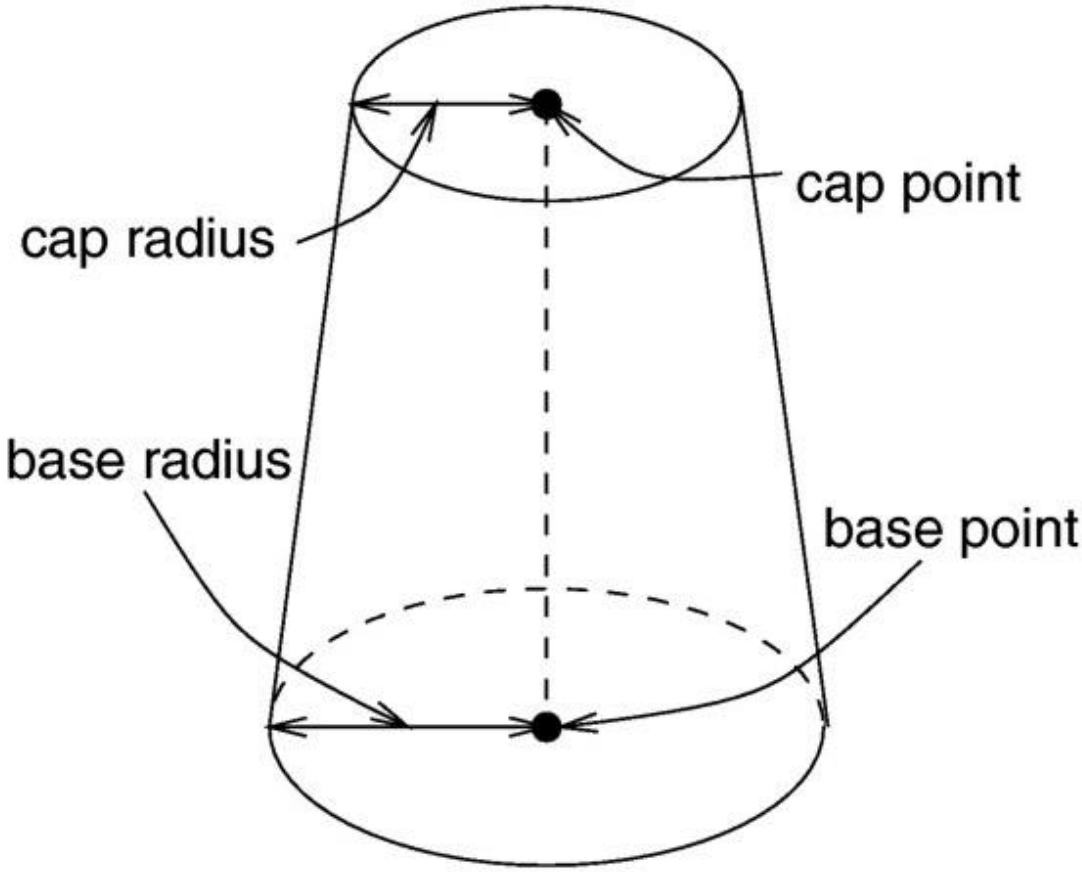
Disc - Scheibe:

```
disc {
    <CENTER>, <NORMAL>, RADIUS [, HOLE_RADIUS ]
}
```

Normalen-Vektor !!!

Kegel, Kegelstumpf:

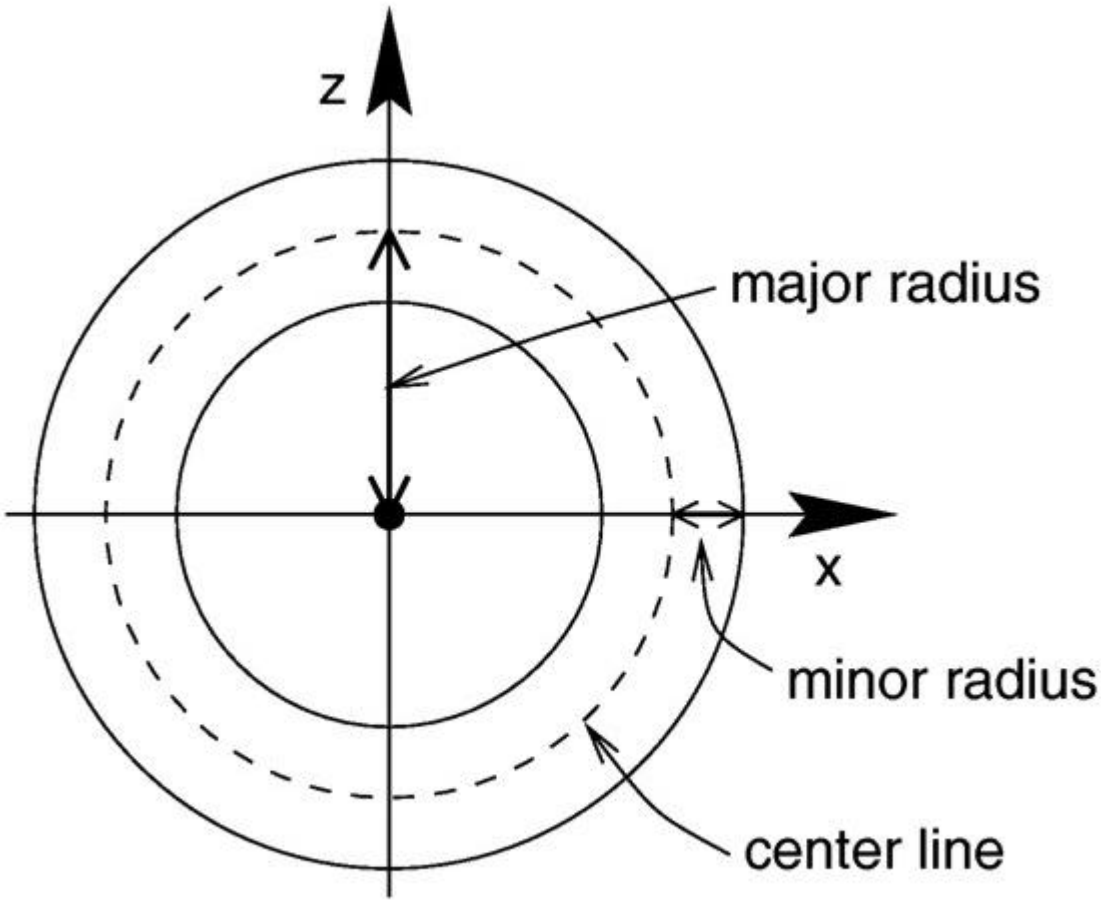
```
cone {
    <BASE_POINT>, BASE_RADIUS, <CAP_POINT>, CAP_RADIUS
    [ open ]
}
```



The geometry of a cone.

Torus:

```
torus {
    MAJOR, MINOR
    [ sturm ]
}
```



Major and minor radius of a torus.

Hollow, No_Shadow:

```
union {
    sphere { -0.5*x, 1 }
    sphere {  0.5*x, 1 }
    hollow
}

object {
    My_Thing
    no_shadow
}
```

Farbliche Gestaltung (Texturen):

```
texture {
    TEXTURE_IDENTIFIER     //include
    pigment {...}
    normal {...}
    finish {...}
    halo {...}
    TRANSFORMATIONS
}
```

pigment:

einfache Farbe:

```
pigment {color Orange}
```

Pattern:

```
pigment { brick COLOR1, COLOR2 MODIFIERS ... }
pigment { checker COLOR1, COLOR2 MODIFIERS ... }
pigment { hexagon COLOR1, COLOR2, COLOR3 MODIFIERS ... }
```

Maps:

```
pigment{
    PATTERN_TYPE
    color_map {
      [ NUM_1 COLOR_1]
      [ NUM_2 COLOR_2]
      [ NUM_3 COLOR_3]
       ...
    }
    PIGMENT_MODIFIERS...
}
```

oder

```
pigment{
    PATTERN_TYPE
    pigment_map {
      [ NUM_1 PIGMENT_BODY_1]
      [ NUM_2 PIGMENT_BODY_2]
      [ NUM_3 PIGMENT_BODY_3]
       ...
    }
    PIGMENT_MODIFIERS...
}
```

Kombinationen sind ebenfalls möglich, z.B.:

```
pigment {
    checker
    pigment { Jade scale .8 }
    pigment { White_Marble scale .5 }
}
```

Image Maps:

Mit Hilfe von Image Maps ist es möglich, zweidimensionale Vorlagen der Grafikformate gif, tga, iff, ppm, pgm, png und sys auf einen Körper zu projizieren.

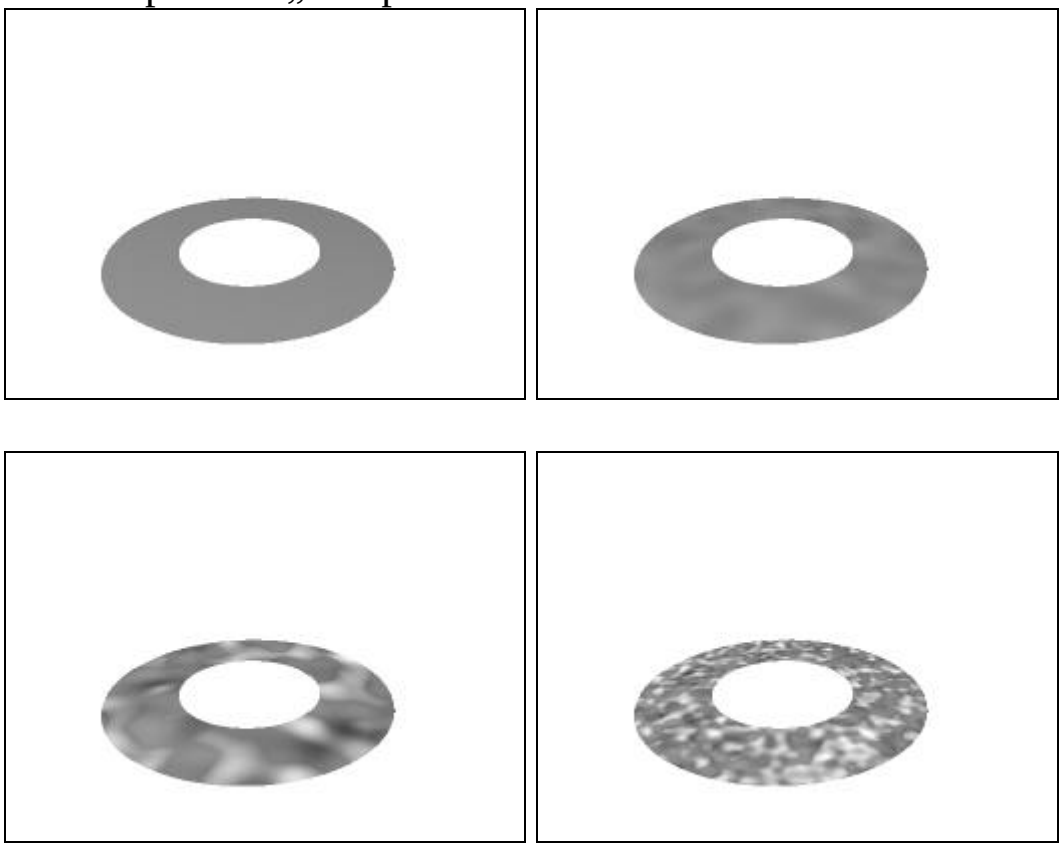
```
pigment {
    image_map {
      FILE_TYPE "filename"
      MAP_TYPE VALUE
      MODIFIERS...
    }
}
```

map_type 0	Ebene
map_type 1	Kugel
map_type 2	Zylinder
map_type 3	//noch nicht belegt
map_type 4	//noch nicht belegt
map_type 5	Torus

normal:

```
normal {
    NORMAL_IDENTIFIER
    PATTERN_TYPE FloatValue
    NORMAL_MODIFIERS
    TRANSFORMATIONS...
}
```

Beispiel mit „bumps“:



Patterntypen können z.B. `bumps, dents` oder `marble` sein.

finish:

```
finish {
    FINISH_IDENTIFIER
    [ ambient COLOR ]
    [ diffuse FLOAT ]
    [ brilliance FLOAT ]
    [ phong FLOAT ]
    [ phong_size FLOAT ]
    [ specular FLOAT ]
    [ roughness FLOAT ]
    [ metallic [ FLOAT ] ]
    [ reflection COLOR ]
    [ refraction FLOAT ]
    [ ior FLOAT ]
    [ caustics FLOAT ]
    [ fade_distance FLOAT ]
    [ fade_power FLOAT ]
    [ irid { thickness FLOAT turbulence VECTOR } ]
    [ crand FLOAT ]
}
```

Texturkombinationen:

```
object {
    My_Object
    texture {T1}  // unterste Schicht
    texture {T2}  // transparente Schicht
    texture {T3}  // oberste Schicht (transparent)
}
```

halo:

Typen:
attenuating ... schwächt das Licht
emitting ... jeder Partikel wirkt als winzig kleine Lichtquelle;
keine gegenseitige Beeinflußung
glowing ... wie emitting, mit gegenseitiger Beeinflußung
dust ... nur Reflexion und Brechung des Lichts

```
halo {
    attenuating | emitting | glowing | dust
    [ constant | linear | cubic | poly ]
    [ planar_mapping | spherical_mapping |
      cylindrical_mapping | box_mapping ]
    [ dust_type DUST_TYPE ]
    [ eccentricity ECCENTRICITY ]
    [ max_value MAX_VALUE ]
    [ exponent EXPONENT ]
    [ samples SAMPLES ]
    [ aa_level AA_LEVEL ]
    [ aa_threshold AA_THRESHOLD ]
    [ jitter JITTER ]
    [ turbulence <TURBULENCE> ]
    [ octaves OCTAVES ]
    [ omega OMEGA ]
    [ lambda LAMBDA ]
    [ colour_map COLOUR_MAP ]
    [ frequency FREQUENCY ]
    [ phase PHASE ]
    [ scale <VECTOR> ]
    [ rotate <VECTOR> ]
    [ translate <VECTOR> ]
}
```

Anwendungsgebiete für Halos ist die Modellierung von Wolken, Nebel, Feuer oder ähnlichen Erscheinungen, welche schlecht mit den gegebenen Grundkörpern zu realisieren sind.

Beispiele: halo01.tga, halo02.tga

Anwendungen auf Objekte und Texturen:

Drehung: `rotate <VECTOR>`
... um Ursprung in Grad
Skalierung: `scale <VECTOR>`
Verschiebung: `translate <VECTOR>`

... relativ zur Position

Matrix:

```
matrix < m00, m01, m02,
         m10, m11, m12,
         m20, m21, m22,
         m30, m31, m32 >
```

Bsp:

```
object {
    MyObject
    matrix < 1, 1, 0,
             0, 1, 0,
             0, 0, 1,
             0, 0, 0 >
}
```

Transformationen sollten in ihrer Reihenfolge gut überlegt werden, da zum Beispiel die folgende Kombination

```
translate <5, 6, 7>
scale 4
```

dazu führt, daß die Translation skaliert wird.
Abhilfe kann hier eine Veränderung der Reihenfolge bzw. korrekte Klammerung der Operationen schaffen.

CSG

Constructive Solid Geometry

Beispiel: csg01.tga

Komplexere Körper:

Dreiecke:

```
triangle {
    <CORNER1>, <CORNER2>, <CORNER3>
}
```

Polygone:

```
polygon {
```

```
    TOTAL_NUMBER_OF_POINTS,
    <A_1>, <A_2>, ..., <A_na>, <A_1>, //Polygon A
    <B_1>, <B_2>, ..., <B_nb>, <B_1>, //Sub-Polygon B
    <C_1>, <C_2>, ..., <C_nc>, <C_1>, //Sub-Polygon C
    ...
}
```

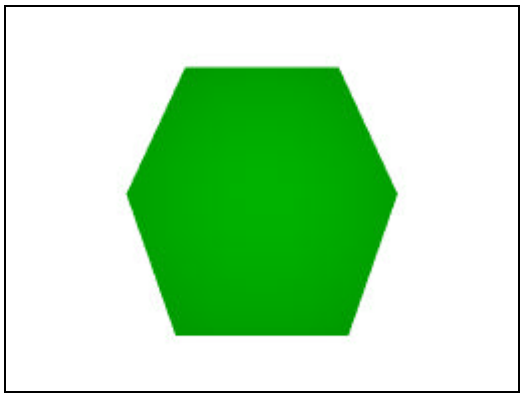
Prismen:

```
prism {
    [ linear_sweep | conic_sweep ]
    [ linear_spline | quadratic_spline | cubic_spline ]
    HEIGHT1,
    HEIGHT2,
    TOTAL_NUMBER_OF_POINTS,
    <POINT_1>, <POINT_2>, ..., <POINT_n>
    [ open ]
    [ sturm ]
}
```

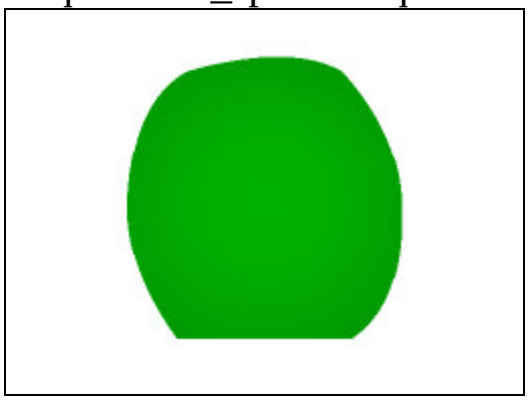
Prismen müssen geschlossen sein (Point_1 = Point_n)

linear_sweep ... gerades Prisma
conic_sweep ... Pyramide mit Spitze im Mittelpunkt und HEIGHT 2

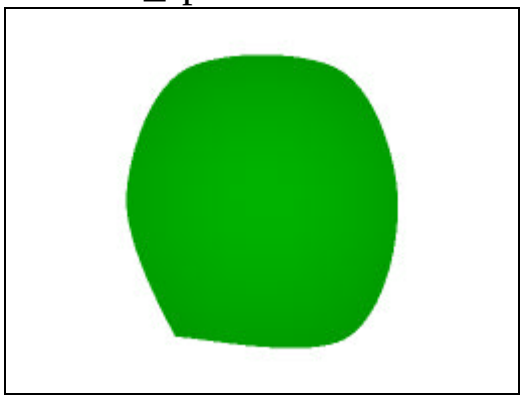
linear_spline ... gerade Verbindung der Eckpunkte



quadratic_spline ... quadratische Annäherung



cubic_spline ... kubische Annäherung



Lathe (dt.: Drehbank):

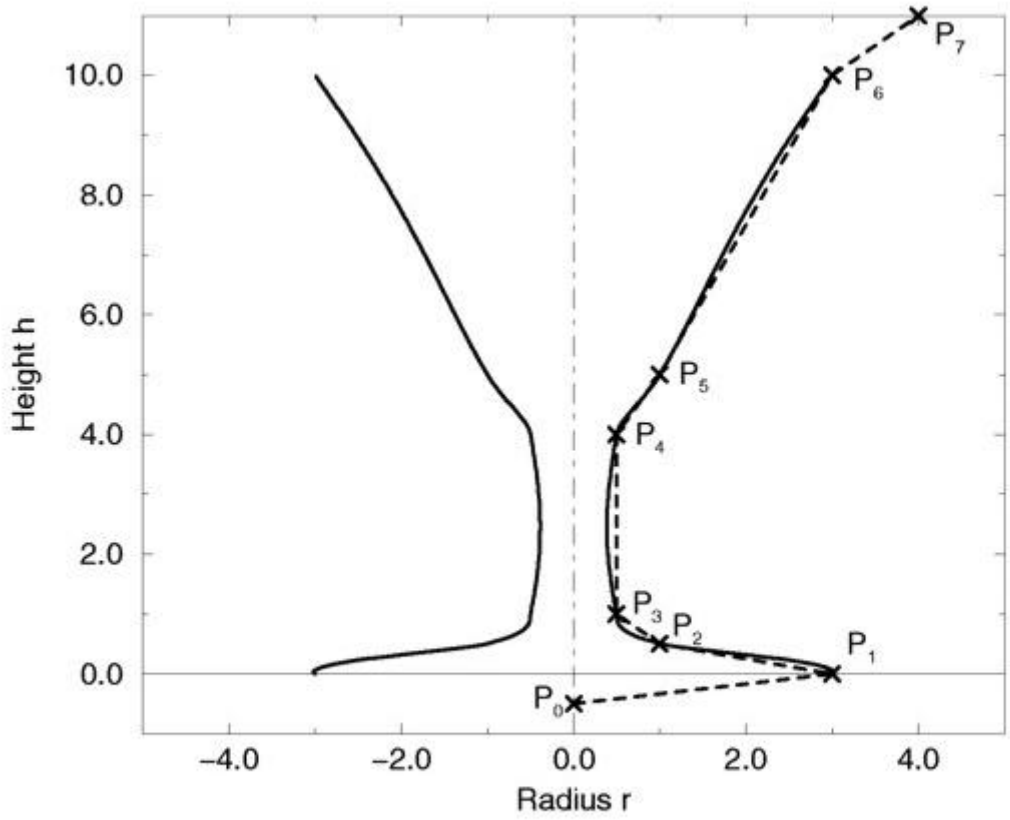
Rotationskörper um die y-Achse, bestehend aus n Kontrollpunkten.

```
lathe {
    [ linear_spline | quadratic_spline | cubic_spline ]
     TOTAL_NUMBER_OF_POINTS,
    <POINT_1>, <POINT_2>, ..., <POINT_n>
}
```

Surface of Revolution:

Ähnliche Funktion wie Lathe, Körper wird aber nicht gefüllt. Außerdem sind keine extra Kontrollpunkte nötig (wie bei quadratic_spline und cubic_spline).

```
sor {
    NUMBER_OF_POINTS,
    <POINT0>, <POINT1>, ..., <POINTn-1>
    [ open ]
    [ sturm ]
}
```



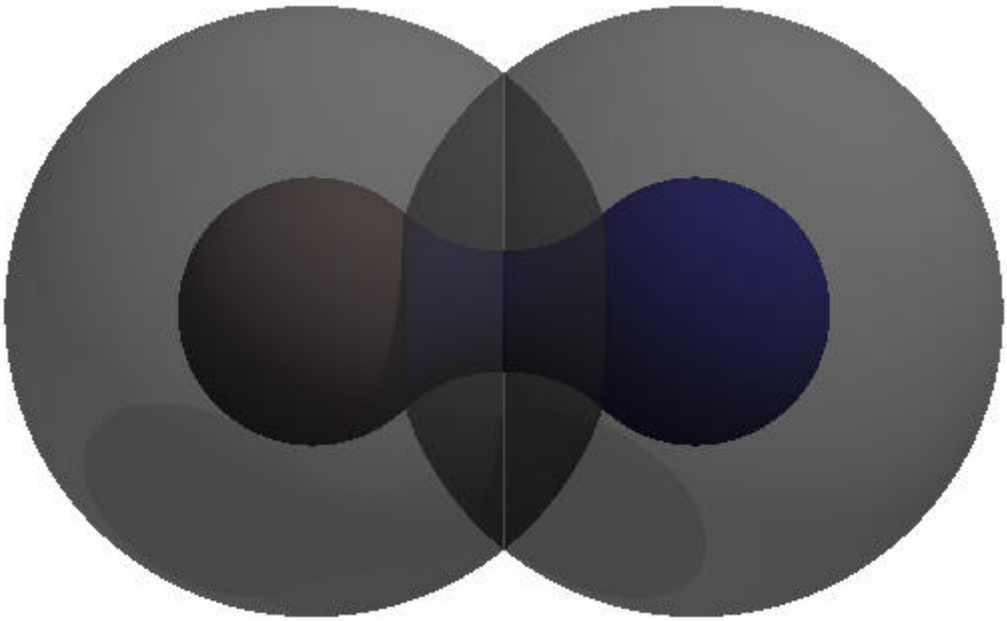
The point configuration of our cup object.

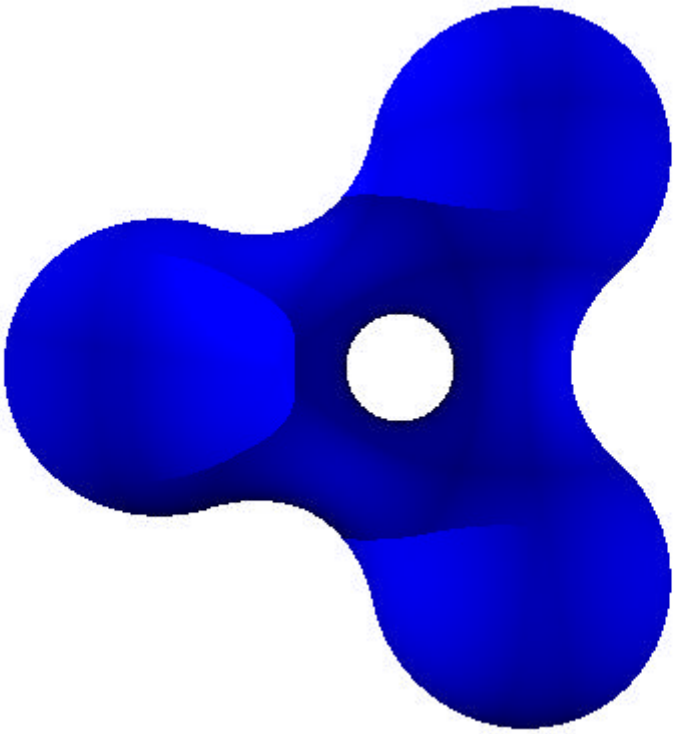
Beispiel: SOR_Cup.tga

Blob:

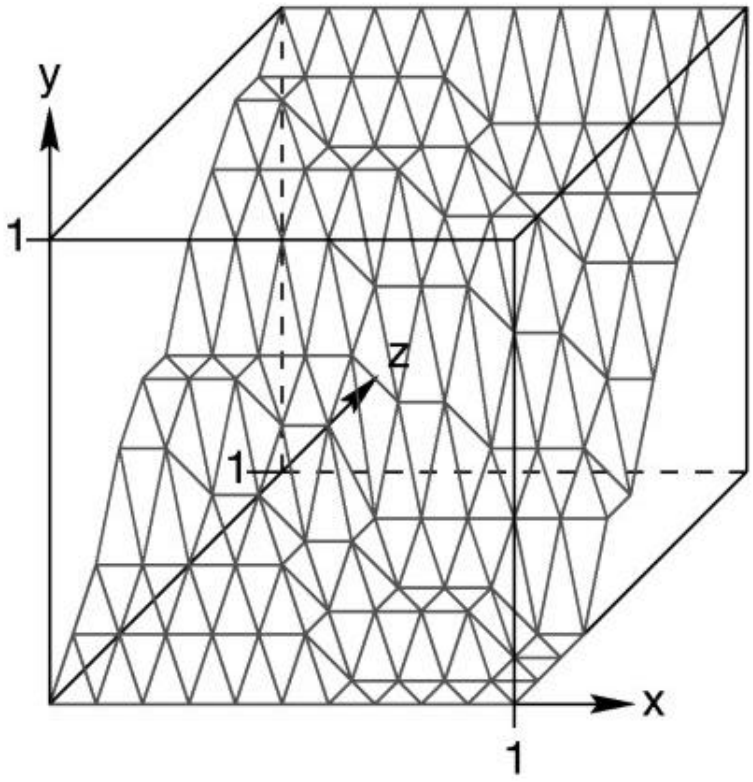
```
blob {
```

```
  threshold THRESHOLD_VALUE
  cylinder {<END1>,<END2>,RADIUS, [ strength ] STRENGTH }
  sphere { <CENTER>, RADIUS, [ strength ] STRENGTH }
  [ component STRENGTH, RADIUS, <CENTER> ]
  [ hierarchy FLAG ]
  [ sturm ]
}
```

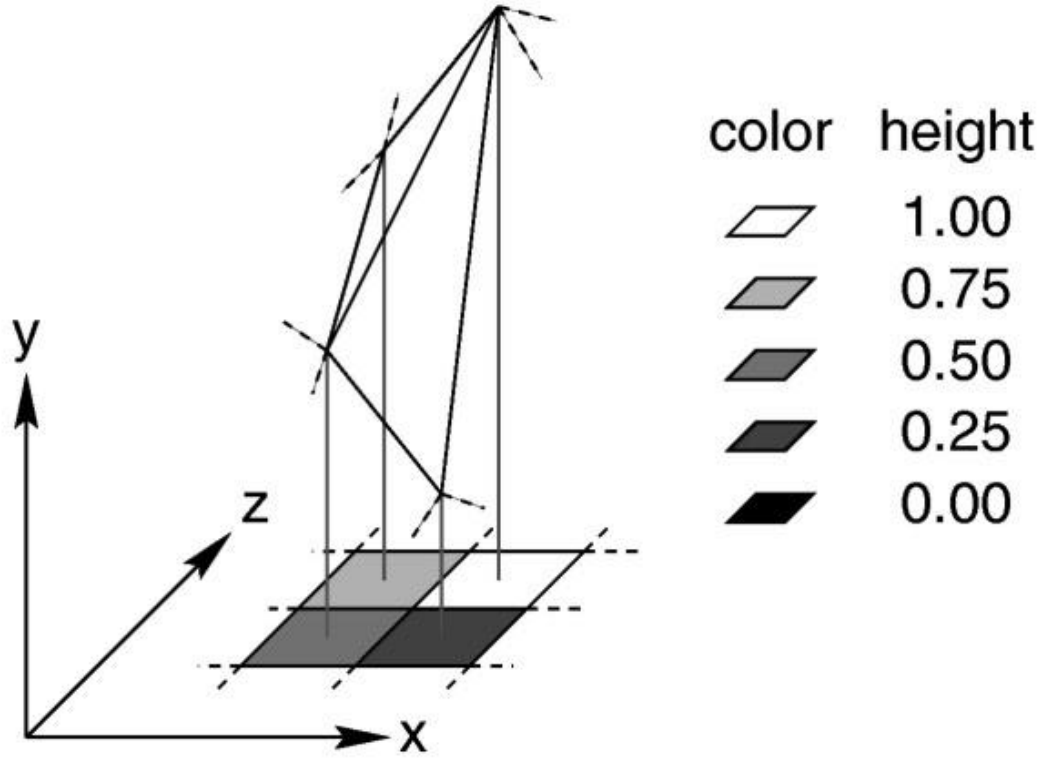




Height_Field:



The size and orientation of an un-scaled height field.



Four pixels of an image and the resulting heights and triangles in the height field.

Bicubic_Patch:

Generiert einen Körper aus der angegebenen Matrix von Kontrollpunkten.

```
bicubic_patch {
    type PATCH_TYPE
    flatness FLATNESS_VALUE
    u_steps NUM_U_STEPS
    v_steps NUM_V_STEPS
    <CP1>,  <CP2>,   <CP3>,   <CP4>,
    <CP5>,  <CP6>,   <CP7>,   <CP8>,
    <CP9>,  <CP10>,  <CP11>,  <CP12>,
    <CP13>, <CP14>,  <CP15>,  <CP16>
}
```

Typ: 0 oder 1

0 ... wenig Speicher benötigt; Zeitaufwendig

1 ... mehr Speicher, dafür schneller

Steps: `sub-pieces = 2^u_steps * 2^v_steps`

gute Ergebnisse schon ab 3

flatness: 0..1

hat Einfluß auf die Genauigkeit

je kleiner desto langsamer

Text:

```
text {
    ttf "FONTNAME.TTF",
    "STRING_OF_TEXT",
    THICKNESS_FLOAT, OFFSET_VECTOR
}
```

thickness_float ... Tiefe in x-Richtung

offset_vector ... bestimmt zusätzlichen Abstand zwischen den einzelnen Zeichen

– Rendering - Ausgabeoptionen:

Auflösung:
+Hn ...Höhe in Pixel
+Wn ... Breite in Pixel

Partielles Rendering:
Es ist möglich nur einen bestimmten Teilbereich des Bildes berechnen zu lassen.
+SRn ... Startzeile
+ERn ... letzte Zeile
+SCn ... Startspalte
+ECn ... letzte Spalte
Parameter n in Pixel bzw. 0.n für Prozentangaben.

Unterbrechung:
+X ... erlaubt den Abbruch per Tastendruck
-X ... off

Wiederaufnahme:
Gestattet die Weiterberechnung eines vorher begonnenen, aber abgebrochenen Rechenvorgangs.
+C ... Continue on
-C ... Continue off

Display:
+D ... mit Grafikausgabe
-D ... ohne Grafikausgabe

+Dxy
x ... Displaytyp (0 = autodetect)
y ... Palette (G = gray scale; T = TrueColor)

Pause:
+P ... Pause nach fertigem Rendering

-P ... keine Pause; zurück zum OS

Dateiausgabe:
-F ... ohne Speicherung auf Festplatte
+Fx ... mit Speicherung auf Festplatte (default)
x ... Filetyp (N = PNG; T = TGA)

+Ofile ... Dateiname der Ausgabedatei
+Ifile ... Dateiname der Eingabedatei (*.pov)

Qualität:
+Qn ... mit 0 <= n <= 11 (default: n = 9)

Anti-Aliasing (Kantenglättung):
-A ... AntiAliasing off (default: AA off)
+An.n ... AA on;
n.n zwischen 0.0 und 3.0; je kleiner, desto genauer
sehr Zeitaufwendig
gute Ergebnisse bei 0.2 bis 0.4

Animation (Clock):
+KFIn ... Nummer des ersten Bildes
+KFFn ... Nummer des letzten Bildes
+KIn.n ... Ausgangswert (Clock)
+KFn.n ... Endwert (Clock)

Partielles Animationsrendering:
+SFn oder +SF0.n ... Beginne mit Frame n (bzw. 0.n Prozent)
+EFn oder + EF0.n ... Ende mit Frame n (bzw. O.n Prozent)

Loop:
+KC ... Loop on (360° Berechnungen)
-KC ... Loop off