Dreidimensionale Geometrie mit Povray

Hubert Grassmann

mit Beiträgen von Andreas Filler, André Henning und Hanne Müller

19. November 2012

Hier soll eine kurze Einführung in das Programm Povray gegeben werden, was man nutzen kann, um dreidimensionale Objekte darzustellen. Man findet Povray im Internet unter

`http://www.povray.org/redirect/www.povray.org/ftp/pub/povray/Official/Linux/povlinux-3.6.tgz`

Nach der Installation befinden sich die Include-Dateien standardgemäß im Verzeichnis

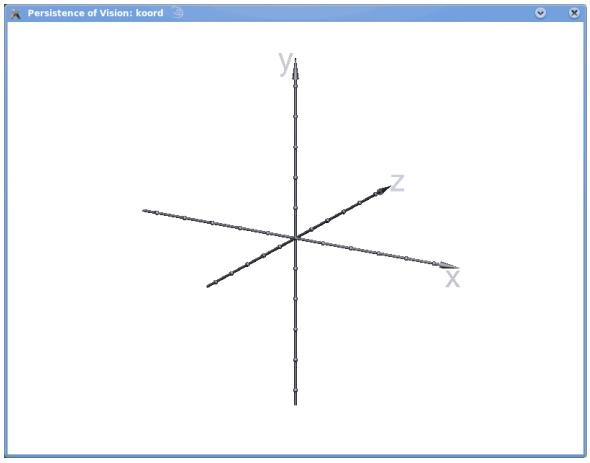
`/usr/share/povray-3.6/include`

Es werden zahlreiche Hilsmittel zur Gestaltung von Oberflächen und Hintergründen bereitgestellt; dies interessiert uns weniger: wir wollen Povray nutzen, um mathematische Objekte darzustellen.

Inhaltsverzeichnis

1 Grundlagen

Wir befinden uns im dreidimensionalen Raum und stellen uns ein linkshändiges Koordinatensystem vor:



Der Punkt mit den Koordinaten x, y, z wird mit `<x,y,z>` bezeichnet, hier können wir ein Objekt anbringen. Den Punkt `<a,a,a>` können wir kurz mit `a` benennen; `x` steht für `<1,0,0>`, `y` für `<0,1,0>`, `z` für `<0,0,1>`.

Wir betrachten das Objekt mit Hilfe einer Kamera:

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

Wir bringen eine Lichtquelle an und geben dem Licht eine Farbe:

```
light_source { <-5,30,-10> White }
```

Die Definition von Farben erfolgt im Paket `colors`, das wir vorher importieren müssen:

```
#include "colors.inc"
```

Wir geben auch dem Hintergrund eine Farbe:

```
background {White}
```

Nun können wir das erste Objekt anbringen:

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

Es soll eine Kugel um den Koordinatenursprung darstellen, der Radius ist 2, die Farbe rot. Wir speichern diesen Text in der Datei `kugel.pov`, starten Povray,

```
povray kugel.pov +W600 +H600 +P
```

und sehen nichts (nur etwas viel Rot). Das liegt daran, daß sich die Kamera fast im inneren der (Voll-)Kugel befindet; wenn wir den Radius verkleinern oder die Kameraposition geeignet ändern, sehen wir, was wir wollten. Mit den Optionen `+W600` stellen wir die Breite des Fensters in Pixeln ein, `+H600` legt die Höhe fest, und die Option `+P` bewirkt, daß das Bild nach seinem Aufbau nicht gleich wieder verschwindet, sondern bis zu einem Mausklick auf das Fenster erhalten bleibt.

Während des Aufbaus des Bildes („die Szene wird gerendert“) gibt das Programm eine Vielzahl von Informationen aus, die man nicht lesen muß, außer es erscheint so etwas:

```
File: beispiel.pov  Line: 5
Parse Warning: Pigment type unspecified or not 1st item.
File: beispiel.pov  Line: 5
File Context (5 lines):
#include "colors.inc"
camera { location <0,10,-2> look_at 0 }
light_source { <-5,30,-10> White }
background {White}
sphere{<0,0,0>, 2 pigment {colo
Parse Error: No matching } in 'pigment',
undeclared identifier 'colo' found instead
```

In Zeile 5 ist ein Syntaxfehler gefunden worden, diese und ein paar vorangehende Zeilen werden aufgelistet.

2 Werkzeuge, Objekte, Anweisungen

2.1 Farben

Der Oberfläche eines Objekts weist man eine Textur zu, zum Beispiel eine Farbe. Hierzu dient das rgb-Modell: Aus den Farben Rot, Grün und Blau wird eine Farbe gemischt, der jeweilige Farbanteil kann Werte zwischen 0 und 1 annehmen, das rgb-Tripel wird wie ein Punkt beschrieben:

```
pigment { color rgb <x,y,z> }
```

In `colors.inc` sind (neben sehr vielen anderen) die folgenden Grundfarben definiert:

```
#declare Red     = rgb <1, 0, 0>;
#declare Green   = rgb <0, 1, 0>;
#declare Blue    = rgb <0, 0, 1>;
#declare Yellow  = rgb <1,1,0>;
#declare Cyan    = rgb <0, 1, 1>;
#declare Magenta = rgb <1, 0, 1>;
#declare Clear   = rgbf 1;
#declare White   = rgb 1;
#declare Black   = rgb 0;
```

Die Oberflächen können transparent gemacht werden:

```
color rgbt <x,y,z,u>
```

Der Wert von `u` liegt ebenfalls zwischen 0 (undurchsichtig) und 1 (voll durchsichtig).

2.2 Grundkörper

Eine (unendlich große) Ebene beschreibt man durch ihren Normalenvektor und ihren Abstand von Koordinatenursprung:

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

Eine Kugel beschreibt man durch ihren Mittelpunkt, Radius und Textur.

```
sphere { <0,1,0>, 1 pigment {color Blue}}
```

Ein Quader ist durch die Angabe der Koordinaten zweier gegenüberliegender Eckpunkte festgelegt:

```
box { <0,1,0>, <2,0,3> pigment {color Blue}}
```

Er liegt parallel zu den Koordinatenachsen, kann aber (später) noch gedreht werden. Um Zylinder und Kegel(stümpfe) zu erzeugen, gibt man die Mittelpunkte des oberen und des unteren Deckkreises und deren Radien an:

```
cone { <0,1,0>, 2 <1,2,0> 1 pigment {color Blue}}
```

Wenn beide Radien gleich sind, entsteht ein Zylinder, wenn ein Radius gleich Null ist, entsteht ein Kegel. Einen Zylinder kann man auch mittels

```
cylinder { <0,1,0>, <1,2,0>, 1 pigment {color Blue}}
```

herstellen.

Ein Torus wird durch zwei Kreise festgelegt: den Radius des Kreises, der die Mittelpunkte der Querschnittskreise verbindet und den Radius eines Querschnittskreises; er liegt in der x-z-Ebene, sein Mittelpunkt ist der Koordinatenursprung, er kann (später) bewegt werden.

```
torus { 5, 2 pigment {color Blue}}
```

Ein auf x-z-Ebene stehendes Prisma wird durch die Höhe seiner Grundfläche, die Höhe seiner Deckfläche, die Anzahl der Eckpunkte sowie deren x-z-Koordinaten festgelegt:

```
prism{linear_sweep 1,3,4 <0,1>,<-1,0>,<0,-1>,<1,0>,<0,1> pigment{Red}}
```

Mein Wörterbuch konnte mir keine rechte Erklärung des Worts „sweep“ liefern; das angegebene Schlüsselwort legt fest, daß die Eckpunkte geradlinig verbunden werden. Der erste Eckpunkt muß mit dem letzten Eckpunkt übereinstimmen. Wenn `conic_sweep` angegeben wird, entsteht eine Pyramide: Die angegebenen Punkte werden in der Höhe $y = 1$ mit dem Ursprung verbunden und dann wird entsprechend der festgelegten Höhen abgeschnitten.

All diese Körper sind Vollkörper.

2.3 Boolesche Operationen

Mit den oben betrachteten Vollkörpern können Vereinigungen, Durchschnitte und Mengen-Differenzen gebildet werden:

```
union{Objekt1 Objekt2 Objekt3 Transformation Textur}
```

```
intersection{Objekt1 Objekt2 Transformation Textur}

difference{Objekt1 Objekt2 Transformation Textur}
```

Dies wurde angewandt, um eine Szene „Häuschen mit Garten“ darzustellen: [1]

```
#include "colors.inc"
camera {ultra_wide_angle angle 75 location<8,3,-2> look_at<0,1,2>}

difference
{
  difference
  {
    intersection
    {
      box { <0, 0, 0>, <2, 2, 2> pigment { color Grey} }
      sphere{ <1,1,1>, 1.5 pigment{color Black}}
    }
    box {<1,1,1>, <2.5, 1.5, 1.5>}
    pigment {color Yellow}
  }
  box {<0.4,1.5,0.4>, <8,1.8,0.7>}
  pigment {color Yellow}
}
plane { < 0, 1, 0>,1 pigment { color Green } }
cone { <1, 2, 1 >,1.1, <1, 4, 1>,0 pigment {color Red} }
difference
{
  difference
  {
    cylinder {<1,0,1>, <1,1.5,1>, 4 pigment {color rgb<0.5,0.3,0>}}
    cylinder {<1,0,1>, <1,3,1>, 3.9}
  }
  box {<3,1,1>, <7,7,7>}
}
light_source {<4, 4, -4> color White}
cylinder{ <3,0,3>, <3,3,3>, 0.1 pigment {color rgb<0.5,0,0>}}
sphere{ <3,3,3>, 0.3 pigment{color Green}}
sphere{ <2.8,2.5,2.8>, 0.3 pigment{color Green}}
sphere{ <3.2,2.5,3.2>, 0.3 pigment{color Green}}
sphere{ <3.1,2.5,2.9>, 0.3 pigment{color Green}}
sphere{ <2.9,2.5,3.1>, 0.3 pigment{color Green}}
// sky ---------------------------
plane{<0,1,0>,1 hollow texture{ pigment{ bozo turbulence 0.92
          color_map
          {
           [0.00 rgb <0.2,0.2,1>*0.9]
```

[1] Diese Szene wurde von Studenten erstellt, deren Namen ich nicht mehr finden kann.

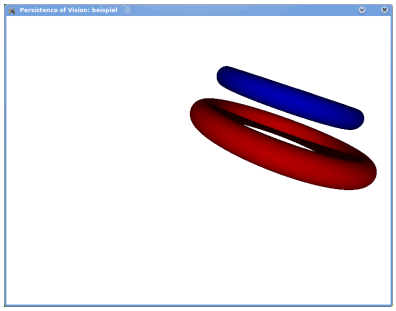
```
            [0.50 rgb <0.2,0.2,1>*0.9]
            [0.70 rgb <1,1,1>]
            [0.85 rgb <0.25,0.25,0.25>]
            [1.0 rgb <0.5,0.5,0.5>]
           }
           scale<1,1,1.5>*2.5
           translate<1.0,0,-1>
           }// end of pigment
         finish {ambient 1 diffuse 0}
         }// end of texture
         scale 10000
      }// end of plane

// fog on the ground -------------
fog { fog_type   2
      distance   50
      color      White
      fog_offset 0.1
      fog_alt    1.5
      turbulence 1.8
    }
fog{distance 300000 color White}
```

2.4 Transformationen

Wenn Objekte in einer Standardlage konstruiert wurden, so kann man sie transformieren: man gibt am Ende der Objektdefinition Anweisungen an, wie es verschoben, gedreht, gedehnt/gestaucht werden soll. Eine Verschiebung erreicht man mit der Anweisung `translate <a,b,c>`, es erfolgt eine Verschiebung um den Ortsvektor des angegebenen Punkts. Eine Drehung erfolgt bei `rotate <a,b,c>`, es wird um die Winkel

a, b, c in x-,y-,z-Richtung verschoben, die Winkel sind im Grad-Maß anzugeben. Eine Größenveränderung erfolgt bei `scale <a,b,c>`, der Maßstab wird um die Faktoren a, b, c in x-,y-,z-Richtung verändert. Die Transformationen werden in der Reihenfolge, wie sie aufgeschrieben wurden, ausgeführt; die Operationen kommutieren nicht, die folgenden beiden Zeilen bewirken Verschiedenes:

```
torus {2, 0.3 translate <2,3,0> rotate <60, 20, 0> pigment{color Blue}}
torus {2, 0.3 rotate <60, 20, 0> translate <2,3,0> pigment{color Red}}
```



Die aus der linearen Algebra bekannten linearen Abbildungen können mit der `matrix`-Anweisung angewandt werden:

```
matrix  < v1.x,v1.y,v1.z,
          v2.x,v2.y,v2.z,
          v3.x,v3.y,z3.z,
          0,0,0 >
```

Dabei sind $v1, v2, v3$ die Bilder der Vektoren `x,y,z` und `v1.x` ist die x-Komponente von $v1$. In der linearen Algebra stehen die Koordinaten der Bilder der Basisvektoren in den Spalten der Darstellungsmatrix einer linearen Abbildung; hier stehen die Koordinaten in den Zeilen, das obige 12-tupel wird als 4×3-Matrix aufgefaßt. Wenn in der letzten Zeile keine Nullen eingetragen werden, so wird die affine Abbildung ausgeführt, wo das letzte Tripel als Bild des Ursprungs interpretiert wird.

2.5 Text

Einen Text gibt man etwa wie folgt auf dem Bildschirm aus:

```
text{ttf "timrom.ttf" "Abc" 0.1, 0 pigment{Red} translate 3*x scale .3}
```

Es werden der Zeichensatz (hier: TimesRoman), die Zeichenkette, die „Dicke“ der Zeichen und ein offset-Vektor, der die Verschiebung des jeweils nachfolgenden Zeichens angibt, festgelegt. Die Zeichen stehen in der x-y-Ebene, beginnen bei `<0,0,0>` und haben eine Standardgröße, die skaliert werden kann. Die Zeichen haben auch eine Dicke, was man sieht, wenn die Kamera schrägt drauf guckt. Im Beispiel ist die Dicke auf ein Zehntel reduziert, damit die Zeichen sich bei Schrägansicht nicht überlappen. Der offset-Vektor ist auf $0 =< 0, 0, 0 >$ eingestellt, das gibt ein normales Schriftbild. Es

wäre schön, wenn die Zeichen (wie oben die Achsenbezeichnungen des Koordinatensystems) so ausgerichtet wären, daß die Kamera senkrecht drauf schaut. Dazu greifen wir etwas vor und benutzen die oben eingeführte Transformationsmatrix:

```
zeigt Text bzw. Zeichenkette senkrecht zur Kamera ausgerichtet.
#macro zeig(stelle, zeichen)
object
{
  text{ttf "timrom.ttf" zeichen 0.1,0 pigment{Red}}
  matrix
  <
  z1.x,z1.y,z1.z,
  z2.x,z2.y,z2.z,
  z3.x,z3.y,z3.z,
  stelle.x,stelle.y,stelle.z>
  translate 2.2*stelle scale 0.3
}
#end
```

Hier ist $z3$ der normierte Ortsvektor der Kamera und $z1, z2$ spannen die dazu orthogonale Ebene auf.

Wenn Zahlen (etwa von Povray berechnete) ausgegeben werden sollen, so müssen diese in Zeichenketten umgewandelt werden, dies geht so:

```
str(Zahl 0,2)
```

Die Parameter geben an, wo der Dezimalpunkt stehen soll und wie viele nachkommende Stellen gezeigt werden sollen.

Wenn ein anderer Zeichensatz verwendet werden soll, so kann man sich bei `http://www.fonts101.com` welche besorgen, später in diesem Text ist mal `standardgreekitalic.ttf` verwendet worden.

2.6 Variable

Povray hat eine Vielzahl von Schlüsselworten sowie fixierte Bezeichner; ich habe mehrfach den Fehler gemacht, eigenen Variablen solche Bezeichner wie `x,y,z,t,u,v` zu geben, diese sind reserviert, es erfolgt eine gewöhnungsbedürtige Fehlermeldung.
Variable werden mit dem `declare`-Befehl deklariert und belegt:

```
#declare bb = 2;
```

(Wenn das Semikolon fehlt, erfolgt eine leicht verständliche Fehlermitteilung.) Dies ist eine *globale* Variable, die später mit `bb = 3` verändert werden kann.

```
#local bb = 2;
```

Dies ist eine nur ein einer bestimmten Umgebung (einer Prozedur-Vereinbarung o.ä.) gültige Variable, auf diese kann von außerhalb nicht zugegriffen werden.
Die Anfangsbuchstaben der Povray-eigenen Bezeichner sind Kleinbuchstaben, bei eigenen Deklarationen ist man (meist) auf der sicheren Seite, wenn man Anfangs-Groß-Buchstaben verwendet. Variable werden uns noch später beschäftigen.

2.7 Mathematische Funktionen

Es steht die gesamte Palette zur Verfügung, z.B.
`sin(a), asin(a), sqrt(a), abs(a), pow(a,b)...` und es werden die Standardnamen verwendet. Es gibt auch `pi` und Umrechnungen zwischen Grad- und Bogenmaß. Schauen Sie in die Datei

`povray-3.7.0-linux2.6-x86_64/include/math.inc`

hinein.
Wir wollen uns mit **Vektoroperationen und -funktionen** beschäftigen.
Ein Punkt `<a,b,c>` wird auch als der entsprechende Ortsvektor interpretiert; wenn also `v,w` Vektoren sind, so werden die Ausdrücke `v+w, 3*v, 5*x` richtig interpretiert. Mit `vnormalize(v)` erhält man den normierten Vektor. Das Skalarprodukt ist `vdot(v,w)`, das Vektorprodukt ist `vcross(v,w)`, den Winkel zwischen Vektoren erhält man mit `VAngle(v,w)` im Bogenmaß, mit `VAngleD(v,w)` im Gradmaß, letztere ist wie folgt definiert:

```
#macro VAngleD(V1, V2)
  degrees(acos(min(1,vdot(vnormalize(V1), vnormalize(V2)))))
#end
```

So ähnlich hätten wir das auch gemacht.

2.8 Bedingungen

Ein Ausdruck, dessen Auswertung gleich 0 ist, kann als der logischer Wert *falsch* verstanden werden, andere Werte ergeben *wahr*. Eine bedingte Anweisung („wenns so ist, tu das“) sieht wie folgt aus

```
#if (a<b)
  rechtsrum()
#else
  linksrum()
#end
```

Solche Anweisungen können auch verschachtelt werden.

2.9 Schleifen

Als Wiederholungsanweisung kennt Povray die `while`-Schleife:

```
#local i = 0;
#local s = 0;
#while (i < m)
  #local s = s + a[i];
  #local i = i+1;
#end
```

Es werden die Komponenten eines Felds addiert.

2.10 Felder

Matrizen werden mittels `array` deklariert und belegt:

```
#declare m = array[2][3]
  { {1,2,2},
    {3,4,5}
  }
```

Die Indizierung beginnt wie bei Java jeweils bei 0, im Beispiel ist $m[1][1] = 4$, rechts unten steht $m[1][2]$.
Das folgende stellt wiederum einen Vorgriff dar, zeigt aber, wie man mit Matrizen hantiert. Den Makros (später) muß nichts über den Typ der Parameter mitgeteilt werden, das merkt Povray selbst.

```
/** Transponierte von a*/
#macro transp(a)
 #local m = dimension_size(a,1);
 #local n = dimension_size(a,2);
 #local i = 0;
 #local at = array[n][m];
 #while (i < m)
   #local j = 0;
   #while (j < n)
     #local at[j][i]= a[i][j];
     #local j = j+1;
   #end
   #local i = i+1;
 #end
 at
#end
```

Das war eine „Funktions-Prozedur“, das Ergebnis wird am Ende „ausgegeben“. Im folgenden Beispiel wird das Ergebnis als Parameter zurückgegeben, dieser muß in der korrekten Größe initialisiert sein.

```
/** c = a * b; c muss initialisiert sein */
#macro mult(a, b, c)
 #local m = dimension_size(a,1);
 #local n = dimension_size(a,2);
 #local l = dimension_size(b,2);
 #local i = 0;
 #while (i < m)
   #local j = 0;
   #while (j < l)
     #local s = 0;
     #local k = 0;
     #while (k < n)
```

```
      #local s = s + a[i][k]*b[k][j];
      #local k = k+1;
    #end
    #local c[i][j] = s;
    #local j = j+1;
  #end
  #local i = i+1;
 #end
#end
```

3 Makros

Ein Makro ist eine Funktion oder eine Prozedur, die, mit Parameter versehen, ein Ergebnis liefert. Es ist nützlich, ein Makro zu deklarieren, wenn dieselben Aktionen mit verschiedenen Parametern ausgeführt werden sollen. Es wird nicht zwischen Wert- und Referenzparametern unterschieden, aber Referenzparameter müssen vor dem Aufruf deklariert sein.
Hier wird der Schnittpunkt der durch die Punkte A, B und C, D verlaufenden Geraden ermittelt; da diese aber windschief sein können, wird das Makro `mpi` aufgerufen

```
//E = Schnittpunkt der Geraden (A,B) und (C,D)
#macro schnitt(A,B,C,D,E)
  #local a = B-A;
  #local b = D-C;
  #local mm = array[3][2]
  {
    {a.x, -b.x},
    {a.y, -b.y},
    {a.z, -b.z}
  }
  #local r = array[3][1]
  {
    {C.x - A.x},
    {C.y - A.y},
    {C.z - A.z}
  }
  #local mp = array[2][3]
  mpi(mm, mp)                //Moore-Penrose-Inverse
  #local e = array[2][1]
  mult(mp,r,e)
  #local E = A + e[0][0]*a;
#end
```

In den obigen Beispielen `mult` und `schnitt` sind `c` bzw. `E` die Ergebnisse (hier ist `E` ein versteckter Referenzparamter).
Um ein Makro später aufzurufen, benutzt man:

```
object{[Name des Makros]([Parameter])}
```

```
  #declare E = <0,0,0>;
  schnitt(A, B, C, D, E)
```

oder (wie bei `transp`)

```
a = transp(b)
```

Das Ergebnis hat das (lokal) im Makro definierte Format.

Den Aufruf eines Makros kann man sich so vorstellen, daß der Text an der entsprechenden Stelle eingefügt wird. Das birgt Gefahren, deren man sich bewußt sein soll, um sie zu umgehen:

```
#macro f(a,b)
  a+b
#end
```

Wenn man nun `f(1,2)*f(3,4)` haben möchte, erhält man als Ergebnis 11, denn die Zeichenkette `"1+2*3+4"` wurde eingefügt. Das gewünschte Resultat erhält man, wenn

```
#macro f(a,b)
  (a+b)
#end
```

programmiert wird.

4 Gestaltungsmöglichkeiten

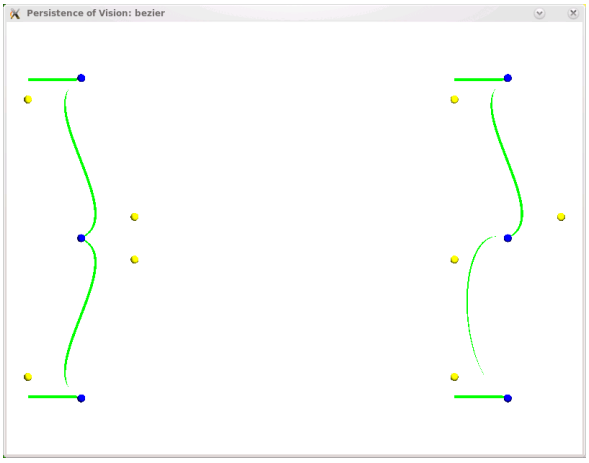
4.1 Drehbank

Wir verbinden einige Punkte in der (x, y)-Ebene und lassen diese Punktfolge um die y-Achse rotieren; die Übersetzung von „Drehbank“ ist `lathe` [2]. Die Art und Weise, wie die Punkte verbunden werden (der `spline_type`), kann festgelegt werden: Die Standardeinstellung ist `linear_spline`, die Punkte werden durch Geraden verbunden. Etwas abgerundet (aber nicht an allen Punkten) werden die Verbindungskurven bei `quadratic_spline`; richtig rund geht es bei `cubic_spline` zu, hier werden zur Bestimmung der Kurve zwischen zwei Punkten die beiden Nachbarpunkte berücksichtigt; Anfangs- und Endpunkt werden nur zur Orientierung benutzt, sie gehören nicht zum Objekt.
Bei LaTeX gibt es Paket `bezier`, mit dem Kommando

```
\bezier{punktzahl}(x1,y1)(x2,y2)(x3,y3)
```

[2] Wikipedia erklärt den Begriff so: Der Begriff stammt aus dem Schiffbau: eine lange dünne Latte (Straklatte, englisch spline), die an einzelnen Punkten durch Molche fixiert wird, biegt sich genau wie ein kubischer Spline mit natürlicher Randbedingung. Die Straklatte ist dabei bestrebt, ihre durch die Biegungen hervorgerufene innere Spannung zu minimieren bzw. zu verteilen.

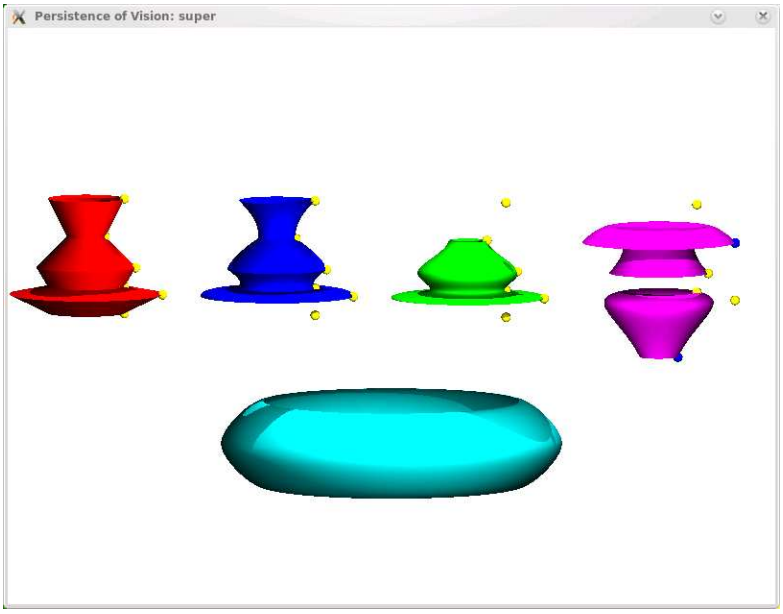
zeichnet man eine Parabel durch die Punkte P1 und P3, deren entsprechende Tangenten sich im Punkt P2 schneiden; durch Zusammensetzen solcher Kurvenstücke erhält man glatte Übergänge.
Bei der Povray-Option `bezier_spline` wird aus je vier Punkten ein Kurvenstück bestimmt, das den ersten und vierten Punkt verbindet; die Tangente der Kurve im Anfangspunkt ist die Gerade zum zweiten Punkt, die Tangente am Endpunkt ist die Grade zum dritten Punkt. Wenn die Nachbarpunkte eines gezeichneten Punkts auf einer Linie liegen, erhält man glatte Übergänge.



Wenn Anfangs- und Endpunkt gleich sind, entsteht eine geschlossene Kurve.

Hier werden die vier Spline-Typen verglichen, die Kontroll-Punkte sind gelb eingezeichnet:

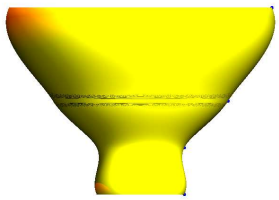
```
lathe{cubic_spline 6 <1,0>,<2,.5>,<1,.7>,<1.3,1.2>,<.5,2>,<1,3>
  pigment {Magenta}}
```



4.2 Rotationsflächen

Beim `sor`-Objekt [3] wird eine Funktion $x = f(y), \quad x > 0$ interpoliert, die durch Punkte `<x_i,y_i>` verläuft, dabei sind Anfangs- und Endpunkt als die Richtungen der Tangenten zum folgenden bzw. vorangehenden Punkt zu verstehen; die y-Werte müssen verschieden sein.

```
union
{
sor{ 6, <0,0>,<1,1>,<1,2>,<2,3>,<3,5>,<0,7>
  pigment{color Yellow}}
punkt(<0,0,0>, pigment{color Blue})
punkt(<1,1,0>, pigment{color Blue})
punkt(<1,2,0>, pigment{color Blue})
punkt(<2,3,0>, pigment{color Blue})
punkt(<3,5,0>, pigment{color Blue})
punkt(<0,7,0>, pigment{color Blue})
translate -3*y
}
```



Wenn die Option `open` verwendet wird, so werden Boden- und Deckfläche weggelassen. Die `sturm`-Option kann verwendet werden, wenn man mit der Qualität der Darstellung nicht zufrieden ist. Die Interpolation ergibt ein Polynom 3. Grades.

```
sor{5, <1,-2>,<0.5,-1>,<3,1>,<0.3,1.5>,<2,2>
    open sturm pigment{Red}}
```

4.3 Superquadratische Ellipsoide

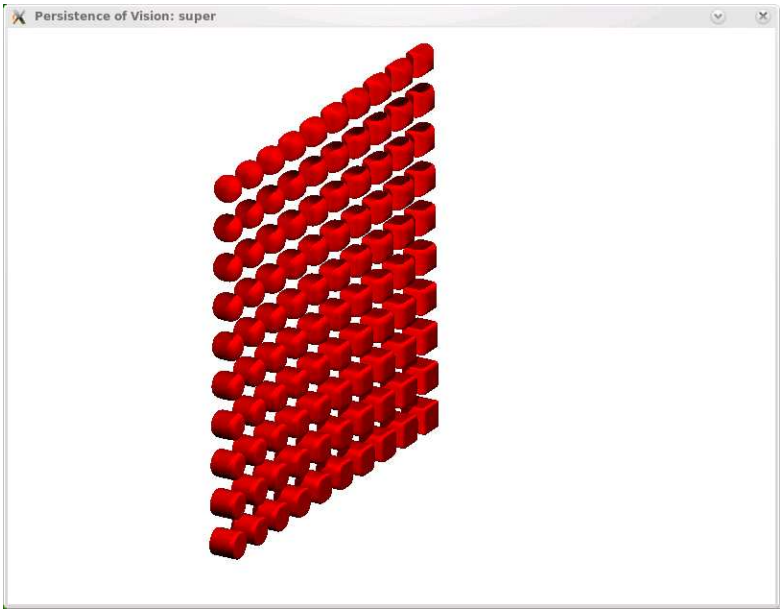
Durch die von zwei Parametern e, n abhängige Funktion

$$f_{e,n}(x,y,z) = (|x|^{\frac{2}{e}} + |y|^{\frac{2}{e}})^{\frac{e}{n}} + |z|^{\frac{2}{n}} - 1 = 0$$

[3] surface of revolution

wird eine Fläche bestimmt, die superquadratisches Ellipsoid genannt wird, die Parameter heißen Ost-West- bzw. Nord-Süd-Exponent und können Werte zwischen Null und Eins annehmen, das Objekt befindet sich innerhalb des Würfels mit den Ecken `<-1,-1,-1>` und `<1,1,1>`. Es entstehen würfelähnliche Figuren mit (in dieser oder jener Richtung mehr oder weniger ausgeprägt) abgerundeten Kanten; für $e = n = 1$ entsteht eine Kugel.

```
#declare i = 0.1;
#while (i <= 1)
  #declare j = 0.1;
  #while (j <= 1)
    superellipsoid{<i, j> pigment{Red}
      translate x*(-7 + 30*i)
      translate y*(-12 + 30*j)
    }
  #declare j = j + 0.1;
  #end
#declare i = i + 0.1;
#end
```



5 Videos

Ein Video ist eine Folge von Bildern, die, schnell nacheinander gezeigt, Bewegungen simulieren können. Wenn man sich eine Szene von allen Seiten anschauen will, so läßt man einfach die Kamera rund um den Schauplatz kreisen.
Der Schlüssel dazu ist die System-Variable `clock`, von der man eine Variable der Szene abhängig macht, etwa so:

```
#declare winkel = clock*360;
```

```
camera {
   location  <20*cos(winkel*pi/180), 8, 20*sin(winkel*pi/180)>
   angle 12
   look_at<0,0,0>}
```

Um die Bilder zu erzeugen, braucht man eine Initialisierungs-Datei, die wie – von Andreas Filler – vorgeschlagen, wie folgt aussehen könnte (hier ist nur bei `Input_File_Name` der aktuelle Dateiname einzusetzen):

```
; Initialisierungsdatei fuer die Erstellung eines Videos mithilfe von
POV-Ray
; A. Filler, 2002-2004 -------------------------------------------
Antialias=On                    ; Festlegungen zur Qualitaet
Antialias_Threshold=0.2         ; der berechneten Bilder
Antialias_Depth=3               ; (Antialiasing)

Input_File_Name=beispiel.pov    ; Name der POV-Ray-Datei, welche die
Szene enthaelt

Initial_Frame=1                 ; Nummer des ersten Bildes (Frame)
Final_Frame=99;                  ; Nummer des letzten Bildes (Frame)
Initial_Clock=0                 ; Wert fuer den "clock"-Parameter, der
dem 1. Bild entspricht
Final_Clock=1                   ; Wert fuer den "clock"-Parameter, der
dem letzten Bild entspricht

Pause_when_Done=off             ; Die Einzelbilder werden ohne Pausen
hintereinander erzeugt
Cyclic_Animation=off            ; Die Animation wird nur 1 mal gerendert
```

Es enstehen so viele Dateien, wie man will, die durchnummeriert sind; die Anzahl legt man ja selbst fest, aber den Nummerierungsmodus legt Povray selbst fest (es fängt bei 01 oder 001 oder 0001 an). Um bewegte Bilder zu sehen, gibt es unter Windows ein paar Werkzeuge, aber da kenne ich mich so gut aus: bei mir läuft nur Linux. Ich habe aber in einem der Java-Bücher von Guido Krüger ein Programm gefunden, das Bilder laufen lehrt, ich habe es an unsere Bedürfnisse angepaßt, hier ist es:

```
hgrass@hubert:~/povray> more video.java
/** Zeigt durch povray erzeugte Bilder als Video
Hubert Grassmann
hubert@grassmann.info
*/

import java.awt.*;
import java.awt.event.*;

public class video
extends Frame
implements Runnable
```

```
{
  Thread th;
  Image[] arImg;
  int actimage;
  static String bild; // 1. Eingabeparameter: Dateiname (ohne .pov)
  static int anzahl;  // 2. Eingabeparameter: Zahl der Bilder
  static int size=100;    // 3. Eingabeparameter: Pixelanzahl

  public static void main(String[] args)
  {
    if (args.length < 1)
    {
      System.out.println
  ("Benutzung: java5 video Bild_Titel Zahl_der_Bilder Pixelanzahl)");
      return;
    }
    bild = args[0];
    if (args.length > 1)
      anzahl = Integer.parseInt(args[1]);
    video wnd = new video();
    int size = Integer.parseInt(args[2]);
    wnd.setSize(size,size);
    wnd.setVisible(true);
    wnd.startAnimation();
  }

  public video()  { super(bild);  }

  public void startAnimation()
  {
    th = new Thread(this);
    actimage = -1;
    th.start();
  }

  public void run()
  {
    //Bilder laden
    arImg = new Image[anzahl];
    MediaTracker mt = new MediaTracker(this);
    Toolkit tk = getToolkit();
    for (int i = 1; i <= anzahl; ++i)
    {
      if (i < 10)
        arImg[i-1] = tk.getImage(bild+"0"+i+".png");
      else
      if (i < 100)
        arImg[i-1] = tk.getImage(bild+i+".png");
```

```
      else
        arImg[i-1] = tk.getImage(bild+i+".png");
      mt.addImage(arImg[i-1], 0);
      actimage = -i;
      repaint();
      try {
        mt.waitForAll();
      } catch (InterruptedException e) {
        //nothing
      }
    }
    //Animation beginnen
    actimage = 0;
    while (true) {
      repaint();
      actimage = (actimage + 1) % anzahl;
      try {
        Thread.sleep(50);
      } catch (InterruptedException e) {
        //nichts
      }
    }
  }

  public void paint(Graphics g)
  {
    if (actimage < 0) {
      g.drawString("Lade Bitmap "+(-actimage),10,50);
    } else {
      g.drawImage(arImg[actimage],10,30,this);
    }
  }
}
```

Das Problem, was mich aber noch nicht echt gestört hat, ist: das Programm läßt sich nur mit Gewalt (`<ctrl> c`) anhalten.

Aufgabe: Was wird wohl beim folgenden Video (das Sie erst herstellen müssen) passieren?

```
#include "colors.inc"
#declare intervall = 2;
#declare winkel = 12;
#declare hoehe = 5;
#include "anageoL2.inc"
#include "transforms.inc"
#version 3.5;
#declare LookAt = <0,0,0>;
#declare Cam = <1,1,-0.4>;
```

```
global_settings { max_trace_level 20 }

#declare winkel = clock*36;
camera {
   location <Cam.x*cos(winkel*pi/18),3,Cam.z*sin(winkel*pi/18)> *10
   angle 12
   look_at<0,0,0>}

background { color rgb <1, 1, 1> }
light_source { Cam * 50  color rgb <1, 1, 1> }

#macro guck(cam, st)
#declare CamN = vnormalize(cam);
#declare z3 = CamN;  // Bild von <0,0,-1>
#declare z1 = vnormalize(vcross(z3,y));
#declare z2 = vnormalize(vcross(z1,z3));
object
{
  text{ttf "timrom.ttf" st 0.1,0 pigment{Yellow}}
  matrix
  <
  z1.x,z1.y,z1.z,
  z2.x,z2.y,z2.z,
  z3.x,z3.y,z3.z,
  0,0,0>
}
#end

koordinatensystem(2)
guck(Cam, "F")
```

6 Makros für die Darstellung grundlegender Objekte

Es folgt die von Andreas Filler erstellte Datei `anageoL2.inc` (ein etwas sperriger Name).
Die Variable „`intervall`“ (s.u.) wurde eingebaut, um die Kameraeinstellung zu vereinfachen und Durchmesser von Zylindern, welche Strecken und Vektoren darstellen für den betrachteten Ausschnitt sinnvoll zu skalieren. Auch die Achsen des Koordinatensystems hängen von „`intervall`“ ab. Der Sinn ist also, eine sinnvolle Darstellung zu bekommen, unabhängig davon, wie groß die sinnervollerweise zu betrachtenden Koordinatenintervalle sind. Deshalb hängen in der `anageoL2` viele Makros von „`intervall`“ ab. Als grobe Faustregel gilt: Alle Objekte, deren x-, y- und z-Koordinaten zwischen `-intervall` und `intervall` liegen, sind in dem Bild zu erkennen. Natürlich gibt es da Ausnahmen, denn natürlich ist nicht genau ein Würfel sichtbar, und die Sichtbarkeit bestimmt sich auch durch die Perspektive. Die Kameraposition habe wird `anageoL2.inc`

wie folgt vorgenommen:

```
declare Cam = <60*intervall*cos(winkel*pi/180), 5*hoehe*intervall,
60*intervall*sin(winkel*pi/180)>;
```

Dazu werden ja im Kopf von Dateien, die `anageoL2.inc` aufrufen, immer `winkel`, `hoehe` und `intervall` definiert. Wie gesagt: Als Faustregel sind Objekte, deren x-, y- und z-Koordinaten zwischen `-intervall` und `intervall` liegen, sichtbar, manchmal muss man aber „nachjustieren“ (durch Veränderung des Wertes für `intervall`).

Bevor diese Datei eingebunden wird, müssen also folgende Variable belegt sein:

```
#declare intervall = 5;// von der Kamera erfasster Bildausschnitt
#declare winkel = -60; // Winkel der Kamera zur x-Achse (in Grad)
#declare hoehe =   4 ; // Hoehe (relativ) der Kamera ueber der x-z-Ebene
#declare KameraManuell = 0; // oder 1


// Makropaket für die Darstellung von Vektorpfeilen, Ebenen,
Koordinatenachsen,...
// Linkshändiges Koordinatensystem

// A. Filler, 2001 - 2004
// H. Grassmann, 2008
// KORREKTUREN: Nov. 2008, Jan. 2009
// Version: 15. Jan. 2009
// ------------------------------------------------------------
#version 3.5;            // benötigte POV-Ray-Version
#include "transforms.inc"
background { color rgb <1, 1, 1> }
```

Nun folgende die Makros, die dort bereitgestellt werden:

- Kamera-Einstellung

```
// ------------------------------------------------------------
// Definition der Kamera und der Lichtquellen
#if (KameraManuell = 1)
 #declare Cam = KamPos;
#else
 #declare Cam = <60*intervall*cos(winkel*pi/180), 5*hoehe*intervall,
   60*intervall*sin(winkel*pi/180)>;
#end
#declare CamN = vnormalize(<Cam.x,Cam.y,Cam.z>);
#declare z3 = CamN;  // Bild von <0,0,-1>
#if (z3.x*z3.x+z3.z*z3.z=0)
 #declare z1 = <1,0,0>;
#else
 #declare z1 = vnormalize(vcross(z3,y));
#end
#declare z2 = vnormalize(vcross(z1,z3));
```

Die Vektoren $z1, z2, z3$ dienen später dazu, Objekte auf die Kamera auszurichten; sie werden als die Zeilen einer Transformationsmatrix interpretiert.

```
camera { orthographic
         location  Cam
         right x*4/3*2.7*intervall   up y*2.7*intervall
         look_at   <0, 0,  0>
}
light_source { <2.5*intervall, 2*intervall, -5*intervall>
color rgb <1, 1, 1> }
light_source { <0,10*intervall,2*intervall>  color rgb <1, 1, 1> }
light_source { CamN*50  color rgb <1, 1, 1> }
```

- Makro für die Darstellung von Ortsvektoren

```
#macro ortsvektor (V , textur)
#declare vektorbetrag=sqrt(V.x*V.x+V.y*V.y+V.z*V.z);
// Betrag des Vektors berechnen
#if ( V.x*V.x+V.y*V.y = 0 )
  #local VORZEICHEN = (V.z)/abs((V.z));
  #declare PFEIL=union
   {
    cylinder{<0,0,0> <0, 0, VORZEICHEN*(vektorbetrag-0.15*intervall)>
     intervall/60}
    cone{<0,0,VORZEICHEN*(vektorbetrag-0.15*intervall)> intervall/30
         <0,0,VORZEICHEN*vektorbetrag> 0 } texture {textur}  no_shadow
   };
  object {PFEIL}
#else
  #declare PFEIL=union{ cylinder{<0,0,0>
    <vektorbetrag-0.15*intervall, 0, 0> intervall/60}
  cone{ <vektorbetrag-0.15*intervall,0,0> intervall/30
        <vektorbetrag,0,0> 0 }
   texture {textur}  no_shadow
  };
  #declare NX=vnormalize(V);
  #declare NY=vnormalize(vcross(NX,z));
  #declare NZ=vnormalize(vcross(NX,NY));
  object {PFEIL
  matrix < NX.x , NX.y , NX.z ,
           NY.x , NY.y , NY.z ,
           NZ.x , NZ.y , NZ.z ,
           0    , 0    ,    0  >
  }
#end
#end
```

- ein Punkt

```
// Punkt (als kleines Kuegelchen)
#macro punkt (P, textur)
sphere{<P.x,P.y,P.z>, intervall/40 texture{textur}}
#end
```

- Verbindungsvektoren zwischen zwei Punkten

```
// Makro fuer die Darstellung von Verbindungsvektoren zwischen
zwei Punkten
#macro verbindungsvektor (P, Q , textur)
object {ortsvektor (Q-P, textur) translate P}
#end
```

- Pfeile mit beliebigen Anfangspunkten

```
// Makro fuer die Darstellung von Pfeilen mit beliebigen
Anfangspunkten
#macro vektoranpunkt (Punkt, Vektor , textur)
verbindungsvektor (Punkt, Punkt + Vektor , textur)
#end
```

- Strecken

```
// Makro fuer die Darstellung von Strecken
#macro strecke (P, Q , textur)
#declare V=Q-P;
#declare vektorbetrag=sqrt(V.x*V.x+V.y*V.y+V.z*V.z);
  // Laenge der Strecke berechnen
#if ( V.x*V.x+V.y*V.y+V.z*V.z = 0 )
  #else
#if ( V.x*V.x+V.y*V.y = 0 )
  #declare Streckenzylinder=cylinder{<0,0,0> V intervall/115
              texture {textur}  no_shadow
  };
  object {Streckenzylinder  translate <P.x,P.y,P.z>}
#else
  #declare Streckenzylinder=cylinder{<0,0,0> <vektorbetrag, 0, 0>
  intervall/115 texture {textur}  no_shadow
  };
  #declare NX=vnormalize(V) ;
  #declare NY=vnormalize(vcross(NX,z)) ;
  #declare NZ=vnormalize(vcross(NX,NY)) ;
  object {Streckenzylinder
  matrix < NX.x , NX.y , NX.z ,
           NY.x , NY.y , NY.z ,
           NZ.x , NZ.y , NZ.z ,
           P.x  , P.y  , P.z   >
  }
```

```
#end
#end
#end
```

- Geraden

```
// Makro fuer die Darstellung von Geraden (als sehr lange Strecken,
//die ueber das Blickfeld hinausreichen
#macro gerade (P, Q , textur)
#declare V=Q-P;
#declare vektorbetrag=sqrt(V.x*V.x+V.y*V.y+V.z*V.z);
// Laenge der Strecke berechnen
#if ( V.x*V.x+V.y*V.y = 0 )
  #declare Streckenzylinder=cylinder{<0,0, - 100 * intervall>
    <0, 0, 100 * intervall> intervall/150 texture {textur} no_shadow
  };
  object {Streckenzylinder  translate <P.x,P.y,P.z>}
#else
  #declare Streckenzylinder=cylinder{< - 100 * intervall,0,0>
   <100 * intervall,0, 0> intervall/150
               texture {textur}  no_shadow
  };
  #declare NX=vnormalize(V) ;
  #declare NY=vnormalize(vcross(NX,z)) ;
  #declare NZ=vnormalize(vcross(NX,NY)) ;
  object {Streckenzylinder
  matrix < NX.x , NX.y , NX.z ,
           NY.x , NY.y , NY.z ,
           NZ.x , NZ.y , NZ.z ,
           P.x  , P.y  , P.z   >
  }
#end
#end
```

- Abzissen

```
// "pluspunkt": Punkt mit Verbindungsstrecken zu den Achsen
#macro pluspunkt (PK, textur)
sphere{<PK.x,PK.y,PK.z>, intervall/40
 texture{textur}}
strecke( <PK.x,PK.y,PK.z>,<PK.x,0,PK.z>,
 texture { pigment { rgbf<0.6,0.6,0.6,0.9> }
 finish { ambient 0.7 diffuse 0.5 roughness 0.1 }} )
strecke(<PK.x,0,0>, <PK.x,0,PK.z>,
 texture { pigment { rgbf<0.6,0.6,0.6,0.9> }
 finish { ambient 0.7 diffuse 0.5 roughness 0.1 }} )
strecke(<0,0,PK.z>, <PK.x,0,PK.z>,
 texture { pigment { rgbf<0.6,0.6,0.6,0.9> }
```

```
 finish { ambient 0.7 diffuse 0.5 roughness 0.1 }} )
#end
```

- Koordinatensystem

  Hier kommt die ganz oben (Kameraeinstellung) eingeführte Transformationsmartrix zur Verwendung, um die Bezeichnungen der Koordinatenachsen auch bei „schräger“ Kameraansicht ordentlich sehen zu können.

```
//  Makro fuer die Darstellung eines Koordinatensystems mit
Achsenbezeichnungen und Skaleneinteilung
#macro koordinatensystem ( achsenlaenge )
#declare kstextur = texture { pigment { rgb<0.1,0.1,0.3> }
    finish { ambient 0.0 diffuse 0.1 reflection 0.1 brilliance 1
  specular 0.3 roughness 0.2 }
} ;
#declare KSFont="cyrvetic.ttf" ;
#declare KSSchriftgroesse=achsenlaenge/5 ;
#declare skalenkugelradius=achsenlaenge/70;
union{
// x-Achse:
 cylinder{-x*1.1*achsenlaenge , x*1.05*achsenlaenge, achsenlaenge/120}
 cone{ x*1.2*achsenlaenge, 0 , x*1.05*achsenlaenge, achsenlaenge/48}
 #declare Count=-achsenlaenge;
 #while (Count < achsenlaenge+1)
   sphere {<Count,0,0>, skalenkugelradius}
 #declare Count=Count+1;
 #end
// y-Achse:
 cylinder{-y*1.1*achsenlaenge , y*1.05*achsenlaenge, achsenlaenge/120}
 cone{ y*1.2*achsenlaenge, 0 , y*1.05*achsenlaenge, achsenlaenge/48}
 #declare Count=-achsenlaenge;
 #while (Count < achsenlaenge+1)
   sphere {<0,Count,0>, skalenkugelradius}
 #declare Count=Count+1;
 #end
// z-Achse:
 cylinder{-z*1.1*achsenlaenge , z*1.05*achsenlaenge, achsenlaenge/120}
 cone{ z*1.2*achsenlaenge, 0 , z*1.05*achsenlaenge, achsenlaenge/48}
 #declare Count=-achsenlaenge;
 #while (Count < achsenlaenge+1)
    sphere {<0,0,Count>, skalenkugelradius}
 #declare Count=Count+1;
 #end
// Achsenbezeichnungen:
 text{ttf KSFont "x",0.1,0  scale KSSchriftgroesse
 matrix  < z1.x,z1.y,z1.z,
           z2.x,z2.y,z2.z,
           z3.x,z3.y,z3.z,
```

```
          0,0,0 >
 translate <1.08*achsenlaenge,-achsenlaenge/7,0>
 }
 text{ttf KSFont "y",0.1,0  scale KSSchriftgroesse
 matrix  < z1.x,z1.y,z1.z,
           z2.x,z2.y,z2.z,
           z3.x,z3.y,z3.z,
           0,0,0 >
  translate <-achsenlaenge/8,1.09*achsenlaenge,0>
 }
 text{ttf KSFont "z",0.1,0  scale KSSchriftgroesse
 matrix  < z1.x,z1.y,z1.z,
           z2.x,z2.y,z2.z,
           z3.x,z3.y,z3.z,
           0,0,0 >

 translate <achsenlaenge/20,0,1.1*achsenlaenge>
 }
texture {kstextur}
no_shadow
}
#end
```

- durch eine Gleichung gegebene Ebene

```
// Makro fuer die Erzeugung von "Ebenen" als duenne Quader aus
einer Gleichung in drei Variablen
#macro ebene (A,B,C,D,textur)
#if ( B = 0 )
 #if ( C = 0 )
  box {<-intervall/1000 , - intervall , -intervall>
       <intervall/1000 , intervall, intervall>
    texture {textur}   no_shadow
    translate <-D/A,0,0>
  }
 #else
  box {<-intervall , - intervall , -intervall/1000>
       <intervall , intervall , intervall/1000>
    texture {textur}   no_shadow
    rotate <0, degrees(atan2(A,C)) ,0>
    translate <0,0,-D/C>
  }
 #end
#else
 #if ( A*A + C*C = 0 )
  box {<-intervall , - intervall/1000 , -intervall>
       <intervall , intervall/1000, intervall>
  translate <0,-D/B,0>
```

```
  texture {textur}   no_shadow
  }
 #else
  #declare VX1 = <0,1,0>;
  #declare VX2=vnormalize(<A,B,C>);
  #declare VY=vnormalize(vcross(VX2,VX1));
  #declare VZ1=vcross(VY,VX1);
  #declare VZ2=vcross(VY,VX2);
  box {<-intervall , - intervall/1000 , -intervall>
       <intervall , intervall/1000, intervall>
  matrix < VX1.x, VY.x, VZ1.x,
           VX1.y, VY.y, VZ1.y,
           VX1.z, VY.z, VZ1.z,
               0     0      0 >
  matrix < VX2.x, VX2.y, VX2.z,
            VY.x,  VY.y,  VY.z,
           VZ2.x, VZ2.y, VZ2.z,
               0,     0,     0 >
  translate <0,-D/B,0>
  texture {textur}   no_shadow
  }
 #end
#end
#end
```

- Ebene, mittels Punkt und aufspannende Vektoren gegeben

```
// Makro fuer die Erzeugung von "Ebenen" als duenne Quader
aus einer Parameterdarstellung
#macro ebenepar (P,a,b,textur)
#declare normvekt = vcross(a,b);
#declare A = normvekt.x;
#declare B = normvekt.y;
#declare C = normvekt.z;
#if ( B = 0 )
 #if ( C = 0 )
  box {<-intervall/1000 , - intervall , -intervall>
       <intervall/1000 , intervall, intervall>
    texture {textur}   no_shadow
    translate P
  }
 #else
  box {<-intervall , - intervall , -intervall/1000>
       <intervall , intervall , intervall/1000>
    texture {textur}   no_shadow
    rotate <0, degrees(atan2(A,C)) ,0>
    translate P
  }
```

```
 #end
#else
 #if ( A*A + C*C = 0 )
  box {<-intervall , - intervall/1000 , -intervall>
       <intervall , intervall/1000, intervall>
  translate P
  texture {textur}   no_shadow
  }
 #else
  #declare VX1 = <0,1,0>;
  #declare VX2=vnormalize(<A,B,C>);
  #declare VY=vnormalize(vcross(VX2,VX1));
  #declare VZ1=vcross(VY,VX1);
  #declare VZ2=vcross(VY,VX2);
  box {<-intervall , - intervall/1000 , -intervall>
       <intervall , intervall/1000, intervall>
  matrix < VX1.x, VY.x, VZ1.x,
           VX1.y, VY.y, VZ1.y,
           VX1.z, VY.z, VZ1.z,
               0     0      0 >
  matrix < VX2.x, VX2.y, VX2.z,
            VY.x,  VY.y,  VY.z,
           VZ2.x, VZ2.y, VZ2.z,
               0,     0,     0 >
  translate P
  texture {textur}   no_shadow
  }
 #end
#end
#end
```

Die (hier gekürzte Datei) enhält noch Definitionen von Farben und Texturen, die im Mathematik-Kontext nicht so wichtig sind; vgl. hierzu die Quellen.

7 Flächen

7.1 Algebraische Flächen

Eine algebraische Fläche ist die Menge aller Nullstellen eines Polynoms $f(x, y, z)$ in x, y, z. Wenn f den Grad 1 hat, so handelt es sich um eine Ebene (`plane`). Flächen vom Grad 2 heißen quadratische Flächen; in der linearen Algebra wird deren Klassifikation behandelt. Man erstellt eine solche Fläche mit dem Makro `quadric`:

```
quadric {<A,B,C>,<D,E,F>,<G,H,I>,J [OBJECT_MODIFIERS...] }
```

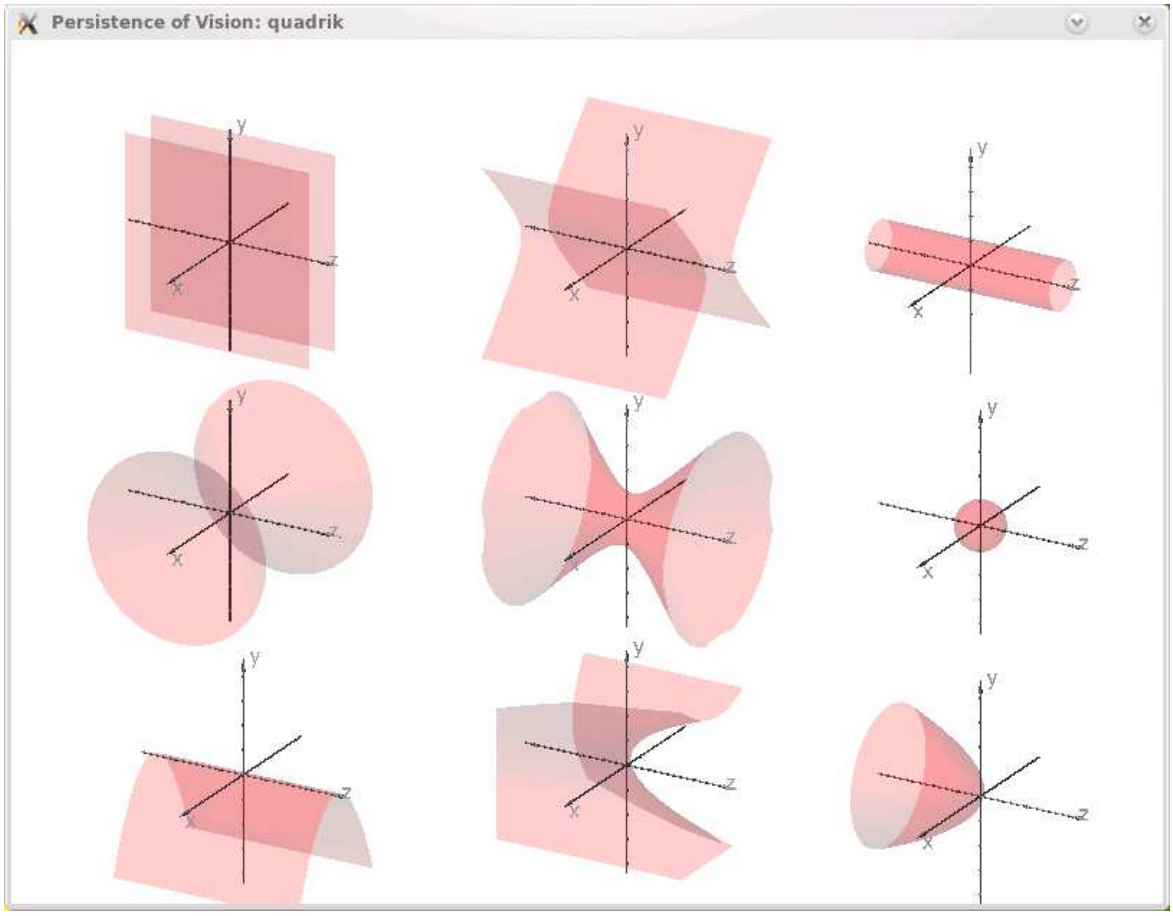
Die zehn Parameter entsprechen den Koeffizienten des Polynoms

$$Ax^2 + By^2 + Cz^2 + Dxy + Exz + Fyz + Gx + Hy + Iz + J.$$

Im 2. Semester lernt man, daß ein solches Polynom durch die „Hauptachsentransformation“ in eine der folgenden Formen transformiert werden kann:

$$\begin{array}{ccc} x^2 = 1 & x^2 - y^2 = 1 & x^2 + y^2 = 1 \\ x^2 - y^2 - z^2 = 1 & x^2 + y^2 - z^2 = 1 & x^2 + y^2 + z^2 = 1 \\ x^2 + 2y = 0 & x^2 - y^2 + 2z = 0 & x^2 + y^2 + 2z = 0 \end{array}$$

Das heißt, die Koeffizienten D,E,F,G,H sind eigentlich überflüssig; durch eine `matrix`-Transformation kann man die Hauptachsentransformation ja rückgängig machen.
Hier sind die Bilder:



Auf einem einschaligen Hyperboloid liegen Geraden: Sei durch

$$H: \; x^2 + y^2 - z^2 - 1 = 0$$

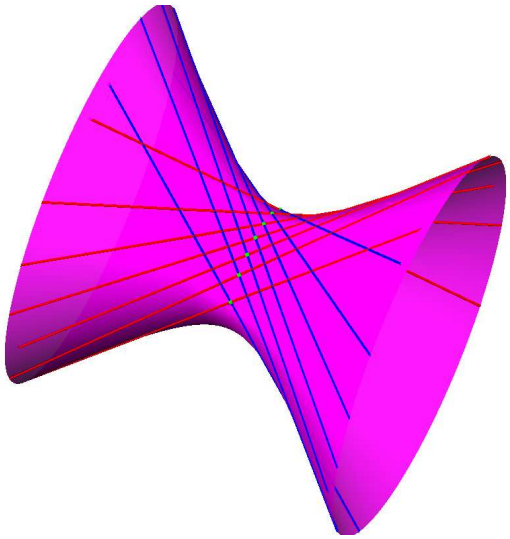
eines gegeben, wir wählen einen Punkt `P = <x1,y1,0>` auf dem „Mittelkreis“, also $x_1^2 + y_1^2 - 1 = 0$ und suchen einen Vektor `v = <x2, y2, 1>`, so daß die Punkte $P + tv$ für beliebiges t alle auf H liegen. Das bedeutet

$$(x_1 + tx_2)^2 + (y_1 + ty_2)^2 - t^2 - 1 = 0.$$

Nach Ausmultplizieren und unter der Beachtung, daß dies für alle t gelten soll, erhalten wir

$$x_2 = \frac{\pm y_1}{\sqrt{x_1^2 + y_1^2}}, \; y_2 = \frac{\mp x_1}{\sqrt{x_1^2 + y_1^2}}.$$

```
 quadric
{
  <1,1,-1>  <0, 0, 0>  <0, 0, 0> (-1)
      texture {  pigment{color rgbt<1,0,1,0.1>}}
   clipped_by { box {-intervall-x-y, intervall+x+y} }
 no_shadow
}

#declare x1 = -0.9;
#while(x1 < 1)
  #declare y1 = sqrt(1-x1*x1);    // <x1,y1,0> auf Kreis
  #declare w = sqrt(x1*x1+y1*y1);
  #declare x2 = -y1/w;            // <x1+x2,y1+y2,1> auf Hyperboloid
  #declare y2 = x1/w;
  punkt(<x1,y1,0>, texture{pigment{color Green}})
  object{ union{
  gerade(<x1,y1,0>, <x1+x2,y1+y2,1> , texture{pigment{color Red}})
  gerade(<x1,y1,0>, <x1-x2,y1-y2,1> , texture{pigment{color Blue}})
  }
  clipped_by { box {-intervall, intervall} }
  }
  #declare x1 = x1 + 0.3;
#end
```



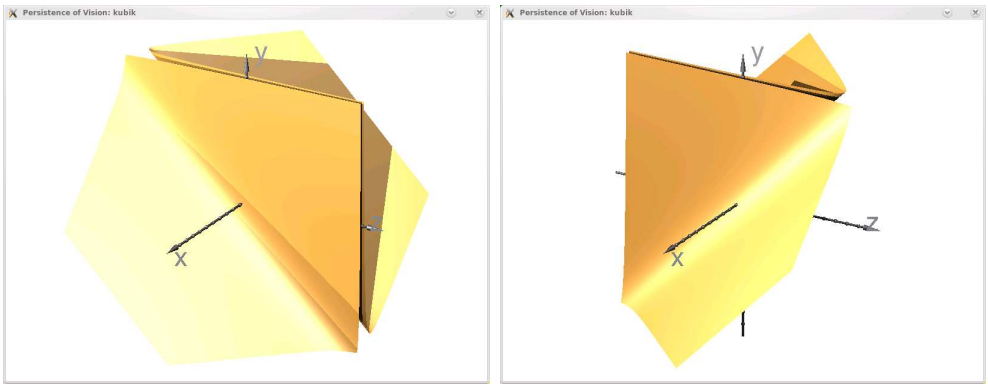
Ein Polynom dritten Grades kann durch

```
cubic { <A1, A2, A3,... A20> [POLY_MODIFIERS...] }
```

angegeben werden:

```
cubic{<1,1,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,-1>
  clipped_by { box { <-10,-10,-10><10,10,10> } }
  bounded_by { clipped_by }
```

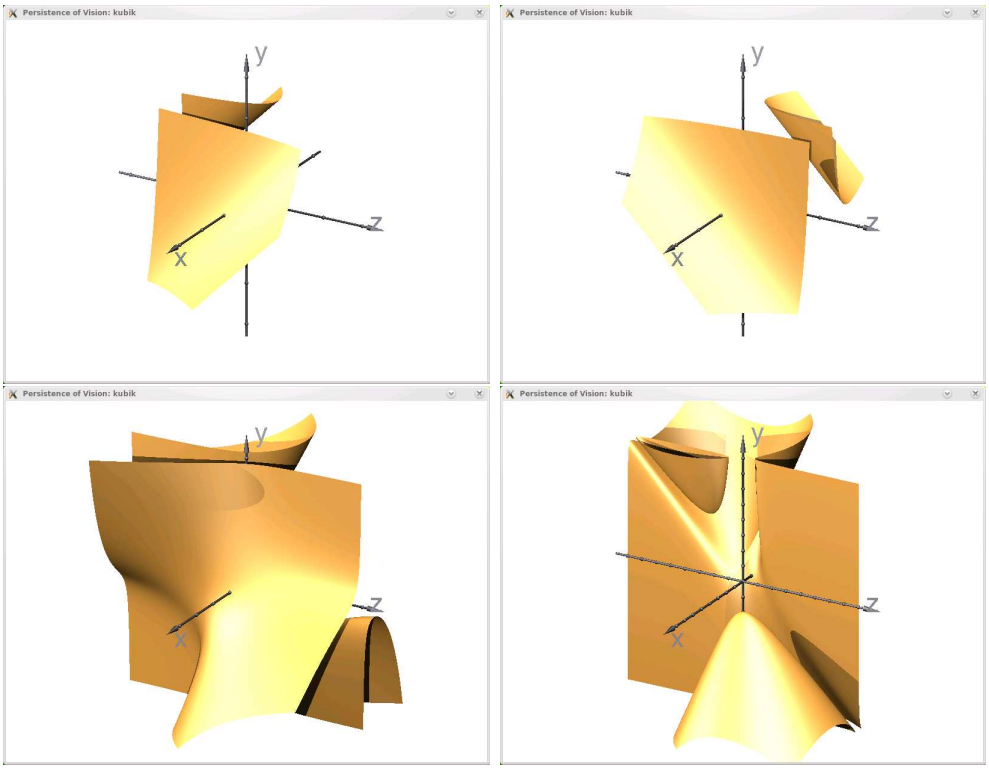
```
  pigment { rgb <1,.7,.3> } finish { specular .5 }
}
```

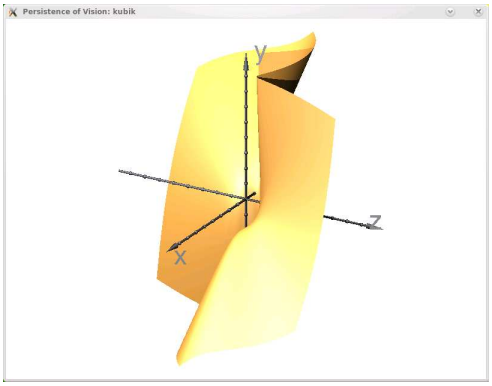


Die folgenden Flächen sind durch

$$x^3 + x^2y - x^z = 1$$
$$x^3 + x^2y + xyz = 1$$
$$x^3 + x^2y + xyz + xz^2 = -1$$
$$x^3 + x^2y + xyz + xz^2 - z^3 = -1$$

gegeben:





Ein Polynom vierten Grades erhalten wir mittels

```
quartic { <A1, A2, A3,... A35> [POLY_MODIFIERS...] }
```

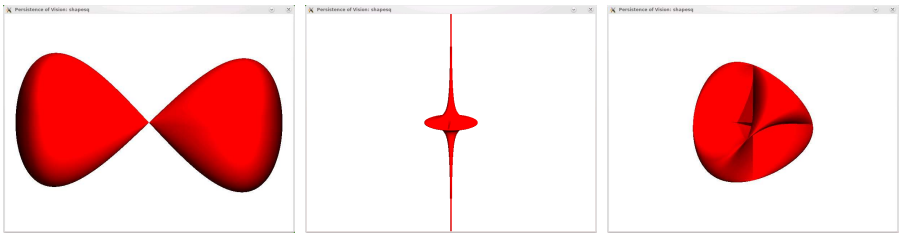
In der Datei `shapesq.inc` sind einige Flächen vierten Grades vorbereitet, die man aber in ein Objekt einfügen muß (sonst sieht man schwarz). In der Quelltextzeile 148 wird ein Fehler angezeigt, man sollte dies weiträumig auskommentieren, das darf allerdings nur `root` machen. Ein Beispiel:

```
/* Lemniscate of Gerono
  This figure looks like two teardrops with their pointed ends connected.
  It is formed by rotating the Lemniscate of Gerono about the x-axis.
  The formula is:
     x^4 - x^2 + y^2 + z^2 = 0. */
#declare Lemniscate =
 quartic
  {< 1,   0,   0,   0,   0,   0,   0,   0,   0, -1,
     0,   0,   0,   0,   0,   0,   0,   0,   0,  0,
     0,   0,   0,   0,   0,   1,   0,   0,   0,  0,
     0,   0,   1,   0,   0>
  }
```

Die Kommandos

```
object{Lemniscate scale 5 pigment{Red}}
/* Quartic Cylinder - a Space Needle?  */
object{Quartic_Cylinder  pigment{Red}}
object{Steiner_Surface  scale 3 pigment{Red}}
```

ergeben diese Bilder:



Diese Polynomdarstellungen sind Spezialfälle des allgemeineren `poly`-Objekts:

```
POLY:
    poly { Grad, <A1, A2, A3,... An> [POLY_MODIFIERS...] }
POLY_MODIFIERS:
    sturm | OBJECT_MODIFIER
```

Als Grad sind die Werte 2 bis 15 zugelassen.
Die Darstellung der Flächen gelingt nicht immer perfekt, insbesondere, wenn die Fläche nicht glatt ist. Hier kann man (auf Kosten der Rechenzeit) versuchen, mit der Option `sturm` eine Verbesserung zu erreichen.

Die im Feld `<A1,...>` angegebenen Werte werden als Koeffizienten der Monome $x^i y^j z^k$ interpretiert, wobei die Monome in lexikografischer Weise geordnet sind (das Leerzeichen ist der letzte Buchstabe); für Polynome vom Grad 2 ist das die folgende Reihenfolge:

$$x^2, xy, xz, x, y^2, yz, y, z^2, z, 1.$$

Die Anzahl der anzugebenden Koeffizienten sieht man hier:

Grad	2	3	4	5	6	7	d	15
Monomzahl	10	20	35	56	84	120	$\binom{3+d}{3}$	816

Die Angabe der Koeffizienten an den „richtigen“ Stellen des Vektors ist etwas unhandlich. Wir können aus der Folge

```
3
xxz
4
yzz
52.3
zzz
600
```

leicht das Polynom $3x^2z+4yz^2+52.3z^3+600$ rekonstruieren (beachte: die letzte Zeile ist hier leer). Wir kodieren ein Monom $x^p y^q z^r 1^s$ durch seinen Exponentenvektor (p, q, r, s), die Summe der Komponenten ist für alle Monome eines Polynoms vom Grad d gleich d. Wenn wir den maximalen Grad auf 9 beschränken, können wir diesen Vektor als vierstellige Dezimalzahl zwischen d und $1000d$ auffassen, die Anzahl dieser Zahlen mit der Quersumme d ist wiederum gleich $\binom{3+d}{3}$, die Menge dieser Zahlen zu erzeugen und zu ordnen ist nicht schwer. Das Ergebnis sei eine Liste L. (Die Ordnung der Zahlen entspricht gerade der lexikografischen Monomordnung.)
Wir lesen die obige Folge aus einer vorbereiteten Datei, suchen zu jedem Monom seinen Index i in der Liste L und schreiben den Koeffizienten in eine weitere Liste an der Stelle i. Daraus erzeugen wir uns den benötigten Vektor, in unserem Fall ist es

```
<.0,.0,3.0,.0,.0,.0,.0,.0,.0,.0,.0,.0,4.0,.0,.0,52.3,.0,.0,600.0>
```

Das folgende Java-Programm leistet dies:

```
import HUMath.Algebra.*;
/**
@author Hubert Grassmann, hgrass@mathematik.hu-berlin.de
@version 24.02.2009
gelesene Polynome in POV-ray "poly" umwandeln
*/
public class PVpoly
{

/** liest Polynom aus Datei name; z.B.
20    Monom-Anzahl
3     Grad
1.2   Koeff
xxy   Monom
-4.0  Koeff
xyz   Monom
... */

static int[] monome;
static double[] PVfeld;

public static int quersumme(int m)
{
  int s = m % 10;
  for (int i = 1; i <= 4; i++)
  {
    m = m / 10;
    s = s + m % 10;
  }
  return s;
}

/** bubble sort */
public static void sort(int[] tt)
{
  int i, j, l = tt.length-1;
  int tausch;
  for (i = 0; i <= l-1; i++)
    for (j = i+1; j <= l; j++)
      if (tt[i] <= tt[j])
      {
        tausch = tt[i];
        tt[i] = tt[j];
        tt[j] = tausch;
      }
```

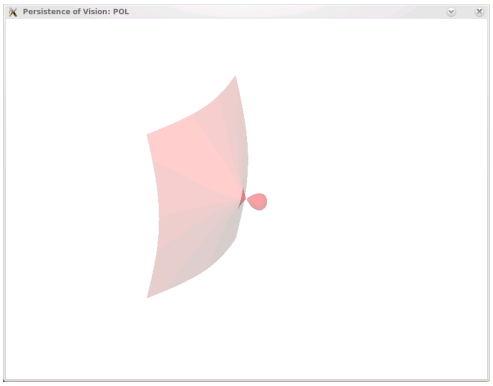
```
}
/** fuellt monome vom Grad d */
public static void init(int d)
{
  monome = new int[(d+3)*(d+2)*(d+1)/6];
  PVfeld = new double [(d+3)*(d+2)*(d+1)/6];
  int j = 0;
  for (int i = d; i <= d*1000; i++)
    if (d == quersumme(i))
    {
      monome[j] = i;
      j++;
    }
  sort(monome);
}

/** sucht Monom m in monome */
public static int find(int m)
{
  int i = 0;
  while (i < monome.length)
    if (m == monome[i])
      return i;
    else
      i++;
    return -1;
}

public static double[] fromFile(String name)
{
  int i,j,d,anz;
  double a;
  anz = Integer.parseInt(B.liesDatei(name, 0, 1)[0]);
  d = Integer.parseInt(B.liesDatei(name, 1, 2)[0]);
  init(d);
  String[] s = B.liesDatei(name, 2, anz*2+2);
  double aus[] = new double[(d+3)*(d+2)*(d+1)/6];
  //Koeff an die Stelle vom Monom-Index
  for (i=0; i<anz*2; i=i+2 )
  {
    a = Double.parseDouble(s[i]);
    j = find(monomCode(s[i+1],d));
    aus[j] = a;
  }
  return aus;
}

/** Monom xxyyzz vom Grad d wird in eine Zahl umgewandelt:
```

```
x = 1000, xx = 2000, y = 100 ... Leer: Grad
Grad < 10 */
public static int monomCode(String s, int d)
{
  int c = 0;
  for (int i = 0; i < s.length(); i++)
  {
    if (s.charAt(i) == 'x')
      c = c + 1000;
    else if (s.charAt(i) == 'y')
      c = c + 100;
    else if (s.charAt(i) == 'z')
      c = c + 10;
  }
  int q = quersumme(c);
  c = c+d-q;
  return c;
}

public static void main(String[] args)
{
double a[] = fromFile("DingDongSurface");
String s = "<" + a[0];
for (int i=1; i < a.length; i++)
{
  B.wr(a[i] + " ");
  s = s +","+ a[i];
}
s = s + ">";
B.wl(s);
B.schreibStringInDatei("POL3", s);
}
}
```

Die sogenannte Ding-Dong-Fläche ist durch $x^2 + y^2 = (1 - z)z^2$ gegeben, das obige Programm ergibt

$$< 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, -1.0, 0.0, 0.0 >$$

und die sieht dann so aus:



Der Torus mit den Radien r_0, r_1 wird durch die Gleichung

$$x^4+y^4+z^4+2x^2y^2+2x^2z^2+2y^2z^2-2(r_0+r_1)x^2+2(r_0-r_1)y^2-2(r_0+r_1)z^2+(r_0-r_1)^2=0$$

gegeben, zum Beispiel:

```
// Torus mit Radien sqrt(40), sqrt(12)
poly { 4,
  < 1,   0,   0,   0,   2,   0,   0,   2,   0,
 -104,   0,   0,   0,   0,   0,   0,   0,   0,
    0,   0,   1,   0,   0,   2,   0,  56,   0,
    0,   0,   0,   1,   0, -104,  0, 784 >
  sturm
}
```

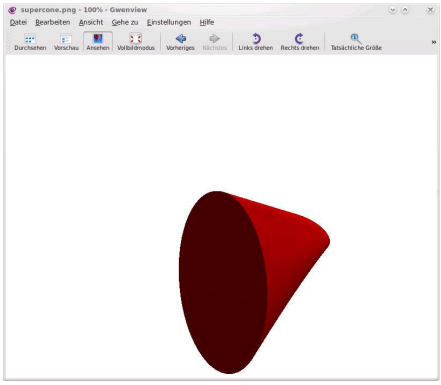
Das folgende Makro aus der Datei `shapes.inc` erzeugt einen Kegelstumpf mit elliptischen Grund- und Oberflächen:

```
// Supercone author: Juha Nieminen
// A cone object where each end is an ellipse, you specify two radii
// for each end.
// SuperCone function: (x^2/a^2+y^2/b^2-1)*(1-z) + (x^2/c^2+y^2/d^2-1)*z = 0
//
#macro Supercone(PtA, A, B, PtB, C, D)
   intersection {
      quartic {
         <0, 0,  0,  0,  0,  0,  0,  B*B-2*B*D+D*D, 2*(B*D-B*B), B*B,
         0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
         0,  0,  0, A*A-2*A*C+C*C, 2*(A*C-A*A), A*A, 0,  0,  0,  0,
         -(A*A-2*A*C+C*C)*(B*B-2*B*D+D*D),
         -(2*((B*D-B*B)*(A*A-2*A*C+C*C)+(A*C-A*A)*(B*B-2*B*D+D*D))),
         -(B*B*(A*A-2*A*C+C*C)+4*(A*C-A*A)*(B*D-B*B)+A*A*(B*B-2*B*D+D*D)),
         -(2*(B*B*(A*C-A*A)+A*A*(B*D-B*B))), -A*A*B*B>
         sturm
      }
      cylinder {0, z, max(max(abs(A), abs(B)), max(abs(C), abs(D)))}
      bounded_by {cone {0, max(abs(A), abs(B)), z, max(abs(C), abs(D))}}
```

```
      #local Dirv = PtB - PtA;
      scale <1,1,vlength(Dirv)>
      #local Dirv = vnormalize(Dirv);
      #if(vlength(Dirv-<0,0,-1>)=0) scale <1,1,-1>
//      #else Reorient_Trans(z, Dirv)
      #end
      translate PtA
   }
#end
```

Der dort angegebene Beispiel-Aufruf wurde korrigiert:

```
camera { location <6,5,-10> look_at 0 angle 35 }
light_source { <100,100,-20>,1 }
light_source { <-100,100,-20>,1 }
object { Supercone(<0,-1.5,0>,1,2, <0,1.5,0>,1,.5)
    pigment { rgb x } finish { specular .5 } }
```



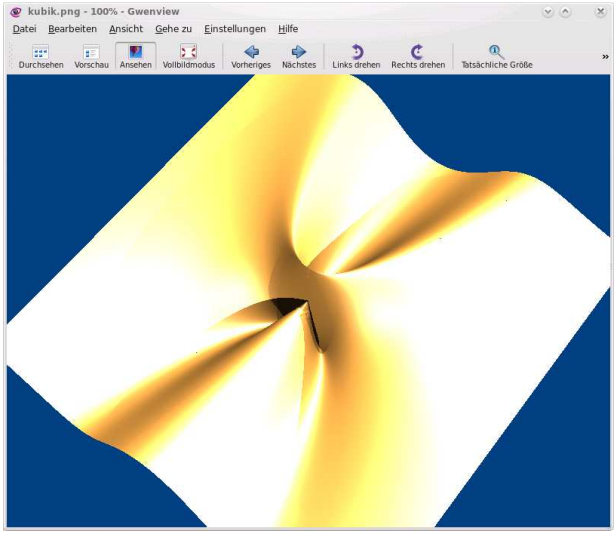
In der Povray-Dokumentation finden wir dieses Polynom:

$$x^2z + y^4z - 2xy^2,$$

wir betrachten die Nullstellenmenge:

```
camera { location <8,20,-10>*.7 look_at x*.01 angle 35 }
light_source { <100,200,20> 1 }
background { rgb <0,.25,.5> }

poly
{ 5,
  <0,0,0,0,0,
   0,0,0,0,0,
   0,0,0,0,0,
   0,0,0,1,0,
   0,0,0,0,0,
   -2,0,0,0,0,
   0,0,0,0,0,
   0,1,0,0,0,
```

```
   0,0,0,0,0,
   0,0,0,0,0,
   0,0,0,0,0,0>
  clipped_by { box { <-4,-4,-1><4,4,1> } }
  bounded_by { clipped_by }
  pigment { rgb <1,.7,.3> } finish { specular .5 }
  rotate <0,90,-90>
}
```



7.2 Isosurfaces

Dies sind Flächen, die durch eine Gleichung $f(x, y, z) = 0$ gegeben sind; die Funktion f kann man selbst definieren (Schlüsselwort `function`), z.B.

```
isosurface
{
  function{ sqrt(pow(x,2)+pow(y,2)) + z - 2}
  contained_by{ box{ -2, 2}}
}
```

Funktionen kann man auch global deklarieren:

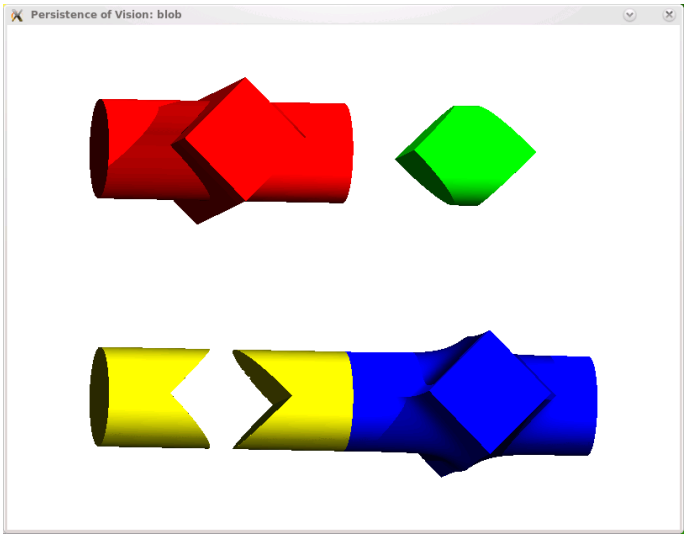
```
#declare factorial = function(n)
  {
    #local i = 2;
    #local w = 1;
    #while (i <= n)
       #local w = w * i;
    #end
    w
  }
```

Der Aufruf erfolgt so:

```
#declare a = factorial(5);
```

Man kann Vereinigungen, Durchschnitt, Differenzen und geglättete Übergänge herstellen:

```
#declare f1 = function{sqrt(pow(y,2) + pow(z,2))-0.8};
#declare f2 = function{abs(x)+abs(y)-1};

isosurface
{
  function{ min(f1(x,y,z), f2(x,y,z))}
  contained_by{box{-2,2}}
  pigment{Red}
  translate <-2,2,0>
}

isosurface
{
  function{ max(f1(x,y,z), f2(x,y,z))}
  contained_by{box{-2,2}}
  max_gradient 4
  pigment{Green}
  translate <2,2,0>
}

isosurface
{
  function{ max(f1(x,y,z), -f2(x,y,z))}
  contained_by{box{-2,2}}
  pigment{Yellow}
  translate <-2,-2,0>
}
#declare schwelle = 0.01;
isosurface
{
  function{
  (1+schwelle)-pow(schwelle,f1(x,y,z))-pow(schwelle, f2(x,y,z))}
  max_gradient 4
  contained_by{box{-2,2}}
  pigment{Blue}
  translate <2,-2,0>
}
```



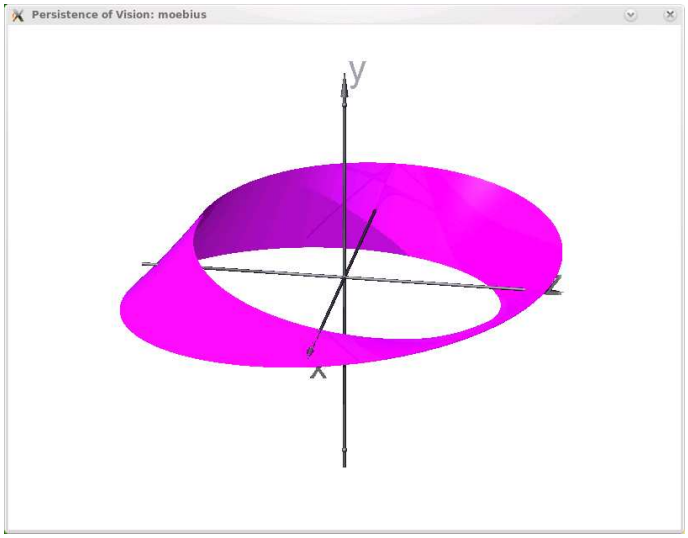
7.3 Parameterdarstellung

Manche Flächen können durch eine Parameterdarstellung beschrieben werden:

$$x = f(u, v); \qquad y = g(u, v); \qquad z = h(u, v).$$

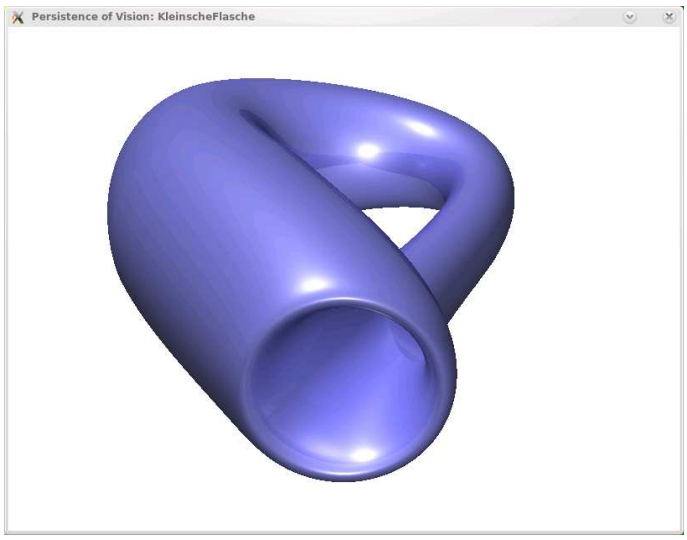
Für die Parameter sind die reservierten Bezeichner `u,v` zu verwenden. Dies geht z.B. so (A. Filler, Möbiusband):

```
parametric {
  function { cos(u) + v * cos(u/2) * cos(u) }, // Funktion x(u,v)
  function { v * sin(u/2) },                   // Funktion y(u,v)
  function { sin(u) + v * cos(u/2) * sin(u) }  // Funktion z(u,v)
  <0,-0.25>, <2*pi, 0.25>              // Intervalle fuer u und v
  contained_by { box { <-1,-1,-1>*4, <1,1,1>*4 } } // Begrenzungsquader
  max_gradient 3    // Maximaler Anstieg: groessere Werte fuehren zu
                    // laengeren Renderzeiten
  accuracy 0.003    // Genauigkeit/Feinheit der Berechnung: kleinere
                    // Werte fuehren zu hoeherer Qualitaetaet der
                    // Darstellung und laengeren Renderzeiten.
  precompute 18 x,y,z    // Vorausberechnung der Geometriedaten
  texture{ pigment { rgb<0.7,0,0.8>}
    finish{ ambient 0.5  diffuse 0.5  phong 0.4  reflection 0.05} }
 }
```



Und hier ist die Kleinsche Flasche; auf meinem Rechner brauchte es 6 Minuten zur Erzeugung, auf dem Vorgängermodell hätte es zehnmal länger gedauert.

```
parametric {
  function { 3*cos(u)*(1+sin(u)) - (2-cos(u))*cos(v) }
  function { 8*sin(u)}
  function { sin(v)*(2-cos(u)) }
  <pi,0>,<2*pi,2*pi>                  // Intervall fuer u und v
  contained_by { box { <-12,-12,-12>, <12,12,12> } }
  max_gradient 4   accuracy 0.003     // Berechnungsparameter
  precompute 18 x,y,z                 // Berechnungsparameter
  texture{ pigment{rgb<0.3,0.3,0.6>}
    finish{ ambient 0.4  diffuse 0.75  phong 0.4 reflection 0.1
    specular 0.2 }    }
  rotate<90,0,0>
  no_shadow
}
```

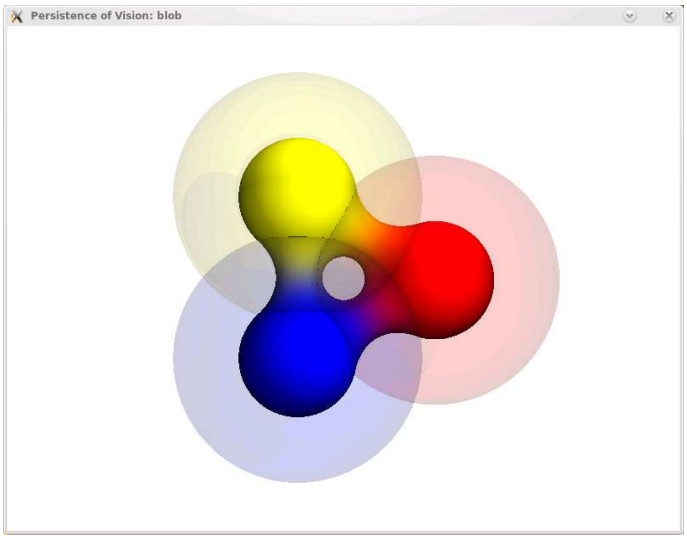


Das zweite Objekt wurde vom Glasbläser aus dem Nachbardorf hergestellt.

7.4 Kleckse

Aus Kugeln und Zylindern kann man Objekte formen, die nur aus Punkten bestehen, die einen Mindestabstand vom jeweiligen Zentrum haben, besser: auf die eine Mindest-Anziehungskraft `threshold` vom Zentrum aus wirkt. Die Anziehungskraft des Zentrums wird als letzter Parameter übergeben, auf der Oberfläche ist die Anziehungskraft Null.
Im folgenden Beispiel sehen wir das Blob-Objekt, das aus den drei durchsichtigen Kugeln entsteht:

```
sphere { <.75, 0, 0>, 1 pigment{color rgbt<1,0,0,0.9> } scale 2}
sphere { <-.375, .64952, 0>, 1 pigment{color rgbt<1,1,0,0.9>} scale 2}
sphere { <-.375, -.64952, 0>, 1 pigment{color rgbt<0,0,1,0.9>} scale 2}

blob {
threshold 0.6
sphere { <.75, 0, 0>, 1, 1 pigment{color rgb<1,0,0>}}
sphere { <-.375, .64952, 0>, 1, 1 pigment{color rgb<1,1,0>}}
sphere { <-.375, -.64952, 0>, 1, 1 pigment{color rgb<0,0,1>}}
scale 2
}
```



In der Povray-Referenz heißt es:
If you have a single blob component then the surface you see will just look like the object used, i.e. a sphere or a cylinder, with the surface being somewhere inside the surface specified for the component. The exact surface location can be determined from the blob equation listed below (you will probably never need to know this, blobs are more for visual appeal than for exact modeling). For the more mathematically minded, here's the formula used internally by POV-Ray to create blobs. You do not need to understand this to use blobs. The density of the blob field of a single component is:

$$\text{density} = \text{strength} \times (1 - (\frac{distance}{radius})^2)^2$$

where distance is the distance of a given point from the spherical blob's center or cylinder blob's axis. This formula has the nice property that it is exactly equal to the strength parameter at the center of the component and drops off to exactly 0 at a distance from the center of the component that is equal to the radius value. The density formula for more than one blob component is just the sum of the individual component densities.

8 Programme

Hier werden einige Beispiele vorgestellt, die sicher hilfreich sind.
Die eingeführten Methoden (Makros) werden im Folgenden benutzt.

8.1 Matrizen

Rechnen mit Matrizen, die Kommentare zeigen die Aktionen an. Es wird vorausgesetzt, daß die Vektoren $z1, z2, z3$ richtig belegt sind (siehe oben).

```
/** Matrizen
Hubert Grassmann
Januar 2009 */

/** druckt n an Stelle (i,j)*/
#macro print(n, i, j)
  text{ttf "timrom.ttf" str(n 0,2)  0.1,0 pigment{Red}
  matrix  < z1.x,z1.y,z1.z,
            z2.x,z2.y,z2.z,
            z3.x,z3.y,z3.z,
            0,0,0 >
  translate 3.3*<j,-i,0> scale 0.3
}
#end

/** zeigt Matrix a, um tr verschoben */
#macro zeig(a,tr)
 #local m = dimension_size(a,1);
 #local n = dimension_size(a,2);
 #local i = 0;
 union
 {
   #while (i < m)
     #local j = 0;
     #while (j < n)
       print(a[i][j], i, j)
       #local j = j+1;
     #end
     #local i = i+1;
   #end
```

```
   translate tr
 }
#end

/** Transponierte von a*/
#macro transp(a)
 #local m = dimension_size(a,1);
 #local n = dimension_size(a,2);
 #local i = 0;
 #local at = array[n][m];
 #while (i < m)
   #local j = 0;
   #while (j < n)
     #local at[j][i]= a[i][j];
     #local j = j+1;
   #end
   #local i = i+1;
 #end
 at
#end

/** Produkt von a und b */
#macro mult(a, b)
 #local m = dimension_size(a,1);
 #local n = dimension_size(a,2);
 #local l = dimension_size(b,2);
 #local c = array[m][l];
 #local i = 0;
 #while (i < m)
   #local j = 0;
   #while (j < l)
     #local s = 0;
     #local k = 0;
     #while (k < n)
       #local s = s + a[i][k]*b[k][j];
       #local k = k+1;
     #end
     #local c[i][j] = s;
     #local j = j+1;
   #end
   #local i = i+1;
 #end
 c
#end

//Determinante von a
#macro det(a)
  (a[0][0]*a[1][1] - a[0][1]*a[1][0])
```

```
#end

// Inverse der 2x2-Matrix a
#macro inv(a)
  #local d= det(a);
  #local b = array[2][2];
  #local b[0][0] =  a[1][1]/d;
  #local b[0][1] = -a[0][1]/d;
  #local b[1][0] = -a[1][0]/d;
  #local b[1][1] =  a[0][0]/d;
  b
#end

//Moore-Penrose-Inverse einer 3x2-Matrix
#macro mpi(a)
  #local at = transp(a)
  #local m =  mult(at,a)
  #local in = inv(m)
  #local ergeb =mult(in, at)
  ergeb
#end

/**Schnittpunkt der Geraden (A,B) und (C,D)*/
#macro schnitt(A,B,C,D)
  #local a = B-A;
  #local b = D-C;
  #local mm = array[3][2]
  {
    {a.x, -b.x},
    {a.y, -b.y},
    {a.z, -b.z}
  }
  #local r = array[3][1]
  {
    {C.x - A.x},
    {C.y - A.y},
    {C.z - A.z}
  }
  #local mp = mpi(mm);
  #local e =  mult(mp,r);
  #local E = A + e[0][0]*a;
  E
#end

/*Vektor als Matrix*/
#macro v2a(w)
  #local a = array[3][1]
  {{w.x},{w.y},{w.z}};
```

```
  a
#end
```

Noch ein Wort zur Moore-Penrose-Inversen einer Matrix. Zu jeder Matrix A gibt es eine eindeutig bestimmte Matrix X, für die gilt

$$AXA = A, \quad XAX = X, \quad (AX)^T = AX, \quad (XA)^T = XA,$$

sie wird meist mit A^- bezeichnet.

Wenn A regulär ist, so ist natürlich $A^- = A^{-1}$, wenn A^TA regulär ist (wenn also A vollen Spaltenrang hat), so ist $A^- = (A^TA)^{-1}A^T$; die obigen Eigenschaften sind leicht zu verifizieren. Für 3×2-Matrizen wurde dies im Makro `mpi` genutzt.

Sei nun A eine singuläre 3×3-Matrix.
Penrose zeigte, daß das Minimalpolynom der Matrix A^TA die Null nur als einfache Nullstelle besitzt und daher in der Form

$$m_{A^TA}(x) = g(x)x^2 - x$$

dargestellt werden kann. Dann ist $A^- = g(A^TA)A^T$ die Moore-Penrose-Inverse von A. Um diese zu berechnen, setzten wir zur Abkürzung $B = A^TA$. Sei

$$c_B(x) = x^3 - \mathtt{Sp}(B)x^2 + (b_{11}b_{22} - b_{12}b_{21} + b_{11}b_{33} - b_{13}b_{31} + b_{22}b_{33} - b_{23}b_{32})x$$

deren charakteristisches Polynom, wir bezeichnen den Koeffizienten von x mit H. Wenn $H \neq 0$ ist, so ist $c_B(x)$ bereits das Minimalpolynom, demnach ist

$$g(x) = -\frac{1}{H}(x - \mathtt{Sp}(B)),$$

also

$$A^- = -\frac{1}{H}(A^TA - \mathtt{Sp}(A^TA))A^T.$$

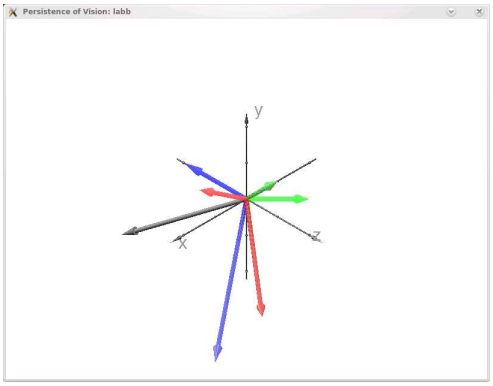
Wenn $H = 0$ ist, so ist $A^- = \frac{1}{Sp(A^TA)}A^T$. Diese Rechnungen kann man mit Hilfe der obigen Makros leicht durchführen.

8.2 Lineare Abbildungen

Hier ist ein vollständiges Programm, das die Wirkung einer linearen Abbildung zeigt.

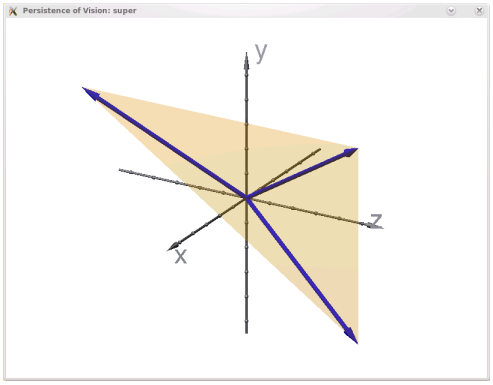
```
#include "colors.inc"
#include "glass.inc"
#declare intervall = 3;
#declare winkel = 30;
#declare hoehe = 0;
#declare KameraManuell=1;
#declare KamPos=<1,1,1>*10;
#include "anageoL2.inc"
#include "Matrizen.inc"
```

```
#version 3.5;

global_settings { max_trace_level 20 }
background { color rgb <1, 1, 1> }

koordinatensystem(2)

#declare a = <1,1,-1>;
#declare b = <1.5,1,0>;
#declare c = <0,1,2>;
#declare n = <1,-2,-3>;
#declare A = v2a(a);
#declare B = v2a(b);
#declare C = v2a(c);
#declare N = v2a(n);

// Vektoren vor der Transformation (hell)
ortsvektor(a, blau_transp)
ortsvektor(b, rot_transp)
ortsvektor(c, gruen_transp)
ortsvektor(n, pigment{color rgb<0.5,0.5,0.5>})
// Vektoren nach der Transformation (dunkler)
#declare T = array[3][3]
{
  {1,2,-1},
  {1,-1,1},
  {2,1,0}
}

#declare TA = array[3][1];
#declare TB = array[3][1];
#declare TC = array[3][1];
#declare TN = array[3][1];

#declare TA=mult(T,A);
#declare TB=mult(T,B);
#declare TC=mult(T,C);
#declare TN=mult(T,N);

ortsvektor(<TA[0][0],TA[1][0],TA[2][0]>, blau_glanz)
ortsvektor(<TB[0][0],TB[1][0],TB[2][0]>, rot_glanz)
ortsvektor(<TC[0][0],TC[1][0],TC[2][0]>, gruen_glanz)
ortsvektor(<TN[0][0],TN[1][0],TN[2][0]>, pigment{Black})
```



8.3 Dreiecke

Mit dem Kommando `triangle{p1, p2, p3 pigment{ color rgbt <1,0.7,0,0.7>}}` kann man eine Dreiecksfläche herstellen:



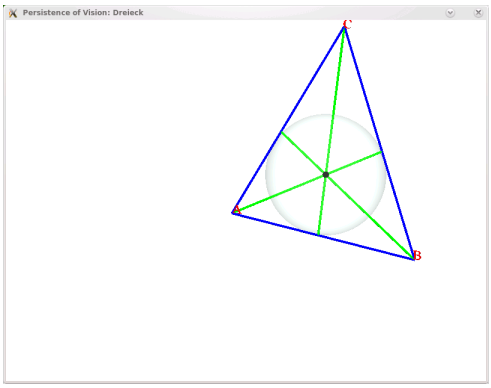
Uns geht es hier um die Linien im Dreieck.

```
/** Dreiecke und so
hubert@grassmann.info
11.9.08 */
#include "colors.inc"
#include "glass.inc"
#declare intervall = 5;
#declare winkel = 30;
#declare hoehe = 5;

#declare A = <-1,-1,2>;
#declare B = <3,3,3>;
#declare C = <1,3,4>;

#declare KameraManuell = 1;
#declare KamPos = vcross(B-A,A-C);

#include "anageoL2.inc"
#include "transforms.inc"
#include "Matrizen.inc"
```

```
#version 3.5;

global_settings { max_trace_level 20 }

/* A.Filler
#declare Schwerpunkt = A/3 + B/3 + C/3;
#declare Kamerapos = Schwerpunkt - 5*vcross(a,b);
camera {
   location Kamerapos
   angle 12
   look_at Schwerpunkt}
*/

background { color rgb <1, 1, 1> }
light_source { <20, 20, -50>  color rgb <1, 1, 1> }

#macro dreieck(A,B,C)
union
{
  strecke(A, B pigment{color rgb<0,0,1>})
  strecke(B, C pigment{color rgb<0,0,1>})
  strecke(C, A pigment{color rgb<0,0,1>})
text{ttf "timrom.ttf" "A"  0.1,0 pigment{Red} translate 3.3*A scale 0.3}
text{ttf "timrom.ttf" "B"  0.1,0 pigment{Red} translate 3.3*B scale 0.3}
text{ttf "timrom.ttf" "C"  0.1,0 pigment{Red} translate 3.3*C scale 0.3}
}
#end

#macro seitenhalb(A,B,C)
  strecke(A,(B+C)/2 pigment{color rgb<0,1,0>})
#end

#declare a = B-C;
#declare b = C-A;
#declare c = A-B;
#declare la = vlength(a);
#declare lb = vlength(b);
#declare lc = vlength(c);

#macro mitten(A,B,C)
union
{
  seitenhalb(A,B,C)
  seitenhalb(B,A,C)
  seitenhalb(C,A,B)
  #declare E = schnitt(A, (B+C)/2, B, (C+A)/2);
  punkt(E, pigment{Black})
}
```

```
#end

#macro hoehen(A,B,C)
  #local lotc = A + vdot(b,vnormalize(c))*vnormalize(c);
  #local lota = B + vdot(c,vnormalize(a))*vnormalize(a);
  #local lotb = C + vdot(a,vnormalize(b))*vnormalize(b);
  union
  {
  strecke(C,lotc,pigment{color rgb<1,1,0>})
  strecke(A,lota,pigment{color rgb<1,1,0>})
  strecke(B,lotb,pigment{color rgb<1,1,0>})
  }
#end

#macro mittelsenkrechte(A,B,C)
  #local lotc = A + vdot(b,vnormalize(c))*vnormalize(c);
  #local lota = B + vdot(c,vnormalize(a))*vnormalize(a);
  #local lotb = C + vdot(a,vnormalize(b))*vnormalize(b);
  #local ma = (B+C)/2;
  #local mb = (C+A)/2;
  #local mc = (A+B)/2;
  union
  {
  strecke(ma,A-lota+ma,pigment{color rgb<1,1,0>})
  strecke(mb,B-lotb+mb,pigment{color rgb<1,1,0>})
  strecke(mc,C-lotc+mc,pigment{color rgb<1,1,0>})
  #local E = schnitt(ma,A-lota+ma,mb,B-lotb+mb);
  punkt(E, pigment{Black})
  sphere{E, vlength(E-A) pigment {Col_Glass_General}}
  }
#end

#macro winkelhalbierende(A,B,C)
  #local ca = la/(lb+la); // Abschnitt auf c entspr. Seite a
  #local ab = lb/(lc+lb);
  #local bc = lc/(la+lc);
  #local wc = B + ca*c;
  #local wa = C + ab*a;
  #local wb = A + bc*b;
  union
  {
  strecke(C,wc, pigment{color Green})
  strecke(A,wa, pigment{color Green})
  strecke(B,wb, pigment{color Green})
  #local E = schnitt(A, wa, B, wb);
  #local lote = wa + vdot(E-wa,vnormalize(a))*vnormalize(a);
  punkt(E, pigment{Black})
  sphere{E, vlength(E-lote) pigment {Col_Glass_General}}
```

```
  }
#end

dreieck(A,B,C)
winkelhalbierende(A,B,C)
/*
object{winkelhalbierende(A,B,C) Axis_Rotate_Trans(<0,0,1>, 45)}
object{winkelhalbierende(A,B,C) Axis_Rotate_Trans(<0,0,1>, 90)}
*/
//mittelsenkrechte(A,B,C)
```



8.4 Sphärische Dreiecke

Als nächstes wollen wir uns mit der Geometrie auf der Oberfläche der Einheitskugel beschäftigen.
Ein sphärisches Dreieck ist durch drei Einheitsvektoren a, b, c gegeben. Wir wählen die Reihenfolge so, daß $\det(a\ b\ c) > 0$ ist. Die „Seiten“ des Dreiecks sind die Winkel A, B, C zwischen den Vektoren a, b, c, die „Winkel“ α, β, γ des Dreiecks sind die Winkel zwischen den Ebenen $L(a, b)$, $L(a, c)$, $L(a, c)$, also die Winkel zwischen deren Normalenvektoren. Da die Vektoren den Betrag 1 haben, gelten folgende Beziehungen:

$$\cos A = \langle b, c\rangle \quad \cos B = \langle a, c\rangle \quad \cos C = \langle b, a\rangle$$

Weiter ist

$$|b \times c|^2 = \langle b \times c, b \times c\rangle = \langle b, b\rangle\langle c, c\rangle - \langle c, b\rangle\langle b, c\rangle = 1 - \langle c, b\rangle^2 = 1 - \cos^2 A = \sin^2 A,$$

also

$$\sin A = |a \times c| \quad \sin B = |a \times c| \quad \sin C = |a \times b|$$

$$\cos\alpha = \frac{\langle a \times c, a \times b\rangle}{|a \times c|\,|a \times b|} \quad \cos\beta = \frac{\langle b \times a, b \times c\rangle}{|b \times a|\,|b \times c|} \quad \cos\gamma = \frac{\langle c \times b, c \times a\rangle}{|c \times b|\,|c \times a|}$$

Aus den Formeln für das Vektorprodukt folgt

$$\begin{aligned} |(a \times b) \times (a \times c)| &= |\det(a\ b\ c)|\,|a| \\ &= \det(a\ b\ c) \end{aligned}$$

$$\begin{aligned} &= |a \times b|\,|a \times c| \sin\alpha \\ &= \sin C \cdot \sin B \cdot \sin\alpha. \end{aligned}$$

Daraus folgt der spärische Sinussatz:

Satz 8.1

$$\frac{\sin\alpha}{\sin A} = \frac{\det(a\ b\ c)}{\sin A \sin B \sin C} = \frac{\sin\beta}{\sin B} = \frac{\sin\gamma}{\sin C}.\square$$

Wenn die Seiten klein sind, so können wir sie die Sinuswerte durch die Argumente ersetzen und erhalten den ebenen Sinussatz.

Wir erhalten zwei Cosinussätze:

Satz 8.2 *1.* $\cos A = \cos B \cos C + \sin B \sin C \cos\alpha$,
2. $\sin C \cos B = \sin B \cos C \cos\alpha + \sin A \cos\beta$.

Aus der ersten Formel geht hervor, daß man die Winkel aus den Seiten berechnen kann.

Das Dreieck, dessen definierenden Vektoren senkrecht auf den Seiten des durch a, b, c gegebenen Dreiecks stehen, wird das Polardreieck genannt, es hat die Ecken

$$a' = \frac{b \times c}{|b \times c|}, \quad b' = \frac{c \times a}{|c \times a|}, \quad c' = \frac{a \times b}{|a \times b|}$$

die Seiten A', B', C' und die Winkel α', β', γ'.

Satz 8.3 (Viétà-Formeln)

$$\cos A' = -\cos\alpha,\ \cos\alpha' = -\cos A.$$

Beweis:

$$\cos A' = \langle b', c' \rangle = \frac{\langle c \times a, a \times b \rangle}{|c \times a|\,|a \times b|}.\square$$

Als Folgerung erhalten wir den polaren Cosinussatz:

Folgerung 8.4

$$-\cos\alpha = \cos\beta \cos\gamma + \sin\beta \sin\gamma \cos A.\square$$

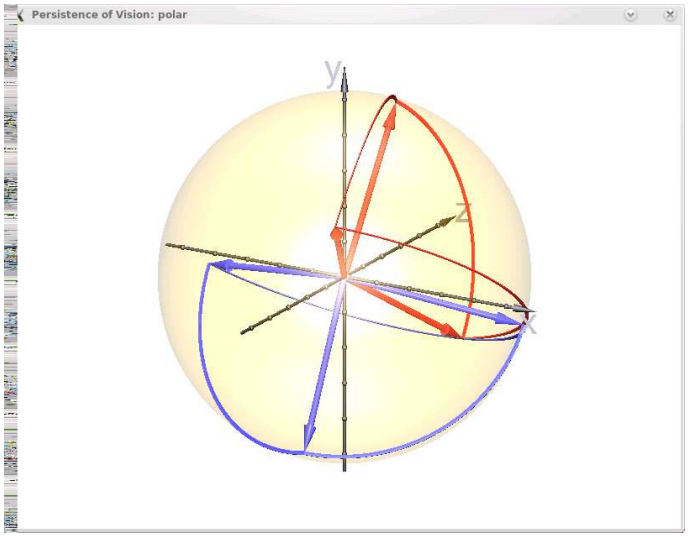
Das bedeutet, daß die Seiten des Dreiecks aus den Winkeln berechnet werden können. Es gibt also keine ähnlichen Dreiecke, die nicht kongruent sind.

Wenn man die geografischen Längen L_i und Breiten B_i zweier Orte kennt, so kann man mit dem ersten Cosinussatz deren Entfernung berechnen. Man betrachtet das Dreieck, das von beiden Orten und dem Nordpol gebildet wird, dessen zwei Seiten sind gleich $\pi/2 - B_i$ und der der gesuchten Seite gegenüberliegende Winkel ist gleich $L_1 - L_2$.
Beispiel: Paris: $(2,3°; 48,8°)$, Berlin: $(13,4°; 52,5°)$, damit berechnet man $\cos A = 0,99...$, also $A = 7,939°$, bei einem Erdradius von 6378 km ergibt dies eine Entfernung von 884 km.

Nun wollen wir ein Dreieck und das dazu polare sehen:; der Schlüssel ist das Makro `bogen`: Es wird die Differenz zweier Kugeln, also eine Kugelschale erzeugt, diese wird mit einer Ebene geschnitten, der entstehende Großkreis wird mit einem Kegel geschnitten, es entsteht eine Dreiecksseite.

```
// A. Filler, 2003, H. Grassmann 8.12.06
//------------------------------------------------------
#version 3.5;          // benoetigte POV-Ray-Version
#declare intervall = 5;// von der Kamera erfasster Bildausschnitt
#declare winkel = -60; // Winkel der Kamera zur x-Achse (in Grad)
#declare hoehe =   4 ; // Hoehe (relativ) der Kamera ueber der x-z-Ebene
#declare KameraManuell = 0;

#include "anageoL2.inc"
background {White}             // Farbe des Hintergrundes

koordinatensystem(intervall)
// drei Punkte
#declare a=vnormalize(<2,3,-4>) * 5;
#declare b=vnormalize(<1,-2,3>) * 5;
#declare c=vnormalize(<0.0, 0.83205, 0.55470>)*5;
// die dazu polaren Punkte
#declare a1= vnormalize(vcross(b,c)) *5;
#declare b1= vnormalize(vcross(a,c)) *5;
#declare c1= vnormalize(vcross(a,b)) *5;

sphere{ <0,0,0>, 5
        texture { pigment{ color rgbt<1,0.7,0,0.9>}
                  finish { diffuse 1 phong 0.2 phong_size 8}
                } // end of texture
      } // end of sphere -------------

  verbindungsvektor(<0,0,0>,a, rot_matt)
  verbindungsvektor(<0,0,0>,b, rot_matt)
  verbindungsvektor(<0,0,0>,c, rot_matt)

  verbindungsvektor(<0,0,0>,a1, blau_matt)
  verbindungsvektor(<0,0,0>,b1, blau_matt)
  verbindungsvektor(<0,0,0>,c1, blau_matt)

/* verbindet 2 Punkte auf der Kugeloberflaeche*/
#macro bogen(punkt1,punkt2,textur)
object
{
  intersection
  {
    intersection
    {
      difference
      {
        sphere{ <0,0,0>, 5.05}
        sphere{ <0,0,0>, 4.95}
      }
```

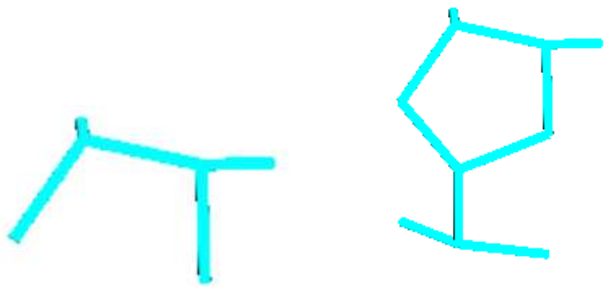
```
      object{ebenepar(<0,0,0>,punkt1,punkt2, textur) scale 3}
    }
    cone {<0,0,0> 0
     (punkt1+punkt2)/2 tan(1/2*acos(vdot(punkt1,punkt2)/
       (vlength(punkt1)*vlength(punkt2))))
       *vlength((punkt1+punkt2)/2)  scale 20}
  }
  texture{textur}  no_shadow
}
#end

bogen(a1,b1,blau_matt)
bogen(b1,c1,blau_matt)
bogen(a1,c1,blau_matt)

bogen(a,b,rot_matt)
bogen(b,c,rot_matt)
bogen(a,c,rot_matt)
```



8.5 Platonische Körper

Die platonischem Körper sind diejenigen, deren Oberfläche aus regelmäßigen 3-, 4- bzw. 5-Ecken besteht: Tetraeder, Hexaeder (Würfel), Oktaeder, Dodekaeder, Ikosameter. Wie man sie konstruieren kann, findet man z.B. bei
`http://www.mathematische-basteleien.de/` von Jürgen Köller.

Als Beispiel konstruieren wir eine Dodekaeder, es wird von 12 regelmäßigen Fünfecken begrenzt:
Wir stellen uns ein Walmdach vor, das eine quadratische Grundfläche hat und deren Dachflächen zur Horizontalen jeweils einen Winkel von 45° einschließen. Wir drehen nun dieses nach rechts und nach vorn, so daß die große Dachfläche an die kleine angrenzt:



Das machen wir nun in alle Richtungen; so entsteht das Dodekaeder.

Johannes Kepler, der die Radien der Bahnen der damals 6 bekannten Planeten berechnete, vermutete, daß diese Radien zu den Radien der ineinandergeschachtelten In- und Umkugeln dieser Körper (in der richtigen Reihenfolge) proportional seien, was bei der von ihm erzielten Meßgenauigkeit nicht abwegig ist.

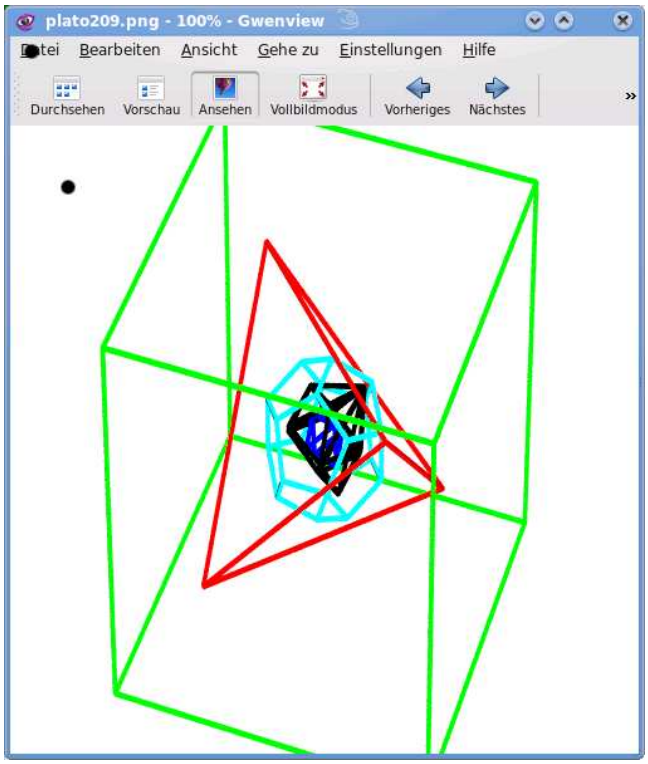
```
#include "colors.inc"
#include "glass.inc"
#declare intervall = 2;
#declare winkel = 10;
#declare hoehe = 5;
#include "vorlage.inc"
#include "anageoL2.inc"
#include "transforms.inc"
#version 3.5;
global_settings { max_trace_level 20 }

#declare Glas1 = material {
texture { pigment { rgbf<0.97, 0.97, 0.97, 0.95> }
          finish { specular 0.07  roughness 0.001  ambient 0  diffuse 0
          reflection 0.02 }
        }
interior {ior 1}
};

#declare Glas2 = material {
texture { pigment { rgbf<0.95, 0.95, 1.0, 0.95> }
          finish { specular 0.07  roughness 0.001  ambient 0  diffuse 0
          reflection 0.02 }
        }
interior {ior 1.5}
};

#macro hohlkugel(MP, Radius, Mat)
difference { sphere{MP, Radius }
             sphere{MP, Radius - 0.01 }
             material{Mat}
```

```
            hollow on
            no_shadow
          }
#end

camera {
   location <7,17,-20>
   angle 10
   look_at<0,0,0>}
background { color rgb <1, 1, 1> }
light_source { <20, 20, -50>  color rgb <1, 1, 1> }

#declare kante = 2.5;  //Kantenlaenge

//Tetraeder
#macro tet(kante, inkugel, umkugel)
#declare a = kante;
union
{
strecke(<a/2,a/2,a/2>, <-a/2,a/2,-a/2>, pigment{color rgb<1,0,0>})//hio
strecke(<-a/2,a/2,-a/2>, <a/2,-a/2,-a/2>, pigment{color rgb<1,0,0>})//liv
strecke(<a/2,-a/2,-a/2>, <a/2,a/2,a/2>, pigment{color rgb<1,0,0>})//rev
strecke(<a/2,-a/2,-a/2>, <-a/2,-a/2,a/2>, pigment{color rgb<1,0,0>})//un
strecke(<-a/2,-a/2,a/2>, <-a/2,a/2,-a/2>, pigment{color rgb<1,0,0>})
strecke(<a/2,a/2,a/2>, <-a/2,-a/2,a/2>, pigment{color rgb<1,0,0>})
#if (umkugel)
hohlkugel(<0,0,0>, k4*R4, material{Glas1})
#end
#if (inkugel)
hohlkugel(<0,0,0>, k4*r4, material{Glas2})
#end
}
#end

//Wuerfel
#macro cub(kante,inkugel,umkugel)
#declare a = kante;   // Laenge der Kanten
union
{
strecke(<a/2,a/2,a/2>, <-a/2,a/2,a/2>, pigment{color rgb<0,1,0>})
strecke(<-a/2,a/2,a/2>, <-a/2,a/2,-a/2>, pigment{color rgb<0,1,0>})
strecke(<-a/2,a/2,-a/2>, <a/2,a/2,-a/2>, pigment{color rgb<0,1,0>})
strecke(<a/2,a/2,-a/2>, <a/2,a/2,a/2>, pigment{color rgb<0,1,0>})
strecke(<a/2,-a/2,a/2>, <-a/2,-a/2,a/2>, pigment{color rgb<0,1,0>})
strecke(<-a/2,-a/2,a/2>, <-a/2,-a/2,-a/2>, pigment{color rgb<0,1,0>})
strecke(<-a/2,-a/2,-a/2>, <a/2,-a/2,-a/2>, pigment{color rgb<0,1,0>})
strecke(<a/2,-a/2,-a/2>, <a/2,-a/2,a/2>, pigment{color rgb<0,1,0>})
strecke(<a/2,a/2,a/2>, <a/2,-a/2,a/2>, pigment{color rgb<0,1,0>})
```

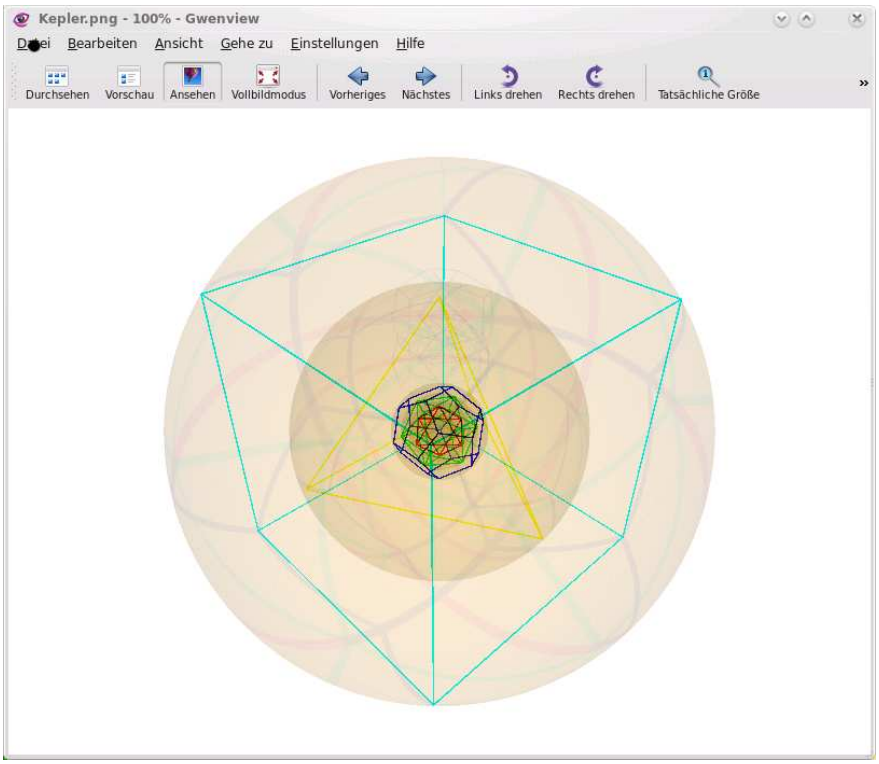
```
strecke(<-a/2,a/2,a/2>, <-a/2,-a/2,a/2>, pigment{color rgb<0,1,0>})
strecke(<-a/2,a/2,-a/2>, <-a/2,-a/2,-a/2>, pigment{color rgb<0,1,0>})
strecke(<a/2,a/2,-a/2>, <a/2,-a/2,-a/2>, pigment{color rgb<0,1,0>})
#if (umkugel)
hohlkugel(<0,0,0>, k6*R6, material{Glas1})
#end
#if (inkugel)
hohlkugel(<0,0,0>, k6*r6, material{Glas2})
#end
}
#end

//Oktaeder
#macro okt(kante, inkugel, umkugel)
#declare a = kante;   // Laenge der Kanten
union
{
strecke(<a/2,0,a/2>, <-a/2,0,a/2>, pigment{color rgb<0,0,1>})//re
strecke(<-a/2,0,a/2>, <-a/2,0,-a/2>, pigment{color rgb<0,0,1>})//hi
strecke(<-a/2,0,-a/2>, <a/2,0,-a/2>, pigment{color rgb<0,0,1>})//li
strecke(<a/2,0,-a/2>, <a/2,0,a/2>, pigment{color rgb<0,0,1>}) //vo
strecke(<a/2,0,a/2>, <0,-a/sqrt(2),0>, pigment{color rgb<0,0,1>})
strecke(<-a/2,0,a/2>, <0,-a/sqrt(2),0>, pigment{color rgb<0,0,1>})
strecke(<-a/2,0,-a/2>, <0,-a/sqrt(2),0>, pigment{color rgb<0,0,1>})
strecke(<a/2,0,-a/2>, <0,-a/sqrt(2),0>, pigment{color rgb<0,0,1>})
strecke(<a/2,0,a/2>, <0,a/sqrt(2),0>, pigment{color rgb<0,0,1>})
strecke(<-a/2,0,a/2>, <0,a/sqrt(2),>, pigment{color rgb<0,0,1>})
strecke(<-a/2,0,-a/2>, <0,a/sqrt(2),>, pigment{color rgb<0,0,1>})
strecke(<a/2,0,-a/2>, <0,a/sqrt(2),0>, pigment{color rgb<0,0,1>})
#if (umkugel)
hohlkugel(<0,0,0>, k8*R8, material{Glas1})
#end
#if (inkugel)
hohlkugel(<0,0,0>, k8*r8, material{Glas2})
#end
}
#end

// Dodekaeder (Kantelaenge = kante)auf Wuerfel mit Seitenlaenge d
#macro dod(kante, inkugel, umkugel)
#declare a = kante;   // Laenge der Kanten
#declare d = 0.5*(1+sqrt(5))*a;
#declare H = 0.5*a; // Hoehe des Walmdachs
#declare dach = union
{
strecke(<d/2,d/2,d/2>, <a/2,d/2+H,0>, pigment{color rgb<0,1,1>})
strecke(<d/2,d/2,-d/2>, <a/2,d/2+H,0>, pigment{color rgb<0,1,1>})
strecke(<-d/2,d/2,d/2>, <-a/2,d/2+H,0>, pigment{color rgb<0,1,1>})
```

```
strecke(<-d/2,d/2,-d/2>, <-a/2,d/2+H,0>, pigment{color rgb<0,1,1>})
strecke(<-a/2,d/2+H,0>, <a/2,d/2+H,0>, pigment{color rgb<0,1,1>})
};
union
{
object{dach}
object{dach rotate 180*z}
object{dach rotate 90*y rotate 90*z}
object{dach rotate 90*y rotate -90*z}
object{dach rotate 90*x rotate -90*z}
object{dach rotate -90*x rotate 90*z}
#if (umkugel)
hohlkugel(<0,0,0>, k12*R12, material{Glas1})
#end
#if (inkugel)
hohlkugel(<0,0,0>, k12*r12, material{Glas2})
#end
        }
#end

// Ikosaeder im Einheitswuerfel
#macro ikos(kante, inkugel, umkugel)
#declare a = (-1 + sqrt(5)) * kante;  // Laenge der Kanten
#declare unten = -1/2;//*kante;
#declare oben = -unten;
#declare lo =strecke(<cos(radians(72)),oben,sin(radians(72))> *k20,
 <cos(radians(36)),unten,sin(radians(36))>*k20,
 pigment{color rgb<0,0,0>})
#declare lu =strecke(<cos(radians(36)),oben,sin(radians(36))>*k20,
 <cos(radians(72)),unten,sin(radians(72))>*k20,
 pigment{color rgb<0,0,0>})
#declare so =strecke(<cos(radians(72)),unten,sin(radians(72))>*k20,
 <cos(radians(144)),unten,sin(radians(144))>*k20,
 pigment{color rgb<0,0,0>})
#declare su =strecke(<cos(radians(72)),oben,sin(radians(72))> *k20,
 <cos(radians(144)),oben,sin(radians(144))>*k20,
 pigment{color rgb<0,0,0>})
#declare qo = strecke(<0,R20,0>*k20, <cos(radians(36)),oben,
  sin(radians(36))>*k20,
 pigment{color rgb<0,0,0>})
#declare qu = strecke(<0,R20,0>*k20,<cos(radians(72)),unten,
  sin(radians(72))>*k20,
 pigment{color rgb<0,0,0>})
#declare i = 1;
union
{
#while (i < 6)
  object{lo rotate (36+72*i)*y }
```

```
  object{lu rotate 72*i*y }
  object{su rotate (36+72*i)*y }
  object{so rotate (72*i)*y }
  object{qu rotate 72*i*y }
  object{qo rotate 72*i*y }
  #declare i = i+1;
#end
#if (umkugel)
hohlkugel(<0,0,0>, k20*R20, material{Glas1})
#end
#if (inkugel)
hohlkugel(<0,0,0>, k20*r20, material{Glas2})
#end
}
#end

//In- und Umkugelradien
#declare r4 = kante/12 * sqrt(6)*sqrt(2);     //0.28867513459481287
#declare R4 = kante/4 * sqrt(6)*sqrt(2);      //0.8660254037844386
#declare r6 = kante/2;                        //0.5
#declare R6 = kante/2 * sqrt(3);              //0.8660254037844386
#declare r8 = kante/6*sqrt(6);                //0.40824829046386296
#declare R8 = kante/2 * sqrt(2);              //0.7071067811865476
#declare r12 = kante/20*sqrt(250+110*sqrt(5));//1.1135163644116068
#declare R12 = kante/4 * (sqrt(3)+sqrt(15));  //1.4012585384440737
#declare r20 = (-1+sqrt(5))*kante/12*sqrt(3)*(3+sqrt(5))/2; //0.9341723589627158
#declare R20 = (-1+sqrt(5))*kante/4*sqrt(10+2*sqrt(5))/2;  //1.1755705045849463

#declare f4 = 1/12 * sqrt(6)*sqrt(2);
#declare f6 = 1/2;
#declare f8 = 1/6*sqrt(6);
#declare f12 = 1/20*sqrt(250+110*sqrt(5));
#declare f20 = (-1+sqrt(5))/12*sqrt(3)*(3+sqrt(5)); // ??

// Kanten der Keplerkugeln
#declare k6 = kante;
#declare k4 = k6/f6*f4;
#declare k12 = k4/f4*f12/16;
#declare k20 = k12/f12*f20;
#declare k8 = k20/f20*f8*2;

object{cub(k6,0,0) Axis_Rotate_Trans(<0,1,0>, clock*10)}
object{tet(k4,0,0) Axis_Rotate_Trans(<0,0,1>, clock*10)}
object{dod(k12,0,0) Axis_Rotate_Trans(<1,0,0>, clock*10)}
object{ikos(k20,0,0) Axis_Rotate_Trans(<1,1,0>, clock*10)} //??
object{okt(k8,0,0) Axis_Rotate_Trans(<0,1,1>, clock*10)}
```



Die Informatik-Studenten Björn Möller und Aljoscha Peters haben folgendes Bild produziert:



8.6 Planeten

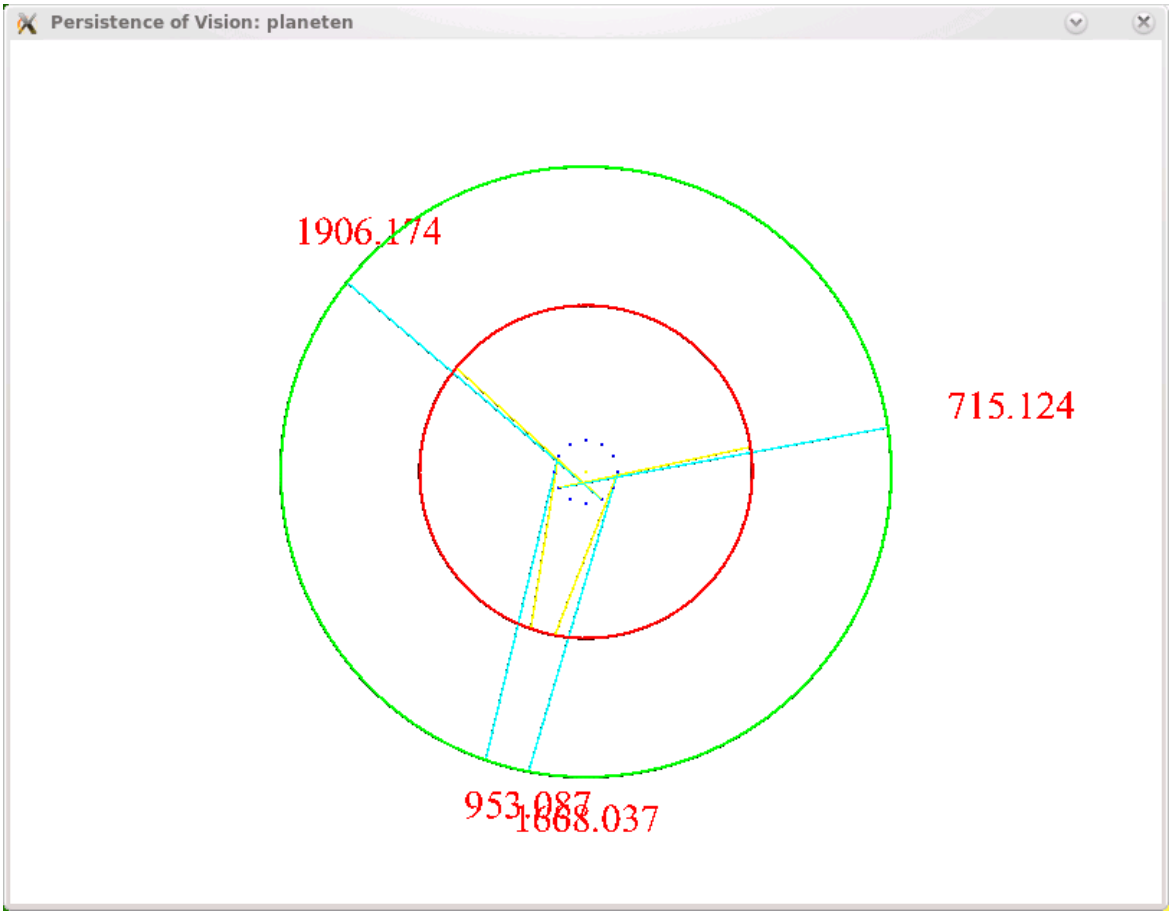
Wir beschränken uns zunächst auf Erde, Jupiter und Saturn; deren (3-fache) Konjunktion hat im Jahr 7 v.Chr. zum „Stern von Betlehem“ geführt, das hat Kepler ausgerechnet. Einen zweiten Beleg für diesen Zeitpunkt gibt die Bibel: „Es begab sich also zu der Zeit, da Kaiser Augustus ein Gebot ausgab, daß alle Welt sich schätzen ließe.“ Und das war tatsächlich im Jahre 7 v.Chr.

Und wir beschränken uns auf die (nur näherungsweise richtige) Annahme, daß die Planeten in *einer* Ebene rotieren. Wir lassen uns die Positionen im Monatsabstand zeigen; wenn der Winkel zwischen den Strahlen Erde/Jupiter und Erde/Saturn klein ist, zeichnen wir die Verbindungsstrecken und geben den aktuellen Monat und den Winkel als Dezimalzahl aus: Monat.Winkel.

```
/** Planeten
hubert@grassmann.info
10.2.09 */
#include "colors.inc"
#include "glass.inc"
#declare intervall = 10;
#declare winkel = 30;
#declare hoehe = 5;
#declare KameraManuell = 1;
#declare KamPos = <0,0,-15>;
#include "anageoL2.inc"
#include "transforms.inc"
#version 3.5;
global_settings { max_trace_level 20 }
background { color rgb <1, 1, 1> }
light_source { <20, 20, -50>  color rgb <1, 1, 1> }

#declare AE = 1;      //astronomische Einheit: Erdbahnradius
#declare ez = 1;      //Erde: Umlaufzeit
#declare jbr = 5.2;   //Jupiter: Bahn-Radius
#declare jz = 11.86;  //Jupiter: Umlaufzeit
#declare sbr = 9.54;  //Saturn: Bahnradius
#declare sz = 29.45;  //Saturn: Umlaufzeit
#declare anz = 2000;//12*60;
#declare winkel = 0;  //Winkel Erde/Saturn und Erde/Jupiter
#declare i = 0;

#macro punkt (P, textur)
sphere{<P.x,P.y,P.z>, intervall/200 texture{textur}}
#end

#macro strecke (P, Q , textur)  // aus anageoL2, aber duenner
#declare V=Q-P;
#declare vektorbetrag=sqrt(V.x*V.x+V.y*V.y+V.z*V.z);
#if ( V.x*V.x+V.y*V.y+V.z*V.z = 0 )
```

```
  #else
#if ( V.x*V.x+V.y*V.y = 0 )
  #declare Streckenzylinder=cylinder{<0,0,0> V intervall/300
              texture {textur}  no_shadow
  };
  object {Streckenzylinder  translate <P.x,P.y,P.z>}
#else
  #declare Streckenzylinder=cylinder{<0,0,0> <vektorbetrag, 0, 0>
  intervall/300 texture {textur}  no_shadow
  };
  #declare NX=vnormalize(V) ;
  #declare NY=vnormalize(vcross(NX,z)) ;
  #declare NZ=vnormalize(vcross(NX,NY)) ;
  object {Streckenzylinder
  matrix < NX.x , NX.y , NX.z ,
           NY.x , NY.y , NY.z ,
           NZ.x , NZ.y , NZ.z ,
           P.x  , P.y  , P.z   >
  }
#end
#end
#end

/** druckt n an der Stelle (i,j)*/
#macro print(n, i, j)
  text{ttf "timrom.ttf" str(n 0,3)  0.1,0 pigment{Red}
  translate <i,j,0> scale 1.2}
#end

// Position im i-ten Monat
#macro Erde(i) <cos(i*2*pi/12), sin(i*2*pi/12),0>
#end

#macro Jupiter(i)
<cos(i*2*pi/(12*jz)), sin(i*2*pi/(12*jz)),0>
#end

#macro Saturn(i)
<cos(i*2*pi/(12*sz)), sin(i*2*pi/(12*sz)),0>
#end

//Sonne
punkt(<0,0,0>, pigment{color Yellow})

#while (i < anz)
  punkt(Erde(i),
         pigment{color Blue})
  punkt(Jupiter(i)*jbr,
```

```
        pigment{color Red})
  punkt(Saturn(i)*sbr,
        pigment{color Green})

  #declare winkel = VAngleD(Jupiter(i) - Erde(i), Saturn(i)  - Erde(i));

  #if (abs(winkel) < 0.2)              // Konjunktion?
    strecke(Erde(i), Jupiter(i)*jbr  pigment{color rgb<1,1,0>})
    strecke(Erde(i), Saturn(i)*sbr   pigment{color rgb<0,1,1>})
    print(i+winkel,Saturn(i).x*sbr,Saturn(i).y*sbr,)
      //vor dem Komma: Anzahl der Monate; nach dem Komma: Winkel
  #end

#declare i = i+1;
#end
```



8.7 Räumliche Vierecke

Unter einem Simplex stellt man sich meist ein symmetrisches Gebilde vor, wir wollen unseren Gesichtskreis erweitern und auch unregelmäßige räumliche Vierecke betrachten. Wir sehen, daß jedes Viereck, das nicht in einer Ebene liegt, eine Umkugel besitzt: Wir wählen drei der Punkte aus, legen durch sie den Umkreis und betrachten die Kugel

mit demselben Radius und demselben Mittelpunkt. Wenn der vierte Punkt außerhalb der Kugel liegt, so verschieben wir den Mittelpunkt in Richtung des vierten Punkts und blasen die Kugel gleichzeitig auf, so daß die drei Punkte (und ihr Umkreis) auf der Kugel verbleiben, bis die Kugel den vierten Punkt erreicht. Wenn der vierte Punkt im Inneren der Kugel liegt, so lassen wir die Kugel schrumpfen.
Der Mittelpunkt der Umkugel ist Schnittpunkt der Mittelebenen je zweier Punkte.

```
/**
3-Simplex
hubert@grassmann.info
4.11.08
*/
#include "colors.inc"
#include "glass.inc"
#declare intervall = 3;
#declare winkel = 30;
#declare hoehe = 5;
#declare KameraManuell = 1;
#declare KamPos = <3,3,-3>;
#include "anageoL2.inc"
#include "transforms.inc"
#version 3.5;

global_settings { max_trace_level 20 }
background { color rgb <1, 1, 1> }
light_source { <20, 20, -50>  color rgb <1, 1, 1> }

#macro start()
  #local A = <1,1,0>;
  #local B = <-2,0,-1>;
  #local C = <0,1,1>;
  #local D = <1,-1,-1>;

simplex(A,B,C,D)

  #local a1=0;  #local b1=0;  #local c1=0;
  #local d1=0;  #local m1 = <0,0,0>;
  #local a2=0;  #local b2=0;  #local c2=0;
  #local d2=0;  #local m2 = <0,0,0>;
  #local a3=0;  #local b3=0;  #local c3=0;
  #local d3=0;  #local m3 = <0,0,0>;
  mittelebenengleichung(a1,a2,a3,d1,m1,A,B)
  mittelebenengleichung(b1,b2,b3,d2,m2,A,C)
  mittelebenengleichung(c1,c2,c3,d3,m3,A,D)

  mittelebene(A,B,pigment{color rgbt<1,0,0,0.9>})
  mittelebene(A,C,pigment{color rgbt<0,1,0,0.9>})
  mittelebene(A,D,pigment{color rgbt<0,0,1,0.9>})
```

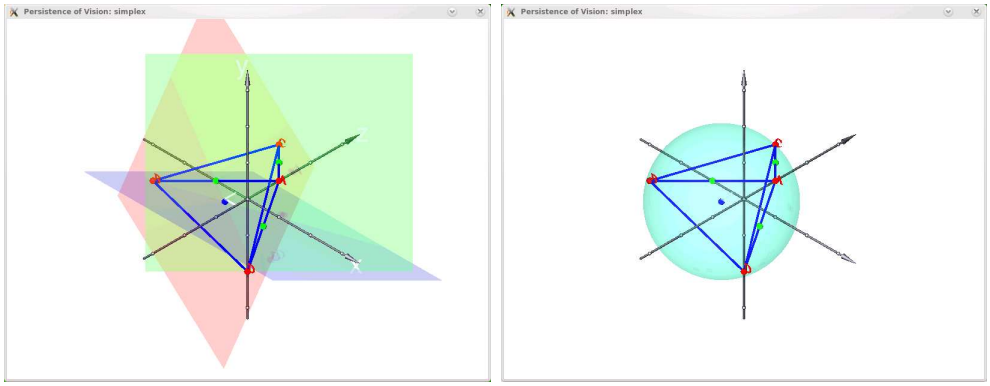
```
  punkt(m1, pigment{color Green})
  punkt(m2, pigment{color Green})
  punkt(m3, pigment{color Green})

  #local e = <0,0,0>;
  #local v1 = <a1,b1,c1>;
  #local v2 = <a2,b2,c2>;
  #local v3 = <a3,b3,c3>;

  cramer(v1,v2,v3,<d1,d2,d3>,e)
  pluspunkt(e, pigment{color Blue})
  sphere{<e.x,e.y,e.z>, vlength(D-e) pigment {Col_Glass_Green}}
  koordinatensystem(3)
#end

/** erstellt 3x3-Matrix m mit den Spalten A,B,C*/
#macro make(A,B,C,m)
  #declare m = array[3][3]
  { {A.x,B.x,C.x}, {A.y,B.y,C.y}, {A.z,B.z,C.z}
  }
#end

/**Determinante */
#macro det3(a)
  ( a[0][0]*a[1][1]*a[2][2] +a[0][1]*a[1][2]*a[2][0]
   +a[1][0]*a[2][1]*a[0][2] -a[2][0]*a[1][1]*a[0][2]
   -a[0][1]*a[1][0]*a[2][2] -a[0][0]*a[2][1]*a[1][2])
#end

/** die Spalten der Koeff.-Matrix sind a,b,c; rechte Seite d;
Loesung e */
#macro cramer(a,b,c,d,e)
  #local null = array[3][3];
  make(<0,0,0>,<0,0,0>,<0,0,0>,null)
  #local m1 = null;  #local m2 = null;
  #local m3 = null;  #local m = null;
  make(a,b,c,m)
  make(d,b,c,m1)
  make(a,d,c,m2)
  make(a,b,d,m3)
  #local de = det3(m);
  #local e1 = det3(m1)/de;
  #local e2 = det3(m2)/de;
  #local e3 = det3(m3)/de;
  #declare e = <e1,e2,e3>;
#end
```

```
/** druckt n an der Stelle (i,j)*/
#macro print(n, i, j)
  text{ttf "timrom.ttf" str(n 0,2)  0.1,0 pigment{Red}
       translate 3.3*<j,-i,0> scale 0.3}
#end

/** Matrixausgabe */
#macro zeig(a)
 #local m = dimension_size(a,1);
 #local n = dimension_size(a,2);
 #local i = 0;
union
{
 #while (i < m)
   #local j = 0;
   #while (j < n)
     print(a[i][j], i, j)
     #local j = j+1;
   #end
   #local i = i+1;
 #end
}
#end

#macro simplex(A,B,C,D)
  strecke(A, B pigment{color rgb<0,0,1>})
  strecke(A, C pigment{color rgb<0,0,1>})
  strecke(A, D pigment{color rgb<0,0,1>})
  strecke(C, B pigment{color rgb<0,0,1>})
  strecke(D, B pigment{color rgb<0,0,1>})
  strecke(D, C pigment{color rgb<0,0,1>})
  punkt(A, pigment{color Red})
  punkt(B, pigment{color Red})
  punkt(C, pigment{color Red})
  punkt(D, pigment{color Red})
text{ttf "timrom.ttf" "A"  0.1,0 pigment{Red} translate 3.3*A scale 0.3}
text{ttf "timrom.ttf" "B"  0.1,0 pigment{Red} translate 3.3*B scale 0.3}
text{ttf "timrom.ttf" "C"  0.1,0 pigment{Red} translate 3.3*C scale 0.3}
text{ttf "timrom.ttf" "D"  0.1,0 pigment{Red} translate 3.3*D scale 0.3}
#end

#macro mittelebenengleichung(a,b,c,d,m,A,B)
/* Die Gleichung der zur Strecke AB senkrechten Ebene, die durch deren
   Mittelpunkt m geht, hat die Gleichung ax+by+cz=d */
#declare ab = B-A;   #declare a = ab.x;
#declare b = ab.y;   #declare c = ab.z;
#declare m = A+ab/2; #declare d = a*m.x + b*m.y + c*m.z;
#end
```

```
// Makro: Mittelebene: Ebene (als duenner Quader) durch den Mittelpunkt
   einer Strecke AB, senkrecht zu AB

#macro mittelebene(A,B,textur)
#declare ab = B-A;
#declare X = ab.x;
#declare Y = ab.y;
#declare Z = ab.z;
#declare m = A+ab/2;
#declare K = X*m.x + Y*m.y + Z*m.z;
#if ( Y = 0 )
 #if ( Z = 0 )
  box {<-intervall/1000, - intervall, -intervall>
       <intervall/1000, intervall, intervall>
    texture {textur}   no_shadow
    translate m
  }
 #else
  box {<-intervall, - intervall, -intervall/1000>
       <intervall , intervall , intervall/1000>
    texture {textur}   no_shadow
    rotate <0, degrees(atan2(X,Z)) ,0>
    translate m
  }
 #end
#else
 #if ( X*X + Z*Z = 0 )
  box {<-intervall, - intervall/1000, -intervall>
       <intervall, intervall/1000, intervall>
  translate <0,-K/Y,0>
  texture {textur}   no_shadow
  }
 #else
  #declare VX1 = <0,1,0>;
  #declare VX2=vnormalize(<X,Y,Z>);
  #declare VY=vnormalize(vcross(VX2,VX1));
  #declare VZ1=vcross(VY,VX1);
  #declare VZ2=vcross(VY,VX2);
  box {<-intervall, - intervall/1000, -intervall>
       <intervall, intervall/1000, intervall>
  matrix < VX1.x, VY.x, VZ1.x,
           VX1.y, VY.y, VZ1.y,
           VX1.z, VY.z, VZ1.z,
               0     0      0 >
  matrix < VX2.x, VX2.y, VX2.z,
            VY.x,  VY.y,  VY.z,
           VZ2.x, VZ2.y, VZ2.z,
```

```
                0,      0,      0 >
  translate m
  texture {textur}   no_shadow
  }
 #end
#end
#end

start()
```

Hier sehen wir drei Mittelebenen und die Umkugel:



Der folgende Text stammt von André Henning.

Hier sollen nun die In- und Umkugeln räumlicher Vierecke (Tetraeder) dargestellt werden. Diese bilden das dreidimensionale Analogon zu den In- und Umkreisen von Dreiecken in der Ebene. Das Tetraeder soll dabei nur an Hand seiner vier Eckpunkte gegeben sein. Die vier Eckpunkte dürfen nicht in einer Ebene liegen, da sonst kein Tetraeder entsteht.
Die Umkugel ist hierbei jene Kugel, die das Tetraeder genau in seinen vier Eckpunkten berührt. Die Inkugel diejenige Kugel, die das Tetraeder in jeder Seitenfläche genau einmal berührt.
Eine ausführliche Beschreibung mit Beweisen und ausführlicherer Herleitung des Quelltextes findet sich in der Bachelorarbeit „In- und Umkugeln räumlicher Vierecke“ (André Henning, Humboldt Universität zu Berlin, 17.06.2009).

Die Umkugel

Für die Umkugel bestimmt man zunächst ihren Mittelpunkt. Dieser ist der Schnittpunkt der mittelsenkrechten Ebenen auf die Kanten des Tetraeders. Man kann relativ schnell Normalenvektoren der Ebenen bestimmen und so ein Gleichungssystem aufstellen, dessen eindeutige Lösung die Koordinaten des Mittelpunkts der Umkugel liefert.
Der Radius der Umkugel ist offensichtlich der Abstand vom Mittelpunkt zu einer der Ecken des Tetraeders.
Als Makro in POV-Ray sieht das folgendermaßen aus:

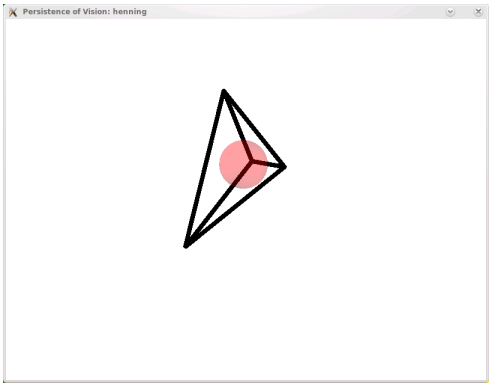
```
#macro tetra_umk(P1, P2, P3, P4)
#if (vdot(vcross(P2-P1,P3-P1),P4-P1) = 0)
  text{ ttf "ARIAL.TTF" "Fehler, die vier Punkte liegen in einer Ebene."
         ,0.1,0  no_shadow }
#else
  #local A = 0.5*(P1+P2);
  #local B = 0.5*(P1+P3);
  #local C = 0.5*(P3+P4);

  #local n1 = 0.5*(P1-P2);
  #local n2 = 0.5*(P3-P1);
  #local n3 = 0.5*(P4-P3);

  #local r3_zaehler =
     vdot(C,n3) - (n3.x/n1.x)*vdot(A,n1) -
     ((vdot(B,n2) - (n2.x*n1.x)*vdot(A,n1)) *
     ((n3.y - n1.y*(n3.x/n1.x)) / (n2.y - n1.y*(n2.x/n1.x))));
   #local r3_nenner =
     n3.z - n1.z*(n3.x/n1.x) -
     (n2.z-n1.z*(n2.x/n1.x))*((n3.y-n1.y*(n3.x/n1.x))
      / (n2.y-n1.y*(n2.x/n1.x)));
   #local r3 = r3_zaehler/r3_nenner;

   #local r2_zaehler = vdot(B,n2) - (n2.x/n1.x)*vdot(A,n1) -
      r3*(n2.z - n1.z*(n2.x/n1.x));
   #local r2_nenner = n2.y - n1.y*(n2.x/n1.x);
   #local r2 = r2_zaehler/r2_nenner;

   #local r1 = (vdot(A,n1) - r2*n1.y - r3*n1.z)/n1.x;

   #local M_Umkugel = <r1,r2,r3>;

   #local Radius_Umkugel = sqrt(vdot((M_Umkugel-P1),(M_Umkugel-P1)));

   cylinder{P1,P2, 0.025 texture{pigment{color Green}}}
   cylinder{P1,P3, 0.025 texture{pigment{color Green}}}
   cylinder{P1,P4, 0.025 texture{pigment{color Green}}}
   cylinder{P2,P3, 0.025 texture{pigment{color Green}}}
   cylinder{P2,P4, 0.025 texture{pigment{color Green}}}
   cylinder{P3,P4, 0.025 texture{pigment{color Green}}}

   sphere{M_Umkugel, Radius_Umkugel
     texture{pigment{color rgb<10,10,10> transmit 0.8}}}
#end
#end
```

Die Inkugel

Für die Inkugel muss ebenfalls der Mittelpunkt bestimmt werden. Dies gestaltet sich jedoch nicht ganz so einfach, wie bei der Umkugel. Man benötigt hier die winkelhalbierenden Ebenen zu den Seitenflächen des Tetraeders. Davon gibt es jedoch pro Seitenflächenpaar immer zwei. Benötigt wird jedoch die Winkelhalbierende, die in das Tetraeder hineingeht. Um diese zu bestimmen, wird zunächst mit Hilfe des Spatprodukts bestimmt, wie ausgewählte Eckpunkte des Tetraeders orientiert sind, um daran eine Fallunterscheidung zur Bestimmung der Winkelhalbierenden zu erhalten.
Dann bestimmt man über die Normalenvektoren der Seitenflächen und Vektoren innerhalb der Schnittkante der Seitenflächen die Normalenvektoren der Winkelhalbierenden.
Im Anschluss daran löst man, wie bei der Umkugel auch, ein Gleichungssystem und erhält so die Koordinaten des Mittelpunkts der Inkugel.
Der Radius ist dann die Länge des Lotes vom Inkugelmittelpunkt auf eine der Seitenflächen des Tetraeders.
Als POV-Ray Makro erhält man folgendes:

```
#macro tetra_ink(P1, P2, P3, P4)
#if (vdot(vcross(P2-P1,P3-P1),P4-P1) = 0)
text{ ttf "ARIAL.TTF" "Fehler, die vier Punkte liegen in einer Ebene."
         ,0.1,0 no_shadow }
#else
#if (vdot(vcross(P2-P1,P3-P1),P4-P1) > 0)
  #local n132 = vcross(P2-P1,P3-P1)/
     sqrt(vdot(vcross(P2-P1,P3-P1),vcross(P2-P1,P3-P1)));
  #local n124 = vcross(P4-P1,P2-P1)/
     sqrt(vdot(vcross(P4-P1,P2-P1),vcross(P4-P1,P2-P1)));
  #local n143 = vcross(P3-P1,P4-P1)/
     sqrt(vdot(vcross(P3-P1,P4-P1),vcross(P3-P1,P4-P1)));
  #local n423 = vcross(P3-P4,P2-P4)/
     sqrt(vdot(vcross(P3-P4,P2-P4),vcross(P3-P4,P2-P4)));
#else
  #local n132 = vcross(P3-P1,P2-P1)/
     sqrt(vdot(vcross(P3-P1,P2-P1),vcross(P3-P1,P2-P1)));
  #local n124 = vcross(P2-P1,P4-P1)/
     sqrt(vdot(vcross(P2-P1,P4-P1),vcross(P2-P1,P4-P1)));
  #local n143 = vcross(P4-P1,P3-P1)/
     sqrt(vdot(vcross(P4-P1,P3-P1),vcross(P4-P1,P3-P1)));
  #local n423 = vcross(P3-P2,P4-P2)/
     sqrt(vdot(vcross(P3-P2,P4-P2),vcross(P3-P2,P4-P2)));
#end

#local n  = vcross(P4-P1,n124 + n143);
#local n0 = n/(sqrt(vdot(n,n)));
#local m  = vcross(P2-P1,n132 + n124);
#local m0 = m/(sqrt(vdot(m,m)));
#local o  = vcross(P3-P2,n423 + n132);
#local o0 = o/(sqrt(vdot(o,o)));

#local r3_zaehler = vdot(P2,o0) - (o0.x/n0.x)*vdot(P1,n0) -
```

```
   (vdot(P1,m0)-(m0.x/n0.x)*vdot(P1,n0)) *
   ((o0.y-(n0.y*o0.x/n0.x))/(m0.y-(n0.y*m0.x/n0.x)));
#local r3_nenner  = o0.z-(n0.z*o0.x/n0.x) -
   (m0.z -(n0.z*m0.x/n0.x))*
   ((o0.y-(n0.y*o0.x/n0.x))/(m0.y-(n0.y*m0.x/n0.x)));
#local r3 = r3_zaehler/r3_nenner ;

#local r2 = (vdot(P1,m0) - (m0.x/n0.x)*vdot(P1,n0) -
   r3*(m0.z - (n0.z*m0.x/n0.x)) ) / (m0.y - (n0.y*m0.x/n0.x));

#local r1 = (vdot(P1,n0) - r2*n0.y - r3*n0.z)/n0.x;

#local M_Inkugel = <r1,r2,r3>;

#local Radius_Inkugel = vdot(M_Inkugel,n124)- vdot(P1,n124);

cylinder{P1,P2, 0.05 }
cylinder{P1,P3, 0.05 }
cylinder{P1,P4, 0.05 }
cylinder{P2,P3, 0.05 }
cylinder{P2,P4, 0.05 }
cylinder{P3,P4, 0.05 }

sphere{M_Inkugel, Radius_Inkugel
        texture{pigment{color White transmit 0.8}}}
#end
#end
```



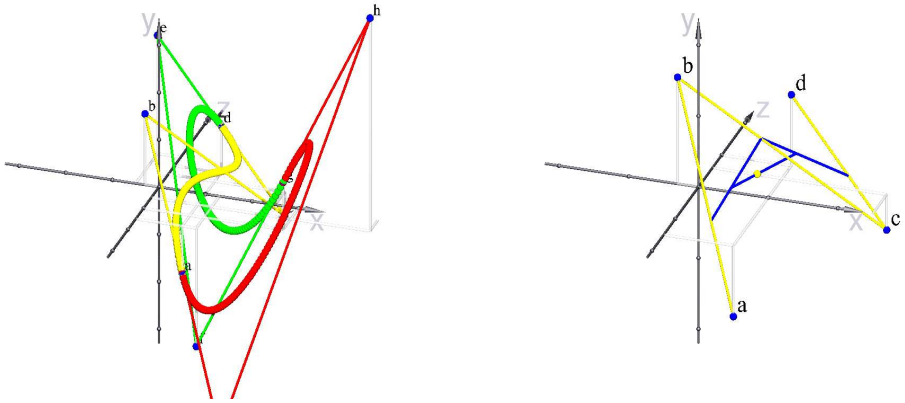
8.8 Bézier-Kurven im Raum

Wir haben uns bereits im Abschnitt 4.1 mit ebenen Bézierkurven beschäftigt, wir wollen das nun im Raum tun.

```
#declare a = <1,-1,-1.5>;      #declare b = <-0.3,1.5,0>;
#declare c = <2.5,-0.5,0.6>;   #declare d = <1,1,1>;
```

```
#declare e = 2*d-c;            #declare f = a-x;

/** Punkt mit Parameter T auf der Strecke ab*/
#macro punktT(a, b, T)
  (1-T)*a + T*b
#end

#macro kurve(a, b, c, d, farbe)
  #local T = 0.0;
  #while (T <= 1)
    #declare ab = punktT(a,b,T);   #declare bc = punktT(b,c,T);
    #declare cd = punktT(c,d,T);   #declare abc = punktT(ab,bc,T);
    #declare bcd = punktT(bc, cd, T);
    #declare abcd = punktT(abc, bcd, T);
    punkt(abcd, farbe)
    #local T = T + 0.01;
  #end
#end

strecke (a, b, pigment{color Yellow}) strecke (b, c, pigment{color Yellow})
strecke (c, d, pigment{color Yellow}) strecke (d, e, pigment{color Green})
strecke (e, f, pigment{color Green}) strecke (f, a, pigment{color Green})
kurve(a,b,c,d, pigment{color Yellow}) kurve(d,e,f,a, pigment{color Green})
```

Hier haben wir vier Orientierungspunkte a, b, c, d, die Kurve beginnt in a, wobei die Gerade ab ihre Tangente ist und endet in d mit der Tangente cd. Wenn die Tangenten gleich sind, erhalten wir glatte Übergänge.



Im rechten Bild sehen wir, wie die Punkte der gelben Kurve gefunden wurden: Wir beschreiben die Punkte der Strecken mit der Parameterdarstellung $p = a + t \cdot \vec{ab}$, hier haben wir auf den Strecken ab, bc und cd $t = 0.4$ gewählt, auf den Verbindungsstrecken dieser Punkte wählen wir den Punkt zum Parameter t und auf der Verbindungsstrecke dieser Punkte wieder den Punkt mit dem Parameter t: das ist unser Kurvenpunkt.

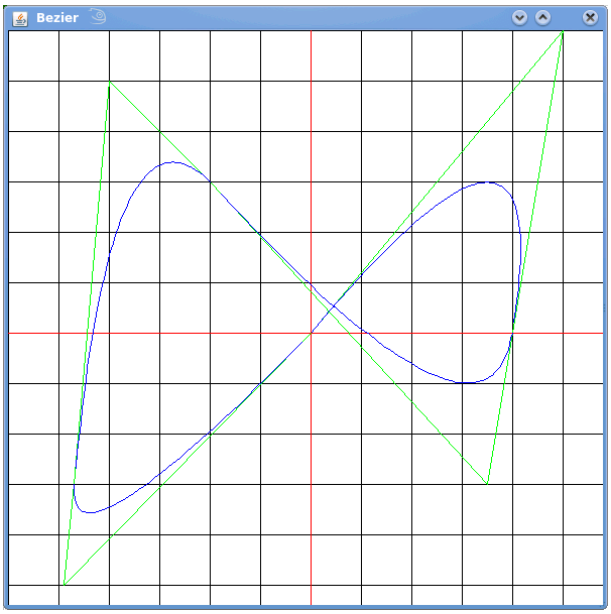
Die Gleichung der Kurve lautet

$$P(t) = -at^3 + 3bt^3 - 3ct^3 + dt^3 + 3at^2 - 6bt^2 + 3ct^2 - 3at + 3bt + a.$$

(Für 3 Punkte, die ja in einer Ebene liegen, wäre dies die Kurvengleichung: $ct^2 - 2bt^2 + at^2 + 2bt - 2at + a$.)

8.9 Bézier-Flächen

Eine Bézier-Kurve ist eine durch drei Punkte bestimmte Kurve 2. Grades: sie beginnt am ersten und endet am dritten Punkt und die Verbindungsgerade zwischen 1. und 2. Punkte ist deren Tagente am ersten Punkt, die Verbindungsgerade zwischen 2. und 3. Punkt ist die Tangente am dritten Punkt. Wenn man mehrere solche Kurvenstücke so zusammensetzt, daß die die Tangentenrichtungen an den Anschlußpunkten übereinstimmen, erhält man glatte Übergänge zwischen den Kurvenstücken. Das folgende Bild ist aus drei Kurvenstücken zusammengesetzt, sie verlaufen zwischen den Berührungspunkten der Tangenten:



(Wikipedia) A given Bézier surface of order (n, m) is defined by a set of $(n+1)(m+1)$ control points $k_{i,j}$. It maps the unit square into a smooth-continuous surface embedded within a space of the same dimensionality as $k_{i,j}$. For example, if k are all points in a four-dimensional space, then the surface will be within a four-dimensional space.
A two-dimensional Bézier surface can be defined as a parametric surface where the position of a point p as a function of the parametric coordinates u, v is given by:

$$p(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} B_i^n(u)\ B_j^m(v)\ k_{i,j}$$

evaluated over the unit square, where

$$B_i^n(u) = \binom{n}{i}\ u^i (1-u)^{n-i}$$

is a Bernstein polynomial.
Some properties of Bézier surfaces:

- A Bézier surface will transform in the same way as its control points under all linear transformations and translations.
- All u = constant and v = constant lines in the (u, v) space, and, in particular, all four edges of the deformed (u, v) unit square are Bézier curves.
- The surface does not in general pass through the control points except for the corners of the control point grid.
- The surface is contained within the convex hull of the control points.

Die folgenden Java-Methoden aus dem Paket `HUMath.Algebra` können helfen, die benötigten Werte zu berechnen:

```
/** i-tes Bernstein-Polynom vom Grad n */
public static DX bernstein(int n, int i)    //erzeugt B^n_i(x)
{
  DX p, q, d, x, e;
  e = new DX();  e.co = 1.0;              // 1
  x = new DX();  x.co = 1.0; x.ex = 1;    // x
  d = DX.sub(e, x);                       // 1 - x
  if ((i < 0) || (n < i)) return null;
  else if (n == 0) return e;
  else                                    // Rekursion
  {
    p = DX.mult(d, bernstein(n-1, i));
    q = DX.mult(x, bernstein(n-1, i-1));
    return DX.add(p, q);
  }
}

/** Zahl einsetzen; hier ist bei evtl. schwach besetzten Polynomen
    das Horner-Schema unangebracht */
public static double wert(DX p, double x)
{
  DX f; double erg = 0.0, h, f = p;
  while (!iszero(f))
  {
    h = Math.pow(x, f.ex);
    h = h * f.co;
    erg = erg + h;
    f = f.next;
  }
  return erg;
}
```

Hier sind die Ergebnisse:

$B_0^1 = -x + 1$

$B_1^1 = x$

$B_0^2 = x^2 - 2x + 1$

$B_1^2 = -2x^2 + 2x$

$B_2^2 = x^2$

$B_0^3 = -x^3 + 3x^2 - 3x + 1$

$B_1^3 = 3x^3 - 6x^2 + 3x$

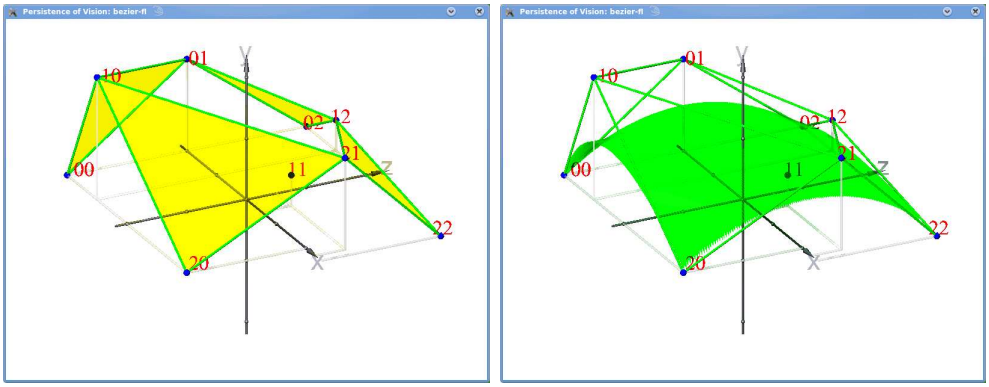
$B_2^3 = -3x^3 + 3x^2$

$B_3^3 = x^3$

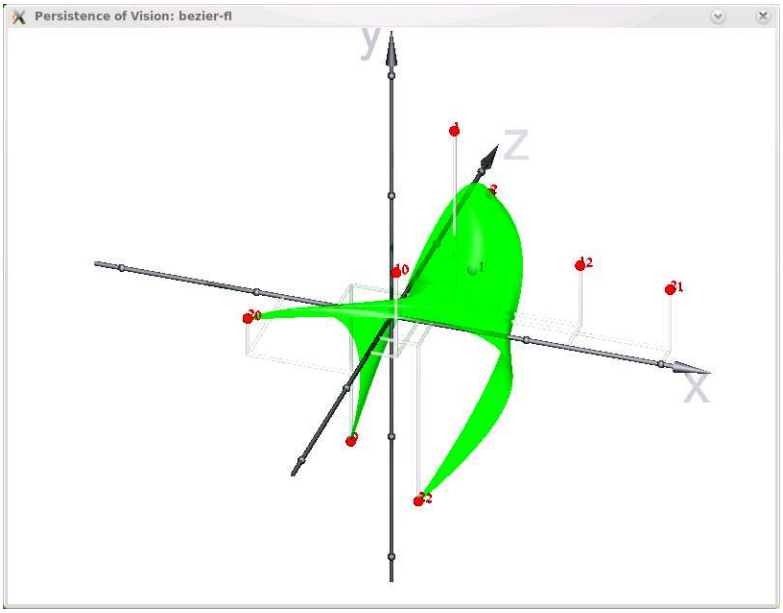
Wir stellen uns die 9 Kontrollpunkte als die Einträge einer 3×3-Matrix vor. Vier der 9 Punkte (die an den „Ecken“ der Matrix gelegenen) liegen auf der Fläche, sie sind deren „Eckpunkte“; die Dreiecke

$$[p_{0,0}, p_{0,1}, p_{1,0}], [p_{0,2}, p_{1,2}, p_{0,1}], [p_{2,0}, p_{1,0}, p_{2,1}], [p_{2,2}, p_{1,2}, p_{2,1}]$$

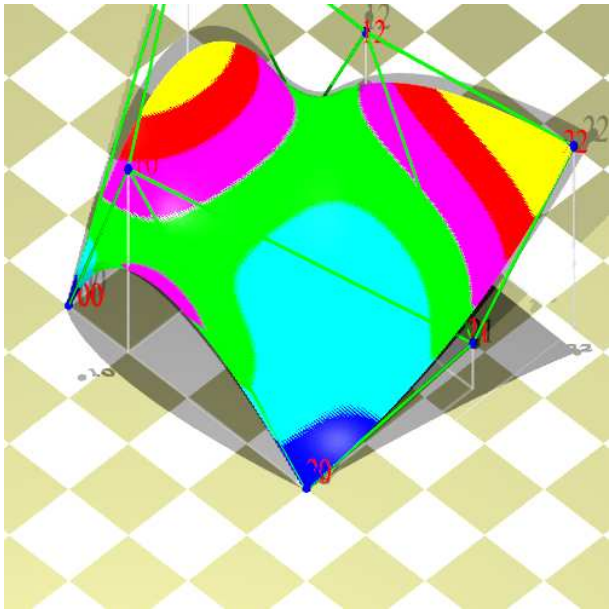
tangieren die Fläche. Wenn man die Punkte etwa so wie die Matrixeinträge anordnet, erhält man so etwas wie ein Dach:



Jedoch können die Punkte beliebig im Raum verteilt sein:



Hier wird der Rock etwas gelüpft und es sind Höheninformationen enthalten:

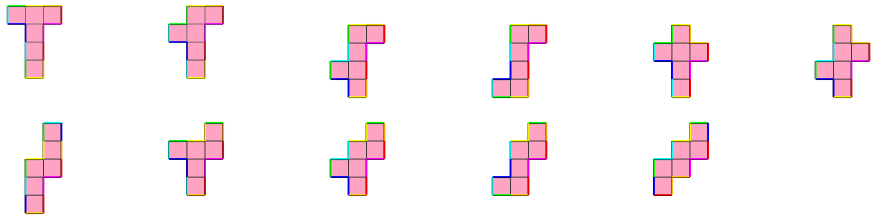


8.10 Quadernetze

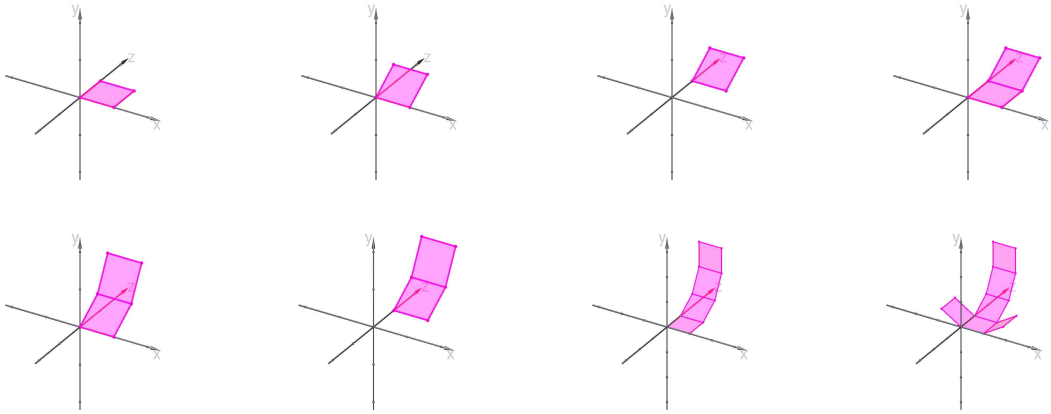
Der folgende Text und die Abbildungen stammen von Hanna Müller (Bachelorarbeit „Würfelnetze und ihre Darstellung in POV-Ray“ Humboldt Universität zu Berlin, 19.2.2010).

Ein Würfel besteht aus sechs kongruenten Quadraten, die seine Seitenflächen ausmachen, zwölf gleichlangen Kanten und acht Ecken, an denen je drei Seitenflächen sowie drei Kanten zusammentreffen. Alle Netze des Würfels bestehen aus diesen Seitenflächen, also aus sechs kongruenten Quadraten, die auf unterschiedliche Arten zusammenhängen.
Im Folgenden betrachte ich die kombinatorischen Anordnungen dieser Qua- drate. In sechs Schritten werde ich jeweils eine Fläche hinzufügen (dranheften) und die verschiedenen Möglichkeiten der Anheftung durchgehen. Dabei werden kongruente Anordnungen nicht beachtet und unmögliche Anordnungen ausgeschlossen. Die Seiten der Quadrate, die beim Falten des Polyeders zusammen geheftet werden, also eine Kante des Würfels bilden, kennzeichne ich durch entsprechende Farbgebung.
1. Bei dem ersten Quadrat gibt es noch keine Variationsmöglichkeiten. 2. Egal an welcher der vier Kanten das zweite Quadrat angeheftet wird, es entsteht immer eine kongruente Figur. Daher bleibt auch hier nur eine Anordnung.
3. Das dritte Quadrat kann an sechs Seiten angeheftet werden, es ent- stehen dabei zwei inkongruente Anordnungen. In der Anordnung 3.2. gibt es erstmalig zwei Seitenflächen, die beim Falten des Polyeders eine gemeinsame Kante bilden. 4. Das vierte Quadrat wird an beide Varianten von 3. angehängt. Es gibt vier mögliche Varianten. Außerdem entsteht hier die erste unmögliche Anordnung. 5. Das Anheften des fünften Quadrates erzeugt acht verschiedene mögliche Anordnungen. Zudem entstehen drei unmögliche Strukturen.

6. Durch das Anheften des sechsten und somit letzten Quadrates werden die elf endgültigen regulären Netze des Würfels vervollständigt.



Nun wollen wir aus diesen Netzen wirklich Würfel falten, es sollen Videos hergestellt werden. Dazu müssen die Seitenflächen schrittweise gedreht werden. Der `rotate`-Befehl dreht um die Koordinatenachsen, wir drehen also ein Quadrat, verschieben es um die Kantenlänge nach hinten, fügen vorn wieder ein Quadrat an, das wir mit dem ersten durch eine `union`-Anweisung verbinden und drehen nun dieses Objekt gemeinsam. Das tun wir mit allen Seitenflächen. Nach einiger Zeit sieht es so aus:

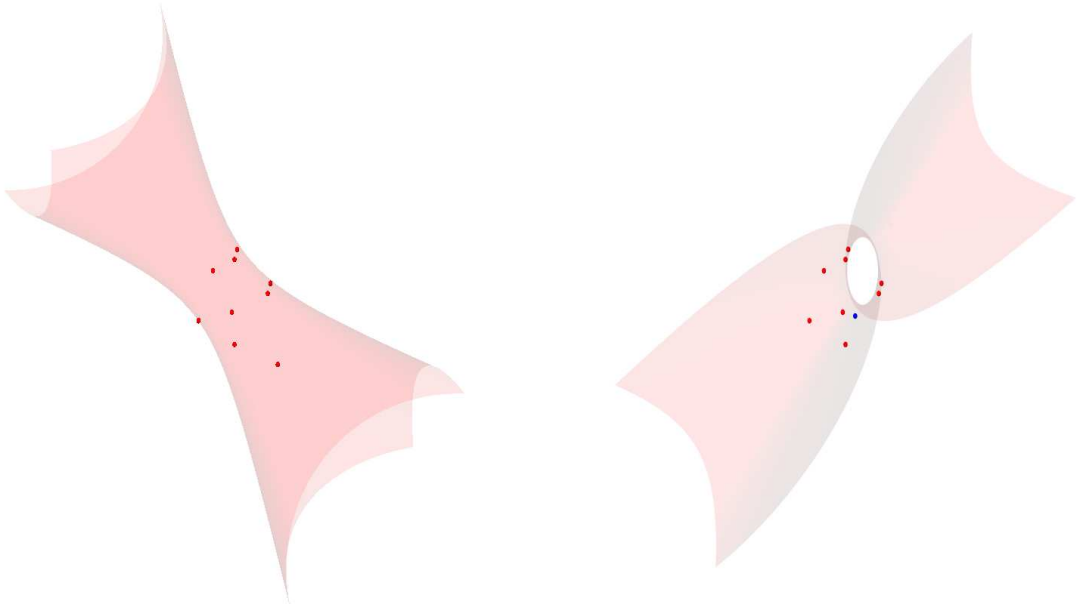


```
#declare Fold_Angle = -90*clock;

union{ // A + B + C
  object{ Square no_shadow} // C
  union{ // A + B
     object{ Square no_shadow} // B
     object{ Square   // A
             rotate<Fold_Angle ,0,0>
             translate<0,0,1> no_shadow
           }
     rotate<Fold_Angle,0,0>
     translate<0,0,1>
  } // end A + B
  rotate<Fold_Angle,0,0>
  translate<0,0,1>
} // end A + B + C
object{ Square no_shadow} // D
```

```
object{ Square   // E
        translate<-1,0,0>
        rotate<0,0,Fold_Angle> no_shadow
      }
object{ Square   // F
        rotate<0,0,-Fold_Angle>
        translate<1,0,0> no_shadow
      }
```

8.11 Neun Punkte

Eine Gerade ist durch zwei Punkte bestimmt, eine Kurve zweiten Grades durch fünf Punkte, und eine quadratische Fläche durch neun Punkte.



Die roten Punkte befinden sich an denselben Stellen, nur der blaue hat sich verändert.

Hier sind die Koordinaten der Punkte eingetragen.

Um die Koeefizienten des Polynoms

$$f(x,y,z) = ax^2 + by^2 + cz^2 + dxy + exz + fyz + gx + hy + iz + j,$$

dessen Nullstellenmenge die Punkte $p_1, \ldots, p_9$ enthalten soll zu bestimmen, setzen wir die Werte der p_i ein und lösen das lineare Gleichungssystem $f(x,y,z) = 0$ für die Koordinaten der Punkte p_i.
Ich hatte keine Lust, den Gaußschen Algorithmus in Povray zu implementieren; hier hilft uns folgende Beobachtung:

Sei A eine $n \times n$-Matrix vom Rang $n-1$, dann ist eine nichttriviale Lösung des Gleichungssystems $Ax = 0$ durch die Adjunkten gegeben:

$$x = (A_{11}, A_{12}, A_{13}, \ldots, A_{1n}),$$

Beweis: Nach dem Laplaceschen Entwicklungssatz ist

$$\sum a_{1i} A_{1i} = det(A) = 0$$

und

$$\sum a_{ji} A_{jj} = 0 \text{ für } j \neq 1,$$

da hier zwei Zeilen übereinstimmen. □
Es müssen also nur noch die Determinanten einiger 9×9-Matrizen berechnet werden. Die Determinante einer Matrix ist der konstante Term des charakteristischen Polynoms, und dessen Koeffizienten berechnet man mit der Formel

$$c_i = \sum_{j=0}^{i-1} c_j Spur(A^{i-j}).$$

Das wird hier alles getan:

```
#include "colors.inc"
#include "Matrizen.inc"
#declare intervall = 10;
#declare winkel = 30;
#declare hoehe = 5;
#declare KameraManuell = 1;
#declare KamPos = <0,0,-3>;
#include "anageoL2.inc"
#include "transforms.inc"
#version 3.5;
global_settings { max_trace_level 20 }
background { color rgbt <1, 1, 1, 0.1> }

/** benannter Punkt */
#macro Punkt(P, Name, textur)
  union{
    sphere{<P.x,P.y,P.z>, intervall/80}
    object
    {
      text{ttf "timrom.ttf" Name 0.1,0 pigment{Red} scale 0.5}
      matrix
      < z1.x,z1.y,z1.z,
z2.x,z2.y,z2.z,
z3.x,z3.y,z3.z,
P.x+0.4,P.y,P.z>
    }
    texture{textur}
  }
#end

/*streicht k-te Zeile von a */
#macro streich(k, a)
#local n = dimension_size(a,2);
#local b = array[n][n];
#local i= 0;
#while (i < k)
  #local j = 0;
  #while (j < n)
    #local b[i][j] = a[i][j];
    #local j = j+1;
  #end
  #local i = i+1;
#end
#while (i < n)
  #local j = 0;
  #while (j < n)
    #local b[i][j] = a[i+1][j];
    #local j = j+1;
```

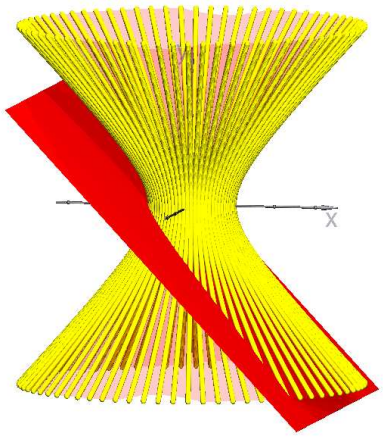
```
  #end
  #local i = i+1;
#end
b
#end

/* stellt Matrix mit Koeff. von 1,x,y,z,xy,...,z^2 her*/
#macro mat(p)
#local m = array[10][9];
#local i = 0;
#while (i < 9) #local m[0][i] = p[i].x*p[i].x; #local i = i+1; #end
#local i = 0;
#while (i < 9) #local m[1][i] = p[i].y*p[i].y; #local i = i+1; #end
#local i = 0;
#while (i < 9) #local m[2][i] = p[i].z*p[i].z; #local i = i+1; #end
#local i = 0;
#while (i < 9) #local m[3][i] = p[i].x*p[i].y; #local i = i+1; #end
#local i = 0;
#while (i < 9) #local m[4][i] = p[i].x*p[i].z; #local i = i+1; #end
#local i = 0;
#while (i < 9) #local m[5][i] = p[i].z*p[i].y; #local i = i+1; #end
#local i = 0;
#while (i < 9) #local m[6][i] = p[i].x; #local i = i+1; #end
#local i = 0;
#while (i < 9) #local m[7][i] = p[i].y; #local i = i+1; #end
#local i = 0;
#while (i < 9) #local m[8][i] = p[i].z; #local i = i+1; #end
#local i = 0;
#while (i < 9) #local m[9][i] = 1; #local i = i+1; #end
m
#end

/** Potenzsummen der Eigenwerte */
#macro potenzsummen(a)
  #local n = dimension_size(a,1);
  #local s = array[n+1];
  #local b = a;
  #local s[0] = n;
  #local s[1] = spur(a);
  #local i = 2;
  #while (i <= n)
    #local b = mult(a, b);
    #local s[i] = spur(b);
    #local i = i+1;
  #end
  s
#end
```

```
/** charakteristisches Polynom als Koeffizientenfolge*/
#macro charpol(a)
  #local n = dimension_size(a,1);
  #local c = array[n+1];
  #local s = potenzsummen(a);
  #local c[0] = 1.0;
  #local i = 1;
  #while (i <= n)
    #local h = 0;
    #local j = 0;
    #while (j < i)
      #local h = h + c[j]*s[i-j];
      #local j = j+1;
    #end
    #local c[i] = -h/i;
    #local i = i+1;
  #end
  c
#end

#declare p = array[9]
{
  <5-10*clock,0,0>,
  <1,2,3>,
  <4,5,6>,
  <7,8,9>,
  <3,6,9>,
  <-1,2,3>,
  <4,-5,6>,
  <7,8,-9>,
  <-3,-6,-9>
}

/*Quadrik durch 9 Punkte */
#macro koeff(p)
#local a = mat(p);
#local k = array[10];
#local i = 0;
#while (i < 10)
  #local b = streich(i,a);
  #local erg = charpol(b);
  #if (odd(i))
    #local k[i] = erg[9];
  #else
    #local k[i] = - erg[9];
  #end
  #local i = i+1;
#end
```

```
k
#end

#declare k = koeff(p);
#declare Q = <k[0],k[1],k[2]>;
#declare R = <k[3],k[4],k[5]>;
#declare S = <k[6],k[7],k[8]>;
#declare T = k[9];
#declare i = 0;
#while (i < 9)
  Punkt(p[i], vstr(3,p[i],",",0,0), pigment{color Red})
  #declare i = i+1;
#end

quadric{Q,R,S,T pigment{rgbt <1,0,0,0.9>}
  clipped_by{box{-intervall,intervall}}
  bounded_by{box{-intervall,intervall}}}
```

8.12 Hyperboloid-Schnitte

Was für Kurven entstehen, wenn man ein Hyperbolid mit einer Ebene schneidet?



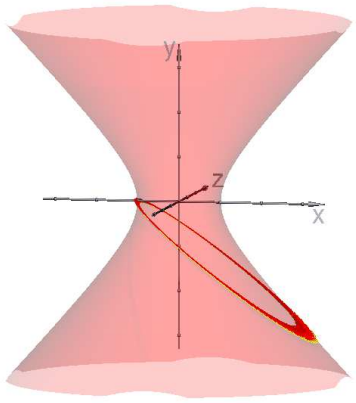
Wenn man die lineare Gleichung der Ebene nach einer Variablen auflöst und dies in die quadratische Gleichung der Ebene einsetzt, entsteht eine Gleichung zweiten Grades in zwei Variablen, und diese stellt bekanntlich einen Kegelschnitt dar.
Ein Hyperboloid erhält man so:

```
quadric{<1,-1,1>,0,0,-1 pigment{color rgbt<1,0,0,0.8>}
```

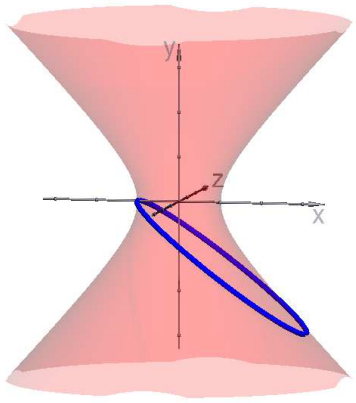
Dies ist aber ein zweidimensionales Gebilde und eignet sich nicht zum Bilden von Durchschnitten. Mas folgende Makro ergibt eine Hyperbel, die aus 800 Punkten zusammengesetzt ist:

```
#macro hyp()
#local i = 4;
#local inc = 0.02;
union
{
#while (i >= -4)
  punkt(<sqrt(1+i*i),i,0> pigment{color Yellow})
  #local i = i - inc;
#end
}
#end
```

Läßt man diese Hyperbel um die y-Achse rotieren, so erhält man ein dreidimensionales Objekt aus Tausenden von Punkten und man kann den Durchschnitt mit einem dünnen Quader bilden:



Die Herstellung dieses Bildes dauerte 90 Minutan und es entstehen unschöne scheifende Schnitte. Besser ist also, die Schnittkurve zu berechnen und punktweise zu zeichnen.



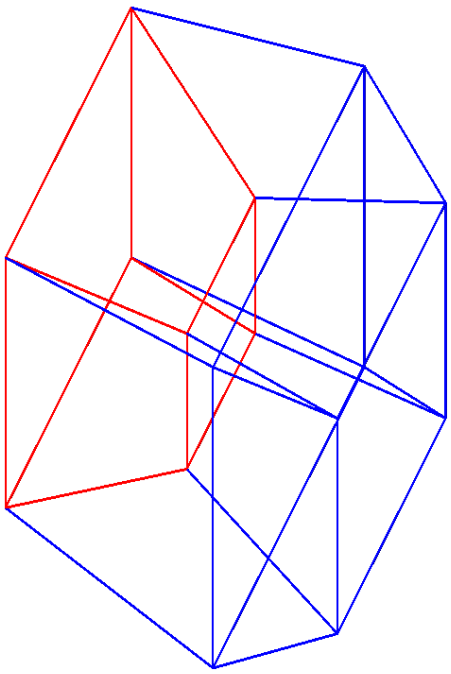
8.13 . . . und was ist mit 4D?

Berd Grave hat ein Java-Applet erstellt, das einen rotierenden vierdimensionalen Würfel zeigt:

`http://www.uni-math.gwdg.de/bgr/wuerfel.php`

Die 16 Eckpunkte des Würfels werden in einem 5-dimensionalen Feld `vertices4` gespeichert, deren erste vier Komponenten die Werte 0 oder 1 annehmen, diese nummerieren die Ecken; die fünfte Komponente enthält die 4-dimensionalen Koordinaten jeder Ecke (initialisiert mit *pminus*1). Bernd Grave projiziert den vierdimensionalen Raum zuerst auf den dreidimensionalen und diesen auf die Ebene (den letzten Schritt übernimmt Povray für uns).

Es folgt eine Anpassung an Povray; die beiden Parameter des Makros `rot` (zwischen 0 und 3) bezeichnen die Ebene, in der gedreht wird. Bei einer Animation sollte die Varable `clock` die Werte 0 bis 99 durchlaufen.



```
/**
4-cube
hubert@grassmann.info
24.5.12
*/
#include "colors.inc"
#include "glass.inc"
#declare intervall = 3;
#declare winkel = 30;
#declare hoehe = 5;
#declare KameraManuell = 1;
#declare KamPos = <1,3,-4>;
#include "anageoL2.inc"
#include "transforms.inc"
#include "Matrizen.inc"
#version 3.5;
//koordinatensystem(intervall)
global_settings { max_trace_level 20 }
background { color Gray90 }
light_source { <20, 20, -50>  color rgb <1, 1, 1> }
light_source { <-5, 5, -2>  color rgb <1, 0, 1> }

/**
 * Es wird ein sich drehender 4-dimensionaler Wuerfel auf die Ebene projiziert
 *
 * @author (Bernd Grave Jakobi)
```

```
 * @version (Mai 2003)
 */

  // Position des Auges bei der Zentralprojektion
  #declare eye= array[4] {0,0,0,-1.5};
  #declare l = 2;
  // Ecken des vierdimensionalen Wuerfels
   #declare vertices4 = array[2][2][2][2][4];
  // Projektion der Ecken ins dreidimensionale
   #declare vertices3 = array[2][2][2][2][3];

    #local i0 = 0;
    #while (i0<=1)
      #local i1 = 0;
      #while (i1<=1)
        #local i2 = 0;
        #while (i2<=1)
          #local i3 = 0;
          #while (i3<=1)
            #local vertices4[i0][i1][i2][i3][0] = 2*i0 - 1;
            #local vertices4[i0][i1][i2][i3][1] = 2*i1 - 1;
            #local vertices4[i0][i1][i2][i3][2] = 2*i2 - 1;
            #local vertices4[i0][i1][i2][i3][3] = 2*i3 - 1;
            #local i3 = i3+1;
          #end
          #local i2 = i2+1;
        #end
        #local i1 = i1+1;
      #end
      #local i0 = i0+1;
    #end

    #macro Sqr(x)
  x*x
#end

  #macro project4to3(vertices4, vertices3)
    #local i0 = 0;
    #while (i0<=1)
      #local i1 = 0;
      #while (i1<=1)
        #local i2 = 0;
        #while (i2<=1)
          #local i3 = 0;
          #while (i3<=1)
            #local dist = sqrt(
                Sqr(eye[0]-vertices4[i0][i1][i2][i3][0])
              + Sqr(eye[1]-vertices4[i0][i1][i2][i3][1])
```

```
            + Sqr(eye[2]-vertices4[i0][i1][i2][i3][2])
            + Sqr(eye[3]-vertices4[i0][i1][i2][i3][3]);
          #local n = 0;
          #while (n<=2)
            #declare vertices3[i0][i1][i2][i3][n]
                   = vertices4[i0][i1][i2][i3][n] * dist/l;
            #local n = n+1;
          #end
          #local i3 = i3+1;
        #end
        #local i2 = i2+1;
      #end
      #local i1 = i1+1;
    #end
    #local i0 = i0+1;
  #end
#end

#macro farbe(i,j,k,l)
  #if (i=0)
    <1,0,0>
  #else
    <0,0,1>
  #end
#end

#macro drawcube(vertices3)
    #local i0=0;
    #while (i0<=1)
      #local j0=0;
      #while (j0<=1)
        #local i1=0;
        #while (i1<=1)
          #local j1=0;
          #while (j1<=1)
            #local i2=0;
            #while (i2<=1)
              #local j2=0;
              #while (j2<=1)
                #local i3=0;
                #while (i3<=1)
                  #local j3=0;
                  #while (j3<=1)
                    #local d1 = i0-j0 + i1-j1 + i2-j2 + i3-j3;
                    #local d2 = abs(i0-j0)+abs(i1-j1)+abs(i2-j2)+abs(i3-j3);
                    #if ((d1=1) & (d2=1))
      strecke(<vertices3[i0][i1][i2][i3][0],
       vertices3[i0][i1][i2][i3][1],
```

```
       vertices3[i0][i1][i2][i3][2]>,
      <vertices3[j0][j1][j2][j3][0],
       vertices3[j0][j1][j2][j3][1],
       vertices3[j0][j1][j2][j3][2]>,
       pigment{color rgb farbe(i0,i1,i2,i3)})
                    #end
                    #local j3 = j3+1;
                  #end
                  #local i3 = i3+1;
                #end
                #local j2 = j2+1;
              #end
              #local i2 = i2+1;
            #end
            #local j1 = j1+1;
          #end
          #local i1 = i1+1;
        #end
        #local j0 = j0+1;
      #end
      #local i0 = i0+1;
    #end
#end

// Drehung in der (i,j)-Ebene
#macro rot(vertices4, i, j)
    #local U = array[4][4];
    #local winkel = clock*pi/50;
    #local U = einh(4);
    #local U[i][i] = cos(winkel);
    #local U[i][j] = -sin(winkel);
    #local U[j][i] = sin(winkel);
    #local U[j][j] = cos(winkel);
    #local i0 = 0;
    #while (i0<=1)
      #local i1 = 0;
      #while (i1<=1)
        #local i2 = 0;
        #while (i2<=1)
          #local i3 = 0;
          #while (i3<=1)
            #local v4 = array[4][1]
                        {{vertices4[i0][i1][i2][i3][0]},
                         {vertices4[i0][i1][i2][i3][1]},
                         {vertices4[i0][i1][i2][i3][2]},
                         {vertices4[i0][i1][i2][i3][3]}};
            #local v4 = mult(U,v4);
            #local n = 0;
```

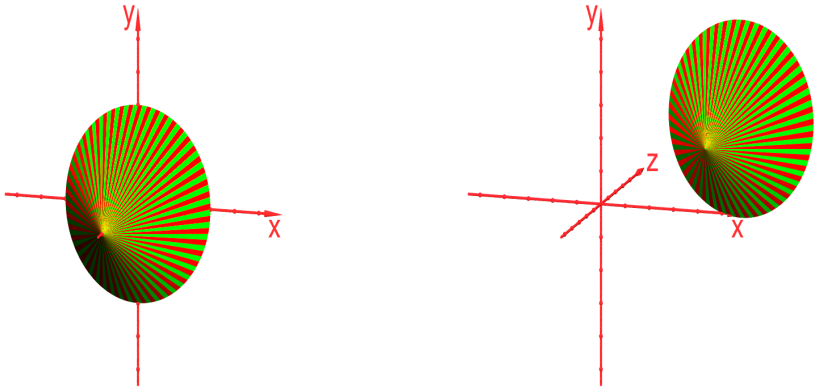
```
            #while (n<=3)
              #local vertices4[i0][i1][i2][i3][n] = v4[n][0];
              #local n = n+1;
            #end
            #local i3 = i3+1;
          #end
          #local i2 = i2+1;
        #end
        #local i1 = i1+1;
      #end
      #local i0 = i0+1;
    #end
#end

rot(vertices4,0,3)
project4to3(vertices4,vertices3)
drawcube(vertices3)
```

9 Aufgaben für alle

9.1 Rollende Körper

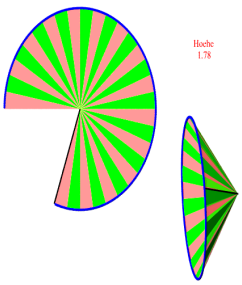
Im Internet findet man Videos eines „aufwärtsrollenden“ Doppelkegels. Ein wesentlicher Punkt war der Kegel selbst:

```
union{
#declare n = 0;
#while (n < 100)
  triangle{<r*sin(n*eps),r*cos(n*eps),0>,
           <r*sin((n+1)*eps),r*cos((n+1)*eps),0>,
           <0,0,-h> pigment{color rgbt <1,0,0>}}
  #declare n = n+2;
#end
#declare n = 1;
#while (n < 100)
  triangle{<r*sin(n*eps),r*cos(n*eps),0>,
           <r*sin((n+1)*eps),r*cos((n+1)*eps),0>,
           <0,0,-h> pigment{color rgbt <0,1,0>}}
  #declare n = n+2;
#end
rotate - z * clock
translate <0.2*clock, 0.1*clock, 0>
}
```



9.2 Tüten kleben

Damit wurden früher Gefängnisinsassen beschäftigt; man brauchte keine Qualifikation und benutzte keine Werkzeuge.
Es ist einfach, aus einem Kreisauschnitt einen Kegel zu formen: Die blauen und die schwarzen Linien sind jeweils gleichlang.



Stellen wir die umgekehrte Frage: Es soll ein Kegel mit vorgegebenem Öffnungswinkel hergestellt werden. Bei Google findet man bei der Suche nach „Kegel berechnen“:

```
www.mathepower.com/kegel.php
Dieses Skript berechnet aus zwei beliebigen Angaben eines
Kegels alle übrigen Maße.
```

Dort kommt allerdings der Öffungswinkel gar nicht vor. Dabei ist der Zusammenhang zwischen dem dem Winkel α des Kreisausschnitts und dem Öffnungswinkel β des Kegels ganz einfach:

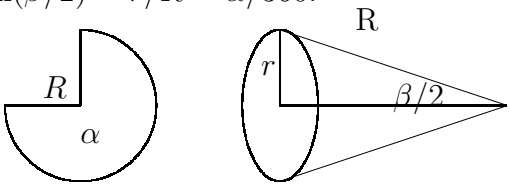
$$\sin(\frac{\beta}{2}) = \frac{\alpha}{360}.$$

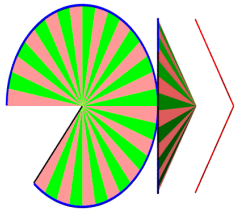
Beweis: Sei α der Ausschnittswinkel und R der Radius des Kreises, dann ist die Länge des Kreibogens gleich $\frac{\alpha}{360} \cdot 2\pi R$, dies ist der Umfang U des Kegel-Grundkreises, also ist

dessen Radius gleich

$$r = \frac{U}{2\pi} = \frac{\alpha}{360} \cdot R;$$

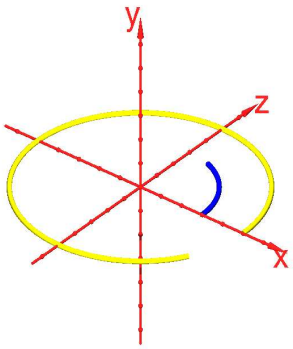
schließlich erhalten wir $\sin(\beta/2) = r/R = \alpha/360$. □





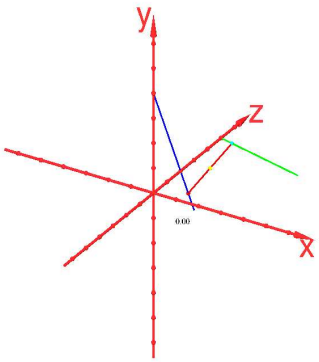
9.3 Winkelbögen

Bei geometrischen Zeichnungen bezeichnet man Winkel durch eine einbeschrieben Bogen. Hier schließ der blaue Bogen 70 Grad und der gelbe 150 Grad ein, sie beginnen jeweils an der x-Achse.



9.4 Kürzeste Verbindung windschiefer Geraden

Die blaue und die grüne Gerade sind gegeben, gesucht sind die am dichtesten benachbarten Punkte dieser Geraden. Die rote Strecke ist deren Verbindungsstrecke, sie steht auf beiden Geraden senkrecht.



9.5 Tische müssen nicht wackeln

`http://mitschriebwiki.nomeata.de/WackelndeTische.pdf`
Unter dieser folgenden Adresse wird ein Problem behandelt, das im Alltag zu lösen ist. Es folgt als Zitat die Einleitung.

WACKELFREIE POSITIONIERUNG VON VIERBEINIGEN RECHTECKIGEN TISCHEN AUF STETIGEN FLÄCHEN

JOACHIM BREITNER

Gewidmet dem Gartentisch in der Bothestr. 30, Heidelberg

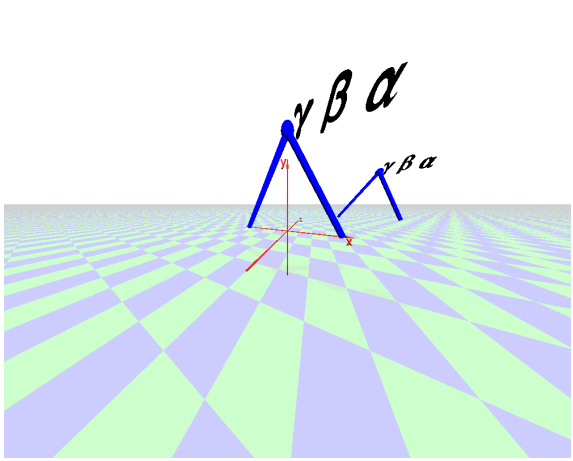
Zusammenfassung. Ein sich alljährlich wiederholendes Problem ist die Positionierung eines Tisches auf unebenem Untergrund, mit dem Ziel der Wackelfreiheit. Wackelfreiheit ist definiert als der Zustand, in dem jeder Fuß des Tisches den Boden berührt. Es gibt verschiedene Methoden, diesen Zustand zu erreichen. Zum Teil benötigen diese Hilfsmittel, beispielweisweise Bierdeckel zum Füllen des leeren Raumes zwischen Boden und Tischbeinen oder Hostsägen zum Verkürzen von zu langen Tischbeinen. Alternativ wird versucht, durch das Probieren verschiedener Positionen des Tisches eine zu finden, in der Wackelfreiheit gewährleistet ist. Beliebt ist dabei der Spezialfall des Drehens, und eben dieser wird hier mit den Mitteln der Mathematik untersucht.

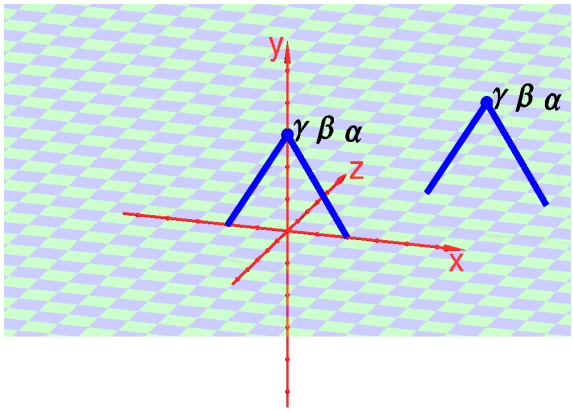
9.6 Angle vs. angel

Unter `http://www.youtube.com/watch?feature=endscreen&NR=1&v=Q9aV_rqzRqE` gibt es ein Lied aus den 60ern, es fängt so an:

You look like an angel
Walk like an angel
Talk like an angel
But I got wise
You're the devil in disguise.

Ersetzen Sie „angel“ durch „angle“ und denken Sie sich eine Szene dazu aus.





9.7 Vermehrung des Dodekaeders

Mit diesem Namen ist die folgende Skulptur im Park der Familie March in Cala Rajada, Mallorca, zu sehen. Versuchen Sie sich daran.