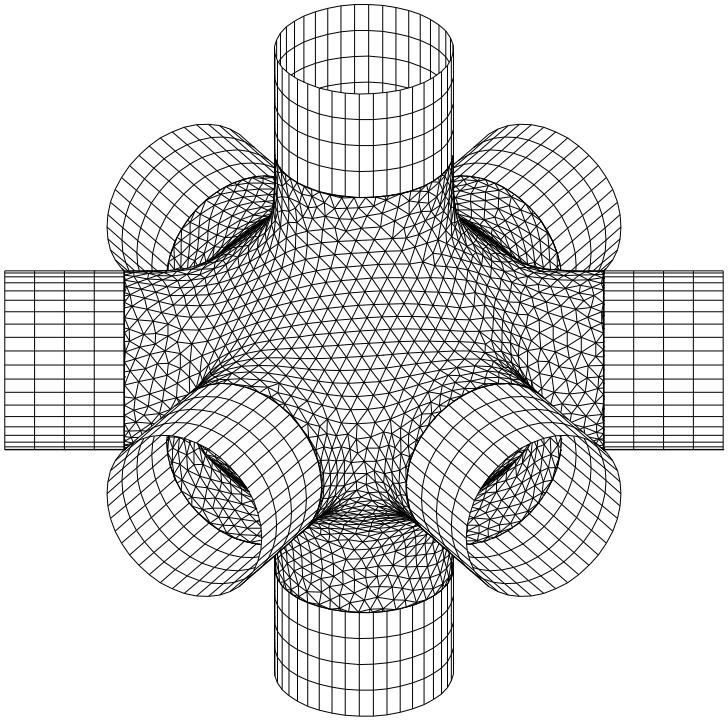
# Software zur Darstellung von Kurven und Flächen



Erich Hartmann

Technische Universität Darmstadt
2003

# Inhaltsverzeichnis

# Kapitel 1

# Einleitung

## 1.1 Aufgabe der DARSTELLENDEN Geometrie

Das Ziel der Darstellenden Geometrie ist, Bilder von räumlichen Gegenständen wie Häuser, Maschinenteile ... in einer Zeichenebene herzustellen. Dabei verwendet man hauptsächlich zwei Methoden:

**I) Parallelprojektion.**
Hierbei projiziert man die Objekte (Punkte, Kanten, Kurven,...) mit Hilfe paralleler Strahlen auf eine Ebene (Bildtafel). Steht die Bildtafel senkrecht zu den Projektionsstrahlen, so spricht man von *senkrechter Parallelprojektion* im anderen Fall von *schiefer Parallelprojektion*. Projiziert man schief auf eine horizontale Ebene (z.B. x-y-Ebene), so nennt man diese Art *Vogelperspektive*. Bei einer *Kavalliersperspektive* projiziert man schief auf eine senkrecht stehende Ebene.

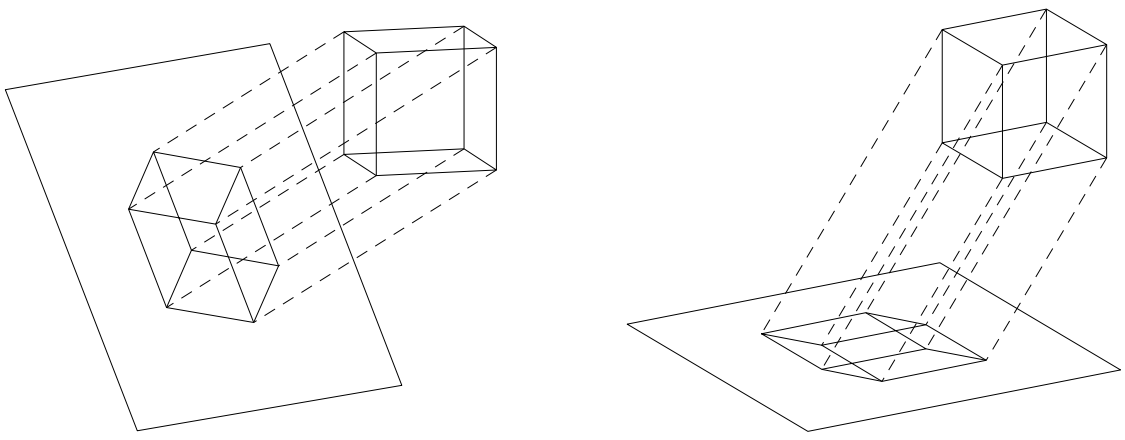


Abbildung 1.1: Senkrechte PARALLELprojektion bzw. Vogelperspektive eines Würfels

**II) Zentralprojektion.**
Dem Sehen ähnlicher ist die Zentralprojektion. Hier werden die Objekte mit Hilfe von durch einen Punkt Z (das Zentrum oder der Augpunkt) gehende Strahlen zur Abbildung auf einer Bildtafel benutzt.
Die Gestaltungsmöglichkeiten bei Zentralprojektion ist durch die Verwendung der zusätzlichen Parameter Augpunkt und Distanz (des Augpunktes zur Bildtafel) vielfältiger. Arbeitet man mit Zirkel und Lineal, so ist allerdings der Aufwand zur Erstellung einer Zeichnung auch wesentlich größer. Auch bei Verwendung eines Rechners muß man den Vorteil von "schönen" Bildern durch eine etwas

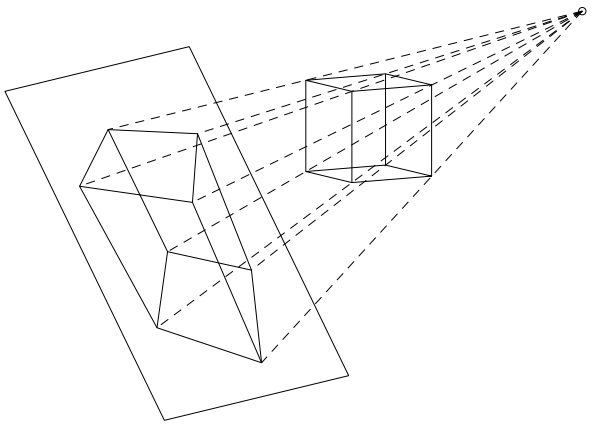


Abbildung 1.2: ZENTRALprojektion eines Würfels

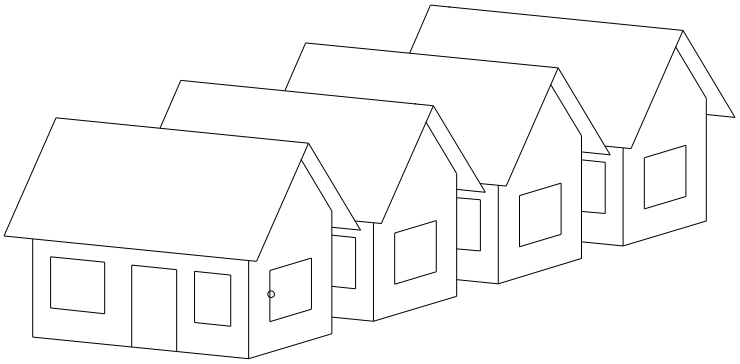


Abbildung 1.3: PARALLELprojektion einer Häuserreihe

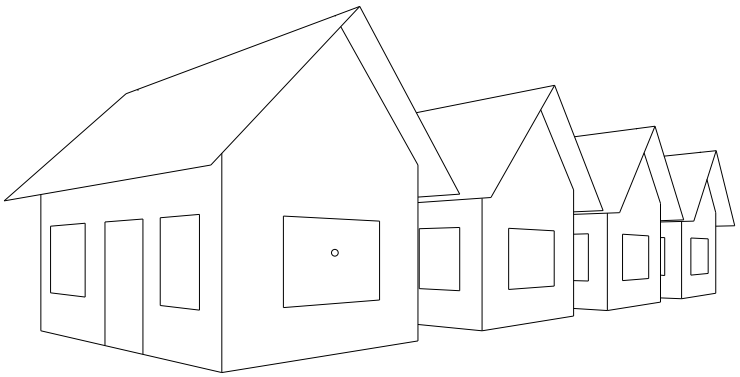


Abbildung 1.4: ZENTRALprojektion einer Häuserreihe

längere Rechenzeit erkaufen, da eine Zentralprojektion, im Gegensatz zu einer Parallelprojektion, nicht durch eine lineare Abbildung beschrieben werden kann. In der Technik gibt man i.a. der senkrechten Parallelprojektion den Vorzug, da bei Parallelprojektionen Proportionen (Teilverhältnisse) erhalten bleiben.

## 1.2 Über den Inhalt

Im Kapitel 2: **Hilfsmittel** werden zunächst die Anforderungen formuliert, die wir an die zu verwendende Graphik-Software stellen. Es wird die hier benutzte Datenstruktur eingeführt und an einem einfachen Beispiel (N-Eck) ihre Verwendung demonstriert. Die zugrunde liegende Programmiersprache ist PASCAL. Doch lassen sich alle Prozeduren ohne Mühe in andere Sprachen übersetzen. Die Übersetzung in C kann sogar "automatisch" mit einer geeigneten Software vorgenommen werden. Ferner enthält das Kapitel viele Grundroutinen aus der analytischen Geometrie.
In Kapitel 3 werden die Prozeduren für die senkrechte Parallelprojektion und die Zentralprojektion besprochen.
Kapitel 4 zeigt wie man parametrisierte und ebene implizite **Kurven** darstellt.
Kapitel 5 enthält einen Hiddenline-Algorithmus zur Darstellung von nicht konvexen Polyedern und seine Anwendung auf **parametrisierte Flächen**.
Kapitel 6 enthält einen Algorithmus zur Triangulierung von **impliziten Flächen** (siehe Beispiele in `trisample.p`), die dann mit dem Hiddenline–algorithmus aus Kapitel 5 visualisiert werden können.

Alle zugehörigen PASCALprogramme sind über das Internet unter

`http://www.mathematik.tu-darmstadt.de/~ehartmann`

zu beziehen.

**Installation** des Programmsystems auf einem LINUX-Rechner:

1. download von `cdg0gv.tgz`
2. Auspacken `tar xvfz cdg0gv.tgz` erzeugt einen Ordner `cdg0gv` mit Unterverzeichnissen `beispiele, include, tools, units`.
   Der FREE-PASCAL-Compiler und `gv` müssen installiert sein.
3. Gehe in Ordner `beispiele` und führe den Befehl `make` aus (es werden die notwendigen units erzeugt, das Beispiel `n_eck` und in tools das Programm `pldv` übersetzt.
4. Übersetzung eines Pogramms, z.B. `tori_h.p`: `make tori_h`
   Start des Programms: `tori_h`
5. Eine PLD-Datei (z.B. n_eck.pld, s. `graph_on(..)` in Abschnitt 2.1.4) kann mit dem Programm `pldv`
   a) auf dem Bildschirm angezeigt werden: `pldv n_eck.pld -a`
   b) eine Postscript-Datei erzeugt werden: `pldv n_eck.pld -a -pps`
   b) eine eps-Datei erzeugt werden: `pldv n_eck.pld -a -peps`
   (Alle Optionen von pldv sieht man mit `pldv`.)
6. Die für den Hiddenline-Algorithmus notwendigen Informationen a) Koordinaten der Punkte b) Punkte in einer Facette (Polygon) kann man mit dem in `flaech_h_off.p` enthaltenen UP `write_nangles_to_offfile` in eine OFF-Datei schreiben und mit dem freien Software-Paket *GEOMVIEW* weiter bearbeiten. GEOMVIEW ist erhältlich unter `http://www.geomview.org`.

**Bemerkung:**
Die ursprüngliche Version von `pldv` stammt von A. Görg.

# Kapitel 2

# Hilfsmittel

Wichtige Hilfsmittel der Computerunterstützten Darstellenden Geometrie stammen aus der Analytischen Geometrie. Es müssen einfache Operationen, wie Summe von Vektoren, oder Schnitte, wie Schnitt einer Gerade mit einem Kreis, berechnet werden. Hierfür stehen keine Standardbefehle in PASCAL zur Verfügung, sodaß wir zunächst entsprechende Unterprogramme bereitstellen müssen.

## 2.1 Aufbau eines Zeichenprogramms

Bevor wir auf konkrete Unterprogramme eingehen, werden häufig verwendete globale Konstanten, Typen und Variablen definiert. Sehr wesentlich sind die Typen `vt2d,vt3d,vts2d,vts3d`, die 2- bzw. 3-komponentige Vektoren bzw. Felder von solchen deklarieren.

### 2.1.1 Globale Konstanten: Datei "geoconst.pas"

Die Datei "geoconst.pas" enthält die Konstante `array_size`, die für die in 2.1.2 erklärten Typen benutzt wird, die Zahlen $\pi, 2\pi, \frac{\pi}{2}$ und eps1, ... , eps8, die bei Abschätzungen nützlich sind. Die Konstanten `black,`... werden zum setzen von Farben (s.u.) verwendet.

```
array_size= 1000;  {...20000 fuer Hiddenline-Alg.}
pi= 3.14159265358;       pi2= 6.2831853;       pih= 1.5707963;
eps1=0.1;      eps2=0.01;       eps3=0.001;     eps4=0.0001;
eps5=0.00001;  eps6=0.000001;   eps7=0.0000001; eps8=0.00000001;
default=-1; black=0; blue=1; green=2; cyan=3; red=4; magenta=5; brown=6;
lightgray=7; darkgray=8; lightblue=9; lightgreen=10; lightcyan=11;
lightred=12; lightmagenta=13; yellow=14; white=15;
```

### 2.1.2 Globale Typen: Datei "geotype.pas"

```
r_array = array[0..array_size] of real;
i_array = array[0..array_size] of integer;
b_array = array[0..array_size] of boolean;
vt2d    = record x,y: real; end;
vt3d    = record x,y,z: real; end;
vts2d   = array[0..array_size] of vt2d;
vts3d   = array[0..array_size] of vt3d;
matrix3d= array[1..3,1..3] of real;
```

### 2.1.3 Globale Variablen: Datei "geovar.pas"

```
   null2d:vt2d;  null3d:vt3d;    {Nullvektoren}
{**fuer area_2d and curve2d:}
   origin2d:vt2d;
{**fuer Parallel- und Zentral-Projektion:}
   u_angle,v_angle,              {Projektionswinkel}
   rad_u,rad_v,                  {rad(u), rad(v)}
   sin_u,cos_u,sin_v,cos_v:real; {sin-,cos- Werte von u, v}
   e1vt,e2vt,n0vt:vt3d;          {Basis-Vektoren und}
                                 {Normalen-Vektor der Bildebene}
{**fuer Zentral-Projektion:}
   mainpt,                       {Hauptpunkt}
   centre:vt3d;                  {Zentrum}
   distance:real;                {Distanz Hauptpunkt-Zentrum}
```

### 2.1.4 Anforderung an die Graphik-Software

Graphik-Software bietet heute sehr viel Komfort. Doch hängt dieser Komfort stark von der verwendeten Software ab. Um die hier angegebenen Programme leicht auf den verschiedensten Systemen zum Laufen zu bringen , wollen wir uns nur auf die folgenden 9 rechnerabhängigen Befehle stützen. Sie sind für LINUX in dem Paket **cdg0** (cdg0/driver/xgeo.c) realisiert.

1. **graph_on(ipl),** `ipl:integer`, ruft die Grafik-Software und belegt die Vektoren `null2d, null3d` mit Nullen;
   `ipl` $= 0$ : Ausgabe nur auf dem Bildschirm,
   `ipl` $\neq 0$ : es wird nach jedem Aufruf von `draw_area(...)` (s.u.) eine Datei `name.pld` angelegt, in die alle Zeichenbefehle der aktuellen Zeichnung incl. Farben und Linienstärke geschrieben werden. Mit dem Programm `pldv` läßt sich dann die Zeichnung noch einmal auf den Bildschirm schicken oder eine *POSTSCRIPT*–Datei herstellen, die man anschließend zu einem Drucker schicken oder in TEX–Dokumente einbinden kann.

2. **draw_area(width,height,x0,y0,scalefactor)** löscht den Bildschirm und legt eine Zeichenfläche mit einem rechtwinkligen Koordinatensystem fest.
   `origin2d = (x0,y0)` ist eine globale Variable.
   Alle Längen werden in mm angegeben (real-Zahlen) !
   1 mm soll auch auf dem Bildschirm als 1 mm erscheinen, falls scalefactor=1 gesetzt wird.
   Falls eine **Skalierung** (Streckung am Koordinaten-Nullpunkt der Bildtafel) gewünscht wird, so muß scalefactor entsprechend gewählt werden.

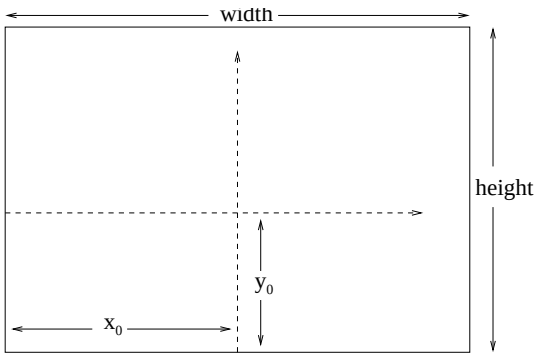



Abbildung 2.1: Koordinatenursprung in der Zeichenfläche

3. **draw_end** schließt die Zeichnung ab. Falls eine neue begonnen werden soll, muß zuerst wieder `draw_area` aufgerufen werden.

4. **graph_off** verabschiedet die Zeichensoftware endgültig.

   **ZEICHENBEFEHLE:**

5. **pointc2d(x,y,style)**, `x,y:real; style:integer`,
   markiert den Punkt (x,y) durch o falls `style` = 0, + falls `style` = 1 , ... .
   Für `style` = 10 oder 50 oder 100 erhält man kleinere ausgefüllte Kreise zur Markierung von Punkten.
   **point2d(p,style)**, `p:vt2d; style:integer`,
   wie `pointc2d`, nur mit Hilfe des Typs `vt2d` des Punktes.

6. **linec2d(x1,y1,x2,y2,style)**, `x1,y1,x2,y2:real; style:integer`,
   zeichnet die Strecke (x1,y1)(x2,y2) und zwar so:
   ———— , falls style = 0, − − − − −, falls style = 1, − · − · −, falls style = 2, .....
   Die folgenden Befehle können mit Hilfe von `linec2d` definiert werden.

   (a) **line2d(p1,p2,style)**, `p1,p2:vt2d; style:integer`,
   wie `linec2d`, nur unter Verwendung des Typs `vt2d` für Anfangs- und Endpunkt.

   (b) **arrowc2d(x1,y1,x2,y2,style)**, `x1,y1,x2,y2:real; style:integer`
   zeichnet einen Pfeil von (x1,y1) nach (x2,y2).

   (c) **arrow2d(p1,p2,style)**, `p1,p2:vt2d; style:integer`
   zeichnet einen Pfeil von $\mathbf{p}_1$ nach $\mathbf{p}_2$.

   (d) **curve2d(p,n1,n2,style)**, `p:vts2d; style:integer`,
   zeichnet den Polygonzug durch die Punkte $\mathbf{p}_{n1}, ..., \mathbf{p}_{n2}$.

   Alle Längen und Koordinaten von Vektoren sind in **mm** (Millimeter) anzugeben!

7. **new_color(color)**, `color: integer`,
   setzt eine neue Farbe. Dabei werden die in TURBO-Pascal üblichen Integercodes benutzt. Z.B.: `color = red`. `color = default` setzt die Standardfarbe schwarz.

8. **new_linewidth(factor)**, `factor: real`,
   setzt eine neue Linienstärke. `factor`=1 bedeutet normale Linienstärke.

**AUFBAU eines ZEICHENPROGRAMMS:**

Packt man alle globalen Konstanten, Typen, Variablen und Prozeduren in ein *unit* `geograph`, so hat ein Zeichenprogramm die einfache Gestalt:

```
program name;
uses geograph;
const  ...
type   ...
var    ...:vts2d;
       ...:integer;
       ...:real;
       ...
     {$i procs.pas}       {weitere Prozeduren}
{******************}
 begin  {Hauptprogramm}
 graph_on(...);
```

```
 ...
{Zeichnen:}
 draw_area(...);
 ...
 ...
 draw_end;
 ....
 graph_off;
end.
```

## 2.2 Funktionen auf $\mathbb{R}$, Operationen mit Vektoren

Im Folgenden werden PASCAL-Funktionen bzw. Prozeduren für einige reelle Funktionen und Operationen mit Vektoren zusammengestellt. Da ihre Realisierungen einfach sind, geben wir hier nur ihre Prozedurköpfe an. Die Prozedur-Texte sind in der Datei `proc_ag.pas` enthalten.

### 2.2.1 Funktionen auf $\mathbb{R}$

1. $r \to \text{sign}(r)$ (Vorzeichen von r)
   `function sign(a:real):integer;`

2. $a, b \to \max\{a, b\}$ (Maximum von a,b)
   $a, b \to \min\{a, b\}$ (Minimum von a,b)
   `function max(a,b:real):real;  function min(a,b:real):real;`

### 2.2.2 Operationen mit Vektoren

1. $x, y \to \mathbf{v} = (x, y)$,
   $x, y, z \to \mathbf{v} = (x, y, z)$
   ```
   procedure put2d(x,y:real; var v:vt2d);
   procedure put3d(x,y,z:real; var v:vt3d);
   ```
   $\mathbf{v} = (x, y, z) \to x, y, z$
   ```
   procedure get3d(v:vt3d; var x,y,z:real);
   ```

2. $r, \mathbf{v} \to r\mathbf{v}$ (Skalierung)
   ```
   procedure scale2d(r:real; v:vt2d; var vs:vt2d);
   procedure scale3d(r:real; v:vt3d; var vs:vt3d);
   ```
   $r_1, r_2, (x, y) \to (r_1 x, r_2 y)$ bzw.
   $r_1, r_2, r_3, (x, y, z) \to (r_1 x, r_2 y, r_3 z)$ (Skalierung der Koordinaten)
   ```
   procedure scaleco2d(r1,r2:real; v:vt2d; var vs:vt2d);
   procedure scaleco3d(r1,r2,r3:real; v:vt3d; var vs:vt3d);
   ```

3. $\mathbf{v}_1, \mathbf{v}_2 \to \mathbf{v} = \mathbf{v}_1 + \mathbf{v}_2$ (Summe zweier Vektoren)
   ```
   procedure sum2d(v1,v2:vt2d; var vs:vt2d);
   procedure sum3d(v1,v2:vt3d; var vs:vt3d);
   ```
   $\mathbf{v}_1, \mathbf{v}_2 \to \mathbf{v} = \mathbf{v}_2 - \mathbf{v}_2$ (Differenz zweier Vektoren)
   ```
   procedure diff2d(v1,v2:vt2d; var vd:vt2d);
   procedure diff3d(v1,v2:vt3d; var vd:vt3d);
   ```

4. $r_1, \mathbf{v}_1, r_2, \mathbf{v}_2 \to \mathbf{v} = r_1\mathbf{v}_1 + r_2\mathbf{v}_2$ (Linearkombination von Vektoren)
   ```
   procedure lcomb2vt2d(r1:real; v1:vt2d; r2:real; v2:vt2d; var vlc:vt2d);
   procedure lcomb2vt3d(r1:real; v1:vt3d; r2:real; v2:vt3d; var vlc:vt3d);
   ```
   und analog Linearkombinationen von 3 bzw. 4 Vektoren:
   `lcomb3vt2d(r1,v1, r2,v2, r3,v3, vlc);`

```
lcomb3vt3d(r1,v1, r2,v2, r3,v3, vlc);
lcomb4vt2d(r1,v1, r2,v2, r3,v3, r4,v4, vlc);
lcomb4vt3d(r1,v1, r2,v2, r3,v3, r4,v4, vlc);
```

5. $\mathbf{v} = (x, y) \to |x| + |y|$ bzw. $\mathbf{v} = (x, y, z) \to |x| + |y| + |z|$
```
function abs2d(v:vt2d):real;     function abs3d(v:vt3d):real;
```

6. $\mathbf{v} = (x, y) \to \|\mathbf{v}\| = \sqrt{x^2 + y^2}$ bzw.
$\mathbf{v} = (x, y, z) \to \|\mathbf{v}\| = \sqrt{x^2 + y^2 + z^2}$
```
function length2d(v:vt2d):real;     function length3d(v:vt3d):real;
```

7. $\mathbf{v} \to \mathbf{v}/\|\mathbf{v}\|$
```
procedure normalize2d(var v:vt2d);  procedure normalize3d(var v:vt3d);
```

8. $\mathbf{p}, \mathbf{q} \to \|\mathbf{p} - \mathbf{q}\|$ $\mathbf{p}, \mathbf{q} \to \|\mathbf{p} - \mathbf{q}\|^2$
```
function distance2d(p,q:vt2d):real;
function distance3d(p,q:vt3d):real;
function distance2d_square(p,q:vt2d):real;
function distance3d_square(p,q:vt3d):real;
```

9. $\mathbf{v}_1, \mathbf{v}_2 \to \mathbf{v}_1 \cdot \mathbf{v}_2$ (Skalarprodukt)
```
function scalarp2d(v1,v2:vt2d):real;
function scalarp3d(v1,v2:vt3d):real;
```

10. $\mathbf{v}_1, \mathbf{v}_2 \to \mathbf{v}_1 \times \mathbf{v}_2$ (Vektorprodukt)
```
procedure vectorp(v1,v2:vt3d; var vp:vt3d);
```

11. $\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3 \to |\mathbf{v}_1 \mathbf{v}_2 \mathbf{v}_3|$
(Spatprodukt $\mathbf{v}_1 \cdot (\mathbf{v}_2 \times \mathbf{v}_3)$, 3x3-Determinante)
```
function determ3d(v1,v2,v3:vt3d):real;
```

12. $\cos\varphi, \sin\varphi, \mathbf{p} = (x, y) \to \mathbf{p}_r = (x\cos\varphi - y\sin\varphi, x\sin\varphi + y\cos\varphi)$
(Rotation um den Nullpunkt, Drehwinkel:$\varphi$)
```
procedure rotor2d(cos_rota,sin_rota:real; p:vt2d; var pr:vt2d);
```

13. $\cos\varphi, \sin\varphi, \mathbf{p}_0, \mathbf{p} \to \mathbf{p}_r$
(Rotation um den Punkt $\mathbf{p}_0$, Drehwinkel:$\varphi$)
```
rotp02d(cos_rota,sin_rota,p0,p, pr);
```

14. $\cos\varphi, \sin\varphi, \mathbf{p} \to \mathbf{p}_r$
(Rotation um x-Achse bzw. y-Achse, z-Achse )
```
procedure rotorx(cos_rota,sin_rota,p, pr);
procedure rotory(cos_rota,sin_rota,p, pr);
procedure rotorz(cos_rota,sin_rota,p, pr);
```

15. $\cos\varphi, \sin\varphi, \mathbf{p}_0, \mathbf{p} \to \mathbf{p}_r$
(Rotation um eine zu einer Koordinatenachse parallele Achse durch $\mathbf{p}_0$ im $\mathbb{R}^3$)
```
procedure rotp0x(cos_rota,sin_rota,p0,p, pr);
procedure rotp0y(cos_rota,sin_rota,p0,p, pr);
procedure rotp0z(cos_rota,sin_rota,p0,p, pr);
```

16. Vertauschen von Zahlen bzw. Vektoren:
$a \leftrightarrow b$ bzw. $\mathbf{v}_1 \leftrightarrow \mathbf{v}_2$
```
procedure change1d(var a,b:real);
procedure change2d(var v1,v2:vt2d);
procedure change3d(var v1,v2:vt3d);
```

**Beispiel 2.1** *Das folgende Programm zeichnet ein regelmäßiges* **n-Eck** *und, auf Wunsch, mit allen möglichen Kanten. Die Punkte des n-Ecks liegen auf einem Kreis. Der Mittelpunkt des Kreises sei der Punkt (0,0), der Radius sei r. Ist (r,0) der "0-te" Punkt des n-Ecks, so hat der i-te Punkt die Koordinaten*

$$x_i = r\cos(i\Delta\varphi), \quad y_i = r\sin(i\Delta\varphi) \quad \text{mit} \quad \Delta\varphi = 2\pi/n, \quad i = 0, ..n-1.$$

*Im Programm wird der Punkt $P_{i+1}$ durch Rotation des Punktes $P_i$ mit* `rotor2d` *berechnet.*

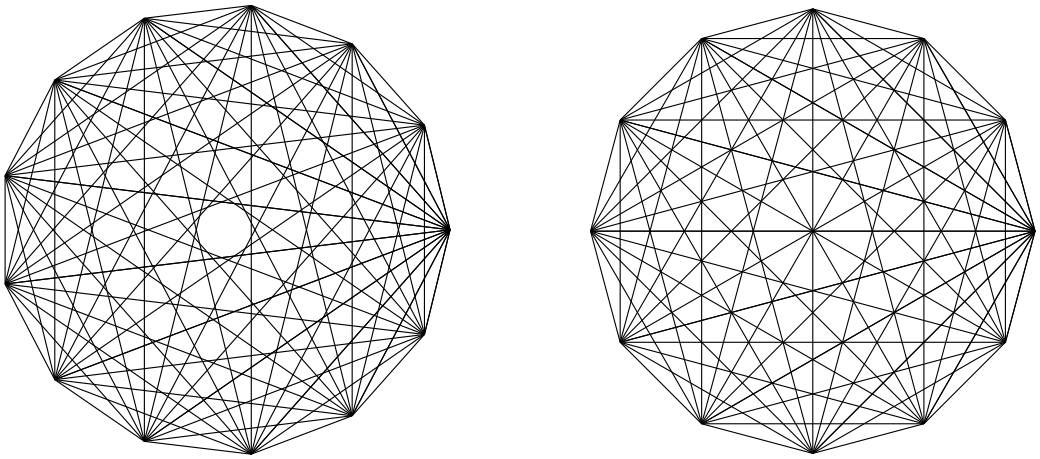


Abbildung 2.2: N-Eck mit allen Diagonalen (Beispiel 2.1)

```
{*****************************}
{***  Regelmaessiges n-Eck  ***}
{*****************************}
program n_eck;
uses geograph;
var  p : vts2d;
     n,iverb,i,j,inz: integer;
     r,dw,cdw,sdw: real;
{******************}
 begin  {Hauptprogramm}
   graph_on(0);
   repeat
     writeln('***  n-Eck  ***');
     writeln('n ?  Radius r des zugeh"origen Kreises ?');         readln(n,r);
     writeln('Jeden Punkt mit jedem Punkt verbinden ? (Ja=1)');   readln(iverb);
{Berechnung der Eckpunkte:}
     put2d(r,0, p[0]);  dw:= pi2/n;  cdw:= cos(dw);  sdw:= sin(dw);
     for i:= 0 to n-1 do  rotor2d(cdw,sdw,p[i], p[i+1]);
     draw_area(2*r+20,2*r+20,r+10,r+10,1);
{Zeichnen:}   new_color(yellow);
     if iverb=1 then
        for i:= 0 to n-1 do
            for j:= i+1 to n do
                line2d(p[i],p[j],0)
       else
        curve2d(p,0,n,0);
     draw_end;
     writeln('Noch eine Zeichnung? (ja:1, nein:0)');  readln(inz);
   until inz=0;
   graph_off;
 end.
```

**Aufgabe 2.1** *Schreibe ein Programm, das die folgenden Bilder erzeugt.*

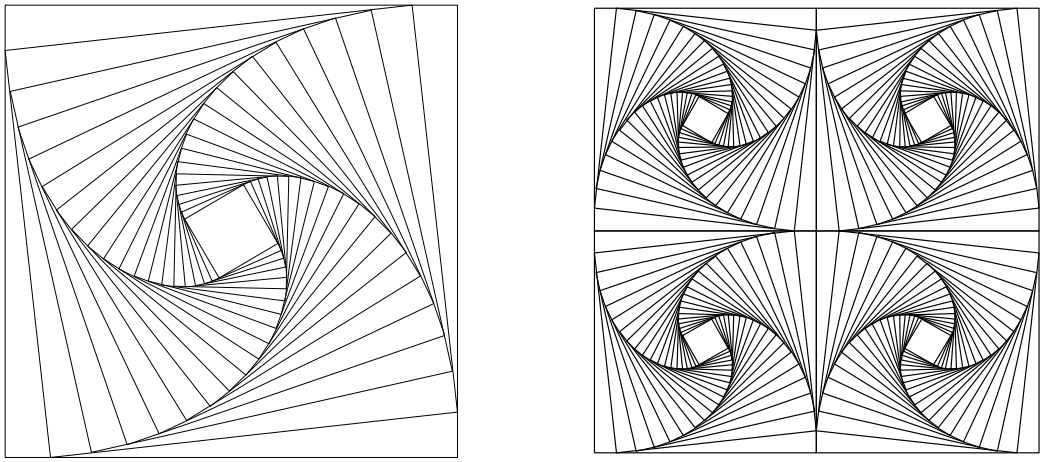


Abbildung 2.3: zur Aufgabe 2.1

## 2.3 Programme zur analytischen Geometrie

### 2.3.1 Polarwinkel und quadratische Gleichung

a) Bei der Umrechnung von rechtwinkligen Koordinaten im $\mathbb{R}^2$ in Polarkoordinaten verwenden wir die folgende Funktion `polar_angle`, die dem Punkt `(x,y)` den zugehörigen Polarwinkel zuordnet:
`function polar_angle(x,y:real):real;`

b) Reelle Lösungen einer quadratischen Gleichung $ax^2 + bx + c = 0$:
(Die Lösungen sind der Größe nach geordnet. `ns` ist die Anzahl der reellen Lösungen)
`procedure equation_degree2(a,b,c:real; var x1,x2:real; var ns:integer);`

Die Texte dieser und der folgenden Prozeduren befinden sich auf der Diskette in der Datei `proc_ag.pas`.

### 2.3.2 Schnitt Gerade-Gerade, Kreis-Gerade, Kreis-Kreis

a) Schnitt Gerade-Gerade :
Das Unterprogramm `is_line_line` verwendet die CRAMERsche Regel um den Schnittpunkt zweier Geraden zu bestimmen.

```
procedure is_line_line(a1,b1,c1, a2,b2,c2:real; var xs,ys:real; var nis:integer);
{Schnittpunkt (xs,ys) (nis=1) der Geraden a1*x+b1*y=c1, a2*x+b2*y=c2.
 Falls die Geraden parallel sind ist nis<>1.}
```

b) Schnitt Kreis-Gerade:
Kreis : $(x - x_m)^2 + (y - y_m)^2 = r^2, \quad r > 0.$
Gerade : $ax + by = c, \quad (a, b) \neq (0, 0)$
Die Substitution $\xi = x - x_m,\ \eta = y - y_m$ führt auf
$a\xi + b\eta = c' \quad$ mit $\quad c' = c - ax_m - by_m \quad$ und
$\xi^2 + \eta^2 = r^2$.
Falls $r^2(a^2 + b^2) - c'^2 > 0$ ist, erhält man die Lösungen
$\xi_{1/2} = (ac' \pm b\sqrt{r^2(a^2 + b^2) - c'^2}/(a^2 + b^2)\ , \qquad \eta_{1/2} = (bc' \mp a\sqrt{r^2(a^2 + b^2) - c'^2}/(a^2 + b^2)$
und damit
$x_{1/2} = x_m + \xi_{1/2}, \qquad y_{1/2} = y_m + \eta_{1/2}.$

```
procedure is_circle_line(xm,ym,r, a,b,c:real; var x1,y1,x2,y2:real; var nis:integer);
{Schnitt Kreis-Gerade: sqr(x-xm)+sqr(y-ym)=r*r, a*x+b*y=c,
 Schnittpkte: (x1,y1),(x2,y2). Es ist x1<=x2, nis Anzahl der Schnittpunkte.}
```

Oft muß der Schnitt des Einheitskreises ($x^2 + y^2 = 1$) mit einer Gerade berechnet werden:

```
procedure is_unitcircle_line(a,b,c:real; var x1,y1,x2,y2:real; var nis:integer);
```

Man beachte, dass in beiden Prozeduren $x_1 \leq x_2$ gilt.

c) Schnitt Kreis-Kreis :
1. Kreis : $\quad (x - x_1)^2 + (y - y_1)^2 = r_1^2, \quad r_1 > 0,$
2. Kreis : $\quad (x - x_2)^2 + (y - y_2)^2 = r_2^2, \quad r_2 > 0, \quad (x_1, y_1) \neq (x_2, y_2).$
Dieses Gleichungssystem ist zu dem folgenden äquivalent:
$(x - x_1)^2 + (y - y_1)^2 = r_1^2, \quad ax + by = c \quad$ mit
$a = 2(x_2 - x_1), \quad b = 2(y_2 - y_1) \quad$ und $\quad c = r_2^2 - x_1^2 - y_1^2 - r_2^2 + x_2^2 + y_2^2.$
D.h. die Schnittpunkte der beiden Kreise sind identisch mit den Schnittpunkten des 1. Kreises und der Geraden $ax + by = c$ .

```
procedure is_circle_circle(xm1,ym1,r1,xm2,ym2,r2:real; var x1,y1,x2,y2:real; var nis:integer);
{Schnitt Kreis-Kreis. Es ist x1<=x2. nis = Anzahl der Schnittpunkte.}
```

### 2.3.3 Gleichung einer Ebene

Gegeben: 3 Punkte $P_i : \mathbf{p}_i, \quad i = 1, 2, 3.$
Gesucht: Gleichung $\mathbf{n} \cdot x = d,$ d.h. Normalenvektor $\mathbf{n}$ und d.
Lösung : $\mathbf{n} = (\mathbf{p}_2 - \mathbf{p}_1) \times (\mathbf{p}_3 - \mathbf{p}_1)$ und $d = \mathbf{n} \cdot \mathbf{p}_1.$
Das folgende Unterprogramm `plane_equ` berechnet $\mathbf{n}$ und d. Es setzt die boolsche Variable `error` auf `true`, falls $\mathbf{n} \approx \mathbf{0}$.

```
procedure plane_equ(p1,p2,p3:vt3d; var nv:vt3d; var d:real; var error:boolean);
{Berechnet die Gleichung nv*x=d der Ebene durch die Punkte p1,p2,p3.
 error=true: die Punkte spannen keine Ebene auf. }
```

### 2.3.4 Schnitt Gerade-Ebene

Gegeben: Gerade $\mathbf{x}(t) = \mathbf{p} + t\mathbf{r}, \quad$ Ebene $\mathbf{n} \cdot \mathbf{x} = d.$
Gesucht: Schnittpunkt $\mathbf{p}_{is}$ der Gerade mit der Ebene.
Lösung: $\mathbf{p}_{is} = \mathbf{p} - ((\mathbf{n} \cdot \mathbf{p} - d)/\mathbf{n} \cdot \mathbf{r})\mathbf{r}.$
Das Unterprogramm `is_line_plane` berechnet den Schnittpunkt, falls er existiert.

```
procedure is_line_plane(p,rv,nv:vt3d; d:real; var pis:vt3d; var nis:integer);
{Schnitt Gerade-Ebene. Gerade: Punkt p, Richtung r. Ebene:  nv*x = d .
 nis=0: kein Schnitt ,nis=1: Schnittpunkt, nis=2: Gerade liegt in der Ebene.}
```

### 2.3.5 Schnitt dreier Ebenen

Gegeben: Drei Ebenen $\varepsilon_i : \quad \mathbf{n}_i \cdot \mathbf{x} = d_i, \quad i = 1, 2, 3, \quad \mathbf{n}_1, \mathbf{n}_2, \mathbf{n}_3$ linear unabhängig.
Gesucht: Schnittpunkt $\mathbf{p}_{is} : \varepsilon_1 \cap \varepsilon_2 \cap \varepsilon_3.$
Der Ansatz $\mathbf{p}_{is} = \xi(\mathbf{n}_2 \times \mathbf{n}_3) + \eta(\mathbf{n}_3 \times \mathbf{n}_1) + \zeta(\mathbf{n}_1 \times \mathbf{n}_2)$
führt auf die Lösung
$\mathbf{p}_{is} = (d_1(\mathbf{n}_2 \times \mathbf{n}_3) + d_2(\mathbf{n}_3 \times \mathbf{n}_1) + d_3(\mathbf{n}_1 \times \mathbf{n}_2))/\mathbf{n}_1 \cdot (\mathbf{n}_2 \times \mathbf{n}_3).$
(Falls die Normalen nicht linear unabhängig sind, existiert eine Schnittgerade oder zwei Ebenen sind parallel.)
Das Unterprogramm `is_3_planes` berechnet den Schnittpunkt. Es liefert `error= true`, falls der Schnitt nicht aus einem Punkt besteht.

```
procedure is_3_planes(nv1:vt3d; d1:real; nv2:vt3d; d2:real; nv3:vt3d; d3:real;
                                              var pis:vt3d; var error:boolean);
{Schnitt der Ebenen nv1*x=d1, nv2*x=d2, nv3*x=d3.
 error= true: Schnitt besteht nicht aus einem Punkt.}
```

### 2.3.6 Schnitt zweier Ebenen

Gegeben: Zwei Ebenen $\varepsilon_i : \quad \mathbf{n}_i \cdot \mathbf{x} = d_i, \quad i = 1, 2, \quad \mathbf{n}_1, \mathbf{n}_2$ linear unabhängig.
Gresucht: $\varepsilon_1 \cap \varepsilon_2 : \mathbf{x} = \mathbf{p} + t\mathbf{r}.$
Die Richtung der Schnittgerade ist $\mathbf{r} = \mathbf{n}_1 \times \mathbf{n}_2$. Einen Punkt $P : \mathbf{p}$ der Schnittgerade erhält man, indem man die Ebenen $\varepsilon_1, \varepsilon_2$ mit der Ebene $\varepsilon_3 : \mathbf{x} = s_1\mathbf{n}_1 + s_2\mathbf{n}_2$ schneidet. $s_1$ und $s_2$ ergeben sich durch Einsetzen in die Gleichungen der Ebenen $\varepsilon_1$ und $\varepsilon_2$.

$$P: \ \mathbf{p} = \frac{d_1{\mathbf{n}_2}^2 - d_2(\mathbf{n}_1 \cdot \mathbf{n}_2)}{{\mathbf{n}_1}^2{\mathbf{n}_2}^2 - (\mathbf{n}_1 \cdot \mathbf{n}_2)^2}\mathbf{n}_1 + \frac{d_2{\mathbf{n}_1}^2 - d_1(\mathbf{n}_1 \cdot \mathbf{n}_2)}{{\mathbf{n}_1}^2{\mathbf{n}_2}^2 - (\mathbf{n}_1 \cdot \mathbf{n}_2)^2}\mathbf{n}_2$$

Das Unterprogramm `is_plane_plane` berechnet den Richtungsvektor $\mathbf{r}$ und einen Punkt P der Schnittgerade. Es liefert error= true, falls die Ebenen parallel sind.

```
procedure is_plane_plane(nv1:vt3d; d1:real; nv2:vt3d; d2:real;
                         var p,rv:vt3d; var error:boolean);
{Schnitt der Ebenen nv1*x=d1, nv2*x=d2. Schnittgerade: x = p + t*rv .
 error= true: Schnitt besteht nicht aus einer Gerade.}
```

### 2.3.7 $\xi$-$\eta$-Koordinaten eines Punktes in einer Ebene

Gegeben: Ebene $\varepsilon : \mathbf{x} = \mathbf{p}_0 + \xi\mathbf{v}_1 + \eta\mathbf{v}_2$ und Punkt P: $\mathbf{p}$ in $\varepsilon$ .
Gesucht: $\xi, \eta$ so, daß $\mathbf{p} = \mathbf{p}_0 + \xi\mathbf{v}_1 + \eta\mathbf{v}_2$ ist.
Durch skalare Multiplikation des Ansatzes für $\mathbf{p}$ mit den Vektoren $\mathbf{v}_1, \mathbf{v}_2$ erhält man das lineare Gleichungssystem
$(\mathbf{p} - \mathbf{p}_0) \cdot \mathbf{v}_1 = \xi {\mathbf{v}_1}^2 + \eta\mathbf{v}_1 \cdot \mathbf{v}_2, \qquad (\mathbf{p} - \mathbf{p}_0) \cdot \mathbf{v}_2 = \xi\mathbf{v}_1 \cdot \mathbf{v}_2 + \eta {\mathbf{v}_2}^2,$
Die Prozedur `ptco_plane3d` berechnet $\xi, \eta$ mit Hilfe der CRAMERschen Regel. Es setzt `error=true`, falls die Determinante des Gleichungssystems $\approx 0$ ist. (Liegt der Punkt $P$ nicht in $\varepsilon$ und ist $\xi, \eta$ die Lösung des obigen Gleichungssystems, so ist P': $\mathbf{p}' = \mathbf{p}_0 + \xi\mathbf{v}_1 + \eta\mathbf{v}_2$ der Fußpunkt des Lotes von $P$ auf die Ebene $\varepsilon$.)

```
procedure ptco_plane3d(p0,v1,v2,p:vt3d; var xi,eta:real; var error:boolean);
{v1,v2 sind linear unabhaengig, p-p0 linear abhaengig von v1,v2.
 Es werden Zahlen  xi,eta berechnet mit  p = p0 + xi*v1 + eta*v2.}
```

### 2.3.8 Koordinaten in einem neuen 3D-Kordinatensystem

Gegeben: Neuer Nullpunkt $B_0 : \mathbf{b_0}$, neue Basisvektoren $\mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3$ und Punkt P: $\mathbf{p}$ .
Gesucht: $\xi, \eta, \zeta$ so, daß $\mathbf{p} = \mathbf{b}_0 + \xi\mathbf{b}_1 + \eta\mathbf{b}_2 + \zeta\mathbf{b}_3$ ist.

$$\begin{aligned}\xi &= \det(\mathbf{p} - \mathbf{b}_0, \mathbf{b}_2, \mathbf{b}_3) / \det(\mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3)\\ \eta &= \det(\mathbf{b}_1, \mathbf{p} - \mathbf{b}_0, \mathbf{b}_3) / \det(\mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3)\\ \zeta &= \det(\mathbf{b}_1, \mathbf{b}_2, \mathbf{p} - \mathbf{b}_0) / \det(\mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3)\end{aligned}$$

```
procedure newcoordinates3d(p,b0,b1,b2,b3: vt3d; var pnew: vt3d);
{Berechnet die Koordinaten von p bzgl. der Basis b1,b2,b3 mit Nullpkt. b0.}
```

## 2.4 Numerik: GAUSS-, NEWTON-Verfahren

Zum Lösen von einem größeren **linearen Gleichungssystem** $A\mathbf{x} = \mathbf{b}$ verwendet man eine geignete Variationen des GAUSS-Algorithmus. Ein PASCAL-Programm hierzu ist z.B. in dem Buch *Meyberg,Vachenauer: Höhere Mathematik 1, Springer-Verlag, 1999,* abgedruckt und beschrieben.

Zum Lösen eines **nicht linearen Gleichungssystems** kann man das NEWTON-Verfahren verwenden. Hier eine kurze Beschreibung:

Gegeben: Funktion $\mathbf{F} : D \to \mathbb{R}^n, D \subseteq \mathbb{R}^n$, und ein Startpunkt $\mathbf{x}_0$ für die Iteration.
Gesucht: Ein Punkt $\mathbf{x}^*$ in der „Nähe“ von $\mathbf{x}_0$ mit $\mathbf{F}(\mathbf{x}^*) = 0$.

**Algorithmus:**
Für $\nu = 0, 1, 2, \ldots$

(1) löse man das lineare Gleichungssystem
$\mathbf{F}'(\mathbf{x}_\nu)\mathbf{d}_\nu = -\mathbf{F}(\mathbf{x}_\nu)$ , wobei $\mathbf{F}' := (\frac{\partial F_i}{\partial x_k})$ und $\mathbf{F} = (F_1, F_2, \ldots, F_n)$, $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ ist.

(2) Setze $\mathbf{x}_{\nu+1} = \mathbf{x}_\nu + \mathbf{d}_\nu$

(3) Falls $\|\mathbf{x}_{\nu+1} - \mathbf{x}_\nu\|$ klein „genug“ (oder andere Abbruchbedingung) setze $\mathbf{x}^* = \mathbf{x}_{\nu+1}$.

# Kapitel 3

# PARALLEL/ZENTRAL–PROJEKTION

## 3.1 Senkrechte Parallelprojektion

In der klassischen Darstellenden Geometrie unterscheidet man zwei Arten von Parallelprojektionen:
a) senkrechte Parallelprojektion b) schiefe Parallelprojektion,
je nachdem, ob die Projektionsstrahlen senkrecht oder schief (nicht senkrecht) zur Bildtafel stehen. Zwar liefern schiefe Parallelprojektionen nicht so gute Bilder wie senkrechte Parallelprojektionen, aber in Form der Kavalier- und Vogelperspektiven lassen sich in vielen Situationen schnell anschauliche Bilder erstellen. Einem Rechner ist es allerdings gleichgültig, ob er eine senkrechte oder schiefe Projektion berechnet. Deshalb werden wir hier nur senkrechte Parallelprojektionen behandeln.

### 3.1.1 Die Projektionsformeln

Um die Parallelprojektion rechnerisch erfassen zu können, führen wir im Raum ein zur Beschreibung des abzubildenden Gegenstandes geeignetes rechtwinkliges Koordinatensystem $(O; x, y, z)$ ein. Die Ebene (Bildtafel), auf die senkrecht projiziert werden soll, nennen wir $\varepsilon_0$. Da eine Verschiebung der Bildtafel an dem Bild des Gegenstandes (außer seine Lage) nichts ändert, können wir annehmen, daß $\varepsilon_0$ den Nullpunkt $O$ des Koordinatensystems enthält. Die Lage der Ebene $\varepsilon_0$ und damit die senkrechte Parallelprojektion ist durch die Angabe eines Normalenvektors $\mathbf{n}_0$ von $\varepsilon_0$ eindeutig bestimmt. Wir wählen den Vektor $\mathbf{n}_0$ so, daß er die Länge 1 ($|\mathbf{n}_0| = 1$) hat und der Projektionsrichtung entgegengesetzt ist ($\mathbf{n}_0$ zeigt zur "Sonne"). Beschreibt man $\mathbf{n}_0$ durch seine Kugelkoordinatenwinkel $u, v$ ($u$ ist die "geographische Länge", $v$ die "geographische Breite"), so gilt:

$$\mathbf{n}_0 = (\cos u \cos v, \sin u \cos v, \sin v), \quad 0 \le u \le 2\pi,\ -\pi/2 \le v \le \pi/2.$$

Zur Beschreibung der Bildpunkte verwenden wir ein rechtwinkliges Koordinatensystem $(O; x_e, y_e)$ in der Bildtafel $\varepsilon_0$, dessen Nullpunkt $O$ mit $O$ übereinstimmt und dessen $y_e$-Achse im Falle $|v| < \pi/2$ das Bild der z-Achse ist. Die $x_e$-Achse liegt dann in der Schnittgerade von $\varepsilon_0$ mit der x-y-Ebene. Die Vektoren

$$\mathbf{e}_1 = (-\sin u, \cos u, 0), \qquad \mathbf{e}_2 = (-\cos u \sin v, -\sin u \sin v, \cos v)$$

bilden eine Orthonormalbasis in $\varepsilon_0$ und $\{\mathbf{e}_1, \mathbf{e}_2, \mathbf{n}_0\}$ ist eine Orthonormalbasis des $\mathbb{R}^3$.
Um die Bildkoordinaten $(x_e, y_e)$ eines Punktes $Q : \mathbf{q} = (x, y, z)$ zu erhalten, muß man also nur die ersten beiden Koordinaten von Q bezüglich der Basis $\{\mathbf{e}_1, \mathbf{e}_2, \mathbf{n}_0\}$ bestimmen:

$$\begin{aligned} x_e &= \mathbf{e}_1 \cdot \mathbf{q} = -x \sin u + y \cos u \\ y_e &= \mathbf{e}_2 \cdot \mathbf{q} = -(x \cos u + y \sin u) \sin v + z \cos v. \end{aligned}$$

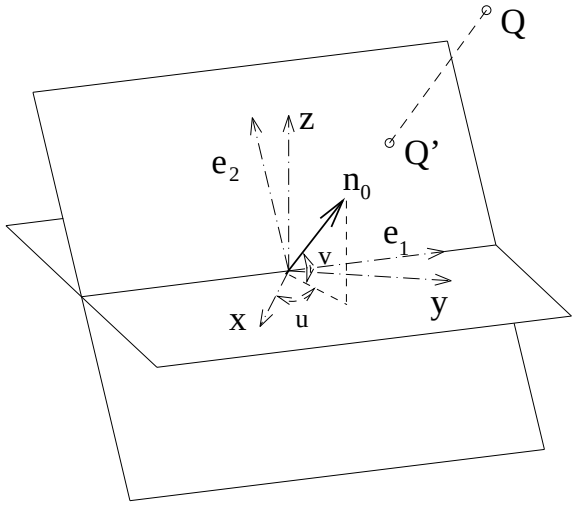



Abbildung 3.1: Parallelprojektion eines Punktes $Q$

Eine senkrechte Parallelprojektion ist also eine lineare Abbildung. Die Koeffizienten der zugehörigen Abbildungsmatrix ergeben sich aus den Projektionsformeln.

### 3.1.2 Prozeduren zur senkrechten Parallelprojektion

Da die Zahlen $\sin u, \cos u, \sin v, \cos v$ und der Normalenvektor $\mathbf{n}_0$ der Bildtafel für eine bestimmte Parallelprojektion oft gebraucht werden, werden wir sie in dem Unterprogramm `init_parallel_projection` nach dem Einlesen der Winkel u, v berechnen und über globale Variablen allen anderen Unterprogrammen zur Verfügung stellen. Die weiteren, unten aufgeführten, Unterprogramme werden durch Kommentare erläutert. All diese Programme sind in der Datei `proc_pp.pas` enthalten.

```
procedure init_parallel_projection;
 begin
   writeln('*** PARALLEL-PROJEKTION ***');
   writeln;
   writeln('Projektionswinkel u, v ? (in Grad)');
   readln(u_angle,v_angle);
   rad_u:= u_angle*pi/180;      rad_v:= v_angle*pi/180;
   sin_u:= sin(rad_u)    ;      cos_u:= cos(rad_u)    ;
   sin_v:= sin(rad_v)    ;      cos_v:= cos(rad_v)    ;
{Normalen-Vektor der Bildebene:}
   n0vt.x:= cos_u*cos_v;  n0vt.y:= sin_u*cos_v;  n0vt.z:= sin_v;
 end; { init_parallel_projection }
{**************}
procedure pp_vt3d_vt2d(p:vt3d; var pp:vt2d);
 {Berechnet das Bild eines Punktes}
{*************}
procedure pp_point(p:vt3d; style:integer);
 {Projiziert einen Punkt und markiert ihn gemaess style}
{*************}
procedure pp_line(p1,p2:vt3d ; style:integer);
 {Projiziert die Strecke p1,p2 gemaess style}
{*************}
procedure pp_arrow(p1,p2:vt3d; style:integer);
 {Projiziert einen Pfeil}
```

```
{*************}
procedure pp_axes(al:real);
 {Projiziert die Koordinatenachsen, al:Achsenlaenge}
{*************}
procedure pp_vts3d_vts2d(var p:vts3d; n1,n2:integer; var pp:vts2d);
 {Berechnet die Bilder pp einer Punktreihe p.}
{*************}
procedure pp_curve(var p:vts3d; n1,n2,style:integer);
 {Projiziert das 3d-Polygon p[n1]...p[n2]}
{*************}
```

## 3.2 Zentralprojektion

Ein höheres Maß an Anschaulichkeit erreicht man durch Darstellung eines Gegenstandes in Zentralprojektion. Man verwendet dabei Strahlen, die von einem festen Punkt $Z$, dem Zentrum oder Augpunkt, ausgehen. Den Gewinn an Anschaulichkeit muß man allerdings i.a. durch einen Verlust an Maßgenauigkeit erkaufen. Ein typischer Unterschied zur Parallelprojektion besteht darin, daß parallele Geraden i.a. in einer Zentralprojektion nicht mehr parallel sind, sondern durch einen Punkt, dem Fluchtpunkt des Parallelbüschels gehen (s. Abb. 3.2).

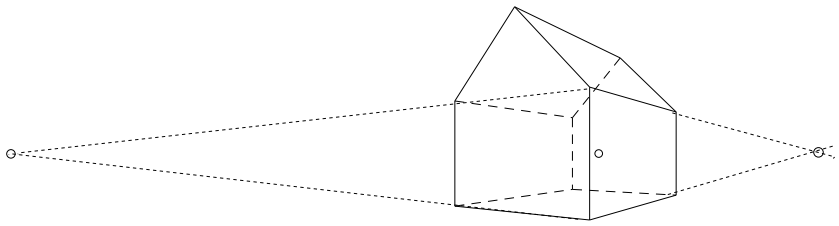


Abbildung 3.2: Haus in Zentralprojektion

Wir werden sehen, daß sich die meisten Programme , die wir für Parallelprojktion geschrieben haben, leicht für Zentralprojektion abändern lassen.

### 3.2.1 Die Projektionsformeln

Sei $\varepsilon$ eine Ebene und $Z : \mathbf{z}$ ein nicht in $\varepsilon$ gelegener Punkt des $\mathbb{R}^3$. Die Abbildung $\Phi$, die einem beliebigen Punkt $P : \mathbf{p}$ den Schnittpunkt $P'$ der Geraden $\overline{ZP}$ mit der Ebene $\varepsilon$ zuordnet, falls dieser existiert, heißt **Zentralprojektion von $Z$ auf $\varepsilon$**.

$Z$ heißt das **Zentrum** oder der **Augpunkt** der Zentralprojektion $\Phi$ (s. Abb. 3.3). Der Lotfußpunkt $H : \mathbf{h}$ des Lotes von $Z$ auf die Bildtafel $\varepsilon_0$ heißt **Hauptpunkt** von $\Phi$. Ist
$\mathbf{n}_0 := (\cos u \cos v, \sin u \cos v, \sin v), u \in [0, 2\pi], v \in [-\pi/2, \pi/2]$,
die Normale von $\varepsilon_0$, so hat $\varepsilon_0$ die Gleichung: $(\mathbf{x} - \mathbf{h}) \cdot \mathbf{n}_0 = 0$. Für das Zentrum $Z : \mathbf{z}$ und den Hauptpunkt $H$ gilt $\mathbf{z} = \mathbf{z} + \delta\mathbf{n}_0$ mit $\delta > 0$. Der Abstand $\delta$ des Zentrums $Z$ zur Ebene $\varepsilon_0$ heißt die **Distanz** von $\Phi$. Alle Punkte des $\mathbb{R}^3$, die durch $\Phi$ nicht abgebildet werden können, liegen in der zu $\varepsilon_0$ parallelen Ebene $\varepsilon_v$ durch den Augpunkt $Z$.
$\varepsilon_v$ heißt die **Verschwindungsebene** von $\Phi$.

Die Zentralprojektion $\Phi$ ist durch die Vorgabe der Parameter $u, v$, des Hauptpunktes $H$ und der Distanz $\delta$ eindeutig bestimmt. Ein Punkt $P : \mathbf{p}$, der nicht in der Verschwindungsebene $\varepsilon_v$ liegt, wird auf den Punkt $P' := \overline{ZP} \cap \varepsilon_0$ abgebildet. Führt man diesen Schnitt des Projektionsstrahls mit der

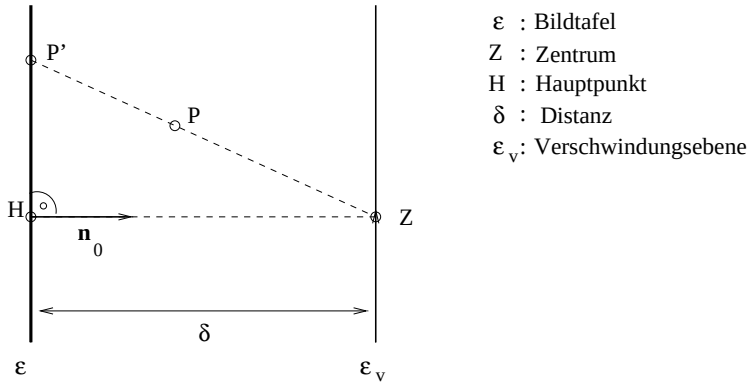



Abbildung 3.3: Hauptpunkt, Distanz und Verschwindungsebene

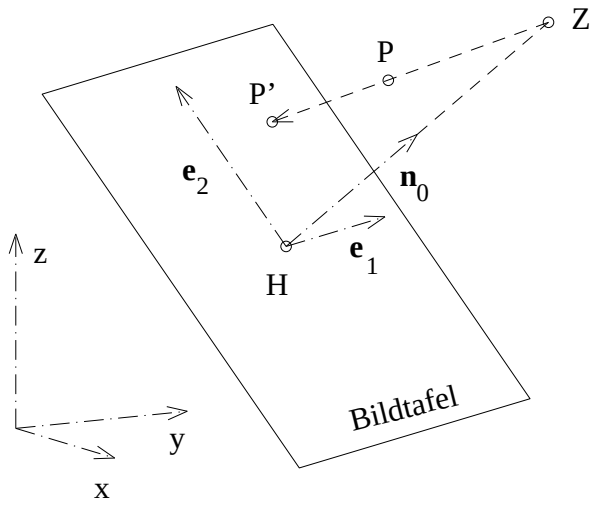



Abbildung 3.4: Zentralprojektion eines Punktes

Bildtafel aus, so ergibt sich

$$P' : \mathbf{p}' = \mathbf{z} + \frac{(\mathbf{h}-\mathbf{z})\cdot\mathbf{n}_0}{(\mathbf{p}-\mathbf{z})\cdot\mathbf{n}_0}(\mathbf{p}-\mathbf{z}) = \mathbf{h} + \delta\mathbf{n}_0 + \frac{\delta}{\delta-(\mathbf{p}-\mathbf{h})\cdot\mathbf{n}_0}(\mathbf{p}-\mathbf{h}-\delta\mathbf{n}_0)$$

Da $P'$ in $\varepsilon_0$ liegt, gilt ferner $(\mathbf{p}'-\mathbf{h})\cdot\mathbf{n}_0 = 0$.

In der Ebene $\varepsilon_0$ führen wir jetzt so Koordinaten ein, daß $H$ der Nullpunkt ist und (analog zur senkrechten Parallelprojektion)

$$\mathbf{e}_1 := (-\sin u, \cos u, 0), \quad \mathbf{e}_2 := (-\cos u \sin v, -\sin u \sin v, \cos v)$$

die Basisvektoren sind. $\{\mathbf{e}_1, \mathbf{e}_2, \mathbf{n}_0\}$ ist eine $ON$-Basis des $\mathbb{R}^3$.(vgl. Abb. 3.4)
Es gibt Zahlen $x_{ez}, y_{ez}$ (die Koordinaten von $P'$ bzgl. der Basis $\{\mathbf{e}_1, \mathbf{e}_2\}$), sodaß $\mathbf{p}' = \mathbf{h} + x_{ez}\mathbf{e}_1 + y_{ez}\mathbf{e}_2$ ist. Mit Hilfe der obigen Darstellung von $\mathbf{p}'$ ergibt sich (unter Beachtung der Orthogonalität von $\mathbf{e}_1, \mathbf{e}_2, \mathbf{n}_0$)

$$x_{ez} = \frac{\mathbf{e}_1\cdot(\mathbf{p}-\mathbf{h})}{1-(\mathbf{p}-\mathbf{h})\cdot\mathbf{n}_0/\delta} \qquad y_{ez} = \frac{\mathbf{e}_2\cdot(\mathbf{p}-\mathbf{h})}{1-(\mathbf{p}-\mathbf{h})\cdot\mathbf{n}_0/\delta}$$

(Für $\mathbf{h} = 0$ und $\delta \to \infty$ ergeben sich die Formeln für die senkrechte Parallelprojektion !) Bei der Ausführung der Projektion ist es üblich, nur solche Punkte zu projizieren , die "vor" der Verschwindungsebene $\varepsilon_v$ liegen, die also der Bedingung $(\mathbf{p}-\mathbf{h})\cdot\mathbf{n}_0 < \delta$ genügen. Der Einfachheit halber wollen wir hier auch nur Strecken und Kurven projizieren, die **vollständig** "vor" der Verschwindungsebene liegen.

### 3.2.2 Prozeduren zur Zentralprojektion

Wir geben jetzt die für die Zentralprojektion wesentlichen Unterprogramme für die Projektion von Punkten, Strecken, Kurven,... an. Dabei beachte man, daß der Hauptpunkt `mainpt`, das Zentrum `centre`, die Normale `n0vt` und die Distanz `distance` globale Variablen sind und in der Datei `geovar`.pas enthalten sind. Die Zentralprojektion eines Punktes führen wir formal in zwei Schritten durch:

(1) Koordinatentransformation in das System $(H; \mathbf{e}_1, \mathbf{e}_2, \mathbf{n}_0)$ mit dem Nullpunkt $H$ und der Basis $\{\mathbf{e}_1, \mathbf{e}_2, \mathbf{n}_0\}$.

$$\mathbf{p} = (x, y, z) \to \bar{\mathbf{p}} = (\bar{x}, \bar{y}, \bar{z}) \text{ mit } \bar{x} = (\mathbf{p}-\mathbf{h})\cdot\mathbf{e}_1,\ \bar{y} = (\mathbf{p}-\mathbf{h})\cdot\mathbf{e}_2,\ \bar{z} = (\mathbf{p}-\mathbf{h})\cdot\mathbf{n}_0,$$

(2) Zentralprojektion in dem System $(H; \mathbf{e}_1, \mathbf{e}_2, \mathbf{n}_0)$ auf die $\bar{x}-\bar{y}$- Ebene:

$$\bar{\mathbf{p}} = (\bar{x}, \bar{y}, \bar{z}) \to \left(\frac{\bar{x}}{1-\bar{z}/\delta},\ \frac{\bar{y}}{1-\bar{z}/\delta}\right)$$

```
procedure init_centralparallel_projection(ind : integer);
 begin
   if ind=1 then
              begin
              writeln('*** ZENTRAL-PROJEKTION ***');
              writeln('Hauptpunkt ?');  readln(mainpt.x,mainpt.y,mainpt.z);
              writeln('Distanz ?');     readln(distance);
              end
             else
              begin
              writeln('*** PARALLEL-Projektion ***');
              mainpt:= null3d;   distance:= 1000000000;
              end;
   writeln('Projektionswinkel u, v ? (in Grad)');  readln(u_angle,v_angle);
   rad_u:= u_angle*pi/180;        rad_v:= v_angle*pi/180;
   sin_u:= sin(rad_u);            cos_u:= cos(rad_u);
   sin_v:= sin(rad_v);            cos_v:= cos(rad_v);
{Basis e1,e2 und Normale n0 der Bildebene:}
   e1vt.x:= -sin_u;         e1vt.y:= cos_u;          e1vt.z:= 0;
   e2vt.x:= -cos_u*sin_v;   e2vt.y:=-sin_u*sin_v;    e2vt.z:= cos_v;
   n0vt.x:= cos_u*cos_v;    n0vt.y:= sin_u*cos_v;    n0vt.z:= sin_v;
{Zentrum:}
   lcomb2vt3d(1,mainpt, distance,n0vt,  centre);
 end; { init_central_projection }
{**************}
procedure transf_to_e1e2n0_base(p : vt3d; var pm : vt3d);
{Berechnet Koordinaten bzgl. System mit Hauptpkt. als Nullpkt. und der
 Basis e1,e2,n0.}
{***************}
procedure cp_vt3d_vt2d(p: vt3d; var pp : vt2d);
 {Zentralprojektion (Koordinaten) eines Punktes}
 var xe,ye,ze,cc : real;   pm : vt3d;
```

```
begin
  diff3d(p,mainpt, pm);
  xe:= scalarp3d(pm,e1vt);     {Koordinaten von p bzgl. dem Koord.-System:}
  ye:= scalarp3d(pm,e2vt);     {Nullpunkt = Hauptpunkt}
  ze = scalarp3d(pm,n0vt);     {und Basis e1,e2,n0}
  cc:= 1-ze/distance;
  if cc>eps6 then begin  pp.x:= xe/cc; pp.y:= ye/cc; end     {Projektion}
             else
             writeln('Punkt liegt in oder hinter der Verschwindungsebene !!');
end;    {cp_vt3d_vt2d}
```

Die folgenden Unterprogramme können wörtlich aus der Parallelprojektion übernommen werden. Man muß nur überall die drei Zeichen `pp_` durch `cp_` ersetzen. Sie sind in der Datei `proc_zp.pas` enthalten.

```
procedure cp_point(p: vt3d; style: integer);
 {markiert einen projizierten Punkt}
{**************}
procedure cp_line(p1,p2 : vt3d ; style : integer);
 {projiziert die Strecke p1 p2}
{**************}
procedure cp_arrow(p1,p2 : vt3d; style : integer);
 {projiziert einen Pfeil}
{**************}
procedure cp_axes(al : real);
 {projiziert die Koordinatenachsen}
{**************}
procedure cp_vts3d_vts2d(p: vts3d; n1,n2 : integer; var pp : vts2d);
 {Koordinaten von projizierten Punkten p[i] , i= n1...n2.}
{*************}
procedure cp_curve(p: vts3d; n1,n2,style : integer);
 {projiziert ein 3D-Polygon}
{*************}
```

**Beispiel:**

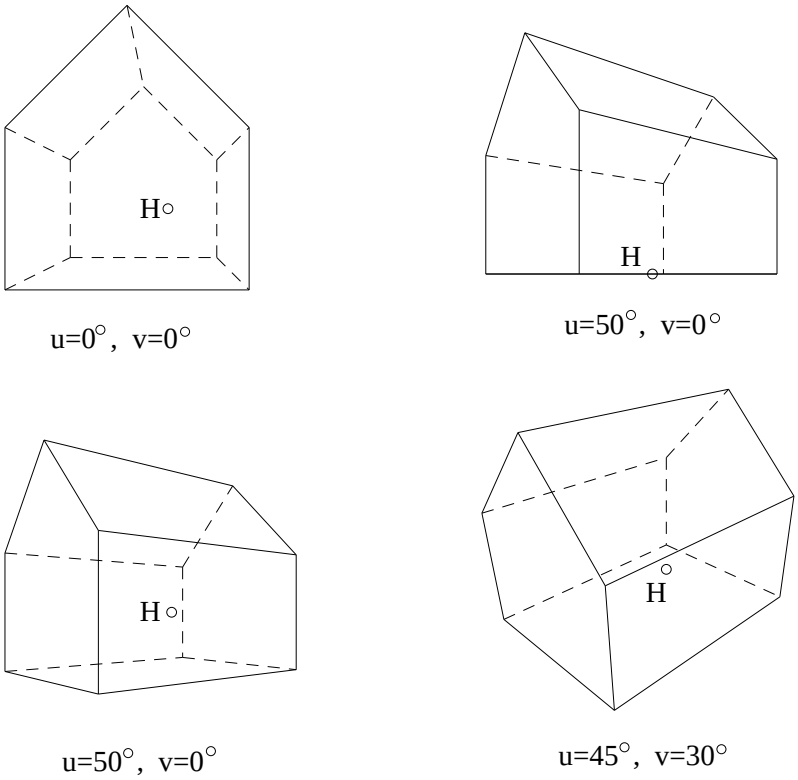



Abbildung 3.5: Zentralprojektionen eines Hauses

# Kapitel 4

# EBENE KURVEN

## 4.1 Parametrisierte Kurven im $\mathbb{R}^2$ und $\mathbb{R}^3$

Unter einer parametrisierten Kurve $\Gamma$ wollen wir eine Kurve verstehen, deren Punkte die Bilder einer Abbildung eines reellen Intervalls $[t_1, t_2]$ in den $\mathbb{R}^2$ bzw. $\mathbb{R}^3$ sind:
$\Gamma = \{\mathbf{c}(t) | t_1 \leq t \leq t_2\}$

**Beispiele:**
a) Ellipse: $(a\cos t, b\sin t)$,
b) Epi- bzw. Hypozykloiden: $\mathbf{c}(t) = (x(t), y(t))$ mit

$$\begin{aligned} x(t) &= (a+b)\cos t - \lambda a \cos((a+b)t/a) \\ y(t) &= (a+b)\sin t - \lambda a \sin((a+b)t/a) \quad b > 0, \quad a+b > 0, \quad \lambda > 0. \end{aligned}$$

Diese Kurven entstehen durch Abrollen eines Kreises mit Radius $a$ auf bzw. in einem größeren Kreis mit Radius $b$ .
Für $\lambda \neq 1$ entstehen *verlängerte* bzw. *verkürzte* Zykloiden.

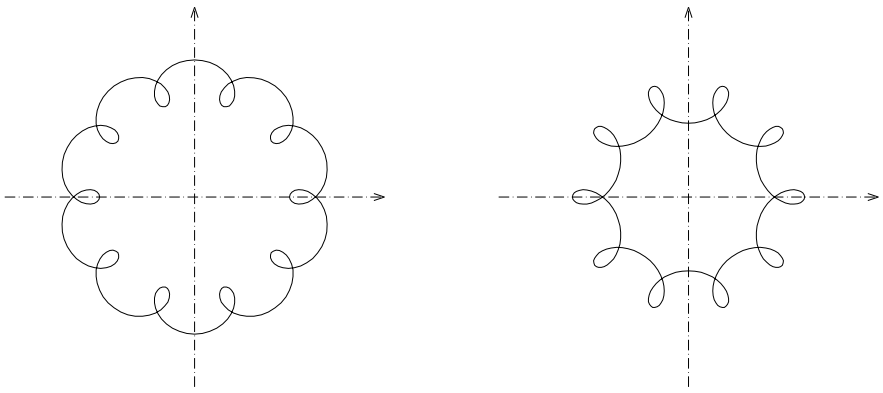


Abbildung 4.1: Zykloiden

Indem man das Parameterintervall in (nicht notwendig gleichlangen) Schritten durchläuft, lassen sich beliebig viele Punkte einer parametrisierten Kurve berechnen und anschließend
a) im Fall einer ebenen Kurve durch einen Polygonzug mit `curve2d` verbinden oder,
b) im Fall einer Kurve im $\mathbb{R}^3$, mit `pp_curve` bzw. `cp_curve` projizieren.

Da der Parameter $t$ i.a. nicht die Bogenlänge ist, können die Abstände benachbarter Punkte stark differieren. Dies läßt sich vermeiden, wenn die Kurve $\mathbf{c}(t)$ differenzierbar ist und man die Schrittweite

im Parameterbereich in Abhängigkeit von $\dot{\mathbf{c}}(t)$ wählt. Soll der Abstand zweier Punkte (ungefähr) $s$ sein, so erhält man aus der Taylorentwicklung von $\mathbf{c}(t)$:

$$\mathbf{c}(t_{i+1}) \approx \mathbf{c}(t_i) + \dot{\mathbf{c}}(t_i)\Delta t_i, \qquad \Delta t_{i+1} = t_{i+1} - t_i,$$

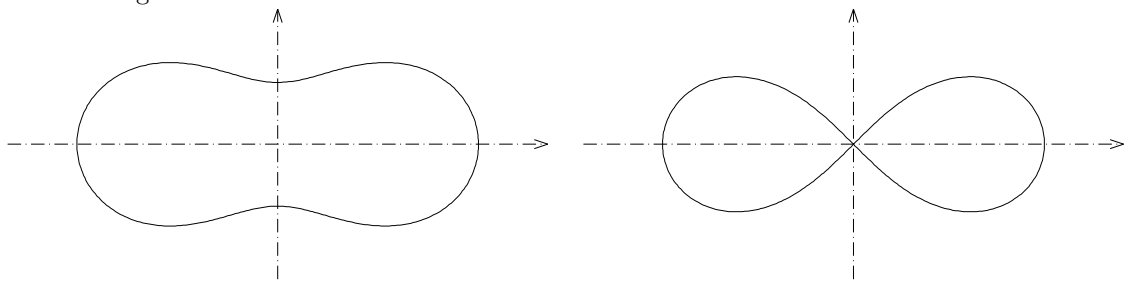
Es ist $\|\mathbf{c}(t_{i+1}) - \mathbf{c}(t_i)\| \approx s$ , wenn man $\Delta t_i = s/\|\dot{\mathbf{c}}(t_i)\|$ setzt, falls $\|\dot{\mathbf{c}}(t_i)\| \neq 0$ ist.
D.h. man erhält damit eine relativ **gleichmäßige Verteilung der** berechneten **Punkte.**

## 4.2 Implizite Kurven

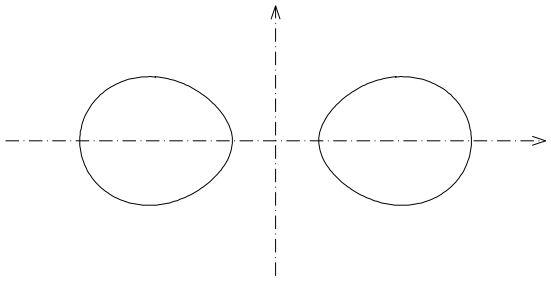
Unter einer impliziten Kurve $\Gamma$ wollen wir eine Punktmenge des $\mathbb{R}^2$ verstehen, die einer Gleichung $f(x, y) = 0$ genügt :

$$\Gamma = \{\mathbf{x} \in D \,|\, f(\mathbf{x}) = 0\} \qquad f : D \to \mathbb{R}, \qquad D \subset \mathbb{R}^2.$$

**Beispiele:**
a) Kreis $x^2 + y^2 - r^2 = 0$, Hyperbel $xy - 1 = 0$
b) Cassini-Kurven $(x^2 + y^2)^2 - 2c^2(x^2 - y^2) - (a^4 - c^4) = 0, \quad a > 0, \quad c > 0.$
Für $a = c$ ergeben sich Lemniskaten.



$a = 1.1, \quad c = 1$ $\qquad$ $a = c = 1$



$a = 1, \quad c = 1.05$

Abbildung 4.2: Cassini-Kurven

Die **Berechnung von Punkten** einer imzpliziten Kurve ist nicht so einfach wie bei parametrisierten Kurven. Es wird hier eine Methode zur Erzeugung von Kurvenpunkten besprochen. Der **Verfolgungsalgorithmus** started mit einem Punkt in der Nähe der Kurve und berechnet sukzessive weitere auf demselben Zweig. Der Algoritmus ist schnell und erzeugt einen zusammenhängenden Polygonzug.
Es sind im wesentlichen zwei Teilprobleme zu lösen:
(1) Finden eines ersten Punktes $P_1$.
(2) Von $P_1$ ausgehend weitere Punkte der Kurve zu finden.

**Idee** des Algorithmus:
Zu (1): Man wählt einen Startpunkt $Q_0 = (x_0, y_0)$ in der Nähe der Kurve. Diesen faßt man als

Grundriß des Punktes $\overline{Q}_0 = (x_0, y_0, f(x_0, y_0))$ auf der Fläche $z = f(x, y)$ auf. Ist $f(x_0, y_0) > 0$ bzw. $< 0$, so „läuft“ man in Richtung des steilsten Abstiegs bzw. Aufstiegs auf die Niveaulinie $f(x, y) = 0$ zu. „Laufen“ bedeutet hier, immer wieder Newton-Schritte auszuführen. Man erhält so den ersten Punkt $P_1$.
Zu (2): Von $P_1$ aus geht man ein Stück entlang der Tangente zu einem neuen Punkt $Q_0$ und wiederholt (1), ... .

Da Teil (1) auch anderweitig genutzt werden kann, schreiben wir hierfür eine eigenständige Prozedur `curvepoint`:

(CP1) Es sei $Q_0 = (x_0, y_0)$ ein Punkt in der Nähe der Kurve.

(CP2) Iteration entlang des steilsten Weges: Es sei $\mathbf{q}_i = (x_i, y_i)$.
$\mathbf{q}_{i+1} = \mathbf{q}_i - \frac{f(\mathbf{q}_i)}{\nabla f(\mathbf{q}_i)^2} \nabla f(\mathbf{q}_i)$.

(CP3) Wiederhole (CP2) bis $\|\mathbf{q}_{i+1} - \mathbf{q}_i\|$ klein genug ist.
(oder andere Abbruchbedingung.)

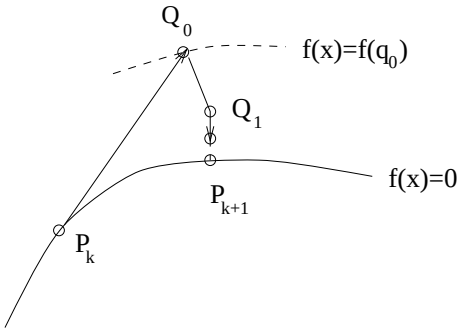



Abbildung 4.3: Berechnung von Punkten einer impliziten Kurve

**Durchführung** des Verfolgungsalgorithmus:

1) Wahl eines geeigneten Startpunktes $Q_0 : \mathbf{q}_0 = (x_0, y_0)$ und einer Schrittweite $s$.

2) Erster Kurvenpunkt ist $P_1 : \mathbf{p}_1 = \texttt{curvepoint}(\mathbf{q}_0)$.

3) Kurvenpunkt $P_{k+1} : \mathbf{p}_{k+1}$ aus $P_k : \mathbf{p}_k$:
$\mathbf{p}_{k+1} = \texttt{curvepoint}(\mathbf{p}_k + s\, \mathbf{t}_k)$, wobei $\mathbf{t}_k = (-f_y(\mathbf{p}_k), f_x(\mathbf{p}_k))/\| \cdots \|$ Einheitstangente im Punkt $P_k$ ist.

4) Führe 3) sooft durch, bis die gewünschte Anzahl von Punkten berechnet ist
oder $P_{k+1} \approx P_0$ (geschlossene Kurve)
oder $\cdots$ (andere Abbruchbedingung)

**Bemerkung:**
Da der Verfolgungsalgorithmus relativ robust ist, läßt er sich auch auf Kurven mit einzelnen Singularitäten anwenden. Beim Überschreiten einer Singularität kann allerdings eine Richtungsumkehr stattfinden. Um zu verhindern, daß der Algorithmus "hängen bleibt", sollte man in Schritt 3) zunächst die Richtung von $\mathbf{t}_k$ überprüfen:
Falls $\mathbf{t}_k \cdot (\mathbf{p}_k - \mathbf{p}_{k-1}) < 0$ ist, ersetze $\mathbf{t}_k$ durch $-\mathbf{t}_k$.
(Siehe Beispiel-Programm `cassini.p`.)

## 4.3 Bézier-Kurven

Da Bézier-Kurven in CAD eine große Rolle spielen und wir diese im Kapitel über Rotationsflächen verwenden wollen, sei hier eine kleine Einführung gegeben. Für weitergehende Informationen zu diesem Thema sei der Leser z. B. auf die Bücher von FARIN (FA'90) oder HOSCHEK/LASSER (HO,LA '89) verwiesen.

Numerisch einfache Kurven in der Ebene sind solche, die mit Hilfe einer Parameterdarstellung $\mathbf{x}(t) = (x(t), y(t)),\ t_1 \le t \le t_2$, wobei $x(t)$ und $y(t)$ Polynome in $t$ sind, beschrieben werden. Ist $x(t) := a_0 + a_1 t + a_2 t^2 + \cdots + a_n t^n$ und $y(t) := b_0 + b_1 t + b_2 t + \cdots + b_n t^n$, so ist

$$\begin{aligned} \mathbf{x}(t) &= (a_0, b_0) + (a_1, b_1)t + \cdots + (a_n, b_n)t^n. \\ &= \mathbf{a}_0 + \mathbf{a}_1 t + \cdots + \mathbf{a}_n t^n \end{aligned}$$

mit den Vektoren $\mathbf{a}_i := (a_i, b_i)$.
I.a. sagen die Punkte $A_i : \mathbf{a}_i$ nicht viel über den Kurvenverlauf aus. Dies ändert sich, wenn man die Polynome $x(t), y(t)$ nicht in der „Monom-Basis" $\{1, t, t^2, \cdots, t^n\}$ sondern in der folgenden **Bernsteinbasis** $\{B_0^n(t), B_1^n(t), \cdots, B_n^n(t)\}$ darstellt:

$$B_i^n(t) := \binom{n}{i} t^i (1-t)^{n-i}, \quad 0 \le i \le n.$$

Es sei nun $n > 0$ festgewählt und die Vektoren $\mathbf{b}_0, \mathbf{b}_1, \cdots, \mathbf{b}_n$ beschreiben ein Polygon. Dann ist $\mathbf{x}(t) := \mathbf{b}_0 B_0^n(t) + \mathbf{b}_1 B_1^n(t) + \cdots + \mathbf{b}_n B_n^n(t), \quad 0 \le t \le 1$, eine **Bézier-Kurve** vom (maximalen)Grad $n$. Die Punkte $\mathbf{b}_0, \cdots, \mathbf{b}_n$ heißen **Kontrollpunkte** der Bézierkurve.

**Eigenschaften der Bernstein-Polynome:**

(1) $B_0^n(t) + B_1^n(t) + \cdots + B_n^n(t) = 1$,

(2) $B_0^n(0) = 1,\ B_i^n(0) = 0$ für $i > 0, \quad B_n^n(1) = 1,\ B_i^n(1) = 0$ für $i < n$.

(3) Das Bernstein-Polynom $B_i^n$ hat genau ein Maximum und zwar an der Stelle $t = i/n$. D.h. eine leichte Veränderung des Punktes $\mathbf{b}_i$ hat nur eine wesentliche Veränderung der Kurve in der Umgebung von $\mathbf{x}(i/n)$ zur Folge.

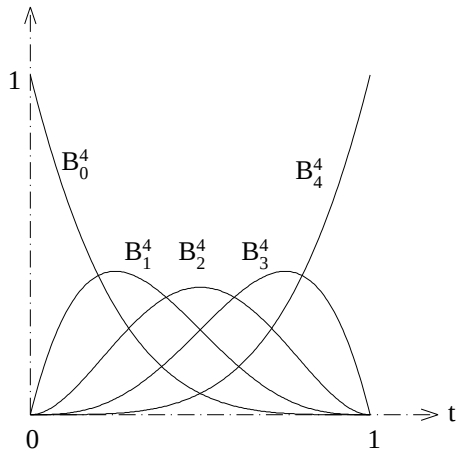



Abbildung 4.4: Bernsteinpolynome $B_i^4$

**Eigenschaften einer Bézier-Kurve:**

(1) $\mathbf{b}_0$ ist der Anfangs- , $\mathbf{b}_n$ der Endpunkt

(2) $\mathbf{b}_1 - \mathbf{b}_0$ ist die Richtung der Tangente im Punkt $\mathbf{b}_0 = \mathbf{x}(0)$, $\mathbf{b}_n - \mathbf{b}_{n-1}$ ist die Richtung der Tangente im Punkt $\mathbf{b}_n = \mathbf{x}(1)$.

(3) Das Polygon $\mathbf{b}_0, \mathbf{b}_1, \cdots, \mathbf{b}_n$ gibt einen ungefähren Verlauf der Kurve an.

Da die Komponenten $x(t), y(t)$ einer Bézierkurve reelle Linearkombinationen von Bernstein-Polynomen sind, können wir zur Berechnung von Punkten einer Bézierkurve ein Unterprogramm verwenden, das solche Linearkombinationen berechnet. Wir werden hier das in FARIN: Kurven und Flächen ..., 1990, S. 48, angegebene Unterprogramm `bezier_comp` (dort mit `hornbez` bezeichnet) verwenden. Es berechnet nach einer Art Horner-Schema den Ausdruck

$$a_0 B_0^n(t) + a_1 B_1^n(t) + \cdots + a_n B_n^n(t), \qquad a_i \in \mathbb{R},$$

bei Vorgabe des Grades $n$ , der Koeffizienten $a_0, \cdots, a_n$ und des Parameters $t$ (siehe Beispiel-Programm `bezkur.p`).

```
function bezier_comp(degree: integer; coeff : r_array; t: real) : real;
{Berechnet eine Komponente einer Bezier-Kurve. (Aus FARIN: Kurven u. Flaechen...)}
 var i,n_choose_i : integer;    fact,t1,aux : real;
 begin
   t1:= 1-t;  fact:=1;  n_choose_i:= 1;
   aux:= coeff[0]*t1;
   for i:= 1 to degree-1 do
       begin
       fact:= fact*t;
       n_choose_i:= n_choose_i*(degree-i+1) div i;
       aux:= (aux + fact*n_choose_i*coeff[i])*t1;
       end;
   aux:= aux + fact*t*coeff[degree] ;
   bezier_comp:= aux;
 end;   bezier_comp
{*************}
```

**Bemerkung:**
Das Unterprogramm `bezier_comp` kann man natürlich auch zum Berechnen von Punkten einer Bézier-Kurve $(x(t), y(t), z(t))$ im Raum verwenden.

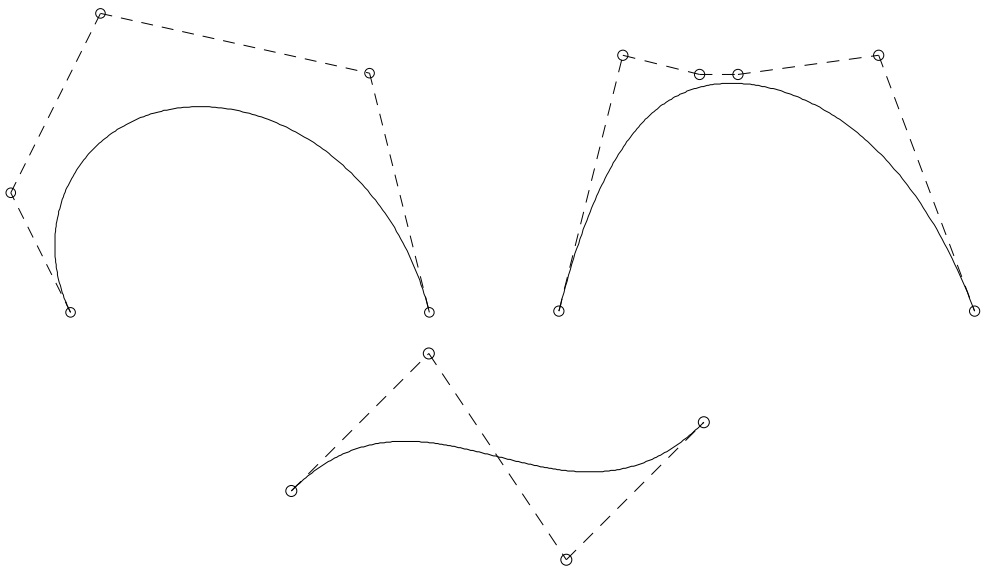


Abbildung 4.5: Bézierkurven mit ihren Kontrollpolygonen

# Kapitel 5

# HIDDENLINE-ALGORITHMUS FÜR NICHTKONVEXE POLYEDER, DARSTELLUNG PARAMETRISIERTER FLÄCHEN

## 5.1 Der Hiddenline-Algorithmus

**Voraussetzungen** und **Vorgaben**:

Gegeben: a) Strecken (Kanten) $\{e_1, e_2, e_3, \ldots\}$
b) **ebene, konvexe** $N-$**Ecke** (Facetten) $\{f_1, f_2, f_3, \ldots\}$
Falls gewisse $N-$Ecke nicht konvex sind, zerlegt man sie in mehrere konvexe Teile. Die hierzu benötigten zusätzlichen Kanten ignoriert man dann beim Zeichnen. Die Kanten, die auf Sichtbarkeit untersucht werden sollen, müssen nicht unbedingt einem der $N-$Ecke angehören. (s. Haus mit Fenster und Tür am Ende dieses Paragraphen.)
Der aufgeführte Algorithmus arbeitet vollkommem korrekt, wenn alle n-Ecke eben sind. Aber auch bei "fast" ebenen n-Ecken (Normalfall bei parametrisierten Flächen) erzielt man kaum verfälschte Ergebnisse.

**Idee:**

(1) Falls die Facetten orientiert sind, sondert man mit Hilfe des schnellen "Normalen-Tests" die unsichtbaren Facetten und deren unsichtbaren Kanten aus.

(2) Der einfache "Fenster-Test" unterdrückt den aufwendigeren Sichtbarkeitstest (3)-(5) bei "offensichtlichen" Fällen.

(3) In der Bildtafel wird untersucht, ob das Bild einer Kante teilweise innerhalb des Bildes einer Facette (konvexes Polygon) liegt.

(4) Falls dies der Fall ist, wird mit Hilfe eines Testpunktes (im Raum) festgestellt, ob die Kante auf derselben Seite der Ebene der Facette wie das Projektionszentrum liegt oder nicht.

(5) Wird die Kante teilweise von der Facette verdeckt, werden die sichtbaren Teile bestimmt und abgespeichert.

(6) Nachdem die Kante gegen alle Facetten auf diese Weise untersucht worden ist, werden die sichtbaren Teile gezeichnet.

Zur Verwirklichung dieser Idee benötigen wir die folgenden drei Unterprogramme:

a) `is_line_convex_polygon`: berechnet den Schnitt einer Strecke mit einem konvexen Polygon.

b) `intmint`: bestimmt die Differenz $[a,b]\backslash[c,d]$ zweier Intervalle von $\mathbb{R}$.

c) `cp_vts3d_vts2d_spez`: ist eine gerinfügige Erweiterung von `cp_vts3d_vts2d`. In einem zusätzlichen Parameter (`pdist`: r_array) werden die Abstände der Punkte von der Bildtafel ($z-$Koordinaten der Punkte im System $(H;\mathbf{e}_1,\mathbf{e}_2,\mathbf{n}_0)$) mitgeliefert.

```
procedure is_line_convex_polygon(p1,p2 : vt2d; p_pol : vts2d_pol; np : integer;
                                          var t1,t2 : real; var ind : integer);
{Berechnet die Parameter t1,t2 der Schnittpunkte der Strecke p1,p2 mit dem konvexen
 Polygon p_pol[0],...p_pol[np]. ind=0 bzw. 2 : Strecke  innerhalb bzw. ausserhalb,
 ind=1: sonst. vts2d_pol: array[0..npfmax] of vt2d. }
{***********}
procedure intmint(a,b,c,d: real; var e1,f1,e2,f2: real; var ind: integer);
 {Berechnet die Intervall-Differenz  [a,b] \ [c,d].
 ind=0: leer, ind=1: 1 Interv., ind=2: 2 Interv.}
{***********}
procedure cp_vts3d_vts2d_spez(var p: vts3d; n1,n2 : integer; var pp : vts2d;
                                                             var pdist : r_array);
 {Zentralprojektion (Koordinaten) einer Punktreihe.
 pdist[i] : Distanz des Punktes p[i]  von der Bildtafel.}
{***********}
```

Zur **Datenstruktur:**
Für jede **Facette** (Polygon) werden in dem *record* `face_dat` die folgenden Daten gespeichert:

1) `npf, nef`: Anzahl der Punkte bzw. Kanten.
2) `fp[1]`, ... `fp[npf]`: Erster, zweiter, ... Punkt der Facette,
   `fe[1]`, ... `fe[nef]`: Erste, zweite, ... Kante der Facette.
   (Falls die Facetten orientiert sind, müssen sie (von außen gesehen) gegen den Uhrzeiger angeordnet sein.)
3) `box`: enthält minimale und maximale $x-$Werte bzw. $y-$Werte der Bildpunkte der Facette.
   `box.zmax`: maximaler Abstand der Punkte von der Bildtafel.
4) Den Normalenvektor `nv` und die Zahl `d` der Gleichung $\mathbf{n}_v \cdot \mathbf{x} = d$ der Ebene, die die Facette enthält.
5) `discentre`: Abstand des Zentrums von der Ebene, die die Facette enthält.
6) `vis`: `vis=true` bzw. `vis=false`, falls die orientierte Facette sichtbar bzw. unsichtbar ist.

Für jede **Kante** legen wir das *record* `edge_dat` an:

1) `ep1,ep2`: Erster bzw. zweiter Punkt der Kante.
2) `color,linew`: Farbe und Liniendicke der Kante
   (wird nur benutzt, wenn Parameter `newstyles=true`,
   s. Prozedur `cp_lines_before_convex_faces(oriented_faces,is_permitted,newstyles)`).
3) `vis`: `vis=true` bzw. `vis=false`, falls die Kante einer sichtbaren bzw. unsichtbaren Facette angehört.

Alle records `face_dat` bzw. `edge_dat` sind zu dem *array* `face` bzw. `edge` zusammengefaßt.
(Details in: `units/hiddenl.p` und `include/proc_zpo.pas`.)

Der Hiddenline-Algorithmus ist in dem folgenden Unterprogramm `cp_lines_before_convex_faces` enthalten.

Zum Unterprogramm `cp_lines_before_convex_faces (oriented_faces,is_permitted,newstyles)`:
`oriented_faces=true`: Die Facetten sind orientiert. Es wird der schnelle Normalentest durchgeführt.
`is_permitted=true`: Die Kanten dürfen die Facetten schneiden. (langsam!)
`newstyles=true`: Farben und Linienstärke gemäß `edge[i].color, edge[i].linew` (siehe Beispielprogramm).

(0) Im Hauptprogramm müssen zur Verfügung stehen:

   a) `np, nf, ne` : Anzahl der Punkte, Facetten, Kanten,

   b) die Koordinaten der Punkte $P_i : \mathbf{p}_i, \quad i = 1, \ldots, np$,

   c) für jede Kante die Daten: Anfangs- und Endpunkt `ep1,ep2`,

   d) für jede Facette die Daten:
   `npf,nef` : Anzahl der Punkte bzw. Kanten der Facette,
   `fp[1]`, ... `fp[npf]` : die Punkte der Facette, `fe[1]`, ... `fe[npf]` : die Kanten der Facette,
   die Gleichung $\mathbf{n}_v\mathbf{x} = d$ der zugehörigen Ebene.

(1) Es werden

   a) die Bildpunkte und deren Abstand zum Projektions-Zentrum berechnet,

   b) für jede Facette
   der "Abstand" (`discentre`) des Zentrums von dieser Ebene,
   die Fensterdaten des Bildpolygons in `box (xmin, xmax, ymin, y max)`,
   der maximale Abstand (`box.zmax`) der Facettenpunkte von der Bildebene berechnet.

   b') der Normalentest für orientierte Facetten durchgeführt.
   Ist die Facette sichtbar so wird sie und ihre Kanten als sichtbar erklärt.

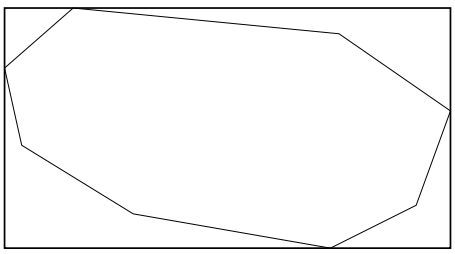


Abbildung 5.1: Fenster eines Polygons

(2) Untersuchung der $i$-ten Kante $e_i$, gegenüber der $j$-ten Facette $f_j$, falls beide sichtbar (vis=true) sind:

   a) Berechnung des Fensters (`xemin, xemax, yemin, yemax`) und des minimalen Abstandes (`zemin`) von der Bildtafel der Kante.

   b) Ist die Kante eine Kante der $j$-ten Facette ? (ja: $\rightarrow$ nächste Facette.)

   c) Falls `box.zmax > zemin` und sich Kantenfenster und Facettenfenster überlappen: Schnitt der Bildkante mit dem (konvexen) Bildpolygon der $j$-ten Facette.

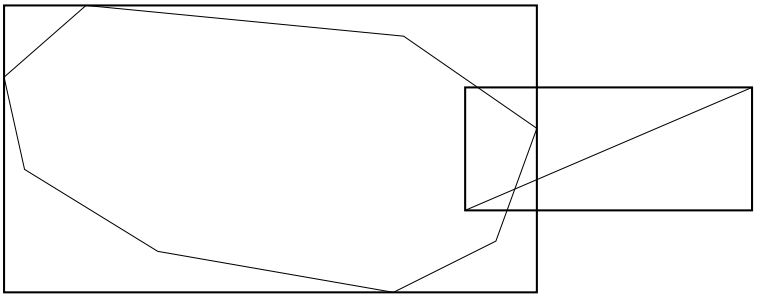


Abbildung 5.2: Fenstertest

d) Falls ein Teil der Bildstrecke in dem Polygon liegt:

d1) falls Schnitt Kante-Facette NICHT zugelassen ist (`is_permitted`=false):
Test mit Testpunkt $\mathbf{p}_t$, ob die Kante (im Raum) vor oder hinter der Ebene der $j$-ten Facette liegt.

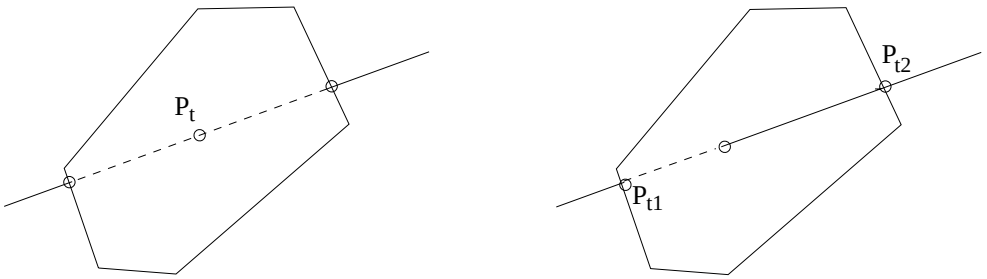



Abbildung 5.3: Fall d1) bzw. d2)

d2) falls Schnitt Kante-Facette zugelassen ist (`is_permitted`=true):
Mit Hilfe zweier Testpunkte $\mathbf{p}_{t1}, \mathbf{p}_{t2}$ wird die Lage der Kante relativ zur Facette festgestellt und der Teil der Kante, der vor der Facette liegt, bestimmt.

e) Falls die Kante teilweise hinter der Facette liegt:
Die Parameterintervalle $[par1(l), par2(l)]$ der noch sichtbaren Teilstrecken werden berechnet und abgespeichert.

(3) Ist die Kante $e_i$ gegen alle Facetten getestet, werden die sichtbaren Teilstrecken gezeichnet.

```
procedure cp_lines_before_convex_faces(oriented_faces,is_permitted,newstyles : boolean);
{Projiziert und zeichnet Kantenteile VOR (orientierten) ebenen n-Ecken.
 oriented_faces=true: die Flaechen sind orientiert,
 is_permitted=true: Kanten duerfen die Flaechen schneiden.
 Aus dem Hauptprogramm muessen bereitstehen:
 np, ne, nf: Anzahl der Punkte, Kanten, Flaechen,
 p[1],...,p[np] : Punkte,
 face[i].fp[k] (face[i].fe[k]): k-ter Punkt (k-te Kante) in i-ter Flaeche,
 face[i].npf (face[i].nef): Anzahl der Punkte (Kanten) in der i-ten Flaeche,
 edge[i].ep1 (edge[i].ep2): Anfangs-(End-)Punkt der i-ten Kante,
 face[i].nv,face[i].d: Koeffizienten der Ebenengleichung.}
{************}
```

**Beispiel 5.1** *a) Polyeder auf einem TORUS (orientierte Facetten)*
*b) Polyeder auf einem TORUS-Teil (nicht orientierte Facetten)*

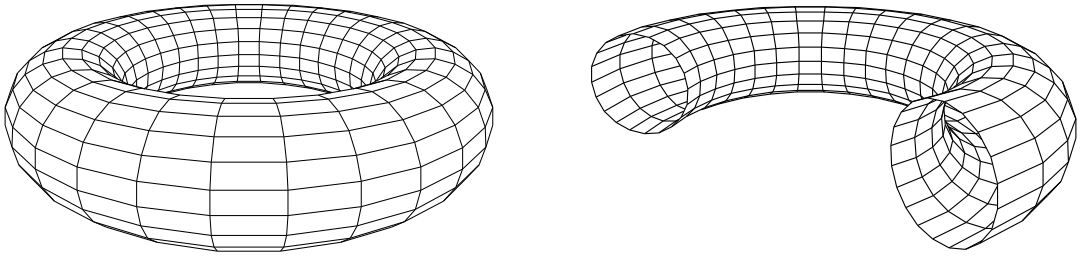


Abbildung 5.4: Torus (orient. Facetten) bzw. Torusteil (nicht orient. Facetten)

## 5.2 Hilfsprogramme zum Hiddenline-Algorithmus

Um die Vorarbeit des Anwenders des Hiddeline-Algorithmus zu reduzieren, werden hier 4 Hilfsprogramme angegeben. Das erste Unterprogramm ist besonders für Polyeder, die drei restlichen zur Darstellung von parametrisierten Flächen geeignet.

### 5.2.1 Das Unterprogramm `aux_polyhedron`

Im Hauptprogramm müssen folgende Daten bereitstehen:

(1) `np,nf` (Anzahl der Punkte bzw. Facetten),

(2) die Punkte $P_i : \mathbf{p}_i,\ i = 1, \ldots,$ `np` der Facetten und Kanten,

(3) für jede Facette: `npf` (Anzahl der Punkte dieser Facette),
`fp[1],...,fp[npf]` (Nummern der Punkte, die das Facettenpolygon bilden).

Das Unterprogramm `aux_polyhedron` numeriert die Kanten und berechnet

(1) `ne` (Anzahl der Kanten),

(2) für jede Kante: `ep1, ep2` (Nummern von Anfangs- und Endpunkt),

(3) für jede Facette: `npf, fe[1],...,fe[npf]` ( Anzahl der Kanten und Nummern der Kanten des Facettenpolygons).

```
procedure aux_polyhedron;
{np, ne, nf : Anzahl der Punkte, Kanten, Flaechen
 face[i].npf   : Anzahl der Punkte (Kanten) der i-ten Flaeche
 edge.ep1, ep2 : Anfangs- bzw. Endpunkt der k-ten Kante
 face[i].fp[k] : k-ter Punkt der i-ten Flaeche (positiv orientiert!!!)
 face[i].fe[k] : k-te Kante der i-ten Flaeche
 ! Dieses UP berechnet aus nf,face[i].fp und face[i].npf:
 ne, face[i].nef, face[i].fe[k], edge[i].ep1 und edge[i].ep2 !.}
```

**Beispiel 5.2** *HÄUSER*
*(Man beachte,*

*a) daß die Dachflächen nicht orientiert sind und*

*b) daß Kanten, die keine Randkanten einer Facette (Fenster,Tür) sind, nicht in dieser Facette liegen dürfen, sondern "etwas" davor. Die Menge der Kanten setzt sich hier aus der Menge der Kanten der Facetten und der zusätzlichen Menge der Kanten der Fenster und Türe zusammen.)*

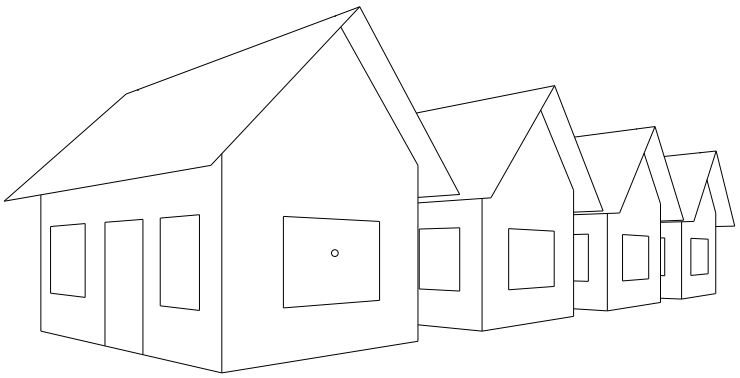


Abbildung 5.5: Häuser

**Beispiel 5.3** *a) 3 Balken b) Kiosk*

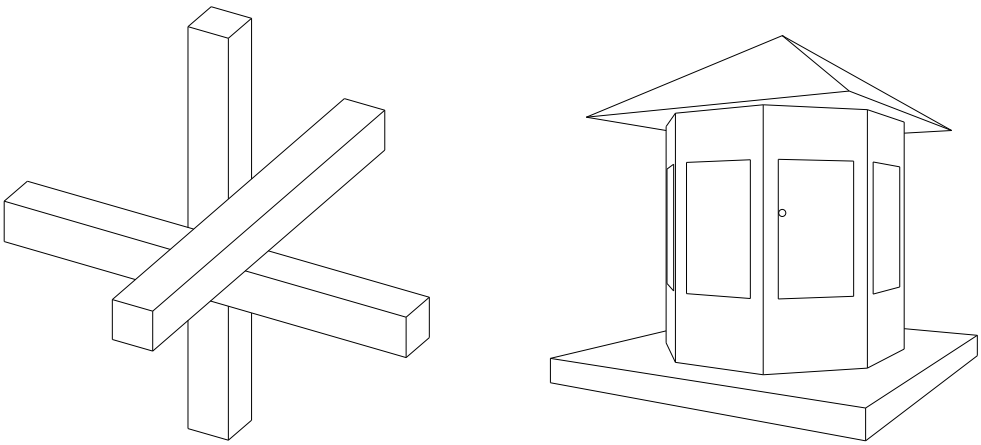
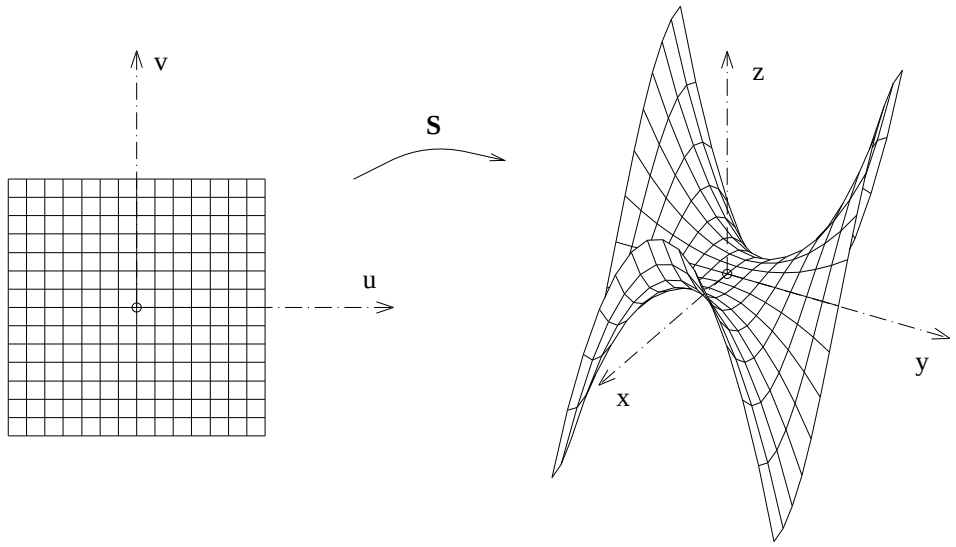


Abbildung 5.6: 3 Balken, Kiosk

### 5.2.2 Die Unterprogramme `aux_quadrangle, aux_cylinder, aux_torus` und Darstellung parametrisierter Flächen

Zur Darstellung einer **parametrisierten** Fläche $\Phi : \mathbf{x} = \mathbf{S}(u, v), u \in [a, b], v \in [c, d]$, wählt man meistens in der Parameterebene ($u - v-$Ebene) ein Rechteckgitter und dessen Bild im $\mathbb{R}^3$.




Parameterebene Fläche im $\mathbb{R}^3$

Abbildung 5.7: Netz in der Parameterebene und sein Bild

Sind die Bilder der Gitterrechtecke (im $\mathbb{R}^3$) fast eben, so kann man den obigen Hiddenline-Algorithmus zur Projektion dieser Vierecke verwenden. (Im Falle des TORUS sind die Vierecke sogar exakt eben.). Der Algorithmus `aux_polyhedron` aus Kap. 5.2 kann zwar hier auch benutzt werden, ist aber für die besondere Struktur (nur viereckige Facetten) hier zu schwerfällig und langsam. Deshalb werden hier für die häufig auftretenden Fälle, daß dieFlächenstücke viereckig oder zylindrisch oder torusartig sind, Hilfsprogramme angegeben, die aus den **Vorgaben** $n_1, n_2$ (Anzahl der Teile des Rechteckgitters in u- bzw. v-Richtung), die Facetten und Kanten numeriert und die folgenden **Daten** berechnen:

(1) für jede Facette: `fp[1],..., fp[npf]` (Nummern der Punkte des Polygons),
`fe[1],..., fe[nef]` (Nummern der Kanten des Polygons),

(2) für jede Kante: `ep1, ep2` (Nummern von Anfangs- und Endpunkt),

(3) `np, ne, nf` : Anzahl der Punkte, Kanten bzw. Facetten,

(4) die Gleichungen $\mathbf{n}_i\mathbf{x} - d_i = 0$ der Ebenen, die die Facetten (`faces`) enthalten.

Die $n_1 \cdot n_2$ Punkte $P_1, P_2, \ldots$ müssen gemäß der folgenden Abbildung dem jeweiligen $u - v-$Netz zugeordnet werden (Für die ersten $n_1$ Punkte ist $v = a = const.$).

Im Fall "quadrangle" ist $\Delta u = (b-a)/(n_1-1)$ und $\Delta v = (c-d)/(n_2-1)$.
Im Fall "cylinder" ist $\Delta u = (b-a)/n_1$ und $\Delta v = (c-d)/(n_2-1)$.
Im Fall "torus" ist $\Delta u = (b-a)/n_1$ und $\Delta v = (c-d)/n_2$.

In jedem Fall ist zu beachten, daß der Punkt mit den Parametern $u = a+(i-1)\cdot\Delta u, v = c+(k-1)\cdot\Delta v$ die Nummer $(k-1)\cdot n_1 + i$ hat.
Um mehrere Flächen in einem Programm behandeln zu können, sind in den folgenden Unterprogrammen die Anzahlen der schon aufgenommenen Punkte `np0`, Kanten `ne0` und Facetten `nf0` nötig. Für die erste darzustellende Fläche ist `np0=ne0=nf0=0`.

```
procedure aux_quadrangle(n1,n2,np0,ne0,nf0: integer);
{****}
```

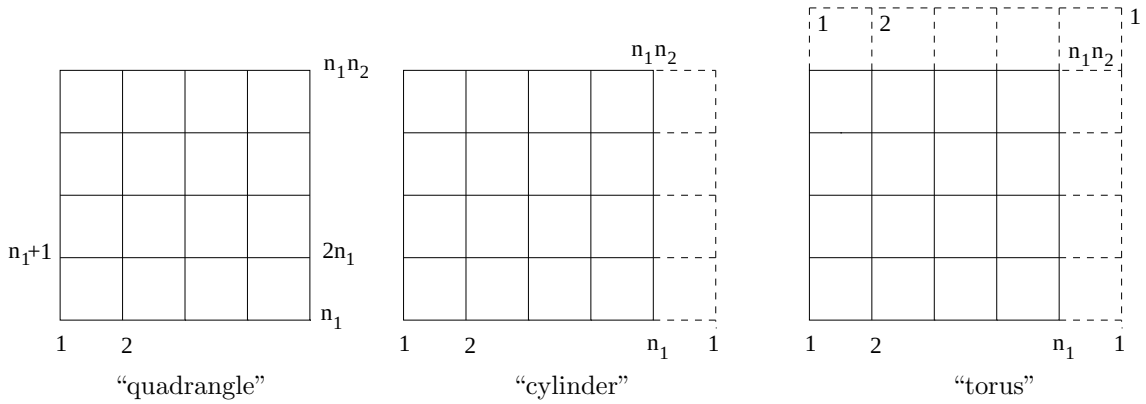



Abbildung 5.8: Netztypen

```
procedure aux_cylinder(n1,n2,np0,ne0,nf0: integer);
{****}
procedure aux_torus(n1,n2,np0,ne0,nf0: integer);
```

Im Hauptprogramm müssen für den Hiddenline-Algorithmus bereitstehen:

a) `n1,n2`,

b) Die Punkte $P_i : \mathbf{p}_i$, zeilenweise numeriert von unten nach oben (s. Zeichnung),

c) `npf,nef`, die Anzahl der Punkte bzw. Kanten in einer Facette. Normalerweise ist `npf=4, nef=4`. Bei Durchdringungen können beide allerdings auch größer sein.

**Beispiel 5.4** *Zwei Tori (s. Beispiel-Programm* `tori_h.p`*)*

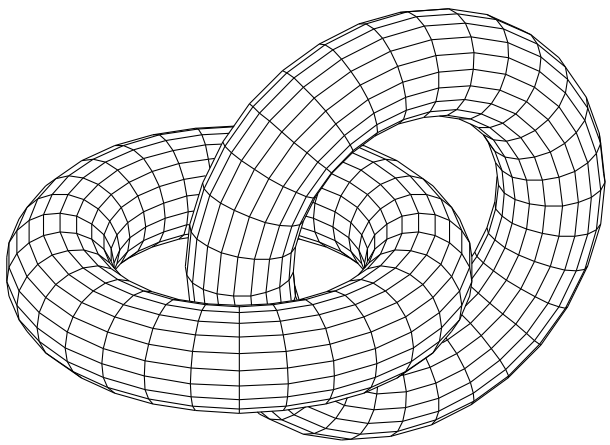


Abbildung 5.9: Zwei Tori

**Aufgabe 5.1** *ROHRKNOTEN*
*Die Mittenkurve eines Rohrknotens liegt auf einem Torus und hat hier die Parameterdarstellung*

$$\mathbf{x} = \mathbf{c}(u) := (h\cos(u), h\sin(u), r_2\sin(cu)),$$

*mit* $h = r_1 + r_2\cos(cu), c = 1.5$ *und* $0 \le u \le 4\pi$.
*($r_1$ ist der "große" Radius, $r_2$ der "kleine" Radius des Torus.) Jeder Punkt der Mittenkurve ist Mittelpunkt eines Kreises, der senkrecht zur Mittenkurve steht. Für die Ebene, die im Punkt* $\mathbf{c}(u)$ *senkrecht zur Mittenkurve steht, wählen wir die folgenden Vektoren als Basiseinheitsvektoren:*

$$\mathbf{e}_1(u) := \dot{\mathbf{c}}(u) \times \mathbf{e}_z / \| \dots \|, \quad \mathbf{e}_2(u) := \mathbf{e}_1 \times \dot{\mathbf{c}}(u) / \|\dot{\mathbf{c}}(u)\|,$$

*wobei* $\mathbf{e}_z := (0, 0, 1)$ *ist.*
*Also ist*

$$\mathbf{x} = \mathbf{S}(u, v) := \mathbf{c}(u) + \mathbf{e}_1(u) r_3 \cos v + \mathbf{e}_2(u) r_3 \sin v, 0 \le r_3 \le r_2,$$

*eine Parameterdarstellung eines Rohrknotens. $r_3$ ist die "Dicke" des Rohres.*

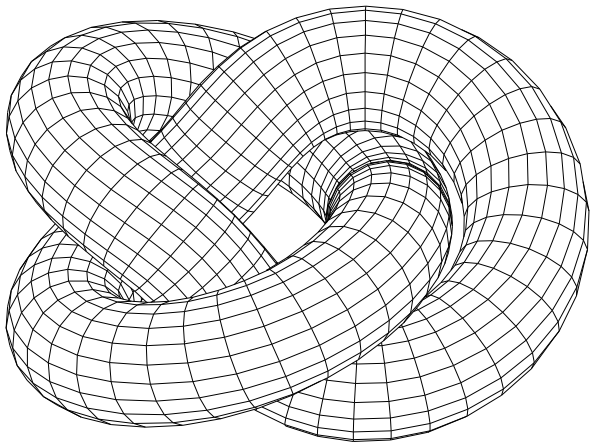


Abbildung 5.10: Rohrknoten ("torus")

**Beispiel 5.5** *Möbiusband (***nicht** *vom Typ "quadrangle")*
*(Die Berandungskurven der Bänder liegen auf einem Torus!)*

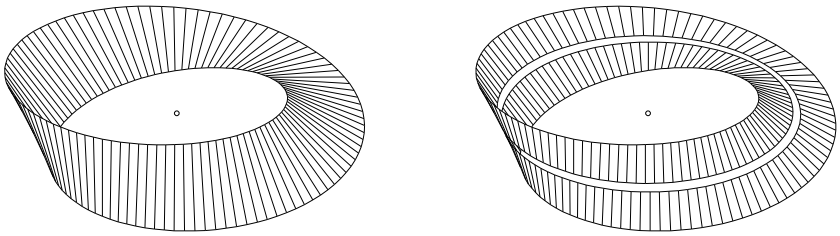


Abbildung 5.11: Möbiusband, geschlitztes Möbiusband

## 5.3 Schnitt zweier Polygone im Raum, Schnitt zweier Polyeder

### 5.3.1 Schnitt zweier ebener von Polygonen begrenzte Flächen im Raum

Als Vorbereitung für die Bestimmung der Schnittkanten zweier sich durchdringender Polyeder überlegen wir uns zunächst ein Unterprogramm, das die Schnittstrecke zweier durch ebene konvexe

Polygone berandete Flächenstücke im Raum berechnet.
Gegeben: Zwei ebene konvexe Polygone $P_{11}, P_{12}, \ldots, P_{1m}, P_{21}, P_{22}, \ldots, P_{2n}$ im $\mathbb{R}^3$. Die Polygone liegen nicht in einer gemeinsamen Ebene.
Gesucht: Der Schnitt (Strecke), der durch die Polygone berandeten ebenen Flächenstücke.

**Idee:**

(1) Man bestimmt zunächst die Schnittgerade der die Polygone enthaltenden Ebenen.

(2) Nun schneidet man diese Schnittgerade mit den Polygonen und enthält zwei Schnittstrecken.

(3) Der Durchschnitt der beiden Schnittstrecken ist die gesuchte Strecke.

Der **Algorithmus** (Prozedur `is_n1gon_n2gon3d`):

(0) Für jedes Polygon wird mit `box3d_of_pts` der kleinste achsenparallele Quader bestimmt, der das eine bzw. das andere Polygon enthält. Nur wenn sich beide Quader schneiden,beginnt der folgende Schnittalgorithmus.

(1) Es werden die Gleichungen $\mathbf{n}_1\mathbf{x} - d_1 = 0, \mathbf{n}_2\mathbf{x} - d_2 = 0$ mit Hilfe von `plane_equ` berechnet. `is_plane_plane` liefert einen Punkt und einen Richtungsvektor der Schnittgerade $g_s$ beider Ebenen.

(2) Um die Gerade $g_s$ mit dem ersten Polygon zu schneiden, führen wir in der Polygonebene das folgende Koordinatensystem ein: $P_{11}$ ist der Nullpunkt und die Punkte $P_{12}, P_{1m}$ sind die Achseneinheitspunkte.

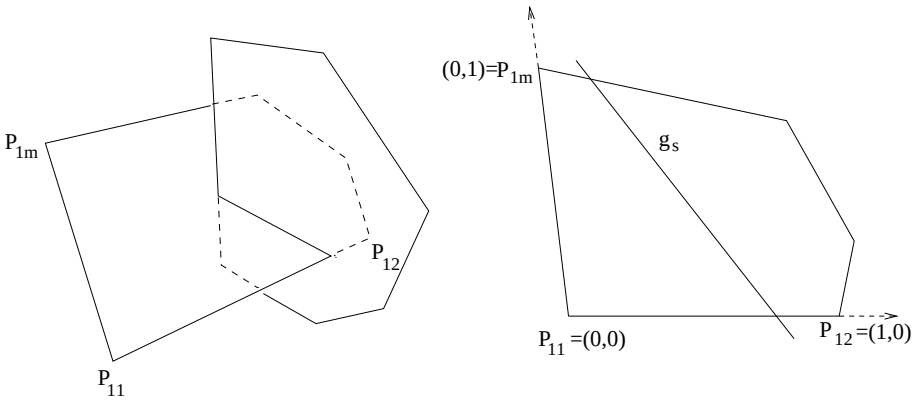



Abbildung 5.12: Zu (2) des Algorithmus

Da die Polygonpunkte i.a. nicht exakt in einer Ebene liegen, berechnen wir mit Hilfe des Unterprogramms `ptco_plane3d` die Koordinaten der zugehörigen Lotfußpunkte $Q_{1i}$ bezüglich des obigen Koordinatensystems. Speziell gilt dann $Q_{11} = (0,0), Q_{12} = (1,0)$ und $Q_{1m} = (0,1)$. Nachdem man noch zwei Punkte der Gerade $g_s$ so in die Ebene projiziert hat, lassen sich mit `is_line_convex_polygon` die Parameter der Schnittkante $t_1, t_2$ bestimmen. Analog verfährt man mit dem zweiten Polygon und erhält Parameter $s_1, s_2$.

(3) `is_interv_interv` berechnet schließlich den Schnitt $[s,t] := [t_1,t_2] \cap [s_1,s_2]$ der Intervalle. Damit ist die Schnittstrecke der beiden Polygone bestimmt.

```
procedure is_interv_interv(var a,b,c,d,aa,bb : real; var inters: boolean);
{Berechnet den Schnitt der Intervalle [a,b], [c,d] .}
```

```
{************}
procedure box3d_of_pts(var p : vts3d_pol; np: integer; var box : box3d_dat);
var a,b : real;   i: integer;
{Bestimmt den zugehrigen (kleinsten) achsenparallelen Quader.}
{************}

function is_two_boxes3d(var box1,box2 : box3d_dat) : boolean;
{Schnitt zweier achsenparalleler Quader.}
{************}
procedure is_line_conv_pol_in_plane3d(var pl,rl: vt3d; var pp : vts3d_pol;
                                      npp : integer;
                                      var t1,t2 : real; var inters : boolean);
{Schnitt eines konv. Polygons mit einer in der Polygonebene liegenden Gerade.}
{************}
```

### 5.3.2 Schnitt zweier Polyeder

Gegeben: Zwei sich durchdringende Polyeder $\Pi_1, \Pi_2$.
Gesucht: Die Schnittkanten.

**Idee:**
Da man die Polyeder in der Regel auch darstellen möchte, ist es zweckmäßig die Datenstruktur des Hiddenlinealgorithmus `is_lines_before_convex_faces` zu verwenden. Dadurch ergeben sich einige Vereinfachungen.

Der **Algorithmus:**

(1) Man berechnet zunächst mit `boxes_of_faces` die für einen schnellen Vortest notwendigen achsenparallelen Quader.

(2) Da die Gleichungen der Flächen schon vorliegen (Sie sind auch für den Hiddenlinealgorithmus nötig.), verwendet man hier das für diesen Fall angepaßte Unterprogramm `is_face_face`, um die Schnittkanten zu bestimmen.

(3) Die Schnittkante der $i$-ten Facette mit der $k$-ten Facette wird als zusätzliche Kante in das Array **edge** aufgenommen und in **face[i]** und **face[k]** vermerkt, daß diese Kante in der $i$-ten bzw. $k$-ten Facette liegt. (Letzteres ist notwendig, damit die Kante nicht mit der Facette, in der sie liegt, verglichen wird. Durch Rechenungenauigkeiten könnte die Kante hinter der zugehörigen Ebene liegen und damit unsichtbar sein.)

Im Folgenden sind die Unterprogrammköpfe von `boxes_of_faces, is_face_face` sowie die im Hauptprogramm notwendige Ergänzung für den Fall zweier sich durchdringender Flchen angegeben (s. Beispiel-Programm `flaech_h.p`).

```
procedure boxes_of_faces;
{Berechnet zu den Facetten achsenparallele Quader.}
{************}
procedure is_face_face(i,k: integer;  var ps1,ps2 : vt3d;
                                      var intersection: boolean);
{Berechnet die Schnittstrecke zweier nicht in einer Ebene liegenden
 (konvexen) Flaechen eines Polyeders.}
{**************}
.............................
{Durchdringung zweier Flaechen: Affensattel-Zylinder  (s. flaech_h.p)}
    boxes_of_faces;
    for i:= 1 to nf1 do         { 1.Flaechenschleife }
```

```
        begin
        for k:= nf1+1 to nf do  { 2.Flaechenschleife }
            begin
            is_face_face(i,k, ps1,ps2,inters);
            if inters then
               begin
               p[np+1]:= ps1;  p[np+2]:= ps2;
               with edge[ne+1] do
                    begin ep1:= np+1; ep2:= np+2; vis:= true; end;
               np:= np+2;  ne:= ne+1;
               with face[i] do begin fe[nef+1]:= ne; nef:= nef+1;  end;
               with face[k] do begin fe[nef+1]:= ne; nef:= nef+1;  end;
               end;  { if }
            end;  { for k }
        end;  { for i }
.................................
```

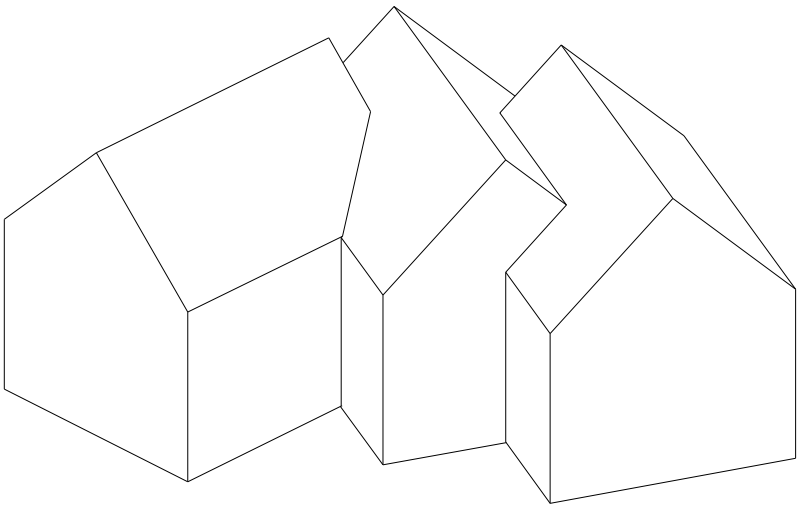
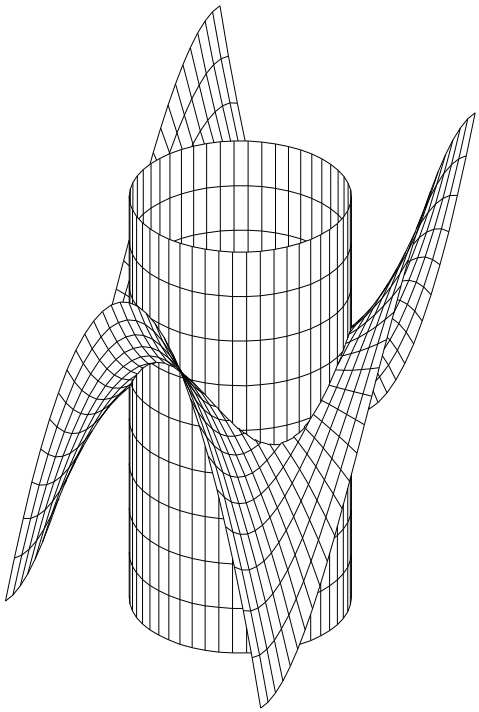
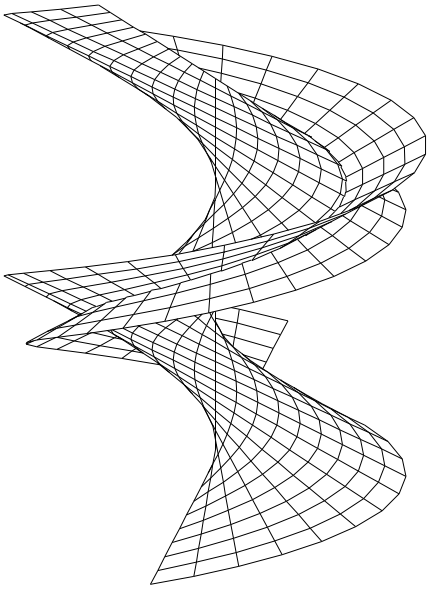


Abbildung 5.13: Durchdringungen

Bei der Bestimmung von **Selbstdurchdringungen** muß man eine Fläche mit sich selbst schneiden. Dies führt bei benachbarten Facetten zu Problemen. Deshalb sollte man mit einem Unterprogramm Nachbarschaften aufspüren und nur nicht benachbarte Facetten auf eventuellen Schnitt untersuchen.



a) sich durchdringende Flächen



b) selbstdurchdringende Strahlschraubfläche

Die Prozeduren dieses Kapitels sind in der Datei `proc_zpo.pas` enthalten und stehen über das unit `hiddenl` zur Verfügung (s. Beispiel-Programme `tori_h.p, flaech_h.p`).

# Kapitel 6

# TRIANGULIERUNG IMPLIZITER FLÄCHEN

## 6.1 Der Triangulierungs–Algorithmus

Der Algorithmus verwendet wesentlich die folgende Prozedur `surfacepoint`

### 6.1.1 Die Prozedur `surfacepoint`

Die Prozedur `surfacepoint` bestimmt zu einem Punkt $\mathbf{q}$ in der Nähe einer impliziten Fläche $\Phi: \ f(\mathbf{x}) = 0$ einen Flächenpunkt $\mathbf{p}$, der in der Nähe des zugehörigen Lotfußpunktes von $\mathbf{q}$ liegt. $\mathbf{p}$ ist im Falle einer Ebene exakt der Lotfußpunkt. Die Prozedur berechnet außerdem eine Flächeneinheitsnormale und zwei orthonormale Tangentenvektoren in $\mathbf{p}$.

Es seien nun die implizite Fläche $\Phi: \ f(\mathbf{x}) = 0$, für die stets der Gradient $\nabla f$ existiert und nicht der Nullvektor ist, und ein Punkt $\mathbf{q}$ in der Nähe der Fläche gegeben.

1. (a) $\mathbf{u}_0 = \mathbf{q}$

   (b) repeat $\mathbf{u}_{k+1} := \mathbf{u}_k - \dfrac{f(\mathbf{u}_k)}{\nabla f(\mathbf{u}_k)^2} \nabla f(\mathbf{u}_k)$
   (Newtonschritt für die Funktion $g(t) := f(\mathbf{u}_k + t\nabla f(\mathbf{u}_k))$ )
   until $\|\mathbf{u}_{k+1} - \mathbf{u}_k\|$ ist "genügend" klein.
   *Flächenpunkt* $\mathbf{p} = \mathbf{u}_{k+1}$.

2. Die *Flächennormale* im Punkt $\mathbf{p}$ ist $\mathbf{n} := \nabla f(\mathbf{p})/\|...\|$.

3. Als *Tangentenvektoren* wählen wir
   $\mathbf{t}_1 := (n_y, -n_x, 0)/\|...\|$ falls $n_x > 0.5$ or $n_y > 0.5$
   oder $\mathbf{t}_1 := (-n_z, 0, n_x)/\|...\|$
   und $\mathbf{t}_2 := \mathbf{n} \times \mathbf{t}_1$ wobei $(n_x, n_y, n_z) := \mathbf{n}$ ist.

### 6.1.2 Idee des Algorithmus

S0 Es sei $\delta_t$ die ungefähre Seitenlänge der Dreiecke der Triangulierung. Wähle eine Punkt $\mathbf{s}$ in der Nähe der Fläche. Bestimme mit der Prozedur `surfacepoint` einen Flächenpunkt $\mathbf{p}_1$. Umgebe $\mathbf{p}_1$ in der Tangentialebene mit einem regulären Sechseck $\mathbf{q}_2, ..., \mathbf{q}_7$. Bestimme mit `surfacepoint` zu $\mathbf{q}_2, ..., \mathbf{q}_7$ die Flächenpunkte $\mathbf{p}_2, ..., \mathbf{p}_7$. Die so entstandenen Dreiecke auf der Fläche sind die ersten 6 Dreiecke der Triangulierung (s. 6.3,6.6).
Wir nennen das Polygon $\mathbf{q}_2, ..., \mathbf{q}_7$ das (erste) *aktuelle Frontpolygon* $\Pi_0$. Falls die Triangulierung begrenzt werden soll (nicht nötig bei geschlossenen Flächen) durch Flächenkurven $\Gamma_1, \Gamma_2, ...$ (s. Beispiele), bestimmen wir weitere *Frontpolygone* $\Pi_1, \Pi_2, ...$ auf diesen Kurven.
Für spezielle Flächen (Zylinder Torus,...) kann es besser sein, mit einem vordefinierten ersten Frontpolygon zu beginnen. (s. Beispiele)

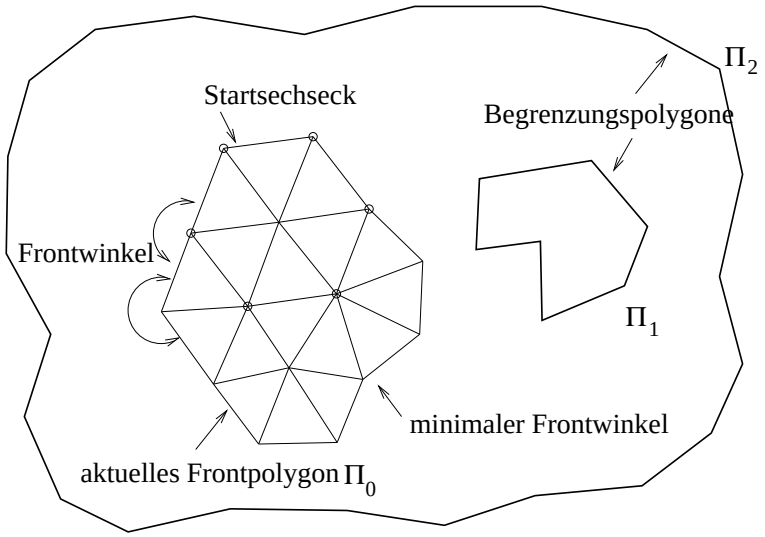



Abbildung 6.1: Bezeichnungen für den Algorithmus

S1 Für jeden Punkt des aktuellen Frontpolygons bestimmen wir den Außenwinkel. Wir nennen diese Winkel *Frontwinkel*.

S2 Prüfe, ob ein aktueller Frontpunkt $\mathbf{p}_i$ nahe einem

- Punkt von $\Pi_0$ ist, der von $\mathbf{p}_i$ und seinen Nachbarn verschieden oder
- Punkt von einem anderen Frontpolygon $\Pi_k, k > 0$ ist.

Im ersten Fall: Teile das aktuelle Frontpolygon $\Pi_0$ in ein kleineres und ein weiteres Frontpolygon. (s. Abb. 6.2, 6.9a,b)
Im zweiten Fall: Ist $\mathbf{p}_i$ nahe einem Punkt des Frontpolygons $\Pi_m$, so vereinige die Polygone $\Pi_0, \Pi_m$ zu einem neuen (größeren) Frontpolygon. Lösche $\Pi_m$ (s. Abb. 6.2, 6.9d,c).

S3 Bestimme einen Frontpunkt $\mathbf{p}_m$ des aktuellen Frontpolygons $\Pi_0$ mit minimalem Frontwinkel. Umgebe $\mathbf{p}_i$ mit $\approx$ reguläre Dreiecke mit Seitenlänge $\approx \delta_t$. Lösche $\mathbf{p}_m$ aus dem Polygon $\Pi_0$ und füge die neuen Punkte in das aktuelle Frontpolygon $\Pi_0$ ein.

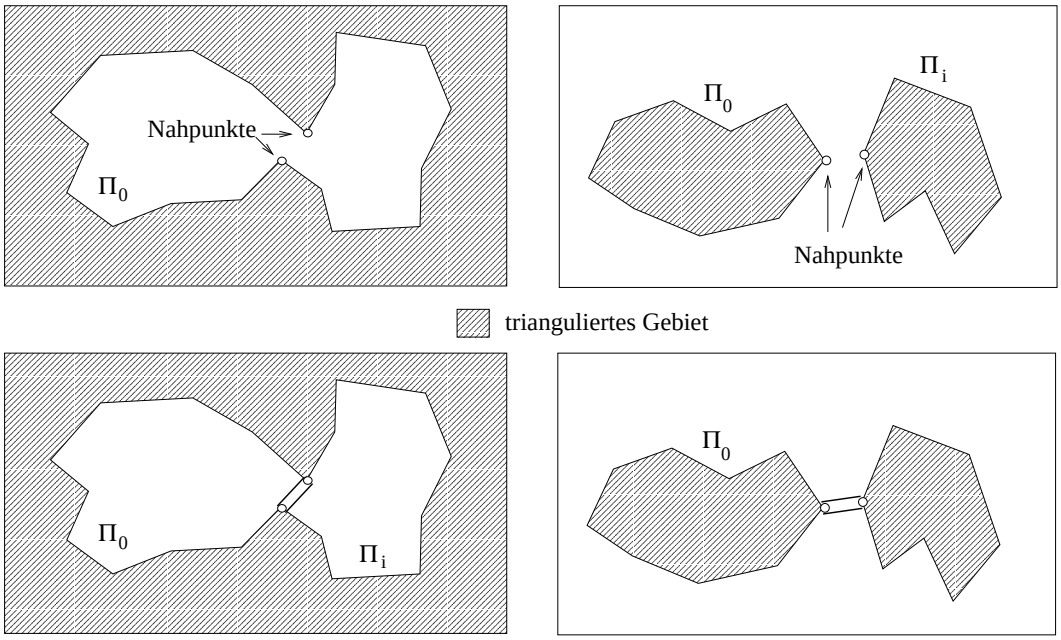



Abbildung 6.2: Teilen (links) und Vereinigen (rechts) des aktuellen Frontpolygons

S4 Wiederhole die Schritte S1,S2,S3 bis das aktuelle Frontpolygon $\Pi_0$ nur noch aus 3 Punkten besteht, die ein neues Dreieck liefern. Falls ein weiteres Frontpolygon existiert, so erkläre dies zum *aktuellen* Frontpolygon $\Pi_0$ und wiederhole die Schritte S1,S2,S3. Wenn kein Frontpolygon mehr existiert, so ist die Triangulierung beendet.
Falls die Fläche nicht beschränkt ist, sollte man sie durch Randpolygone oder durch eine "bounding box" begrenzen. (s. 6.8c,d).

### 6.1.3 Die Datenstruktur

$\delta_t$ ist die ungefähre *Kantenlänge* der Dreiecke.
Die *Punkte* der Triangulierung werden fortlaufend durchnumeriert. Für jeden Punkt speichern wir die folgenden Informationen:

- die Koordinaten,
- die Flächennormale $\mathbf{n}$ und Tangentenvektoren $\mathbf{t}_1, \mathbf{t}_2$ so, daß $\mathbf{n}, \mathbf{t}_1, \mathbf{t}_2$ orthonormal sind,
- der aktuelle Frontwinkel, falls $\mathbf{p}_i$ ein aktueller Frontpunkt ist,
- die boolsche Variable `angle_changed` mit `...= true`, falls sich der aktuelle Winkel durch Einfügen von Punkten geändert hat und deshalb neu berechnet werden muß,
  die boolsche Variable `boarder_point`. `..= true` bedeutet, daß $\mathbf{p}_i$ auf dem Rand der Triangulierung liegt und bei weiteren Betrachtungen (Frontwinkel, Distanzcheck) ignoriert wird.

Die *Dreiecke* werden fortlaufend numeriert. Für jedes Dreieck werden die Nummern der Eckpunkte gespeichert.

Die *Frontpolygone* werden durch die Integer-Arrays ihrer Punkte repräsentiert.

### 6.1.4 Der Schritt S0

Es sei $\mathbf{s}$ ein Startpunkt in der Nähe der Fläche.
Die Prozedur `surfacepoint` bestimmt den ersten Punkt $\mathbf{p}_1$ der Triangulierung und das Orthonormalsystem $\mathbf{n}_1, \mathbf{t}_{11}, \mathbf{t}_{12}$. Die sechs Punkte $\mathbf{p}_2, ..., \mathbf{p}_7$ sind die zu

$$\mathbf{q}_{i+2} := \mathbf{p}_1 + \delta_t \cos(i\pi/3)\mathbf{t}_{11} + \delta_t \sin(i\pi/3)\mathbf{t}_{12},\ i := 0, ...5$$

gehörigen Flächenpunkte, ($\mathbf{p}_i = \texttt{surfacepoint}(\mathbf{q}_i)$. Die Punkte $\mathbf{q}_2, ..., \mathbf{q}_7$ liegen in der Tangentialebene von $\mathbf{p}_1$ und bilden ein reguläres Sechseck.
Wir erhalten die ersten 6 Dreiecke:
$(\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3), (\mathbf{p}_1, \mathbf{p}_3, \mathbf{p}_4), (\mathbf{p}_1, \mathbf{p}_4, \mathbf{p}_5), (\mathbf{p}_1, \mathbf{p}_5, \mathbf{p}_6), (\mathbf{p}_1, \mathbf{p}_6, \mathbf{p}_7), (\mathbf{p}_1, \mathbf{p}_7, \mathbf{p}_2)$.

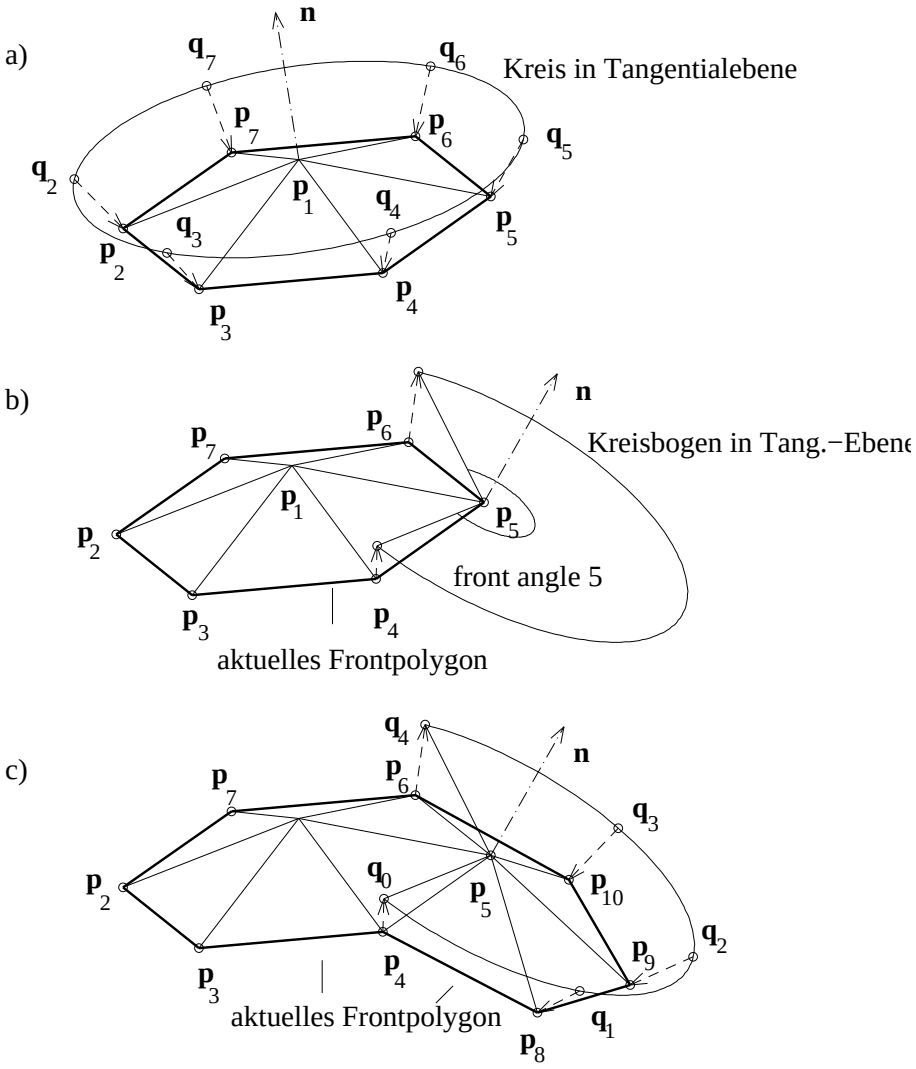



Abbildung 6.3: Die ersten Schritte des Algorithmus

### 6.1.5 Der Schritt S1

Falls ein Punkt $\mathbf{p}_{0i}$ des aktuellen Frontpolygons $\Pi_0 = (\mathbf{p}_{01}, \mathbf{p}_{02}, ..., \mathbf{p}_{0N_0})$ gerade eingefügt worden ist oder falls ein Nachbar von $\mathbf{p}_{0i}$ ein neuer Punkt ist, ist es notwendig, den aktuellen Frontwinkel $\omega$ des Punktes $\mathbf{p}_{0i}$ neu zu berechnen. Es sei
$\mathbf{v}_1 := \mathbf{p}_{0,i-1}$ if $i > 1$ oder $\mathbf{v}_1 := \mathbf{p}_{0N_0}$ if $i = 1$,
$\mathbf{v}_2 := \mathbf{p}_{0,i+1}$ if $i < N_0$ oder $\mathbf{v}_2 := \mathbf{p}_{01}$ if $i = N_0$ und
$(\xi_1, \eta_1, \zeta_1)$ die Koordinaten von $\mathbf{v}_1$, $(\xi_2, \eta_2, \zeta_2)$ die Koordinaten von $\mathbf{v}_2$ bezgl. des orthonormalen Systems $\mathbf{n}, \mathbf{t}_1, \mathbf{t}_2$ am Punkt $\mathbf{p}_{0i}$,
$\omega_1 :=$ Polarwinkel von $(\xi_1, \eta_1)$, $\omega_2 :=$ Polarwinkel von $(\xi_2, \eta_2)$, dann ist der Winkel am Punkt $\mathbf{p}_{0i}$ $\omega = \omega_2 - \omega_1$ , falls $\omega_2 \geq \omega_1$ andernfalls $\omega = \omega_2 - \omega_1 + 2\pi$.

### 6.1.6 Der Schritt S2

Um zu verhindern, daß neue Dreiecke alte Dreiecke überdecken, prüfen wir

- die Abstände von Punktepaaren des aktuellen Frontpolygons $\Pi_0$. Falls es Punkte $\mathbf{p}_{0i}, \mathbf{p}_{0j}, i < j$, (Nahpunkte) gibt, die nicht benachbart oder Nachbarn von Nachbarn sind und $\|\mathbf{p}_{0i} - \mathbf{p}_{0j}\| < \delta_t$ ist, wird das aktuelle Frontpolygon in das neue aktuelle Frontpolygon $(\mathbf{p}_{01}, ..., \mathbf{p}_{0i}, \mathbf{p}_{0j}, ..., \mathbf{p}_{0N_0})$ mit $N_0 - (j - i - 1)$ Punkte und ein weiteres Frontpolygon $(\mathbf{p}_{0i}, ...., \mathbf{p}_{0j})$ mit $j - i + 1$ Punkte zerlegt. (s. 6.2,6.9a,b) $\mathbf{p}_{0i}, \mathbf{p}_{0j}$ dürfen bei späteren Abstandsprüfungen nicht mehr verwendet werden.

- den Abstand der Punkte des aktuellen Frontpolygons $\Pi_0$ zu Punkten aller restlichen Frontpolygone $\Pi_k, k > 0$. Falls es Punkte $\mathbf{p}_{0i} \in \Pi_0$ und $\mathbf{p}_{mj} \in \Pi_m$ mit $\|\mathbf{p}_{0i} - \mathbf{p}_{mj}\| < \delta_t$ (Nahpunkte) gibt, werden die Polygone
  $\Pi_0 = (\mathbf{p}_{01}, ..., \mathbf{p}_{0N_0})$ und $\Pi_m = (\mathbf{p}_{m1}, ..., \mathbf{p}_{mN_m})$
  vereinigt zu einem neuen aktuellen Frontpolygon

  $$\Pi_0 = (\mathbf{p}_{01}, ..., \mathbf{p}_{0i}, \mathbf{p}_{mj}, ...\mathbf{p}_{mN_m}, \mathbf{p}_{m1}, ..., \mathbf{p}_{mj}, \mathbf{p}_{0i}, ..., \mathbf{p}_{0N_0})$$

  mit $N_0 + N_m + 2$ Punkte. Die Punkte $\mathbf{p}_{0i}$ und $\mathbf{p}_{mj}$ erscheinen zweimal ! Deshalb sollte man als nächtes die Frontwinkel dieser Punkte bei ihrem ersten Erscheinen im Polygon $\Pi_0$ bestimmen und zuerst den Punkt vervollständigen (mit Dreiecken umgeben, s. Schritt S3), der den kleinsten Winkel besitzt, und dann den zweiten (s. 6.9e). Danach wird das erste Erscheinen dieser beiden Punkte aus dem aktuellen Frontpolygon gestrichen. $\mathbf{p}_{0i}, \mathbf{p}_{mj}$ dürfen bei späteren Abstandsprüfungen nicht mehr verwendet werden.

**Bemerkung:**

a) Für "einfache" Flächen kann die Abstandsprüfung weggelassen werden. (s. Beispiele: Kugel, 6-peak-Fläche.)

b) Vor der Abstandsprüfung sollte man Punkte mit Frontwinkeln kleiner (ungefähr) $86° = 1.5$ vervollständigen.

c) Aussetzen der Abstandsprüfung nach einer Teilung des aktuellen Frontpolygons oder, falls $|\Pi_0| < 10$.

1. "Schlechte"Nahpunkte, die durch ein schon trianguliertes Gebiet verbunden sind (Abb. 6.4), können dadurch entdeckt werden, indem man den Winkel $\omega$ am Punkt $\mathbf{p}_{0i}$ gemäß Schritt 2 berechnet, wobei man anstelle von $\mathbf{v}_2$ den Punkt $\mathbf{p}_{mj}$ verwendet. Das Punktepaar $\mathbf{p}_{0i}, \mathbf{p}_{mj}$ ist ein "schlechtes", wenn der so berechnete Winkel $\omega$ größer ist als der Frontwinkel am Punkt $\mathbf{p}_{0i}$.

2. Eine wesentliche Beschleunigung des Anstandstests erzielt man, durch Verwendung von "bounding boxes" für die Frontpolygone.

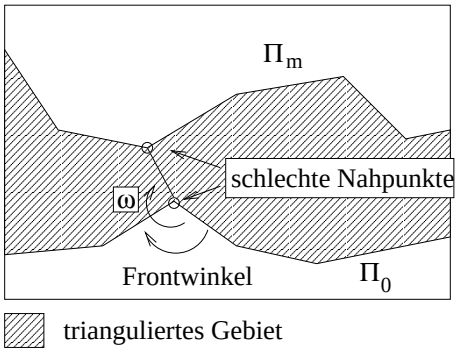



Abbildung 6.4: Schlechte Nahpunkte und ihre Entdeckung

### 6.1.7 Der Schritt S3

Es sei $\mathbf{p}_{0m}$ ein Punkt des aktuellen Frontpolygons $\Pi_0$ mit minimalem Frontwinkel $\omega$. *Vervollständige* die Triangulierung am Punkt $\mathbf{p}_{0m}$ auf die folgende Weise:

1. Bestimme die Nachbarn $\mathbf{v}_1, \mathbf{v}_2$ von $\mathbf{p}_{0m}$ (vgl. Schritt S1).

2. Bestimmung der Anzahl $n_t$ von Dreiecke, die erzeugt werden sollen:
   Es sei $n_t := \texttt{trunc}(3\omega/\pi) + 1$, $\Delta\omega := \omega/n_t$
   Korrektur von $\Delta\omega$ in extremen Fällen:
   Falls $\Delta\omega < 0.8$ und $n_t > 1$ dann $n_t \rightarrow n_t - 1$ und $\Delta\omega = \omega/n_t$.
   Falls $n_t = 1$ und $\Delta\omega > 0.8$ und $\|\mathbf{v}_1 - \mathbf{v}_2\| > 1.25\delta_t$ dann $n_t = 2$ und $\Delta\omega \rightarrow \Delta\omega/2$.
   Falls ( $\|\mathbf{v}_1 - \mathbf{p}_{0m}\|^2 < 0.2\ \delta_t^2$ oder $\|\mathbf{v}_2 - \mathbf{p}_{0m}\|^2 < 0.2\ \delta_t^2$ ), dann sei $n_t = 1$ (s. Abb. 6.5).

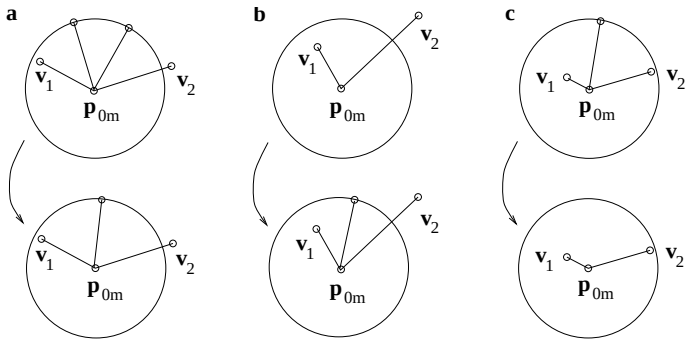



Abbildung 6.5: Korrekturen für extreme Fälle

3. Erzeugung von Dreiecken:
   Falls $n_t = 1$, dann erhalten wir 1 neues Dreieck: $(\mathbf{v}_1, \mathbf{v}_2, \mathbf{p}_{0m})$
   andernfalls seien $\mathbf{q}_0, \mathbf{q}_{n_t}$ die senkrechten Projektionen von $\mathbf{v}_1, \mathbf{v}_2$ in die Tangentialebene am Punkt $\mathbf{p}_{0m}$ und $\mathbf{q}_i$ sei der Punkt, der durch Drehung von $\mathbf{p}_{0m} + \delta_t(\mathbf{q}_0 - \mathbf{p}_{0m})/\|\mathbf{q}_0 - \mathbf{p}_{0m}\|$ um den Winkel $i\Delta\omega$ mit der Normale im Flächenpunkt $\mathbf{p}_{0m}$ als Drehachse entsteht. (Falls eine globale bounding box aktiv ist, wird die Kante $\mathbf{p}_{0m}\mathbf{q}_i$ gekürzt und die Variable `boarder_point` des entsprechenden neuen Flächenpunktes auf `true` gesetzt. Randpunkte werden bei weiteren Betrachtungen ignoriert.) Anwendung von `surfacepoint` auf $\mathbf{q}_i$ liefert die neuen Punkte $\mathbf{p}_{N+i}$, $i = 1, .., n_t - 1$, und die $n_t$ neuen Dreiecke:
   $(\mathbf{v}_1, \mathbf{p}_{N+1}, \mathbf{p}_{0m}), (\mathbf{p}_{N+1}, \mathbf{p}_{N+2}, \mathbf{p}_{0m}), ..., (\mathbf{p}_{N+n_t-1}, \mathbf{v}_2, \mathbf{p}_{0m})$.

4. Erneuere das aktuelle Frontpolygon:
   Streiche den Punkt $\mathbf{p}_{0m}$ und, falls $n_t > 1$, füge an seine Stelle die neuen Punkte $\mathbf{p}_{N+1}, ..., \mathbf{p}_{N+n_t-1}$

ein. Alle boolschen Variablen `angle_changed` der Punkte $\mathbf{v}_1, \mathbf{v}_2, \mathbf{r}_1, ..., \mathbf{r}_{n_t-1}$ werden auf `true` gesetzt.

## 6.2 Beispiele

**Beispiel 6.1** *Kugel*
*Triangulierung der Kugel* $x^2 + y^2 + z^2 - 4 = 0$ *mit Startpunkt (1,1,1) und Kantenlänge* $\delta_t = 0.3$. *Die folgenden Abbildungen zeigen die ersten vier aktuelle Frontpolygone und die Situation nach der Erzeugung von 101 und 1531 Dreiecke. Die volle Triangulierung der Kugel umfaßt 1544 Dreiecke.*

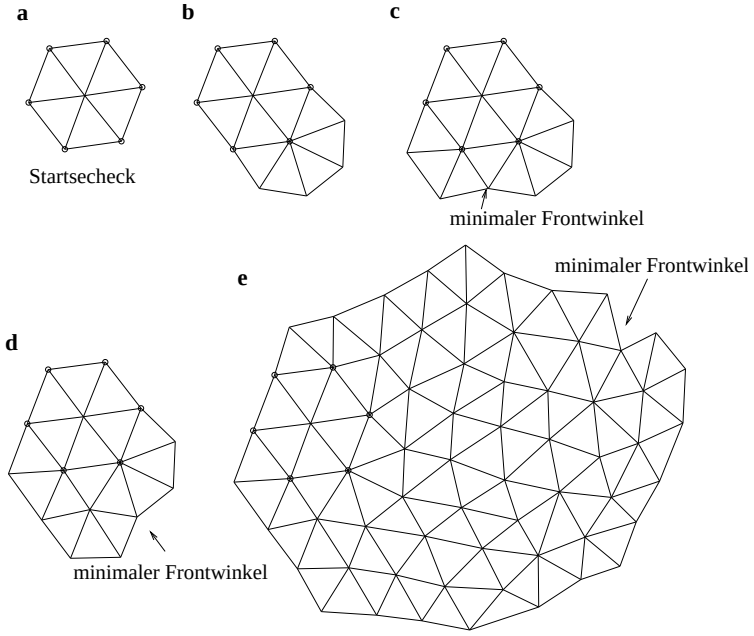



Abbildung 6.6: Triangulierung einer Kugel

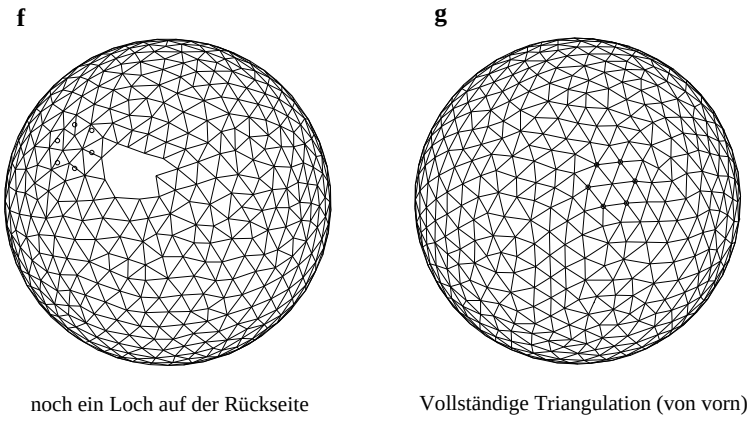



Abbildung 6.7: Triangulierung einer Kugel (Forts.)

**Beispiel 6.2** *Zylinder*
*Triangulierung des Zylinders* $x^2 + y^2 - 1 = 0$*,*

1. *mit Starpukt* $(1, 0, 0)$ *und* $\delta_t = 0.2$*. Abb. 6.8a,b zeigt die Triangulation vor und nach der ersten Teilung des aktuellen Frontpolygons. Der Zylinder ist durch eine bounding box beschränkt.*

2. *mit Punkte auf dem Deckelkreis als erstes Frontpolygon und Punkte auf dem Bodenkreis als weiteres (begrenzendes) Frontpolygon (Abb. 6.8).*

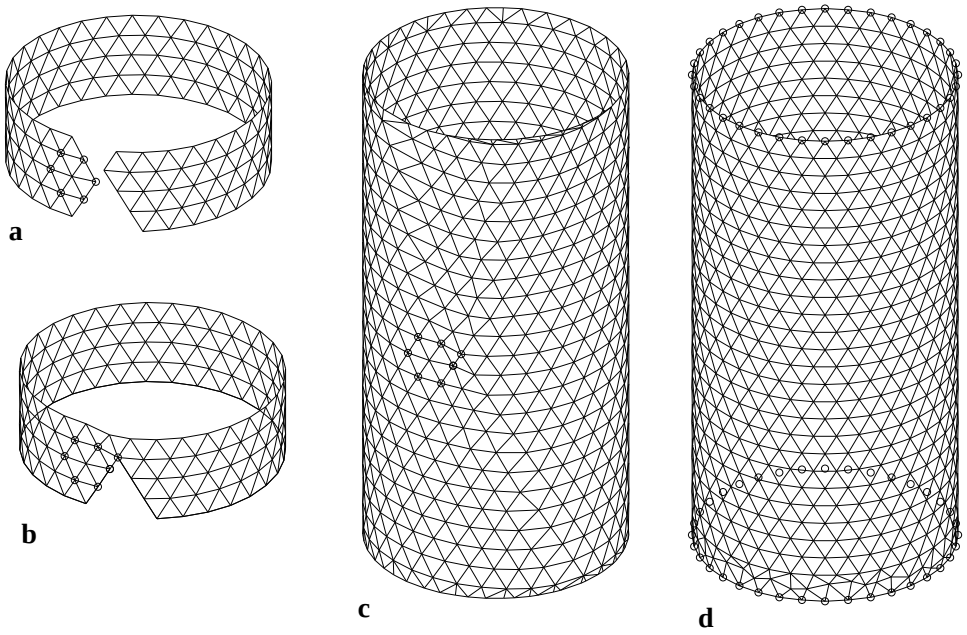



Abbildung 6.8: Triangulierung eines Zylinders

**Beispiel 6.3** *Torus*
*Triangulierung des Torus*

$$(x^2 + y^2 + z^2 + r^2 - a^2)^2 - 4r^2(x^2 + y^2) = 0,\ r = 1, a = 0.35$$

*mit* $\delta_t = 0.1$*,*

1. *Startpunkt* $(1, 0, 0.5)$*. Abb. 6.9d,e zeigt die Triangulierung vor und nach der Vereinigung des aktuellen Frontpolygons mit dem weiteren Frontpolygon, das bei einer früheren Teilung des aktuellen Frontpolygons erzeugt wurde. Abb. 6.10f zeigt die vollständige Triangulation.*

2. *Punkte auf einem Kreis als erstes aktuelles Frontpolygon und*

   (a) *dasselbe Polygon (mit umgekehrter Orientierung) als weiteres (begrenzendes) Frontpolygon (Abb. 6.10h).*

   (b) *einem zweiten Polygon als weiteres (begrenzendes) Polygon (Abb. 6.10g). (Torusteil)*

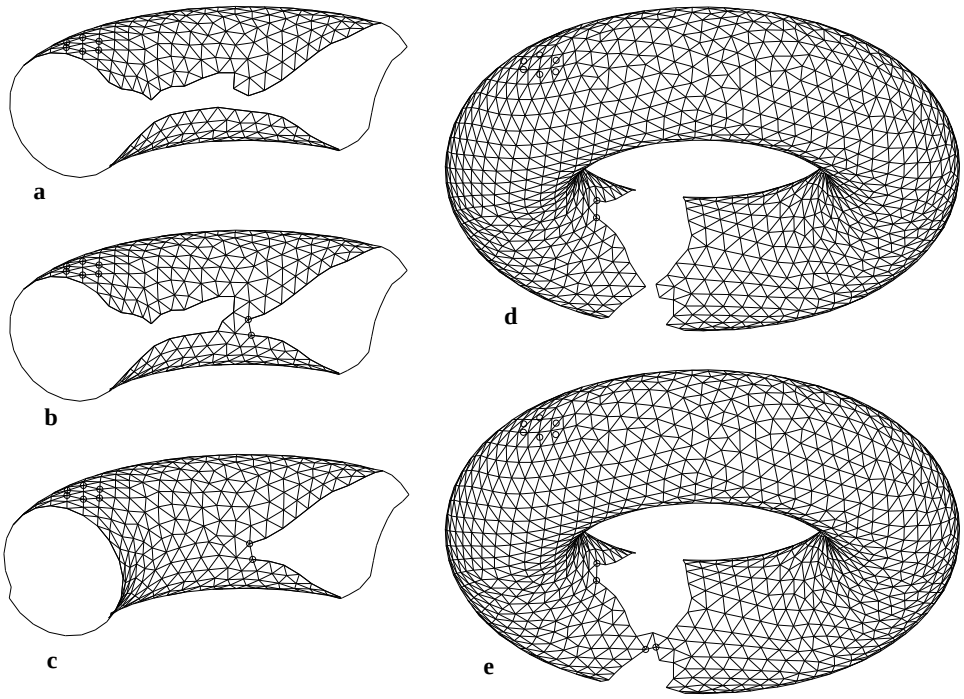



Abbildung 6.9: Triangulierung eines Torus

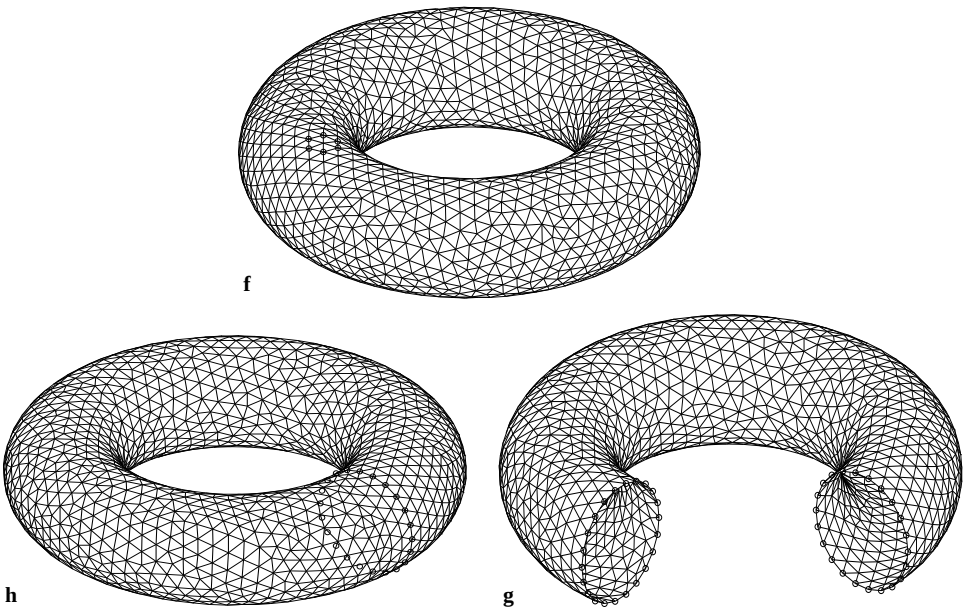



Abbildung 6.10: Triangulierung eines Torus (Forts.)

**Beispiel 6.4** *Fläche vom Geschlecht 3*
*Triangulierung einer impliziten Fläche vom Geschlecht 3 mit der Gleichung*

$$r_z^4 z^2 - (1 - (x/r_x)^2 - (y/r_y)^2)((x - x_1)^2 + y^2 - r_1^2)(x^2 + y^2 - r_1^2)((x + x_1)^2 + y^2 - r_1^2) = 0,$$

$r_x = 6, r_y = 3.5, r_z = 4, r_1 = 1.2, x_1 = 3.9$ *und Startpunkt* $(0, 3, 0)$*, Kantenlänge* $\delta_t = 0.3$*. Die Trianagulierung besteht aus 7354 Dreiecke.*

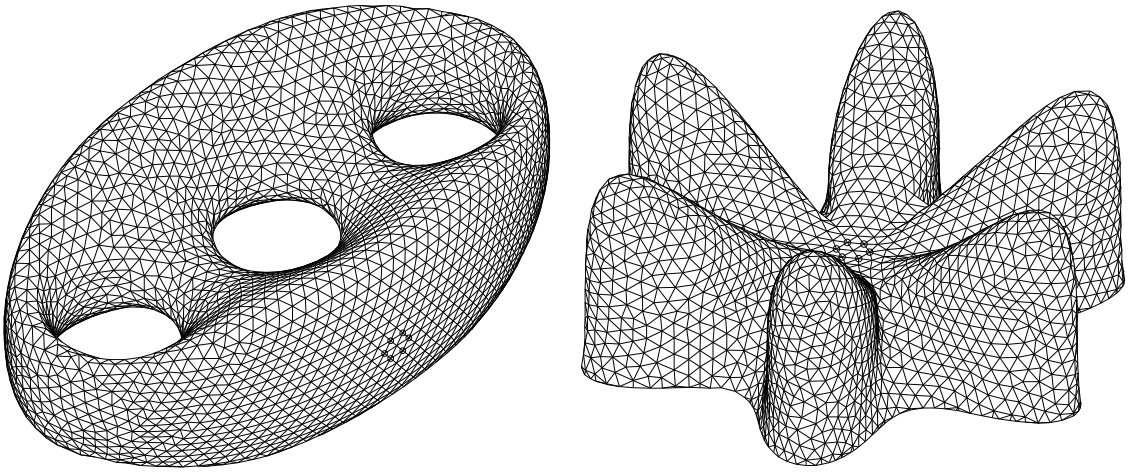


Abbildung 6.11: Fläche vom Geschlecht 3 bzw. Fläche mit 6 Peaks

**Beispiel 6.5** *6-peak-Fläche*
*Die Triangulierung der ziemlich komplizierten impliziten Fläche*

$$(3x^2 - y^2)^2 y^2 - (x^2 + y^2)^4 - z^3 - 0.001z = 0$$

*ist möglich ohne Teilung und Vereinigung.*

Zum Schluß noch ein Beispiel, dessen implizite Darstellung Normalformen umfaßt (s. Vorlesung). Die Normalformen sind überhaupt der Schlüssel zur Triangulierung fast beliebiger Flächen. Mit Hilfe von Normalformen können sogar parametrisierte Flächen mit dem hier vorgestellten Algorithmus trianguliert werden.

**Beispiel 6.6** *Gegeben seien*

1. *die implizite Fläche* $\Phi_1 :\ (x-2)^4 + y^4 - r_1^4 = 0,\ r_1 = 2,$

2. *das parametrisierte Flächenstück*
   $\Phi_2 :\ \mathbf{x} = (10v - 5, 10u - 5, 6(u - u^2 + v - v^2)),\ 0 \le u \le 1, 0 \le v \le 0.8,$

3. *das parametrisierte Flächenstück*
   $\Phi_3 :\ \mathbf{x} = (6(u - u^2 + v - v^2) - 5, 10u - 5, 10v - 5),\ 0 \le u \le 1, 0.5 \le v \le 1.$

$h_1(\mathbf{x}) = 0, h_2(\mathbf{x}) = 0, h_3(\mathbf{x}) = 0$ *seien ihre Normalformen. Die implizite Fläche* $\Phi :\ f(\mathbf{x}) := h_1(\mathbf{x})h_2(\mathbf{x})h_3(\mathbf{x}) = c,\ c > 0$ *ist eine glatte Approximation des Flächenkomplexes* $\Phi_1, \Phi_2, \Phi_3$*. Die Approximation ist unabhängig von ihren Darstellungen.*
*Abb. 6.12 zeigt eine Triangulierung von* $f(\mathbf{x}) = c$ *für* $c = 0.2$*.*

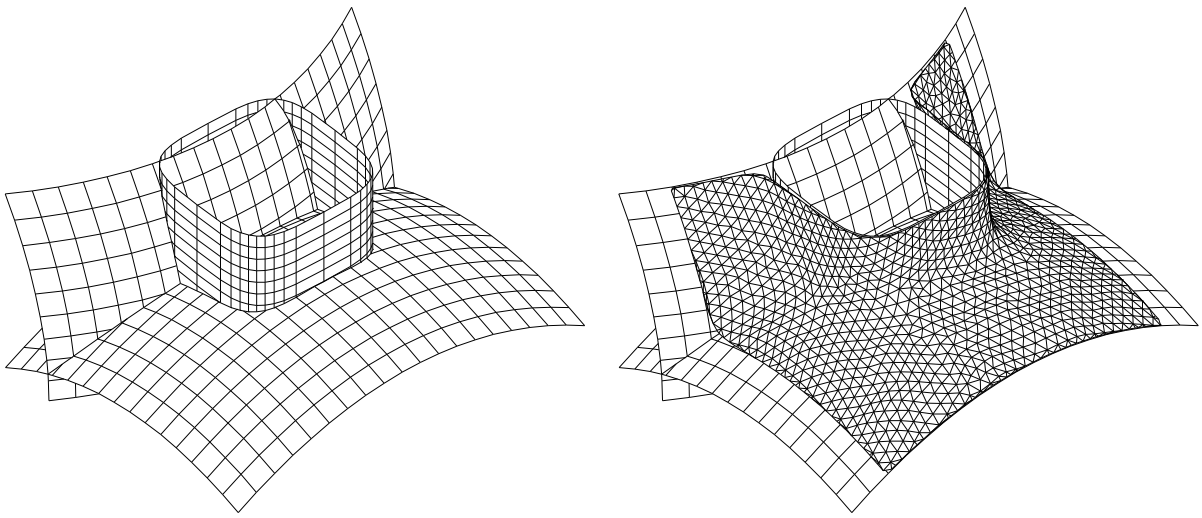


Abbildung 6.12: zu Beispiel 6.6