Dokumentation zu MGA-TEX

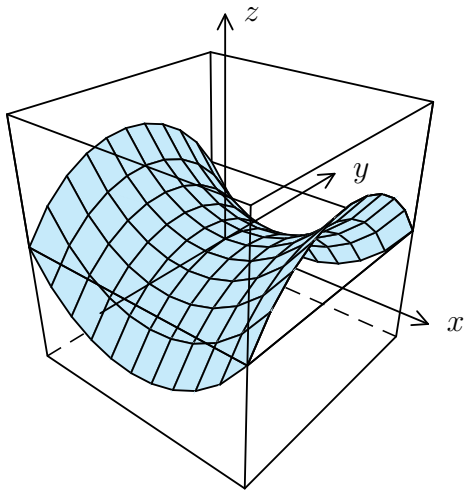
Version 2.4

– ausgedruckt am 30. Juni 2013 –

K. Fritzsche – März 2006

Graphische Darstellungen mit LATEX und pdfLATEX

Anhang 2: Das Modul „mg24_3d.sty“ (3D-Darstellungen)



Inhaltsverzeichnis

Kapitel 1

3d-Graphik

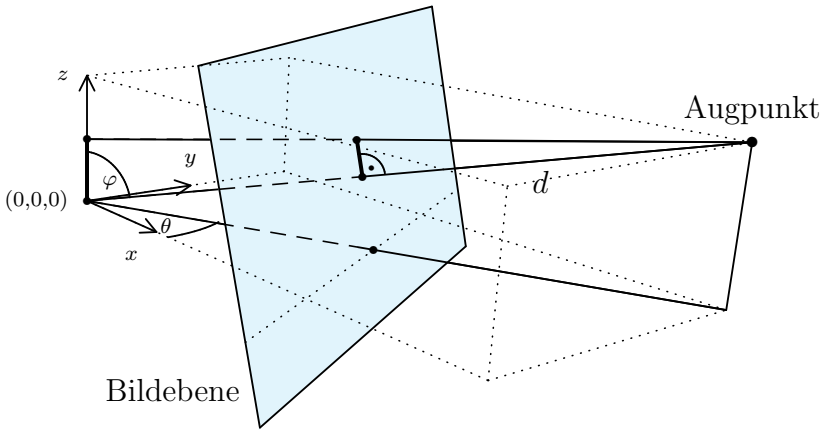
§ 1 Das Koordinatensystem

Mit der Option `[3d]` lädt man das 3d-Modul.

Ein 3-dimensionales Bild (in einer `\InitGraph`...–`\CloseGraph`-Umgebung) wird eingeleitet mit dem Befehl

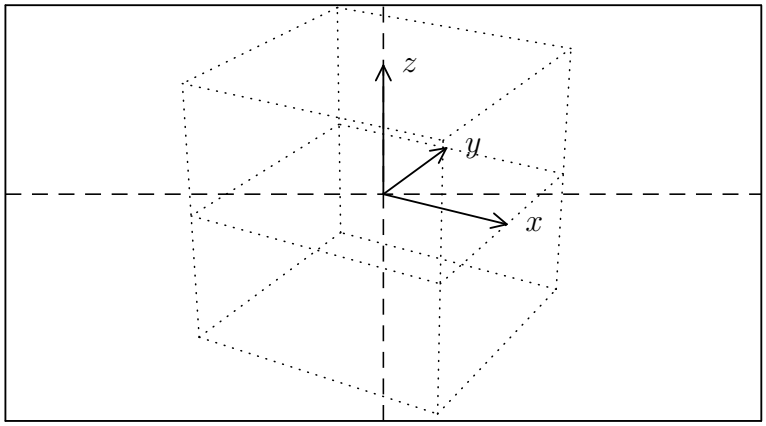
`\Viewpoint`$(\theta,\ \varphi,\ \varrho,\ d)[k]$

Damit wird der „Augpunkt“ in einem 3-dimensionalen (Polar-)Koordinatensystem festgelegt. θ ist der Winkel zur x-Achse, φ der Winkel zur z-Achse (jeweils in Grad gemessen), ϱ die Entfernung des Augpunktes zum Koordinaten-Nullpunkt (der mit dem Nullpunkt aus dem `\InitGraph`-Befehl übereinstimmt) und d die „Distanz“, die Entfernung des Augpunktes zur Bildebene.



Wenn die Bilder zu klein werden, kann man mit dem Faktor k für eine geeignete Vergrößerung sorgen. Versuchsweise sollte man zunächst $k = 2$ wählen.

Das folgende Bild zeigt, wie sich eine perspektivische Darstellung mit $\theta = 300$, $\varphi = 65$, $\varrho = 10$, $d = 9$ und $k = 2$ in ein Bild mit den Abmessungen 10×5.5 (cm) und dem Nullpunkt bei $(5, 3)$ einfügt.



Für den Anfang reicht eventuell auch schon der Befehl \StandardViewpoint , der die Parameter wie folgt besetzt:

$$\theta := 60,\ \varphi := 60,\ \varrho := 5,\ d := 6 \text{ und } k := 2.$$

§ 2 Die ersten 3D-Zeichenbefehle

Es gibt einen imaginären 3D-Cursor.

\DDLineAt$(x_1, y_1, z_1)(x_2, y_2, z_2)$ zeichnet perspektivisch die Strecke von (x_1, y_1, z_1) nach (x_2, y_2, z_2) und setzt den Cursor auf den Endpunkt.

\DDLineTo$(x, y, z)$ zeichnet eine Strecke von der aktuellen Cursorposition nach (x, y, z).

\DDRelLine$(\Delta x, \Delta y, \Delta z)$ zeichnet eine Linie vom aktuellen Punkt (x_0, y_0, z_0) nach $(x_0 + \Delta x, y_0 + \Delta y, z_0 + \Delta z)$.

\DDPolyLine$(x_1, y_1, z_1)(x_2, y_2, z_2) \ldots (x_n, y_n, z_n)$ zeichnet einen 3-dimensionalen Linienzug.

\DDJoinPoints$(x_1, y_1, z_1)(x_2, y_2, z_2) \ldots (x_n, y_n, z_n)$ arbeitet wie \DDPolyLine..., setzt dabei aber noch jeweils das aktuelle Punktsymbol.

\DDTriangle$(x_1, y_1, z_1)(x_2, y_2, z_2)(x_3, y_3, z_3)$ zeichnet ein Dreieck. Man beachte, dass drei Punkte im Raum immer auf einer Ebene liegen.

\DreiDeBox$(x_1, y_1, z_1)(x_2, y_2, z_2)$ zeichnet den Quader $[x_1, x_2] \times [y_1, y_2] \times [z_1, z_2]$

mit den Ecken (x_1, y_1, z_1), (x_2, y_1, z_1), (x_2, y_2, z_1) und (x_1, y_2, z_1),

sowie (x_1, y_1, z_2), (x_2, y_1, z_2), (x_2, y_2, z_2) und (x_1, y_2, z_2).

\DDMoveTo$(x, y, z)$ bewegt den Cursor auf die Position (x, y, z).

\DDArrowAt$(x_1, y_1, z_1)(x_2, y_2, z_2)$ zeichnet einen Pfeil zwischen den angegebenen Punkten.

\DDArrowHead$(x_1, y_1, z_1)(x_2, y_2, z_2)$ zeichnet nur die Spitze des Pfeils.

`\DDArrowTo`(x, y, z) zeichnet einen Pfeil vom aktuellen Punkt nach (x, y, z).

`\DDRelArrow`$(\Delta x, \Delta y, \Delta z)$ zeichnet einen Pfeil vom aktuellen Punkt (x_0, y_0, z_0) nach $(x_0 + \Delta x, y_0 + \Delta y, z_0 + \Delta z)$, `\DDRelArrowHead`$(\Delta x, \Delta y, \Delta z)$ nur die Spitze.

`\DDPoint` (bzw. `\DDPointAt`(x, y, z)) zeichnet einen Punkt an der aktuellen (bzw. der angegebenen) Position,

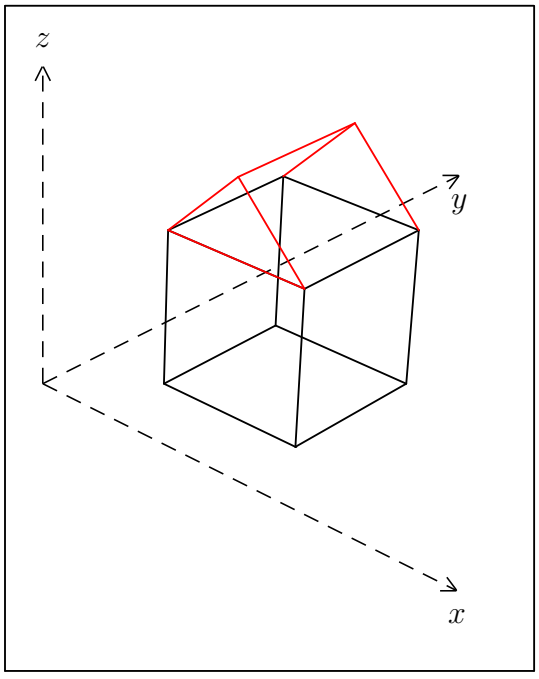
`\DDMedPoint` (bzw. `\DDMedPointAt`(x, y, z)) zeichnet einen etwas dickeren Punkt,

`\DDBigPoint` (bzw. `\DDBigPointAt`(x, y, z)) zeichnet einen dicken Punkt.

Im *Kompatibilitätsmodus* (der zur Zeit noch automatisch eingestellt ist) stehen noch einige ältere Befehle zur Verfügung:

- Der Befehl `\MoveDreiDeTo`(x_1, y_1, z_1) setzt den Cursor auf den Punkt (x_1, y_1, z_1). Mit Hilfe der Befehle `\Store`$(\ldots)$, `\MoveToLoc`$(\ldots)$ und `\LineToLoc`$(\ldots)$ könnte man nun schon auf etwas primitive Weise 3-dimensional zeichnen.
- `\DrawDreiDe`$(x_1, y_1, z_1)(x_2, y_2, z_2)$ zeichnet die Strecke von (x_1, y_1, z_1) nach (x_2, y_2, z_2).
- `\DrawDreiDeTo`(x, y, z) benutzt als Ausgangspunkt die letzte Cursor-Position und zeichnet die Strecke nach (x, y, z).
- `\DreiDeArrowTo`(x, y, z) zeichnet einen Pfeil von der augenblicklichen Cursor-Position zum Punkt (x, y, z).

Beispiel:

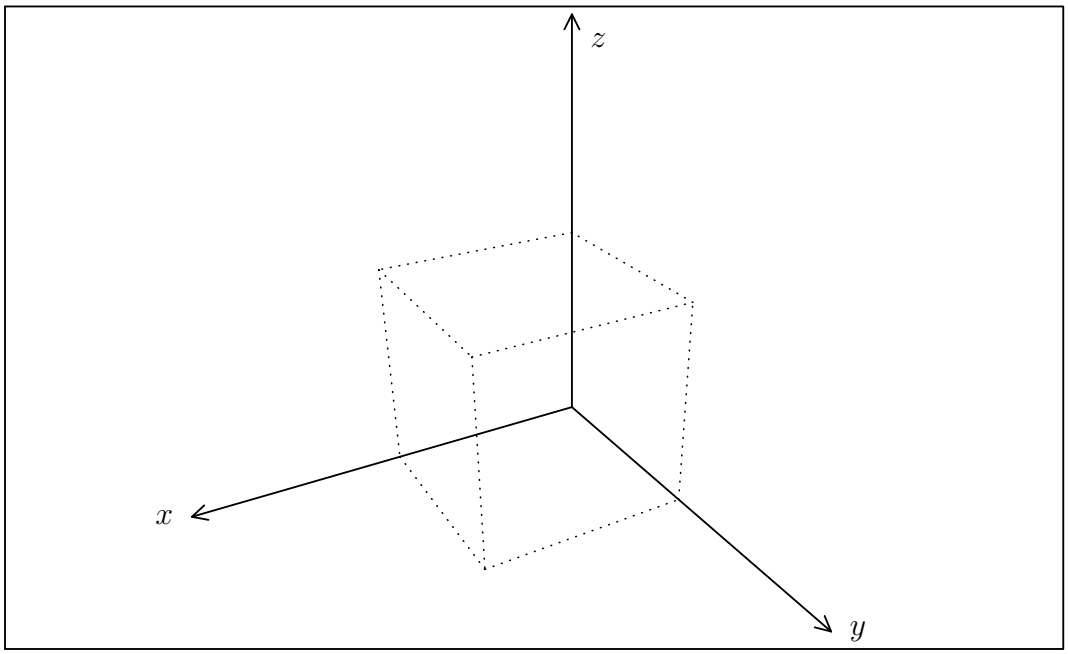



```
\InitGraph{7}{8.8}{0.5}{3.8}{1cm}
\DrawBoundary
\Viewpoint(315,60,15,17)[2]
\SetDashed
\DDArrowAt(0,0,0)(3,0,0)
\Text[b]{$x$}
\DDArrowAt(0,0,0)(0,4,0)
\Text[b]{$y$}
\DDArrowAt(0,0,0)(0,0,2)
\Text[t]{$z$}
\SetNormal
\DreiDeBox(0.5,0.5,0)(1.5,1.5,1)
\SetRed
\DDTriangle(0.5,0.5,1)%
   (1.5,0.5,1)(1,0.5,1.5)
\DDLineAt(1,0.5,1.5)(1,1.5,1.5)
\DDLineTo(1.5,1.5,1)
\DDLineAt(1,1.5,1.5)(0.5,1.5,1)
\ResetColor
\CloseGraph
```

§ 3 Der Einheitswürfel

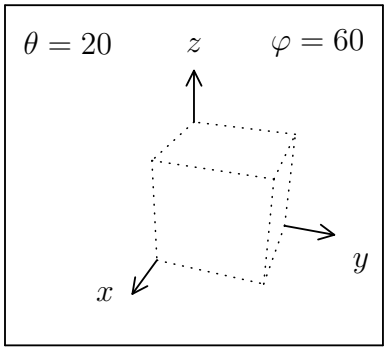
Mit folgenden Befehlen zeichnen wir einen Einheitswürfel in einem 3D-Koordinatensystem.

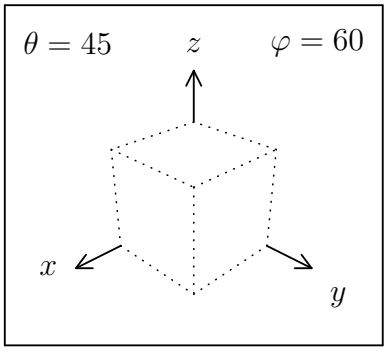
```
\InitGraph{14}{8.5}{7.5}{3.2}{1cm}
\DrawBoundary
\StandardViewpoint
\DDArrowAt(0,0,0)(2,0,0) \Text[l]{$x$}
\DDArrowAt(0,0,0)(0,2,0) \Text[r]{$y$}
\DDArrowAt(0,0,0)(0,0,2) \Text[rb]{$z$}
\SetDotted
\DDLineAt(0,0,1)(1,0,1)
\DDLineTo(1,0,0)
\DDLineTo(1,1,0)
\DDLineTo(0,1,0)
\DDLineTo(0,1,1)
\DDLineTo(0,0,1)
\DDLineAt(1,0,1)(1,1,1)
\DDLineTo(0,1,1)
\DDLineAt(1,1,1)(1,1,0)
\CloseGraph
```

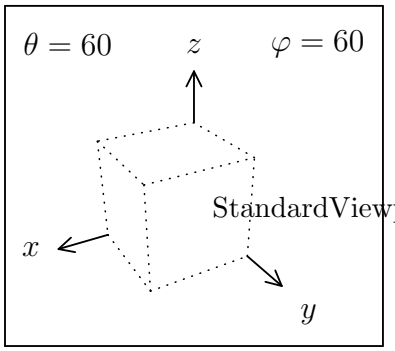


Nun soll der Einfluß der Viewpoint-Parameter gezeigt werden. Wir behalten $\varrho = 5$ und die Distanz $d = 6$ aus dem Standard-Viewpoint bei. Die Korrekturkonstante k setzen wir $= 1$, um die Größe zu verringern. Die Winkel θ und φ werden variiert.

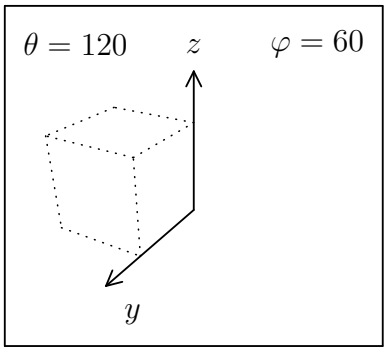
Zunächst setzen wir $\varphi = 60$ und variieren nur θ:

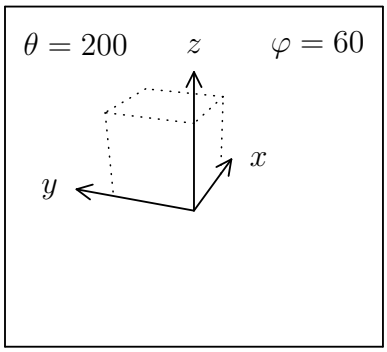


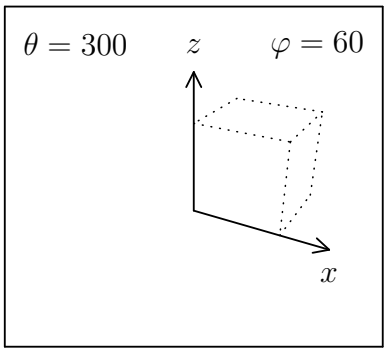




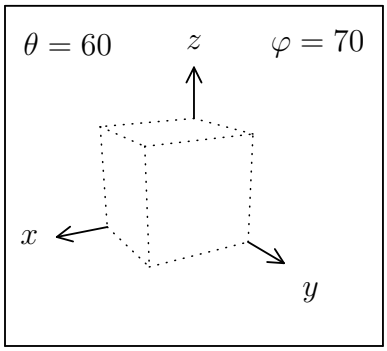
StandardViewpoint

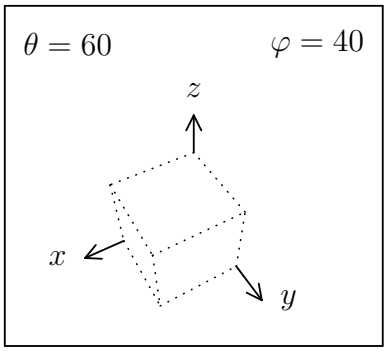


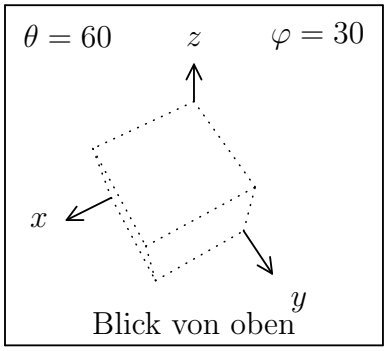


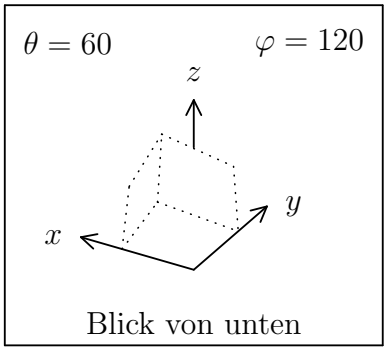


Nun halten wir $\theta = 60$ fest und variieren φ.



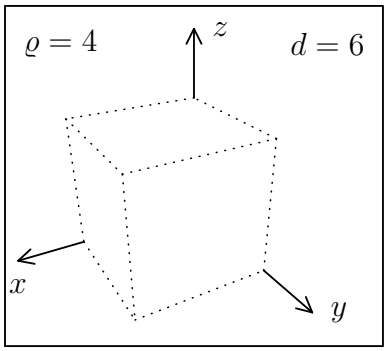


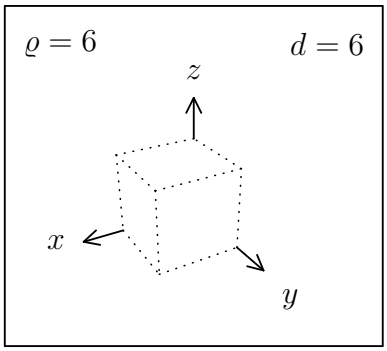




Man beachte, dass kein Hidden-Line-Algorithmus geliefert wird! Sie müssen selbst dafür sorgen, dass verdeckte Linien nicht gezeichnet werden. In einzelnen Fällen kann man sich so behelfen:
`\DreiDeObstruction`$(x_1, y_1, z_1)(x_2, y_2, z_2)$ speichert die Gerade durch die beiden angegebenen Punkte als „Hindernis“.
`\ClippedDreiDe`$(x_3, y_3, z_3)(x_4, y_4, z_4)$ zeichnet die Strecke zwischen den angegebenen Punkten, aber nur bis zu dem unmittelbar vorher gespeicherten Hindernis.

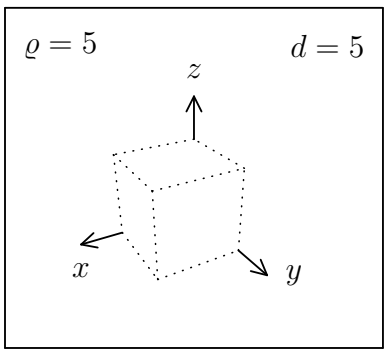
Ist $\theta = 60$, $\varphi = 60$ und die Distanz $d = 6$ konstant, so wirkt sich wechselndes ϱ folgendermaßen aus:

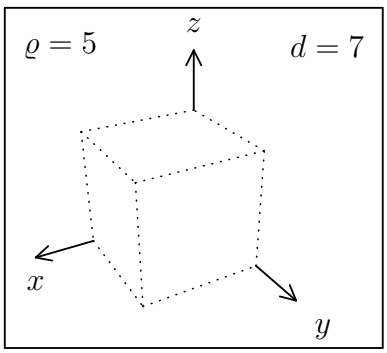




Im ersten Fall liegt der Nullpunkt **vor** der Bildebene (beim Standard-Viewpoint liegt er dahinter). Im zweiten Fall liegt der Nullpunkt genau auf der Bildebene.

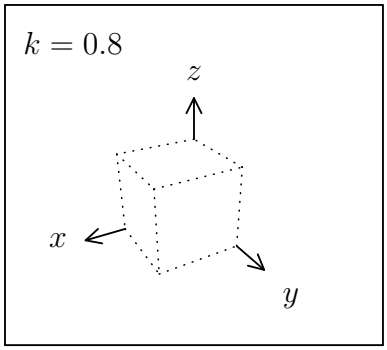
Ist $\theta = 60$, $\varphi = 60$ und der Abstand $\varrho = 5$ (zwischen Augpunkt und Koordinaten-Ursprung) konstant, so wirkt sich wechselnde Distanz d folgendermaßen aus:

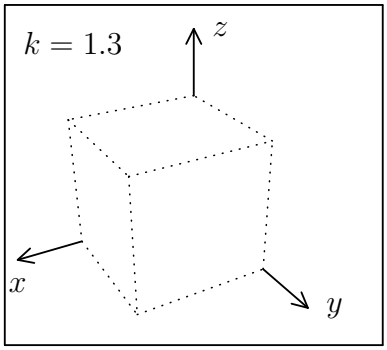




Im ersten Fall liegt der Nullpunkt auf der Bildebene, im zweiten dahinter.

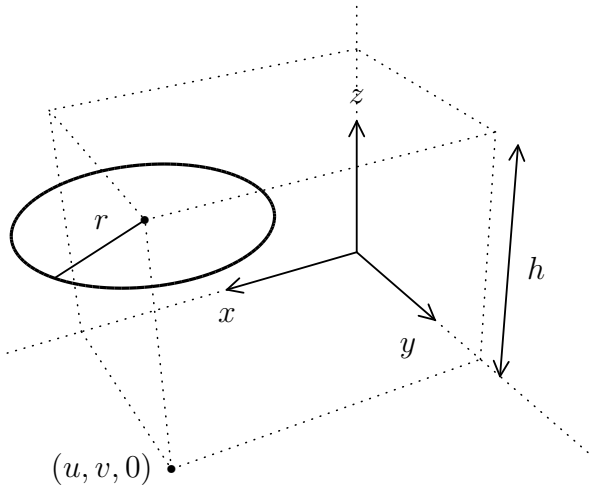
Je kleiner ϱ gewählt wird, desto stärker ist die perspektivische Verzerrung. Bei festem ϱ dienen d und die Korrektur-Konstante k beide als Vergrößerungsfaktoren. Hier zwei Beispiele zum Einfluss von k:





§ 4 Horizontale und vertikale Kreise

\PCircle$(u, v, h, r)$ (mit „P“ wie „plane“) liefert eine **horizontale Kreislinie** in der durch die Höhe h bestimmten horizontalen Ebene mit dem Mittelpunkt (u, v) und dem Radius r.

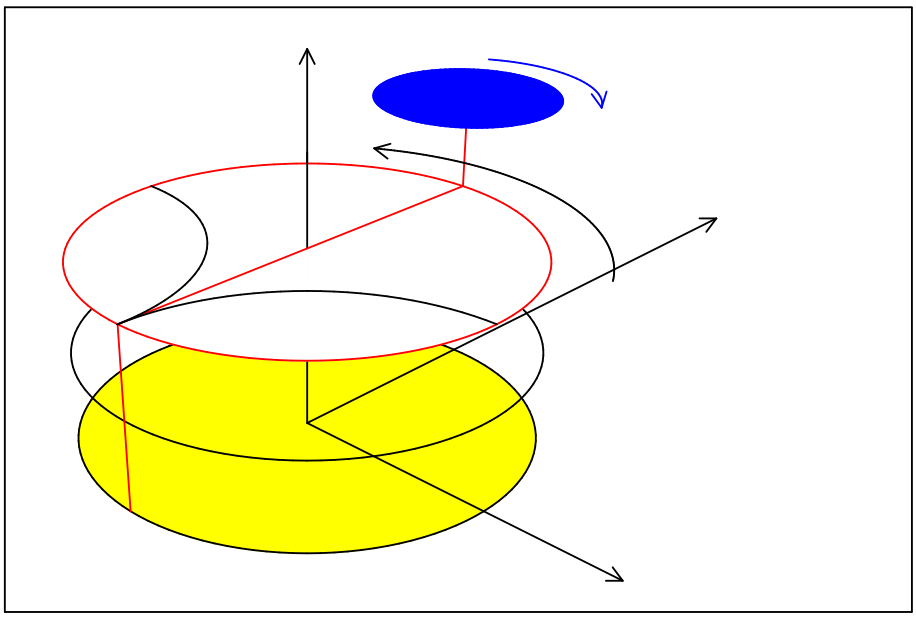


\PArc$(u, v, h, r)(w_1, w_2)$ ergibt einen horizontalen Kreisbogen zwischen den Winkeln w_1 und w_2. Die Parameter u, v, h, r haben die gleiche Bedeutung wie oben, die Winkel beziehen sich beide auf die positive x-Achse.

\PArrowArc$(u, v, h, r)(w_1, w_2)[w]$ setzt an den Kreisbogen noch eine Pfeilspitze. Der optionale Parameter w steuert die Position der Pfeilspitze. Standard ist $w = 30$.

Mit \PDisk$(u, v, h, r)$ kann man eine gefüllte horizontale Kreisscheibe in der gewählten Farbe zeichnen, \PSektor$(u, v, h, r)(w_1, w_2)$ liefert einen Kreissektor.

Hier ist ein **Beispiel** mit ein paar Spielereien:



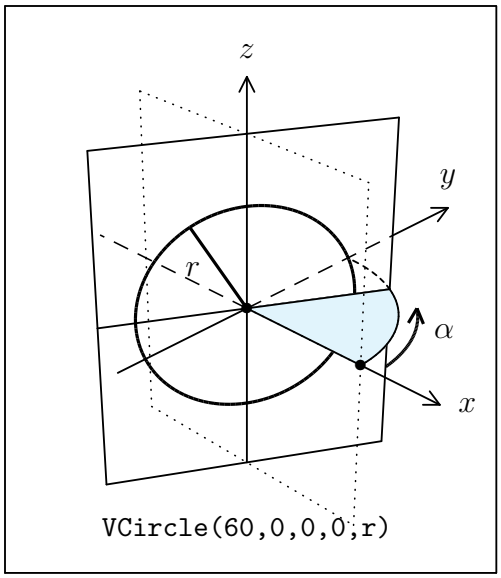
```
\InitGraph{12}{8}{4}{2.5}{1cm}
\DrawBoundary
\Viewpoint(315,60,8,10)[2]
\SetYellow \PDisk(0,0,0,1.2) \ResetColor
\DDArrowAt(0,0,0)(2,0,0)
\DDArrowAt(0,0,0)(0,4,0)
\DDArrowAt(0,0,0)(0,0,2)
\PCircle(0,0,0,1.2) \PCircle(0,0,0.5,1.2)
\SetWhite \PDisk(0,0,1,1.2)
\SetRed \PCircle(0,0,1,1.2) \ResetColor
\DDLineAt(0,0,1)(0,0,1.5)
\SetRed \DDLineAt(0,-1.2,0)(0,-1.2,1)
\DDLineTo(0,1.2,1) \DDLineTo(0,1.2,1.5)
\SetBlue \PDisk(0,1.2,1.5,0.5)
\PArrowArc(0,1.2,1.5,0.7)(120,30)[17] \ResetColor
\PArrowArc(0,0,1,1.5)(30,120)
\PArc(-1.2,-1.2,1,1.2)(0,90) \PArc(1.2,-1.2,1,1.2)(90,180)
\CloseGraph
```

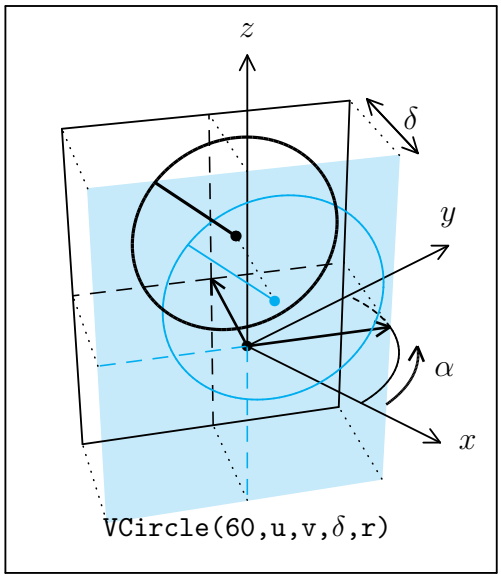
`\PMoveTo`$(u, v, h, r)(w)$ bewegt den Cursor auf die durch den Winkel w festgelegte Position des horizontalen Kreises.

Mit `\VCircle`$(\alpha, u, v, \delta, r)$ zeichnet man die **vertikale Kreislinie**

$$t \mapsto \big((u + r\cos t)\cos\alpha - \delta\sin\alpha, (u + r\cos t)\sin\alpha + \delta\cos\alpha, v + r\sin t\big).$$

Diese Kreislinie liegt in einer auf der x-y-Ebene senkrecht stehenden Ebene, die man folgendermaßen gewinnt: Zunächst wird die x-z-Ebene um den Winkel α (gegen die positive x-Achse) gedreht und dann um die Strecke δ verschoben. Der Nullpunkt liegt dann an der Stelle $(-\delta\sin\alpha, \delta\cos\alpha, 0)$.





Auf den neuen Nullpunkt beziehen sich die Koordinaten (u, v) des Mittelpunktes des Kreises (in der linken Skizze ist $(u, v) = (0, 0)$, in der rechten Skizze ist $(u, v) = (0.15, 0.25)$). Der Radius des Kreises wird mit r bezeichnet.

\VArc$(\alpha, u, v, d, r)(w_1, w_2)$ liefert einen vertikalen Kreisbogen. Die Parameter α, u, v, d und r haben die gleiche Bedeutung wie bei \VCircle , w_1 und w_2 begrenzen den Winkelbereich.

\VDisk$(\alpha, u, v, d, r)$ malt eine gefüllte vertikale Kreisscheibe in der gewählten Farbe, \VSektor$(\alpha, u, v, d, r)(w_1, w_2)$ einen Kreissektor im angegebenen Winkelbereich.

Der Befehl \VArrowArc$(\alpha, u, v, d, r)(w_1, w_2)$ ist noch nicht implementiert!

Zum Füllen von ebenen Flächen gibt es den Befehl

$$\texttt{\PaintDDTriangle}(x_1, x_2, x_3)(x_4, x_5, x_6)(x_7, x_8, x_9),$$

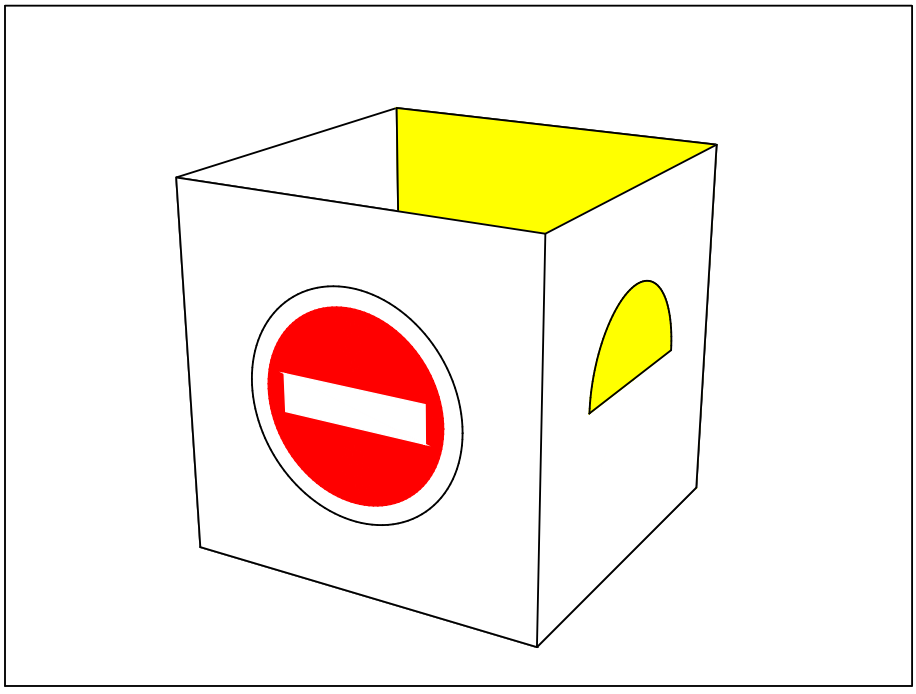
der das Dreieck mit den angegebenen Ecken in der aktuellen Farbe füllt. Mit zwei Dreiecken füllt man ein Rechteck. Man beachte die Reihenfolge bei solchen Operationen.

Hat man drei Punkte mit den Befehlen \Store$(P_1)$, ..., \Store$(P_3)$ abgespeichert, so kann man ein gefülltes Dreieck mit dem Befehl

$$\texttt{\PaintStoredTriangle}(P_1)(P_2)(P_3)$$

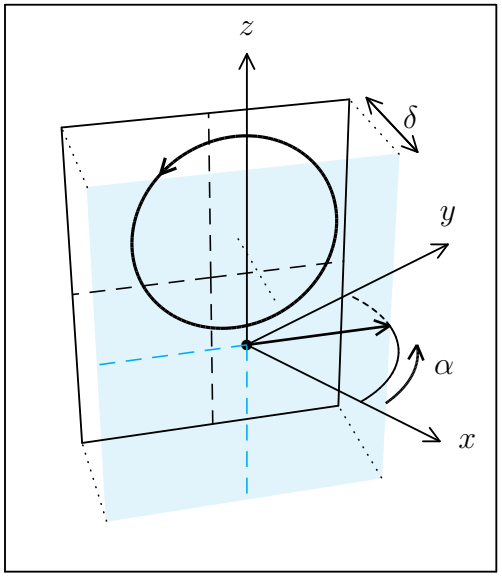
zeichnen.

Beispiel:



```
\InitGraph{12}{9}{6}{4.5}{1cm}
\DrawBoundary
\Viewpoint(300,70,8,10)[2]
\DreiDeBox(-1,-1,-1)(1,1,1)
\SetYellow
\PaintDDTriangle(1,1,-1)(1,1,1)(-1,1,1)
\PaintDDTriangle(-1,1,-1)(1,1,-1)(-1,1,1)
\ResetColor
\DDPolyLine(1,1,1)(-1,1,1)(-1,1,-1)
\SetWhite
\PaintDDTriangle(-1,-1,-1)(1,-1,-1)(-1,-1,1)
\PaintDDTriangle(1,-1,-1)(1,-1,1)(-1,-1,1)
\PaintDDTriangle(1,-1,-1)(1,1,1)(1,-1,1)
\PaintDDTriangle(1,-1,-1)(1,1,-1)(1,1,1)
\ResetColor
\DDPolyLine(-1,-1,-1)(1,-1,-1)(1,-1,1)(-1,-1,1)(-1,-1,-1)
\DDPolyLine(1,-1,-1)(1,1,-1)(1,1,1)(1,-1,1)
\SetYellow \VSektor(90,0,0,-1,0.5)(0,180) \ResetColor
\VArc(90,0,0,-1,0.5)(0,180)
\DDLineAt(1,-0.5,0)(1,0.5,0)
\VCircle(0,0,0,-1,0.6)
\SetRed \VDisk(0,0,0,-1,0.5) \SetWhite
\PaintDDTriangle(0.4,-1,-0.1)(0.4,-1,0.1)(-0.4,-1,0.1)
\PaintDDTriangle(-0.4,-1,-0.1)(0.4,-1,-0.1)(-0.4,-1,0.1)
\CloseGraph
```

\VMoveTo$(\alpha, u, v, d, r)(w)$ bewegt den Cursor auf die durch den Winkel w bestimmte Position auf dem vertikalen Kreis.



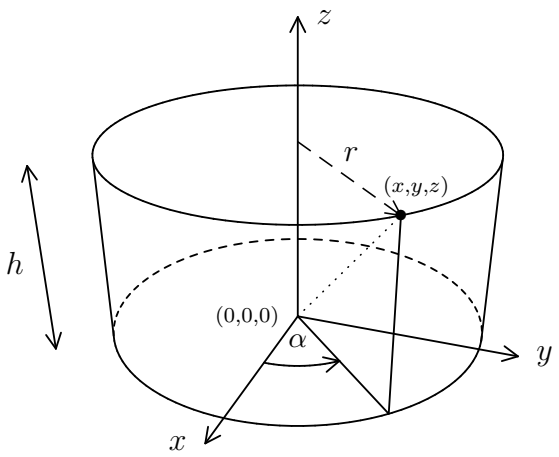
Um etwa einen Pfeil (mit Spitze an der Position φ) auf einen vertikalen Kreis zu setzen, kann man z.B. folgendermaßen vorgehen:

\VCircle$(\alpha, u, v, \delta, r)$
\VMoveTo$(\alpha, u, v, \delta, \sqrt{2}\,r)(\varphi - 45)$
\Store$(A)$
\VMoveTo$(\alpha, u, v, \delta, r)(\varphi)$
\Store$(B)$
\MoveToLoc$(A)$
\ArrowHeadToLoc$(B)$

§ 5 Kugel- und Zylinderkoordinaten

Es ist auch möglich, beim Zeichnen mit **Zylinderkoordinaten** zu arbeiten. Dabei werden die Parameter (r, α, h) verwendet. r ist der Radius eines Zylinders um die z-Achse, α der Winkel gegen die positive x-Achse und h die Höhe über der x-y-Ebene. Dann besteht die Beziehung

$$x = r\cos\alpha, \quad y = r\sin\alpha, \quad z = h.$$



`\DDZylMoveTo`(r, α, h) bewegt den Cursor auf die angegebene Position.

`\DDZylLineAt`$(r_1, \alpha_1, h_1)(r_2, \alpha_2, h_2)$ zeichnet eine Verbindungslinie.

`\DDZylLineTo`(r, α, h) verbindet weiter zum angegebenen Punkt.

`\DDZylArrowAt`$(r_1, \alpha_1, h_1)(r_2, \alpha_2, h_2)$ zeichnet einen verbindenden Pfeil.

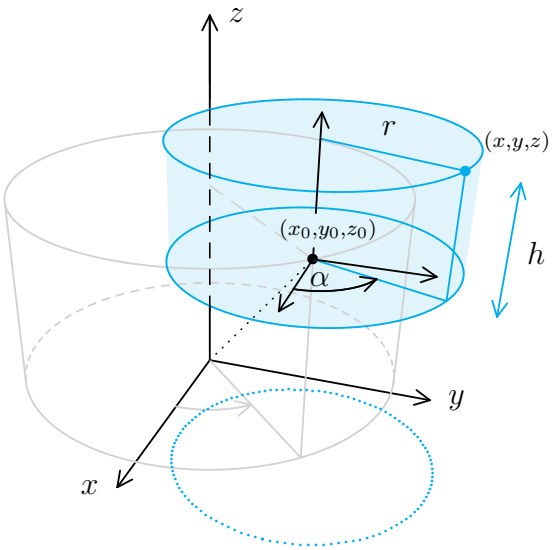
`\DDZylArrowHead`$(r_1, \alpha_1, h_1)(r_2, \alpha_2, h_2)$ zeichnet nur den Kopf des Pfeils.

Der Befehl `\DDZylRelLine`(r, α, h) ist etwas komplizierter. Der aktuelle Punkt (x_0, y_0, z_0) wird als neuer Nullpunkt eines Systems von Zylinderkoordinaten (mit der Achse $x = x_0$, $y = y_0$) angesehen, und von dort wird eine Linie zum Punkt

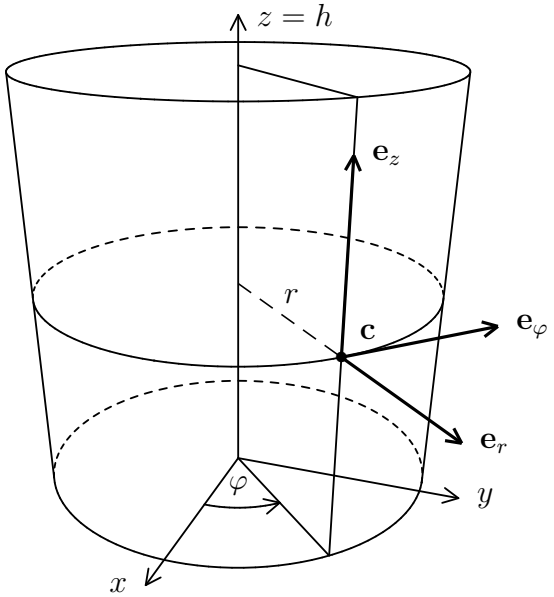
$$(x, y, z) = (x_0 + r\cos\alpha, y_0 + r\sin\alpha, z_0 + h)$$

gezogen.

`\DDZylRelArrow`(r, α, h) und `\DDZylRelArrowHead`(r, α, h) arbeiten analog, zeichnen aber einen Pfeil (bzw. den Kopf eines Pfeils). `\DDZylRelMoveTo`(r, α, h) bewegt nur den Cursor auf die neue Position. Man beachte, dass bei den relativen Befehlen immer der aktuelle Punkt neuer Nullpunkt ist.



Es folgt noch ein einfaches **Beispiel** für die Verwendung von Zylinderkoordinaten:




```
\InitGraph{12}{8}{4}{2}{1cm}
\Viewpoint(20,60,5,6)[2]
%
\SetDashed
\PArc(0,0,0,1)(95,305)
\DDMoveTo(0,0,1)
\DDZylRelLine(1,45,0)
\SetNormal
\DDMoveTo(0.4,0.4,1) \Text[t]{$r$}
\PArc(0,0,0,1)(305,360)
\PArc(0,0,0,1)(0,95)
```

```
\PCircle(0,0,2,1)
\SetNormal
\DDZylLineAt(1,95,0)(1,95,2)
\DDZylLineAt(1,305,0)(1,305,2)
%
\DDZylLineAt(0,45,0)(1,45,0)
\DDZylLineAt(1,45,0)(1,45,2)
\DDZylLineAt(0,45,2)(1,45,2)
\PArc(0,0,0,0.5)(0,45)
\DDZylArrowHead(0.7,0,0)(0.5,45,0)
\PArc(0,0,1,1)(305,360)
\PArc(0,0,1,1)(0,95)
\SetDashed
\PArc(0,0,1,1)(95,305)
\SetNormal
\SetThick{1.6pt}
\DDZylMoveTo(1,45,1)
\BigPoint \Text[rt]{$\V{c}$}
\SetNormal
%
\DDArrowAt(0,0,0)(0,0,2.2) \Text[r]{$z=h$}
\DDArrowAt(0,0,0)(1.2,0,0) \Text[l]{$x$}
\DDArrowAt(0,0,0)(0,1.2,0) \Text[r]{$y$}
%
\SetThick{1.2pt}
\DDZylMoveTo(1,45,1)% r,\alpha,h
\DDZylRelArrow(0.8,135,0)% phi,R
\Text[r]{$\V{e}_\phi$}
\DDZylMoveTo(1,45,1)
\DDRelArrow(0,0,0.8)
\Text[r]{$\V{e}_z$}
\DDZylMoveTo(1,45,1)
\DDZylRelArrow(0.8,45,0)
\Text[r]{$\V{e}_r$}
\SetNormal
\DDMoveTo(0,0,0) \Text[b]{\small $\phi$}
\CloseGraph
```

Kommen wir jetzt zu den **Kugelkoordinaten**:

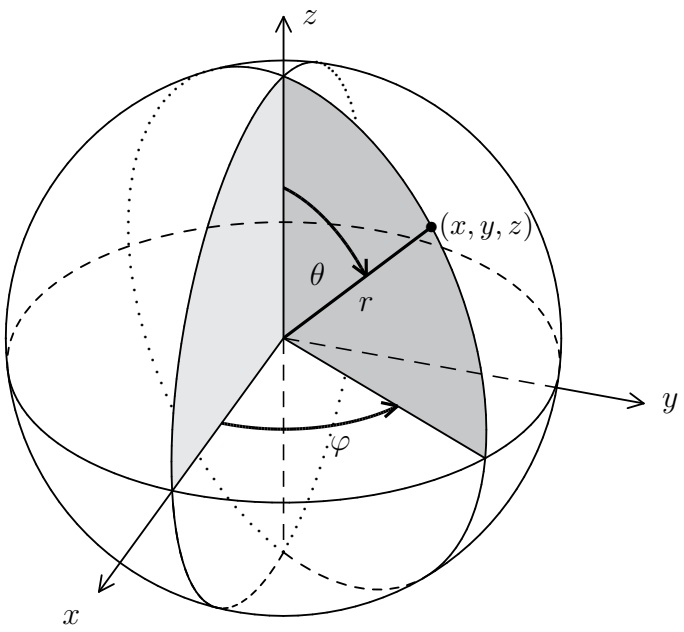
`\SphMoveTo`(r, φ, θ) bewegt den Cursor zu einem Punkt (x, y, z), der in spärischen Koordinaten (Kugelkoordinaten) gegeben sind. Dabei beschreiben die Parameter (r, φ, θ) den Radius (Abstand vom Nullpunkt), den Winkel φ gegen die positive x-Achse (zwischen 0 und 2π) und den Winkel θ gegen die positive z-Achse (zwischen 0 und π). Es besteht die Beziehung

$$\begin{aligned} x &= r\cos\varphi\sin\theta, \\ y &= r\sin\varphi\sin\theta, \\ z &= r\cos\theta. \end{aligned}$$

\SphLineAt$(r_1,\varphi_1,\theta_1)(r_2,\varphi_2,\theta_2)$ zeichnet eine Linie zwischen zwei Punkten, die in spärischen Koordinaten gegeben sind.

\SphLineTo$(r,\varphi,\theta)$ zeichnet eine Linie von der aktuellen Cursorposition zum Punkt mit den sphärischen Koordinaten (r,φ,θ).

\SphArrowTo$(r,\varphi,\theta)$ fügt eine Pfeilspitze hinzu.



Der Befehl \SphRelMoveTo$(r,\varphi,\theta)$ funktioniert genauso wie \SphMoveTo$(r,\varphi,\theta)$, aber die Kugelkoordinaten beziehen sich auf die aktuelle Cursorposition als Ursprung.

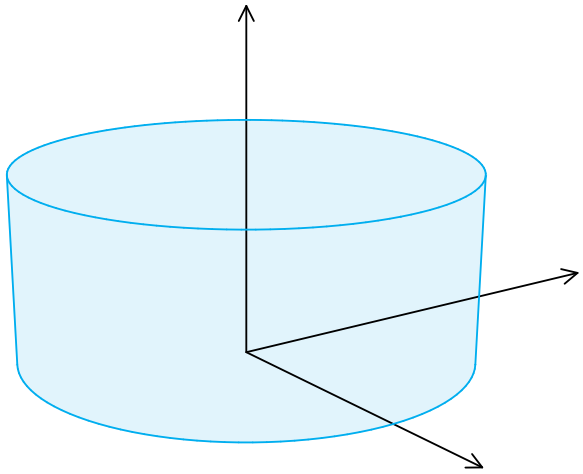
§ 6 Zylinder, Kegel, Kugeln

Für einen Zylinder mit vertikaler Achse durch $x = u$ und $y = v$, sowie Radius r liefert der Befehl `\Cylinder`$(u,v)(r)(h_u,h_o)$ den Umriss zwischen den Höhen h_u und h_o. Dabei werden nur die sichtbaren Linien gezeichnet.

Mit `\FullCylinder`$(u,v)(r)(h_u,h_o)$ wird der volle Zylinder mit einer vorher gewählten Farbe eingefärbt.

```
\InitGraph{12}{5}{9}{2}{1cm}
\Viewpoint(325,70,8,10)[2]
\SetCMYKColor(0.1,0,0,0)
\FullCylinder(0,0)(1.2)(0,1)
\ResetColor
\DDArrowAt(0,0,0)(1.8,0,0)
\DDArrowAt(0,0,0)(0,2.5,0)
\DDArrowAt(0,0,0)(0,0,1.8)
\SetCMYKColor(1,0,0,0)
\Cylinder(0,0)(1.2)(0,1)
\ResetColor
\CloseGraph
```

Dies ergibt folgendes Bild:

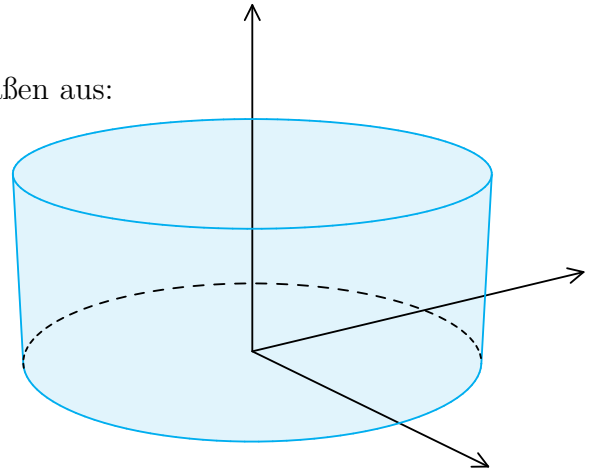


Will man die unsichtbaren Linien einzeichnen, so muss man dies mit Hilfe des Befehls `\PArc`(u,v,h,r) „zu Fuß“ erledigen:

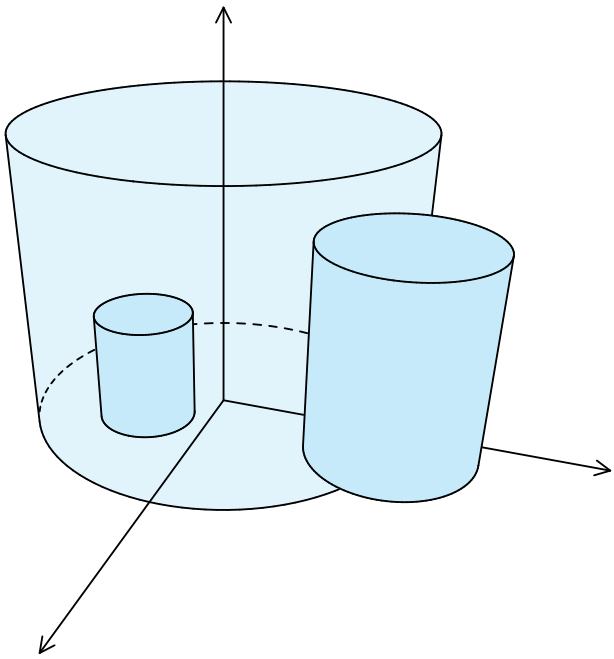
```
\InitGraph{12}{5}{9}{2}{1cm}
\Viewpoint(325,70,8,10)[2] %
\SetCMYKColor(0.1,0,0,0)
\FullCylinder(0,0)(1.2)(0,1)
\ResetColor
\DDArrowAt(0,0,0)(1.8,0,0)
\DDArrowAt(0,0,0)(0,2.5,0)
\DDArrowAt(0,0,0)(0,0,1.8)
\SetCMYKColor(1,0,0,0)
```

```
\Cylinder(0,0)(1.2)(0,1) %
\ResetColor

\SetDashed
\PArc(0,0,0,1.2)(50,250)
\SetNormal

\CloseGraph
```

Das Ergebnis sieht folgendermaßen aus:



Mit den Cylinder-Befehlen kann man auch Zylinder um beliebige Achsen zeichnen.



Die Befehle dazu lauten folgendermaßen:

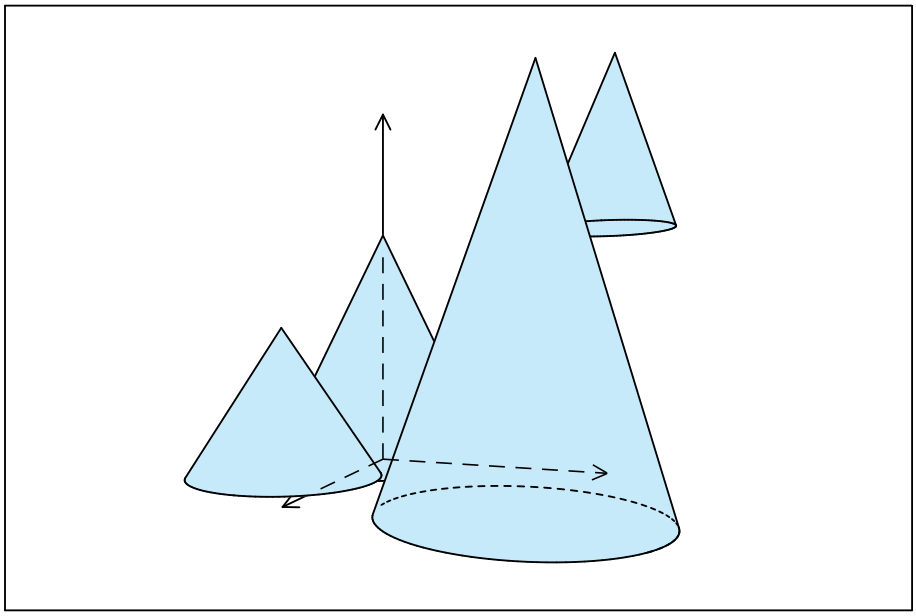
```
\InitGraph{12}{9}{6}{3.5}{1cm}
\Viewpoint(20,60,5,6)[2]
```

```
\SetCMYKColor(0.1,0,0,0)
\FullCylinder(0,0)(1)(0,1.5)
\ResetColor
\DDArrowAt(0,0,0)(2,0,0)
\DDArrowAt(0,0,0)(0,2,0)
\DDArrowAt(0,0,0)(0,0,2)
\Cylinder(0,0)(1)(0,1.5)
\SetDashed \PArc(0,0,0,1)(60,300) \SetNormal
\SetCMYKColor(0.2,0,0,0) \FullCylinder(0.5,1)(0.4)(0.2,1.2)
\ResetColor
\Cylinder(0.5,1)(0.4)(0.2,1.2)
\SetCMYKColor(0.2,0,0,0) \FullCylinder(-0.5,-0.7)(0.3)(-0.5,0.2)
\ResetColor
\Cylinder(-0.5,-0.7)(0.3)(-0.5,0.2)
\CloseGraph
```

Der Befehl \Cone$(u, v)(r)(h_u, h_o)$ liefert den Umriss eines Kreis-Kegels mit vertikaler Achse durch $(u, v, 0)$ und Radius r, dessen Grundfläche die Höhe h_u und dessen Spitze die Höhe h_o hat. Dabei werden nur die sichtbaren Linien gezeichnet.

Mit \FullCone$(u, v)(r)(h_u, h_o)$ wird der volle Kegel mit einer vorher gewählten Farbe eingefärbt.

Hier ist ein Beispiel dafür:



```
\InitGraph{12}{8}{5}{2}{1cm}
\Viewpoint(20,80,7,5)[2]
\DDArrowAt(0,0,0)(2,0,0)
\DDArrowAt(0,0,0)(0,2,0)
\DDArrowAt(0,0,0)(0,0,3)
```

```
\SetCMYKColor(0.2,0,0,0) \FullCone(0,0)(1)(0,2) \ResetColor
\Cone(0,0)(1)(0,2)
\SetCMYKColor(0.2,0,0,0) \FullCone(1.5,2)(0.4)(1.8,2.8) \ResetColor
\Cone(1.5,2)(0.4)(1.8,2.8)
\SetCMYKColor(0.2,0,0,0) \FullCone(1,-0.5)(0.8)(0,1.2) \ResetColor
\Cone(1,-0.5)(0.8)(0,1.2)
\SetCMYKColor(0.2,0,0,0) \FullCone(2.5,1.7)(0.8)(0.2,2.5) \ResetColor
\Cone(2.5,1.7)(0.8)(0.2,2.5)
\SetDashed \PArc(2.5,1.7,0.2,0.8)(55,270)
\DDLineAt(0,0,0)(1.5,0,0)
\DDArrowAt(0,0,0)(0,2,0)
\DDLineAt(0,0,0)(0,0,1.9)
\SetNormal
\CloseGraph
```

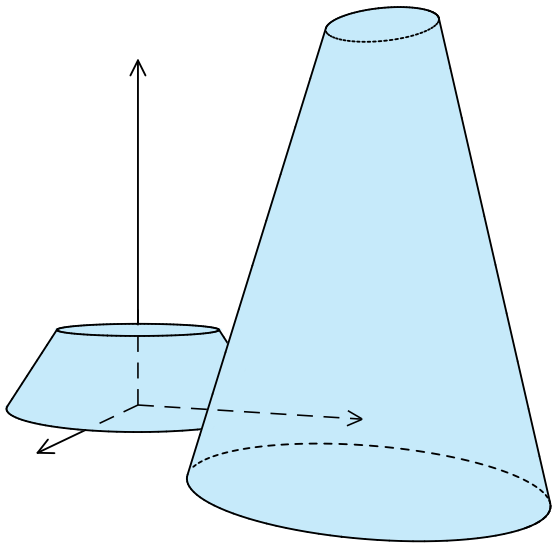
Ein Kegelstumpf mit Höhe H, dessen (fiktive) Spitze die Höhe h_o hat, wird mit `\TruncatedCone`$(u,v)(R)(h_u,h_o,H)$ gezeichnet, der volle eingefärbte Kegelstumpf mit `\FullTruncatedCone`$(u,v)(R)(h_u,h_o,H)$.

Will man hier den unsichtbaren Teil des „Deckels“ einzeichnen, so braucht man den Radius r dieses Deckels. Den kann man nach der Formel

$$r = \frac{h_o - H}{h_o - h_u} \cdot R$$

berechnen.

Er steht aber auch – unmittelbar nach Aufruf des Befehls `\TruncatedCone`(...) oder `\FullTruncatedCone`(...) – in `\kleinerradius` zur Verfügung.



Hier ist die Befehlsfolge zu dieser Zeichnung:

```
\InitGraph{12}{7.5}{5}{2}{1cm}
\Viewpoint(20,80,7,5)[2]
\DDArrowAt(0,0,0)(2,0,0)
\DDArrowAt(0,0,0)(0,2,0)
\DDArrowAt(0,0,0)(0,0,3)
\SetCMYKColor(0.2,0,0,0)
\FullTruncatedCone(0,0)(1.2)(0,1.8,0.7)
\ResetColor
\TruncatedCone(0,0)(1.2)(0,1.8,0.7)
\SetCMYKColor(0.2,0,0,0)
\FullTruncatedCone(2,2.1)(1)(0,3.5,2.5)
\ResetColor
\TruncatedCone(2,2.1)(1)(0,3.5,2.5)
\SetDashed
\PArc(2,2.1,2.5,\kleinerradius)(55,270)
\PArc(2,2.1,0,1)(55,270)
\DDLineAt(0,0,0)(1,0,0)
\DDArrowAt(0,0,0)(0,2,0)
\DDLineAt(0,0,0)(0,0,0.9)
\SetNormal
\CloseGraph
```

Es folgen nun einige Befehle zu **Geraden** und **Ebenen**.

`\TruncatedLine`$(x_1, y_1, z_1)(x_2, y_2, z_2)(a, b)$ zeichnet eine parametrisierte Strecke

$$t \mapsto (x_1, y_1, z_1) + t \cdot \big((x_2, y_2, z_2) - (x_1, y_1, z_1)\big),\ a \le t \le b.$$

Das ermöglicht es, einen Bruchteil einer Strecke zwischen zwei Punkten zu zeichnen oder eine solche Strecke über den Endpunkt hinaus zu verlängern.

Dieser Befehl benutzt die Standardroutine zum Zeichnen von Kurven und beeinflusst daher nicht die Position des Graphik-Cursors. Will man den auf einen bestimmten Punkt der oben beschriebenen Geraden setzen, so geht das mit dem Befehl `\TruncatedMove`$(x_1, y_1, z_1)(x_2, y_2, z_2)(t_0)$, der den Cursur auf den Punkt

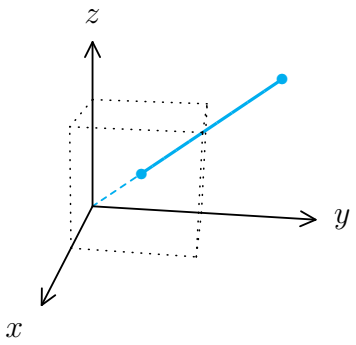
$$\mathbf{x}_0 := (x_1, y_1, z_1) + t_0 \cdot \big((x_2, y_2, z_2) - (x_1, y_1, z_1)\big)$$

setzt. Hier ist ein Anwendungsbeispiel:

```
\InitGraph{12}{4.5}{6}{1.5}{1cm}
\Viewpoint(10,70,7,5)[2]
\SetCMYKColor(1,0,0,0) \SetDashed
\TruncatedLine(0,0,0)(1,1,1)(0,0.5)
\TruncatedMove(0,0,0)(1,1,1)(0.5) \BigPoint
\SetThick{1.2pt} \TruncatedLine(0,0,0)(1,1,1)(0.5,1.5)
```

```
\TruncatedMove(0,0,0)(1,1,1)(1.5) \BigPoint
\SetNormal \ResetColor
\DDArrowAt(0,0,0)(2,0,0) \Text[bl]{$x$}
\DDArrowAt(0,0,0)(0,2,0) \Text[r]{$y$}
\DDArrowAt(0,0,0)(0,0,2) \Text[t]{$z$}
\SetDotted
\DDLineAt(1,0,0)(1,1,0) \DDLineTo(0,1,0) \DDLineAt(0,0,1)(1,0,1)
\DDLineTo(1,1,1) \DDLineTo(0,1,1) \DDLineTo(0,0,1)
\DDLineAt(1,0,0)(1,0,1) \DDLineAt(1,1,0)(1,1,1)
\DDLineAt(0,1,0)(0,1,1)
\CloseGraph
```

Damit wird folgendes Bild erzeugt:

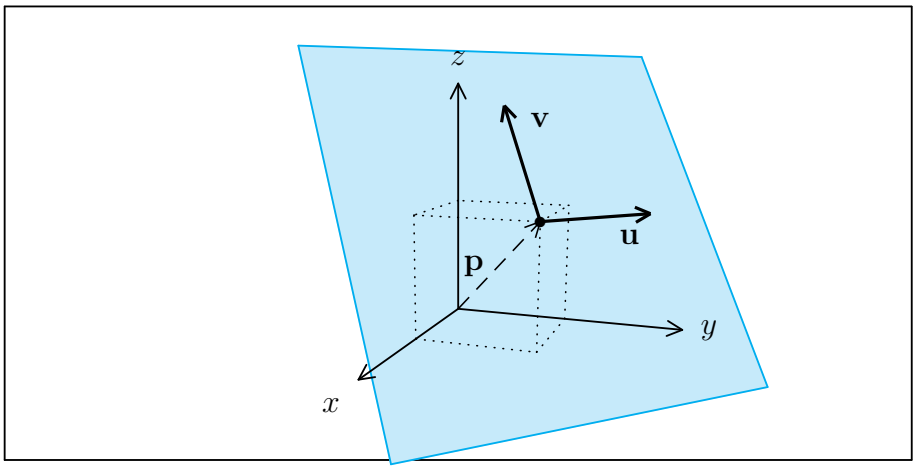


Eine Ebene kann dadurch festgelegt werden, dass man einen Punkt $\mathbf{p} = (x_0, y_0, z_0)$ angibt, durch den die Ebene geht, sowie einen Vektor $\mathbf{n} = (n_1, n_2, n_3)$, auf dem die Ebene senkrecht steht. Der Befehl

$$\texttt{\textbackslash SetParametricPlaneThrough}(x_0, y_0, z_0)\texttt{NormalTo}(n_1, n_2, n_3)$$

bestimmt eine Orthonormalbasis $\{\mathbf{u}, \mathbf{v}\}$, so dass die Ebene folgendermaßen parametrisiert wird:

$$(s, t) \mapsto \mathbf{p} + s\mathbf{u} + t\mathbf{v}, \text{ mit } \mathbf{u} \bullet \mathbf{v} = 0 \text{ und } \|\mathbf{u}\| = \|\mathbf{v}\| = 1.$$



Intern passiert Folgendes:

Zunächst wird der Einheitsnormalenvektor

$$\mathbf{N} = (u, v, w) := \frac{1}{\sqrt{n_1^2 + n_2^2 + n_3^2}}(n_1, n_2, n_3)$$

bestimmt. Die Basisvektoren gewinnt man dann so:

1. Ist $\mathbf{N} = (0, 0, 1)$, so wird $\mathbf{u} := (1, 0, 0)$ und $\mathbf{v} := (0, 1, 0)$ gesetzt.
2. Ist $(u, v) \neq (0, 0)$, so wird $\mathbf{u} := (\widetilde{v}, -\widetilde{u}, 0) = \frac{1}{\sqrt{u^2 + v^2}}(v, -u, 0)$ gesetzt, und

$$\mathbf{v} := \mathbf{N} \times \mathbf{u} = \big(w\widetilde{u}, w\widetilde{v}, -u\widetilde{u} - v\widetilde{v}\big).$$

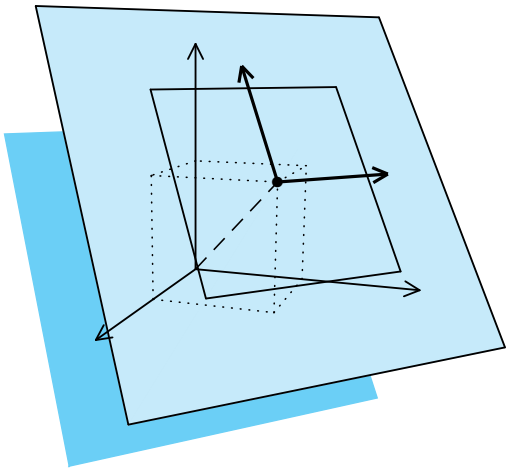
Nach Ausführung des Befehls findet man die Daten in

$\mathbf{N}$ =`(\normx,\normy,\normz)`, $\mathbf{u}$ =`(\sux,\suy,\suz)` und $\mathbf{v}$ =`(\svx,\svy,\svz)`.

Mit `\ShowPlaneThrough`(x_0, y_0, z_0)`NormalTo`$(n_1, n_2, n_3)(k)$ wird in der angegebenen Ebene das Quadrat mit den Eckpunkten (k, k), $(-k, k)$, $(-k, -k)$ und $(k, -k)$ (bezogen auf die Basis $\{\mathbf{u}, \mathbf{v}\}$) gezeichnet, um die Ebene sichtbar zu machen.

Mit `\ShowFullPlaneThrough`(x_0, y_0, z_0)`NormalTo`$(n_1, n_2, n_3)(k)$ wird das Quadrat als farbig gefüllte Fläche gezeigt.

Mit `\ShowBaseIn`(x_0, y_0, z_0)`NormalTo`(n_1, n_2, n_3) wird die (wie oben konstruierte) Orthonormalbasis $\{-\mathbf{u}, -\mathbf{v}\}$ gezeichnet. Man beachte, dass die Basis selbst bei fester Orientierung der Ebene nicht eindeutig bestimmt ist.



Das gezeigte Bild wird folgendermaßen erzeugt:

```
\InitGraph{12}{5.5}{6}{2}{1cm}
\Viewpoint(20,75,7,5)[2]
\SetCMYKColor(0.5,0,0,0)
\ShowFullPlaneThrough(0,0,0)NormalTo(1,1,1)(1.5)
```

```
\SetCMYKColor(0.2,0,0,0)
\ShowFullPlaneThrough(1,1,1)NormalTo(1,1,1)(1.5)
\ResetColor
\DDArrowAt(0,0,0)(2,0,0)
\DDArrowAt(0,0,0)(0,2,0)
\DDArrowAt(0,0,0)(0,0,2)
\SetDotted
\DDLineAt(1,0,0)(1,1,0) \DDLineTo(0,1,0)
\DDLineAt(0,0,1)(1,0,1) \DDLineTo(1,1,1) \DDLineTo(0,1,1) \DDLineTo(0,0,1)
\DDLineAt(1,0,0)(1,0,1)
\DDLineAt(1,1,0)(1,1,1)
\DDLineAt(0,1,0)(0,1,1)
\SetDashed
\DDLineAt(0,0,0)(1,1,1) \BigPoint
\SetNormal
\ShowPlaneThrough(1,1,1)NormalTo(1,1,1)(1.5)
\ShowPlaneThrough(1,1,1)NormalTo(1,1,1)(0.8)
\SetThick{1.2pt}
\ShowBaseIn(1,1,1)NormalTo(1,1,1)
\SetNormal
\CloseGraph
```

Einen Kreis in der Ebene, die durch

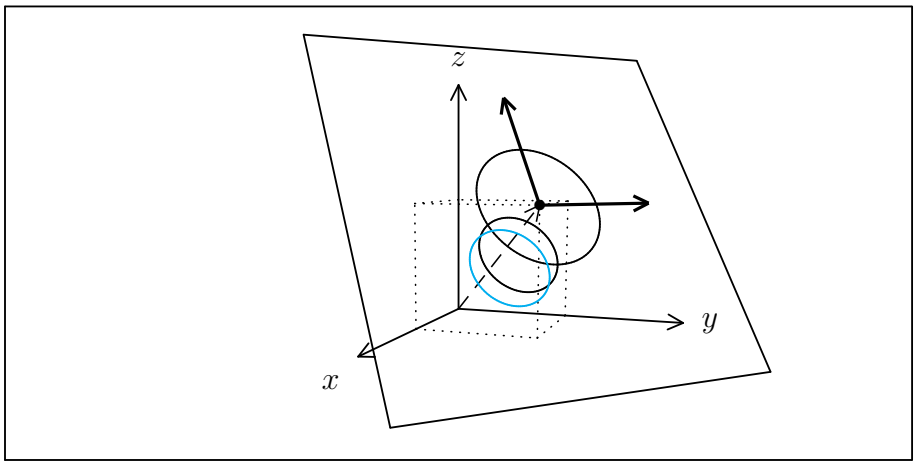
$$\texttt{\textbackslash SetParametricPlaneThrough}(x_0, y_0, z_0)\texttt{NormalTo}(n_1, n_2, n_3)$$

festgelegt wird, erhält man direkt mit dem Befehl

$$\texttt{\textbackslash TranslatedPlaneCircleAt}(u, v)(r)\texttt{Through}(x_0, y_0, z_0)\texttt{NormalTo}(n_1, n_2, n_3).$$

Bezüglich der Basis $\{\mathbf{u}, \mathbf{v}\}$, die wie oben konstruiert wird, erhält man den Kreis mit Mittelpunkt (u, v) und Radius r. Er wird parametrisiert durch

$$t \mapsto \mathbf{p} + (a + r\cos t)\mathbf{u} + (b + r\sin t)\mathbf{v}, \;\; 0 \le t \le 2\pi.$$



Mit \PlaneCircleAt$(u,v)(r)$NormalTo(n_1,n_2,n_3) erhält man den Kreis in der Ebene durch die aktuelle Cursorposition. Also sind die Befehlsfolgen

```
\TranslatedPlaneCircleAt(0.4,0.5)(0.3)Through(1,1,1)NormalTo(1,1,1)
```

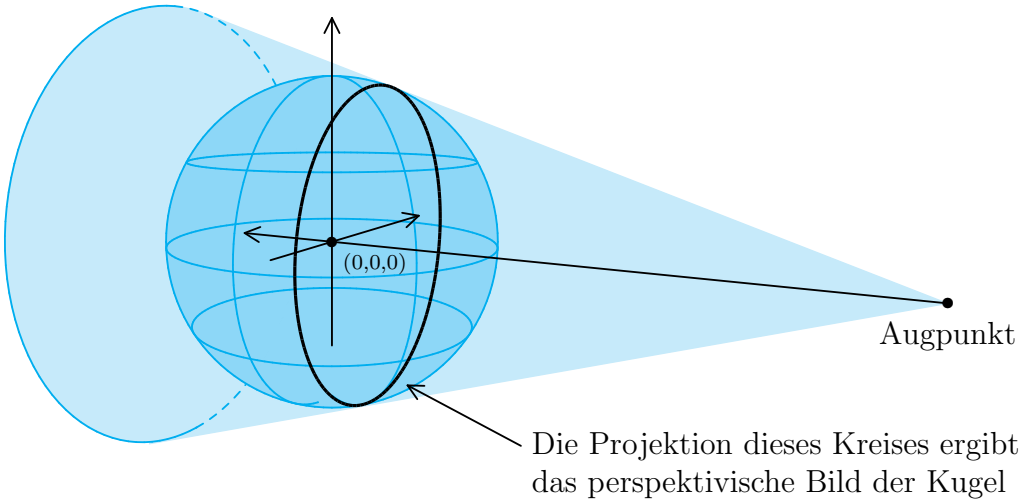
und

```
\DDMoveTo(1,1,1)
\PlaneCircleAt(0.4,0.5)(0.3)NormalTo(1,1,1)
```

gleich. Beides wurde beim obigen Bild benutzt:

```
\InitGraph{12}{6}{6}{2}{1cm}
\DrawBoundary
\Viewpoint(20,80,7,5)[2]
\DDArrowAt(0,0,0)(2,0,0) \Text[bl]{$x$}
\DDArrowAt(0,0,0)(0,2,0) \Text[r]{$y$}
\DDArrowAt(0,0,0)(0,0,2) \Text[t]{$z$}
\SetDotted
\DDLineAt(1,0,0)(1,1,0) \DDLineTo(0,1,0)
\DDLineAt(0,0,1)(1,0,1) \DDLineTo(1,1,1) \DDLineTo(0,1,1) \DDLineTo(0,0,1)
\DDLineAt(1,0,0)(1,0,1)
\DDLineAt(1,1,0)(1,1,1)
\DDLineAt(0,1,0)(0,1,1)
\SetDashed
\DDArrowAt(0,0,0)(1,1,1) \BigPoint
\SetThick{1.2pt}
\ShowBaseIn(1,1,1)NormalTo(1,1,1)
\SetNormal
\DDMoveTo(1,1,1)
\PlaneCircleAt(0,0)(0.5)NormalTo(1,1,1)
\DDMoveTo(1,1,1)
\PlaneCircleAt(0.3,0.4)(0.3)NormalTo(1,1,1)
\SetCMYKColor(1,0,0,0)
\TranslatedPlaneCircleAt(0.4,0.5)(0.3)Through(1,1,1)NormalTo(1,1,1)
\ResetColor
\ShowPlaneThrough(1,1,1)NormalTo(1,1,1)(1.5)
\CloseGraph
```

Die Darstellung von Kreisen auf beliebigen Ebenen ist wichtig, wenn man die perspektivische Darstellung einer Kugel sucht. Die vom Augpunkt ausgehenden Sehstrahlen, die eine Kugel berühren, treffen die Kugel entlang eines Kreises. Und dieser Kreis ist der Schnitt der Kugel mit der sogenannten Polarebene zum Augpunkt.

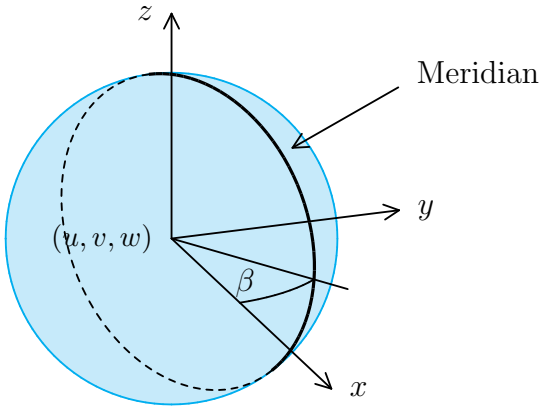


`\SphereAt`$(x_0, y_0, z_0)(R)$ zeichnet den Umriss einer Kugel vom Radius R um (x_0, y_0, z_0), `\FullSphereAt`$(x_0, y_0, z_0)(R)$ liefert die farbig gefüllte Kugel.

`\Meridian`(u, v, w, β, R) zeichnet einen „Meridian“, also einen Längenkreis der Kugel mit Radius R um (u, v, w). Der Winkel β ist der Winkel der Ebene, in der der Meridian liegt, gegen die positive x-Achse. Der Meridian wird parametrisiert durch

$$t \mapsto \mathbf{p} + (\cos t)\,\mathbf{u} + (\sin t)\,\mathbf{v},$$

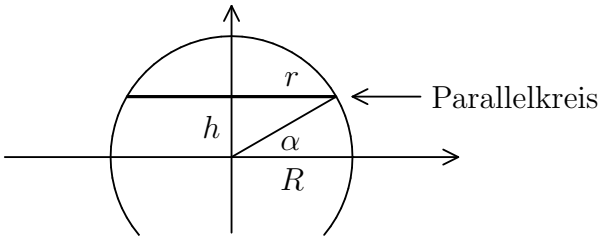
wobei $\mathbf{p} = (u, v, w)$ der Mittelpunkt der Kugel, $\mathbf{u} = (R\cos\beta, R\sin\beta, 0)$ und $\mathbf{v} = (0, 0, 1)$ ist. Mit `\MeridianArc`$(u, v, w, \beta, R)(\alpha_1, \alpha_2)$ kann man einen Ausschnitt des Meridians zeichnen, mit $\alpha_1 \le t \le \alpha_2$.



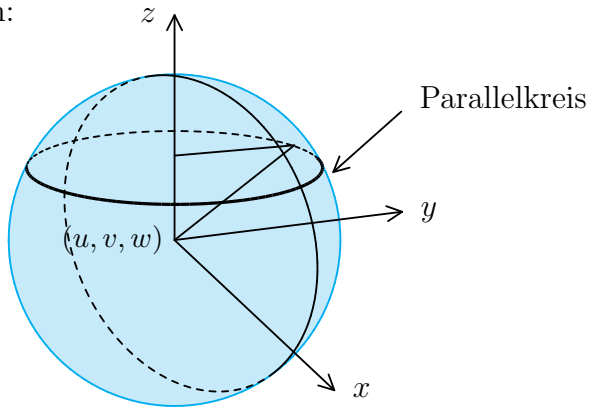
`\Parallel`(u, v, w, α, R) zeichnet einen Parallel- oder Breitenkreis der Kugel mit Radius R um (u, v, w), `\ParallelDisk`(u, v, w, α, R) zeichnet stattdessen eine gefüllte Scheibe, und mit `\ParallelArc`$(u, v, w, \alpha, R)(\alpha_1, \alpha_2)$ erhält man einen Ausschnitt des Kreises, parametrisiert durch

$$t \mapsto (u, v, w + h) + \cos t\,(r, 0, 0) + \sin t(0, r, 0),$$

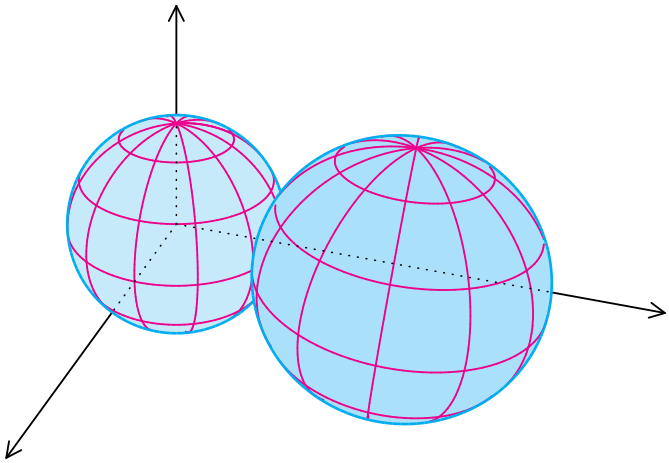
mit $r = R\cos\alpha$ und $h = R\sin\alpha$.



bzw. perspektivisch:



Hier ist ein Beispiel mit diesen Befehlen.

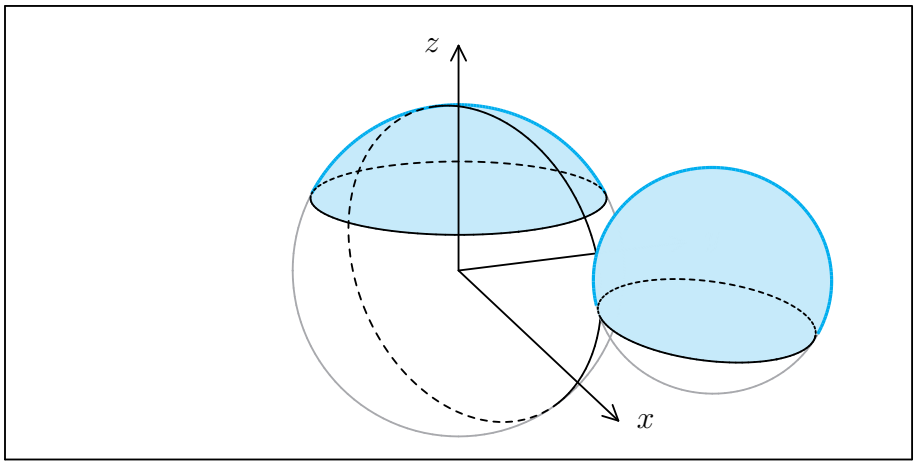


```
\InitGraph{12}{7}{4}{3.5}{1cm}
\Viewpoint(20,60,7,5)[2]
\DDArrowAt(0,0,0)(3,0,0) \DDArrowAt(0,0,0)(0,4,0) \DDArrowAt(0,0,0)(0,0,2)
%
% Hintere Kugel, um (0,0,0) mit Radius 1
%
\SetCMYKColor(0.2,0,0,0) \FullSphereAt(0,0,0)(1)
%
\SetCMYKColor(0,1,0,0)
```

```
\MeridianArc(0,0,0,300,1)(0,120)
\MeridianArc(0,0,0,330,1)(320,360) \MeridianArc(0,0,0,330,1)(0,120)
\MeridianArc(0,0,0,0,1)(310,360) \MeridianArc(0,0,0,0,1)(0,120)
\MeridianArc(0,0,0,30,1)(300,360) \MeridianArc(0,0,0,30,1)(0,120)
\MeridianArc(0,0,0,60,1)(320,360) \MeridianArc(0,0,0,60,1)(0,120)
\MeridianArc(0,0,0,90,1)(0,90)
\MeridianArc(0,0,0,120,1)(40,90)
%
\ParallelArc(0,0,0,330,1)(320,360) \ParallelArc(0,0,0,330,1)(0,80)
\ParallelArc(0,0,0,0,1)(300,360) \ParallelArc(0,0,0,0,1)(0,100)
\ParallelArc(0,0,0,30,1)(280,360) \ParallelArc(0,0,0,30,1)(0,120)
\ParallelArc(0,0,0,60,1)(270,360) \ParallelArc(0,0,0,60,1)(0,130)
%
\SetCMYKColor(1,0,0,0)
\SetThick{1pt} \SphereAt(0,0,0)(1) \SetNormal
\ResetColor

%
% Vordere Kugel, um (1,2,0.5) mit Radius 1
%
\SetCMYKColor(0.3,0,0,0) \FullSphereAt(1,2,0.5)(1)
%
\SetCMYKColor(0,1,0,0)
\MeridianArc(1,2,0.5,270,1)(30,150)
\MeridianArc(1,2,0.5,300,1)(330,360) \MeridianArc(1,2,0.5,300,1)(0,120)
\MeridianArc(1,2,0.5,330,1)(310,360) \MeridianArc(1,2,0.5,330,1)(0,120)
\MeridianArc(1,2,0.5,0,1)(300,360) \MeridianArc(1,2,0.5,0,1)(0,120)
\MeridianArc(1,2,0.5,30,1)(320,360) \MeridianArc(1,2,0.5,30,1)(0,120)
\MeridianArc(1,2,0.5,60,1)(340,360) \MeridianArc(1,2,0.5,60,1)(0,120)
%
\ParallelArc(1,2,0.5,330,1)(310,360) \ParallelArc(1,2,0.5,330,1)(0,60)
\ParallelArc(1,2,0.5,0,1)(280,360) \ParallelArc(1,2,0.5,0,1)(0,80)
\ParallelArc(1,2,0.5,30,1)(280,360) \ParallelArc(1,2,0.5,30,1)(0,85)
\ParallelArc(1,2,0.5,60,1)(250,360) \ParallelArc(1,2,0.5,60,1)(0,110)
%
\SetCMYKColor(1,0,0,0)
\SetThick{1pt} \SphereAt(1,2,0.5)(1) \SetNormal
\ResetColor
%
\SetDotted
\DDLineAt(0,0,0)(1.5,0,0) \DDLineAt(0,0,0)(0,3.2,0) \DDLineAt(0,0,0)(0,0,1)
\SetNormal
%
\CloseGraph
```

Mit dem Befehl \UpperSphereCapAt$(x_0, y_0, z_0)(r)(\alpha)$ erhält man eine (obere) Kugelkappe (in der Kugel mit Radius r um (x_0, y_0, z_0)). Abgeschnitten wird entlang des durch α gegebenen Breitenkreises (mit $-90 < \alpha < 90$). Der Befehl zeichnet nur die Umrissline der Kugelkappe auf dem Kugelumriss. Den Schnittkreis muss man gesondert zeichnen.

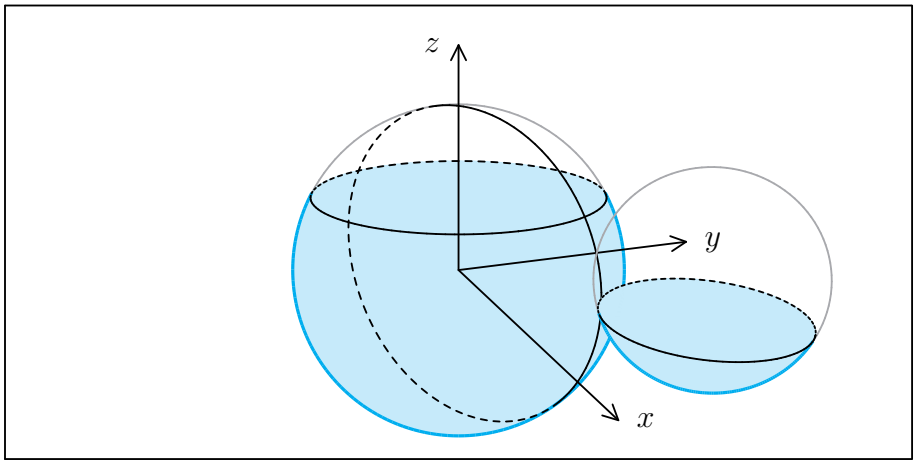
Mit \FullUpperSphereCapAt$(x_0, y_0, z_0)(r)(\alpha)$ wird die Kugelkappe als gefärbte Fläche erzeugt (inkl. Schnittfläche, falls diese sichtbar ist). Hier ist ein Beispiel:



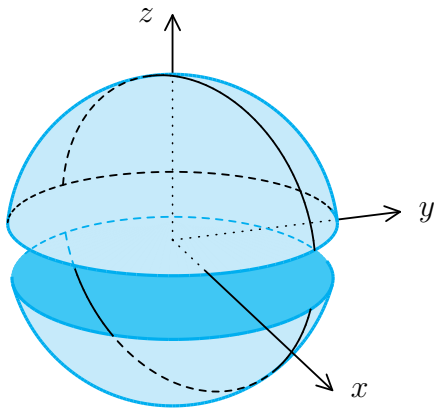

```
\InitGraph{12}{6}{6}{2.5}{1cm}
\Viewpoint(340,70,7,5)[2]
\SetCMYKColor(0,0,0,0.4) \SphereAt(0,0,0)(1.5)
\SetCMYKColor(0.2,0,0,0)
\FullUpperSphereCapAt(0,0,0)(1.5)(30)
\SetCMYKColor(1,0,0,0) \SetThick{1.2pt}
\UpperSphereCapAt(0,0,0)(1.5)(30)
\SetNormal \ResetColor
\DDArrowAt(0,0,0)(2.8,0,0) \Text[r]{$x$}
\DDArrowAt(0,0,0)(0,2.5,0) \Text[r]{$y$}
\DDArrowAt(0,0,0)(0,0,2) \Text[l]{$z$}
\MeridianArc(0,0,0,30,1.5)(0,100)
\MeridianArc(0,0,0,30,1.5)(320,360)
\ParallelArc(0,0,0,30,1.5)(0,70)
\ParallelArc(0,0,0,30,1.5)(260,360)
\SetDashed
\MeridianArc(0,0,0,30,1.5)(100,320)
\ParallelArc(0,0,0,30,1.5)(75,260)
\SetNormal
%
\SetCMYKColor(0,0,0,0.4) \SphereAt(1,2,0)(1)
\SetCMYKColor(0.2,0,0,0)
\FullUpperSphereCapAt(1,2,0)(1)(340)
```

```
\SetCMYKColor(1,0,0,0) \SetThick{1.2pt}
\UpperSphereCapAt(1,2,0)(1)(340)
\ResetColor
\SetDashed \ParallelArc(1,2,0,-20,1)(40,250) \SetNormal
\ParallelArc(1,2,0,-20,1)(250,360)
\ParallelArc(1,2,0,-20,1)(0,42)
%
\CloseGraph
```

Ersetzt man \UpperSphereCapAt$(x_0, y_0, z_0)(r)(\alpha)$ durch \LowerSphereCapAt$(x_0, y_0, z_0)(r)(\alpha)$ und \FullUpperSphereCapAt$(x_0, y_0, z_0)(r)(\alpha)$ durch \FullLowerSphereCapAt$(x_0, y_0, z_0)(r)(\alpha)$, so erhält man untere Kugelkappen. Lässt man im obigen Beispiel alles andere unverändert, so sieht das Bild folgendermaßen aus:



Es folgt noch ein etwas komplizierteres Beispiel:




```
\InitGraph{12}{6}{6}{2.5}{1cm}
\Viewpoint(340,70,7,5)[2]
\DDArrowAt(0,0,0)(0,2.5,0) \Text[r]{$y$}
```

```
\DDArrowAt(0,0,0)(0,0,2) \Text[l]{$z$}
\SetCMYKColor(0.2,0,0,0)
\FullLowerSphereCapAt(0,0,0)(1.5)(-10)
\SetCMYKColor(0.6,0,0,0) \ParallelDisk(0,0,0,-10,1.5)
\SetCMYKColor(1,0,0,0) \SetThick{1.2pt}
\LowerSphereCapAt(0,0,0)(1.5)(-10)
\ParallelArc(0,0,0,-10,1.5)(0,75)
\ParallelArc(0,0,0,-10,1.5)(260,360)
\SetNormal
\SetCMYKColor(0.2,0,0,0)
\FullUpperSphereCapAt(0,0,0)(1.5)(10)
\SetCMYKColor(1,0,0,0) \SetThick{1.2pt}
\UpperSphereCapAt(0,0,0)(1.5)(10)
\ParallelArc(0,0,0,10,1.5)(0,75)
\ParallelArc(0,0,0,10,1.5)(260,360)
\SetNormal \ResetColor
\DDArrowAt(0.8,0,0)(2.8,0,0) \Text[r]{$x$}
\SetDotted
\DDLineAt(0,0,0)(0,0,1.5)
\DDLineAt(0,0,0)(0,1.7,0)
\DDLineAt(0,0,0)(0.8,0,0)
\SetNormal
\MeridianArc(0,0,0,30,1.5)(10,100)
\MeridianArc(0,0,0,30,1.5)(320,350)
\MeridianArc(0,0,0,30,1.5)(205,237)
\SetDashed \SetCMYKColor(1,0,0,0)
\ParallelArc(0,0,0,-10,1.5)(90,240)
\MeridianArc(0,0,0,30,1.5)(190,205)
\ResetColor
\MeridianArc(0,0,0,30,1.5)(100,170)
\ParallelArc(0,0,0,10,1.5)(75,260)
\MeridianArc(0,0,0,30,1.5)(240,320)
\SetNormal
\CloseGraph
```

§ 7 Sphärische Geometrie

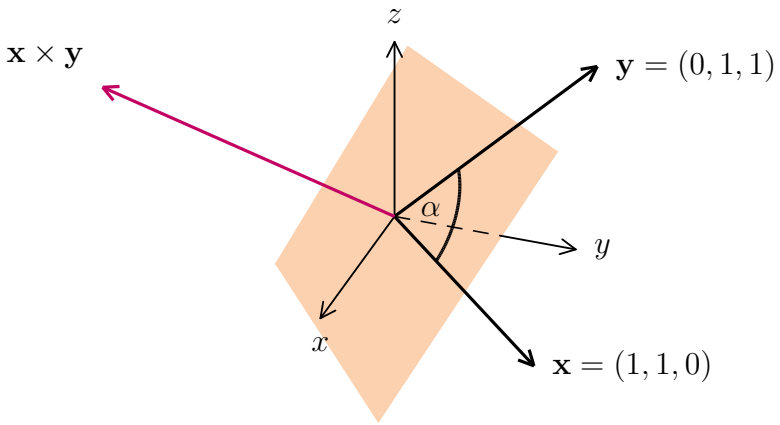
Es gibt ein paar Befehle, mit denen Großkreise und sphärische Dreiecke gezeichnet werden können.

Dazu werden einige grundlegende Befehle aus der 3-dimensionalen analytischen Geometrie bereitgestellt. Manche sind nur für den internen Gebrauch vorgesehen.

\VectorProductOf$(x_1,x_2,x_3)$And(y_1,y_2,y_3) berechnet intern das Kreuz- oder Vektor-Produkt

$$(x_1,x_2,x_3)\times(y_1,y_2,y_3) := (x_2y_3-x_3y_2,\ x_3y_1-x_1y_3,\ x_1y_2-x_2y_1).$$

Das Ergebnis wird im Vektor (\vecprode ,\vecprodz ,\vecprodd) gespeichert. Mit \DrawVectorProductOf$(x_1,x_2,x_3)$And(y_1,y_2,y_3) wird dieser Vektor gezeichnet.



Das Vektorprodukt $\mathbf{x}\times\mathbf{y}$ steht auf der von $\mathbf{x}$ und $\mathbf{y}$ aufgespannten Ebene senkrecht. Außerdem ist

$$\|\mathbf{x}\times\mathbf{y}\| = \|\mathbf{x}\|\cdot\|\mathbf{y}\|\cdot\sin\alpha,$$

wobei α der Winkel zwischen $\mathbf{x}$ und $\mathbf{y}$ ist, und $\|\mathbf{x}\|$ die Länge des Vektors $\mathbf{x}$, die übrigens durch \Lengthof$(x_1,x_2,x_3)$To(λ) berechnet wird.

In der Befehlsfolge für die obige Bezeichnung wird noch der bisher nicht beschriebene Befehl \PlaneCircularArcAt$(u,v)(r)$NormalTo$(n_1,n_2,n_3)(t_1,t_2)$ benutzt, der wie der entsprechende Kreisbefehl funktioniert, aber nur einen Kreisbogen zwischen den Parametern t_1 und t_2 (im Bogenmaß) zeichnet. Die Stelle $t=0$ muss man selbst herausfinden.

```
\InitGraph{12}{5.5}{6}{2.5}{1cm}
\Viewpoint(20,60,5,6)[2] %
\SetCMYKColor(0,0.2,0.3,0)
\VectorProductOf(1,1,0)And(0,1,1)
\ShowFullPlaneThrough(0,0,0)NormalTo(\vecprode,\vecprodz,\vecprodd)(0.8)
\ResetColor
\DDArrowAt(0,0,0)(1,0,0) \Text[b]{$x$}
```

```
\DDArrowAt(0,0.6,0)(0,1,0) \Text[r]{$y$}
\DDArrowAt(0,0,0)(0,0,1) \Text[t]{$z$}
\SetDashed \DDLineAt(0,0,0)(0,0.6,0)
\SetThick{1.2pt}
\DDArrowAt(0,0,0)(1,1,0) \Text[r]{$\V{x}=(1,1,0)$}
\DDArrowAt(0,0,0)(0,1,1) \Text[r]{$\V{y}=(0,1,1)$}
\SetCMYKColor(0,1,0.2,0.2) \DrawVectorProductOf(1,1,0)And(0,1,1)
\ResetColor
\Text[tl]{$\V{x}\times\V{y}$}
\DDMoveTo(0,0,0)
\PlaneCircularArcAt(0,0)(0.5)NormalTo
      (\vecprode,\vecprodz,\vecprodd)(\EinPi,4.2)
\SetNormal
\MoveTo(0.1,0.1) \Text[r]{$\alpha$}
\CloseGraph
```

Durch \ScalarProductOf$(x_1, x_2, x_3)$\And$(y_1, y_2, y_3)$ wird das Skalarprodukt

$$(x_1, x_2, x_3) \bullet (y_1, y_2, y_3) := x_1y_1 + x_2y_2 + x_3y_3$$

berechnet. Das Ergebnis steht dann in der Variablen \scalarproductotv. Es ist

$$\mathbf{x} \bullet \mathbf{y} = \|\mathbf{x}\| \cdot \|\mathbf{y}\| \cdot \cos\alpha \quad \text{und} \quad \|\mathbf{x} \times \mathbf{y}\| = \sqrt{\|\mathbf{x}\|^2 \cdot \|\mathbf{y}\|^2 - (\mathbf{x} \bullet \mathbf{y})^2}.$$

Mit \SetSphereAt$(x_0, y_0, z_0)(R)$ werden die Daten einer Kugel abgespeichert, der Mittelpunkt (x_0, y_0, z_0) in den Variablen (\sphcenterx, \sphcentery, \sphcenterz) und der Radius R in der Variablen \sphradius. Alle nachfolgenden Befehle, die mit „Großkreisen“ zu tun haben, beziehen sich auf die gewählte Kugel. Mit \ShowSphere wird der Umriss dieser Kugel gezeichnet.

Ein Großkreis auf einer Kugel ist der Schnitt der Kugel mit einer Ebene durch den Mittelpunkt der Kugel. Sind zwei Punkte auf der Kugel gegeben, die nicht gerade Antipodenpunkte sind, so geht genau ein Großkreis durch sie, und die Länge des Kreisabschnittes durch die beiden Punkte ist deren Entfernung auf der Kugel. Großkreise sind also „Geodätische“, Kurven kürzester Länge.

Hier werden die Punkte auf der Kugel in Kugelkoordinaten gegeben, die den Mittelpunkt $\mathbf{x}_0$ der Kugel als Ursprung benutzen. Der Radius R muss zuvor durch Festlegung der Kugel bestimmt werden. Mit \GreatCircleAt$(\varphi_1, \theta_1, \varphi_2, \theta_2)$ wird der Großkreis durch die Punkte $\mathbf{x}_1$ und $\mathbf{x}_2$ mit den Kugelkoordinaten (R, φ_1, θ_1) und (R, φ_2, θ_2) gezeichnet. Das funktioniert intern folgendermaßen:

1. Zunächst werden die kartesischen Koordinaten von $\mathbf{x}_1$ und $\mathbf{x}_2$ berechnet (eigentlich die Koordinaten der Vektoren $\mathbf{x}_1 - \mathbf{x}_0$ und $\mathbf{x}_2 - \mathbf{x}_0$, wir tun hier so, als sei $\mathbf{x}_0 = \mathbf{0}$). Der Winkel α zwischen den beiden Punkten ist durch $\alpha = \arccos(\mathbf{x}_1 \bullet \mathbf{x}_2)$ gegeben.

2. Sei $\mathbf{N} := \mathbf{x}_1 \times \mathbf{x}_2$ und $\mathbf{N}_0 := \mathbf{N}/\|\mathbf{N}\|$. Dann ist die von $\mathbf{x}_1$ und $\mathbf{x}_2$ aufgespannten Ebene E die Ebene durch den Mittelpunkt der Kugel, die auf $\mathbf{N}$ senkrecht steht, und $\mathbf{x}_3 := \mathbf{N} \times \mathbf{x}_1$ liegt in E.

3. $\{\mathbf{x}_1, \mathbf{x}_3\}$ ist eine „Orthonormalbasis“ der Ebene E, d.h. es ist

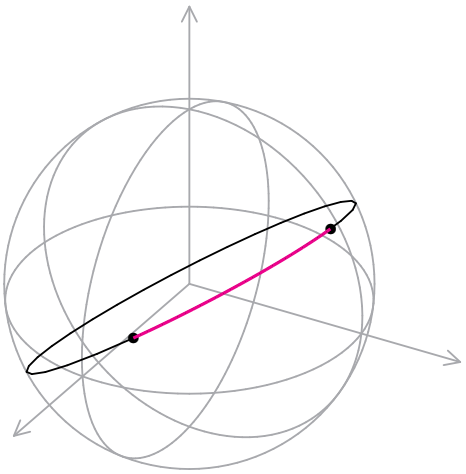
$$\|\mathbf{x}_1\| = \|\mathbf{x}_3\| = 1 \quad \text{und} \quad \mathbf{x}_1 \bullet \mathbf{x}_3 = 0.$$

   Insbesondere ist $\mathbf{x}_1 \times \mathbf{x}_3 = \mathbf{N}$.

4. Der Großkreis durch $\mathbf{x}_1$ und $\mathbf{x}_2$ ist gegeben durch

$$t \mapsto \mathbf{x}_0 + (\cos t)\mathbf{x}_1 + (\sin t)\mathbf{x}_3, \quad 0 \le t \le 2\pi.$$

   Bei $t = 0$ erhält man $\mathbf{x}_1$, bei $t = \alpha$ den Punkt $\mathbf{x}_2$ und bei $t = \pi/2$ den Punkt $\mathbf{x}_3$.



Mit \GreatCircleAt$(\varphi_1, \theta_1, \varphi_2, \theta_2)$ wird nur das Stück des Großkreises gezeichnet, das sich zwischen den beiden Punkten befindet. Mit

$$\backslash\texttt{GreatCircularArcAt}(\varphi_1, \theta_1, \varphi_2, \theta_2)(t_1, t_2)$$

kann man den Großkreis zwischen beliebigen Parametern t_1 und t_2 zeichnen. Mit $t_1 = 0$ und $t_2 = 2\pi$ erhält man den gesamten Großkreis. Die obige Zeichnung entsteht folgendermaßen:

```
\InitGraph{12}{6.5}{6}{3}{1cm}
\Viewpoint(30,60,5,6)[2]
\SetCMYKColor(0,0,0,0.4)
\DDArrowAt(0,0,0)(1.5,0,0)
\DDArrowAt(0,0,0)(0,1.5,0)
\DDArrowAt(0,0,0)(0,0,1.5)
\StandardSphere(1)
```

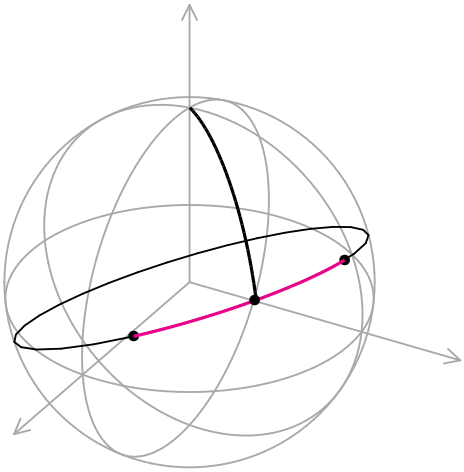
```
\Meridian(0,0,0,0,1) \Meridian(0,0,0,90,1) \Parallel(0,0,0,0,1)
\ResetColor
%
\SetSphereAt(0,0,0)(1)
\SphMoveTo(1,15,75) \BigPoint
\SphMoveTo(1,85,55) \BigPoint
\GreatCircularArcAt(15,75,85,55)(0,\ZweiPi)
\SetThick{1.2pt} \SetCMYKColor(0,1,0,0)
\GreatCircleAt(15,75,85,55)
\ResetColor \SetNormal
\CloseGraph
```

\GreatCircleTo$(\varphi, \theta)$ zeichnet einen Großkreis-Bogen von der letzten Position zu dem Punkt mit den angegebenen Koordinaten.

Der Befehl \SphPOLvon$< x, y, z >$ nach $< r, \varphi, \theta >$ berechnet aus den kartesischen Koordinaten eines Punktes auf der voreingestellten Kugel dessen Kugelkoordinaten. Dabei ist zu beachten, dass die Winkel hier im Bogenmaß ausgegeben werden (während die Kugelkoordinaten bei uns sonst im Gradmaß einzugeben sind).

Mit \FindSphericalCenter$(\varphi_1, \theta_1, \varphi_2, \theta_2)$ wird die Mitte auf dem Großkreis zwischen den durch (φ_1, θ_1) und (φ_2, θ_2) gegebenen Punkten bestimmt. Der Cursor wird auf diesen Punkt bewegt. Seine kartesischen Koordinaten stehen danach in (\xmitt ,\ymitt ,\zmitt), die Kugelkoordinaten in (\rmitt ,\phimitt ,\Thetamitt), letztere auch im Gradmaß!

Beispiel:

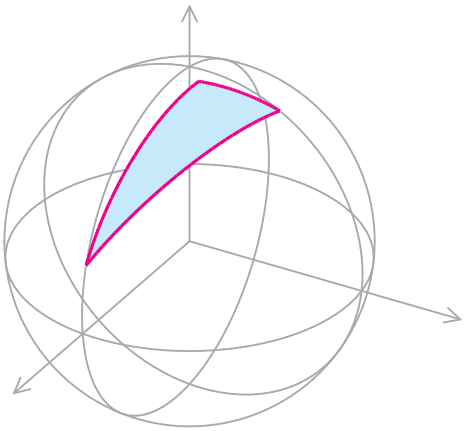


Da die Winkel φ und θ in Gestalt der Variablen \phimitt und \Thetamitt von \FindSphericalCenter im Bogenmaß ausgegeben werden, müssen sie anschließend eventuell ins Gradmaß umgewandelt werden. Das geht mit der internen Routine \DEGREEvon$< B >$ nach $< \beta >$. Dabei muss B eine dimensionierte Variable

sein, die den Winkel im Bogenmaß enthält, und β ein Zähler (Ganzzahl-Register). Für Letzteres bieten sich die Zähler `\Phigrad` und `\Thetagrad` an. Will man den Inhalt eines Zählers weiterverarbeiten, so muss man ihn mit `\the\...` aufrufen. Das erklärt das folgende Listing.

```
\InitGraph{12}{7}{6}{3}{1cm}
\Viewpoint(30,60,5,6)[2]
\SetCMYKColor(0,0,0,0.4)
\DDArrowAt(0,0,0)(1.5,0,0)
\DDArrowAt(0,0,0)(0,1.5,0)
\DDArrowAt(0,0,0)(0,0,1.5)
\StandardSphere(1)
\Meridian(0,0,0,0,1) \Meridian(0,0,0,90,1) \Parallel(0,0,0,0,1)
\ResetColor
%
\SetSphereAt(0,0,0)(1)
\SphMoveTo(1,15,75) \BigPoint
\SphMoveTo(1,85,65) \BigPoint
\GreatCircularArcAt(15,75,85,65)(0,\ZweiPi)
\SetThick{1.2pt} \SetCMYKColor(0,1,0,0)
\GreatCircleAt(15,75,85,65) \ResetColor
\FindSphericalCenter(15,75,85,65) \BigPoint
\DEGREEvon<\phimitt\ONEPOINT>nach<\Phigrad>
\DEGREEvon<\Thetamitt\ONEPOINT>nach<\Thetagrad>
\GreatCircleAt(\the\Phigrad,\the\Thetagrad,0,0)
\SetNormal
\CloseGraph
```

`\SphericalTriangle`$(\varphi_1,\theta_1)(\varphi_2,\theta_2)(\varphi_3,\theta_3)$ zeichnet ein Dreieck auf der Kugel, dessen Seiten von Großkreisen gebildet werden.
`\FullSphericalTriangle`$(\varphi_1,\theta_1)(\varphi_2,\theta_2)(\varphi_3,\theta_3)$ füllt das sphärische Dreieck in der jeweiligen voreingestellten Farbe aus.

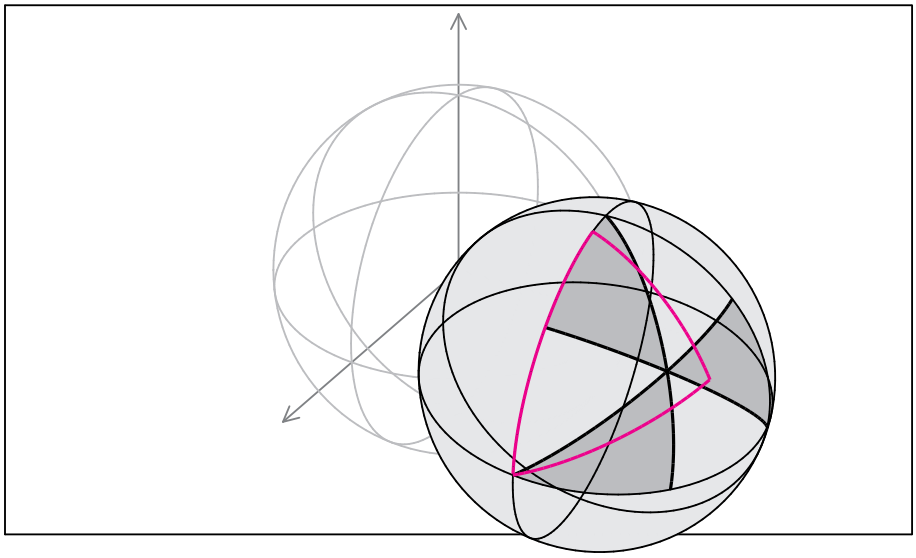


Die Füllroutine kann natürlich nur funktionieren, wenn das gesamte Dreieck sichtbar ist. Darum muss sich der Benutzer selbst kümmern. Und es kann auch Extremsituationen geben, in denen sogar ein komplett sichtbares Dreieck nicht ordnungsgemäß gefüllt wird.

Hier kommt das Listing:

```
\InitGraph{12}{6}{6}{3}{1cm}
\Viewpoint(30,60,5,6)[2]
\SetCMYKColor(0,0,0,0.4)
\DDArrowAt(0,0,0)(1.5,0,0)
\DDArrowAt(0,0,0)(0,1.5,0)
\DDArrowAt(0,0,0)(0,0,1.3)
\StandardSphere(1)
\Meridian(0,0,0,0,1) \Meridian(0,0,0,90,1) \Parallel(0,0,0,0,1)
\ResetColor
%
\SetSphereAt(0,0,0)(1)
\SetCMYKColor(0.2,0,0,0)
\FullSphericalTriangle(0,70)(90,30)(45,10)
\SetThick{1.2pt}
\SetCMYKColor(0,1,0,0)
\SphericalTriangle(0,70)(90,30)(45,10)
\ResetColor
\SetNormal
%
\CloseGraph
```

Und hier ist noch ein etwas komplexeres Beispiel:



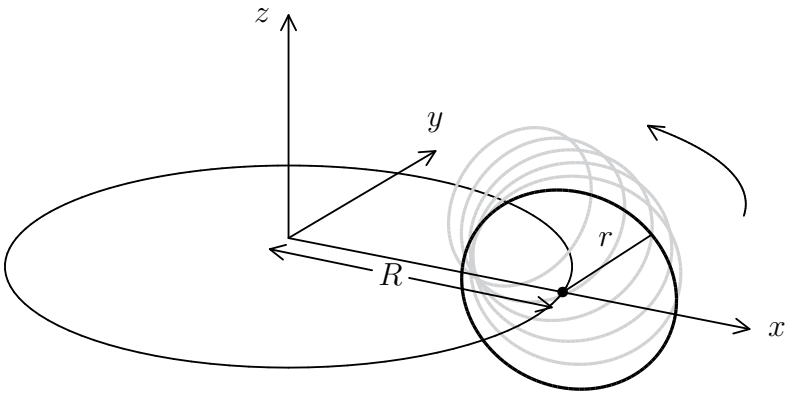
```
\InitGraph{12}{7}{6}{3.5}{1cm}
```

```
\DrawBoundary
\Viewpoint(30,60,5,6)[2] %
%
\SetCMYKColor(0,0,0,0.6)
\DDArrowAt(0,0,0)(1.5,0,0)
\DDArrowAt(0,0,0)(0,1.5,0)
\DDArrowAt(0,0,0)(0,0,1.4)
\SetCMYKColor(0,0,0,0.3)
\StandardSphere(1)
\Meridian(0,0,0,0,1)
\Meridian(0,0,0,90,1)
\Parallel(0,0,0,0,1)
\ResetColor
%
\SetCMYKColor(0,0,0,0.1)
\FullSphereAt(0.5,1,0)(0.8)
\ResetColor
\SetSphereAt(0.5,1,0)(0.8)
\ShowSphere
\SetCMYKColor(0,0,0,0.3)
\FullSphericalTriangle(0,90)(45,90)(45,55)
\FullSphericalTriangle(45,55)(90,90)(90,45)
\FullSphericalTriangle(0,45)(45,55)(0,0)
\ResetColor
\Parallel(0.5,1,0,0,0.8)
\Meridian(0.5,1,0,0,0.8)
\Meridian(0.5,1,0,90,0.8)
\SetThick{1.2pt}
\GreatCircleAt(0,90,90,45)
\GreatCircleAt(45,90,0,0)
\GreatCircleAt(0,45,90,90)
\SetCMYKColor(0,1,0,0)
\SphericalTriangle(0,90)(60,60)(0,10)
\ResetColor
\SetNormal
\CloseGraph
```

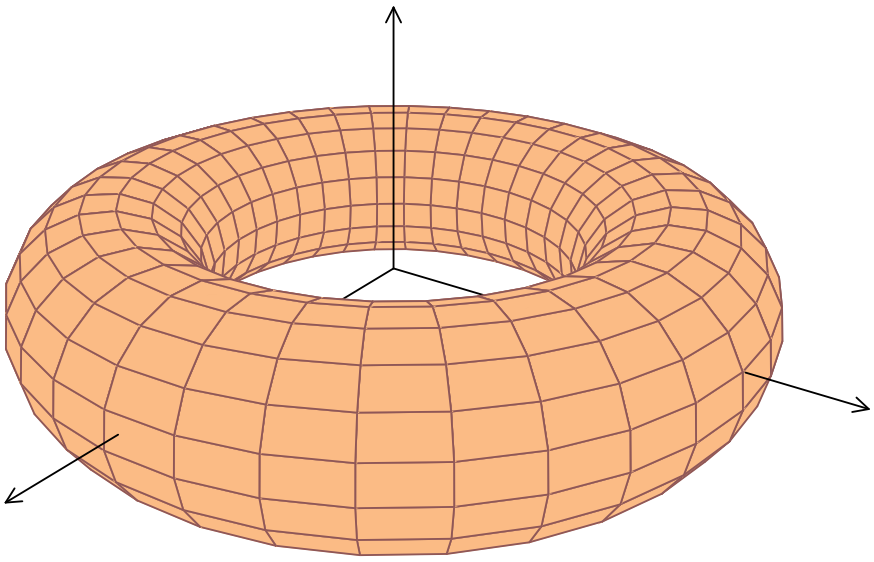
§ 8 Torus, Rotationsflächen und Sichtbarkeit

In diesem Abschnitt werden einige etwas anspruchsvollere Befehle vorgestellt. Da es um Sichtbarkeitsprüfungen geht, sind die Rechenzeiten manchmal ein bißchen länger.

Wir beginnen mit dem Torus: \`VisibleTorusAt`$(x_0, y_0, z_0)(R, r)$ erzeugt einen Torus mit Mittelpunkt (x_0, y_0, z_0), großem Radius R und kleinem Radius r. Er entsteht, indem man den Kreis mit Mittelpunkt $(R, 0)$ und Radius r in der x-z-Ebene um die z-Achse rotieren lässt:



Das Ergebnis sieht dann folgendermaßen aus:



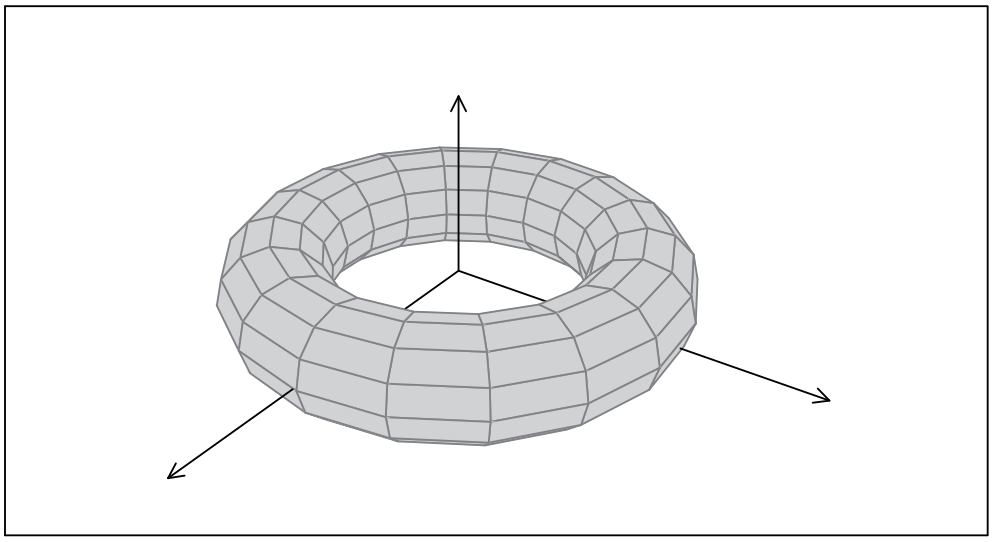
Die Farbgebung beim Torus wird mit den Befehlen \`SetBodyColor`(c, m, y, k) und \`SetLineColor`(c, m, y, k) gesteuert. So werden CMYK-Farben für die Flächen und

die Linien gewählt. Mit `\ResetBodyColor` und `\ResetLineColor` werden die Farben auf Graustufen zurückgesetzt.

Um das Bild etwas glatter zu machen, wurden hier noch die Befehle

`\SetMeridianDensity`$\{n\}$ und `\SetParallelDensity`$\{m\}$

eingesetzt. Es sind die Werte $1, 2$ für n und die Werte $1, 2, 3$ für m möglich. Ohne besondere Angaben oder mit den Befehlen `\ResetMeridianDensity` und `\ResetParallelDensity` wird $n = 2$ und $m = 3$ gesetzt. Normalerweise sollte man die Standardwerte benutzen, um die Performance in vernünftigen Grenzen zu halten;



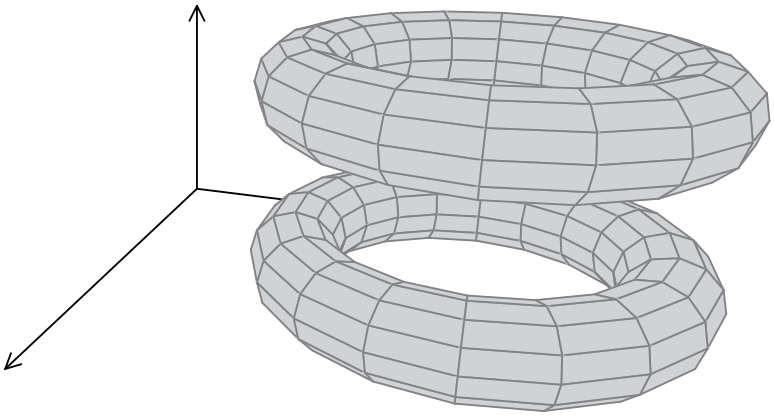
Mit den hier angegebenen Befehlsfolgen erhält man die beiden Versionen:

```
\SetBodyColor(0,0.3,0.5,0)
\SetLineColor(0,0.5,0.3,0.5)
\SetMeridianDensity{1}
\SetParallelDensity{2}
\VisibleTorusAt(0,0,0)(1.5,0.5)
```

bzw.

```
\VisibleTorusAt(0,0,0)(1,0.3)
```

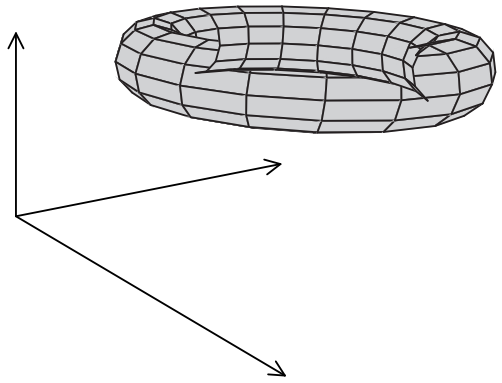
Man kann natürlich auch den Mittelpunkt des Torus wechseln.



Das geht mit

```
\InitGraph{13}{7}{5}{3.5}{1cm}
\Viewpoint(20,70,5,6)[2] %
\DDArrowAt(0,0,0)(2,0,0)
\DDArrowAt(0,0,0)(0,2,0)
\DDArrowAt(0,0,0)(0,0,1)
\VisibleTorusAt(0.5,1.5,0)(0.8,0.2)
\VisibleTorusAt(0.5,1.5,0.8)(0.8,0.2)
\CloseGraph
```

Wechselt man die Perspektive-Parameter, so kann man unangenehme Überraschungen erleben. Das liegt daran, dass sich der Torus zum Teil selbst verdeckt. Das wird vom Programm nicht abgefangen:



```
\InitGraph{13}{7}{5}{3.5}{1cm}
```

```
\Viewpoint(-30,70,5,6)[2]
\DDArrowAt(0,0,0)(2,0,0)
\DDArrowAt(0,0,0)(0,2,0)
\DDArrowAt(0,0,0)(0,0,1)
\VisibleTorusAt(0.5,1.5,0.8)(0.8,0.2)
\CloseGraph
```

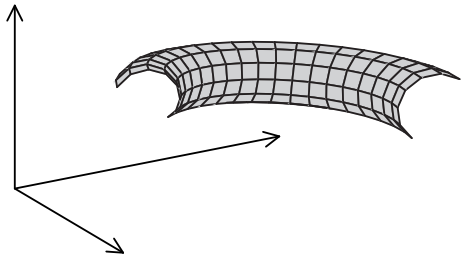
Die Vorsilbe „Visible“ sollte eigentlich suggerieren, dass man nur die sichtbaren Seiten des Torus erhält. Das geht hier leider schief, aber man kann sich behelfen.

`\TruncatedVisibleTorusAt`$(x_0, y_0, z_0)(R, r)(\alpha_1, \alpha_2)(\varphi_1, \varphi_2)$ zeichnet nur einen Teil des Torus

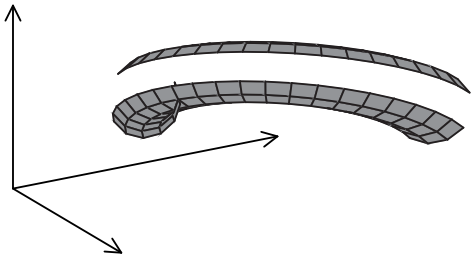
$$(u, v) \mapsto \big((R + r\cos\varphi)\cos\alpha, (R + r\cos\varphi)\sin\alpha, r\sin\varphi\big)$$

für $\alpha_1 \le \alpha \le \alpha_2$ und $\varphi_1 \le \varphi \le \varphi_2$.

`\TruncatedInvisibleTorusAt`$(x_0, y_0, z_0)(R, r)(\alpha_1, \alpha_2)(\varphi_1, \varphi_2)$ zeichnet den gleichen Teil, aber nur die unsichtbaren Partien.

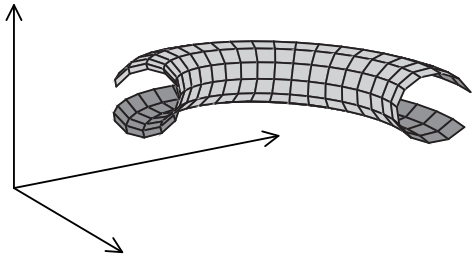


`\TruncatedVisibleTorusAt...`



`\TruncatedInvisibleTorusAt...`

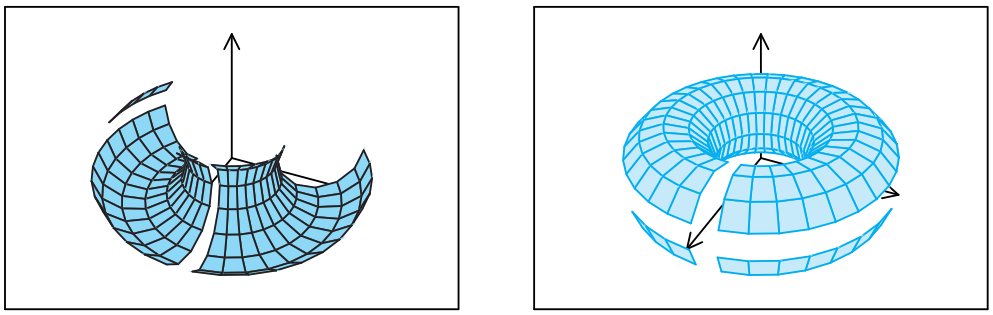
Zusammen ergibt das:



```
\SetBodyColor(0,0,0,0.5)
\TruncatedInvisibleTorusAt(0.5,1.5,0.4)(0.8,0.2)(45,245)(0,350)
\ResetBodyColor
\TruncatedVisibleTorusAt(0.5,1.5,0.4)(0.8,0.2)(45,245)(0,350)
```

Mit geschickten Kombinationen dieser Befehle kann man das oben angesprochene Problem lösen.

Hier ist ein weiteres Beispiel:



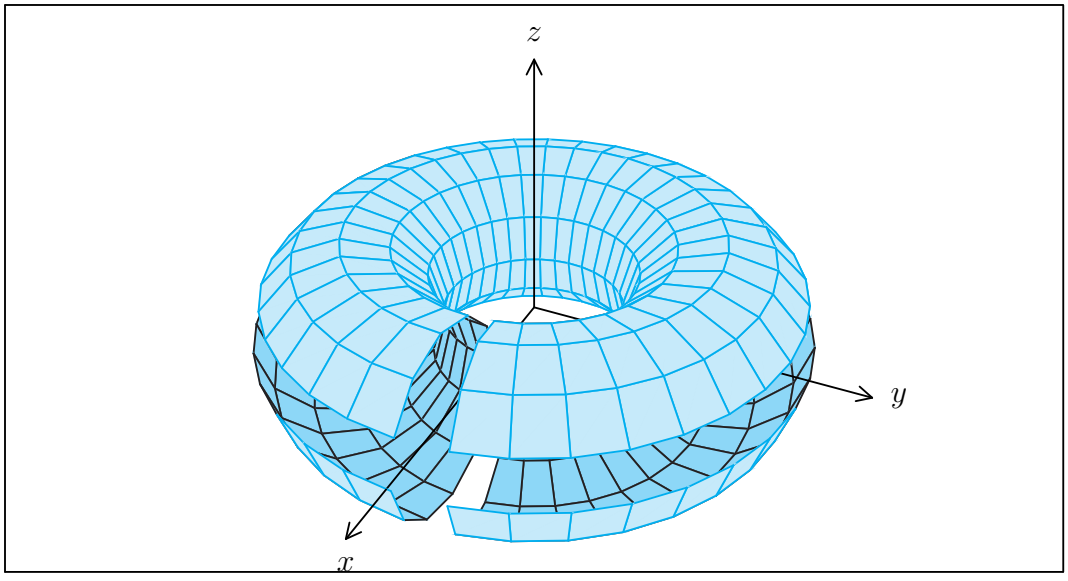
Die links verwendete Befehlsfolge

```
\SetBodyColor(0.4,0,0,0) \SetLineColor(0,0,0,1)
\TruncatedInvisibleTorusAt(0,0,0)(1,0.5)(239,361)(10,361)
\TruncatedInvisibleTorusAt(0,0,0)(1,0.5)(1,130)(10,361)
```

und die rechts verwendete Befehlsfolge

```
\SetBodyColor(0.2,0,0,0) \SetLineColor(1,0,0,0)
\TruncatedVisibleTorusAt(0,0,0)(1,0.5)(90,181)(10,361)
\TruncatedVisibleTorusAt(0,0,0)(1,0.5)(150,281)(10,361)
\TruncatedVisibleTorusAt(0,0,0)(1,0.5)(269,361)(10,361)
\TruncatedVisibleTorusAt(0,0,0)(1,0.5)(1,130)(10,361)
```

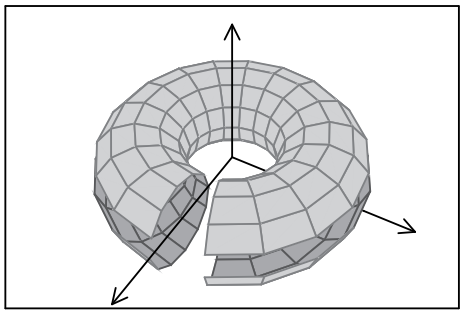
ergeben zusammen:



Für die richtige Positionierung der Achsenpfeile muss man selbst sorgen. Das ergibt folgendes Listing:

```
\InitGraph{14}{7.5}{7}{3.5}{1cm}
\DrawBoundary
\Viewpoint(25,55,5,6)[2]
\DDLineAt(0,0,0)(0.5,0,0)
\DDLineAt(0,0,0)(0,0.5,0)
\SetBodyColor(0.4,0,0,0) \SetLineColor(0,0,0,1)
\TruncatedInvisibleTorusAt(0,0,0)(1,0.5)(239,361)(10,361)
\TruncatedInvisibleTorusAt(0,0,0)(1,0.5)(1,130)(10,361)
\DDLineAt(0.5,0,0)(1.5,0,0)
\DDArrowAt(0,0.5,0)(0,1.8,0) \Text[r]{$y$}
\SetBodyColor(0.2,0,0,0) \SetLineColor(1,0,0,0)
\TruncatedVisibleTorusAt(0,0,0)(1,0.5)(90,181)(10,361)
\TruncatedVisibleTorusAt(0,0,0)(1,0.5)(150,281)(10,361)
\TruncatedVisibleTorusAt(0,0,0)(1,0.5)(269,361)(10,361)
\TruncatedVisibleTorusAt(0,0,0)(1,0.5)(1,130)(10,361)
\DDArrowAt(0,0,0)(0,0,1.4) \Text[t]{$z$}
\DDArrowAt(1.5,0,0)(1.8,0,0) \Text[b]{$x$}
\CloseGraph
```

Mit dem Befehl \SlicedTorusAt$(x_0, y_0, z_0)(R, r)$ erhält man ein ähnliches Bild. Allerdings sollte man nicht weit von der Einstellung \Viewpoint$(30, 45, 5, 6)[2]$ abweichen, sonst wird das Bild verfälscht.



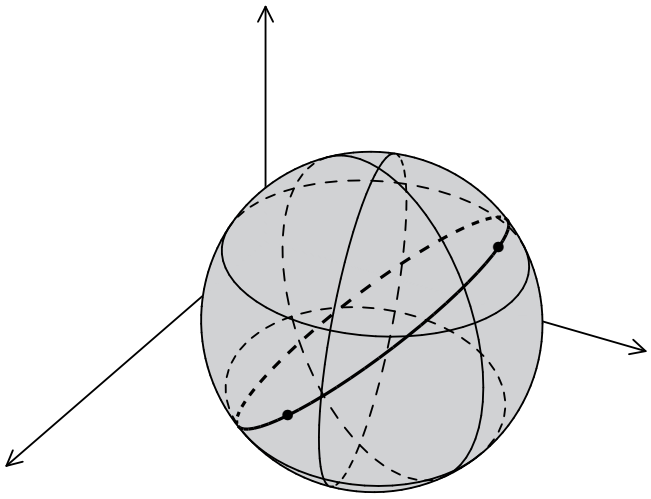
Bei der Entwicklung der Torus-Befehle ergaben sich auch komfortablere Befehle zum Zeichnen von Meridianen, Parallelkreisen und Großkreisen:

\VisibleMeridianAt$(x_0, y_0, z_0)(\alpha, r)$ zeichnet den sichtbaren Teil des Meridians der Kugel vom Radius r um (x_0, y_0, z_0). Dabei ist α der Winkel gegen die positive x-Achse. \InvisibleMeridianAt$(x_0, y_0, z_0)(\alpha, r)$ zeichnet den unsichtbaren Teil des Meridians als gestrichelte Linie.

\VisibleParallelAt$(x_0, y_0, z_0)(\alpha, r)$ und \InvisibleParallelAt$(x_0, y_0, z_0)(\alpha, r)$ besorgen das Gleiche für einen Breitenkreis (mit Breitenwinkel α).

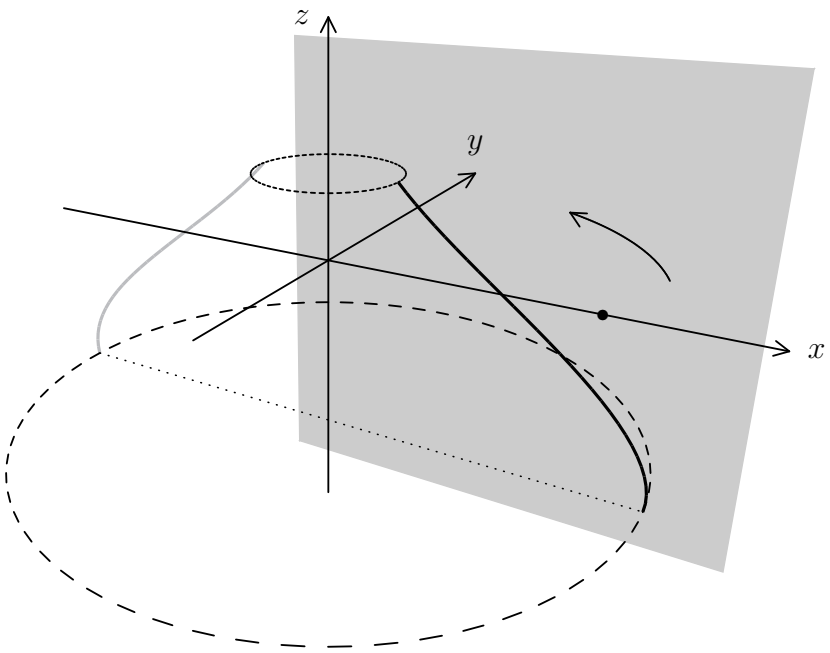
\VisibleGreatCircle$(\varphi_1, \theta_1, \varphi_2, \theta_2)$ zeichnet den sichtbaren Teil des Großkreises durch die beiden Punkte mit den Kugelkoordinaten (φ_1, θ_1) und (φ_2, θ_2), \InvisibleGreatCircle$(\varphi_1, \theta_1, \varphi_2, \theta_2)$ gestrichelt den unsichtbaren Teil. Zuvor muss die Kugel mit \SetSphereAt$(x_0, y_0, z_0)(R)$ festgelegt werden.

Hier kommt ein Anwendungsbeispiel:

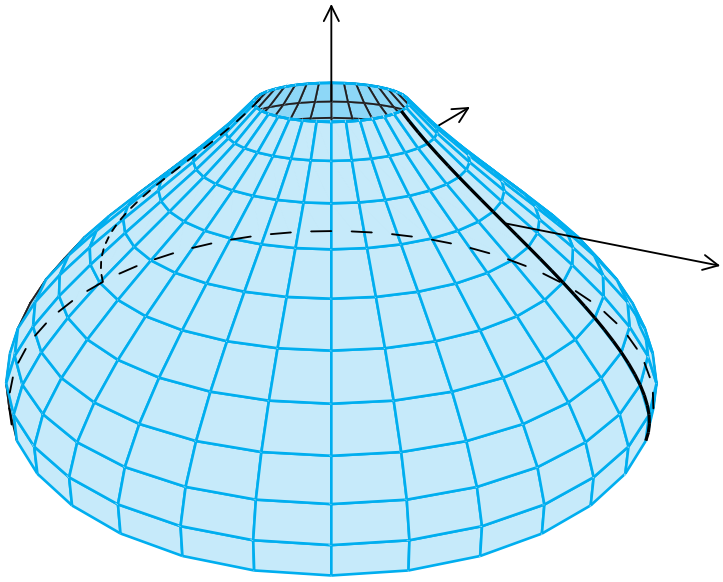


```
\InitGraph{12}{7}{6}{3.5}{1cm}
\Viewpoint(30,60,5,6)[2]
\DDArrowAt(0,0,0)(2,0,0)
\DDArrowAt(0,0,0)(0,2,0)
\DDArrowAt(0,0,0)(0,0,1.3)
\SetSphereAt(0.4,0.8,0)(0.8)
\SetCMYKColor(0,0,0,0.2)
\FullSphereAt(0.4,0.8,0)(0.8)
\ResetColor
\ShowSphere
\VisibleMeridianAt(0.4,0.8,0)(60,0.8)
\InvisibleMeridianAt(0.4,0.8,0)(60,0.8)
\VisibleMeridianAt(0.4,0.8,0)(10,0.8)
\InvisibleMeridianAt(0.4,0.8,0)(10,0.8)
\VisibleParallelAt(0.4,0.8,0)(30,0.8)
\InvisibleParallelAt(0.4,0.8,0)(30,0.8)
\VisibleParallelAt(0.4,0.8,0)(-30,0.8)
\InvisibleParallelAt(0.4,0.8,0)(-30,0.8)
\SetThick{1.2pt}
\VisibleGreatCircle(0,90,90,45)
\InvisibleGreatCircle(0,90,90,45)
\SetNormal
\DDMoveTo(0.4,0.8,0)
\SphRelMoveTo(0.8,0,90) \BigPoint
\DDMoveTo(0.4,0.8,0)
\SphRelMoveTo(0.8,90,45) \BigPoint
\CloseGraph
```

Lässt man eine beliebige Kurve in der x-z-Ebene um die z-Achse rotieren, so erhält man eine Rotationsfläche:



Diese Rotationsfläche sieht dann folgendermaßen aus:



Es gibt zwei Möglichkeiten, die Rotationsfläche zu erzeugen:

Die eine ist der Befehl \VisibleRotSurface$(\alpha_1, \alpha_2, t_1, t_2)$, wobei durch (α_1, α_2) (Angaben in Grad) der zu zeichnende Sektor begrenzt wird, und durch (t_1, t_2) das Parameterintervall für die erzeugende Kurve. Dazu muss dem System diese erzeugende Kurve vorher bekannt gemacht werden. Das geschieht durch \CALCRFUNCTIONBefehlsfolge.

Der Laufparameter steckt in der dimensionierten Variablen \TLAUF. Die Werte (x, z) müssen in den (ebenfalls dimensionierten) Variablen \XEINS und \ZEINS ausgegeben werden.

Im vorliegenden Fall ist die erzeugende Kurve gegeben durch

$$z \mapsto x = 0.5\,z^3 - 1.3\,z + 0.5.$$

Deshalb schreibt man z.B.:

```
\def\CALCRFUNCTION{%
  \ZEINS=\TLAUF
  \Wertvon<\ZEINS>nach<\zlauf>
  \Wertvon<\ZEROPOINT>nach<\ylauf>
  \KUBPOL<-0.5,0,1.3,-1>von<\TLAUF>nach<\UEINS>
  \Wertvon<\UEINS>nach<\xlauf>}
```

Soll der unsichtbare Teil ausgegeben werden, so benutzt man \InvisibleRotSurface$(\alpha_1, \alpha_2, t_1, t_2)$. Die Färbung wird wieder mit den Befehlen \SetBodyColor$(c, m, y, k)$ und \SetLineColor$(c, m, y, k)$ gesteuert.

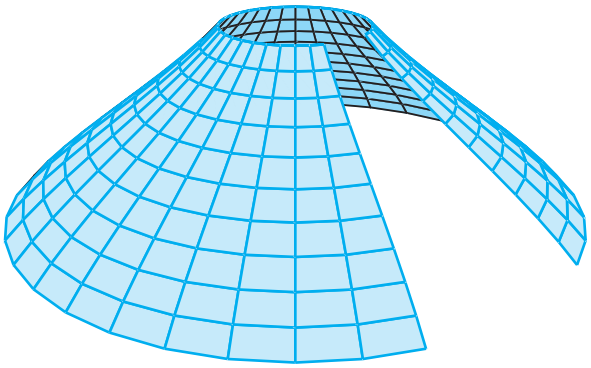
Die zweite Möglichkeit für das Zeichnen einer Rotationsfläche liefert der Befehl \VisUserRotatedSurface$(\alpha_1, \alpha_2, t_1, t_2)[n]${Befehlsfolge}, der ebenfalls nur die sichtbare Seite zeichnet. Die Befehlsfolge ist die Gleiche wie oben bei Benutzung von \CALCRFUNCTION. Mit \InvisUserRotatedSurface... erhält man den unsichtbaren Teil.

Das Listing für das Bild mit der Rotationsfläche kann z.B. folgendermaßen aussehen:

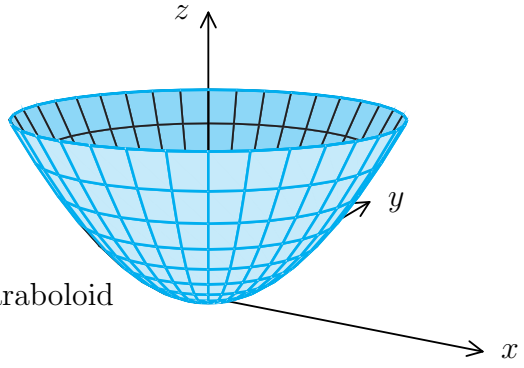
```
\InitGraph{14}{8.5}{6.5}{5.5}{1cm}
\Viewpoint(-60,70,5,6)[2]
\DDArrowAt(0,0,0)(0,2,0)
\SetBodyColor(0.4,0,0,0)
\SetLineColor(0,0,0,1)
\InvisUserRotatedSurface(0,360,-1,0.5)[10]{%
    \ZEINS=\TLAUF
    \KUBPOL<0.5,0,-1.3,1>von<\TLAUF>nach<\XEINS>}
\SetBodyColor(0.2,0,0,0)
\SetLineColor(1,0,0,0)
\VisUserRotatedSurface(0,360,-1,0.5)[10]{%
    \ZEINS=\TLAUF
    \KUBPOL<0.5,0,-1.3,1>von<\TLAUF>nach<\XEINS>}
\SetThick{1.2pt}
\DDUserCurve(-1,0.5)[30]{%
  \ZEINS=\TLAUF
  \Wertvon<\ZEINS>nach<\zlauf>
```

```
  \Wertvon<\ZEROPOINT>nach<\ylauf>
  \KUBPOL<0.5,0,-1.3,1>von<\TLAUF>nach<\UEINS>
  \Wertvon<\UEINS>nach<\xlauf>}
\SetDashed
\DDUserCurve(-1,0.5)[30]{%
  \ZEINS=\TLAUF
  \Wertvon<\ZEINS>nach<\zlauf>
  \Wertvon<\ZEROPOINT>nach<\ylauf>
  \KUBPOL<-0.5,0,1.3,-1>von<\TLAUF>nach<\UEINS>
  \Wertvon<\UEINS>nach<\xlauf>}
\SetNormal
\InvisibleParallelAt(0,0,-1)(0,1.8)
\DDArrowAt(0,0,0.5)(0,0,1)
\DDArrowAt(1,0,0)(2,0,0)
\CloseGraph
```

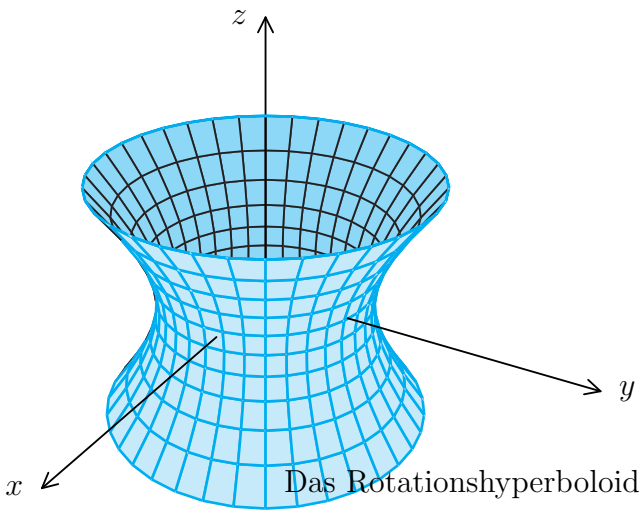
Mit den Parametern $\alpha_1, \alpha_2, t_1, t_2$ kann man steuern, dass nur ein Ausschnitt der Fläche dargestellt wird:



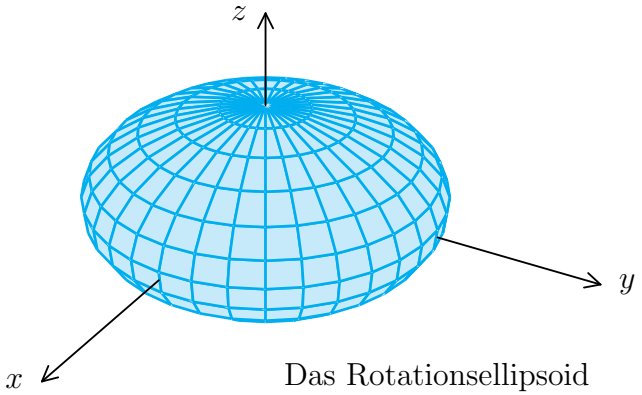
Es folgen einige voreingestellte Rotationsflächen.



Das Rotationsparaboloid



Das Rotationshyperboloid



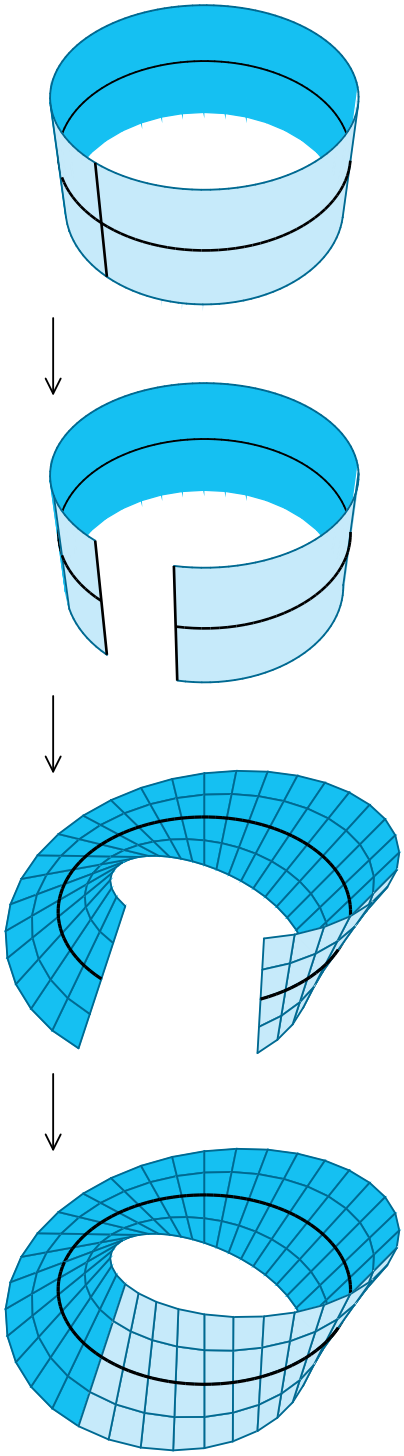
Das Rotationsellipsoid

Man erzeugt sie mit den Befehlen

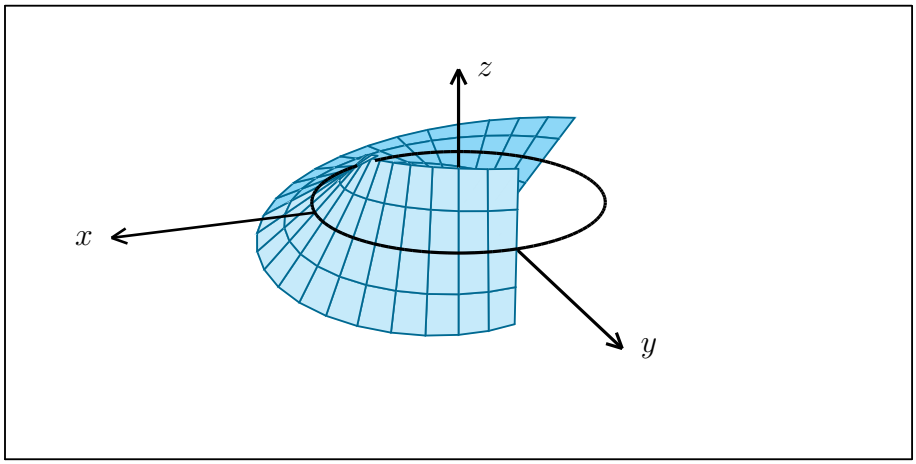
`\Paraboloid`, `\Hyperboloid` und `\Ellipsoid`.

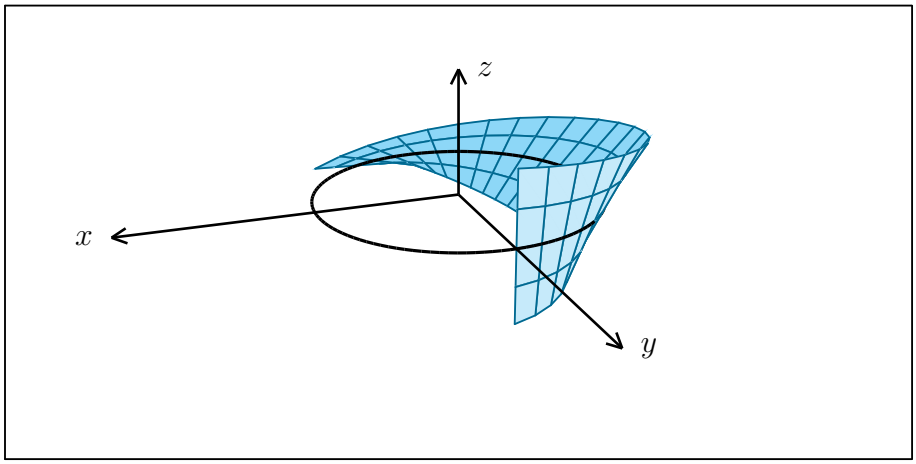
§ 9 Möbiusband und Wendelfläche

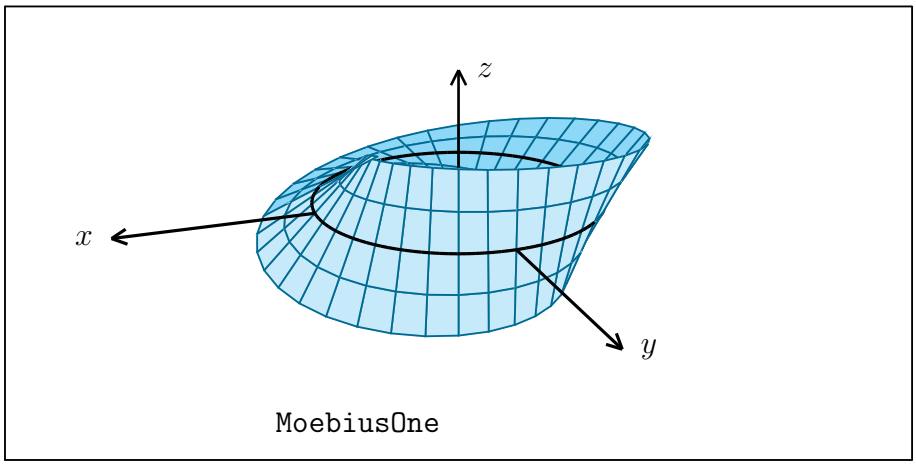
Jeder kennt das Möbiusband, das entsteht, indem man einen zylindrischen Papierstreifen zerschneidet und verkehrt herum zusammenklebt:



Da es relativ schwierig ist, sich einen klaren Überblick über die Geometrie des Möbiusbandes zu verschaffen, werden hier im Folgenden erst mal zwei Teile des Bandes und dann das komplette Band dargestellt.







Benutzt werden dabei der Befehl \MoebiusOne (für das komplette Möbiusband) und \MoebiusOnePart(w_1, w_2) für einen Teil des Möbiusbandes zwischen den Winkeln w_1 und w_2.

```
\InitGraph{12}{6}{6}{3.5}{1cm}
\DrawBoundary
\Viewpoint(70,70,5,6)[2]
\SetBodyColor(0.4,0,0,0) \SetLineColor(1,0,0,0.5)
\MoebiusOnePart(100,240)
\SetThick{1.2pt}
\DDArrowAt(0,0,0)(0,1.8,0) \Text[r]{$y$}
\DDArrowAt(0,0,0)(0,0,0.7) \Text[r]{$z$}
\SetNormal
\SetBodyColor(0.2,0,0,0)
\MoebiusOnePart(0,120)
\SetThick{1.2pt}
\PArc(0,0,0,0.8)(0,290)
\PArc(0,0,0,0.8)(300,360)
\SetThick{1pt}
\DDArrowAt(0.8,0,0)(1.8,0,0) \Text[l]{$x$}
\SetNormal
\CloseGraph

\InitGraph{12}{6}{6}{3.5}{1cm}
\DrawBoundary
\Viewpoint(70,70,5,6)[2]
\SetBodyColor(0.4,0,0,0) \SetLineColor(1,0,0,0.5)
\MoebiusOnePart(150,300)
\SetThick{1.2pt} \PArc(0,0,0,0.8)(90,360) \SetNormal
\SetBodyColor(0.2,0,0,0)
\MoebiusOnePart(300,360)
\SetThick{1.2pt}
\PArc(0,0,0,0.8)(0,130)
\SetThick{1pt}
\DDArrowAt(0,0,0)(1.8,0,0) \Text[l]{$x$}
\DDArrowAt(0,0,0)(0,1.8,0) \Text[r]{$y$}
\DDArrowAt(0,0,0)(0,0,0.7) \Text[r]{$z$}
\SetNormal
\CloseGraph

\InitGraph{12}{6}{6}{3.5}{1cm}
\DrawBoundary
\Viewpoint(70,70,5,6)[2]
\SetBodyColor(0.4,0,0,0) \SetLineColor(1,0,0,0.5)
\MoebiusOnePart(120,300)
```

```
\SetThick{1.2pt}
\PCircle(0,0,0,0.8)
\SetThick{1pt}
\DDArrowAt(0,0,0)(0,0,0.7) \Text[r]{$z$}
\SetNormal
\SetBodyColor(0.2,0,0,0)
\MoebiusOnePart(0,120)
\MoebiusOnePart(300,360)
\SetThick{1.2pt}
\PArc(0,0,0,0.8)(0,130) \PArc(0,0,0,0.8)(330,360)
\DDArrowAt(0.8,0,0)(1.8,0,0) \Text[l]{$x$}
\DDArrowAt(0,0.8,0)(0,1.8,0) \Text[r]{$y$}
\SetNormal
\TextAt(0,-3)[l]{\texttt{MoebiusOne}}
\CloseGraph
```

Das „One“ im Namen des Befehls signalisiert, dass es noch eine zweite Version gibt.

`\MoebiusOne` wird parametrisiert durch

$$(u,v) \mapsto \big((0.8 - v\sin(u/2)) \cdot \sin u,\ (0.8 - v\sin(u/2)) \cdot \cos u,\ v\cos(u/2)\big),$$

mit $0 \le u \le 2\pi$ und $-0.4 \le v \le 0.4$. Im Falle $v = 0$ erhält man den Kreis

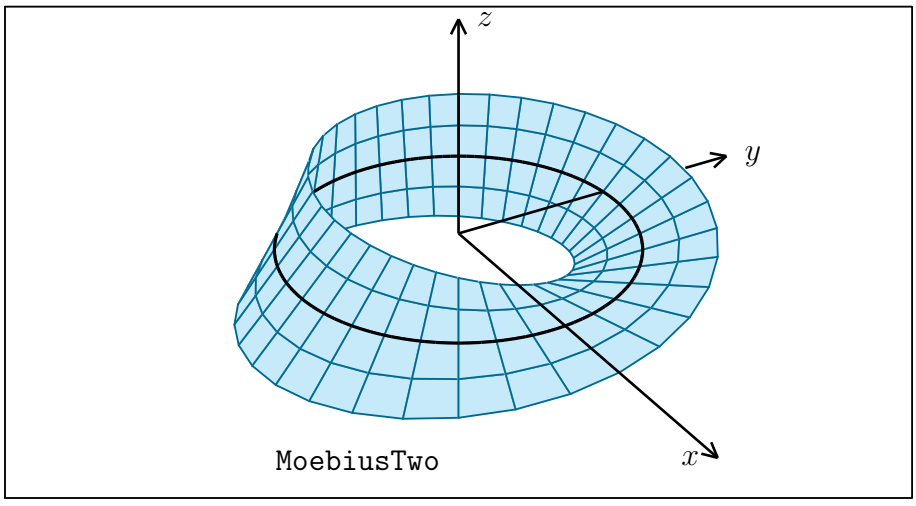
$$u \mapsto \big(0.8\sin u,\ 0.8\cos u,\ 0\big).$$

Wie man sieht, wird dieser Kreis „verkehrt herum“ durchlaufen. Die Parameter 0, $\pi/2$, π, $3\pi/2$ und 2π liefern nacheinander die Punkte $(0, 0.8, 0)$, $(0.8, 0, 0)$, $(0, -0.8, 0)$, $(-0.8, 0, 0)$ und wieder $(0, 0.8, 0)$.

Alternativ wird der Befehl `\MoebiusTwo` zur Verfügung gestellt. Dahinter steht die Parametrisierung

$$(u,v) \mapsto \big((1 + v\cos(u/2)) \cdot \cos u,\ (1 + v\cos(u/2)) \cdot \sin u,\ v\sin(u/2)\big),$$

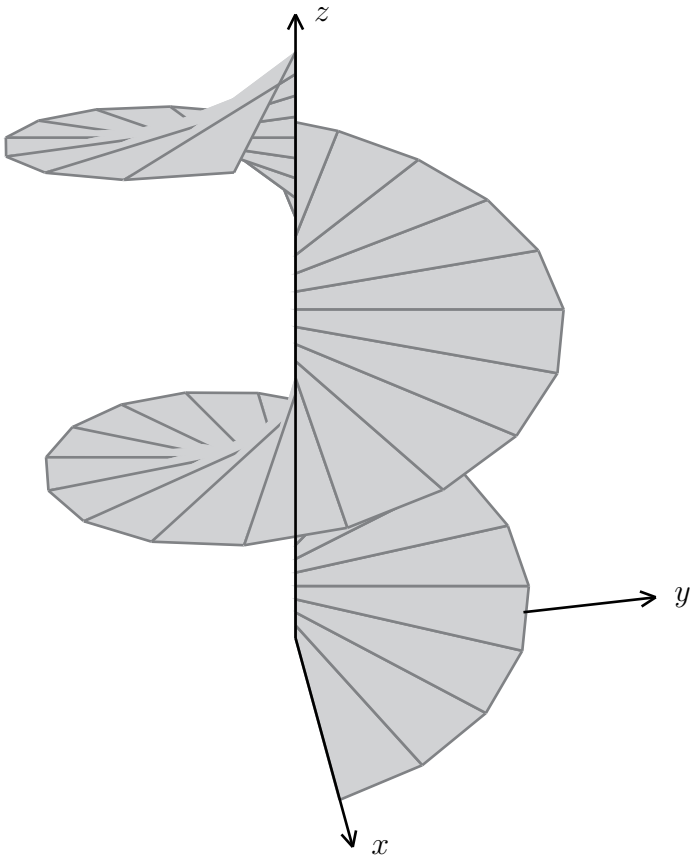
mit $0 \le u \le 2\pi$ und $-0.4 \le v \le 0.4$. Das Bild sieht dann folgendermaßen aus:



```
\InitGraph{12}{6.5}{6}{3.5}{1cm}
\DrawBoundary
\Viewpoint(-30,60,5,6)[2]
\SetLineColor(1,0,0,0.5)
\SetBodyColor(0.2,0,0,0)
\MoebiusTwo
\SetThick{1.2pt}
\PArc(0,0,0,1)(0,210)
\PArc(0,0,0,1)(240,360)
\SetThick{1pt}
\DDArrowAt(0,0,0)(2,0,0) \Text[l]{$x$}
\DDLineAt(0,0,0)(0,1,0)
\DDArrowAt(0,1.65,0)(0,2,0) \Text[r]{$y$}
\DDArrowAt(0,0,0)(0,0,1.2) \Text[r]{$z$}
\SetNormal
\TextAt(0,-3)[l]{\texttt{MoebiusTwo}}
\CloseGraph
```

Im Falle $v = 0$ erhält man diesmal den Kreis $u \mapsto (\cos u, \sin u, 0)$, und der wird richtigherum durchlaufen.

Das Möbiusband ist eine „nicht-orientierbare“ Fläche, im Gegensatz zu der nun betrachteten Wendelfläche.



Sie wird erzeugt mit dem Befehl `\Helicoid`(t_1, t_2). Sie ist gegeben durch die Parametrisierung

$$(u, v) \mapsto \big(v\,\cos u,\; v\,\sin u,\; \frac{1}{4}\,u\big),$$

mit $0 \le v \le 1.2$ und $t_1 \le u \le t_2$.

Damit ein schönes Bild entsteht, muss man die Perspektive-Einstellungen geeignet wählen, etwa wie folgt:

```
\InitGraph{12}{10.5}{6}{2.5}{1cm}
\Viewpoint(-10,50,8,10)[2]
\Helicoid(0,12)
\SetThick{1pt}
\DDArrowAt(0,0,0)(1.5,0,0) \Text[r]{$x$}
\DDArrowAt(0,1.25,0)(0,2,0) \Text[r]{$y$}
\DDArrowAt(0,0,0)(0,0,3.2) \Text[r]{$z$}
\SetNormal
\CloseGraph
```

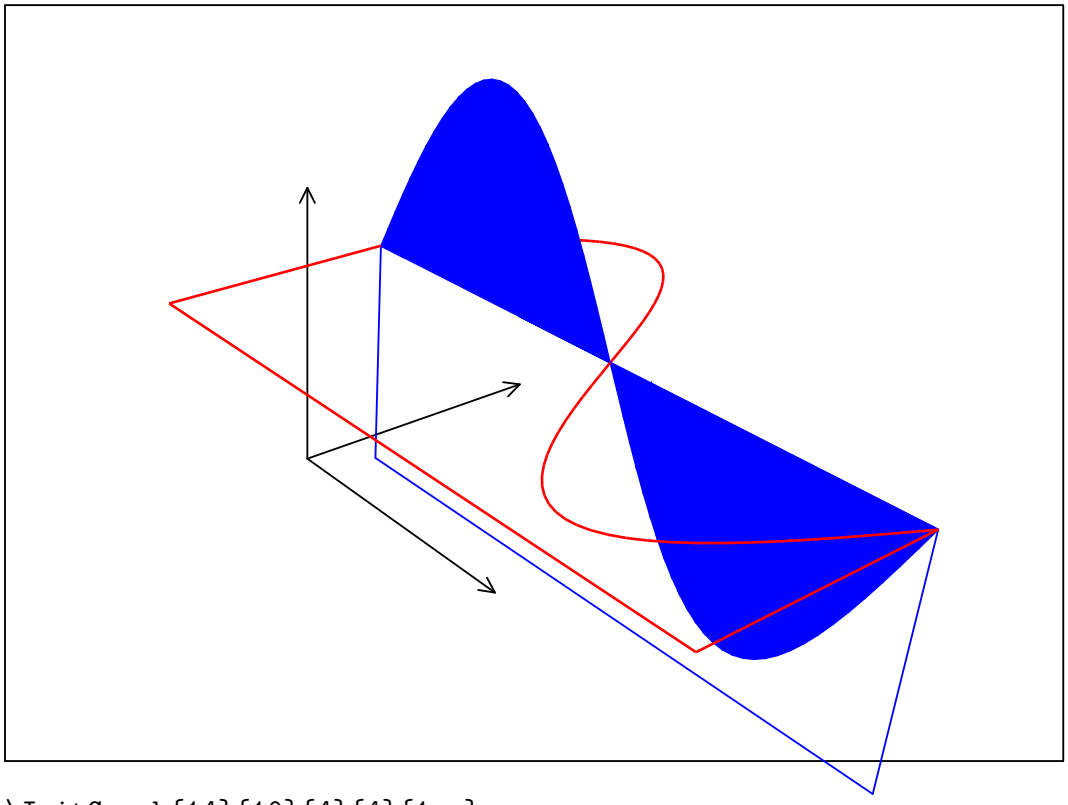
§ 10 Kurven und Graphen

\DDUserCurve$(t_1,t_2)[n]\{\ldots\}$ funktioniert so ähnlich wie die 2-dimensionale User-Kurve, zeichnet aber 3-dimensional. In der Funktionsroutine ist \TLAUF der Lauf-Parameter. Übergeben werden die dimensionslosen Koordinaten \xlauf, \ylauf und \zlauf.

Möchte man unter der Kurve farbig schattieren, so geht das mit der Befehlsfolge

```
\SetBlue % <--- Farbe festlegen
\ZweiDeOf<0.8,0.3,1.2>To<\ueins,\veins> % <--- Kegelspitze festlegen
\SetVertex(\ueins,\veins) % (2d-Koordinaten der Kegelspitze angeben)
\PaintCone % ab jetzt Kegel zeichnen!
\DDUserCurve(...,...)[...]{...}
\StopCone % jetzt nicht mehr Kegel zeichnen!
```

Beispiel:



```
\InitGraph{14}{10}{4}{4}{1cm}
\DrawBoundary
\Viewpoint(-35,60,8,10)[2]
\DDArrowAt(0,0,0)(1.5,0,0)
\DDArrowAt(0,0,0)(0,1.5,0)
\DDArrowAt(0,0,0)(0,0,1.5)
\SetRed
\SetThick{1pt}
```

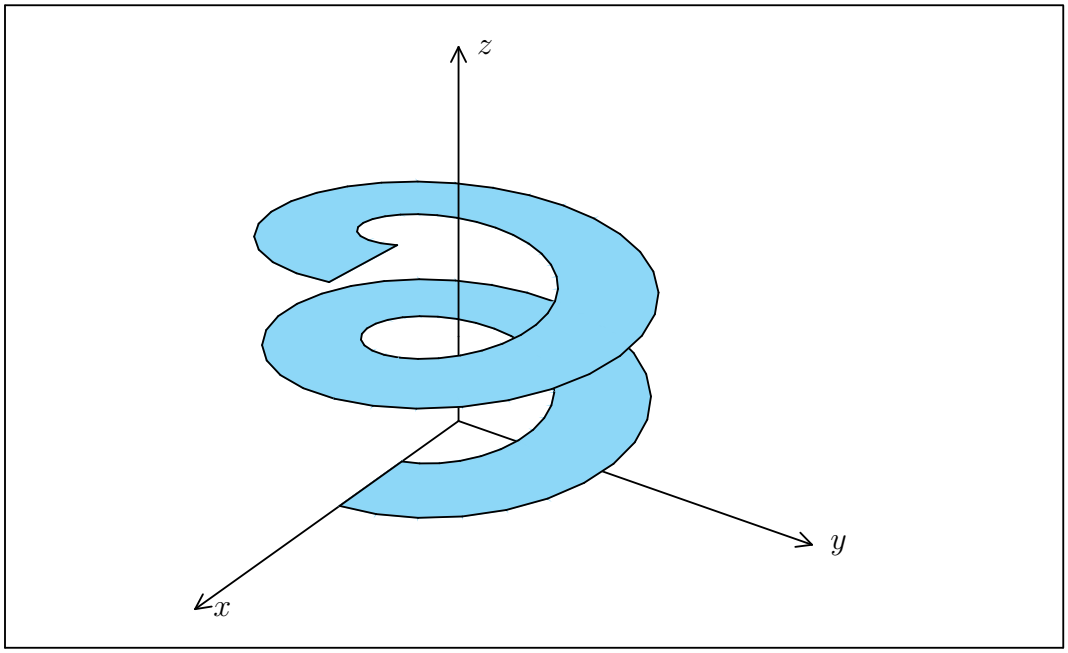
```
\DDUserCurve(0,\PiHalbe)[30]{%
  \UEINS=\TLAUF
  \advance\UEINS by 0.2\ONEPOINT
  \Wertvon<\UEINS>nach<\xlauf>
  \VEINS=1.2\ONEPOINT
  \Wertvon<\VEINS>nach<\zlauf>
  \SINFKT<1,2,0,0>von<\TLAUF>nach<\ZEINS>
  \advance\ZEINS by 0.3\ONEPOINT
  \Wertvon<\ZEINS>nach<\ylauf>}
\SetNormal
\SetBlue
\ZweiDeOf<0.8,0.3,1.2>To<\ueins,\veins>
\SetVertex(\ueins,\veins)
\PaintCone
\DDUserCurve(0,\PiHalbe)[30]{%
  \UEINS=\TLAUF
  \advance\UEINS by 0.2\ONEPOINT
  \Wertvon<\UEINS>nach<\xlauf>
  \VEINS=0.3\ONEPOINT
  \Wertvon<\VEINS>nach<\ylauf>
  \SINFKT<1,2,0,0>von<\TLAUF>nach<\ZEINS>
  \advance\ZEINS by 1.2\ONEPOINT
  \Wertvon<\ZEINS>nach<\zlauf>}
\StopCone
\ZweiDeOf<2,0.3,1.2>To<\ueins,\veins>
\SetVertex(\ueins,\veins)
\PaintCone
\DDUserCurve(\PiHalbe,\EinPi)[30]{%
  \UEINS=\TLAUF
  \advance\UEINS by 0.2\ONEPOINT
  \Wertvon<\UEINS>nach<\xlauf>
  \VEINS=0.3\ONEPOINT
  \Wertvon<\VEINS>nach<\ylauf>
  \SINFKT<1,2,0,0>von<\TLAUF>nach<\ZEINS>
  \advance\ZEINS by 1.2\ONEPOINT
  \Wertvon<\ZEINS>nach<\zlauf>}
\StopCone
\DDLineAt(0.2,0.3,1.2)(0.2,0.3,0)
\DDLineTo(3.34,0.3,0)
\DDLineTo(3.34,0.3,1.2)
\DDLineAt(0.2,0.3,1.2)(3.34,0.3,1.2)
\SetRed
\SetThick{1pt}
\DDUserCurve(\PiHalbe,\EinPi)[30]{%
```

```
  \UEINS=\TLAUF
  \advance\UEINS by 0.2\ONEPOINT
  \Wertvon<\UEINS>nach<\xlauf>
  \VEINS=1.2\ONEPOINT
  \Wertvon<\VEINS>nach<\zlauf>
  \SINFKT<1,2,0,0>von<\TLAUF>nach<\ZEINS>
  \advance\ZEINS by 0.3\ONEPOINT
  \Wertvon<\ZEINS>nach<\ylauf>}
\DDPolyLine(0.2,0.3,1.2)(0.2,-0.9,1.2)(3.34,-0.9,1.2)(3.34,0.3,1.2)
\SetNormal
\ResetColor
\CloseGraph
```

Interessante Effekte kann man mit dem Befehl

$$\texttt{\textbackslash DDBetweenUserCurves}(t_1,t_2)[n]\{\text{Befehlsfolge1}\}\{\text{Befehlsfolge2}\}$$

erzielen. Wählt man eine Farbe, so wird der Bereich zwischen zwei benutzerdefinierten Kurven eingefärbt.



Der Lauf-Parameter wird wie üblich in der Variablen `\TLAUF` übergeben. Das Ergebnis der ersten Befehlsfolge wird dann in den dimensionslosen Koordinaten `\xlauf`, `\ylauf` und `\zlauf` gespeichert, das Ergebnis der zweiten Befehlsfolge in `\xulauf`, `\yulauf` und `\zulauf`.

Die obige Zeichnung mit der spiralförmigen Rampe erhält man mit

```
\InitGraph{14}{8.5}{6}{3}{1cm}
\DrawBoundary
```

```
\Viewpoint(35,60,8,10)[2]
\DDArrowAt(0,0,0)(2,0,0) \Text[r]{$x$}
\DDArrowAt(0,0,0)(0,2,0) \Text[r]{$y$}
\DDLineAt(0,0,0)(0,0,0.5)
%
\SetCMYKColor(0.4,0,0,0)
\DDBetweenUserCurves(0,\ZweiPi)[30]{%
  \COSvon<\TLAUF>nach<\XEINS>
  \SINvon<\TLAUF>nach<\YEINS>
  \ZEINS=0.0942\TLAUF
      \Wertvon<\XEINS>nach<\xlauf>
      \Wertvon<\YEINS>nach<\ylauf>
      \Wertvon<\ZEINS>nach<\zlauf>}{%
  \COSvon<\TLAUF>nach<\XEINS>
  \XEINS=0.5\XEINS
  \SINvon<\TLAUF>nach<\YEINS>
  \YEINS=0.5\YEINS
  \ZEINS=0.0942\TLAUF
      \Wertvon<\XEINS>nach<\xulauf>
      \Wertvon<\YEINS>nach<\yulauf>
      \Wertvon<\ZEINS>nach<\zulauf>}
%
\ResetColor
\DDUserCurve(0,\ZweiPi)[30]{%
  \COSvon<\TLAUF>nach<\XEINS>
  \SINvon<\TLAUF>nach<\YEINS>
  \ZEINS=0.0942\TLAUF
      \Wertvon<\XEINS>nach<\xlauf>
      \Wertvon<\YEINS>nach<\ylauf>
      \Wertvon<\ZEINS>nach<\zlauf>}
\DDUserCurve(0,\ZweiPi)[30]{%
  \COSvon<\TLAUF>nach<\XEINS>
  \XEINS=0.5\XEINS
  \SINvon<\TLAUF>nach<\YEINS>
  \YEINS=0.5\YEINS
  \ZEINS=0.0942\TLAUF
      \Wertvon<\XEINS>nach<\xlauf>
      \Wertvon<\YEINS>nach<\ylauf>
      \Wertvon<\ZEINS>nach<\zlauf>}
%
\SetCMYKColor(0.4,0,0,0)
\DDBetweenUserCurves(\ZweiPi,12.56637)[30]{%
  \COSvon<\TLAUF>nach<\XEINS>
  \SINvon<\TLAUF>nach<\YEINS>
```

```
  \ZEINS=0.0942\TLAUF
      \Wertvon<\XEINS>nach<\xlauf>
      \Wertvon<\YEINS>nach<\ylauf>
      \Wertvon<\ZEINS>nach<\zlauf>}{%
  \COSvon<\TLAUF>nach<\XEINS>
  \XEINS=0.5\XEINS
  \SINvon<\TLAUF>nach<\YEINS>
  \YEINS=0.5\YEINS
  \ZEINS=0.0942\TLAUF
      \Wertvon<\XEINS>nach<\xulauf>
      \Wertvon<\YEINS>nach<\yulauf>
      \Wertvon<\ZEINS>nach<\zulauf>}
%
\ResetColor
\DDUserCurve(\ZweiPi,12.56637)[30]{%
  \COSvon<\TLAUF>nach<\XEINS>
  \SINvon<\TLAUF>nach<\YEINS>
  \ZEINS=0.0942\TLAUF
      \Wertvon<\XEINS>nach<\xlauf>
      \Wertvon<\YEINS>nach<\ylauf>
      \Wertvon<\ZEINS>nach<\zlauf>}
\DDUserCurve(\ZweiPi,12.56637)[30]{%
  \COSvon<\TLAUF>nach<\XEINS>
  \XEINS=0.5\XEINS
  \SINvon<\TLAUF>nach<\YEINS>
  \YEINS=0.5\YEINS
  \ZEINS=0.0942\TLAUF
      \Wertvon<\XEINS>nach<\xlauf>
      \Wertvon<\YEINS>nach<\ylauf>
      \Wertvon<\ZEINS>nach<\zlauf>}
%
\DDLineAt(0.5,0,0)(1,0,0)
\DDLineAt(0.5,0,1.183752)(1,0,1.183752)
\DDArrowAt(0,0,0.5)(0,0,2) \Text[r]{$z$}
\CloseGraph
```

Das Listing ist so kompliziert, weil man zunächst die Teile zeichnen muss, die später überdeckt werden, und dann die überdeckenden Teile.

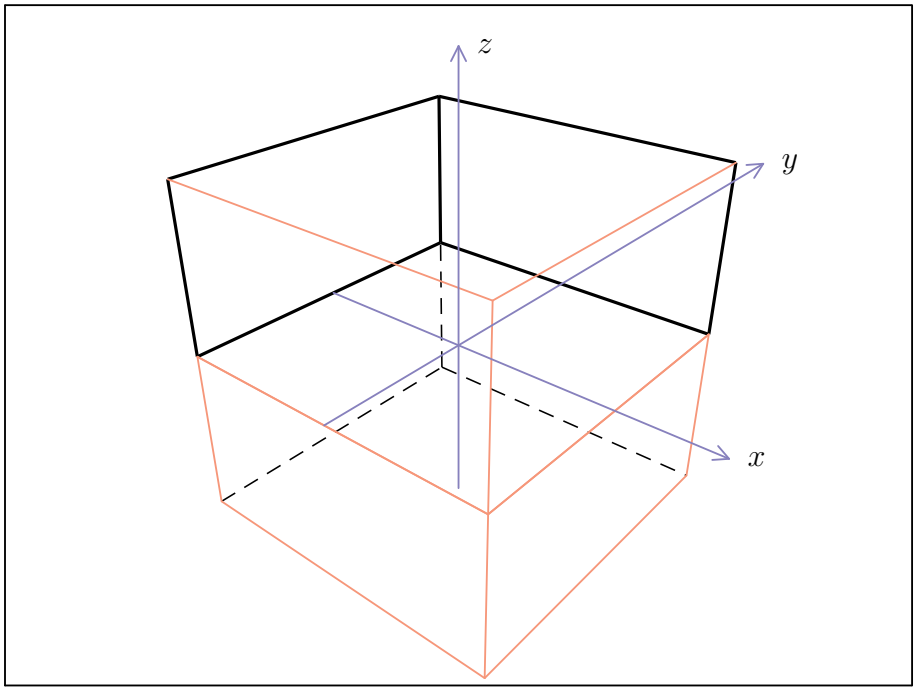
Es ist auch möglich, 2-dimensionale Funktionsgraphen zu zeichnen. Das ist ein etwas komplexes Unterfangen. Man benutzt den Befehl

$$\texttt{\textbackslash DDFunctionLine}(t_1, t_2)\{\text{1.Befehlsfolge}\}\{\text{2.Befehlsfolge}\}.$$

Das Verfahren lässt sich am besten an einem Beispiel erklären. Betrachtet werde die Funktion

$$(x, y) \mapsto f(x, y) := \sin(xy), \qquad \text{auf } [-1, 1] \times [-1, 1].$$

Um den Graphen gut sichtbar zu machen, wird eine Box gezeichnet, innerhalb der sich alles abspielt, nämlich der Quader $[-1, 1] \times [-1, 1] \times [-1, 1]$. In dieser Box wird der darin enthaltene Teil der x-y-Ebene eingezeichnet, der Graph ist eine Fläche über diesem Ebenenteil:



Für die Perspektive wird \Viewpoint$(-50, 60, 5, 6)[2]$ verwendet, alles bezieht sich auf diese spezielle Blickrichtung. Die beiden Wände im Hintergrund (die von der Blickrichtung abhängen) müssen zuerst gezeichnet werden, da sie später zum Teil verdeckt werden. Der Teil, der unter der x-y-Ebene liegt, ist gestrichelt. Die Wände **vor** dem Graphen werden als letztes gezeichnet, sie sind hier nur der Deutlichkeit halber farbig angedeutet, ebenso die Koordinatenachsen, um die man sich am besten ganz zum Schluss kümmert.

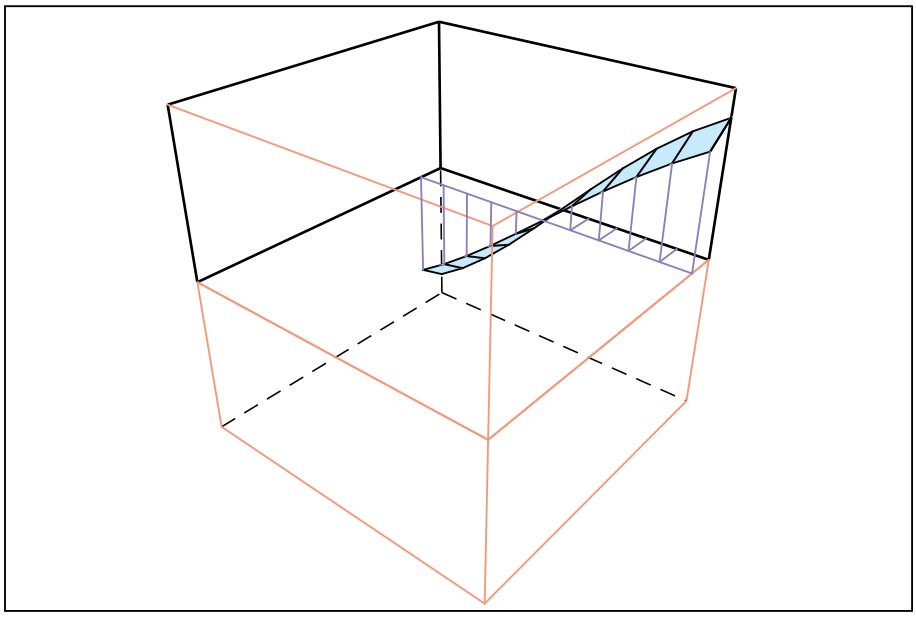
Der eigentliche Graph wird schichtweise von hinten nach vorne gezeichnet. Das hat den Vorteil, dass verdeckte Teile - wegen der deckenden Farbgebung - tatsächlich auch von den weiter vorne liegenden Partien des Graphen verdeckt werden. Mit

```
\DDFunctionLine(-1,1){%
\Wertvon<\TLAUF>nach<\xlauf>
\TZWEI=1pt%
\Wertvon<\TZWEI>nach<\ylauf>
\XEINS=\TLAUF
\SINFKT<1,1,0,0>von<\XEINS>nach<\TEINS>
```

```
\Wertvon<\TEINS>nach<\zlauf>}{%
\Wertvon<\TLAUF>nach<\xulauf>
\TZWEI=0.8pt
\Wertvon<\TZWEI>nach<\yulauf>
\XEINS=0.8\TLAUF
\SINFKT<1,1,0,0>von<\XEINS>nach<\TEINS>
\Wertvon<\TEINS>nach<\zulauf>}
```

wird die hinterste Schicht gezeichnet. Dabei läuft x von -1 bis 1, und y wird einmal $= 1$ und einmal $= 0.8$ gesetzt. Der Befehl \DDFunctionLine$(-1,1)\{B_1\}\{B_2\}$ erzeugt intern die beiden Befehle \CALCPOINTS und \UCALCPOINTS, deren Wirkung durch die Befehlsfolgen B_1 bzw. B_2 festgelegt sind. Sie berechnen die Werte der Kurven $t \mapsto \big(t, 1, \sin(t \cdot 1)\big)$ und $t \mapsto \big(t, 0.8, \sin(t \cdot 0.8)\big)$.

Nachdem dies geschehen ist, ruft \DDFunctionLine den Befehl \DDFuncQuad auf, der seinerseits in einer Schleife kleine hellblau gefärbte Vierecke zeichnet, jeweils eins über dem Quadrat mit den Ecken $(x, 0.8)$, $(x+0.2, 0.8)$, $(x+0.2, 1)$ und $(x, 1)$. Das ist eine Approximation des Funktionsgraphen.

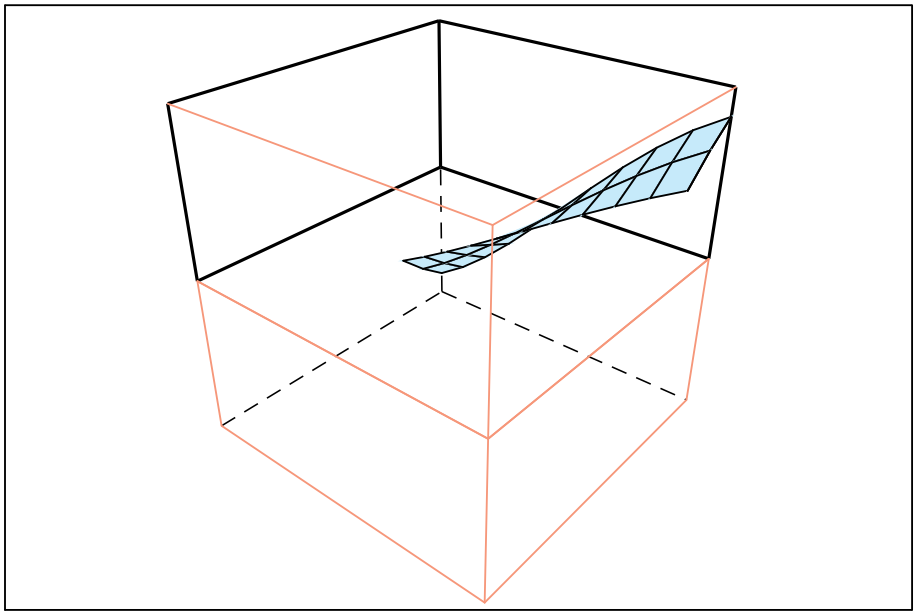


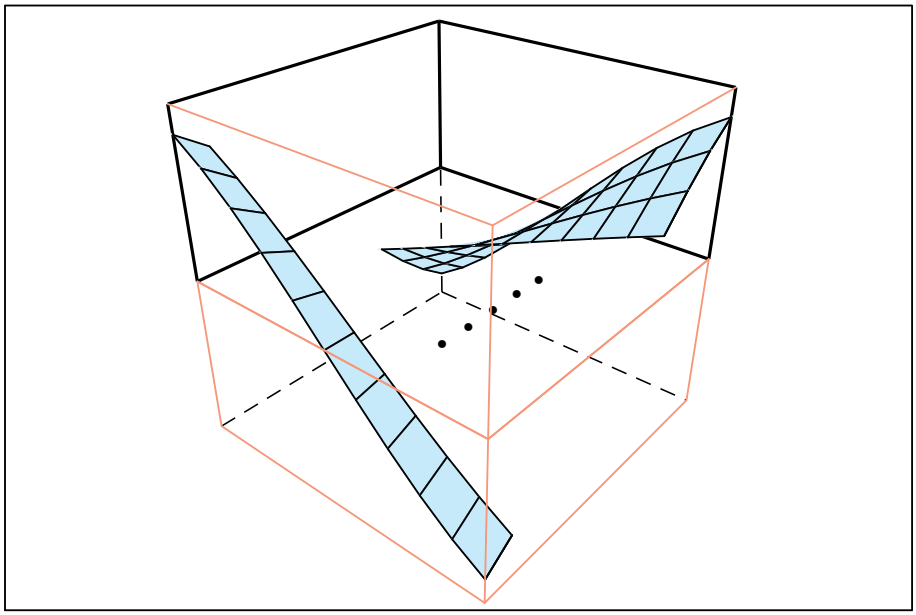
\DDFuncQuad berechnet die Bild-Vierecke in den Routinen

$$\texttt{\CurvePointAt}< t > \texttt{is}(u, v) \text{ und } \texttt{\UCurvePointAt}< t > \texttt{is}(u, v),$$

die ihrerseits wieder auf die bereitgestellten Befehle \CALCPOINTS und \UCALCPOINTS zurückgreifen.

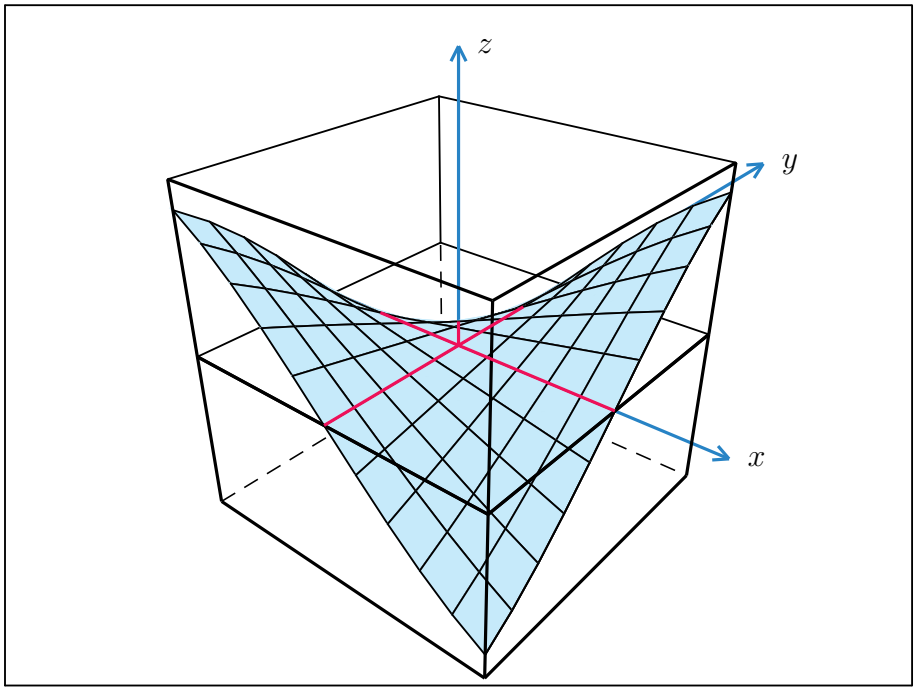
In den nächsten Schritten wird die vorletzte Reihe gezeichnet, dann die Reihe davor usw.



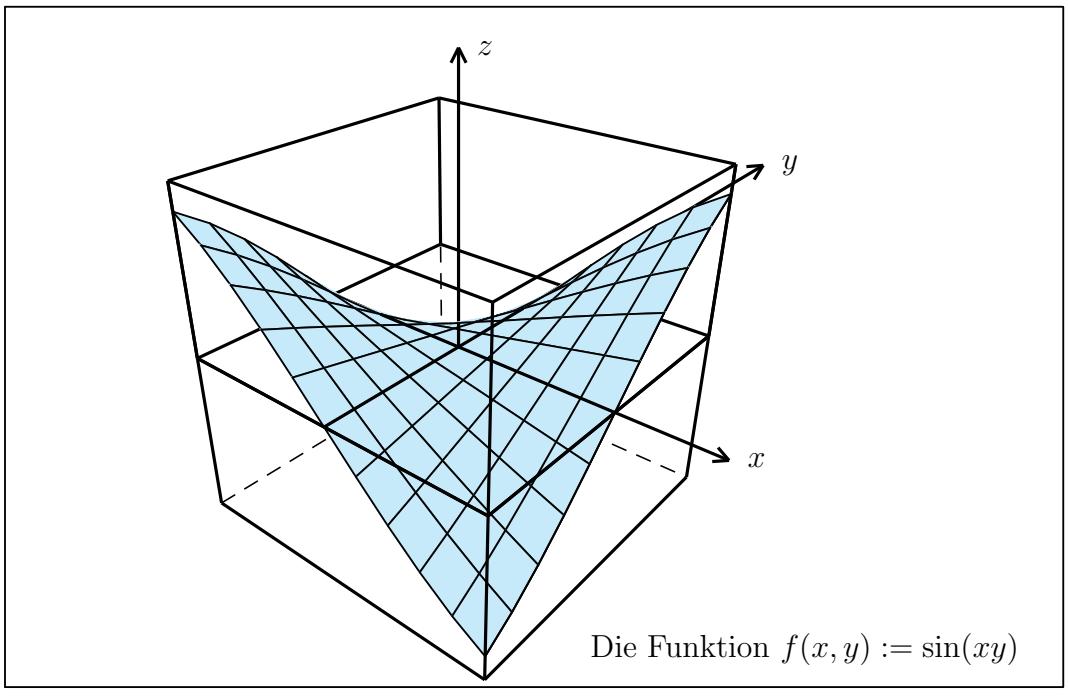


Zum Schluss wird die Vorderseite der Box gezeichnet, und dann versucht man, das Koordinatenkreuz einzupassen. Am besten zeichnet man es einmal, bevor der Graph eingefügt wird (in der folgenden Skizze blau gekennzeichnet), und am Schluss ergänzt man die fehlenden Teile (rot gekennzeichnet).

Man beachte, dass die Geraden $t \mapsto (t, 0, f(t, 0)) = (t, 0, 0)$ und $t \mapsto (0, t, f(0, t)) = (0, t, 0)$ auf dem Graphen liegen.



Alles zusammen ergibt nun das endgültige Bild:



Das verwendete Verfahren ist sehr umständlich, weil `\DDFunctionLine` zehnmal aufgerufen werden muss.

Ein effektiveres Verfahren benutzt die Routine `\DDXFunctionLine`$(-1,1)\{B_1\}\{B_2\}$, die direkt in einer Schleife aufgerufen wird:

```
\global\BIGEINH=\ONEPOINT%              <--  Y_1 = 1
\loop%                                  <--  Hier beginnt die Schleife
  \ifdim\BIGEINH>-0.8pt%                <--  Abbruch, wenn  Y_1 <
                                                   oder = -0.8 ist
  \bgroup
    \global\SMALLEINH=\BIGEINH
    \global\advance\SMALLEINH by -0.2pt% <-- Y_2 := Y_1 - 0.2
    \DDXFunctionLine(-1,1){%
      \TZWEI=\BIGEINH%
      \DDCALCPOINTS}{%                   <-- Berechne Werte fuer  Y = Y_1
      \TZWEI=\SMALLEINH
      \DDCALCPOINTS}%                    <-- Berechne Werte fuer  Y = Y_2
    \global\BIGEINH=\SMALLEINH%          <-- Y_1 = Y_2
  \egroup
  \relax
\repeat%                                 <-- Ende der Schleife
\relax
```

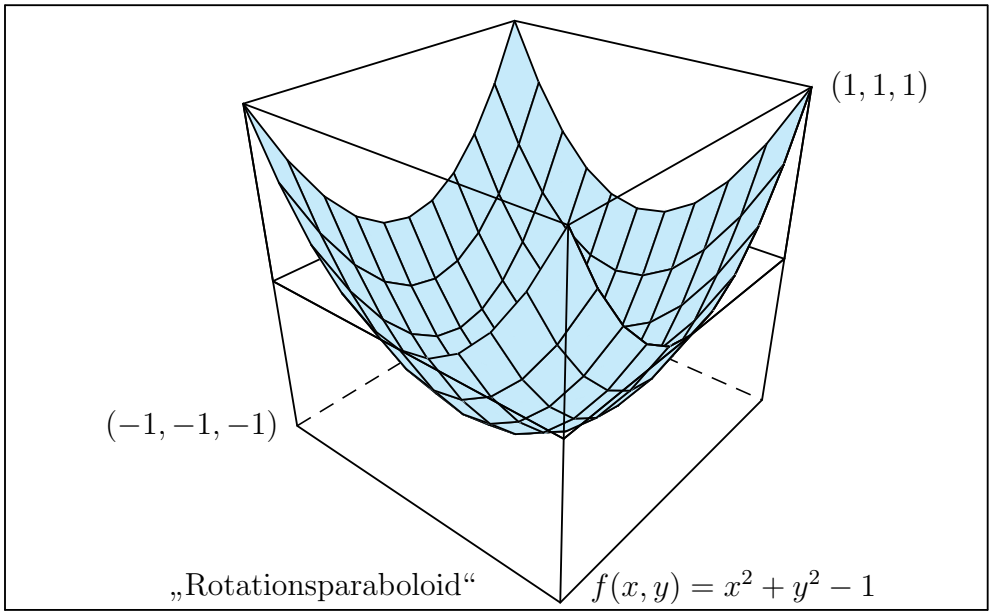
Wie man sieht, wird intern der Rechenbefehl \DDCALCPOINTS benutzt, dessen Inhalt einmal B_1 und einmal B_2 ist. Diese Inhalt werden wiederum von \CALCPOINTS und \UCALCPOINTS benutzt. Als Benutzer muss man nur zuvor \DDCALCPOINTS beschicken. Das geschieht folgendermaßen:

Bei festem y_0 wird dem Laufparameter t der Punkt $(x, y, z) = \big(t, y_0, f(t, y_0)\big)$ zugeordnet. Dabei steht t in der dimensionierten Variablen \TLAUF zur Verfügung, die Komponenten x, y, z müssen in (\xlauf ,\ylauf ,\zlauf) übergeben werden.

Im Falle der Funktion $(x, y) \mapsto f(x, y) := x^2 + y^2 - 1$ geschieht das z.B. durch

```
\def\DDCALCPOINTS{%
      \Wertvon<\TLAUF>nach<\xlauf>%          x := t
      \Wertvon<\TZWEI>nach<\ylauf>%          y wird in der Schleife
                                               uebergeben
      \XEINS=\xlauf\TLAUF%                   quadriere x
      \YEINS=\ylauf\TZWEI%                   quadriere y
      \advance\XEINS by \YEINS%
      \advance\XEINS by -\ONEPOINT%
      \Wertvon<\XEINS>nach<\zlauf>}%         z := x^2 + y^2 - 1
```

Dann erhält man folgendes Bild:

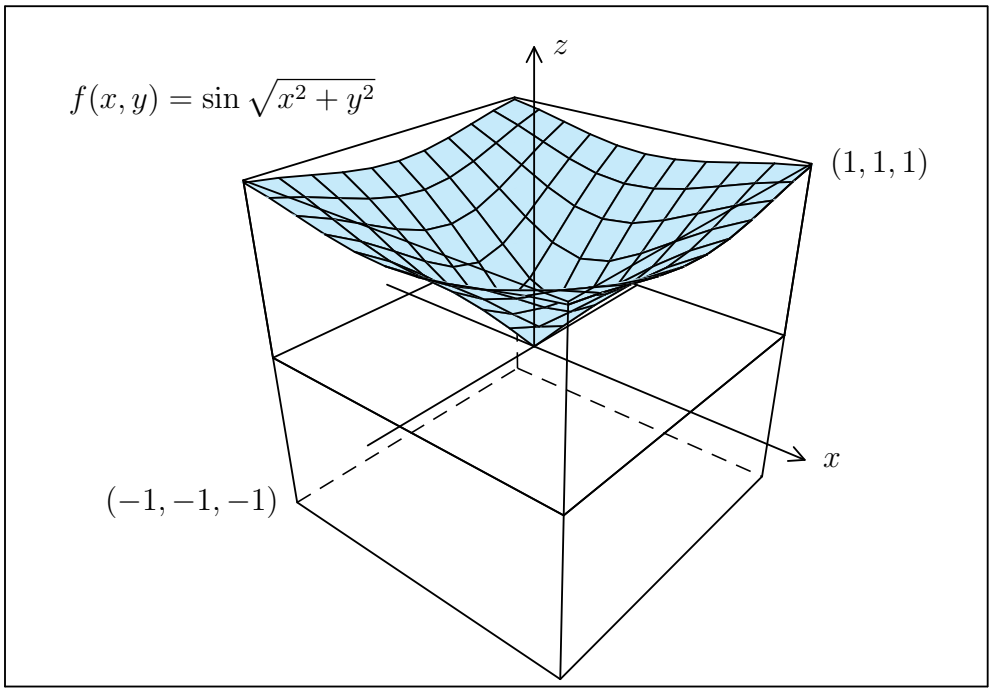


Das Listing hat folgende Gestalt:

```
\InitGraph{13}{8}{7}{4.5}{1cm}
\DrawBoundary
\Viewpoint(-50,60,5,6)[2]
%
% Die Rueckwaende der Box zeichnen
%
\def\DDCALCPOINTS{...} % s.o.
%
\global\BIGEINH=\ONEPOINT
\loop
  \ifdim\BIGEINH>-0.8pt
  \bgroup
     ... Inhalt der Hauptschleife (s.o.)
  \egroup
  \relax
\repeat
\relax
%
% Die Vorderwaende der Box zeichnen
%
\DDMoveTo(-1,-1,-1) \Text[l]{$(-1,-1,-1)$}
\DDMoveTo(1,1,1) \Text[r]{$(1,1,1)$}
\TextAt(-0.5,-4.2)[l]{"‘Rotationsparaboloid"’}
\TextAt(0.5,-4.2)[r]{$f(x,y)=x^2+y^2 - 1$}
\CloseGraph
```

Es folgen nun einige Beispiele von Graphen:

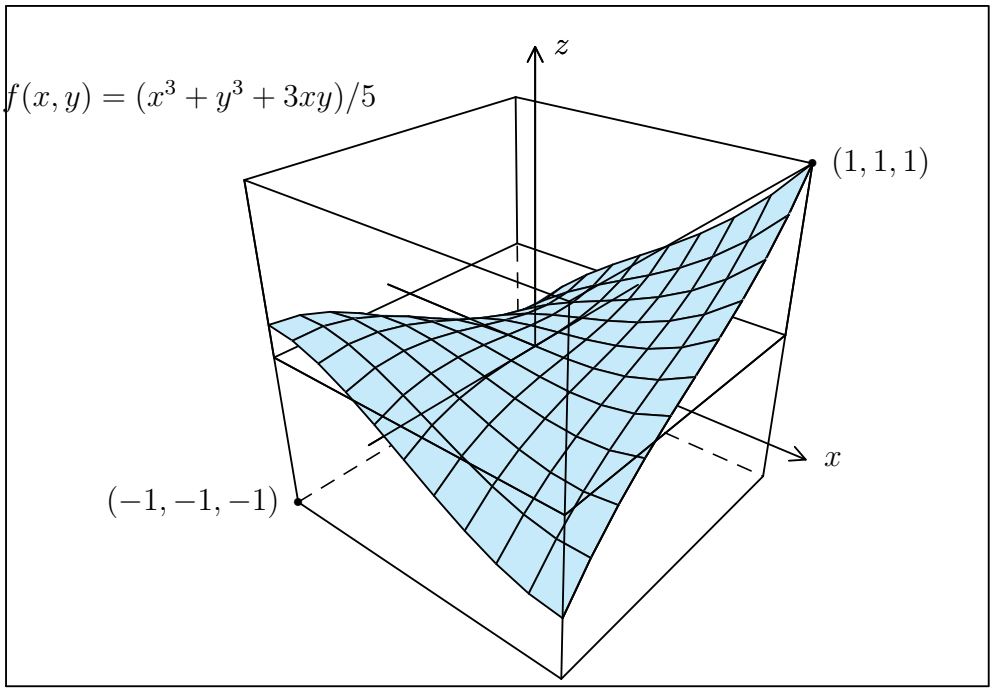
Die Funktion $f(x,y) = \sin\sqrt{x^2+y^2}$ wird durch mehrfachen Aufruf des Befehls `\DDFunctionLine`$(-1,1)\ldots$ dargestellt:

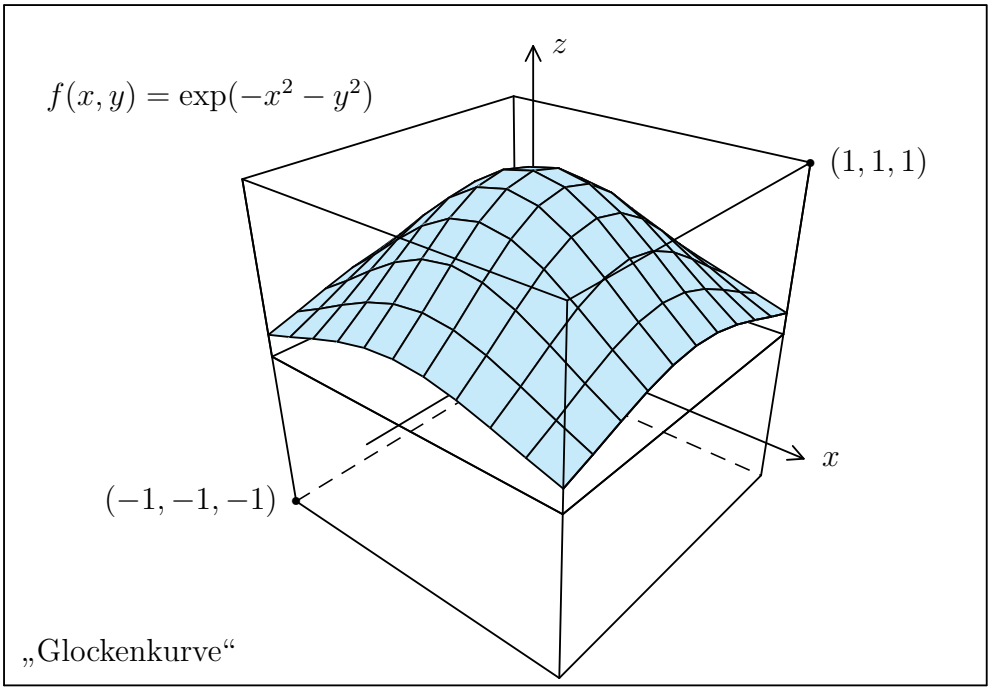



```
\DDFunctionLine(-1,1){%
\Wertvon<\TLAUF>nach<\xlauf>
\TZWEI=1pt%
\Wertvon<\TZWEI>nach<\ylauf>
\Hypotenusevon<\TLAUF>und<\TZWEI>nach<\XEINS>
\SINFKT<1,1,0,0>von<\XEINS>nach<\TEINS>
\Wertvon<\TEINS>nach<\zlauf>}{%
\Wertvon<\TLAUF>nach<\xulauf>
\TZWEI=0.8pt
\Wertvon<\TZWEI>nach<\yulauf>
\Hypotenusevon<\TLAUF>und<\TZWEI>nach<\XEINS>
\SINFKT<1,1,0,0>von<\XEINS>nach<\TEINS>
\Wertvon<\TEINS>nach<\zulauf>}
```

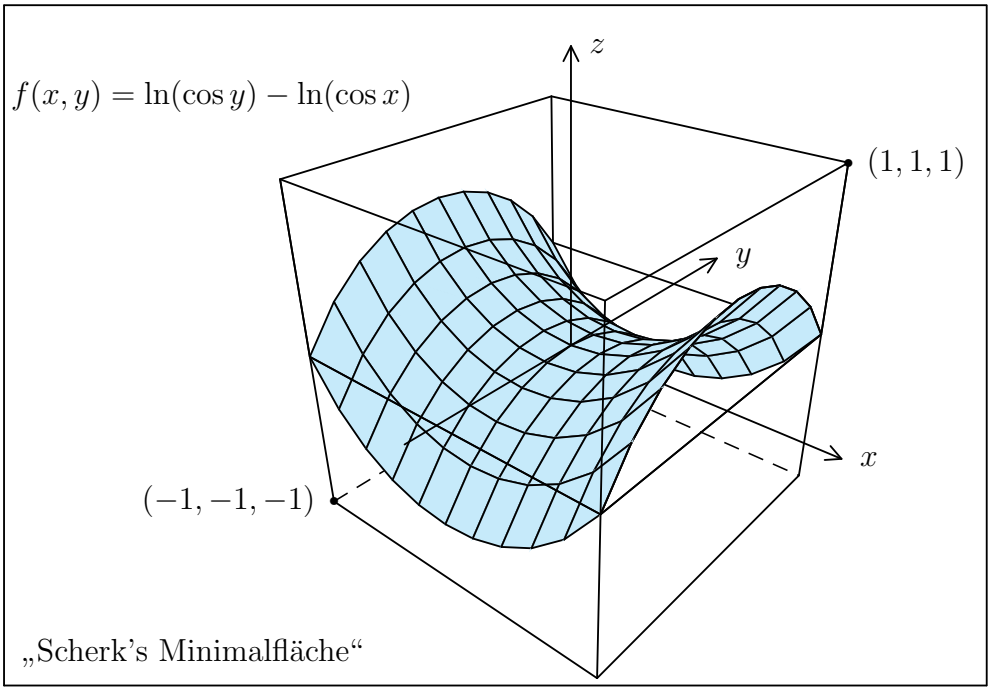
Dieses Listing erzeugt die hinterste Reihe von Kästchen.

Die folgenden Graphen wurden mit `\DDCALCPOINTS`... und einer Schleife erstellt. Die entscheidenden Teile des Listings finden sich am Ende.

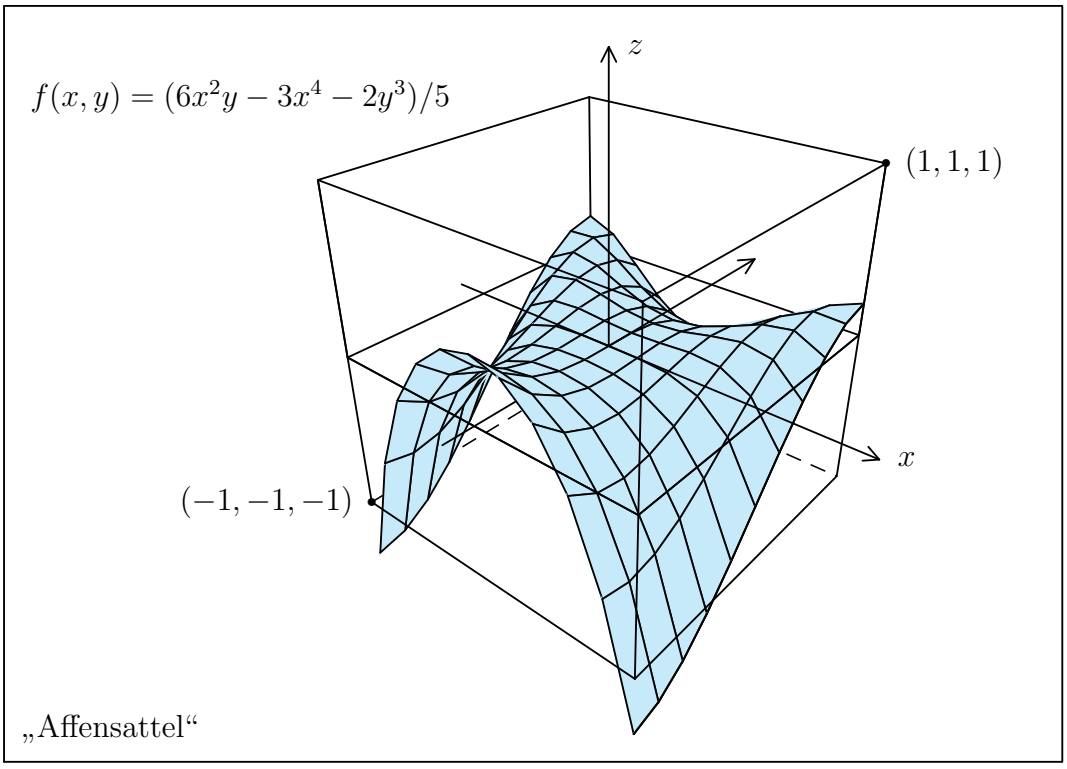
1. 

2. 

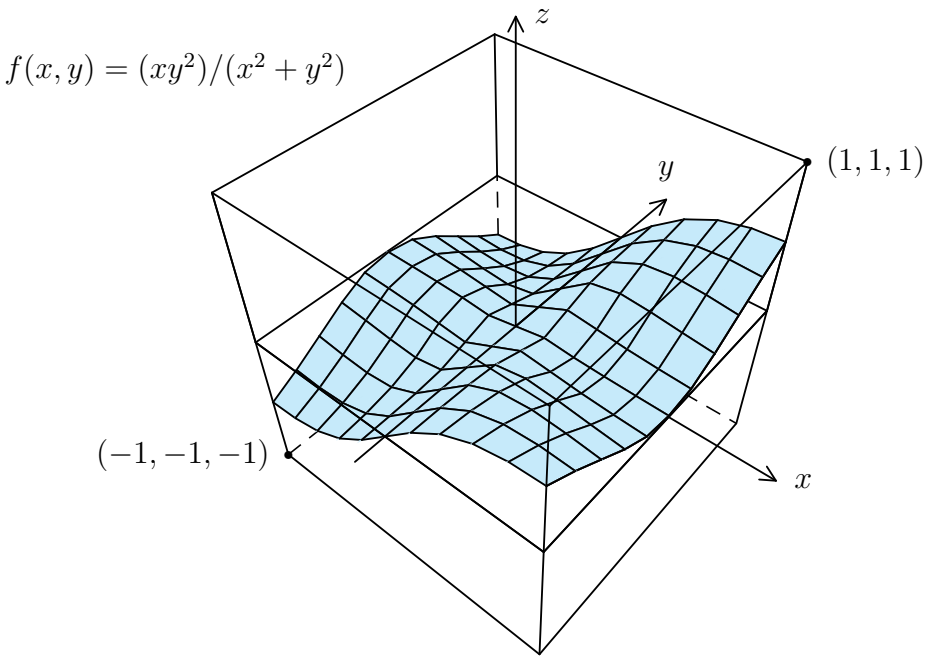
3.


$f(x,y) = \ln(\cos y) - \ln(\cos x)$
z
$(1,1,1)$
y
x
$(-1,-1,-1)$
„Scherk's Minimalfläche“

4.

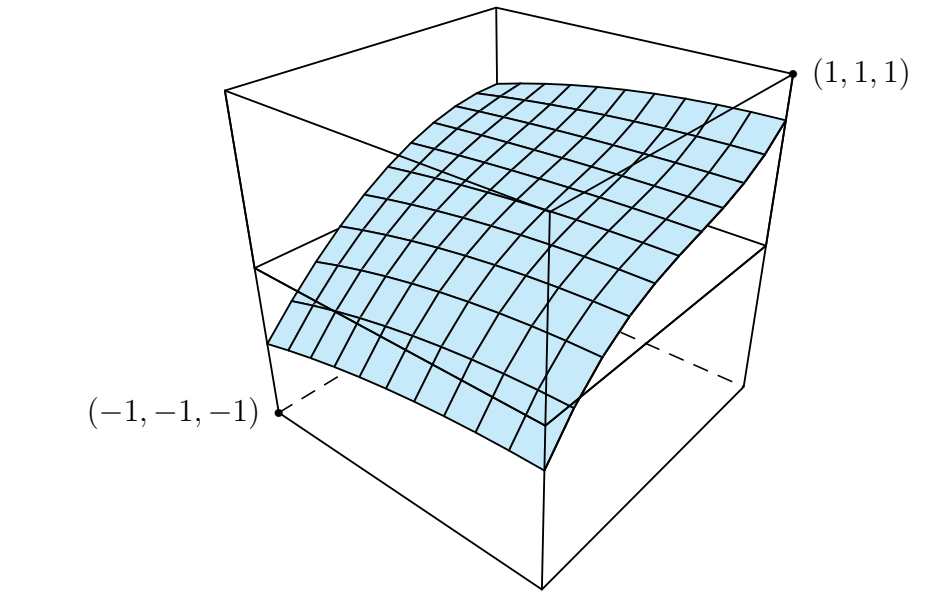

$f(x,y) = (6x^2y - 3x^4 - 2y^3)/5$
z
$(1,1,1)$
x
$(-1,-1,-1)$
„Affensattel“

5.



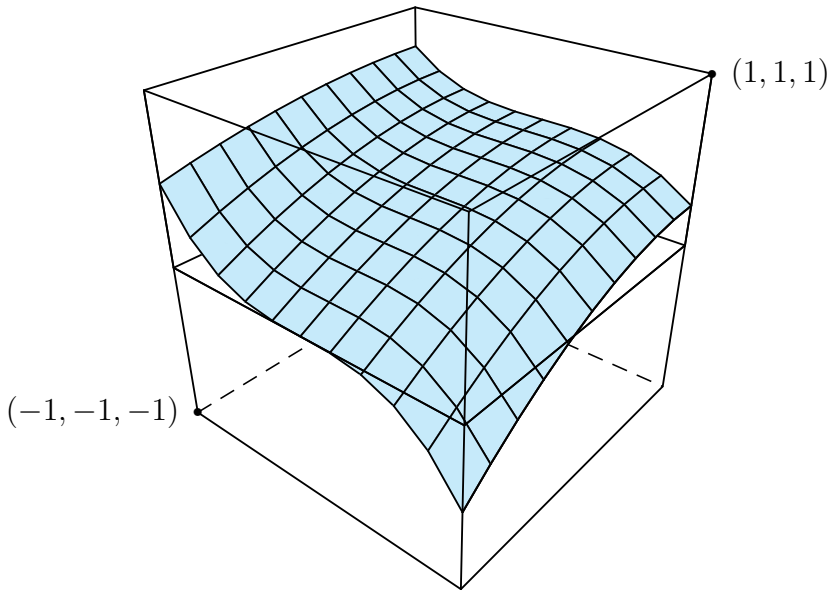
In $(0,0)$ nicht „total“ differenzierbar

6.



$f(x,y) = y^3 \cdot \big(1-(x-1)^2/4\big)/4 - (y-1)^2 \cdot \big((x-1)^2/4+1\big)/8 + 0.5$

$g_1(x) := -x^3/2$ und $g_2(x) := g_1(x)/2 + 1/4 = (1 - x^3)/4$.



7. $f(x, y) = g_1(x) \cdot \big(1 - h(y)\big) + g_2(x) \cdot h(y)$ mit $h(y) := 1 - (y - 1)^2/4$.

Listing zu (1):

```
\InitGraph{13}{9}{7}{4.5}{1cm}
\Viewpoint(-50,60,5,6)[2]
...
\def\DDCALCPOINTS{%
      \Wertvon<\TLAUF>nach<\xlauf>
      \Wertvon<\TZWEI>nach<\ylauf>
      \XEINS=\xlauf\TLAUF
      \XEINS=\xlauf\XEINS% x_1 = x^3
      \YEINS=\ylauf\TZWEI
      \YEINS=\ylauf\YEINS% y_1 = y^3
      \advance\XEINS by \YEINS% x_1 = x^3 + y^3
      \TEINS=\ylauf\TLAUF% t_1 = xy
      \advance\XEINS by 3\TEINS% x_1 = x^3 + y^3 + 3xy
      \XEINS=0.2\XEINS% x_1 = (x^3 + y^3 + 3xy)/5
      \Wertvon<\XEINS>nach<\zlauf>}
...
```

Listing zu (2, Glockenkurve):

```
\InitGraph{13}{9}{7}{4.5}{1cm}
\Viewpoint(-50,60,5,6)[2]
...
```

```
\def\DDCALCPOINTS{%
      \Wertvon<\TLAUF>nach<\xlauf>
      \Wertvon<\TZWEI>nach<\ylauf>
      \XEINS=\xlauf\TLAUF
      \advance\XEINS by \ylauf\TZWEI
      \EXPvon<-\XEINS>nach<\TEINS>
      \Wertvon<\TEINS>nach<\zlauf>}
...
```

Listing zu (3, Scherk's Minimalfläche):

```
\InitGraph{13}{9}{7.5}{4.5}{1cm}
\Viewpoint(-50,60,5,6)[2]
...
\def\DDCALCPOINTS{%
      \Wertvon<\TLAUF>nach<\xlauf>
      \Wertvon<\TZWEI>nach<\ylauf>
      \SINFKT<1,1,\PiHalbe,0>von<\TZWEI>nach<\YEINS>
      \LOGNATvon<\YEINS>nach<\XEINS>
      \SINFKT<1,1,\PiHalbe,0>von<\TLAUF>nach<\YEINS>
      \LOGNATvon<\YEINS>nach<\TEINS>
      \advance\XEINS by -\TEINS%
      \Wertvon<\XEINS>nach<\zlauf>}
...
```

Listing zu (4, Affensattel):

```
\InitGraph{14}{10}{8}{5.5}{1cm}
\Viewpoint(-50,60,5,6)[2]
...
\def\DDCALCPOINTS{%
      \Wertvon<\TLAUF>nach<\xlauf>
      \Wertvon<\TZWEI>nach<\ylauf>
      \XEINS=\xlauf\TLAUF
      \Wertvon<\XEINS>nach<\xquadrat>
      \XEINS=\xlauf\XEINS% x_1 = x^3
      \XEINS=\xlauf\XEINS% x_1 = x^4
      \XEINS=-3\XEINS
      \YEINS=\ylauf\TZWEI
      \YEINS=\ylauf\YEINS% y_1 = y^3
      \advance\XEINS by -2\YEINS% x_1 = -2y^3 - 3x^4
      \TEINS=\xquadrat\TZWEI% t_1 = yx^2
      \advance\XEINS by 6\TEINS% x_1 = 6yx^2 - 2y^3 - 3x^4
      \XEINS=0.2\XEINS
      \Wertvon<\XEINS>nach<\zlauf>}
...
```

Listing zu (5, nicht differenzierbare Funktion):

```
\InitGraph{14}{10}{8}{5.5}{1cm}
\Viewpoint(-50,45,5,6)[2]
...
\def\DDCALCPOINTS{%
      \Wertvon<\TLAUF>nach<\xlauf>
      \Wertvon<\TZWEI>nach<\ylauf>
      \Hypotenusevon<\TLAUF>und<\TZWEI>nach<\XEINS>
      \ifdim\XEINS<0.1pt
           \Wertvon<\ZEROPOINT>nach<\zlauf>
      \else
         \ifdim\TZWEI<\ZEROPOINT
              \TZWEI=-\TZWEI
         \fi\relax
         \ifdim\TZWEI<0.01pt
             \TEINS=10\TLAUF
         \else
             \Divide<\TLAUF>by<\TZWEI>forming<\TEINS>
         \fi\relax
         \Wertvon<\TEINS>nach<\teins>
         \TEINS=\teins\TEINS
         \advance\TEINS by \ONEPOINT
         \Divide<\TLAUF>by<\TEINS>forming<\XEINS>
         \Wertvon<\XEINS>nach<\zlauf>
      \fi\relax}
```

Listing zu (6):

```
\InitGraph{14}{10}{8}{5.5}{1cm}
\Viewpoint(-50,60,5,6)[2]
...
\def\DDCALCPOINTS{%
      \Wertvon<\TLAUF>nach<\xlauf>
      \Wertvon<\TZWEI>nach<\ylauf>
      \advance\TLAUF by -\ONEPOINT
      \Wertvon<\TLAUF>nach<\xeins>
      \XEINS=\xeins\TLAUF
      \XEINS=0.25\XEINS % x_1 = (x-1)^2/4
      \YEINS=\ylauf\TZWEI
      \YEINS=\ylauf\YEINS % y_1 = y^3
      \Wertvon<\YEINS>nach<\yeins>
      \advance\TZWEI by -\ONEPOINT
      \Wertvon<\TZWEI>nach<\yzwei>
      \YZWEI=\yzwei\TZWEI % y_2 = (y-1)^2
```

```
      \YZWEI=0.125\YZWEI % y_2 = (y-1)^2/8
      \Wertvon<\YZWEI>nach<\yzwei>
      \XZWEI=\ONEPOINT
      \advance\XZWEI by -\XEINS % x_2 = 1 - (x-1)^2/4
      \XZWEI=\yeins\XZWEI
      \XZWEI=0.25\XZWEI
      \advance\XEINS by \ONEPOINT
      \XEINS=\yzwei\XEINS
      \advance\XZWEI by -\XEINS
      \advance\XZWEI by 0.5pt
      \Wertvon<\XZWEI>nach<\zlauf>}
...
```

Listing zu (7):

```
\InitGraph{14}{10}{8}{5.5}{1cm}
\Viewpoint(-50,60,5,6)[2]
...
\def\DDCALCPOINTS{%
      \Wertvon<\TLAUF>nach<\xlauf>
      \Wertvon<\TZWEI>nach<\ylauf>
      \Wertvon<\TLAUF>nach<\xeins>
      \XEINS=\xeins\TLAUF
      \XEINS=\xeins\XEINS % x_1 = x^3
      \Wertvon<\XEINS>nach<\xeins>
      \XZWEI=\ONEPOINT
      \advance\XZWEI by -\xeins\ONEPOINT
      \XZWEI=0.25\XZWEI % x_2 = (1 - x^3)/4
      \advance\XZWEI by 0.25pt
      \Wertvon<\XZWEI>nach<\xzwei> % g_2(x) := (1 - x^3)/4 + 0.25
      \advance\TZWEI by -\ONEPOINT
      \Wertvon<\TZWEI>nach<\yzwei>
      \YZWEI=\yzwei\TZWEI % y_2 = (y-1)^2
      \YZWEI=0.25\YZWEI % y_2 = (y-1)^2/4
      \Wertvon<\YZWEI>nach<\yzwei>
      \YEINS=\ONEPOINT
      \advance\YEINS by -\YZWEI % y_1 = h(t) := 1 - (y-1)^2/4
      \YEINS=\xzwei\YEINS % y_1 = g_2(x) * h(y) =
            ((1-x^3)/4 + 0.25) * (1 - (y-1)^2/4)
      \XEINS=-0.5\XEINS % g_1(x) := -0.5 x^3
      \advance\YEINS by \yzwei\XEINS % addiere g_1(x) * (1-h(y))
      \Wertvon<\YEINS>nach<\zlauf>}
...
```