5.8 3D-Polygonflächen

5.8.1 Programm polyarea.cpp

Das Beispiel-Programm **polyarea.cpp** zeichnet Körper, die von Polygonflächen begrenzt werden. Die Strategie ist weitgehend gleich mit derjenigen, die für das "geordnete Zeichnen" von Stäben in den Programmen **stab1.cpp** und **stab2.cpp** verwendet wurde:

- Das Graphik-Modell wird mit **CGMod::rddfl_md** von einer Datei gelesen. Die "Elemente" sind allerdings keine mit zwei Punkten zu beschreibenden Stäbe, sondern mit mindestens 3 Punkten zu beschreibende Polygonflächen.
- In **CMainFrame::OnPaint** wird ein (in einem binären Baum sortiertes) "Objekt-Modell" aufgebaut. Die Objekte sind (im Gegensatz zu den Knoten und den Stäben in den Vorgänger-Programmen) nun 3D-Polygone, die als Instanzen der aus **CGObj** abgeleiteten Klasse **CGPoly** erzeugt werden. Gezeichnet wird (wie in den Vorgänger-Programmen) mit der Methode **CGObj::dromd_md**. Die Deklaration der Klasse **CGPoly** findet man in der Datei **cgiw.h**:

```
class CGPoly : public CGObj
{
     protected:
       int      m_npoin     ;
       double   *m_xyz_p    ;
       int      *m_points_p ;
       COLORREF m_linecol   ;
       COLORREF m_fillcol   ;
     public:
        CGPoly  (int npoin , double   *xyz_p  , int      *points_p ,
                             COLORREF fillcol , COLORREF linecol = 0) ;
        CGPoly  (int npoin = 0 , double *xyz_p    = NULL ,
                                 int    *points_p = NULL) ;
       ~CGPoly  (void) { }
        void drobj_md (CGI *cgi_p) ;
} ;
```

Verwaltet werden in der Klasse **CGPoly** die Anzahl der Polygon-Punkte, ein Pointer auf ein Koordinatenfeld ("World coordinates"), ein Pointer auf die Punktnummern (Punkte im Koordinatenfeld) und zwei COLORREF-Werte (Farben für den Rand bzw. die Füllung). Die in der Basisklasse **CGObj** rein virtuell deklarierte Funktion **drobj_md** wird natürlich für die abgeleitete Klasse definiert (File **cgpol_md.cpp**):

```
void CGPoly::drobj_md (CGI *cgi_p)
{
   CDC* pDC = cgi_p->getdc_gi () ;
   CPen  pen (PS_SOLID , 1 , m_linecol) ;
   CPen* penold_p = pDC->SelectObject (&pen) ;
   CBrush  brush (m_fillcol) ;
   CBrush* brushold_p = pDC->SelectObject (&brush) ;
   cgi_p->ptfpl_gi (m_npoin , m_xyz_p , m_points_p) ;
   pDC->SelectObject (penold_p)   ;
   pDC->SelectObject (brushold_p) ;
}
```

Die eigentliche Zeichenaktion wird von der Methode **CGI::ptfpl_gi** erledigt, die mit der Anzahl der Punkte und den beiden Pointern genau die Parameter erwartet, die in der Klasse **CGPoly** verwaltet werden. Die als "World coordinates" zu übergebenen Punkt-Koordinaten

werden einzeln der aktuellen Transformation unterworfen, auf die Zeichenebene projiziert und danach für das Zeichnen eines "gefüllten Polygons" wie mit der im Abschnitt 2.2.3 vorgestellten Methode **CGI::ufpol_gi** verwendet.

Man beachte, daß bei mehr als drei Punkten die 3D-Polygonfläche durchaus keine Ebene mehr beschreiben muß. Befriedigende Ansichten erhält man auf diesem Wege also nur, wenn entweder die 3D-Polygonflächen (wie die Seitenflächen eines Quaders) tatsächlich Ebenen sind oder aber tolerierbar im Rahmen der angestrebten Abbildung von Ebenen abweichen.

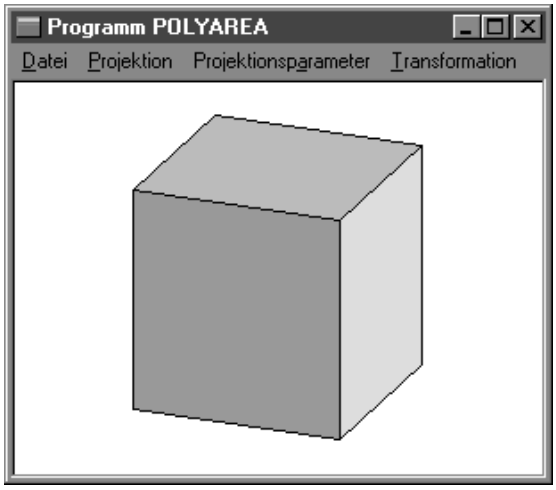
Die vom Beispiel-Programm **polyarea.cpp** darzustellenden Körper werden in Dateien beschrieben, die von **CGMod::rddfl_md** in den Vorgänger-Programmen bisher nicht benötigte Fähigkeiten ausnutzen. Zunächst wird zum Vergleich mit der im Abschnitt 5.2 vorgestellten Datei **d_wuerfl.dat** die nebenstehend zu sehende Datei **p_wuerfl.dat** vorgestellt, die einen durch Polygonflächen begrenzten Würfel beschreibt:

```
   6  8  4  3  1
-1.5  -1.5  -1.5
 1.5  -1.5  -1.5
 1.5   1.5  -1.5
-1.5   1.5  -1.5
-1.5  -1.5   1.5
 1.5  -1.5   1.5
 1.5   1.5   1.5
-1.5   1.5   1.5
 1 2 3 4   1
 1 4 8 5   2
 4 3 7 8   3
 1 2 6 5   4
 2 3 7 6   5
 5 6 7 8   6
```

Datei p_wuerfl.dat

- In der ersten Zeile stehen die "Anzahl der Elemente" (der Würfel hat 6 Flächen ...), die "Anzahl der Knoten" (... und 8 Ecken), die "Anzahl der zu einem Element gehörenden Knoten" (zu einer Fläche gehören 4 Ecken), ein Knoten wird durch 3 Koordinaten beschrieben, zusätzlich zu den 4 Knoten wird zu jeder Fläche eine Farbe angegeben.
- In den folgenden 8 Zeilen stehen jeweils 3 Koordinaten für die 8 Knoten. Durch die Reihenfolge der Zeilen wird eine Knotennumerierung von 1,...,8 festgelegt, auf die sich die nachfolgenden Daten beziehen.
- Es folgen die 6 Zeilen mit den Element-Beschreibungen (Polygon-Flächen). Hier werden jeweils 4 Knotennummern und ein Farbwert angegeben.

Das nebenstehend zu sehende Bild zeigt die Darstellung, die das Programm **polyarea.cpp** mit dieser Datei erzeugt. Die Farben für die einzelnen Flächen wurden der Datei entnommen, die Farbe für das Zeichnen der Kanten (Polygon-Ränder) wird im Programm einheitlich auf "Schwarz" gesetzt.

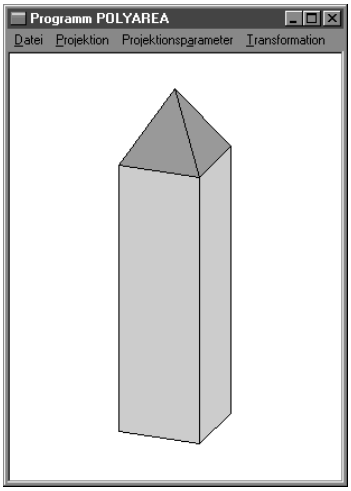


Programm polyarea.cpp, Datei p_wuerfl.dat

Es sind jedoch auch Modelle mit Polygonflächen möglich, die nicht alle die gleiche Punkt-Anzahl haben. Dies muß in der ersten Zeile der Datei dadurch angezeigt werden, daß auf der dritten Position (in **p_wuerfl.dat** steht dort die 4 für Vierecke) eine **0** (oder eine negative Zahl) steht. Dann wird in jeder "Element-Zeile" als erster Wert die Punkt-Anzahl für das Polygon erwartet.

Nebenstehend sind eine solche Datei und das damit erzeugte Bild zu sehen:

Der "Turm" ist durch 5 Vierecke und 4 Dreiecke begrenzt. Auf der dritten Position der ersten Zeile steht deshalb eine **0**, nach den 9 Koordinaten-Tripeln in den folgenden Zeilen folgen die 9 Zeilen für die Flächen, die jeweils die Anzahl der Polygon-Punkte (4 bzw. 3) als ersten Wert enthalten, es folgen die Knotennummern und schließlich ein Farbwert.



Programm polyarea.cpp, Datei p_turm.dat

```
 9  9  0  3   1
       0   0   0
       8   0   0
       0   8   0
       8   8   0
       0   0  24
       8   0  24
       0   8  24
       8   8  24
       4   4  30
 4     1 2 4 3    3
 4     1 2 6 5    1
 4     2 4 8 6    1
 4     3 4 8 7    1
 4     1 3 7 5    1
 3     5 6 9      4
 3     6 8 9      4
 3     7 8 9      4
 3     5 7 9      4
```

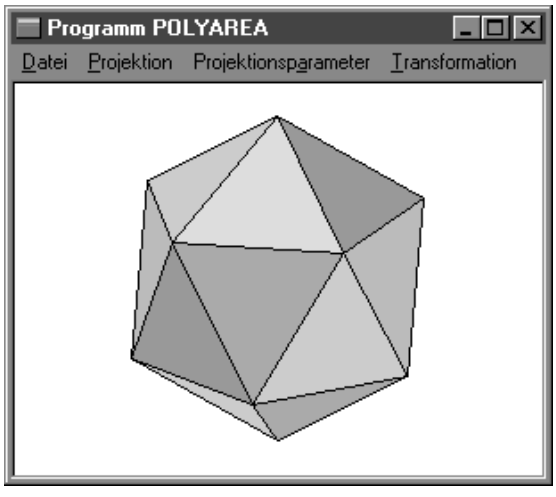
Datei p_turm.dat

Alle diese Werte werden beim Lesen der Datei mittels **CGMod::rddfl_md** in die **CGElem**-Instanzen übernommen. Sie können beim Aufbau des Objekt-Modells dort abgefragt werden. Dies geschieht in **CMainFrame::OnPaint**:

```
void CMainFrame::OnPaint ()
{
   CGPoly  *m_cgpoly_p ;
   double  xc , yc , zc , *coord_p ;
   int     i , n , *knr_p ;
   if (!m_cgmod_p) return ;                        // Kein Graphik-Modell
   // Es wird ein "Objekt-Modell" in einem binaeren Baum erzeugt:
   // Alle Graphik-Elemente (hier nur Polygone als Instanzen der Klasse
   // CGPoly) sind Instanzen von Klassen, die von der abstrakten Klasse
   // CGObj abgeleitet sind, sie werden geordnet in einem binaeren Baum
   // gespeichert:
   if (m_objroot_p) delete m_objroot_p ;           // Root-Pointer auf den
   m_objroot_p = NULL ;                            // binaeren Baum
   double *xyz_p  = m_cgmod_p->gtxyp_md () ;       // ... Pointer auf Koordi-
   CGElem *elem_p = m_cgmod_p->gtfel_md () ;       //     natenfeld und Pointer
                                                   //     auf erstes Element
   while (elem_p)
   {
      xc = 0. ;
      yc = 0. ;
      zc = 0. ;
      n     = elem_p->gtnop_md () ;               // Anzahl der Punkte
      knr_p = elem_p->gtpap_md () ;               // Pointer auf Punktnummern
      for (i = 0 ; i < n ; i++)
      {
         coord_p = xyz_p + (*(knr_p + i) - 1) * 3 ;
         xc += *(coord_p)     ;
         yc += *(coord_p + 1) ;
         zc += *(coord_p + 2) ;
      }
      xc /= n ;
      yc /= n ;                                   // Mittelwerte der Koordinaten
      zc /= n ;                                   // der Polgon-Punkte
```

```
      m_cgpoly_p = new CGPoly (n , xyz_p , knr_p , elem_p->gtcl1_md ()) ;
      // ... erzeugt eine Instanz der Klasse CGPoly und initialisiert sie
      //     mit den Werten, die dem Modell m_cgmod_p entnommen wurden.
      m_objroot_p = m_cgpoly_p->insot_md
                        (m_objroot_p , m_cpt.ptdis_pt (xc , yc , zc)) ;
      // ... fuegt die Instanz von CGPoly in den binaeren Baum ein, wobei
      //     die Einordnung von der Distanz vom Betrachter abhaengig gemacht
      //     wird, die mit aktueller Projektion und aktueller Transformation
      //     (m_cpt) von der CPT-Methode ptdis_pt berechnet wird (es werden
      //     jeweils die "Mittelpunkte" der Polygone als Bezugspunkte
      //     verwendet, was bei Modellen mit stark unterschiedlich grossen
      //     Polygonen durchaus einen "Ueberdeckungsfehler" verursachen
      //     kann).
      elem_p = m_cgmod_p->gtnel_md () ;         // ... naechstes Element
   }
   CPaintDC  dc (this) ;
   CGI       gi (this , &dc , &m_cpt) ;
   gi.stuci_gi (m_xumin , m_yumin , m_xumax , m_yumax , 10.) ;
   // Das Zeichnen eines "Objekt-Modells" ist besonders einfach, weil
   // die komplette Information im binaeren Baum gespeichert ist und
   // die Ableitung aller Klassen von der abstrakten Klasse CGObj mit
   // der dort nicht definierten virtuellen Zeichenfunktion drobj_md
   // dafuer sorgt, dass jeweils die zur Klasse der Instanz gehoerende
   // Methode drobj_md die Zeichenarbeit uebernimmt. Dies alles wird von
   // der CGObj-Methode dromd_md realisiert:
   if (m_objroot_p) m_objroot_p->dromd_md (&gi) ;
}
```

- In einer **while**-Schleife über alle Elemente (Polygone) der verketteten Liste werden zunächst mit den **inline**-Funktionen **CGElem::gtnop_md** die Anzahl der Punkte des Polygons und mit **CGElem::gtpap_md** der Pointer auf das Feld der Punktnummern besorgt.
- Mit den Punktnummern wird auf das zur **CGMod**-Instanz gehörende Koordinatenfeld zugegriffen. Es wird das arithmetische Mittel der Koordinaten eines Polygons berechnet.
- Es wird eine **CGPoly**-Instanz erzeugt, die mit der "Anzahl der Polygonpunkte", den Pointern auf das Koordinatenfeld und auf das Feld der Punktnummern und der mit der **inline**-Funktion **CGElem::gtcl1_md** besorgten Farbe für die Füllung der Polygone (wurde auch von der Datei gelesen) initialisiert. Der Konstruktor **CGPoly::CGPoly**, der dies erledigt, kann ein 5. Argument verarbeiten, setzt dafür in diesem Fall den Standardwert (Farbe "Schwarz" für den Polygonrand) ein.
- Schließlich wird die erzeugte **CGPoly**-Instanz in den binären Baum der zu zeichnenden Objekte geordnet einfügt (mit **CGObj::insot_md**).



Programm polyarea.cpp, Datei p_ikosa.dat

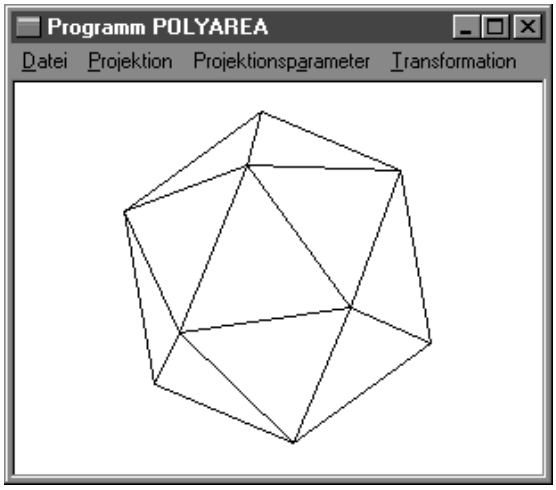
Als Kriterium für das Einfügen wird der "Abstand des Betrachters" vom berechneten "Polygon-Mittelpunkt" (genauer: Artihmetischer Mittelwert der Polygonpunkt-Koordinaten) verwendet, der wie in den Vorgänger-Programmen mit **CPT::ptdis_pt** unter Beachtung von Transformation und Projektion berechnet wird.

- Die eigentliche Zeichenaktion ist wieder besonders einfach zu programmieren, weil die rekursive Abarbeitung des binären Baums von **CGObj::dromd_md** (unter Vewendung von **CGPoly::drobj_md**) erledigt wird.

Die Abbildung auf der vorigen Seite zeigt das Bild, das vom Programm **polyarea.cpp** mit der Datei **p_ikosa.dat** erzeugt wird, ein Ikosaeder, das bereits in vorangegangenen Abschnitten als "Drahtmodell" und als "Stabmodell" gezeichnet wurde. Weil die 20 Dreiecksflächen in der "richtigen Reihenfolge" gezeichnet werden, sind nur die 10 sichtbaren auch tatsächlich zu sehen.

Die Flächen heben sich voneinander durch die unterschiedlichen Farben und die schwarz gezeichneten Ränder ab. Ohne diese beiden visuellen Hilfen würde nur der Eindruck eines ebenen Sechsecks bleiben (vgl. Abschnitt 5.5).

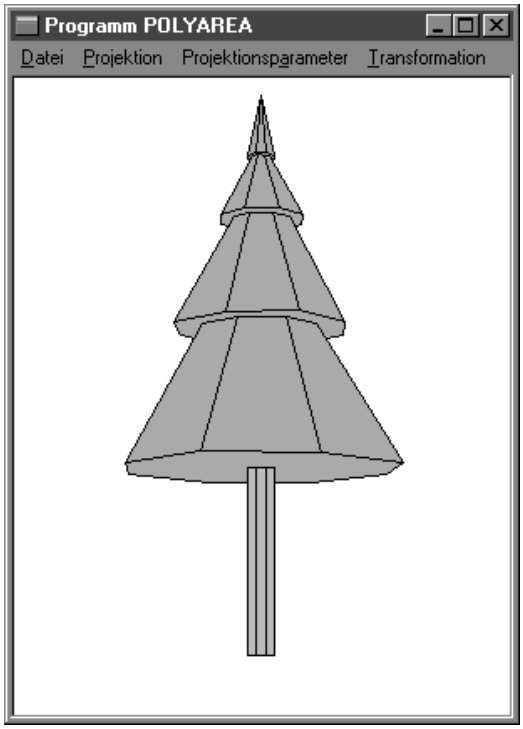
Die Strategie, mit "gefüllten Flächen" die unsichtbaren Flächen und Kanten zu überzeichnen, kann auch verwendet werden, um ein reines "Hidden line"-Kantenmodell zu erzeugen: Die Ränder der Flächen werden gezeichnet, und die Flächen werden mit der **Farbe des Bild-Hintergrunds** gefüllt. Die nebenstehende Abbildung zeigt ein so erzeugtes Bild des Ikosaeders. Alle Flächen werden (hier über die Information in der Datei **p_ikosal.dat**) mit der gleichen Farbe (Weiß) wie der Bild-Hintergrund gefüllt.



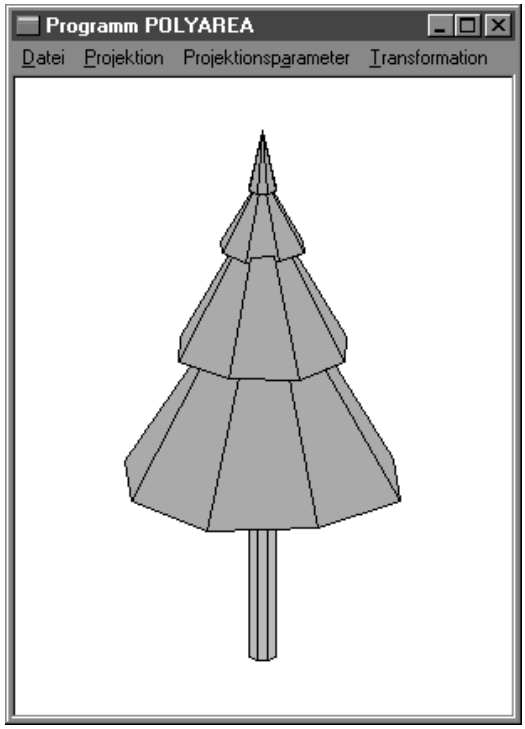
Ikosaeder: "Hidden line"-Kantenmodell

Der einfache Algorithmus des Programms **polyarea.cpp**, der die Flächen nach dem Abstand des durch die Mittelwerte der Eckpunkte festgelegten Punktes vom Betrachter sortiert, arbeitet für einen einfachen Körper wie ein Ikosaeder fehlerfrei. Bei auch nur etwas komplizierteren Flächen kann man sich nicht mehr darauf verlassen, daß auf diesem Weg alle Flächen in der Reihenfolge gezeichnet werden, daß sich immer die "gewünschte Überdeckung" ergibt. Generell gilt: Dieser einfache Algorithmus funktioniert nur, wenn die Gesamt-Oberfläche in genügend viele ausreichend kleine Teilflächen unterteilt wird. Eine entsprechende Aussage gilt auch für die "Stabmodelle", die mit dem Programm **stab2.c** (Abschnitt 5.7) dargestellt werden.

Die beiden Abbildungen auf der folgenden Seiten demonstrieren das. Das dargestellte Objekt (Datei **p_baum.dat**) wird durch 73 Dreiecke, Vierecke und Achtecke begrenzt. In fast allen Ansichten bei beliebigen Transformationen ist die Darstellung korrekt (z. B. wie im linken Bild). Für ganz wenige Transformationen schleicht sich ein Fehler ein (rechtes Bild, ein Viereck, das eigentlich unter einem anderen liegen müßte, verdeckt dieses), weil die Flächen zu groß sind. Wenn man alle Vierecke noch einmal teilt, ist die Darstellung immer korrekt.



Korrekte Darstellung



Darstellung mit einem kleinen Fehler

Die Frage, wie fein die Unterteilung der Flächen sein muß, kann nicht allgemein beantwortet werden. Die Antwort hängt schließlich auch davon ab, ob man eher einen Fehler tolerieren oder eher größere Antwortzeiten akzeptieren kann.

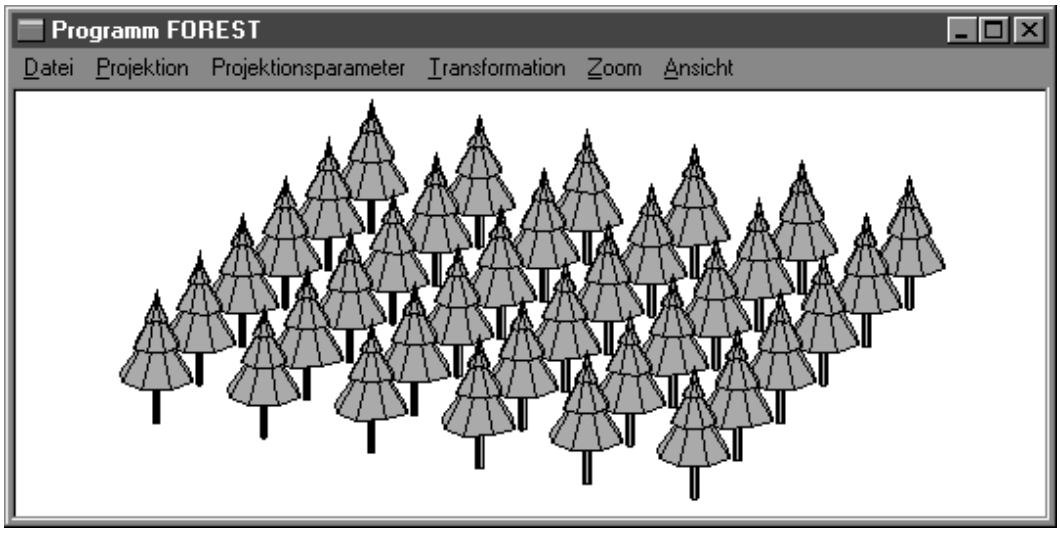
5.8.2 Programm forest.cpp

Das Programm **forest.cpp** arbeitet mit Modell-Dateien des gleichen Typs wie **polyarea.cpp**, ist jedoch in der Lage, das beschriebene Objekt mehrfach darzustellen, wobei jeweils eine andere Transformation angewendet wird, so daß jedes Objekt eine andere Lage im Raum einnimmt. Dieses Problem kann grundsätzlich auf zwei Wegen gelöst werden:

a) Es wird nur eine Modellbeschreibung verwendet, bei jeder neuen Darstellung des beschriebenen Objekts wird die zugehörige Transformation eingestellt. Dieser Weg sollte in jedem Fall beschritten werden, wenn keine korrekte Überdeckung erforderlich ist (wie z. B. bei den "Draht-Modellen" im Abschnitt 5.3).

b) Das Objekt wird vervielfältigt, so daß schließlich das Modell sämtliche darzustellenden Objekte enthält.

Im Programm **forest.cpp** wird ein Mittelweg beider Strategien gewählt. Weil ein korrekte Überdeckung der Flächen erreicht werden soll, müssen die darzustellenden Flächen sortiert werden, und jede darzustellende Fläche muß die Information enthalten, wo sie gezeichnet werden soll. Es müßten also entweder "Transformations-Informationen angehängt" werden, oder die Koordinaten werden (wie in **forest.cpp**) tatsächlich erzeugt.



Nach dem Start von forest.cpp sieht man eine "Baumschule" mit 36 Bäumen

Nach dem Programmstart zeigt sich das oben zu sehende Bild (Rücksichtnahme auf LARGE-Modelle von Windows 3.1 limitiert die "Baumschule" hier auf 36 Bäume, in 32-Bit-Versionen kann die Firma fast beliebig vergrößert werden). Dies wird durch die nachfolgend beschriebene Strategie erreicht. Nach dem Einlesen der Modell-Datei (wie in allen Vorgänger-Programmen wird das Modell von **CGMod::rddfl_md** gelesen und in einer **CGMod**-Instanz gespeichert) werden mit **CGBas::avsiz_gi** die extremen Abmessungen des Modells ("World coordinates") ermittelt:

```
int CMainFrame::ReadFile (char const *filename)
{
   if (m_cgmod_p) delete m_cgmod_p ;
   m_cgmod_p = new CGMod ;       // ... erzeugt ein "Graphik-Modell"-Objekt
   if (m_cgmod_p->rddfl_md (filename) == 0) return 0 ;
   // ... liest ein komplettes Graphik-Modell von einer Datei
   m_cgmod_p->avsiz_gi (m_cgmod_p->gtmnk_md () ,
                        m_cgmod_p->gtxyp_md () ,
                        &m_xwmin , &m_xwmax , &m_ywmin , &m_ywmax  ,
                                              &m_zwmin , &m_zwmax) ;
   m_opt = 1 ;
   return 1 ;
}
```

Die von **CGBas** geerbte Methode **CGMod::avsiz_gi** erwartet die Anzahl der Punkte des Koordinatenfeldes und den Pointer auf das Koordinatenfeld, in dem die Koordinaten der Punkte dicht gespeichert sind. Die abgelieferten Extremwerte der "World coordinates", die für sinnvolle Translations-Transformationen benötigt werden (die einzelnen Objekte sollen einander nicht durchdringen), werden in der **CMainFrame**-Instanz als m_xwmin, m_xwmax, m_ywmin, m_ywmax , m_zwmin und m_zwmax gespeichert.

Während das in der **CGMod**-Instanz gespeicherte Modell das mehrfach darzustellende Objekt nur einmal enthält, enthält das **CGObj**-Modell, das für das geordnete Zeichnen aller Polygonflächen aller Objekte benutzt wird, jede Fläche mit Bezug auf ihre tatsächlichen Koordinaten.

Dies wird in **CMainFrame::OnPaint** realisiert. Hier werden nur die Passagen gelistet, die von der entsprechenden Methode des Programms **polyarea.cpp** abweichen. Der Kernpunkt der Strategie besteht darin, eine neue Instanz der Klasse **CPT** zu erzeugen, die nacheinander die gewünschten 3D-Transformationen aufnimmt, um mit diesen Transformationen die Koordinaten aller darzustellenden Objekte zu erzeugen. Die beiden folgenden **CPT**-Methoden werden dafür verwendet:

- Die mit dem Prototyp

```
void t3abs_pt (double tx  = 0. , double ty = 0. , double tz = 0. ,
               double phi = 0. , int  axis = GI_AXISX ,
               double sx  = 1. , double sy = 1. , double sz = 1.) ;
```

  deklarierte **CPT**-Methode setzt eine 3D-Transformation absolut (ohne die vorherige Einstellung zu berücksichtigen). Es werden nacheinander (in dieser Reihenfolge!) die Skalierungen bezüglich des Nullpunktes (mit sx, sy und sz), eine Drehung um eine der drei Koordinatenachsen (phi ist der Drehwinkel im Bogenmaß, Daumen in positive Achsenrichtung, dann zeigen die Finger der Faust die positive Drehrichtung an, axis definiert die Achse, um die gedreht wird) und schließlich eine Translation (mit ty, ty und tz) ausgeführt.

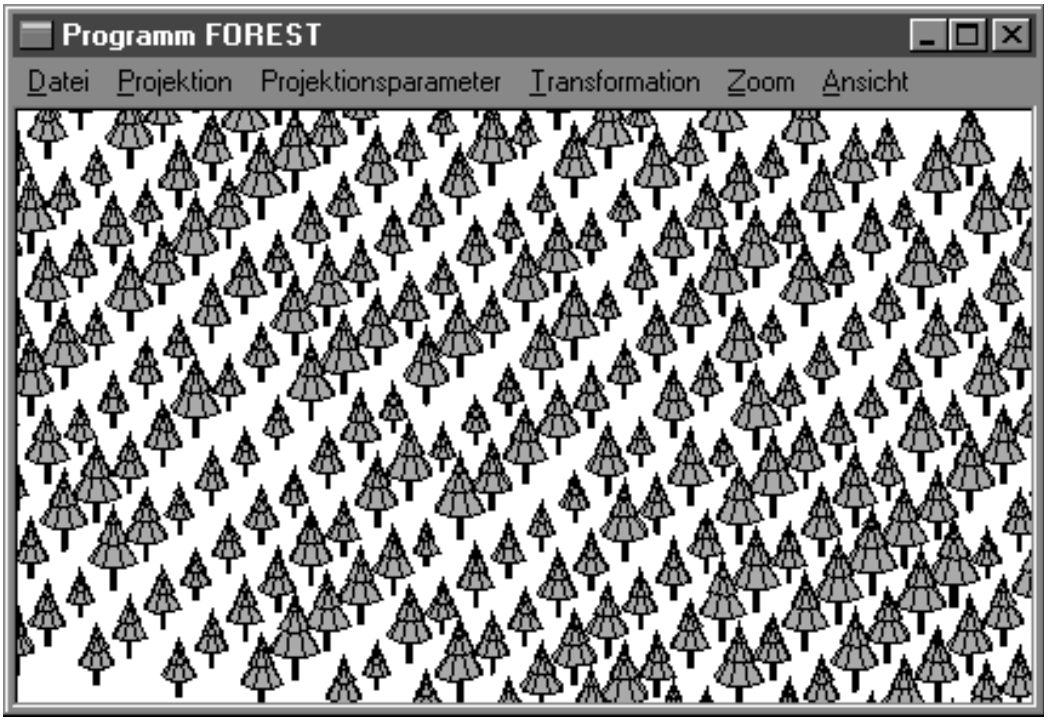
- Die mit dem Prototyp

```
void taw2w_pt (int np , double *xyzw_p , double *txyzw_p) ;
```

  deklarierte **CPT**-Methode **t**ransformiert ein **A**rray von "**W**orld coordinates" in ("to") "**W**orld coordinates". Es müssen np Koordinaten-Tripel (dicht gespeichert) ab xyzw_p angeliefert werden, die mit der aktuellen Transformation berechneten Koordinaten werden ab txyzw_p abgeliefert.

Man beachte, daß in der Methode CMainFrame::OnPaint diese Transformationen mit einer speziellen CPT-Instanz realisiert werden, so daß die Transformation für das Gesamtbild davon nicht beeinflußt wird. Folgende Parameter werden für das Einstellen der Transformationen genutzt:

- m_nx, m_ny, m_nz (Klasse **CMainFrame**) definieren die Anzahl der darzustellenden Objekte in x-, y- und z-Richtung (Startwerte sind m_nx=6, m_ny=6, m_nz=1, die Werte können über einen Dialog geändert werden).
- Mit den nach dem Einlesen der Datei ermittelten Grenzwerten werden die Abmessungen des Modells in den 3 Koordinatenrichtungen berechnet und mit dem Faktor m_dfac multipliziert (Startwert: 1.5, kann ebenfalls über einen Dialog geändert werden). Die so berechneten Abstände werden für die Translationen verwendet.
- Rotationen sind in **forest.cpp** nur um die z-Achse vorgesehen. Jedes darzustellende Objekt ist gegenüber seinem Vorgänger um m_phiz gedreht (Voreinstellung: m_phiz=0, kann ebenfalls geändert werden).
- Skalierungen sind nach dem Zufallsprinzip möglich: Mit der rand-Funktion werden Zufallszahlen erzeugt, mit denen Skalierungsfaktoren im Bereich m_scmin...m_scmax berechnet werden. Voreinstellungen sind m_scmin=1 und m_scmax=1, so daß zunächst nicht skaliert wird (deshalb die "Baumschule" nach dem Programmstart). Auch diese beiden Werte können natürlich über den Dialog geändert werden. Dann wird aus der "Baumschule" ein "Wald" (siehe Darstellung auf der folgenden Seite).



Zufalls-Skalierungen machen aus der "Baumschule" einen "Wald"

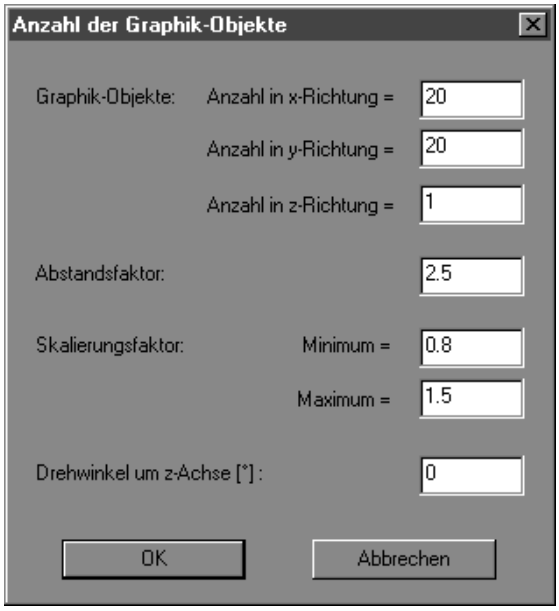
Nachfolgend wird die Passage aus **CMainFrame::OnPaint** gelistet, mit der alle Koordinaten erzeugt werden. Sie werden auf einem gesonderten Koordinatenfeld abgelegt, für das mit **new** Speicherplatz angefordert wird:

```
CPT cptw ;
double  dx = (m_xwmax - m_xwmin) * m_dfac ;     // "Abmessungen * Faktor"
double  dy = (m_ywmax - m_ywmin) * m_dfac ;     // fuer die Translationen
double  dz = (m_zwmax - m_zwmin) * m_dfac ;
int    count  = 0  ;
double phiz   = 0. ;
int    nk     = m_cgmod_p->gtmnk_md () ;        // Anzahl der Knoten
int    ngobjs = m_nx * m_ny * m_nz ;            // Darzustellende Objekte
double *xyz_p = new double [ngobjs * nk * 3] ;  // Speicherplatz fuer alle
for (int iz = 0 ; iz < m_nz ; iz++)             // Objekt-Koordinaten
{
   for (int iy = 0 ; iy < m_ny ; iy++)
   {
      for (int ix = 0 ; ix < m_nx ; ix++ , count += nk * 3)
      {
          scal = m_scmin + (m_scmax - m_scmin) *       // "Random"-
                 (double (rand ()) / RAND_MAX) ;       // Skalierungsfaktor
          cptw.t3abs_pt (dx * ix , dy * iy , dz * iz ,
                         phiz , cptw.GI_AXISZ , scal , scal , scal) ;
          cptw.taw2w_pt (nk , m_cgmod_p->gtxyp_md () , xyz_p + count) ;
          phiz += m_phiz ;                             // ... fuer Rotation
          if (phiz > GI_PI * 2.) phiz -= GI_PI * 2. ;
      }
   }
}
```

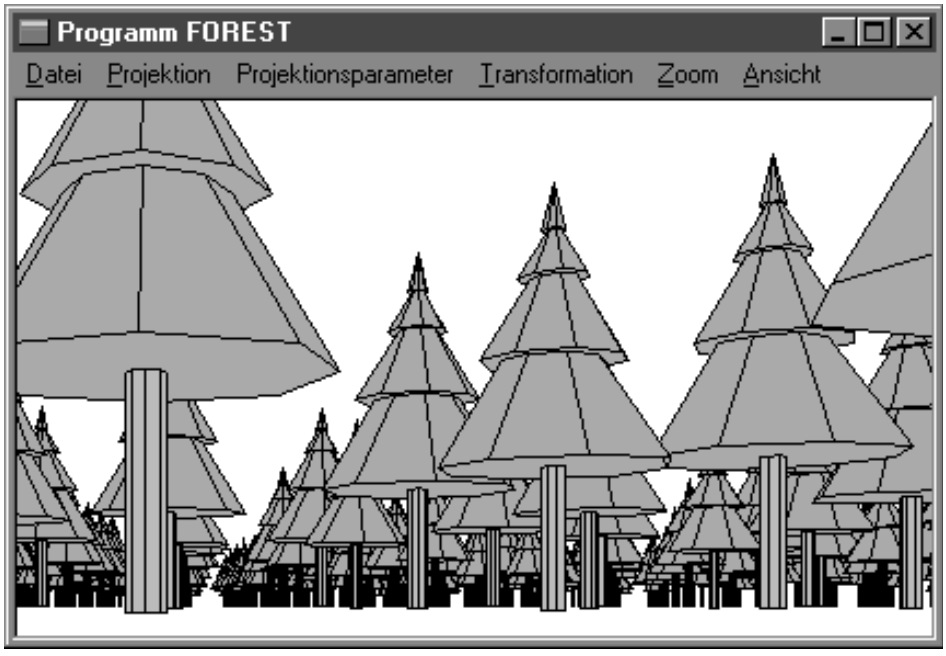
Der Rest unterscheidet sich nur unwesentlich vom Aufbau des geordneten binären Baums für die darzustellenden Polygonflächen im Programm **polyarea.cpp**. In die **while**-Schleife über alle Elemente des eingelesenen Modells wird eine **for**-Schleife eingebettet, die die Aktion für die ngobjs=m_nx*m_ny*m_nz Objekte ausführt:

```
while (elem_p)
{
   for (int io = 0 ; io < ngobjs ; io++)
   {
      xyzobj_p = xyz_p + nk * 3 * io ;   // Pointer auf Koordinaten
                                         // des Objekts io
            // ...
   }
   elem_p = m_cgmod->gtnel_md () ;
}
```

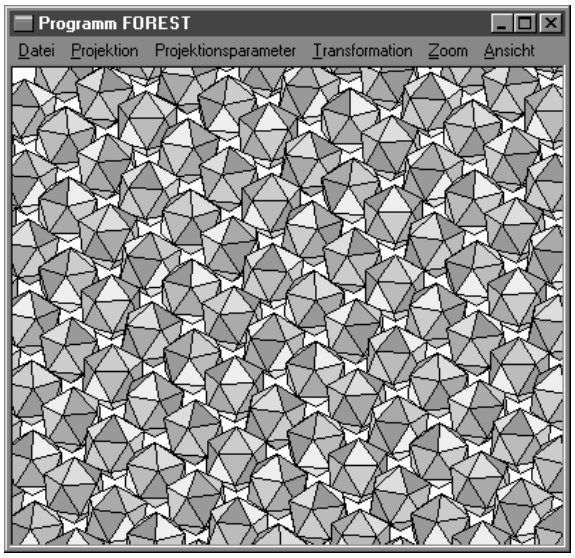
Die nebenstehende Abbildung zeigt die Dialog-Box, mit der die Parameter für die Transformationen geändert werden können. Eingestellt sind 20x20=400 "Bäume" mit einem "Abstandsfaktor" 2,5, die mit Skalierungsfaktoren im Bereich 0,8...1,5 skaliert werden sollen. Das Ergebnis ist der auf der vorigen Seite zu sehende "Wald", in den noch "hineingezoomt" wurde.



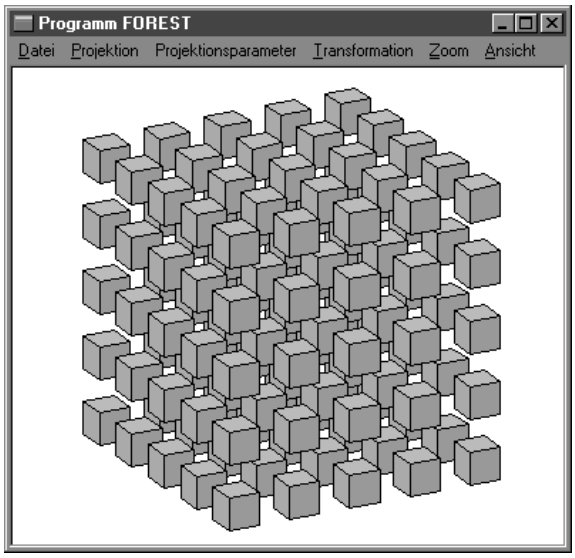
Da auch die Projektionsparameter geändert werden können, läßt sich mit der gleichen Parameter-Einstellung auch das unten rechts zu sehende Bild erzeugen, bei dem ein "Eye point" in Augenhöhe eines Spaziergängers vorgegeben wurde.



"Ich glaub', ich steh' im Wald"



"Ikosaeder-Teppich"



125 Würfel

Die beiden oben zu sehenden Bilder zeigen zwei mögliche Varianten, die in den "Wald-Bildern" bisher nicht zu sehen waren. Im linken Bild wurde auch die Rotations-Transformation um die z-Achse genutzt. Jedes Ikosaeder ist gegenüber seinem Nachbarn etwas verdreht, so daß es "schön bunt" wird. Im rechten Bild wurde auch die Vervielfältigung in z-Richtung genutzt.

5.9 Mathematisch beschriebene 3D-Flächen

Das Problem, ein darzustellendes 3D-Modell durch Daten zu beschreiben, kann für Flächen, die mit Formeln definiert werden können, dadurch gelöst werden, daß die Koordinaten der Punkte im Programm berechnet werden. Die beiden Programme, die in diesem Abschnitt beschrieben werden, behandeln diese Aufgabenstellung.

In der Datei **cgiw.h** wird eine Klasse **CArea3d** deklariert, die u. a. als Basisklasse für solche Flächen dienen kann:

```
class CArea3d : public CGBas // Klasse fuer die Darstellung einer
{                            // 3D-Flaeche
   private:
       double  m_xu1 ;       // Grenzen der beiden unabhaengigen Variablen
       double  m_yv1 ;       // (bei Flaechendefinition in der Darstellung
       double  m_xu2 ;       // z = z(x,y): x1,...,x2 und y1,...,y2, bei
       double  m_yv2 ;       // Parameterdarstellung u1,...,u2 und v1,...,v2)
       int     m_xusteps ;   // Anzahl der Polygone in den beiden
       int     m_yvsteps ;   // Koordinatenrichtungen
   public:
       CArea3d (double xu1     = -1. , double yv1     = -1. ,
                double xu2     =  1. , double yv2     =  1. ,
                int    xusteps = 10  , int    yvsteps = 10) ;
       virtual ~CArea3d  () { }
```

```
        double gtxu1_md () { return m_xu1 ; }
        double gtyv1_md () { return m_yv1 ; }
        double gtxu2_md () { return m_xu2 ; }
        double gtyv2_md () { return m_yv2 ; }
        double gtdxu_md () { return (m_xu2 - m_xu1) / m_xusteps ; }
        double gtdyv_md () { return (m_yv2 - m_yv1) / m_yvsteps ; }
        int    gtxus_md () { return m_xusteps ; }
        int    gtyvs_md () { return m_yvsteps ; }
        void   stint_md (double xu1  , double yv1   ,
                         double xu2  , double yv2)  ;
        void   ststp_md (int xusteps , int yvsteps) ;
        virtual int gtptc_md (double xu , double yv , double *xyz_p) = 0 ;
} ;
```

Die Klasse **CArea3d** ist eine abstrakte Klasse, die eingerichtet ist für die Verwaltung von Funktionen mit zwei unabhängigen Variablen. Durch die aus ihr abgeleiteten Klassen sollten Funktionen beschrieben werden, die in der Form

$$z = z(x,y)$$

oder in Parameterdarstellung

$$x = x(u,v) ;$$
$$y = y(u,v) ;$$
$$z = z(u,v)$$

gegeben sind. In **CArea3d** werden ein "Rechteckbereich" (zwei x,y-Wertepaare bzw. zwei u,v-Wertepaare) und zwei Diskretisierungsparameter (für x und y bzw. u und v) verwaltet, mit denen z. B. für die graphische Darstellung der Darstellungsbereich und die Feinheit der Unterteilung in Richtung der unabhängigen Variablen gesteuert werden können.

Die mit **gt...** beginnenden Methoden liefern die Werte der **private**-Daten, die mit **st...** beginnenden Methoden setzen diese Werte. Der Konstruktor setzt sämtliche Werte, gegebenenfalls alle mit Standard-Werten.

Die rein virtuell deklarierte Funktion **gtptc_md** übernimmt entweder eine x- und eine y-Koordinate oder einen u- und einen v-Parameter und liefert ab xyz_p die drei Koordinaten (x, y und z) des Punktes ab. Sie muß in den abgeleiteten Klassen definiert werden.

5.9.1 Programm mtharea1.cpp

Das Programm **mtharea1.cpp** demonstriert das Arbeiten mit einer aus **CArea3d** abgeleiteten Klasse. Es wird das "Parabolische Hyperboloid"

$$z = x \cdot y$$

dargestellt. Weil alle Standard-Vorgaben der Klasse **CArea3d** akzeptiert werden sollen, kann die abgeleitete Klasse in der einfachen Form

```
class CParHyp : public CArea3d
{
     public:
         virtual int gtptc_md (double xu , double yu , double *xyz_p) ;
} ;
```

deklariert werden. Das "Parabolische Hyperboloid" wird (Standard-Werte im Konstruktor von **CArea3d**) also über einem Quadrat x = -1 ... 1 und y = -1 ... 1 dargestellt, in jeder Richtung mit 10 Schritten, so daß 100 Vierecke gezeichnet werden.

Die Funktion **CParHyp::gtptc_md** liefert die Koordinaten eines Punktes:

```
int CParHyp::gtptc_md (double xu , double yv , double *xyz_p)
{
   *xyz_p       = xu ;       // Ab xyz_p werden die drei Koordinaten
   *(xyz_p + 1) = yv ;       // eines Punktes abgeliefert (hier fuer das
   *(xyz_p + 2) = xu * yv ;  // "Parabolische Hyperboloid"  z = x*y)
   return 1 ;
}
```

In der Klasse **CMainFrame** wird ein Pointer auf eine **CArea3d**-Instanz angesiedelt, im Konstruktor **CMainFrame::CMainFrame** wird das Objekt erzeugt:

```
CMainFrame::CMainFrame ()
{
   Create (NULL , "Programm MTHAREA1" , WS_OVERLAPPEDWINDOW ,
           rectDefault , NULL , MAKEINTRESOURCE (IDR_MAINFRAME)) ;
   m_rot   = 1    ;
   m_xyz_p = NULL ;
   m_carea_p = new CParHyp ;
   MakeCoords () ;
}
```

Die Koordinaten für die Eckpunkte der (in diesem Fall 100) Vierecke, die gezeichnet werden sollen, werden in **CMainFrame::MakeCoords** erzeugt und auf einem Feld m_xyz_p (Pointer gehört zur Klasse **CMainFrame**) abgelegt:

```
void CMainFrame::MakeCoords ()
{
   double  dxu , dyv , xu , yv , yv1 , *xyzp_p ;
   int     mxu , nyv , i  , j  , ij  ;
   dxu = m_carea_p->gtdxu_md () ;
   dyv = m_carea_p->gtdyv_md () ;
   mxu = m_carea_p->gtxus_md () ;
   nyv = m_carea_p->gtyvs_md () ;
   if (m_xyz_p) delete [] m_xyz_p     ;
   m_xyz_p = new double [mxu * nyv * 12] ;
   if (!m_xyz_p)
   {
      AfxMessageBox ("Nicht genuegend Speicherpaltz fuer Koordinaten") ;
      return ;
   }
   xu  = m_carea_p->gtxu1_md () ;
   yv1 = m_carea_p->gtyv1_md () ;
   for (i = 0 , ij = 0 ; i < mxu ; i++)        // Schleifenanweisungen ueber
   {                                           // mxu*nyv Polygone, die die
      for (j = 0 , yv = yv1 ; j < nyv ; j++)   // Flaeche approximieren
      {                                        // sollen
         xyzp_p = m_xyz_p + ij * 12 ;
         m_carea_p->gtptc_md (xu       , yv       , xyzp_p) ;
         m_carea_p->gtptc_md (xu + dxu , yv       , xyzp_p + 3) ;
         m_carea_p->gtptc_md (xu + dxu , yv + dyv , xyzp_p + 6) ;
         m_carea_p->gtptc_md (xu       , yv + dyv , xyzp_p + 9) ;
         // ... erzeugt die Koordinaten fuer die 4 Punkte eines Polgons
         yv += dyv ;
         ij++      ;
      }
      xu += dxu ;
   }
   m_cpt.ptmx3_pt (mxu * nyv * 4 , m_xyz_p ,
                   &m_xumin , &m_xumax , &m_yumin , &m_yumax)  ;
}
```

Mit dem Aufruf von **CPT::ptmx3_pt** am Ende von **CMainFrame::MakeCoords** werden die Extremwerte der Koordinaten ermittelt, die sich bei Anwendung der aktuellen Transformation und der eingestellten Projektion in der Projektionsebene ergeben. Sie werden in der **CMainFrame**-Instanz gespeichert.

Für die 100 Vierecke werden in **CMainFrame::OnPaint** 100 **CGPoly**-Instanzen erzeugt, die mit **CGObj::insot_md** in den binären Baum geordnet eingefügt und schließlich mit **CGObj::dromd_md** gezeichnet werden:

```
void CMainFrame::OnPaint ()
{
   CGObj   *objroot_p = NULL ;
   double  *xyzp_p , zmin , zmax ;
   CGPoly  *cgpoly_p ;
   double  xc , yc , zc ;
   int     i , n ;
   n = (m_carea_p->gtxus_md ()) * (m_carea_p->gtyvs_md ()) ;
   m_carea_p->avsiz_gi (n * 4 , m_xyz_p , NULL , NULL , NULL  , NULL ,
                                                    &zmin , &zmax) ;
   // ... als Vorbereitung fuer die Zuordnung "z-abhaengiger Farben"
   //     fuer die Polygone
   for (i = 0 ; i < n ; i++)
   {
      xyzp_p = m_xyz_p + i * 12 ;
      cgpoly_p->plcen_gi (&xc , &yc  , &zc , 4 , xyzp_p) ;
      cgpoly_p  = new CGPoly (4 , xyzp_p , NULL ,
                              cgpoly_p->cdcol_gi (zc , zmin , zmax)) ;
      // ... erzeugt ein Polygon, dem die mit cdcol_gi berechnete
      //     (hier "z-abhaengig gewaehlte") Farbe zugeodnet wird
      if (!cgpoly_p)
      {
         AfxMessageBox ("Nicht genuegend Speicherpaltz fuer Polygone") ;
         return ;
      }
      objroot_p = cgpoly_p->insot_md
                      (objroot_p , m_cpt.ptdis_pt (xc , yc , zc)) ;
   }
   CPaintDC  dc (this) ;
   CGI       gi (this , &dc , &m_cpt) ;
   gi.stuci_gi (m_xumin , m_yumin , m_xumax , m_yumax , 10.) ;
   objroot_p->dromd_md (&gi) ;
   delete  objroot_p ;
}
```

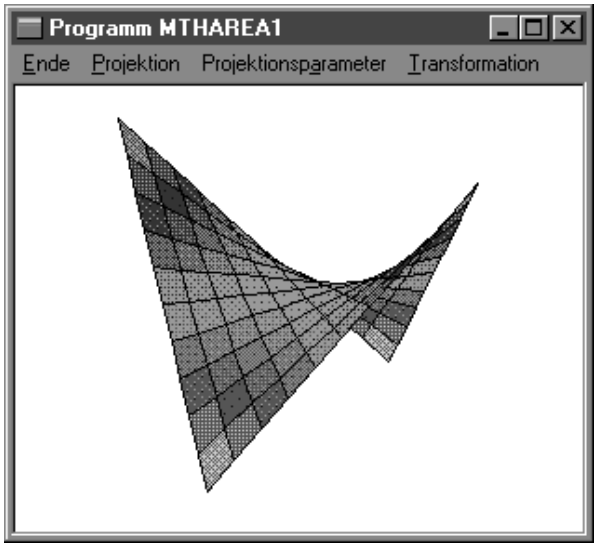
Auf folgende Funktionen, die in **CMainFrame::OnPaint** aufgerufen werden, soll noch aufmerksam gemacht werden:

- Die Methode **CGBas::avsiz_gi** übernimmt die Anzahl der Punkte und den Pointer auf ein Array, in dem 3D-Koordinaten gespeichert sind, und liefert die Extremwerte ("World coordinates") ab. Hier werden nur die Extremwerte der z-Koordinaten benötigt. In diesem Fall können auf den anderen Positionen NULL-Pointer übergeben werden.
- Für jedes zu zeichnende Viereck werden mit **CGBas::plcen_gi** die arithmetischen Mittelwerte der Koordinaten berechnet. Der so ermittelte "charakteristische Punkt" für das Viereck ("World coordinates") wird zweifach verwendet:

- Die z-Koordinate des "charakteristischen Punktes" wird für das "Einfärben" des Vierecks verwendet. Sie wird gemeinsam mit den Extremwerten aller z-Koordinaten an die Methode **CGBas::cdcol_gi** übergeben, die einen Farbwert so "zusammenmischt", daß von "reinem Grün" für die maximalen z-Werte bis zu "reinen Blau" für die minimalen z-Werte gleitende Übergänge realisiert werden. Dieses "Einfärben" ist willkürlich und könnte durch eine beliebige andere Strategie (z. B. nach dem Einfallswinkel des Lichts einer gedachten Lichtquelle) ersetzt werden. Die in **mtharea1.cpp** realisierte Variante liefert "schöne bunte Bilder".
- Der mit **CGBas::plcen_gi** berechnete Punkt wird auch für die Berechnung des "Abstandes vom Betrachter" (mit **CPT::ptdis_pt**) verwendet, das Ergebnis dient als Kriterium für das Einsortieren in den binären Baum, so daß die am weitesten entfernten Vierecke zuerst gezeichnet werden.

Die nebenstehende Abbildung zeigt das Ergebnis des Programms **mtharea1.cpp**. Das "Parabolische Hyperboloid" ist eine recht interessante Fläche. Sie ist doppelt gekrümmt mit Schnittflächen, die Parabeln oder Hyperbeln sind. Man kann sie trotzdem durch eine Bewegung einer Geraden im Raum erzeugen.

Durch Drehen der Fläche (im Programm realisiert ist die Änderung der Transformation) kann man von diesen Eigenschaften der Fläche einen sehr schönen Eindruck gewinnen.



Parabolisches Hyperboloid

5.9.2 Programm mtharea2.cpp

Das Programm **mtharea2.cpp** ist eine Erweiterung des Programms **mtharea1.cpp**, ergänzt um die bereits in verschiedenen anderen Programmen vorgestellte "Zoom"-Fähigkeit.

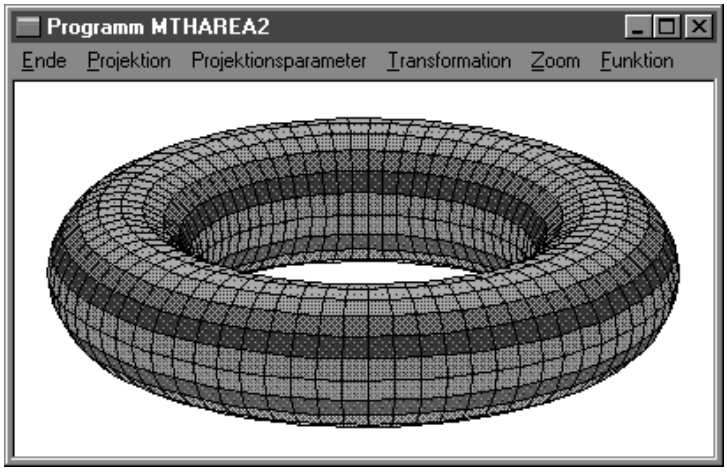
- Es können unterschiedliche (über das Menü-Angebot **Funktion** auszuwählende) Funktionen dargestellt werden, vornehmlich Funktionen, deren Definition nur in Parameter-Darstellung möglich ist.
- Die Feinheit der Diskretisierung (Anzahl der zu zeichnenden Vierecke) kann vom Programm-Benutzer geändert werden. Die Voreinstellung ist absichtlich recht grob gewählt worden, weil man (speziell beim Arbeiten mit Windows 3.1) durchaus Speicherplatzprobleme bekommen kann.

Beim Arbeiten mit Windows 3.1 ist das Memory-Modell **Large** zu empfehlen, aber auch das hat recht enge Grenzen. Das **Huge**-Modell würde einige Änderungen in einigen Funktionen (Anforderung von Speicherplatz vom Heap) erforderlich machen. Dies ist in der C-Version des **GIW**-Interfaces direkt vorgesehen, in der hier beschriebenen C++-Klassen-Bibliothek

wurde darauf verzichtet, weil sich die Probleme mit dem Sterben der "16-Bit-Welt" von allein lösen. Mit den gewählten Voreinstellungen des Programms kann man allerdings auch in der "16-Bit-Welt" alle Funktionen betrachten.

Übrigens kann nicht nur der Heap eine Grenze setzen, auch der Stack wird unter Umständen durch die sehr tiefe Rekursion beim Abarbeiten des binären Baumes stark belastet. Allerdings enthält die für das Zeichnen verwendete Methode **CGObj::drrec_md** (zu sehen in der Datei **dromd_md.cpp**) keine lokalen Variablen, so daß auf dem Stack nur jeweils (neben den Organisations-Informationen) ein Pointer pro Aufruf abgelegt wird.

Die gegenüber dem Programm **mtharea1.cpp** erweiterte und geänderte Strategie wird am Beispiel der Realisierung der Torus-Dartellung (nebenstehende Abbildung) erläutert. Wenn man einen in der x-z-Ebene liegenden Kreis mit dem Radius ***r***, dessen Mittelpunkt von der z-Achse um ***R*** entfernt ist, um die z-Achse rotieren läßt, entsteht ein Torus, der mathematisch wie folgt beschrieben werden kann:



Torus, approximiert durch 1500 Vierecke

$$x = (R + r\cos u)\cos v\ ;$$
$$y = (R + r\cos u)\sin v\ ;$$
$$z = r\sin u\ .$$

In der Abbildung ist der Torus für die Parameter-Bereiche $u = 0 \ldots 2\pi$ und $v = 0 \ldots 2\pi$ zu sehen (Voreinstellung im Programm).

Es wird eine Klasse **CTorus** (in der Datei **mtharea2.h**) deklariert, die aus der Basisklasse **CArea3d** abgeleitet wird:

```
class CTorus : public CArea3d
{
   private:
       double  m_R ;
       double  m_r ;
   public:
       CTorus (double R   = 4.         , double r   = 1.2        ,
               double xu1 = 0.         , double yv1 = 0.         ,
               double xu2 = GI_PI * 2. , double yv2 = GI_PI * 2. ,
               int    xusteps = 15     , int    yvsteps = 30) :
        CArea3d (xu1 , yv1 , xu2 , yv2 , xusteps , yvsteps)
       {
          m_R = R ;
          m_r = r ;
       }
       virtual int gtptc_md (double xu , double yv , double *xyz_p) ;
} ;
```

Der Konstruktor der Klasse erwartet die Werte für ***R*** und ***r***, die in der Klasse **CTorus** verwaltet werden, und die Grenzen für die Parameter ***u*** und ***v*** und die "Diskretisierungs-Parameter", die sämtlich an den Konstruktor der Basisklasse **CArea3d** weitergereicht werden,

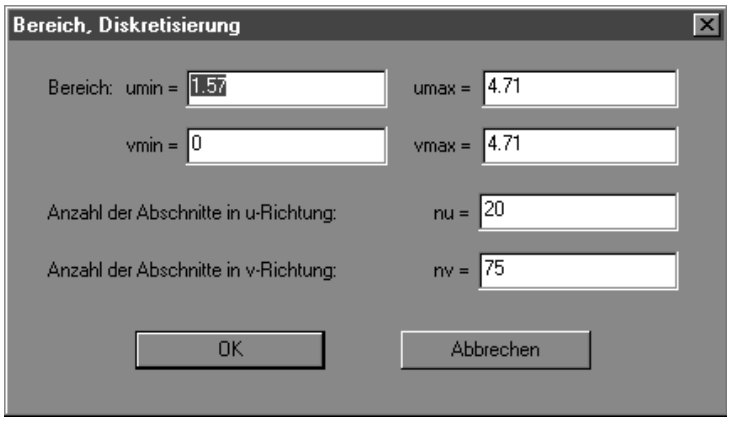
von der **CTorus** sie geerbt hat. Alle Parameter, die **CTorus::CTorus** erwartet, sind mit sinnvollen Werten vorbelegt.

Die in **CArea3d** rein virtuell deklarierte Funktion **gtptc_md** wird für **CTorus** definiert und realisiert das Berechnen eines Punktes aus einem u-v-Wertepaar:

```
int CTorus::gtptc_md (double xu , double yv , double *xyz_p)
{
   *xyz_p       = (m_R + m_r * cos (xu)) * cos (yv) ;
   *(xyz_p + 1) = (m_R + m_r * cos (xu)) * sin (yv) ;
   *(xyz_p + 2) = m_r * sin (xu) ;
   return 1 ;
}
```

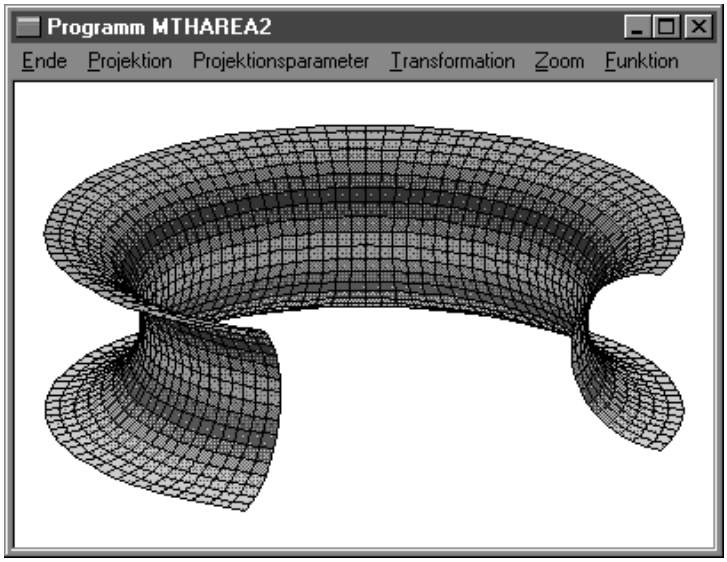
Ansonsten läuft alles ähnlich ab wie in **mtharea1.cpp**: Nach Wahl des Menü-Angebots **Torus** wird (hier vom Empfänger der Menü-Botschaft **CMainFrame::FuvArea**) eine **CTorus**-Instanz (mit den Standard-Werten des Konstruktors) erzeugt, und die Koordinaten werden in **CMainFrame::MakeCoords** berechnet. Dies geschieht wie auch das Zeichnen in **CMainFrame::OnPaint** nach der gleichen Strategie wie in **mtharea1.cpp**.

Bei der ersten Wahl des Angebots **Torus** im Popup-Menü **Funktion** wird der Torus mit derStandard-Einstellung gezeichnet, die der Konstruktor der Klasse **CTorus** vorgegeben hat. Wird dieses Menü-Angebot ein weiteres Mal gewählt (der Torus ist noch zu sehen), öffnet sich zunächst die nebenstehend zu sehende Dialog-Box.

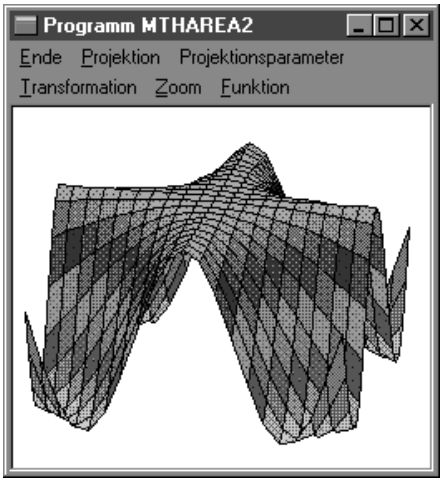


Hier kann man alle Werte ändern, die **CTorus** von **CArea3d** geerbt hat. In der dargestellten Dialog-Box sind fast alle Werte geändert worden.

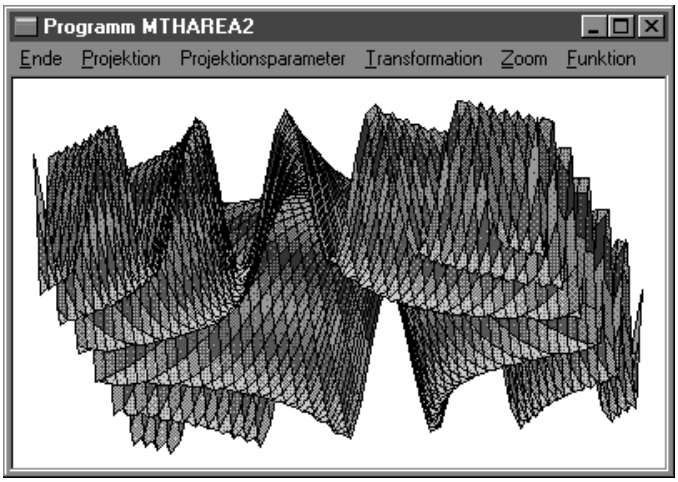
Wenn man die mathematische Darstellung der Torus-Funktion analysiert, stellt man fest, daß der Parameter ***u*** eine Winkelkoordinate ist, die einen Punkt auf dem kleinen Kreis verfolgt, ***v*** ist die Winkelkoordinate für die Rotation um die z-Achse (beide Koordinaten werden im Bogenmaß gemessen). Deshalb ergibt sich mit den gewählten Werten, die in der Dialog-Box zu sehen sind, das nebenstehnde Bild.



Torus-"Torso"



z = cos (2 x y) mit Stadard-Werten...



... und ein größerer Bereich in der Ansicht von unten

Nachfolgend werden die weiteren darstellbaren Flächen mit den mathematischen Gleichungen zusammengestellt, mit denen die Punkte berechnet werden. Neben dem "Parabolischen Hyperboloid", das bereits im Programm **mtharea1.cpp** zu sehen war, kann man eine weitere Funktion darstellen, die in der Form $z=z(x,y)$ definiert ist. Es ist die in den beiden Bildern oben zu sehende Fläche

$$z = a \cos(b x y) \ .$$

Für die Parameter ***a*** und ***b*** werden vom Konstruktor der Klasse **CCosxy** die Werte

$$\boldsymbol{a = 1 \ ,}$$
$$\boldsymbol{b = 2}$$

festgelegt.

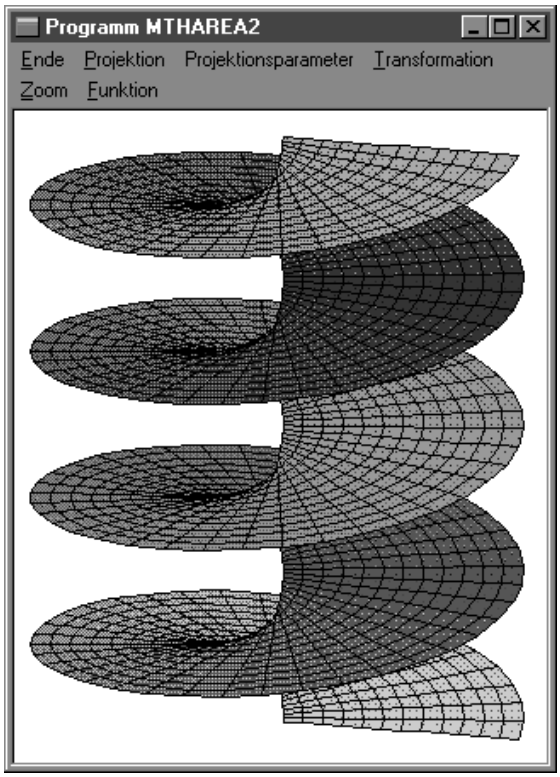
Die übrigen Funktionen werden alle durch Parameter-Dartstellungen beschrieben. Die nebenstehend abgebildete Funktion ist eine sogenannte "Helix", die durch die schraubenförmige Rotation eines Geradenstücks um die z-Achse entsteht und mathematischen folgendermaßen beschrieben wird:

$$x = u \cos(v) \ ;$$
$$y = u \sin(v) \ ;$$
$$z = k v \qquad .$$

Vom Konstruktor der Klasse **CHelix** wird für das "Steigungsmaß" ***k*** der Wert

$$\boldsymbol{k = 0{,}4}$$

festgelegt.



Helix

Eine Helix kann in verschiedenen Varianten definiert werden, abhängig von der Linie, die als "Erzeugende" gewählt wird. Die nebenstehende Abbildung zeigt eine "Kreis-Helix" (ein Kreis rotiert schraubenförmig um die z-Achse), die mathematisch folgendermaßen beschrieben werden kann:

$$x = (R + r\cos u)\cos v \;;$$
$$y = (R + r\cos u)\sin v \;;$$
$$z = (R + r\sin u) + k\,v \;.$$

Die Parameter werden vom Konstruktor der Klasse **CKreishelix** wie folgt festgelegt:

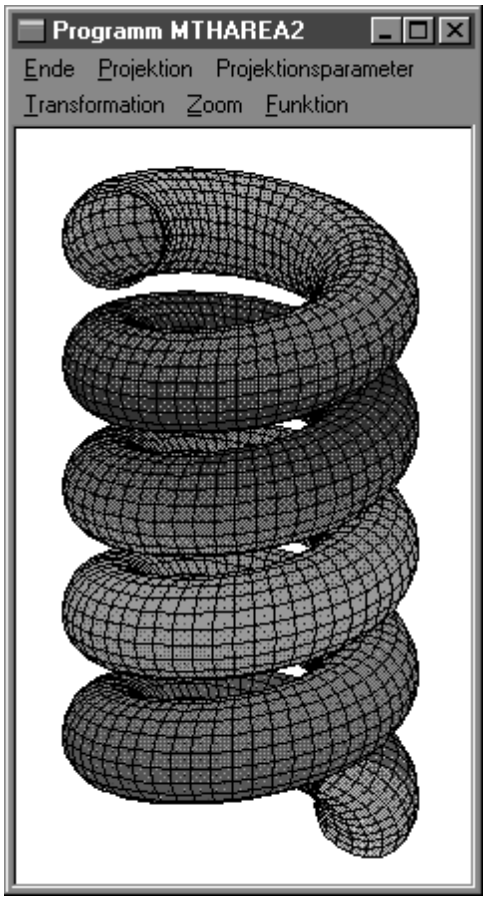
$$R = 5 \;;\quad r = 2 \;;\quad k = 0{,}8 \;.$$

Die unten zu sehende "Parabelschnecke" entsteht durch spiralförmige Rotation eines Parabelstücks um die z-Achse:

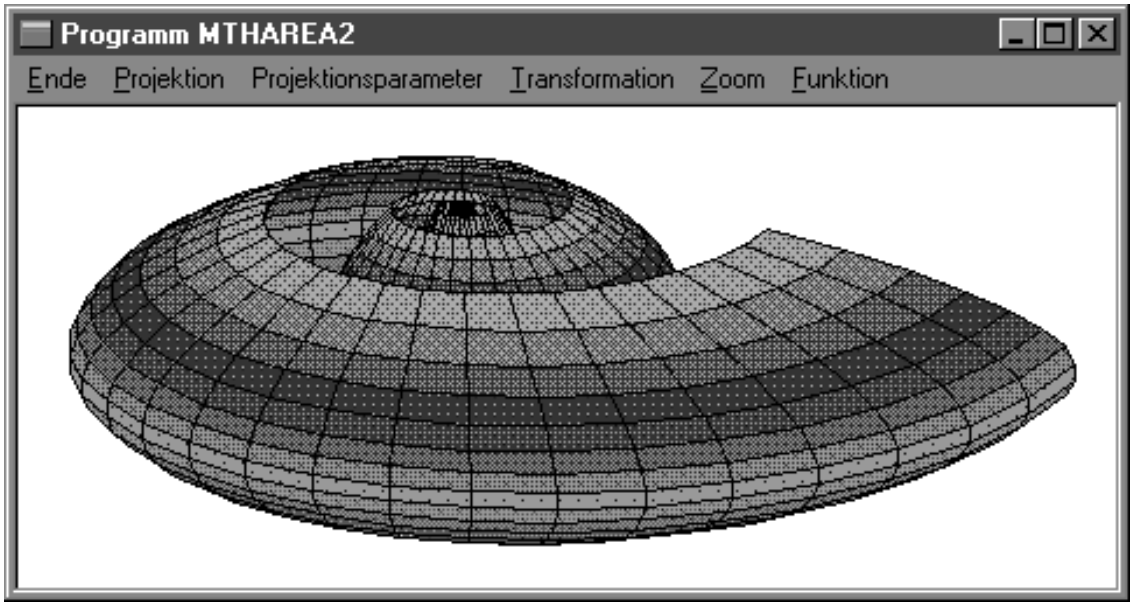
$$x = (a\,u^2 + b)\cos v\, e^{cv} \;;$$
$$y = (a\,u^2 + b)\sin v\, e^{cv} \;;$$
$$z = d\,u \;.$$

Die Parameter werden vom Konstruktor der Klasse **CParabelschnecke** wie folgt festgelegt:

$$a = -1 \;;\quad b = 2 \;;$$
$$c = 0{,}17 \;;\quad d = 30 \;.$$

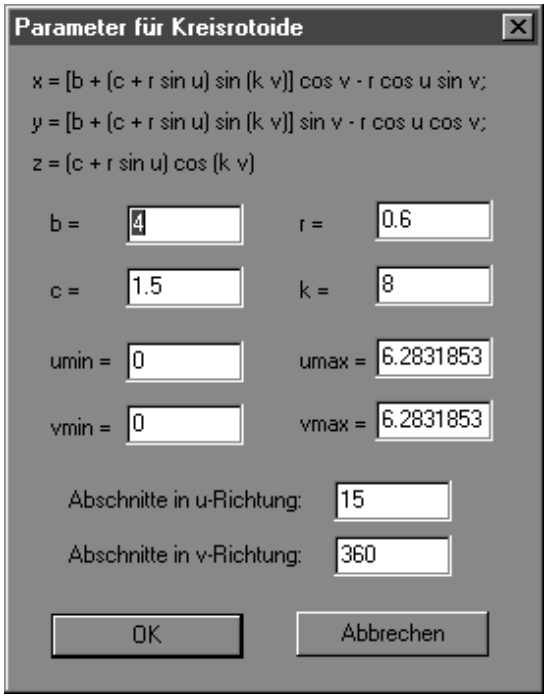


Kreis-Helix



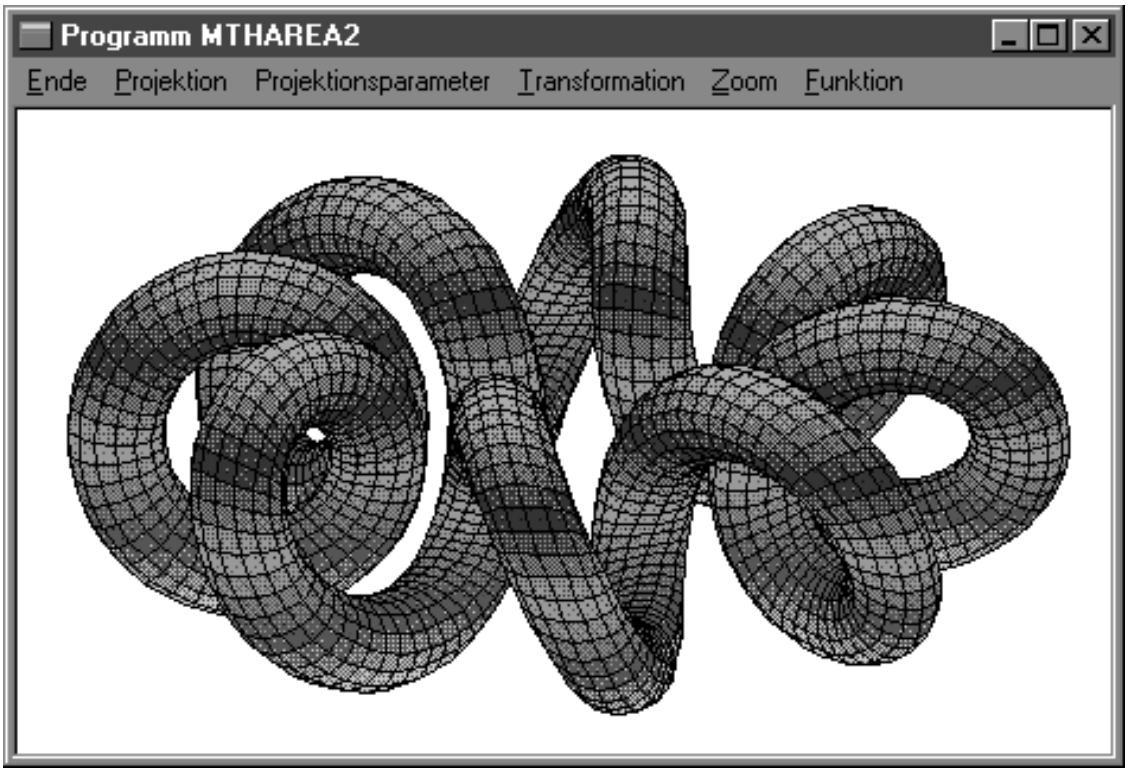
"Parabelschnecke"

Natürlich ist es für den Programmbenutzer wünschenswert, neben den Grenzen und den Diskretisierungsparametern auch die Koeffizienten ändern zu können, die in den mathematischen Funktionen stehen. Für die Darstellung der Kreis-Rotoide ist das im Programm **mtharea2.cpp** realisiert worden:

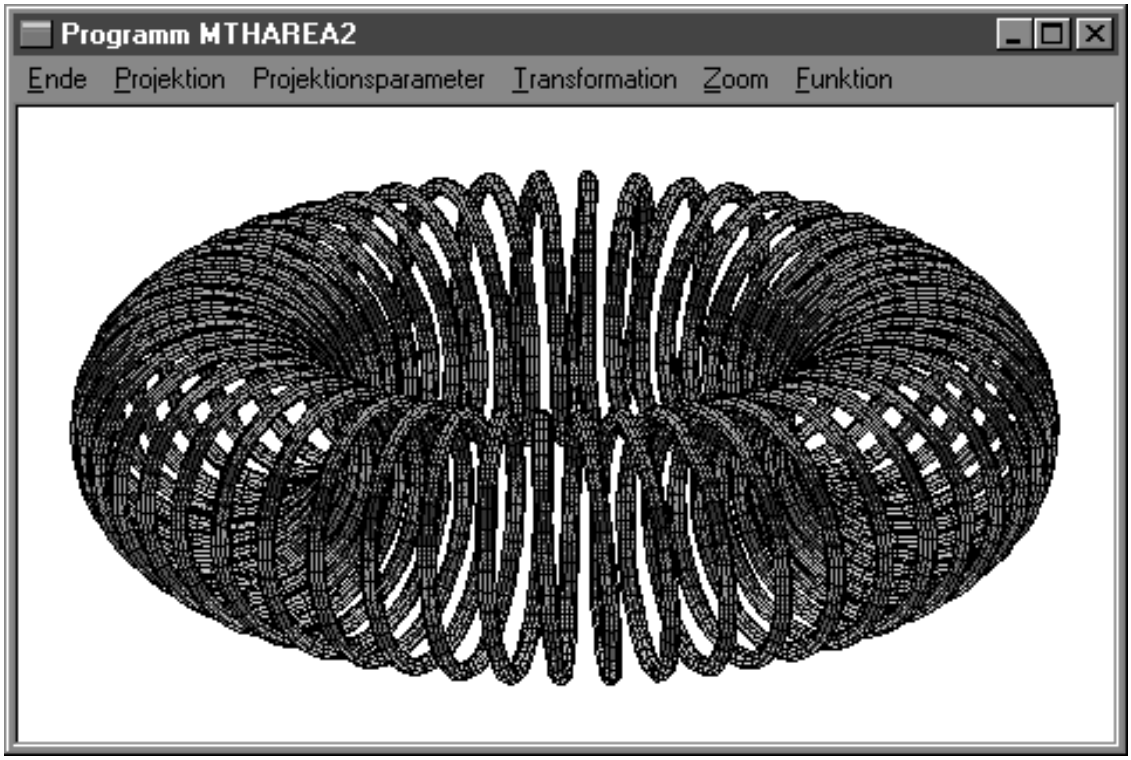


In der Klasse **CArea** ist eine Instanz der Klasse **CKreisrotoide** angesiedelt und wird mit den Standard-Werten des Konstruktors initialisiert. Vor jeder Darstellung der Funktion öffnet sich eine Dialog-Box, in der sämtliche Parameter geändert werden können (nebenstehende Abbildung, die eingestellten Werte liefern das unten zu sehende Bild). Diese werden in der Instanz der Klasse **CKreisrotoide** gespeichert und für die nachfolgende Zeichenaktion verwendet.

Eine Rotoide entsteht, wenn ein Kurvenstück schraubenförmig um einen Kreis



Kreis-Rotoide mit 8 Schlingen



Kreis-Rotoide mit 50 Schlingen

rotiert, bei einer Kreis-Rotoide rotiert ein Kreis schraubenförmig um einen Kreis. Im oberen Teil der Dialog-Box auf der vorigen Seite sind die Gleichungen zu sehen, mit denen eine solche Funktion beschrieben wird.

Rotoiden zeigen sich bei unterschiedlicher Wahl der Koeffizienten in besonders vielfältiger Form. Deshalb wurden auf dieser Seite einige Varianten dargestellt.