5 3D-Graphik-Modelle verwalten und darstellen

In diesem Kapitel werden Klassen vorgestellt, die die Verwaltung von 3D-Graphik-Modellen unterstützen. Es werden relativ einfache Modelle beschrieben, um die Anwendung der Methoden der Klassen demonstrieren zu können. Der Erweiterung, Ableitung leistungsfähigerer Klassen durch den Programmierer sind keine Grenzen gesetzt.

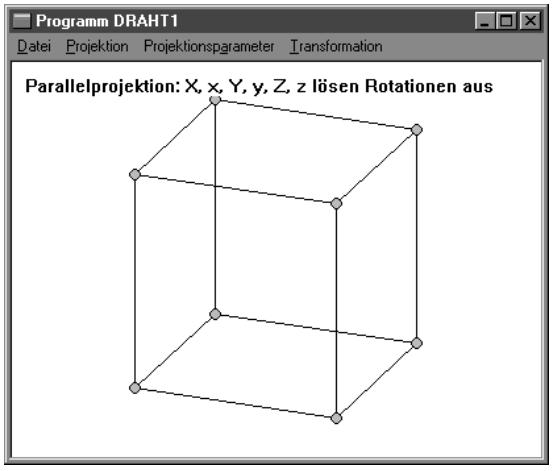
Alle behandelten Beispiele sind 3D-Probleme, obwohl von den vorgestellten Klassen auch zweidimensionale Probleme unterstützt werden. Darauf wird gelegentlich hingewiesen. Die Bearbeitung der deutlich einfacheren 2D-Probleme dürfte dem mit 3D-Aufgaben trainierten Programmierer keine Schwierigkeiten bereiten.

5.1 Die Klassen CGMod und CGElem

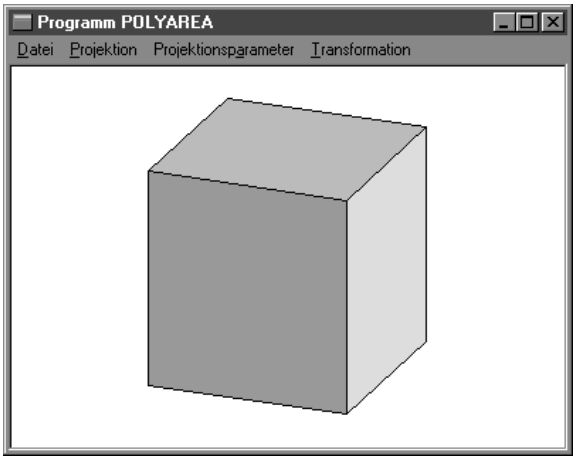
Die Klassen **CGMod** und **CGElem** sind spezialisiert auf die Verwaltung von Graphik-Modellen, die sehr einfache Objekte durch "Elemente" und "Knoten" beschreiben:

- "Knoten" sind Punkte, die durch jeweils die gleiche Anzahl von Koordinaten definiert werden (sinnvoll sind ein-, zwei- und dreidimensionale Modelle, diese "Dimensionalität" gilt immer für das gesamte Modell).
- Zu jedem "Element" gehört eine bestimmte Anzahl von Knoten. Typische Elemente sind z. B. Punkte (mit einem Knoten), gerade Linien (mit zwei Knoten) und Polygone (mit mindestens drei Knoten).

Die Darstellungen unten zeigen zwei Varianten, einen Würfel als "Knoten/Element-Modell" zu interpretieren: In beiden Fällen müssen 8 Knoten durch ihre Koordinaten beschrieben werden, im linken "Draht-Modell" gehören dazu 12 "Linien-Elemente", im rechten "Flächen-Modell" sind es 6 "Flächen-Elemente" (Polygon-Flächen mit 4 Punkten).

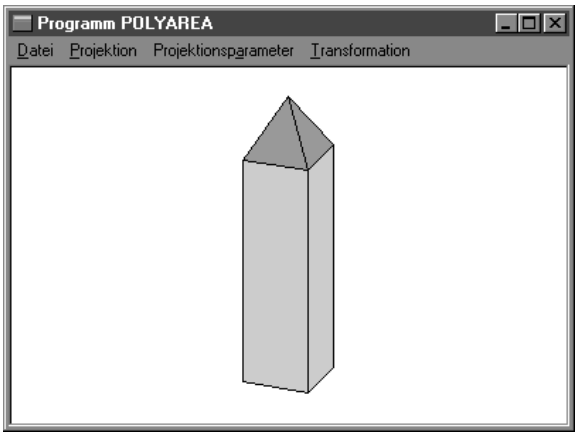


"Draht-Modell" eines Würfels



"Flächen-Modell" eines Würfels

Die nebenstehende Abbildung zeigt ein Modell mit unterschiedlichen Elementen, dreieckige und viereckige Flächen. Auch die Verwaltung solcher Modelle wird von den Klassen **CGMod** und **CGElem** unterstützt.



Modell mit unterschiedlichen Elementtypen

Weil im Regelfall ein Knoten zu mehreren Elementen gehört, wird ein Graphik-Modell in der Klasse **CGMod** durch ein Array mit den Knotenkoordinaten und eine verkette Liste der Elemente (**CGElem**-Instanzen) beschrieben. In der Klasse **CGElem** gibt es keine Koordinaten, die Elemente nehmen Bezug auf das Koordinatenfeld in **CGMod** (das könnte man natürlich auch ganz anders organisieren, i. a. jedoch mit erheblich größerem Speicherplatzbedarf).

Die Deklaration der Klasse **CGMod** findet man in **cgiw.h**:

```
class CGMod : public CGBas
{
     private:
       CGElem   *m_root_p ;            // Pointer fuer die Verwaltung
       CGElem   *m_last_p ;            // der verketteten Liste der
       CGElem   *m_next_p ;            // der CGElem-Objekte
       int      m_ne      ;            // Anzahl der Elemente
       int      m_nk      ;            // Anzahl der Knoten
       int      m_ke      ;            // Anzahl der Knoten pro Element
       int      m_kx      ;            // "Dimensionalitaet"
       double   *m_xy_p   ;            // Feld der Knotenkoordinaten
                                       // (m_nk * m_kx double-Werte)
     public:
       // Konstruktor, Destruktor:
        CGMod  (CGElem *root_p = NULL) ;
       ~CGMod  (void) ;
       // Inline-Funktionen:
       int     gtmne_md ()          const { return m_ne ; }
       int     gtmnk_md ()          const { return m_nk ; }
       int     gtmke_md ()          const { return m_ke ; }
       int     gtmkx_md ()          const { return m_kx ; }
       double* gtxyp_md (int i = 1) const { return m_xy_p + (i - 1) * m_kx ; }
       // Modellbeschreibung vom File lesen:
       void    appel_md (CGElem *elem_p) ;
       int     rddfl_md (char const *pathnm_p) ;
       // Elementliste bearbeiten:
       CGElem* gtfel_md (int *nop_p = NULL) ;
       CGElem* gtnel_md (int *nop_p = NULL) ;
} ;
```

Durch die Anordnung der Knotenkoordinaten im **m_xy_p**-Array der Klasse **CGMod** (sie müssen knotenweise eingebracht werden, z. B.: x_1 , y_1 , z_1 , x_2 , y_2 , z_2 , ...) erhalten sie implizit eine Numerierung, auf die die Elemente, die als **CGElem**-Objekte gespeichert werden, sich beziehen können.

Die Elemente werden als Objekte der Klasse **CGElem** durch ganzzahlige Werte beschrieben: Ein **int**-Array **m_param_p** enthält in der Regel die Knotennummern der Knoten, die zu dem Element gehören. Man sollte sich die Knoten von **1 , ... , m_nk** durchnumeriert vorstellen (die Anzahl der Knoten **m_nk** gehört zur Klasse **CGMod**), die zu **CGMod** gehörende **inline**-Methode **gtxyp_md** liefert für eine solche Numerierung den Pointer auf die erste Koordinate des Knotens (in der Regel die x-Koordinate).

Außer dem Array **m_param_p** findet man in **CGElem** noch zwei Farbwerte vom Typ COLORREF und den Pointer zur Verkettung in der Liste:

```
class CGElem : public CGBas
{
     private:
       int      m_nop      ;      // Anzahl der int-Werte im m_param-Array
       int      *m_param_p ;      // ... zur Elementbeschreibung (i. a.
                                  // Knotennummern, deren Koordinaten im
                                  // CGMod-Objekt gespeichert sind)
       COLORREF m_color1   ;
       COLORREF m_color2   ;
       CGElem   *m_next_p  ;      // ... fuer Verkettung in der Liste
       // ...
} ;
```

Die Erzeugung eines Modells einer 3D-Graphik ist natürlich grundsätzlich ein Problem, weil schon kleinere Modelle nur mit erheblichem Aufwand zu erzeugen sind. Für den Programm-Benutzer müßte man schon den Komfort eines modernen CAD-Systems bereitstellen, wenn man den Aufwand in Grenzen halten will. Um die Probleme zu verdeutlichen und die Benutzung der Hilfsmittel, die in der **CGIW**-Klassenbibliothek verfügbar sind, zu demonstrieren, werden zwei Wege beschritten: In den folgenden Abschnitten werden in Files bereitgestellte Modelle benutzt, abschließend wird die Modellerzeugung auf der Basis einer mathematischen Formulierung (Raumflächen) gezeigt.

5.2 Lesen eines Modells vom File mit CGMod::rddfl_md

Zur Klasse **CGMod** gehört die Methode **rddfl_md**, die einen Dateinamen einer "Graphik-Modell-Datei" erwartet, um das Objekt, mit dem sie aufgerufen wurde, mit den Informationen aus der Datei zu füllen. Die (editierbare) Datei muß folgendes Format haben:

- In der ersten Zeile müssen mindestens 3 int-Werte stehen:
  - "Anzahl der Elemente",
  - "Anzahl der Knoten" und
  - "Anzahl der ein Element beschreibenden int-Werte" (in der Regel: Anzahl der Knoten pro Element).

  Diese Werte werden von **rddfl_md** als **m_ne**, **m_nk** und **m_ke** in die aufrufende Instanz übernommen.

  Es können zwei weitere int-Werte folgen:

  - Als **m_kx** wird der vierte Wert aus der ersten Zeile der Datei gespeichert (bei Fehlen des Wertes in der Datei wird der Standardwert 3 angenommen), der

angibt, mit wieviel Koordinaten ein Punkt beschrieben wird (sinnvolle Werte sind normalerweise 1, 2 oder 3).

- Als 5. Wert kann die Anzahl der Farben angegeben werden, die in jeder Zeile, die ein Element beschreibt, am Ende angegeben werden können (maximal 2). Zulässige Farbenwerte sind die ganzzahligen Werte 0,...,7 (entsprechen den in **CGBas** definierten Standardfarben), die als COLORREF-Werte in den **CGElem**-Objekten als **m_color1** und **m_color2** gespeichert werden. Wenn keine Farben vom File gelesen werden, bleiben die vom Standard-Konstruktor eingetragenen Werte erhalten (Schwarz für **m_color1** und Weiss fuer **m_color2**).

♦ Ab Zeile 2 folgen **m_nk** Zeilen mit jeweils **m_kx** Koordinaten ("World coordinates" der Knoten), diese werden in einem double-Feld dicht gepackt in der Reihenfolge x_1 , y_1 , z_1 , x_2 , y_2 , z_2 , ... abgeliefert (das Feld wird in **CGMod::rddfl_md** dynamisch erzeugt, der Pointer auf das erste Element wird als **m_xy_p** in die aufrufende Instanz eingesetzt).

♦ Danach folgen zeilenweise die Elementbeschreibungen, das sind jeweils **m_ke** int-Werte pro Zeile. Diese werden von **CGMod::rddfl_md** in einer verketteten Liste von **CGElem**-Instanzen abgeliefert (es wird der Pointer auf das "Anchor-Element" der Liste in die aufrufende Instanz eingetragen). Eventuell können (siehe oben) nach den **m_ke** int-Werten in der Zeile noch maximal zwei int-Werte (für Farben) folgen.

Die gegebene Beschreibung ist ein Spezialfall. Der allgemeinste Fall wird indiziert durch die Angabe des Wertes 0 (oder eines negativen Wertes) auf der dritten Position der ersten Zeile des Files ("Anzahl der ein Element beschreibenden int-Werte"). Dann muß in jeder Zeile der Elementbeschreibungen ein int-Wert an der ersten Position stehen, der angibt, wieviel int-Werte für die Beschreibung dieses Elements folgen.

In der nebenstehenden Box ist die "einfachste Variante des Spezialfalls" zu sehen. Die Datei beschreibt das "Draht-Modell" des Würfels, dessen Bild am Beginn des Abschnitts 5.1 zu sehen ist:

♦ In der ersten Zeile stehen die "Anzahl der Elemente" (ein Würfel hat 12 Kanten ...), die "Anzahl der Knoten" (... und 8 Ecken) und die "Anzahl der zu einem Element gehörenden Knoten" (zu einer Kante gehören 2 Ecken).

♦ In den folgenden 8 Zeilen stehen die Koordinaten der 8 Knoten, von links nach rechts: ***x***, ***y***, ***z***. Durch die Reihenfolge der Zeilen wird eine "Knoten-Numerierung" festgelegt (in diesem Fall gibt des die Knoten **1 ... 8**), auf die sich die nachfolgenden Daten beziehen.

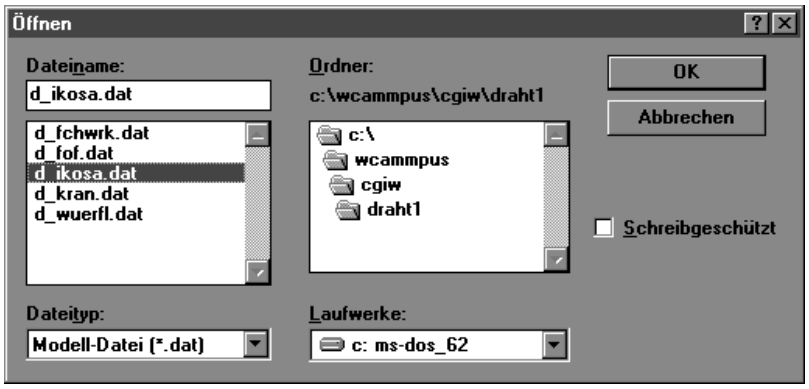
♦ Es folgen 12 Zeilen mit den Beschreibungen der Elemente (Kanten des Würfels), hier werden jeweils 2 Knotennummern angegeben, die eine Kante festlegen.

```
  12     8    2
-2.5   -2.5   -2.5
 2.5   -2.5   -2.5
 2.5    2.5   -2.5
-2.5    2.5   -2.5
-2.5   -2.5    2.5
 2.5   -2.5    2.5
 2.5    2.5    2.5
-2.5    2.5    2.5
 1 2
 2 3
 3 4
 4 1
 5 6
 6 7
 7 8
 8 5
 1 5
 2 6
 3 7
 4 8
```

Datei d_wuerfl.dat

5.3 Darstellung von "Drahtmodellen"

Das Programm **draht1.cpp** stellt ein Objekt dar, das durch seine geradlinigen Kanten beschrieben wird (Drahtmodell). Um unterschiedliche Modelle darstellen zu können, wird die Modell-Beschreibung von einer Datei gelesen. Dies geschieht in der Methode **CMainFrame::ReadFile**, die über den Standard-Dialog zum Erfragen eines Dateinamens (siehe die nebenstehende Abbildung) erreicht wird (zum Programmstart wird **CMainFrame::ReadFile** mit dem Dateinamen **d_wuerfl.dat** aufgerufen, die - wenn sie vorhanden ist - gelesen wird).



Windows-Datei-Dialog, gestartet von CMainFrame::OnGetFileName

In **CMainFrame::ReadFile** wird ein **CGMod**-Objekt erzeugt (Pointer auf das Objekt wird in die **CMainFrame**-Instanz eingetragen), mit dem **CGMod::rddfl_md** aufgerufen wird. Wenn die Datei, deren Name an **CGMod::rddfl_md** übergeben wird, existiert (und das im vorigen Abschnitt beschriebene Format hat), wird ein komplettes Modell erzeugt:

```
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_cpt.ptmx3_pt (m_cgmod_p->gtmnk_md () , m_cgmod_p->gtxyp_md () ,
                   &m_xumin , &m_xumax , &m_yumin , &m_yumax) ;
   // ... ermittelt die Extremwerte der "User coordinates", die sich fuer
   //     das Modell bei eingestellter Projektion und eingestellter
   //     Transformation ergeben
   return 1 ;
}
```

Die **CPT**-Methode, die in **CMainFrame::ReadFile** abschließend aufgerufen wird, ist in **cgiw.h** mit folgendem Prototyp vertreten:

```
int  ptmx3_pt  (int    npoin    , double xyzw[]    ,
                double *xumin_p , double *xumax_p  ,
                double *yumin_p , double *yumax_p) ;
```

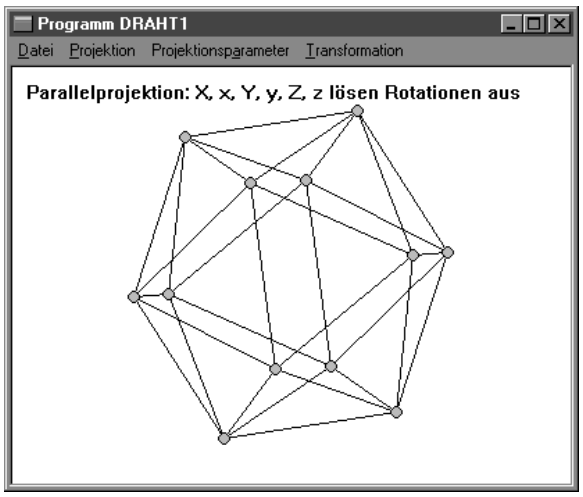
CPT::ptmx3_pt erwartet die Anzahl **npoin** der 3D-Punkte und die Koordinaten der Punkte im Feld **xyzw** ("World coordinates"), wendet auf alle Punkte die aktuelle Transformation und die aktuelle Projektion an und liefert die Extremwerte der 2D-Koordinaten ("User coordinates") in der Zeichenfläche. Die 4 Werte werden in der **CMainFrame**-Instanz gespeichert und später (in **CMainFrame::OnPaint**) für die Einstellung geeigneter "User coordinates" mit **CGI::stuci_gi** verwendet.

Gezeichnet wird (wie immer) in **CMainFrame::OnPaint**:

```
void CMainFrame::OnPaint ()
{
   if (!m_cgmod_p) return ;                      // Kein Graphik-Modell
   double    *coord_p ;
   int       k1 , k2  ;
   CPaintDC  dc (this) ;
   CGI       gi (this , &dc , &m_cpt) ;
   CPen   PenBlue     (PS_SOLID , 1 , gi.mkrgb_gi (gi.GI_BLUE)) ;
   CBrush BrushYellow (gi.mkrgb_gi (gi.GI_YELLOW )) ;
   CPen*   PenOld_p   = dc.SelectObject (&PenBlue)     ;
   CBrush* BrushOld_p = dc.SelectObject (&BrushYellow) ;
   gi.stuci_gi (m_xumin , m_yumin , m_xumax , m_yumax , 12.) ;
   CGElem *elem_p = m_cgmod_p->gtfel_md () ;    // ... liefert Pointer auf
                                                //     erstes Element
   while (elem_p)
   {
      k1 = *(elem_p->gtpap_md ())     ;         // ... zugehoerige
      k2 = *(elem_p->gtpap_md () + 1) ;         // Knotennummern
      coord_p = m_cgmod_p->gtxyp_md (k1) ;      // Pointer auf x-Koordinate
      gi.ptmov_gi (*coord_p , *(coord_p+1) , *(coord_p+2)) ;
      coord_p = m_cgmod_p->gtxyp_md (k2) ;      // Pointer auf x-Koordinate
      gi.ptlin_gi (*coord_p , *(coord_p+1) , *(coord_p+2)) ;
      elem_p = m_cgmod_p->gtnel_md () ;         // ... naechstes Element
   }
   int nk = m_cgmod_p->gtmnk_md () ;            // Anzahl der Knoten
   for (int i = 1 ; i <= nk ; i++)              // ... mit "natuerlicher
   {                                            //     Numerierung": 1...nk
      coord_p = m_cgmod_p->gtxyp_md (i) ;       // ... Pointer auf x-Koordi-
                                                //     nate des Punktes i
      gi.ptmrk_gi (gi.GI_MKFCIRCLE , 1. ,
                   *coord_p , *(coord_p+1) , *(coord_p+2) , gi.GI_NOOFFSET) ;
      // ... zeichnet Marker (gefuellten Kreis) in Standard-Groesse
   }
   dc.TextOut (10 , 10 , CString ((gi.prgtp_pt () == gi.GI_CENTRAL) ?
                         "Zentralprojektion" : "Parallelprojektion") +
                         CString (": X, x, Y, y, Z, z l÷sen ") +
                         CString (m_rot ? "Rotationen aus" :
                                          "Translationen aus"));
   dc.SelectObject (PenOld_p)   ;
   dc.SelectObject (BrushOld_p) ;
}
```

- Zuerst werden die Elemente gezeichnet: Mit **CGElem::gtfel_md** und **CGElem::gtnel_md** wird in einer typischen "Get-first-get-next"-Strategie die komplette verkettete Liste abgearbeitet.
- Die zu einem Element gehörenden Knotennummern werden mit der **inline**-Funktion **CGElem::gtpap_md** ermittelt, die zugehörigen Koordinaten mit der **inline**-Funktion **CGMod::gtxyp_md**. gezeichnet wird mit den bereits bekannten **CGI**-Methoden **ptmov_gi** und **ptlin_gi**, die "World coordinates" übernehmen.
- Die Knoten werden durch "Marker" besonders hervorgehoben. Für die Steuerung der **for**-Schleife wird zunächst die Knotenanzahl mit **CGMod::gtmnk_md** ermittelt. Die Methode **CGI::ptmrk_gi**, mit der die Marker gezeichnet werden, arbeitet wie die im Abschnitt 2.2.5 ausführlich behandelte Methode **CGI::umark_gi**, übernimmt aber "World coordinates", die mit Transformation und Projektion umgerechnet werden.

Dem Vorteil, Drahtmodelle mit wenig Aufwand (und damit sehr schnell) zeichnen zu können, steht ein gravierender Nachteil gegenüber: Die Bilder enthalten keine Information darüber, welche Kanten vorn bzw. hinten liegen. Schon beim Betrachten des Würfels, der nach dem Programmstart erscheint, kann es passieren, daß man ihn plötzlich nicht mehr "von oben" sondern "von unten" sieht. Wenn man den "Finger auf der z-Taste läßt", wechselt er sogar manchmal (natürlich nur scheinbar) die Drehrichtung.

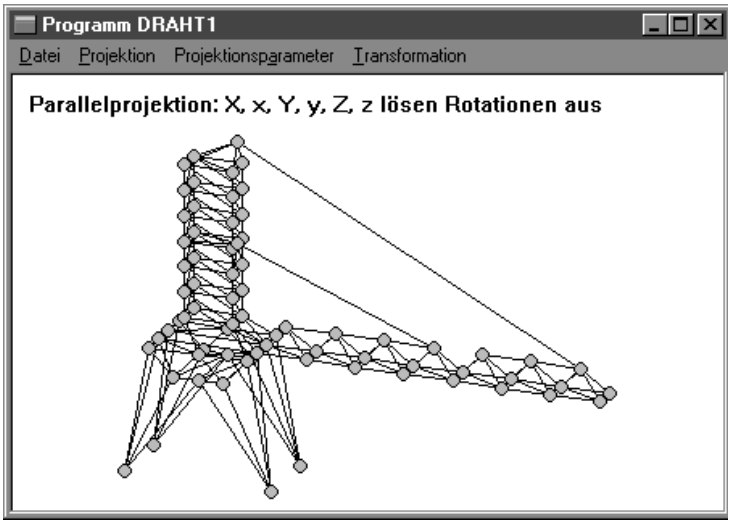


Programm draht1.cpp mit Datei d_ikosa.dat

Bei dem nebenstehend dargestellten Ikosaeder[1] ist es noch schwieriger zu entscheiden, welche Kanten vorn und welche weiter hinten liegen.

Weil die Modell-Beschreibungen aber die Informationen enthalten, wo sich jedes Element im Raum befindet, kann man sie auch auswerten. Da das Drahtmodell in der Regel ohnehin nicht der Realität entspricht (ein Würfel wird nicht durch seine 12 Kanten, sondern durch 6 Flächen begrenzt), stellt sich die Frage, ob sich für diese Modelle der Aufwand lohnt.

In der technischen Praxis gibt es allerdings zahlreiche Modelle, für die nur die "Informationen des Drahtmodells" benutzt werden. Das nebenstehende Bild zeigt so ein Modell. Natürlich sind die Stäbe eines solchen Tragwerks in der Realität Körper, aber für viele Zwecke ist diese Modellierung als Stabwerk völlig ausreichend (z. B. für die Festigkeitsberechnung, wenn man zusätzlich für alle Stäbe die Querschnittsabmessungen und die Materialeigenschaften kennt).



Programm draht1.cpp mit Datei d_kran.dat

Für die graphische Darstellung solcher Modelle (ab sofort als "Stabmodelle" bezeichnet) ist es wünschenswert, die Information, welcher Stab hinter einem anderen liegt, auch sichtbar zu machen (vor allen Dingen, um Fehler in der Modellierung zu erkennen, die fast ausschließlich durch die graphische Darstellung zu entdecken sind).

---

[1] Das Ikosaeder ist wie der Würfel einer von den fünf "Platonischen Körpern". Es wird von 20 gleichseitigen Dreiecken begrenzt.

5.4 "Breite zweifarbige Linien"

Für die Visualisierung der Information, welche von zwei sich kreuzenden Geraden vor der anderen liegt, kann man "breite zweifarbige Linien" verwenden, die mit **CGI::uwidl_gi** (ebene "User coordinates") bzw. **CGI::ptwdl_gi** ("World coordinates", die unter Beachtung der aktiven Transformation und Projektion umgerechnet werden) gezeichnet werden können. Eine Linie besteht dabei gewissermaßen aus "drei nebeneinanderliegenden Linien", von denen die mittlere eine andere Farbe hat.

Nachfolgend wird nur die Methode **CGI::ptwdl_gi** beschrieben, die übrigens nach der Transformation der Koordinaten und Projektion auf die Zeichenebene selbst **CGI::uwidl_gi** aufruft. Der etwas kompliziert aussehende Prototyp findet sich in **cgiw.h**:

```
int  ptwdl_gi  (double x1w  , double y1w , double z1w ,
                double x2w  , double y2w , double z2w ,
                CPen   *ipen_p  , CPen   *wpen_p ,
                double d1w = 0. , double d2w = 0.) ;
```

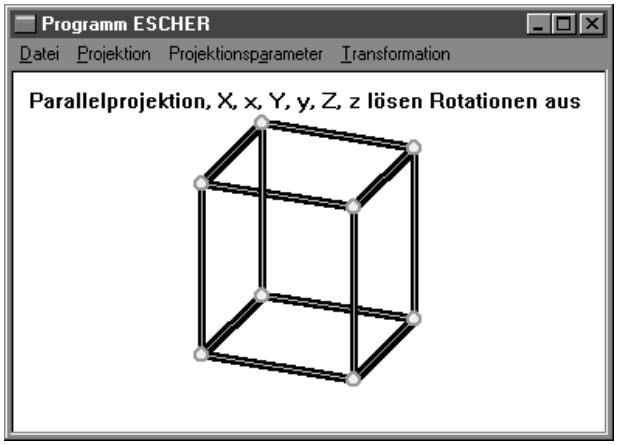
Es müssen die Koordinaten für zwei 3D-Punkte ("World coordinates") übergeben werden (**CGI::ptwidl_gi** erledigt sowohl "Move to" als auch "Line to"), außerdem die Pointer auf zwei CPen-Objekte. Der erste "Pen" ist für die "innere Linie", der zweite für die äußere Linie sollte mindestens 2 Pixel breiter sein. Die Gesamtbreite der Linie entspricht der Breite der äußeren Linie (die innere Linie wird auf die äußere Linie gezeichnet). Die beiden letzten Parameter sind optional und werden erst im Programm **stab2.cpp** (Abschnitt 5.7) mit Werten, die von den Standard-Werten abweichen, belegt und in diesem Zusammenhang dann auch erläutert.

Im Programm **escher.cpp** ist gegenüber dem Programm **draht1.cpp** (Abschnitt 5.3) nur bei der Bearbeitung der WM_PAINT-Botschaft in **CMainFrame::OnPaint** die Schleife geändert worden, mit der die Linien (Elemente) des Objekts gezeichnet werden. An die Stelle von **CGI::ptmov_gi** und **CGI::ptlin_gi** ist der Aufruf von **CGI::ptwidl_gi** getreten. Es wird eine 5 Pixel breite schwarze Line gezeichnet, die von einer 1 Pixel breiten gelben Linie überlagert wird. Die beiden "Pens" **ipen** und **wpen** werden erzeugt und die Pointer auf die beiden Objekte an **CGI::ptwdl_gi** übergeben. Nachfolgend wird nur die gegenüber dem Programm **draht1.cpp** geänderte Passage in **CMainFrame::OnPaint** gelistet:

```
CPen   PenRed    (PS_SOLID , 2 , gi.mkrgb_gi (gi.GI_RED)) ;
CBrush BrushCyan (gi.mkrgb_gi (gi.GI_CYAN)) ;
CPen*   PenOld_p   = dc.SelectObject (&PenRed) ;
CBrush* BrushOld_p = dc.SelectObject (&BrushCyan) ;
CPen   ipen (PS_SOLID , 1 , gi.mkrgb_gi (gi.GI_YELLOW)) ;
CPen   wpen (PS_SOLID , 5 , gi.mkrgb_gi (gi.GI_BLACK)) ;
CGElem *elem_p = m_cgmod_p->gtfel_md () ;    // ... liefert Pointer auf
                                             //     erstes Element
while (elem_p)
{
   k1 = *(elem_p->gtpap_md ())     ;         // ... zugehoerige
   k2 = *(elem_p->gtpap_md () + 1) ;         // Knotennummern
   coord1_p = m_cgmod_p->gtxyp_md (k1) ;     // Erster Punkt
   coord2_p = m_cgmod_p->gtxyp_md (k2) ;     // Zweiter Punkt
   gi.ptwdl_gi (*coord1_p , *(coord1_p+1) , *(coord1_p+2) ,
                *coord2_p , *(coord2_p+1) , *(coord2_p+2) ,
                &ipen , &wpen) ;
   elem_p = m_cgmod_p->gtnel_md () ;         // ... naechstes Element
}
```

♦ Nur zur Demonstration wurden vor dem Aufruf von **CGI::ptwdl_gi** auch die erst später für das Zeichnen der Marker zu verwendenden GDI-Objekte erzeugt und in den "Device context" auch eingesetzt (das ist natürlich nicht besonders sinnvoll und wird in den nachfolgenden Programmen wieder anders gehandhabt). Demonstriert wird damit, daß **CGI::ptwidl_gi** die vorher eingestellten GDI-Objekte nach dem Zeichnen wieder in den "Device context" einsetzt: Der schwarze und der gelbe Zeichenstift werden zwar verwendet, aber nach jedem Aufruf von **CGI::ptwidl_gi** ist der vorher eingesetzte rote Stift wieder gültig und wird für das Zeichnen der Markerränder benutzt.

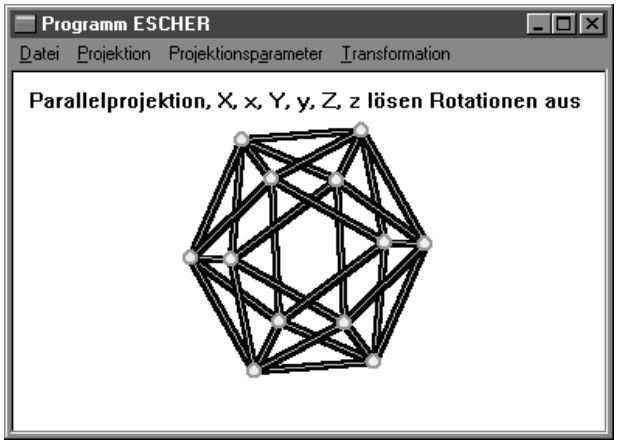
Das Ergebnis dieser Änderung ist das in der nebenstehenden Abbildung zu sehende "Escher-Bild" (nach dem holländischen Maler M. C. Escher, 1898 - 1972, der zahlreiche derartige Bilder mit "unmöglichen" Figuren gezeichnet hat).



Typisches "Escher-Bild"

Die Lösung des Problems liegt auf der Hand: Wenn man die Linien in der richtigen Reihenfolge zeichnet (die vom Betrachter am weitesten entfernte zuerst), dann überdecken die weiter vorn liegenden Linien stets die hinter ihnen liegenden. Natürlich ist es nicht sinnvoll, in den Dateien, die die zu zeichnenden Objekte beschreiben, schon die "richtige Reihenfolge" festzulegen, denn eine Drehung des dargestellten Objektes würde wieder zu einem "unmöglichen Bild" führen.

Die zu zeichnenden Linien müssen also im Programm "sortiert" werden (jeweils neu, wenn sich die aktuelle Transformation oder die Projektion geändert haben). Der Realisierung dieser Strategie im Abschnitt 5.6 werden im folgenden Abschnitt einige allgemeine Bemerkungen zum Problem der verdeckten Kanten und Flächen vorangestellt.



Schöner hätte es M. C. Escher auch nicht gekonnt

Da die Lösung des Problems zu korrekten Überdeckungen führt, wird dann der zwar falsche aber immerhin doch schöne "Escher-Effekt" verschwunden sein. Deshalb ist nebenstehend das ebenfalls mit dem Programm **escher.cpp** erzeugte Ikosaeder zu sehen, "falsch, aber schön".

5.5 "Hidden lines" und "Hidden surfaces"

Die Darstellung eines dreidimensionalen Objektes auf zweidimensionalen Ausgabemedien stößt stets auf das Problem, das Informationsdefizit, das infolge der fehlenden Tiefenwirkung eigentlich nie zu vermeiden ist, in der Weise zu reduzieren, daß die Darstellung dem angestrebten Zweck gerecht wird. Während für möglichst realitätsnahe Ansichten die Sichtbarkeit der eigentlich verdeckten Kanten nicht tolerierbar ist, werden in technischen Zeichnungen verdeckte Kanten häufig bewußt eingezeichnet, weil eine möglichst komplette Information über das dargestellte Objekt wesentlich höher als die "Schönheit der Darstellung" bewertet werden muß.

Ob aber verdeckte Kanten ausgeblendet (realistische Darstellung) oder zum Beispiel gestrichelt dargestellt werden sollen (technische Zeichnung), ändert am Problem nichts: Es muß ermittelt werden, welche Kanten dies sind.

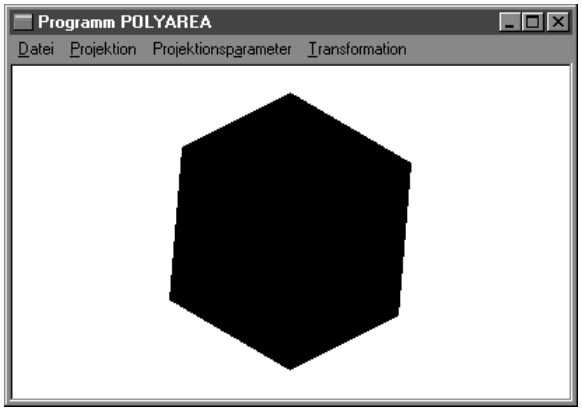
Eine weitgehend fotorealistische Darstellung ist mit dem sogenannten "Raytracing" ("Strahlenverfolgung") möglich. Dieses Verfahren wird von den CGIW-Klassen nicht unterstützt. Wer Probleme dieser Art lösen möchte, sollte sich unbedingt eines dafür geeigneten Software-Paketes bedienen. Auch dann, wenn die wesentlichen Teile des sehr aufwendigen Algorithmus von den Funktionen eines solchen Graphik-Paketes übernommen werden, bleibt für den Programmierer noch genug zu tun.

Da auch beim Verzicht auf fotorealistische Darstellung (für Konstruktionszeichnungen ohnehin nicht erwünscht) die erforderliche Rechenzeit für das Ausblenden verdeckter Linien und Flächen erheblich sein kann, werden zur Erzielung eines akzeptablen Antwortverhaltens im Dialogbetrieb gern Kompromisse bei den Algorithmen in Kauf genommen, so daß in speziellen Fällen Fehler in der Darstellung nach dem Sichtbarkeitstest eher als sehr große Rechenzeiten toleriert werden. Natürlich sollte immer das "Umschalten auf einen sauberen und aufwendigen Algorithmus" möglich sein.

Ein für die Bildschirmausgabe besonders schnelles Verfahren ist, **alle** Oberflächen des Körpers in der Reihenfolge einer "Prioritätenliste" zu zeichnen. Wenn die Reihenfolge in dieser Liste z. B. durch die Entfernung der Flächen vom "Eye point" bestimmt wird und die am weitesten entfernten Flächen zuerst gezeichnet werden, dann überzeichnen die Flächen mit kürzerer Entfernung automatisch die Flächen, die von ihnen ganz oder teilweise überdeckt werden. Dieses Verfahren verlangt im allgemeinen noch einige Verfeinerungen:

- Da die "Entfernung" einer Fläche vom "Eye point" sich natürlich immer nur auf einen (ziemlich willkürlich zu wählenden) Punkt der Fläche beziehen kann, sind bei großen und gekrümmten Flächen viele Fehler möglich (man denke daran, daß eine Kugel nur eine Fläche hat, welcher Punkt sollte für die Entfernungsbestimmung genommen werden). Abhilfe schafft man durch Einteilung aller Oberflächen in genügend kleine Teilflächen, die dann selbständig in der Prioritätenliste auftauchen.
- Ein wesentlicher Mangel des sehr schnellen Verfahrens ist, daß Körperkanten nicht automatisch sichtbar werden (man denke an einen Würfel, von dem am Ende zwar nur drei Flächen sichtbar wären, die sich jedoch nicht voneinander abgrenzen, siehe Abbildung auf der folgenden Seite). Um diesem Mangel abzuhelfen, könnte man den Flächen unterschiedliche Farben geben, was jedoch vielfach kaum praktikabel und häufig auch nicht gewollt ist. Eine andere Möglichkeit ist, Helligkeitsunterschiede

z. B. so zu generieren, daß der Winkel, den die Flächennormale mit einer Verbindungslinie zu **einer** gedachten Lichtquelle die Helligkeit bestimmt (je kleiner der Winkel, desto heller die Fläche). Auf diese Weise wird auch die Krümmung einer (in Teilflächen unterteilten) Oberfläche besonders deutlich.



Der "schwarze Körper" könnte ein Würfel sein, es ist ein Ikosaeder

Unabdingbar für technische Zeichnungen ist jedoch die Möglichkeit, auch (meist sogar ausschließlich) die gesamte gewünschte Information durch Linien auszudrücken, bei denen zwei Typen zu unterscheiden (und auf unterschiedliche Weise zu erzeugen) sind:

- **Kanten** entstehen an der Verbindungsstelle zweier Flächen, wenn diese nicht tangential aneinanderstoßen.
- **Sichtkanten** ("Silhouette lines") begrenzen bei gekrümmten Flächen (z. B.: Kugel, Torus, ...) den sichtbaren vom verdeckten Teil der Oberfläche.

Auch für die zu zeichnenden Linien muß ermittelt werden, welche Teile von davor liegenden Flächen verdeckt werden. Ein recht effektives Verfahren kann die Kombination mit dem Zeichnen von Flächen nach Prioritätenliste sein (wird im Abschnitt 5.8 am Beispiel des Zeichnens von Polygonflächen demonstriert). Dieses Verfahren ist nicht für alle Ausgabemedien geeignet (ein Stiftplotter kann nicht die schwarzen Linien anschließend mit weißen Flächen überdecken), bietet sich aber natürlich gerade für die Bildschirmausgabe, bei der ein schnelles Anwortverhalten gewünscht ist, an. Für PostScript-Ausgabe kann es direkt übernommen werden (im Speicher eines PostScript-Gerätes wird das Bild erst komplett erzeugt und dann ausgegeben), für HPGL-Ausgabe ist diese Strategie des "Übereinanderzeichnens" ungeeignet.

Natürlich kann das beschriebene Verfahren die "Hidden lines" (verdeckte Linien) und die "Hidden surfaces" (verdeckte Flächen) nach der gleichen Strategie ermitteln und alle Ausgaben zunächst in einen programminternen Speicher bringen (wie es das PostScript-Gerät auch macht) und erst das fertige Bild zeichnen. Im interaktiven Betrieb ist es für den Konstrukteur im allgemeinen eher akzeptabel, wenn "auf dem Bildschirm wenigstens immer etwas passiert".

Eine Besonderheit ist für Stabmodelle (Abschnitte 5.6 und 5.7) oder andere "linienförmige" räumliche Objekte zu beachten. Die häufig fehlenden Querschnittsinformationen bei solchen Modellen gestatten natürlich keine Untersuchung des Verdeckens von Linien durch Flächen. Im Abschnitt 5.4 wurde der dafür in der **CGI**-Klasse vorgesehene spezielle Linientyp vorgestellt, in den beiden folgenden Abschnitten wird die Lösung des Überdeckungsproblems bei Stabmodellen mit dem "nach dem Abstand vom Betrachter geordneten Zeichnen breiter zweifarbiger Linien" demonstriert.

5.6 Die Klasse CGObj, Darstellung von "Stabmodellen"

5.6.1 Die abstrakte Klasse CGObj

Zur Realisierung des "Zeichnens der Objekte in der Reihenfolge ihres Abstands vom Betrachter" ist das Anlegen einer sortierten Liste der Objekte erforderlich. Dies ist besonders effektiv durch eine Anordnung der Objekte in einem "binären Baum" zu realisieren (zur Arbeit mit binären Bäumen vgl. z. B. "Dankert: Praxis der C-Programmierung", Teubner-Verlag 1997, Kapitel 8). Dies wird durch die Klasse **CGObj** unterstützt, so daß der Programmierer von den rekursiven Techniken beim Arbeiten mit binären Bäumen nichts merkt.

Die **abstrakte Klasse CGObj** ist die Klammer für die aus ihr abzuleitenden Klassen für spezielle Graphik-Objekte. Sie enthält als Datenelemente neben dem Sortierkriterium (**m_dist** - "Abstand des Objekts vom Betrachter") nur zwei Pointer auf Instanzen des eigenen Typs (**m_left_p** und **m_right_p**), mit denen eine Instanz einer von **CGObj** abgeleiteten Klasse in einem binären Baum verankert wird:

```
class CGObj : public CGBas
{
     private:
       double   m_dist    ;
       CGObj   *m_left_p  ;
       CGObj   *m_right_p ;
     public:
        CGObj  (double dist = 0.) ;
       ~CGObj  (void) ;
       // Inline-Funktion:
       double  gtdis_md () { return m_dist ; }
       // Bearbeiten des binaeren Baumes:
       void    dromd_md (CGI   *cgi_p) ;
       void    drrec_md (CGObj *anchor_p) ;
       CGObj*  insot_md (CGObj *root_p , double dist) ;
       void    updbt_md (CGObj *anchor_p) ;
       // Virtuelle Funktion, die das Zeichnen fuer ein Objekt einer
       // aus CGObj abgeleiteten Klasse ausfuert:
       virtual void drobj_md (CGI *cgi_p) = 0 ;
} ;
```

Die wichtigste Methode der Klasse für den Programmierer ist **CGObj::insot_md**, mit der das Einsortieren einer Instanz in den binären Baum erledigt wird. Sie wird mit einer Instanz einer aus **CGObj** abgeleiteten Klasse aufgerufen und erwartet den "Anchor pointer" **root_p** des binären Baums. Dies darf beim ersten Aufruf der NULL-Pointer sein, als Return-Wert liefert **CGObj::insot_md** den "Anchor pointer" des Baums. Beim ersten Aufruf ist das der Pointer auf die aufrufende Instanz, die zum "Anchor" des Baums wird. Bei jedem nachfolgenden Aufruf muß dann dieser "Anchor pointer" als erstes Argument übergeben werden und wird auch als Return-Wert wieder abgeliefert.

Der zweite Parameter, den **CGObj::insot_md** erwartet, ist das Sortierkriterium ("Abstand vom Betrachter"). Es wird als **m_dist** in die aufrufende Instanz übernommen (die beiden Pointer **m_left_p** und **m_right_p** werden mit NULL initialisiert) und dient gleichzeitig als Wert, mit dem die "geordnete Positionierung" der Instanz vorgenommen wird: Jeder "linke Nachfolger" im Baum ist weiter entfernt vom Betrachter und kein "rechter Nachfolger" ist weiter entfernt vom Betrachter als die Instanz, die auf die Nachfolger pointert.

Die Klasse **CGObj** enthält die rein virtuelle Funktion **drobj_md**, die für das Zeichnen des Graphik-Objekts zuständig ist und in den abgeleiteten Klassen definiert sein muß (nur die abgeleiteten Klassen haben die Informationen, die zum Zeichnen eines Objektes benötigt werden).

Der Programmierer kann also für beliebige Graphik-Objekte Klassen aus **CGObj** ableiten. Bereits vorgesehen in **cgiw.h** sind drei abgeleitete Klassen, **CGNod** für das Zeichnen von Knoten, **CGRod** für das Zeichnen von Stäben (als "breite zweifarbige Linien") und **CGPoly** für das Zeichnen von Polygonflächen. Stellvertretend für diese Klassen wird hier die Klasse **CGNod** betrachtet:

```
class CGNod : public CGObj
{
     protected:
       double   *m_xyz_p  ;
       COLORREF m_linecol ;
       COLORREF m_fillcol ;
     public:
        CGNod  (double *xyz_p , COLORREF linecol , COLORREF fillcol) ;
        CGNod  (double *xyz_p = NULL) ;
       ~CGNod  (void) ;
       void drobj_md (CGI *cgi_p) ;
} ;
```

Der üblicherweise zu verwendende Konstruktor übernimmt einen Pointer auf das Koordinaten-Tripel (Punkt, an dem der Marker gezeichnet werden soll) und zwei Farben (für den Rand bzw. das Ausfüllen des Markers) und initialisiert damit die drei Datenelemente (die von **CGObj** geerbten Datenelemente werden mit dem Standard-Konstruktor von **CGObj** initialisiert).

Zwingend ist, daß die in **CGObj** rein virtuell deklarierte Funktion **drobj_md** in **CGNod** definiert wird. Die Funktion erwartet einen Pointer auf das aktuelle **CGI**-Objekt. Ihre Definition findet man (neben den Konstruktoren) in der Datei **cgnod_md.cpp**:

```
void CGNod::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->ptmrk_gi (cgi_p->GI_MKFCIRCLE , 1. ,
                    *m_xyz_p , *(m_xyz_p + 1) , *(m_xyz_p + 2) ,
                    cgi_p->GI_NOOFFSET) ;
   pDC->SelectObject (penold_p)   ;
   pDC->SelectObject (brushold_p) ;
}
```

Der Pointer auf den "Device context" wird in **CGNod::drobj_md** beim **CGI**-Objekt erfragt, "Pen" und "Brush" werden konstruiert und in den "Device context" eingesetzt. Schließlich wird die eigentliche Arbeit, das Zeichnen des Markers mit **CGI::ptmrk_gi**, erledigt. Zum Schluß wird "aufgeräumt", indem der ursprüngliche Zustand des "Device contextes" wieder hergestellt wird.

Das Arbeiten mit der abstrakten Klasse **CGObj** und den daraus abgeleiteten Klassen wird am besten an den nachfolgend demonstrierten Beispielen deutlich.

5.6.2 Das Programm stab1.cpp

Das Programm **stab1.cpp** unterscheidet sich vom Programm **escher.cpp** (Abschnitt 5.4) dadurch, daß alle darzustellenden Objekte (hier: Knoten und Stäbe) in **CMainFrame::On-Paint** zunächst geordnet (nach dem "Abstand vom Betrachter") in einen binären Baum eingebracht werden. Erst danach wird gezeichnet, indem der binäre Baum abgearbeitet wird, so daß die Objekte in der gewünschten Reihenfolge gezeichnet werden und sich korrekte Überdeckungen ergeben.

Die ausführlich kommentierte Methode **CMainFrame::OnPaint** wird zunächst gelistet und danach noch zusätzlich erläutert (man beachte, daß das Graphik-Modell wie bei allen Vorgänger-Versionen bereits existiert, es wurde durch Einlesen von einer Datei erzeugt, ein Pointer auf die **CGMod**-Instanz des Modells ist als **m_cgmod** in der Klasse **CMainFrame** enthalten):

```
void CMainFrame::OnPaint ()
{
   CGNod   *m_cgnod_p ;
   CGRod   *m_cgrod_p ;
   double  *coord1_p , *coord2_p ;
   int     k1 , k2 ;
   if (!m_cgmod_p) return ;                     // Kein Graphik-Modell
   // Es wird ein "Objekt-Modell" in einem binaeren Baum erzeugt:
   // Alle Graphik-Elemente (hier nur Knoten als Instanzen der Klasse
   // CGNod und Staebe als Instanzen der Klasse CGRod) 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 ;        // "Anchor pointer" auf
   m_objroot_p = NULL ;                         // den binaeren Baum
   int nk = m_cgmod_p->gtmnk_md () ;            // Anzahl der Knoten
   for (int i = 1 ; i <= nk ; i++)              // ... ueber alle Knoten
   {
      m_cgnod_p   = new CGNod (m_cgmod_p->gtxyp_md (i)) ;
      // ... erzeugt eine Instanz der Klasse CGNod und initialisiert sie
      //     mit den Werten, die dem Modell m_cgmod_p entnommen wurden.
      m_objroot_p = m_cgnod_p->insot_md (m_objroot_p ,
                        m_cpt.ptdis_pt (m_cgmod_p->gtxyp_md (i))) ;
      // ... fuegt die Instanz von CGNod in den binaeren Baum ein, wobei
      //     die Einordnung von der Distanz vom Betrachter abhaengig gemacht
      //     wird, die von der ueberladenen CPT-Methode ptdis_pt hier
      //     mit dem Pointer auf das Koordinatentripel berechnet wird.
   }
   CGElem *elem_p = m_cgmod_p->gtfel_md () ;    // ... liefert Pointer auf
                                                //     erstes Element
   while (elem_p)
   {
      k1 = *(elem_p->gtpap_md ())     ;         // ... zugehoerige
      k2 = *(elem_p->gtpap_md () + 1) ;         // Knotennummern
      coord1_p = m_cgmod_p->gtxyp_md (k1) ;     // Erster Punkt
      coord2_p = m_cgmod_p->gtxyp_md (k2) ;     // Zweiter Punkt
      m_cgrod_p   = new CGRod (coord1_p , coord2_p , 5 , 1) ;
      // ... erzeugt eine Instanz der Klasse CGRod und initialisiert sie
      //     mit den Werten, die dem Modell m_cgmod_p entnommen wurden.
      m_objroot_p = m_cgrod_p->insot_md
                       (m_objroot_p , m_cpt.ptdis_pt
                              ((*coord1_p      + *coord2_p      ) / 2 ,
                               (*(coord1_p + 1) + *(coord2_p + 1)) / 2 ,
                               (*(coord1_p + 2) + *(coord2_p + 2)) / 2)) ;
```

```
      // ... fuegt die Instanz von CGRod 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 Elemente als Bezugspunkte
      //     verwendet, was bei Modellen mit stark unterschiedlichen
      //     Stablaengen 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 , 20.) ;
   // 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 Zeichenfunkgion 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) ;
   dc.TextOut (10 , 10 , CString ((gi.prgtp_pt () == gi.GI_CENTRAL) ?
                         "Zentralprojektion" : "Parallelprojektion") +
                         CString (": X, x, Y, y, Z, z l÷sen ") +
                         CString (m_rot ? "Rotationen aus" :
                                          "Translationen aus"));
}
```

- Der "Anchor pointer" auf den binären Baum **m_objroot_p** wird mit NULL initialisiert, danach wird die Pointer-Variable bei jedem **insot_md**-Aufruf als erstes Argument übergeben und zur Aufnahme des Return-Wertes verwendet. Alles übrige erledigt **insot_md**.

- Zur Konstruktion der **CGNod**-Objekte (Knoten) wird der vereinfachte Konstruktor verwendet, der nur einen Pointer auf das Koordinaten-Tripel erwartet. Dieser wird mit der bereits im Abschnitt 5.1 beschriebenen Methode **CGMod::gtxyp_md** ermittelt. Da diesem Konstruktor keine Farbenwerte übergeben werden, initialisiert er die entsprechenden Datenelemente mit Standardwerten ("Schwarz" für das Zeichnen des Randes und "Weiß" für die Füllung des Markers).

- Die von **CGObj** geerbte Methode **insot_md** (geordnetes Einfügen eines Objektes in den binären Baum) erwartet als zweiten Parameter den "Abstand des Objektes vom Betrachter". Dieser wird vereinfacht mit der Methode **CPT::ptdis_pt** berechnet, die in zwei Varianten mit folgenden Prototypen verfügbar ist:

  ```
  double  ptdis_pt  (double xw , double yw , double zw) ;
  double  ptdis_pt  (double *xyzw_p) ;
  ```

  In der ersten Form werden die drei "World coordinates" des Punktes, der für die Abstandsberechnung benutzt werden soll, erwartet. Diese Variante wird für das Einfügen der "Stab-Objekte" verwendet. Die zweite Variante erwartet einen Pointer auf das Koordinaten-Tripel und wird für das Einsetzen der "Knoten-Objekte" benutzt.

- Die Methode **CPT::ptdis_md** berücksichtigt sowohl die aktuelle 3D-Transformation als auch die eingestellte Projektion, liefert allerdings nur ein vergleichendes (aber für den Verwendungszweck völlig ausreichendes) Maß für den "Abstand des Punktes vom Betrachter":

- Bei eingestellter Zentralprojektion wird das Quadrat des Abstandes des übergebenen Punktes vom "Eye point" abgeliefert.
- Bei eingestellter Parallelprojektion ist der Begriff "Abstand vom Betrachter" ohnehin nicht sinnvoll, weil dieser sich "im Unendlichen" befindet. Deshalb wird ein vorzeichenbehafteter Wert geliefert, der der mit einem konstanten Faktor multiplizierte Abstand des Punktes von der Projektionsebene ist.

Für den Programmierer ist das eigentlich nur von sekundärem Interesse, wenn er darauf vertraut, daß **CPT::ptdis_md** einen für den Verwendungszweck (Sortieren) geeigneten Wert liefert.

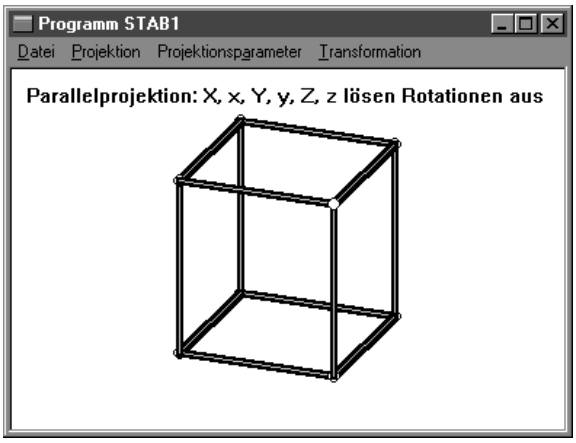
♦ Auch zur Konstruktion der **CGRod**-Objekte (Stäbe) wird ein vereinfachter Konstruktor verwendet, der nur die Pointer auf die beiden Koordinaten-Tripel und die beiden Linienbreiten erwartet. Da diesem Konstruktor keine Farbenwerte übergeben werden, initialisiert er die entsprechenden Datenelemente mit Standardwerten ("Schwarz" für das Zeichnen der äußeren Linie und "Gelb" für die innere Linie).

♦ Für das Einsortieren der Stäbe wird der Stabmittelpunkt als Punkt für die "Abstands-Berechnung" gewählt. Dies ist nicht ganz ohne Risiko und kann bei Modellen mit Stäben mit stark unterschiedlichen Längen durchaus zu einem "Überdeckungs-Fehler" führen. Hier könnte (mit einigem Aufwand) als Verbesserung das Aufteilen "langer" Stäbe in mehrere kürzere vorgesehen werden, die dann jeweils selbständig in den binären Baum eingefügt werden.

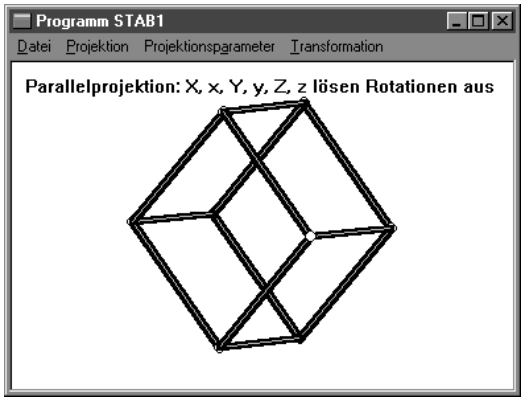
♦ Wenn der binäre Baum komplett ist, kann das Zeichnen aller darin verankerten Objekte besonders einfach erledigt werden: Die Methode **CGObj::dromd_md** durchläuft den gesamten Baum rekursiv und ruft für jedes Objekt die in **CGObj** rein virtuell deklarierte Funktion **drobj_md** auf, so daß stets die passende Funktion dieses Namens der abgeleiteten Klassen verwendet wird. Für Interessenten der rekursiven Programmierung (eigentlich braucht den Programmierer nicht zu interessieren, wie die Funktion arbeitet) nachstehend das Listing von **CGObj::dromd_md**:

```
static CGI *gi_p ;
void CGObj::dromd_md (CGI *cgi_p)
{
  gi_p = cgi_p ;
  drrec_md (this) ;
}
void CGObj::drrec_md (CGObj *anchor_p)
{
  if (anchor_p->m_left_p)
  {
     drrec_md (anchor_p->m_left_p) ;
  }
  anchor_p->drobj_md (gi_p) ;
  if (anchor_p->m_right_p)
  {
     drrec_md (anchor_p->m_right_p) ;
  }
}
```

Da die Rekursionstiefe (und damit die Belastung des Stacks) sehr groß werden kann, wird der Pointer auf das **CGI**-Objekt, der für die Zeichenaktion benötigt wird, vorab einer globalen Variablen (mit Datei-Gültigkeitsbereich) zugewiesen.



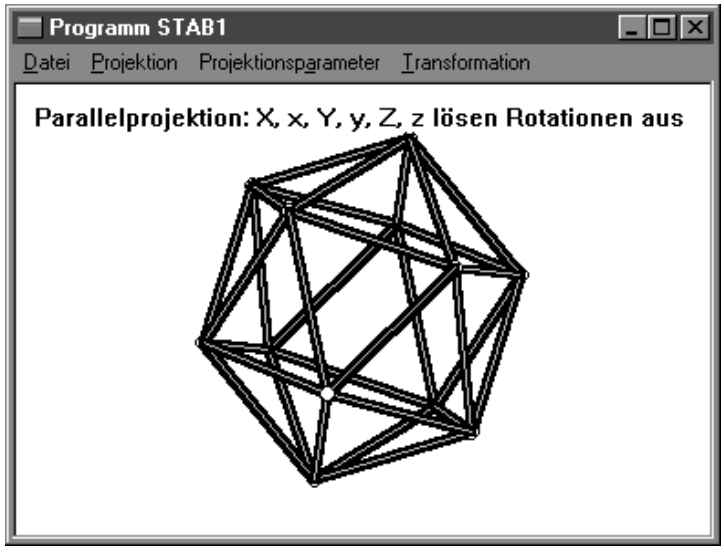
Die gegenseitige Überdeckung der Stäbe ist korrekt ...



... und bleibt korrekt bei beliebiger Rotation

Die beiden Bilder oben zeigen, daß der Mangel des Programms **escher.cpp**, die gegenseitige Überdeckung der Stäbe dem Zufall zu überlassen, behoben ist. Die Darstellung bleibt bei beliebigen Transformationen und Projektionen korrekt, weil alle Berechnungen, die zur Festlegung der Reihenfolge der Zeichenaktionen erforderlich sind, unter Berücksichtigung der jeweils aktuellen Transformation und der eingestellten Projektion durchgeführt werden.

Es zeigt sich allerdings ein kleiner Mangel, der in den bisherigen Programmen, bei denen die Knoten jeweils als letzte Objekte gezeichnet wurden, nicht auffiel: Eigentlich müßten alle Elemente um die "Abmessungen der Knoten" verkürzt werden, um ein sauberes Bild zu erzeugen, denn mit der Verwendung der Knotenkoordinaten als Endpunkte der Elemente ergeben sich zwangsweise "Durchdringungen".



Ikosaeder: Die gegenseitigen Überdeckungen der Stäbe sind korrekt, an den Knoten sind allerdings noch Verbesserungen wünschenswert

Da hier zunächst nur das Einbringen von unterschiedlichen Objekt-Typen in den binären Baum demonstriert werden sollte, wurde dieser Mangel zunächst in Kauf genommen. Die kleine "Schönheitsoperation" zur Beseitigung des Mangels, die von der für das Zeichnen der Stäbe benutzten Funktion unterstützt wird, ist der im folgenden Abschnitt vorgestellten verbesserten Programm-Version vorbehalten.

5.7 Mögliche und realisierte Verbesserungen, Programm stab2.cpp

In diesem Abschnitt sollen einige mögliche (und wünschenswerte) Verbesserungen des Programms **stab1.cpp** (Abschnitt 5.6) diskutiert werden. Die Verbesserungen, bei denen die **CGIW**-Klassen den Programmierer unterstützen, sind im Programm **stab2.cpp** realisiert. Damit soll auch demonstriert werden, wie der Programmierer dann, wenn die **CGIW**-Klassen nicht genau das anbieten, was zu seinem Problem paßt, diese mit Vorteil als Basisklassen für eigene abgeleitete Klassen nutzen kann.

5.7.1 Knoten als Kugeln, "verkürzte Stäbe"

Der bereits im Abschnitt 5.6 diskutierte Mangel der Durchdringung der Stäbe mit den (als Marker dargestellten) Knoten wird behoben, indem die von der Methode **CGI::ptwdl_gi** angebotene Möglichkeit genutzt wird, die Linie "verkürzt zu zeichnen". Um die Verkürzungen angeben zu können, werden die Knoten nicht mehr (wie in **stab1.cpp**) als Marker sondern als "durch Kreise dargestellte Kugeln" gezeichnet, deren Radius in "World coordinates" angegeben werden muß. Dies bringt noch einen weitren Vorteil:

Marker haben eine feste Größe, die auch bei Zentralprojektionen unverändert bleibt, so daß sehr nah am Betrachter liegende Knoten in der gleichen Größe wie weit entfernte Knoten dargestellt werden. Damit geht ein wichtiger Effekt der Zentralprojektion verloren. Die "als Kreise dargestellten Kugeln" dagegen ändern ihre Größe in Abhängigkeit von der Entfernung vom "Eye point". Die Methode zum Zeichnen dieser Objekte hat den Prototyp

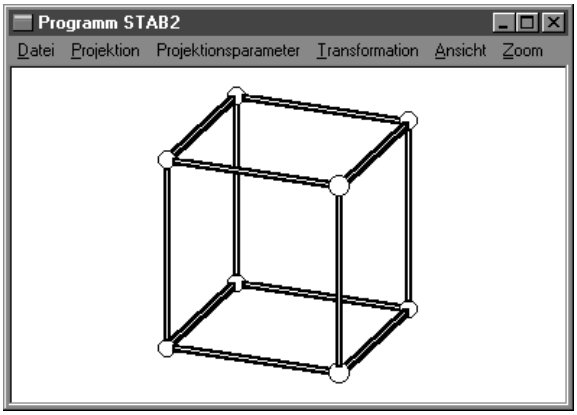
```
int ptbll_gi (double xw , double yw , double zw , double radw) ;
```

und ist eine Ausnahme unter den **CGI**-Methoden. Transformationen und Projektionen beziehen sich stets auf einzelne Punkte, und deshalb müssen jede Linie und Fläche und jeder Körper punktweise transformiert und projiziert werden (natürlich nutzt man z. B. aus, daß eine Gerade im Raum auch in der Projektionsebene eine Gerade sein wird, und transformiert und projiziert nur zwei Punkte).

Aber selbst für eine Kugel ist es schon schwierig. Es könnten z. B. unterschiedliche Skalierungen in den Koordinatenrichtungen eingestellt sein, so daß die Kugel als Ellipsoid dargestellt werden müßte. **CGI::ptbll_gi** arbeitet wesentlich einfacher: Der in "World coordinates" anzugebene Mittelpunkt (die ersten drei Parameter) wird unter Auswertung der aktuellen Transformation und der Projektion in "User coordinates" umgerechnet, der Radius (vierter Parameter) wird nur bei eingestellter Zentralprojektion proportinal zum Abstand vom "Eye point" (mit dem Parameter λ aus Gleichung 4.13) vergrößert oder verkleinert. In der Projektionsebene wird stets ein Kreis gezeichnet. Man beachte diese eingeschränkte Verwendbarkeit dieser Methode.

Der Radius der Kugeln, die in **stab2.cpp** die Knoten repräsentieren, gehört zur Klasse **CMainFrame** als **m_rad**, erhält beim Laden eines Modells eine sinnvolle Vorbelegung und kann gegebenenfalls über einen Dialog (öffnet sich nach Anklicken des Menü-Angebots **Ansicht**) geändert werden. Genau um diesen Radius werden die mit **CGI::ptwdl_gi** zu zeichnenden Stäbe an jedem Ende verkürzt. Diese Methode sieht dafür bereits zwei zusätzliche Parameter vor, die im Regelfall (wenn nicht verkürzt werden soll) nicht angegeben werden müssen, weil sie die Standard-Vorbelegung **0.** haben.

Das Ziel ist schließlich, daß die Durchdringungen der Stäbe und Knoten verhindert werden, so daß sich ein "sauberes Bild" zeigt (die Abbildung zeigt den auch mit **stab1.cpp** schon dargestellten Würfel, die Knoten wurden via **Ansicht** etwas vergrößert, um den Unterschied zu verdeutlichen).



Bildschirm nach dem Start von stab2.cpp

Die Funktionen, die diese spezielle Darstellung ermöglichen, sind in den Klassen **CGRod** und **CGNod**, die im Programm **stab1.cpp** verwendet wurden, natürlich nicht vorgesehen. Deshalb werden zwei neue Klassen deklariert, die aus den genannten Klassen abgeleitet werden. Man findet sie in **stab2.h**:

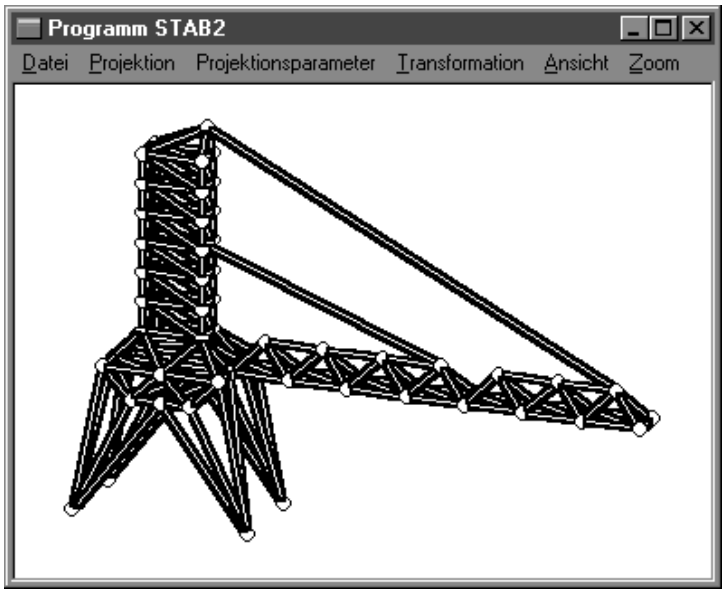
```
class CGNod_D : public CGNod
{
     private:
       double   m_rad ;
     public:
        CGNod_D  (double *xyz_p , double rad) : CGNod (xyz_p) { m_rad = rad ; }
       ~CGNod_D  (void) {}
       void drobj_md (CGI *cgi_p) ;
} ;

class CGRod_D : public CGRod
{
     private:
       double   m_rad ;
     public:
        CGRod_D (double   *xyz1_p , double   *xyz2_p ,
                 int      wpix    , int      ipix    ,
                 COLORREF wcol    , COLORREF icol    , double rad) :
              CGRod  (xyz1_p , xyz2_p , wpix , ipix , wcol , icol)
        {
           m_rad = rad ;
        }
       ~CGRod_D  (void) {}
       void drobj_md (CGI *cgi_p) ;
} ;
```

Die Konstruktoren reichen die meisten Parameter an den Konstruktor der jeweiligen Basisklasse weiter. Für jede der beiden Klassen muß natürlich die (in **CGObj**, der "Mutter dieser Klassen", rein virtuell deklarierte) Funktion **drobj_gi** geschrieben werden. Diese Funktionen findet man in **stab2.cpp**:

```
void CGNod_D::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->ptbll_gi (*m_xyz_p , *(m_xyz_p + 1) , *(m_xyz_p + 2) , m_rad) ;
   pDC->SelectObject (penold_p)   ;
   pDC->SelectObject (brushold_p) ;
}
```

```
void CGRod_D::drobj_md (CGI *cgi_p)
{
   CPen  ipen (PS_SOLID , m_ipix , m_icol) ;
   CPen  wpen (PS_SOLID , m_wpix , m_wcol) ;
   cgi_p->ptwdl_gi (*m_xyz1_p , *(m_xyz1_p + 1) , *(m_xyz1_p + 2) ,
                    *m_xyz2_p , *(m_xyz2_p + 1) , *(m_xyz2_p + 2) ,
                    &ipen , &wpen , m_rad , m_rad) ;
}
```

Die Methode **CGI::ptwdl_gi** erwartet für jeden Endpunkt einen gesonderten Wert für die Verkürzung. Hier wird auf beiden Positionen das gleiche Argument übergeben. Im Abschnitt 5.6 wurde bereits das Problem diskutiert, daß bei Stäben mit stark unterschiedlichen Längen der Stabmittelpunkt als Bezugspunkt für die Berechnung der "Abstände vom Betrachter" unter Umständen zu Überdeckungsfehlern führen könnte. Dann müßten die langen Stäbe (unter Umständen mehrfach) unterteilt werden, um ein sauberes Bild zu erzeugen, wobei für die Teilstäbe dann maximal an einem Ende eine Verkürzung vorzusehen wäre, was man mit **CGI::ptwdl_gi** aber problemlos realisieren kann. Diese Verfeinerung ist in **stab2.cpp** nicht vorgesehen. Das nebenstehende Bild zeigt einen solchen Kandidaten, weil die beiden Seile zu den Auslegern deutlich länger sind als die übrigen Stäbe. Aber man müßte schon einen recht raffiniert gewählten "Eye point" und einen dazu passenden Referenzpunkt einstellen, um einen "Überdeckungsfehler" zu erzeugen (möglich ist es allerdings).



Programm stab2.cpp mit der Datei s_kran.dat

5.7.2 Stäbe mit unterschiedlichen Farben zeichnen

Bereits im Abschnitt 5.2 wurde beschrieben, daß in der Datei, die ein Modell beschreibt, für die "Elemente" (hier: Stäbe) zwei Farbwerte angegeben werden können, die von **CGMod::rddfl_md** gelesen und in den **CGElem**-Objekten abgelegt werden. Bisher wurden nur Dateien verwendet, die eine solche Farb-Information nicht enthielten, und die vom Konstruktor der Klasse **CGElem** gesetzten Standardwerte (Schwarz und Weiß) blieben unverändert (sie wurden von **stab1.cpp** aber ohnehin nicht verwendet).

In **stab2.cpp** werden die in den **CGElem**-Objekten gespeicherten Farb-Informationen verwendet, um die beiden Farben zu setzen, mit denen die Stäbe gezeichnet werden. Dies geschieht dadurch, daß beim Erzeugen der Objekte für den binären Baum, mit dem das "geordnete Zeichnen" realisiert wird, dem Konstruktor der **CGRod_D**-Instanzen die aus den

CGElem-Objekten entnommenen Farben übergeben werden. In **CMainFrame::OnPaint** sieht das so aus:

```
m_cgrod_p   = new CGRod_D (coord1_p , coord2_p , 5 , 1 ,
                           elem_p->gtcl1_md () ,
                           elem_p->gtcl2_md () , m_rad) ;
```

... und der Konstruktor **CGRod_D::CGRod_D** (vgl. Listing der Klassen-Deklaration im Abschnitt 5.7.1) reicht die beiden Farbwerte an den Konstruktor der Basisklasse **CGRod::CGRod** weiter.

Es lohnt sich also, in den Modell-Dateien Farb-Informationen unterzubringen. In der nebenstehenden Box ist die Datei **s_ikosa.dat** zu sehen, die solche Informationen enthält. Dabei ist folgendes zu beachten:

In der ersten Zeile, die mindestens 3 ganze Zahlen enthalten muß, sind nun 5 ganze Zahlen verzeichnet. Um die 5. Zahl einbringen zu können (Anzahl der Farbwerte pro Element, maximal: 2), muß auch die 4. vorgesehen werden, die die Anzahl der Koordinaten pro Knoten (Standard-Wert: 3) angibt.

Die beiden Farbwerte werden in den Zeilen, in denen die Elemente mit den beiden Knotennummern beschrieben werden, angehängt (in der Box sind die beiden Farbwerte fett hervorgehoben). Es sind nur die Zahlenwerte für die in der Klasse **CGBas** definierten Grundfarben erlaubt (Zahlenwerte 0,...,7, die Zuordnung zu den Farben wurde im Abschnitt 2.2.3 angegeben).

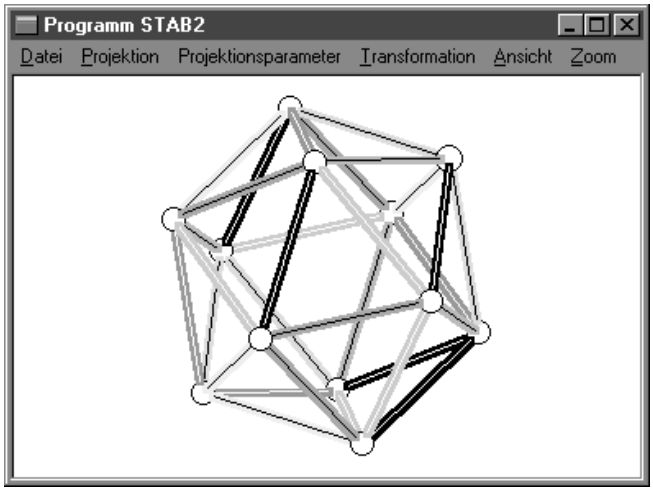
```
 30           12       2       3         2
  0            0                2.853169549
-2.064572881  -1.5              1.275976213
-2.064572881   1.5              1.275976213
 0.788593338   2.427050983      1.275976213
 2.551952425   0                1.275976213
 0.788593338  -2.427050983      1.275976213
-2.551952425   0               -1.275976213
-0.788593338   2.427050983     -1.275976213
 2.064572881   1.5             -1.275976213
 2.064572881  -1.5             -1.275976213
-0.788593338  -2.427050983     -1.275976213
 0            0                -2.853169549
 1  2      0  7
 2  3      1  7
 1  3      2  0
 3  4      3  0
 1  4      4  7
 4  5      5  0
 1  5      6  0
 5  6      0  7
 1  6      1  7
 2  6      2  0
 2  7      3  0
 7  3      4  7
 7  8      5  0
 3  8      6  0
 8  4      0  7
 8  9      1  7
 4  9      2  0
 9  5      3  0
 9 10      4  7
 5 10      5  0
10  6      6  0
10 11      0  7
 6 11      1  7
11  2      2  0
11  7      3  0
 7 12      4  7
 8 12      5  0
 9 12      6  0
10 12      0  7
11 12      1  7
```

Datei s_ikosa.dat

Der erste der beiden Farbwerte wird als **m_color1** (umgerechnet in einen COLORREF-Wert) im **CGElem**-Objekt abgelegt, der zweite dementsprechend als **m_color2**, und **m_color1** wird schließlich für das Zeichnen der äußeren Linie

mit **CGI::ptwdl_gi** verwendet, **m_color2** für das Zeichnen der inneren Linie. Für letztere sind in **s_ikosa.dat** Schwarz und Weiß vorgesehen, für die äußeren Linien sind alle Grundfarben verwendet worden, so daß es "schön bunt" wird.

Die nebenstehende Abbildung zeigt das Ikosaeder, wie es durch die beschriebene Datei erzeugt wird.



Ikosaeder, "schön bunt"

Man beachte: In dem Graphik-Modell, das in einer **CGMod**-Instanz verwaltet wird, sind in allen **CGElem**-Instanzen Farb-Informationen gespeichert (Werte aus der Datei oder Standard-Werte). Ob diese Werte in das für das Zeichnen zu erzeugende **CGObj**-Modell (Graphik-Objekte im binären Baum) übernommen werden, entscheidet der Programmierer, der z. B. auch vorsehen kann, daß die Werte über Dialoge vom Programm-Benutzer verändert werden können.

5.7.3 Zoom

Mit dem Programm **stab2.cpp** sind beinahe beliebig komplizierte Stabmodelle darstellbar. Um bei Modellen mit sehr vielen Stäben noch Details erkennen zu können, ist es erforderlich, in das Bild "hineinzoomen" zu können.

Die Strategie dafür wurde im Abschnitt 2.3.3 ausführlich beschrieben. Hier werden noch einmal alle Passagen gelistet, die im Programm **stab2.cpp** für das Zoomen verantwortlich sind, weil eine gegenüber dem Abschnitt 2.3.3 etwas geänderte Bedienung durch den Benutzer erwartet wird: Mit dem Drücken der linken Maustaste wird der erste Punkt festgelegt, danach wird das Rechteck **bei gedrückter linker Maustaste "aufgezogen"**, und der zweite Punkt wird durch das Lösen der linken Maustaste festgelegt.

- Nach Wahl des Menü-Angebots **Zoom** öffnet sich ein Pull-Down-Menü, in dem **Rechteck aufziehen** gewählt wird. Die dadurch ausgelöste WM_COMMAND-Botschaft wird an **CMainFrame::OnZoom** weitergeleitet:

```
void CMainFrame::OnZoom ()
{
   m_gi.scapp_gi (this , (m_xumin + m_xumax) * .5 ,
                         (m_yumin + m_yumax) * .5 , m_gi.GI_CUBIGCROSS) ;
   m_zoom  = 1 ;
}
```

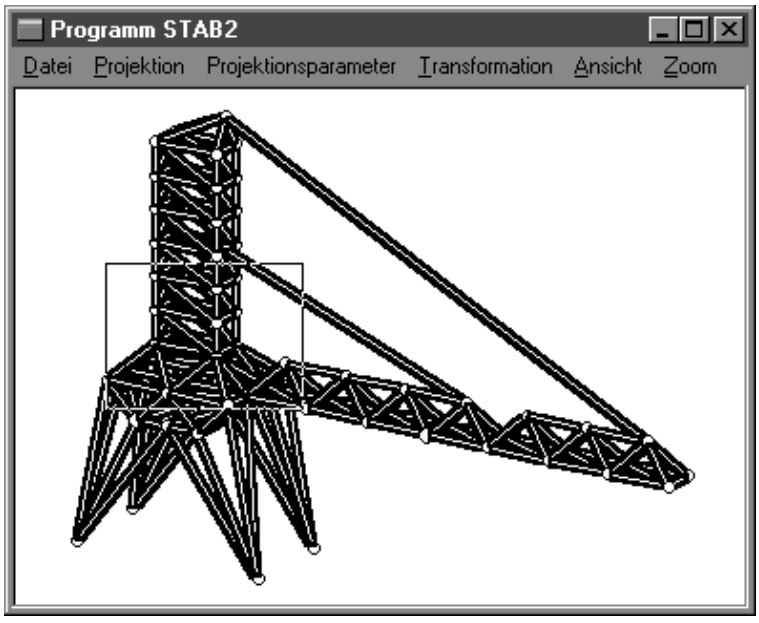
  Mit **CGI::scapp_gi** wird der Spezialcursor "Großes Kreuz" erzeugt. Hier wird im Gegensatz zum Abschnitt 2.3.3 die überladene Variante von **scapp_gi** verwendet, die "User coordinates" erwartet für den Punkt, an dem der Spezialcursor erscheinen soll (hier erscheint er in der Mitte der "Client area"). Die Variable **m_zoom** gehört zur

Klasse **CMainFrame**, wird in **CMainFrame::CMainFrame** mit dem Wert **0** initialisiert und zeigt nun mit dem Wert **1** an, daß eine Zoom-Aktion eingeleitet wurde.

- Wenn **m_zoom** ungleich **0** ist, wird die Botschaft WM_LBUTTONDOWN ausgewertet, die an **CMainFrame::OnLButtonDown** übergeben wird:

```
void CMainFrame::OnLButtonDown (UINT nFlags , CPoint point)
{
   if (m_zoom)
   {
      double x , y ;
      m_gi.rpick_gi (this , point , &x , &y , &x , &y) ;
      // Die gepickte Position wurde nicht gespeichert, weil rpick_gi
      // auch beim zweiten Aufruf (hier in OnLButtonUp) die erste
      // Position noch einmal abliefert.
      SetCapture () ;
      // ... "faengt" die Maus und gibt sie erst nach WM_BUTTONUP
      //     wieder frei
   }
}
```

Mit **CGI::rpick_gi** wird der Spezialcursor auf die Form "Rechteck" umgeschaltet. Die von **rpick_gi** gelieferten "User coordinates" des gepickten Punktes werden hier noch nicht gespeichert, weil sie beim Abschluß der Zoom-Aktion noch einmal angeliefert werden.



"Aufziehen des Rechtecks" mit gedrückter Maustaste

Im Gegensatz zu der im Abschnitt 2.3.3 beschriebenen Variante wird hier die Maus mit **SetCapture** "eingefangen". Damit werden **alle** Maus-Botschaften an das Fenster geliefert, das diese Funktion aufgerufen hat, auch wenn sie außerhalb des Fensters anfallen. Damit wird gesichert, daß das Lösen der linken Maustaste auf keinen Fall unbemerkt bleibt, auch wenn es (versehentlich oder absichtlich) über einem anderen Fenster geschieht. Im Abschnitt 2.3.3, wo im Programm **zoom.cpp** das Rechteck durch zweimaliges Drücken der linken Maustaste erzeugt wurde, war dies nicht erforderlich, weil nach dem Lösen der Taste auf den nächsten Maus-Klick im Fenster gewartet werden kann (der "Rechteck"-Cursor "wartet" gegebenenfalls, bis die Maus wieder im Fenster erscheint).

- Wenn die Variable **m_zoom** einen Wert ungleich **0** hat, wird auch die Botschaft WM_MOUSEMOVE ausgewertet, die an **CMainFrame::OnMouseMove** geleitet wird:

```
void CMainFrame::OnMouseMove (UINT nFlags , CPoint point)
{
   if (m_zoom) m_gi.scupd_gi (this , point) ;
   // ... aktualisiert Position des Spezialcursors
}
```

  Von **CGI::scupd_gi** wird der jeweils aktuelle Spezialcursor ("Großes Kreuz" oder "Rechteck") der aktuellen Maus-Position angepaßt.

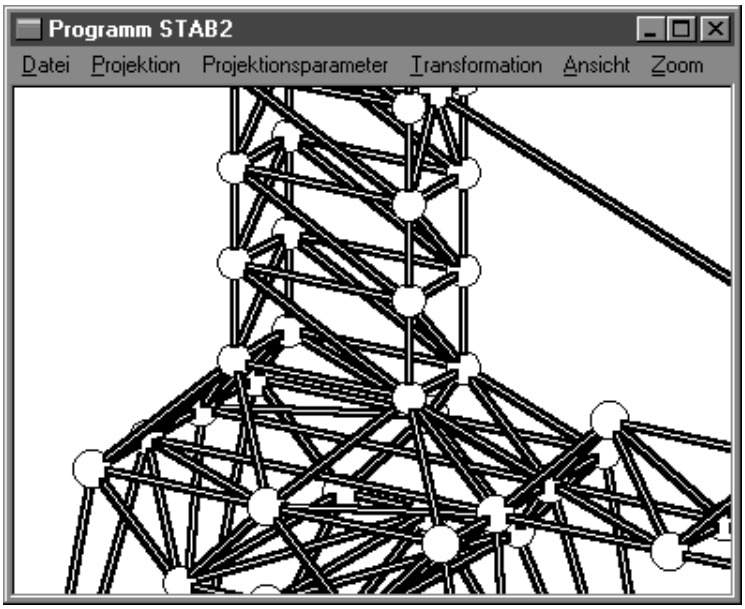
- Das Ende der Aktion wird beim Lösen der linken Maus-Taste eingeleitet. Die Botschaft WM_LBUTTONUP wird an die Methode **CMainFrame::OnLButtonUp** weitergeleitet:

```
void CMainFrame::OnLButtonUp (UINT nFlags , CPoint point)
{
   if (GetCapture () == this) ReleaseCapture () ;
   // ... gibt die "eingefangene" Maus wieder frei (mit GetCapture wird
   //     geprueft, ob dieses Fenster tatsaechlich der "Faenger" war)
   if (m_zoom)
   {
      m_zoom = 0 ;
      double x1 , y1 , x2 , y2 ;
      if (m_gi.rpick_gi (this , point , &x1 , &y1 , &x2 , &y2))
      {
         if (!m_gi.picid_gi ())
         {                                  // ... gibt es einen
             m_xumin = min (x1 , x2) ;      //     "gezoomten Bereich"
             m_yumin = min (y1 , y2) ;
             m_xumax = max (x1 , x2) ;
             m_yumax = max (y1 , y2) ;
             Invalidate () ;
         }
      }
      else
      {                           // ... wurde Maus-Button ausserhalb der
         m_gi.scdap_gi (this) ; //     Zeichenflaeche geloest, es soll
      }                          //     nichts geschehen ausser "Cursor
   }                             //     abschalten"
}
```

  Mit **ReleaseCpature** wird die "gefangene Maus" wieder freigegeben. **CGI::rpick_gi** liefert **beide** Eckpunkte des Rechtecks in "User coordinates" ab. Mit **CGI::picid_gi** wird (auf Pixel-Basis) geprüft, ob die beiden Punkte tatsächlich ein Rechteck beschreiben. Nur in diesem Fall werden die "User coordinates", die das Rechteck definieren, auf die entsprechenden Variablen der **CMainFrame**-Instanz übernommen. Mit **Invalidate** wird eine Botschaft WM_PAINT erzeugt, so daß unmittelbar nach der Zoom-Aktion in **CMainFrame::OnPaint** mit **CGI::stuci_gi** die neuen "User coordinates" definiert werden.

  Wenn die linke Maus-Taste außerhalb des Fensterbereichs gelöst wurde, kommt diese Botschaft trotzdem (wegen des "Einfangens") hier an. Mit **CGI::rpick_gi** wird dann allerdings festgestellt, daß kein gültiger Punkt angeliefert wurde. In diesem Fall wird mit **CGI::scdap_gi** der noch sichtbare Cursor abgeschaltet (wird bei einem gültigen Punkt von **CGI::rpick_gi** selbst erledigt).

♦ Das nebenstehende Bild zeigt das Ergebnis der Zoom-Aktion, die natürlich beliebig oft wiederholt werden kann. Deshalb muß bei einer Möglichkeit zum Zoomen auch immer die Chance bestehen, wieder das komplette Bild zu erzeugen. Dies wird über das Menü-Angebot **Zoom** und das im Popup-Menü offerierte **Optimal** erreicht. Die zugehörige WM_COMMAND-Botschaft wird an die **CMainFrame**-Methode **OnOptimal** geleitet:



Darstellung nach der Zoom-Aktion

```
void CMainFrame::OnOptimal ()
{
   m_gi.scdap_gi (this) ;       // Es koennte ein spezieller Cursor
                                // angeschaltet sein
   m_opt = 1 ;
   Invalidate () ;
}
```

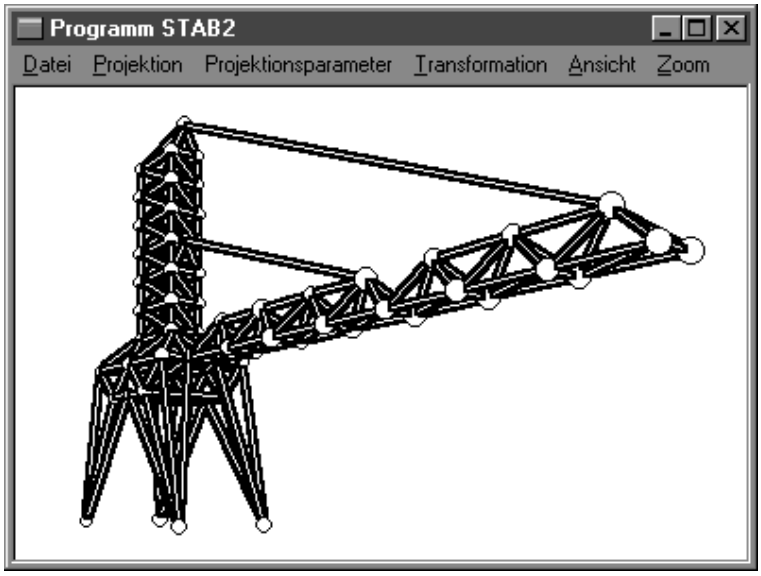
Hier zeigt sich eine gegenüber dem Programm **stab1.cpp** geänderte Strategie: Es wird nur der zur Klasse **CMainFrame** gehörende Indikator **m_opt** auf den Wert **1** gesetzt, das Ermitteln der zum optimalen Fenster gehörenden "User coordinates" wird in **CMainFrame::OnPaint** verlegt, weil es in verschiedenen Programmsituationen (z. B. auch beim Einlesen eines neuen Modells von einer Datei) erforderlich ist. Dort wird es in Abhängigkeit vom Indikator **m_opt** mit **CPT::ptmx3_pt** erledigt.

♦ Ansonsten sind noch folgende Ergänzungen am Programm vorgenommen worden, die allerdings exakt so wie im Programm **zoom.cpp** (Abschnitt 2.3.3) aussehen und hier nur aufgelistet werden:

- Das **CGI**-Objekt wird für ein "langes Leben" (Existenz des Hauptfensters) angelegt, indem es als Element in der Klasse **CMainFrame** angesiedelt wird.
- Alle genannten Botschaften müssen an die beschriebenen **CMainFrame**-Methoden geleitet werden. Die "MESSAGE_MAP" in **stab2.cpp** wurde entsprechend ergänzt.
- Alle beschriebenen **CMainFrame**-Methoden müssen in der Deklaration der Klasse **CMainFrame** mit Prototypen vertreten sein.

5.7.4 Zentralprojektion und Parallelprojektion

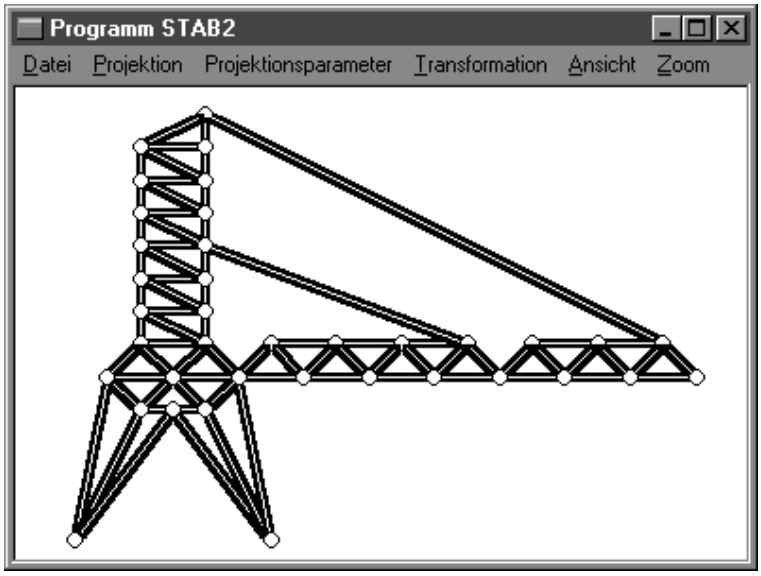
Das nebenstehend zu sehende (mit der Datei **s_kran.dat** erzeugte) Bild zeigt eine Zentralprojektion mit einem "Eye Point" etwa in Augenhöhe eines in der Nähe des Krans stehenden Betrachters. Man erkennt, daß die Zentralprojektion die räumliche Wirkung besonders gut hervorhebt.



Zentralprojektion, "Eye point" etwa dort, wo ein auf dem Erdboden stehender Betrachter die Augen hat

Mit der Zentralprojektion sind mit Abstand die größten Möglichkeiten der Darstellung des Objekts gegeben. Man kann sich auch "unter den Kran stellen" und ihn aus der "Frosch-Perspektive" betrachten und sogar "in ihn hineinkriechen". Die Probleme, die mit solchen Darstellungen verbunden sind, werden im folgenden Abschnitt beschrieben.

Aber bei allen Vorteilen und der Flexibilität der Zentralprojektion hat doch auch die Darstellung in Parallelprojektion ihre Stärken. Dort, wo die Verzerrungen der Zentralprojektion unerwünscht sind und vor allem für "technische Ansichten" ist die Parallelprojektion zu bevorzugen. Gerade für die Kontrolle der im Programm gespeicherten Modell-Daten sind die in Richtung der Koordinatenachsen gewählten "Blickrichtungen" häufig ein schneller Indikator für Koordinatenfehler oder falsche Zuordnung der Elemente zu den Knoten.

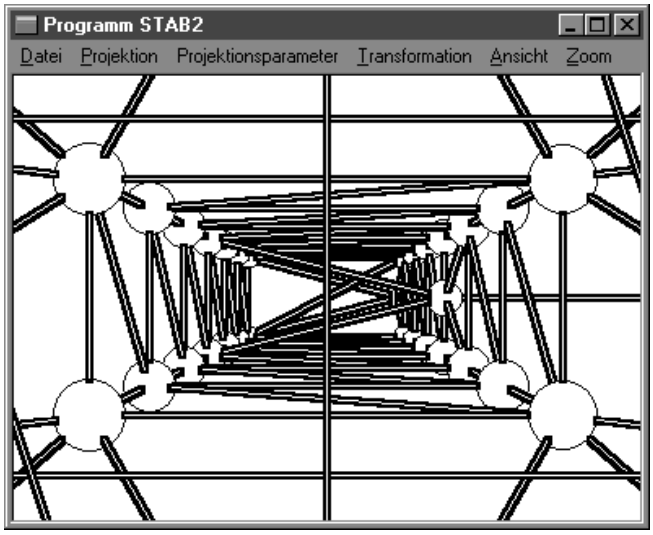


Seitenansicht in Parallelprojektion

In der nebenstehenden Abbildung sieht man, daß (zumindest in der betrachteten Richtung) "hintereinander liegt, was hintereinander gehört". Für die Daten-Kontrolle sollte auf diese "weniger attraktiven" Ansichten auf keinen Fall verzichtet werden.

5.7.5 Spezielle Probleme der Zentralprojektion, der "Kamera-Winkel"

Wenn der "Eye point" in ausreichender Entfernung von dem darzustellenden Objekt liegt, gibt es bei der Zentralprojektion eigentlich keine Probleme. Die Möglichkeit, den "Eye point" auch in das darzustellende Objekt hineinzulegen (vgl. nebenstehende Abbildung), verlangt einige besondere Maßnahmen, die der Programmierer zu beachten hat. In diesem Abschnitt werden die Probleme diskutiert und die Vorsichtsmaßnahmen beschrieben, die in den **CGIW**-Klassen eingebaut sind, um einerseits kritische Operationen zu vermeiden, andererseits möglichst mit jeder Einstellung "ein vernünftiges Bild" zu erzeugen.



Der Betrachter befindet sich "innerhalb des Objektes": Er steht zwischen den Füßen des Krans und blickt senkrecht nach oben

- Prinzipiell sind nur Punkte darzustellen, die im "Blickfeld" des Betrachters liegen. Eine natürliche Grenze ist durch eine Ebene gegeben, die durch den "Eye point" geht und parallel zur Projektionsebene liegt. Diese Ebene ("Vanishing plane") teilt den Raum in zwei Halbräume auf. Punkte, die in dem Halbraum liegen, in dem sich die Projektionsebene nicht befindet ("Forbidden half space") werden nicht dargestellt. Aber natürlich können auch Punkte, die in der "Vanishing plane" liegen, nicht dargestellt werden, weil der Sehstrahl parallel zur Projektionsebene liegt. Deshalb muß dafür gesorgt werden, daß nur Punkte jenseits einer zwischen Projektionsebene und "Vanishing plane" liegenden (und zu beiden parallelen) Ebene dargestellt werden. Letztere darf durchaus sehr nah an der "Vanishing plane" liegen und wird in der englischsprachigen Literatur als "Near clipping plane" oder "Hither clipping plane" bezeichnet.
- Aber weder der Mensch noch eine Kamera haben gleichzeitig alle Punkte des "erlaubten Halbraumes" im Blick. Deshalb sollten für einen realitätsnahen Eindruck nur Punkte aus einem Kegel berücksichtigt werden, dessen Spitze mit dem "Eye point" zusammenfällt und dessen Höhe die Projektionsebene senkrecht schneidet. Der Öffnungswinkel des Kegels ("Kamera-Winkel", "Field-of-vision-angle") bestimmt das Blickfeld. Dieser Kegel schneidet einen Kreis aus der Projektionsebene heraus. Da aber üblicherweise keine kreisförmigen Bilder angefertigt werden, wird ein Quadrat gewählt, dessen Seitenlängen dem Durchmesser des Kreises entsprechen. An die Stelle des beschriebenen Kegels tritt also eine vierseitige (quadratische) Pyramide (mit unendlicher Höhe).
- Vielfach ist es sinnvoll, auch sehr weit vom Betrachter entfernte Punkte nicht mehr darzustellen. Dann definiert man noch eine (ebenfalls zur Projektionsebene parallel

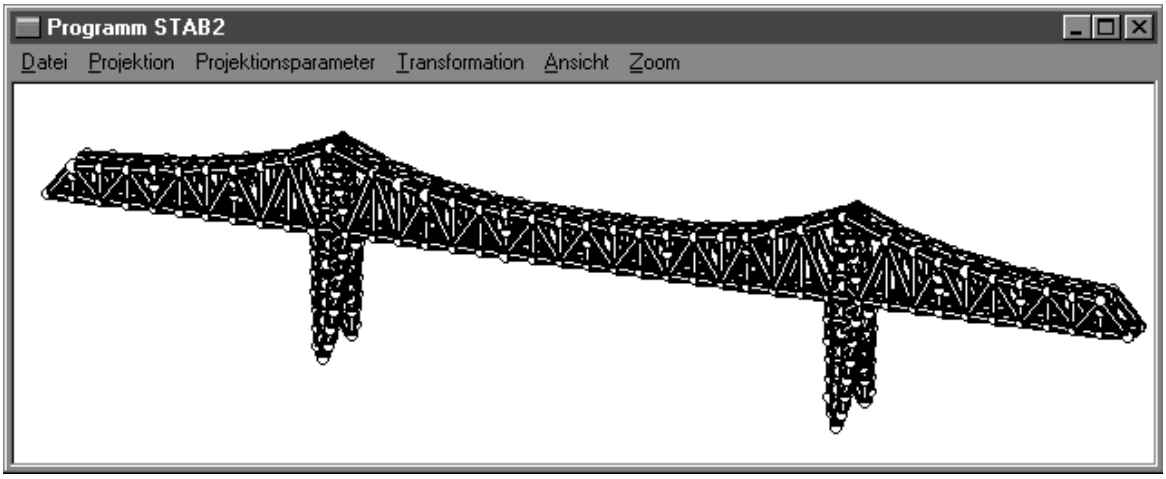
liegende) "Far clipping plane" ("Yon clipping plane"). Wenn nur Punkte abgebildet werden, die innerhalb der vierseitigen Pyramide und zwischen den beiden "Clipping planes" liegen, so ist der Raum, in dem diese Punkte liegen, ein quadratischer Pyramidenstumpf ("Viewing frustum").

- Der Test, ob ein Punkt innerhalb des "Viewing frustums" liegt, kann recht aufwendig sein. Der Aufwand wird noch dadurch erheblich vergrößert, daß Linien (es genügt, wenn man sich das nur für gerade Linien verdeutlicht), deren Endpunkte außerhalb des "Viewing frustums" liegen, durchaus zum Teil innerhalb liegen können, so daß Schnittpunkte berechnet werden müßten.

In den **CGIW**-Klassen sind einige Vorsichtsmaßnahmen getroffen worden, die (bei recht geringem Aufwand) die meisten der genannten Probleme umgehen, so daß der Programmierer in der Regel auf die (für ihn und das Programm) aufwendigen Algorithmen zur Realisierung der beschriebenen "3D-Clipping-Strategie" verzichten kann:

- Zur Klasse **CPT** gehört der "Kamera-Winkel" als **double**-Wert (Bogenmaß). Er kann mit **CPT::prgta_pt** abgefragt werden und mit **CPT::prsta_pt** auf einen anderen Wert gesetzt werden. Er wird im Konstruktor **CPT::CPT** (via **CPT::prini_pt**) mit **π/4** vorbelegt. Dieser Wert (45°) ist für die meisten Darstellungen ausreichend, man bedenke, daß eine normale Kleinbildkamera (24x36-mm-Negative) mit einem gängigen 35-mm-Objektiv einen "Kamera-Winkel" von nur 38° hat.
- Der "Kamera-Winkel" wird nur in **CPT::ptmx3_pt** (vgl. Abschnitt 5.3) benutzt. Mit dieser Funktion werden die Maxima und Minima der 2D-"User coordinates" berechnet, die sich aus allen 3D-"World coordinates"-Punkten ergeben, wenn auf diese die aktuelle Transformation und die eingestellte Projektion angewendet werden. Zwei Einschränkungen werden zusätzlich gemacht:
  - Punkte des "Forbidden Halfspace" werden überhaupt nicht berücksichtigt, ebensowenig Punkte, deren Abstand vom "Eye point" kleiner als 1/1000 der Strecke auf dem Sehstrahl vom "Eye point" bis zur Projektionsebene ist (die also "fast im Auge liegen").
  - Nachdem die Minima und Maxima aller Punkte in der Projektionsebene berechnet wurden, wird das so ermittelte Rechteck gegebenenfalls noch verkleinert auf die Abmessungen, die der "Kamera-Winkel" vorgibt, so daß die von **CPT::ptmx3_pt** zurückgegebenen Werte nicht den Bereich des Quadrats überschreiten, den das "Viewing frustum" aus der Projektionsebene herausschneidet.
- Die "zeichnenden" **CGI**-Methoden zeichnen nicht, wenn eine der ihnen übergebenen Koordinaten im "Forbidden halfspace" oder so nah wie oben beschrieben am "Eye point" liegt.

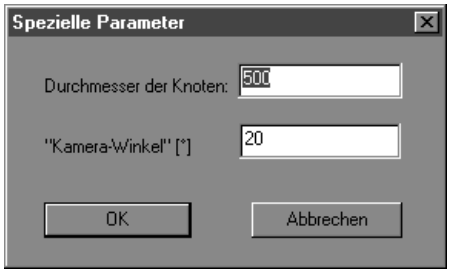
Das "Clipping" wird also in den 2D-Bereich verlegt (und ist damit nicht sonderlich aufwendig). Linien, deren Endpunkte außerhalb des "Viewing frustums" liegen, sind also mit einem eventuell sichtbaren Anteil durchaus zu sehen. Es fehlen nur die Linien, bei denen ein Punkt im "Forbidden halfspace" (oder "beinahe im Auge") liegt und die trotzdem einen Teil innerhalb des "Viewing frustums" haben. Diese seltenen Ausnahmen rechtfertigen aber kaum eine aufwendige Schnittpunktberechnung.



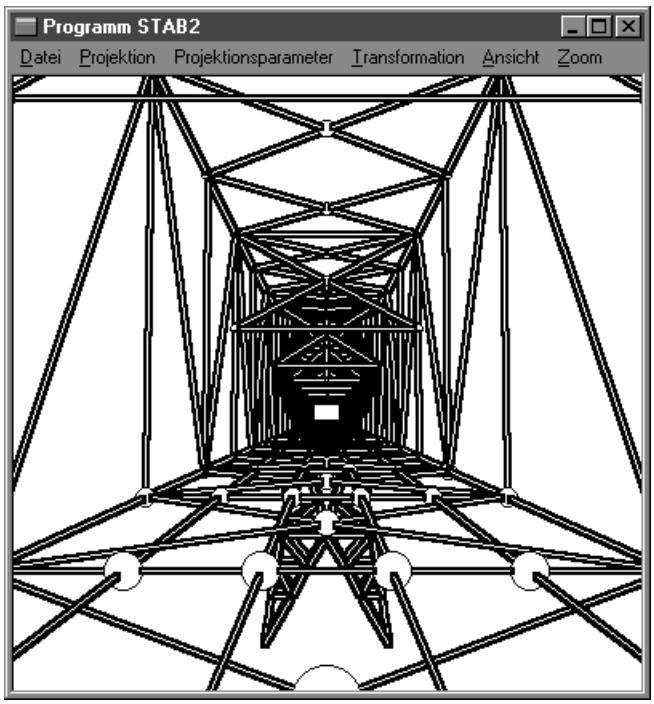
Tragkonstruktion der Eisenbahnbrücke über den Nord-Ostsee-Kanal bei Rendsburg

Wenn man die oben dargestellte Tragkonstruktion einer Brücke[2] aus der "Sicht des Lokführers" betrachtet, der über die Brücke fährt, ergibt sich das unten rechts zu sehende Bild.

Um den Einfluß des "Kamera-Winkels" auf die Darstellung ausprobieren zu können, kann dieser über das Menü-Angebot **Ansicht** geändert werden. Es öffnet sich die unten links zu sehende Dialog-Box. Weil Ansichten wie die rechts dargestellte häufig mit einem kleineren "Kamera-Winkel" besser wirken, wurde ein Winkel von

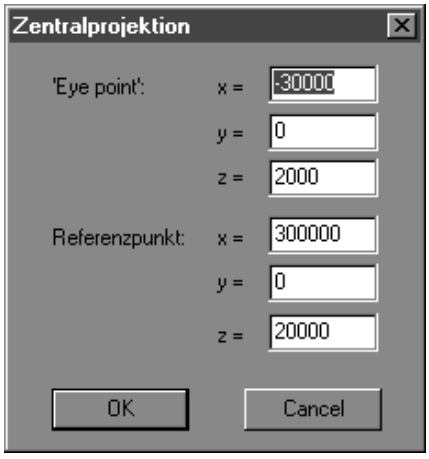


Nach Wahl des Menü-Angebots "Ansicht" öffnet sich eine Dialog-Box: Der "Kamera-Winkel" wird geändert

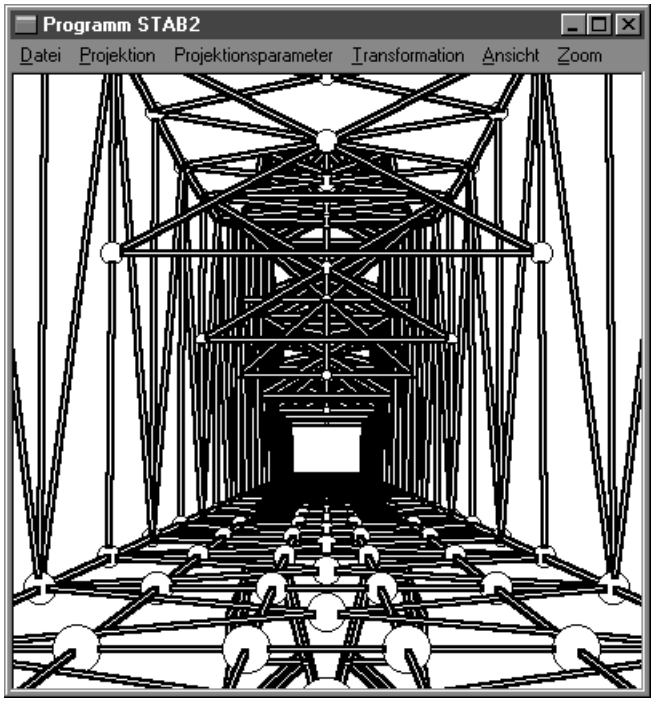


"Sicht des Lokführers" ("Kamera-Winkel": 45°)

[2]Das Modell wurde nach Original-Zeichnungen von den Studenten Holger Glüß und Karsten Ullrich im Rahmen einer Semesterarbeit im Fach Numerische Methoden (3. Semester) für eine Festigkeitsberechnung nach der Finite-Elemente-Methode erstellt. Das war sicher eine recht mühsame (aber durchaus erfolgreiche) Angelegenheit.



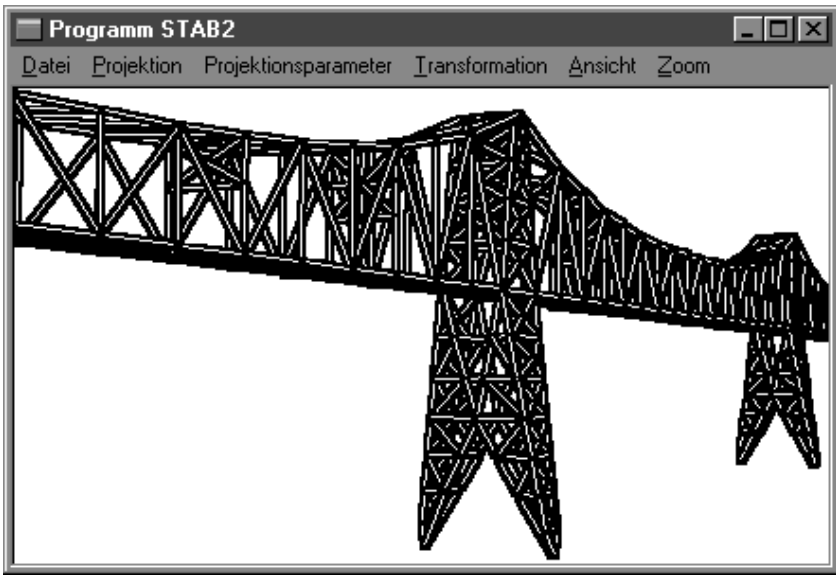
Beide Bilder, die die Brücke aus der "Sicht des Lokführers" zeigen, wurden mit den in der Dialog-Box zu sehenden Einstellungen für die Zentralprojektion erzeugt (alle Angaben in mm).



"Sicht des Lokführers" ("Kamera-Winkel": 20°)

nur 20° gewählt. Damit ergibt sich die oben rechts zu sehende Darstellung (links sieht man die Projektions-Parameter, die Koordinaten in der Datei **s_rendsb.dat** sind in mm angegeben, deshalb wurde auch für die Paramter diese Längeneinheit gewählt).

Während bei der Wahl von "Eye points", die innerhalb des Objekts liegen, kleinere "Kamera-Winkel" im allgemeinen zu schöneren Bildern führen, darf man für "Außen-Ansichten" durchaus mit dem voreingestellten Wert (45°, nebenstehende Abbildung) arbeiten, eventuell sogar etwas größeren Werten, was dann jedoch zu deutlichen Verzerrungen bei den Punkten in der Nähe des "Eye points" führen kann.



Ansicht vom Kanalufer ("Kamera-Winkel": 45°)