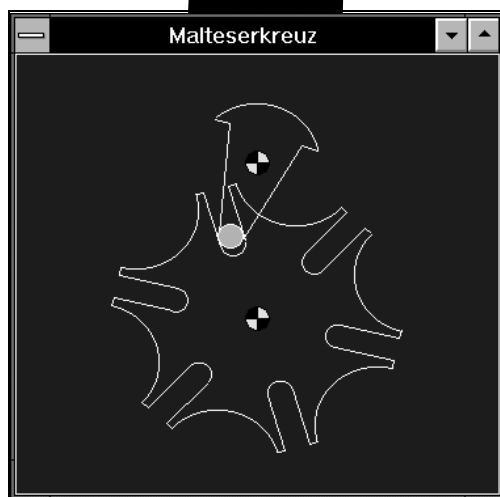
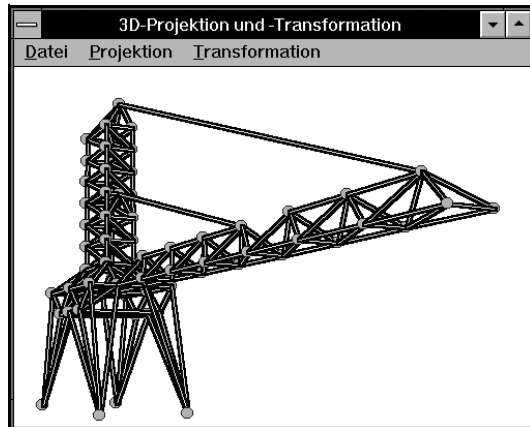


# Graphik-Routinen für die C-Programmierung unter MS-Windows 3.1, Windows 95 und Windows NT



**Library GIW.LIB**  
**\*\* Version 1.0 \*\***

Die Library **GIW.LIB** enthält C-Funktionen, die das Arbeiten mit dem Entwicklungssystem MS-Visual-C++ unterstützen (für die C++-Programmierung existiert eine Klassen-Bibliothek mit gesonderter Dokumentation) .

Die Library und die in der Dokumentation erwähnten Beispiel-Programme können **für den Gebrauch in Ausbildung und Lehre** über die Internet-Adresse

**<http://www.fh-hamburg.de/rzbt/dankert/giw.html>**

frei kopiert werden.

# Inhalt

<b>1</b>	<b>Vorbemerkungen</b>	<b>1</b>
<b>2</b>	<b>Koordinatensysteme</b>	<b>1</b>
2.1	Bearbeiten der Botschaft WM_PAINT, das GIW-Viewport-Konzept	2
2.2	Problembezogene Koordinatensysteme	5
2.2.1	"User Coordinates" mit anisotroper Skalierung	6
2.2.2	"User Coordinates" mit isotroper Skalierung	8
2.2.3	"Gefüllte Flächen" mit "User Coordinates"	9
2.2.4	Der Versuch, "pixelgenaue Anschlüsse" zu realisieren	11
2.2.5	Marker an "User Coordinates"-Positionen	12
2.3	"User Coordinates"-Punkte picken, Zoom	13
<b>3</b>	<b>Transformationen und Projektionen</b>	<b>19</b>
3.1	Homogene Koordinaten	19
3.2	Ebene Transformation	20
3.2.1	Ebene geometrische Transformation mit homogenen Koordinaten	21
3.2.2	Ebene Koordinatentransformation mit homogenen Koordinaten	22
3.2.3	Verknüpfung von Transformationen	23
3.3	Die "t...-Funktionen" des GIW	24
3.4	Projektionen	31
3.4.1	Allgemeine Theorie der Zentralprojektion	31
3.4.2	Allgemeine Theorie der Parallelprojektion	34
3.4.3	Empfehlungen zur Definition von Projektionen	35
3.4.4	Vereinfachte Definition der Projektionen im GIW	36
3.4.5	Die "pr...-Funktionen" des GIW	37
3.5	3D-Transformationen	40
3.5.1	Geometrische Transformation mit homogenen Koordinaten	41
3.5.2	Koordinatentransformation mit homogenen Koordinaten	42
3.5.3	"t3...-Funktionen" und "pt...-Funktionen"	43
3.6	Darstellung von "Drahtmodellen"	45
3.7	"Breite zweifarbige Linien"	50
3.8	"Hidden Lines" und "Hidden Surfaces"	51
3.9	Darstellung von "Stabmodellen"	53
3.10	3D-Polygonflächen	57
<b>4</b>	<b>Header-Datei und GIW-Funktionen</b>	<b>61</b>
4.1	GIW-Funktionen	61
4.2	Header-Datei giw.h	64
4.3	GIW-Funktionen in alphabetischer Reihenfolge	67

# 1 Vorbemerkungen

Die **GIW-Toolbox** ist für die Windows-C-Programmierung mit dem Entwicklungssystem Microsoft-Visual-C++ konzipiert (C++-Programmierer sollten die Klassen-Bibliothek **CGI** benutzen, für die eine gesonderte Dokumentation verfügbar ist). In dieser Dokumentation werden zunächst einige Basis-Informationen gegeben, die für das Arbeiten mit der GIW-Toolbox erforderlich sind, im Kapitel 4 werden die Interface-Routinen, die der C-Programmierer direkt aufrufen kann, beschrieben. Die Funktionen werden durch den Anfangskommentar dokumentiert, der direkt aus dem Quelltext herausgelöst wurde.

Die GIW-Toolbox versteht sich als Ergänzung zu den GDI-Routinen ("Graphics Device Interface") des Windows-API ("Application Interface"), die Funktionen des GIW können gemischt mit den GDI-Funktionen verwendet werden.

Als Demonstrations-Beispiele für die Verwendung der **GIW-Toolbox** dienen einfache Programme, die im Quelltext verfügbar sind (und in Ausschnitten in dieser Dokumentation abgedruckt werden).

# 2 Koordinatensysteme

Das Windows-GDI stellt dem Programmierer insgesamt 8 verschiedene Koordinatensysteme zur Verfügung, ein für die Bedürfnisse von Ingenieuren und Naturwissenschaftlern brauchbares ist leider nicht dabei (eine ausführlichere Betrachtung hierzu findet sich im Abschnitt 14.4.13 im Teil 4 von "J. Dankert: C und C++ für UNIX, DOS und MS-Windows 3.1/95/NT"). Alle GDI-Funktionen, die sich auf die GDI-Koordinatensysteme beziehen, erwarten die Koordinaten als **int**-Argumente, eine Einschränkung, die sich bei der Darstellung mathematischer Funktionen und technischer Objekte als ausgesprochen lästig erweist.

Die GIW-Toolbox arbeitet intern ausschließlich mit dem GDI-Koordinatensystem **MM\_TEXT**, stellt aber dem Programmierer Routinen für das Arbeiten mit unterschiedlichen Koordinatensystemen zur Verfügung:

- ◆ Die Routinen, die mit "**Viewport Coordinates**" arbeiten, erwarten **int**-Argumente, die als "Geräte-Einheiten" interpretiert werden (für die Bildschirm-Ausgabe: Pixel).
- ◆ Die Routinen, die mit "**User Coordinates**" arbeiten, erwarten **double**-Argumente, die in Abhängigkeit von der Definition dieser Koordinaten einen rechteckigen Zeichenbereich isotrop (mit gleicher Skalierung in beiden Richtungen) oder anisotrop (mit unterschiedlicher Skalierung in beiden Richtungen) auf den Viewport abbilden.
- ◆ Die Routinen, die mit "**World Coordinates**" arbeiten, erwarten **double**-Argumente, die Punkte in einem dreidimensionalen kartesischen Koordinatensystem beschreiben, die von den Funktionen des GIW vor dem Zeichnen entsprechend einer einzustellenden Projektion in die zweidimensionale Zeichenfläche projiziert werden.

## 2.1 Bearbeiten der Botschaft WM\_PAINT, das GIW-Viewport-Konzept

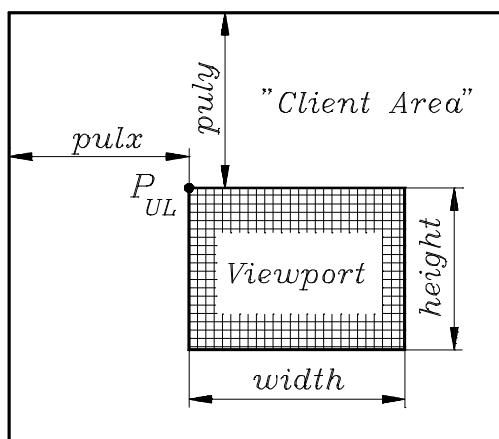
Bei der Bearbeitung der Windows-Botschaft **WM\_PAINT** werden die Zeichenaktionen üblicherweise von den Aufrufen der Funktionen **BeginPaint** und **EndPaint** eingerahmt. Beim Arbeiten mit den **GIW**-Funktionen ist der Aufruf von **BeginPaint** durch den Aufruf von **gstrt\_gi** zu ersetzen, an die Stelle von **EndPaint** tritt **gstop\_gi**. Diese beiden Funktionen übernehmen die Aufrufe von **BeginPaint** bzw. **EndPaint** und initialisieren die GIW-Parameter bzw. "räumen im GIW auf".

Der Funktion **gstrt\_gi** werden als Argumente ein "Handle auf das Window", in das gezeichnet werden soll, und die Abmessungen der Zeichenfläche in Geräte-Koordinaten übergeben. Bei der Bearbeitung der Botschaft **WM\_PAINT** ist ein "Window-Handle" als Parameter der Fenster-Funktion verfügbar, die Abmessungen der Zeichenfläche können z. B. mit **GetClientRect** ermittelt oder bei der Bearbeitung der Botschaft **WM\_SIZE** mit den Makros **LOWORD** und **HIWORD** aus dem zu dieser Botschaft gehörenden Parameter herausgefiltert und in einer **static**-Variablen gespeichert werden.

In **gstrt\_gi** wird die gesamte Zeichenfläche als "Current Viewport" festgelegt (das Geräte-Koordinatensystem liegt in der linken oberen Ecke des Viewports), mit **stcvp\_gi** kann ein beliebiges rechteckiges Teilgebiet der Zeichenfläche zum "Current Viewport" erklärt werden.

Ein **GIW-Viewport** ist ein rechteckiger Teilbereich der Zeichenfläche, der

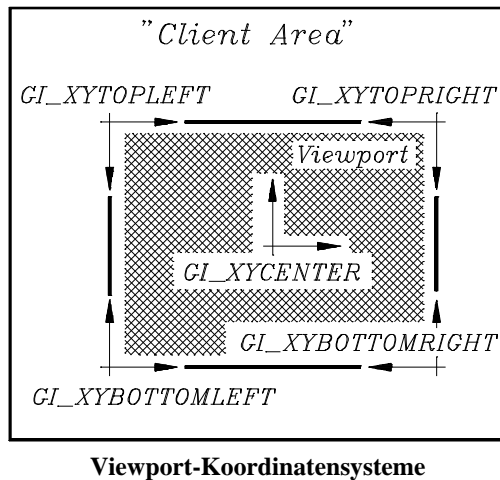
- ◆ durch die **Geräte-Koordinaten seiner linken oberen Ecke** (bezüglich der linken oberen Ecke der Zeichenfläche) und
- ◆ seine **Breite und Höhe** (ebenfalls in Geräte-Koordinaten) definiert wird.
- ◆ Der Rechteck-Bereich des Viewports begrenzt die Ausgabe der nachfolgenden Zeichenaktionen, ist also gleichzeitig "**Clipping-Bereich**".
- ◆ Für die **GIW**-Funktionen, die mit "Viewport Coordinates" arbeiten, wird bei der Definition eins von **5 möglichen Koordinatensystemen** festgelegt.



Viewport in der "Client Area"

Die nebenstehende Abbildung zeigt die beiden Abmessungen (in Geräte-Koordinaten), mit denen der "Upper Left Point"  $P_{UL}$  des Viewports festgelegt wird, und die beiden Abmessungen des Viewports. Diese vier Werte müssen als Argumente beim Aufruf von **stcvp\_gi** übergeben werden.

Das fünfte Argument, das **stcvp\_gi** übergeben werden muß (Typ des Viewport-Koordinatensystems), legt fest, mit welchem Koordinatensystem die nachfolgenden "v...-Routinen" (Funktionen, die sich auf die Viewport-Koordinaten beziehen) arbeiten sollen.



Es stehen fünf Viewport-Koordinatensysteme zur Wahl, der Koordinaten-Ursprung liegt entweder in einer Ecke des Viewports oder in der Viewport-Mitte. In der Header-Datei **giw.h**, die in die Programme aufgenommen werden muß, die mit **GIW**-Funktionen arbeiten, sind dafür sinnvolle Konstanten-Namen definiert. Die nebenstehende Abbildung zeigt die Lage und die Richtung der Achsen für die Koordinatensysteme.

Der Return-Wert der Funktion **gstrt\_gi** ist ein "Handle auf einen Device Context" (ermittelt mit dem Aufruf von **BeginPaint**). Der "Device Context" kann mit den GDI-Routinen manipuliert werden, und mit dem Handle können sowohl die

Zeichen-Routinen des Windows-GDI als auch die hier vorgestellten Routinen der GIW-Toolbox aufgerufen werden. Dabei ist nur folgender Unterschied zu beachten, der am Beispiel der "Line To"-Routinen (Zeichnen einer geraden Linie von der "Current Position" bis zu einer angegebenen Position) demonstriert werden soll:

Sowohl die GDI-Funktion

```
LineTo (HDC hdc , int x , int y)
```

als auch die GIW-Funktion

```
vline_gi (HDC hdc , int xv , int yv)
```

erwarten neben dem "Handle auf den Device Context" zwei **int**-Werte, die Koordinaten des Zielpunktes. Die **GDI**-Funktion **LineTo** bezieht diese auf das in **gstrt\_gi** eingestellte Koordinatensystem **MM\_TEXT**, während **vline\_gi** die Koordinaten auf das mit dem Aufruf von **stcvp\_gi** eingestellte Koordinatensystem bezieht.

Das Beispiel-Programm **viewpt.c** demonstriert die typische Bearbeitung der Botschaft **WM\_PAINT** mit den GIW-Funktionen und das GIW-Viewport-Konzept: Mit **gstrt\_gi** und **gstop\_gi** werden alle Zeichenaktionen "eingerahmt", **stcvp\_gi** legt einen "Current Viewport" in der Zeichenfläche fest und legt das Viewport-Koordinatensystem in die Viewport-Mitte, die mit **vmove\_gi** und **vline\_gi** ausgeführten Zeichenaktionen beziehen sich dann auf dieses Koordinatensystem und werden an den Viewport-Rändern "geclippt". In jeden Viewport wird eine "Rosette" gezeichnet (gleichmäßig über einen Kreis verteilte Punkte werden so miteinander verbunden, daß von jedem Punkt zu jedem anderen eine gerade Linie existiert). Dabei kann das Verhältnis des Kreisdurchmessers zur kleineren Viewport-Abmessung über einen Dialog verändert werden, so daß für Kreise, die nicht komplett in den Viewport passen, das Clipping an den Viewport-Rändern deutlich wird.

Nachfolgend wird der Quellcode, der in der Fenster-Funktion des Programms **viewpt.c** für die Bearbeitung der Botschaft **WM\_PAINT** zuständig ist, gelistet:

```
case WM_PAINT :
```

```
    hdc = gstrt_gi (hwnd , cxClient , cyClient) ;

    for (ix = 0 ; ix < nViewpx ; ix++)
        for (iy = 0 ; iy < nViewpy ; iy++)
        {
```

```

stcvc_gi (hdc , (int) (cxClient * ix / nViewpx) ,
          (int) (cyClient * iy / nViewpy) ,
          (int) (cxClient / nViewpx) ,
          (int) (cyClient / nViewpy) , GI_XYCENTER) ;
/* ... definiert den "Current Viewport" mit einem (Pixel-)Viewport-
   Koordinatensystem in Viewportmitte */

Radius = (cxClient / nViewpx > cyClient / nViewpy) ?
          cyClient / nViewpy : cxClient / nViewpx ;
Radius = Radius * QuotDiamLowDist / 2 ;
dPhi   = atan (1.) * 8. / nPoints ;      /*      pi * 2 / nPoints      */

if (Radius > 10000.) Radius = 10000. ; /* Das ist eine brutale, aber
   immerhin wirksame Moeglichkeit, Ueberlauf bei der unvermeidlichen
   Konvertierung auf Integer-Koordinaten zu vermeiden, bessere Moeg-
   lichkeiten findet man in nachfolgenden Demonstrationsprogrammen */

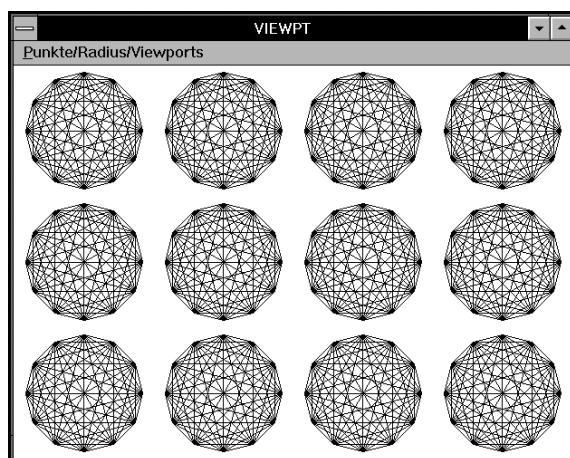
for (i = 0 ; i < nPoints ; i++)
{
    xs = (int) (Radius * cos (dPhi * i) + .5) ;
    ys = (int) (Radius * sin (dPhi * i) + .5) ;

    for (j = 0 ; j < nPoints ; j++)
    {
        if (j != i)
        {
            vmove_gi (hdc , xs , ys) ;
            vline_gi (hdc , (int) (Radius * cos (dPhi * j) + .5) ,
                      (int) (Radius * sin (dPhi * j) + .5)) ;
        }
    }
}

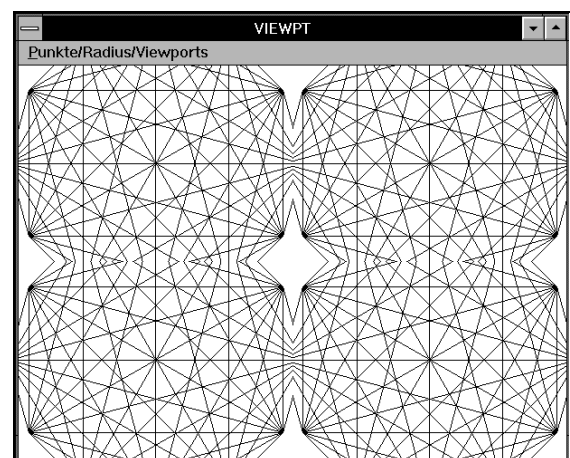
gstop_gi (hwnd) ;
return 0 ;

```

Nachfolgend sind die Fenster für zwei unterschiedliche Viewport-Konstellationen zu sehen. In der linken Abbildung wurden 12 Viewports eingestellt, das Verhältnis von Kreisdurchmesser und kleinerer Viewport-Abmessung ist kleiner als 1, so daß die Zeichnungen in die Viewports "passen". In der rechten Abbildung sind nur 4 Viewports eingestellt, das Verhältnis von Kreisdurchmesser zu kleinerer Viewport-Abmessung ist größer als 1, an den Viewport-Rändern wird "geclipt":



Zeichnungen "passen" in die Viewports



"Clipping" an den Viewport-Rändern

## 2.2 Problembezogene Koordinatensysteme

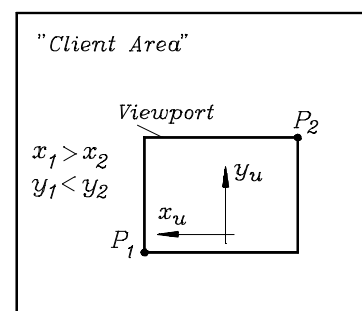
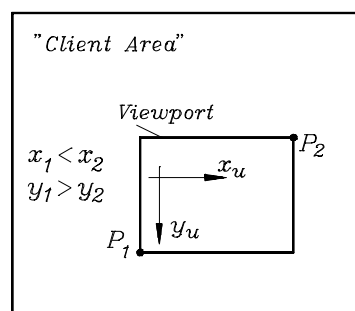
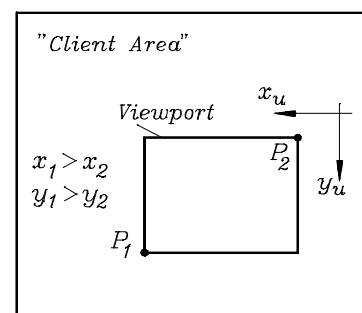
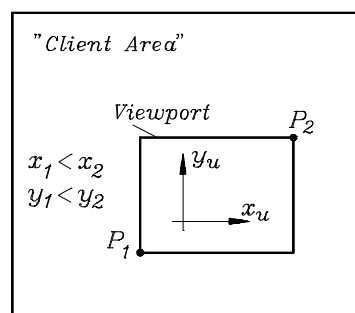
Als Ergänzung zu den (ausschließlich mit **int**-Werten arbeitenden) Koordinatensystemen des Windows-GDI und den im vorigen Abschnitt vorgestellten "Viewport Coordinates" des GIW werden problembezogene Koordinaten definiert, im GIW bezeichnet als

### "User Coordinates":

Für den "Current Viewport" (festgelegt durch **stcvp\_gi** bzw. **gstvt\_gi**) wird ein Koordinatensystem durch Angabe der Koordinaten zweier Punkte  $P_1$  und  $P_2$  definiert.  $P_1$  ist stets die **LINKE UNTERE**,  $P_2$  die **RECHTE OBERE** Ecke eines rechteckigen Zeichenbereichs. Zunächst darf angenommen werden, daß diese beiden Punkte mit den entsprechenden Ecken des "Current Viewport" identisch sind (tatsächlich gilt dies nur für "anisotrope Skalierung ohne Rand").

- ♦ Die "User Coordinates" werden mit **double**-Werten definiert, alle Zeichenroutinen, die mit diesem Koordinatensystem arbeiten (die Namen dieser GIW-Funktionen beginnen mit **u**), erwarten **double**-Koordinaten.
- ♦ Für  $P_1$  und  $P_2$  dürfen beliebige Werte festgelegt werden. Der Ursprung des Koordinatensystems kann dabei durchaus außerhalb des "Current Viewport" liegen. Da die Koordinatenachsen natürlich immer so gerichtet sind, daß sie von kleineren zu größeren Werten zeigen, können mit den Koordinaten der Definitions-Punkte  $P_1$  und  $P_2$  alle vier Kombinationen realisiert werden (wenn die beiden  $P_1$ -Koordinaten kleiner sind als die entsprechenden  $P_2$ -Koordinaten, zeigt die  $x$ -Achse nach rechts, und die  $y$ -Achse ist nach oben gerichtet).

Die nebenstehende Abbildung zeigt die vier Möglichkeiten, die sich für die Richtungen der Achsen der "User Coordinates" ergeben können. Alle nachfolgenden Aufrufe von "**u**...-Funktionen" beziehen sich dann auf diese Koordinatensysteme. Das oben rechts dargestellte Beispiel zeigt ein Koordinatensystem, dessen Ursprung (in diesem Fall durch Angabe ausschließlich positiver Koordinaten für  $P_1$  und  $P_2$ ) außerhalb des "Current Viewport" liegt.



## 2.2.1 "User Coordinates" mit anisotroper Skalierung

Mit der GIW-Funktion **stuca\_gi**, der 5 **double**-Argumente übergeben werden müssen, werden die "User Coordinates" so festgelegt, daß sie sich einem beliebigen Breiten-Höhen-Verhältnis des Viewports anpassen, so daß sich in der Regel unterschiedliche Skalierungen für die beiden Koordinatenrichtungen ergeben. Dies ist z. B. sinnvoll für die Darstellung von Diagrammen, bei denen die beiden Achsen Werte mit unterschiedlichen Dimensionen repräsentieren (Weg-Zeit-Funktion, Druck-Volumen-Diagramm, ...).

Wenn auf die Vorgabe eines "Randes" verzichtet wird, werden mit dieser anisotropen Skalierung beide Abmessungen des Viewports voll ausgenutzt, **stuca\_gi** wird mit den 4 **double**-Werten der Koordinaten der Punkte  $P_1$  und  $P_2$  aufgerufen, für den "Rand-Parameter" **pmarg** wird der Wert **0.** übergeben. In diesem Fall ist es häufig sinnvoll, den durch die beiden Punkte definierten Rechteck-Bereich etwas größer zu wählen als die maximal zu erwartenden Koordinaten für die Zeichen-Routinen.

Wenn **pmarg** ungleich Null vorgegeben wird (sinnvoll sind nur positive Werte), wird die Distanz der "User Coordinates" der beiden Punkte  $P_1$  und  $P_2$  für die **kleinere Viewport-Abmessung** auf jeder Seite um **pmarg** Prozent vergrößert, und für die größere Viewport-Abmessung wird ein Rand gleicher Breite eingestellt, so daß  $P_1$  und  $P_2$  nicht mehr in den Viewport-Ecken liegen. Man beachte, daß in die so eingestellten Ränder nur dann nicht hineingezeichnet wird, wenn die Graphik-Ausgabe innerhalb des durch  $P_1$  und  $P_2$  definierten Rechtecks bleibt, weil das "Clipping" ausschließlich durch die Viewport-Grenzen festgelegt wird.

Das Beispiel-Programm **usrcoor1.c** demonstriert das Arbeiten mit "User Coordinates" (Funktion **stuca\_gi** für das Definieren der Koordinaten und die Funktionen **umove\_gi** und **uline\_gi**, die mit diesen Koordinaten arbeiten). Es wird die Funktion

$$y = 4 e^{-x/5} \cos 3x$$

dargestellt. Dabei kann in mehreren Viewports gearbeitet werden, so daß alle Varianten der Koordinatenrichtungen und auch das Vorgeben einer Rand-Einstellung möglich sind. Nachfolgend wird der Quellcode, der in der Fenster-Funktion des Programms die Botschaft **WM\_PAINT** bearbeitet, gelistet:

```
static int cxClient , cyClient /* .. werden bei WM_SIZE-Bearbeitung gesetzt */
double    x , pmarg = 0. ;
int       width , height , ix , iy ;
HDC       hdc ;                /* ... */

case WM_PAINT :

    hdc = gstrt_gi (hwnd , cxClient , cyClient) ;
    width = cxClient / nViewpx - 2 ; /* ... etwas kleiner fuer die Rahmen */
    height = cyClient / nViewpy - 2 ; /*      um die Viewports          */

    for (iy = 0 ; iy < nViewpy ; iy++)
    {
        for (ix = 0 ; ix < nViewpx ; ix++)
        {
            stcvp_gi (hdc , (int) (cxClient * ix / nViewpx) + 1 ,
                      (int) (cyClient * iy / nViewpy) + 1 ,
                      width , height , GI_XYBOTTOMLEFT) ;
            /* ... definiert den "Current Viewport" mit einem (Geraete-)
               Viewport-Koordinatensystem in der linken unteren Ecke      */
        }
    }
```



```

vrect_gi (hdc , 0 , 0 , width - 1 , height - 1 ) ;
/* ... zeichnet einen Rahmen um den Viewport */

if (ix == 0) stuca_gi ( 0. , -3.5 , 10.5 , 4.5 , pmarg) ;
/* ... definiert "User Coordinates": Punkt (0;-3.5) wird auf die
linke untere Viewport-Ecke gelegt, Punkt (10.5;4.5) auf
die rechte obere Ecke (da die Werte keinen Bezug auf die
Viewport-Abmessungen nehmen, werden die beiden Achsen
im Regelfall unterschiedlich skaliert), im Durchlauf mit
ix=0 (obere Reihe) gilt pmarg=0. (kein Rand), danach
pmarg=10. */
if (ix == 1) stuca_gi ( 0. , 4.5 , 10.5 , -3.5 , pmarg) ;
if (ix == 2) stuca_gi (10. , 4.5 , 0. , -3.5 , pmarg) ;
if (ix >= 3) stuca_gi (10. , -3.5 , 0. , 4.5 , pmarg) ;
/* ... und in den ersten vier Viewports einer Reihe sind alle
vier Kombinationen fuer die Richtungen der beiden
Koordinatenachsen realisiert */

umove_gi (hdc , 0. , 4.5) ;
uline_gi (hdc , 0. , -3.5) ; /* Horizontale Linie als x-Achse */
umove_gi (hdc , 0. , 0. ) ;
uline_gi (hdc , 10.5 , 0. ) ; /* Vertikale Linie als y-Achse */

umove_gi (hdc , 0. , 4. ) ; /* Startpunkt fuer Funktionsgraph */

for (x = 0.05 ; x <= 10. ; x += 0.05)
{
/*
        -x/5
    Funktion y = 4 e cos 3x :
    uline_gi (hdc , x , 4. * exp (-.2 * x) * cos (3. * x)) ;
}

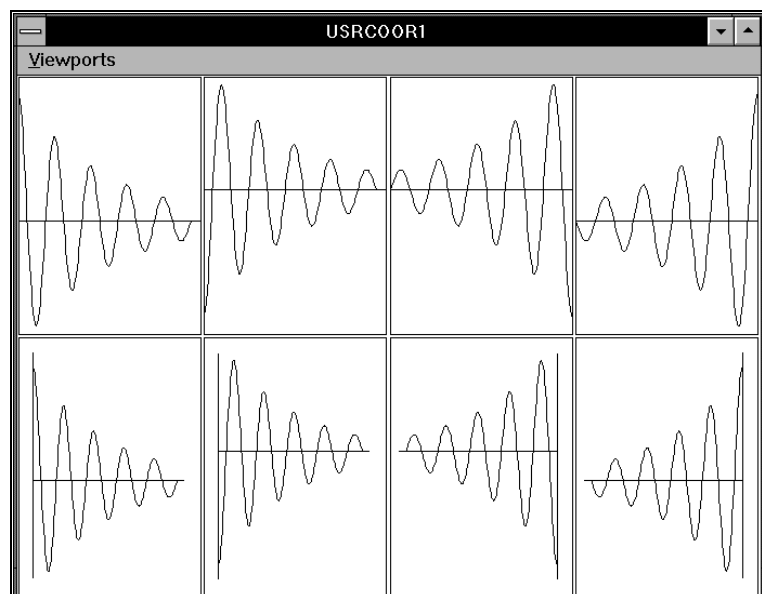
pmarg = 10. ; /* ... untere Reihe mit 10% Rand */
}

gstop_gi (hwnd) ;
return 0 ;

```

Die folgende Abbildung zeigt die Ausgabe des Programms **usrcoor1.c** mit 8 Viewports: In jeder Reihe sind in den einzelnen Viewports die vier möglichen Richtungen der Koordinatenachsen zu sehen. Bei der Definition der "User Coordinates" für die Viewports in der unteren Reihe wurde eine Randeinstellung (10%) vorgegeben, während in der oberen Reihe die beiden Punkte  $P_1$  und  $P_2$ , mit denen die Koordinaten definiert werden, mit den Viewport-Ecken identisch sind.

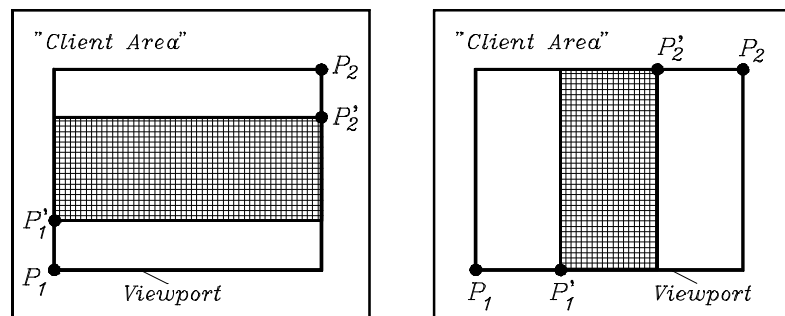
Die Zeichnungen in den Viewports passen sich jeder Änderung der Zeichenfläche in beiden Richtungen an, so daß die Viewportfläche jeweils optimal ausgenutzt wird.



## 2.2.2 "User Coordinates" mit isotroper Skalierung

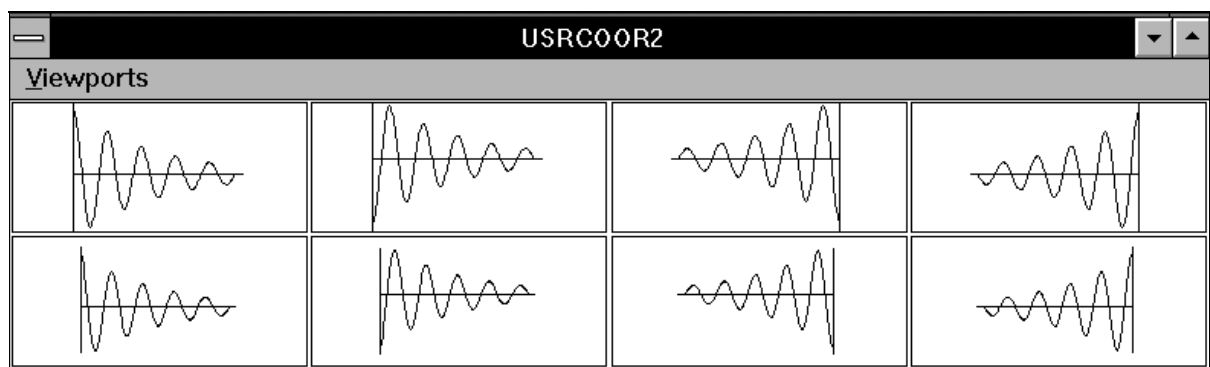
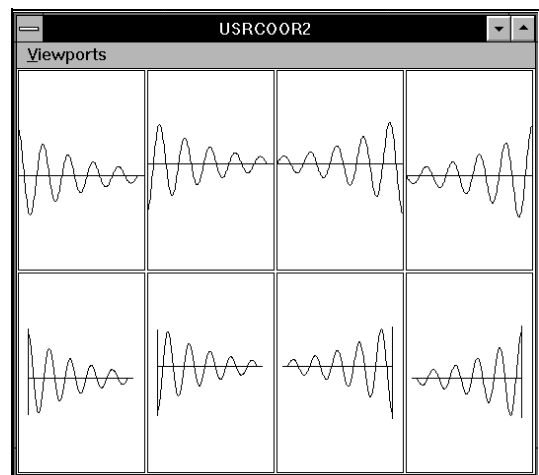
Die GIW-Funktion **stuci\_gi** erwartet wie die im vorigen Abschnitt vorgestellte Funktion **stuca\_gi** 5 **double**-Argumente, die auch die gleiche Bedeutung haben. Die beiden Punkte  $P_1$  und  $P_2$ , die die "User Coordinates" definieren, werden jedoch in jedem Fall so verändert, daß sich in beiden Koordinatenrichtungen gleiche Skalierungen ergeben (isotrope Skalierung). Dabei bleibt in der Regel in einer Richtung ein Teil des Viewport-Bereichs ungenutzt, das Koordinatensystem wird automatisch so gelegt, daß die Distanz der beiden Punkte in dieser Richtung in die Mitte des Viewports fällt.

Die nebenstehende Abbildung zeigt die beiden Möglichkeiten der Platzierung des Zeichenbereichs im Viewport in Abhängigkeit davon, welche Richtung durch den von den "User Coordinates" abzubildenden Bereich besser auszufüllen ist.



Auch für die isotrope Skalierung kann ein Randbereich festgelegt werden. In diesem Fall wird die Distanz der Punkte  $P_1$  und  $P_2$  in der "ungünstigeren Richtung" auf jeder Seite um **pmarg** Prozent vergrößert. In der anderen Richtung liegt der Zeichenbereich auch in diesem Fall in der Viewportmitte.

Das Beispiel-Programm **usrcoor2.c** unterscheidet sich vom Programm **usrcoor1.c**, das im vorigen Abschnitt vorgestellt wurde, nur dadurch, daß die "User Coordinates" isotrop mit **stuci\_gi** definiert werden. Die nebenstehende Abbildung zeigt, wie die Graphiken in "hohe Viewports" eingepaßt werden, in der Abbildung unten ist die Darstellung mit "breiten Viewports" zu sehen. In jedem Fall sind die Einheiten auf der  $x$ -Achse und auf der  $y$ -Achse gleich.



### 2.2.3 "Gefüllte Flächen" mit "User Coordinates"

Neben den im vorigen Abschnitt vorgestellten Funktionen **umove\_gi** und **uline\_gi** gibt es in der GIW-Toolbox noch zahlreiche weitere Funktionen, die mit "User Coordinates" arbeiten. In diesem Abschnitt werden die Funktionen (sämtlich zum Zeichnen "gefüllter Flächen") **ufrec\_gi** (Rechteck mit Seitenlinien, die parallel zu den Viewport-Rändern verlaufen, definiert durch zwei Punkte), **ufell\_gi** (Ellipse, definiert durch zwei Punkte des umschließenden Rechtecks), **ufpol\_gi** (geschlossenes Polygon) und **ufpie\_gi** (elliptischer Sektor) vorgestellt. Ein geschlossenes Polygon wird durch die Anzahl der Punkte und die Koordinaten dieser Punkte definiert.

Ein elliptischer Sektor wird durch zwei Punkte, die das umschließende Rechteck für die komplette Ellipse festlegen würden, und zwei weitere Punkte definiert: Der dritte Punkt definiert gemeinsam mit dem Ellipsen-Mittelpunkt eine Gerade, auf der der Startpunkt des elliptischen Bogens liegt, der Endpunkt wird durch eine entsprechende Gerade mit dem vierten Punkt definiert (gezeichnet wird vom Startpunkt zum Endpunkt entgegen dem Uhrzeigersinn).

Die Ränder der Flächen werden mit der "Farbe des aktuellen Zeichenstiftes" gezeichnet, ausgefüllt werden die Flächen mit der Farbe des aktuellen "Brushs". Die Farben werden im Windows-GDI durch einen **long**-Wert (Typ **COLORREF**), in dem die Information über die RGB-Anteile enthalten ist, definiert (Farbe kann mit dem RGB-Makro aus **windows.h** "gemischt" werden). Für die "acht Grundfarben" (schwarz, blau, grün, cyan, rot, magenta, gelb und weiß) kann dies mit der GIW-Funktion **mkrrgb\_gi** etwas einfacher geschehen.

Von dem Programm **fillcol.c**, das die Verwendung der oben genannten Funktionen zum Zeichnen gefüllter Flächen mit "User Coordinates", das Ausfüllen des Viewport-Hintergrunds mit der mit "Viewport Coordinates" arbeitenden Funktion **vfrec\_gi** und die Verwendung der Funktion **mkrrgb\_gi** demonstriert, wird nachfolgend der Ausschnitt für die Bearbeitung der Botschaft **WM\_PAINT** gelistet:

```
case WM_PAINT :

    hdc      = gstrt_gi (hwnd , cxClient , cyClient) ;
    hBrush1 = CreateSolidBrush (mkrrgb_gi (GI_BLUE)) ;
    hBrush2 = CreateSolidBrush (mkrrgb_gi (GI_CYAN)) ;

    stcvp_gi (hdc , 2 , 2 , cxClient / 2 - 4 , cyClient - 4 , GI_XYBOTTOMLEFT) ;
    /* ... definiert "Current Viewport" in linker Fensterhaelfte */

    SelectObject (hdc , hBrush1) ;
    vfrec_gi (hdc , 0 , 0 , cxClient / 2 - 5 , cyClient - 5) ;
    /* ... fuellt Viewport mit Farbe, zeichnet Rahmen */

    stuca_gi (-1. , -4. , 11. , 5. , 0.) ;
    /* ... definiert "User Coordinates": Punkt (-1;-4) wird auf die
       linke untere Viewport-Ecke gelegt, Punkt (11;5) auf die
       rechte obere Ecke (da die Werte keinen Bezug auf die
       Viewport-Abmessungen nehmen, werden die beiden Achsen
       im Regelfall unterschiedlich skaliert) */

    SelectObject (hdc , hBrush2) ;

    ufrec_gi (hdc , 0. , 0. , 4. , 4.) ;
    /* ... zeichnet ein i. a. "zum Rechteck verzerrtes Quadrat" */
    ufell_gi (hdc , 6. , - 2. , 10. , 2.) ;
    /* ... zeichnet einen i. a. "zur Ellipse verzerrten Kreis" */
```

```

ufell_gi (hdc , 8. , 3. , 14. , 6.) ;
/* ... zeichnet Ellipse, die am Viewport-Rand "geclippt" wird */
ufpol_gi (hdc , 4 , poly_x , poly_y) ;
/* ... zeichnet geschlossenes Polygon mit 4 Punkten */
ufpie_gi (hdc , 4. , 0. , 8. , 4. , 7. , 4. , 4. , 3.) ;
/* ... zeichnet einen i. a. verzerrten Kreissektor */

stcvp_gi (hdc , cxClient / 2 + 2 , 2 ,
          cxClient / 2 - 4 , cyClient - 4 , GI_XYBOTTOMLEFT) ;
/* ... definiert "Current Viewport" in rechter Fensterhaelfte */

/* Man beachte, dass die folgenden Aktionen im rechten Viewport exakt
durch die gleichen Funktionsaufrufe wie fuer den linken Viewport
ausgelöst werden. Lediglich die fuer den Viewport festgelegten
"User Coordinates" fuehren auf ein abweichendes Ergebnis: */

vfrec_gi (hdc , 0 , 0 , cxClient / 2 - 5 , cyClient - 5) ;
/* ... fuellt Viewport mit Farbe, zeichnet Rahmen */
stuci_gi (-1. , -4. , 11. , 5. , 0.) ;
/* ... definiert "User Coordinates": Punkt (-1;-4) wird auf die
linke untere Viewport-Ecke gelegt, Punkt (11;5) auf die
rechte obere Ecke (die beiden Richtungen werden
"isotrop skaliert") */

SelectObject (hdc , hBrush1) ;

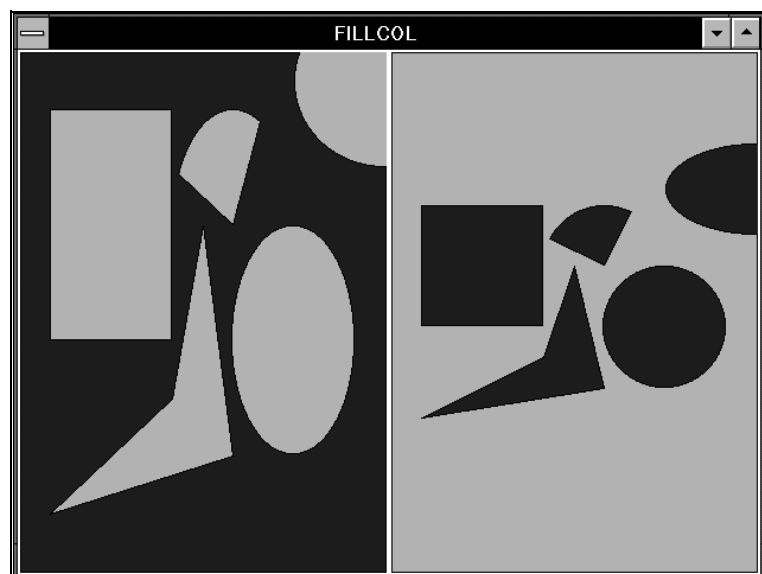
ufrec_gi (hdc , 0. , 0. , 4. , 4.) ;
/* ... zeichnet immer ein Quadrat */
ufell_gi (hdc , 6. , - 2. , 10. , 2.) ;
/* ... zeichnet immer einen Kreis */
ufell_gi (hdc , 8. , 3. , 14. , 6.) ;
/* ... zeichnet Ellipse, die am Viewport-Rand "geclippt" wird */
ufpol_gi (hdc , 4 , poly_x , poly_y) ;
/* ... zeichnet geschlossenes Polygon mit 4 Punkten */
ufpie_gi (hdc , 4. , 0. , 8. , 4. , 7. , 4. , 4. , 3.) ;
/* ... zeichnet einen Kreissektor */

SelectObject (hdc , GetStockObject (WHITE_BRUSH)) ;
DeleteObject (hBrush1) ;
DeleteObject (hBrush2) ;

gstop_gi (hwnd) ;
return 0 ;

```

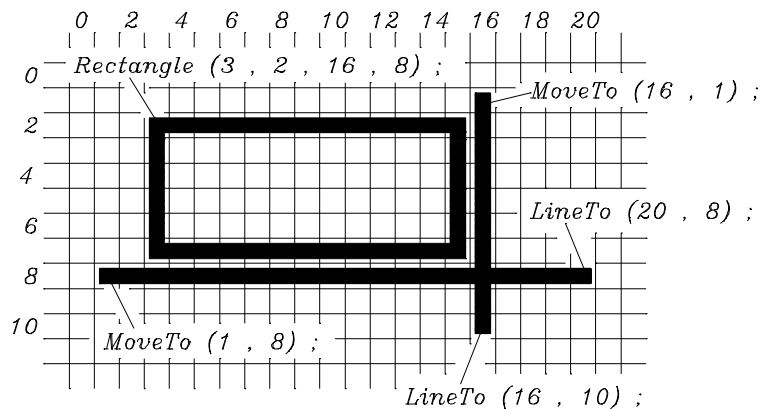
Die nebenstehende Abbildung zeigt die Ausgabe des Programms **fillcol.c**: Im linken Viewport werden die Flächen mit anisotropen "User Coordinates" gezeichnet, ihre Abmessungen passen sich bei Veränderung der Fenstergröße dem Höhen-Breiten-Verhältnis an. Im rechten Viewport wurden isotrope "User Coordinates" eingestellt, "ein Kreis bleibt ein Kreis, ein Quadrat bleibt ein Quadrat".



## 2.2.4 Der Versuch, "pixelgenaue Anschlüsse" zu realisieren

Charles Petzold schrieb bereits in seinem 1992 erschienenen Buch "Programming Windows", daß sich "Murphys Gesetze" um eine Variante erweitern ließen: "Egal, wie man eine Figur zeichnet, um ein Pixel liegt man immer daneben." Und Microsofts Windows-GDI tut einiges, um diese Aussage zu unterstützen.

Die nebenstehende Skizze zeigt die Realisierung eines speziellen Aufrufs der GDI-Funktion **Rectangle** (in den Klammern sind nur die Koordinaten der Eckpunkte angegeben, zum Funktionsaufruf gehört noch ein **HDC-Handle**). Es fällt auf, daß der Punkt links oben exakt die Koordinaten hat, die als Argumente übergeben werden, während die Koordinaten des Punktes rechts unten



Interpretation der Argumente im Windows-GDI

jeweils um 1 kleiner sind als die Argumente. Der hier am Beispiel des Rechtecks gezeigte Effekt gilt für alle GDI-Funktionen, die mit einem "umschließenden Rechteck" arbeiten. Petzold empfiehlt die Modell-Vorstellung, daß mit den Koordinaten nicht die Pixel adressiert werden, sondern die imaginären Gitterlinien zwischen den Pixeln. Dann kann man sich alle mit einem "umschließenden Rechteck" arbeitenden Funktionen als "innerhalb des angegebenen Rechtecks zeichnend" vorstellen.

Dieses Gitter-Koordinaten-Modell versagt natürlich für die GDI-Funktionen **MoveTo** (bzw. **MoveToEx**) und **LineTo**, die genau auf die Pixelpositionen zeichnen (wie natürlich die Funktion **Rectangle** auch, doch für diese ist die oben genannte Modell-Vorstellung möglich). Und lästig wird es eigentlich nur dann, wenn in einer Zeichnung beide Arten von Funktionen aufgerufen werden. Korrigieren kann der Programmierer diesen Schönheitsfehler nur, wenn er ein Koordinatensystem verwendet, das mit Geräte-Koordinaten arbeitet.

Die GIW-Funktionen arbeiten intern nur mit dem Geräte-Koordinatensystem **MM\_TEXT** und "korrigieren" die aus den **double**-Koordinaten berechneten Geräte-Koordinaten um dieses eine Pixel, so daß z. B. die Anschlüsse von **ufrec\_gi**- und **umove\_gi**-Aufrufen "passen". Etwas problematisch kann es bei Kreisen und Kreissektoren werden, die im GDI als Sonderfälle von Ellipse bzw. elliptischem Sektor gezeichnet werden, wobei zwangsläufig der Mittelpunkt nur indirekt (Mittelwerte der Punkte des umschließenden Rechtecks) definiert ist, was bei Linien durch den Mittelpunkt wieder zu sichtbaren Abweichungen führen kann (beim Sonderfall "Kreissektor mit einem begrenzenden Radius, der eigentlich genau vertikal oder horizontal liegen sollte", ist eine Abweichung um ein Pixel besonders deutlich sichtbar).

Im GIW werden deshalb für das Zeichnen von Kreisen und Kreissektoren zusätzliche Funktionen bereitgestellt (z. B. **ufcirc\_gi** für das Zeichnen eines "gefüllten Kreises", **ufsec\_gi** für das Zeichnen eines "gefüllten Kreissektors", ...), die für diese Spezialfälle bevorzugt werden sollten und das beschriebene Problem weitgehend beseitigen.

## 2.2.5 Marker an "User Coordinates"-Positionen

Für viele Probleme, die mit "User Coordinates" bearbeitet werden, ist es sinnvoll, bestimmte Punkte einer Graphik speziell zu markieren (Meßpunkte in Diagrammen, Mittelpunkte, Schwerpunkte, die "Strichelchen" auf Koordinatenachsen, ...) und dafür "Marker" zu verwenden, die bei einer Änderung der Größe des Ausgabebereichs ihre Größe beibehalten. Das Windows-GDI bietet dafür keine speziellen Funktionen an, und für den Programmierer kann es recht unbequem sein, mit den gewählten Koordinaten zu positionieren, um dann mit einem anderen Koordinatensystem, das z. B. wie MM\_LOMETRIC mit festen Abmessungen arbeitet, zu zeichnen.

Die Funktion **umark\_gi** bietet die Möglichkeit, aus einem vorgegebenen Katalog von Markertypen (Kreis, Quadrat, Kreuz, vertikales und horizontales "Strichelchen", gefüllter Kreis und gefülltes Quadrat) zu wählen, mit "User Coordinates" zu positionieren und den Marker mit einer festen voreingestellten Größe zu zeichnen. Die Größe kann über ein Argument beliebig verändert werden, zusätzlich kann die Position des Zeichenstiftes **nach** dem Zeichnen des Markers festgelegt werden, um für eine eventuelle anschließende Beschriftung gleich die aktuelle Textposition zu erzeugen.

Die Standardgröße für die Marker wird mit der Windows-GDI-Dimension "Logical Inch" festgelegt, die für Ausgabe auf Druckern exakt ein Inch ist, für Bildschirmausgabe etwas mehr. Die Anzahl der Geräteeinheiten, die einem "Logical Inch" entspricht, wird durch die in **giw.h** definierte Konstante **MARDIV\_GI** dividiert, und das Ergebnis wird als "Standard-Offset" (Abstand vom Marker-Mittelpunkt bis zu einem Rand des umschließenden Quadrats) verwendet. Bei dem Standardwert **MARDIV\_GI = 20** (dieser kann gegebenenfalls vom Programmierer in **giw.h** geändert werden) beträgt also die Standard-Markergröße (Kantenlänge des umschließenden Quadrats) 1/10 "Logical Inches" (auf dem Bildschirm etwa 3 mm, stimmt natürlich nie genau, weil nur "ganze Pixel" angesprochen werden können).

Das Beispiel-Programm **marker.c** demonstriert die Verwendung der Funktion **umark\_gi**, indem es "Marker in das Fenster regnen läßt": Die Marker werden mit Zufallszahlen im Fenster plaziert, die Zufallszahlen bestimmen auch Markergröße und Markertyp (und die Farben) nach folgendem Prinzip: Am unteren Fensterrand wird Markertyp 1 (Kreis), am oberen Fensterrand Markertyp 7 (gefülltes Quadrat) gezeichnet. Die Markergröße steigt vom linken zum rechten Fensterrand an, am linken Rand gilt **msize = 0**. (es wird die minimale Größe gezeichnet mit einem Abstand von einem Pixel von Markermitte bis zu einem Rand des umschließenden Quadrats), am rechten Rand gilt **msize = 2**., in der Mitte wird die Standard-Markergröße (**msize = 1**.) gezeichnet.

Von dem Programm **marker.c** werden nachfolgend die Ausschnitte für die Bearbeitung der Botschaften **WM\_CREATE**, **WM\_SIZE**, **WM\_PAINT** und **WM\_DESTROY** gelistet:

```
case WM_CREATE :
    for (i = 0 ; i < 8 ; i++)
    {
        hPen    [i] = CreatePen (PS_SOLID , 1 , mkrrgb_gi (i)) ;
        hBrush  [i] = CreateSolidBrush (mkrrgb_gi (i)) ;
    }
    /* ... stellt 8 "Zeichenstifte" und 8 "Brushs" (für die
       "gefüllten Markertypen") bereit, die mit den von
       mkrrgb_gi gelieferten 8 Grundfarben arbeiten */
    return 0 ;
```

```

case WM_SIZE :

    cxClient = LOWORD (lParam) ; /* ... sind die Abmessungen (Pixel) des */
    cyClient = HIWORD (lParam) ; /*   Zeichen-Fensters                */
    return (0) ;

case WM_PAINT :

    hdc = gstrte_gi (hwnd , cxClient , cyClient) ;

    for (i = 1 ; i <= NMARK ; i++)
    {
        rand1 = (double) rand () / RAND_MAX ;
        rand2 = (double) rand () / RAND_MAX ;
        /* ... die erzeugten Zufallszahlen liegen im Bereich 0...1 */

        xs    = (int) (rand1 * cxClient) ;
        ys    = (int) (rand2 * cyClient) ;

        mtype = (int) (rand2 * 7) + 1 ;
        msize = rand1 * 2. ;
        /* ... und Zufallszahl rand1 bestimmt die horizontale
           Koordinate und die Markergroesse, Zufallszahl rand2
           die vertikale Koordinate und den Markertyp */

        rand1 = (double) rand () / RAND_MAX ;
        rand2 = (double) rand () / RAND_MAX ;
        /* ... sind neue Zufallszahlen fuer die Farbauswahl, damit
           es "schoen bunt" wird */

        SelectObject (hdc , hPen  [(int) (rand1 * 8.)]) ;
        SelectObject (hdc , hBrush [(int) (rand2 * 8.)]) ;

        umark_gi (hdc , mtype , msize , xs , ys , 0) ;
    }

    gstop_gi (hwnd) ;
    return 0 ;

case WM_DESTROY :

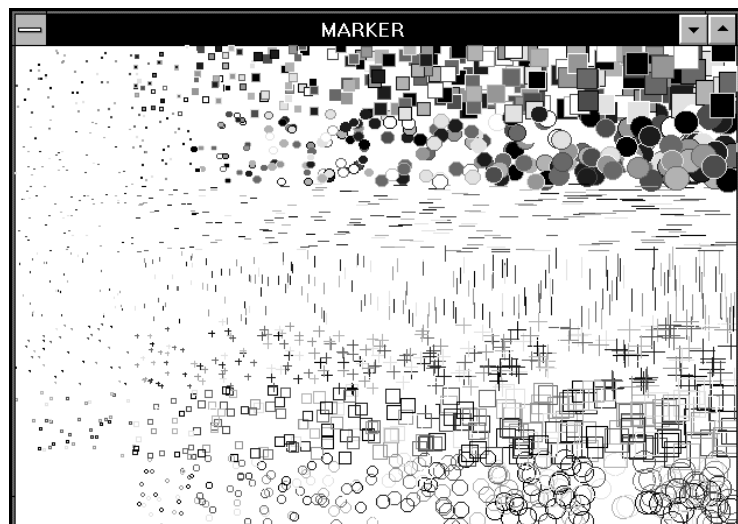
    for (i = 0 ; i < 8 ; i++)
    {
        DeleteObject (hPen  [i]) ;
        DeleteObject (hBrush [i]) ;
    }

    PostQuitMessage (0) ;
    return 0 ;

```

Die nebenstehende Abbildung zeigt die Ausgabe des Programms **marker.c**. Am linken Rand des Fensters sieht man die Marker in minimaler Größe, am rechten Rand gilt der Größen-Parameter `msize = 2`. In der Fenstermitte sind die Marker in der vorgegebenen Standardgröße (`msize = 1`) dargestellt.

Es hat "Marker geregnet"



## 2.3 "User Coordinates"-Punkte picken, Zoom

Für die Eingabe von Punkten durch Mauspick und das Aufziehen eines rechteckigen Bereichs, der z. B. für ein anschließendes "Zoomen" der Graphik benutzt werden kann, stehen folgende GIW-Funktionen zur Verfügung:

- ♦ **umpos\_gi** liefert die Mausposition in "User Coordinates".
- ♦ **upick\_gi** liefert die Mausposition in "User Coordinates" und schaltet einen eventuell vorher eingeschalteten "speziellen GIW-Cursor" ab.
- ♦ **scapp\_gi** läßt einen "speziellen GIW-Cursor" erscheinen (z. B. ein über den gesamten Viewport ausgedehntes Kreuz), dieser wird im GIW gezeichnet, ist also kein Cursor im Sinne der "Windows-Cursor".
- ♦ **scupd\_gi** aktualisiert die Position eines mit **scapp\_gi** erzeugten "speziellen GIW-Cursors" (wird sinnvollerweise als Reaktion auf die Botschaft WM\_MOUSEMOVE aufgerufen).
- ♦ **scdap\_gi** läßt einen mit **scapp\_gi** erzeugten "speziellen GIW-Cursor" wieder verschwinden.
- ♦ **rpick\_gi** ist speziell für das Definieren eines Rechteckbereichs zuständig, setzt den vorherigen Aufruf von **scapp\_gi** voraus. Die Funktion **rpick\_gi** "merkt sich" beim ersten Aufruf (z. B. als Reaktion auf WM\_LBUTTONDOWN) die Koordinaten des aktuellen Punktes und ändert den "speziellen GIW-Cursor" auf die Form "Rechteck". Beim nächsten Aufruf von **rpick\_gi** (z. B. wieder als Reaktion auf WM\_LBUTTONDOWN oder auch als Reaktion auf WM\_LBUTTONUP) werden die beiden Punkte in "User Coordinates" abgeliefert.

Die Verwendung dieser Funktionen wird im Beispiel-Programm **zoom.c** demonstriert. Dieses Programm stellt die in Parameter-Darstellung gegebene mathematische Funktion

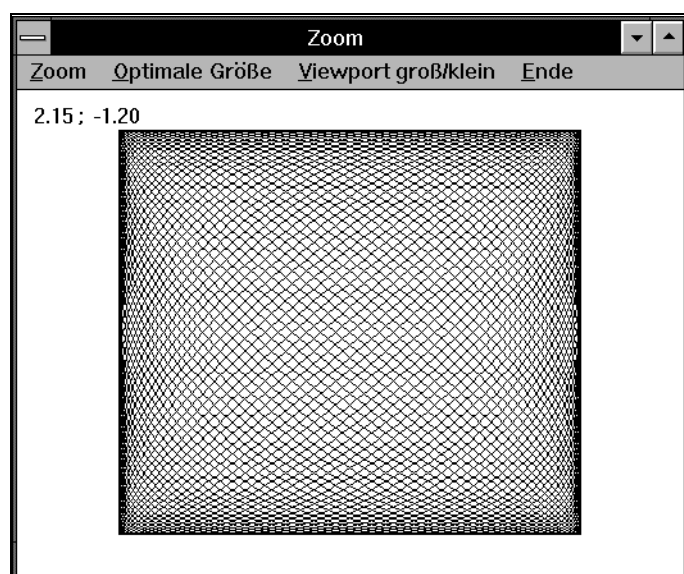
$$x = 4 \cos 51 t \quad ; \quad y = 4 \sin 50 t$$

(Überlagerung einer horizontalen und einer vertikalen Schwingung, deren Frequenzen sich nur geringfügig unterscheiden) im Bereich

$$0 \leq t \leq 2\pi$$

dar.

Die nebenstehende Abbildung zeigt das Hauptfenster nach dem Start des Programms ("Current Viewport" ist mit der "Client Area" identisch). Die Funktion wird unter Verwendung von "User Coordinates" (mit isotroper Skalierung) gezeichnet, links oben sieht man die "User Coordinates" der aktuellen Cursor-Position.







- ♦ Der Funktion **umpos\_gi** wird als erstes Argument der **long**-Wert übergeben, mit dem in Windows die beiden Cursor-Koordinaten beschrieben werden. Auf den Positionen 2 und 3 liefert **umpos\_gi** die entsprechenden "User Coordinates" ab (bezogen auf die letzte im GIW gespeicherte Viewport- und "User Coordinates"-Definition), allerdings nur dann, wenn mit dem Return-Wert **1** signalisiert wird, daß sich die Position im "Current Viewport" befindet.
- ♦ Der Aufruf von **scupd\_gi** wird nur ausgeführt, wenn "gezoomt" werden soll. Dies wird weiter unten beschrieben.

Mit den 4 Menü-Angeboten können ein "Zoom" eingeleitet, auf die Standardgröße zurückgegangen, die Viewport-Größe verändert und das Programm beendet werden, gesteuert über die Auswertung von WM\_COMMAND:

case WM\_COMMAND:

```
switch (wParam)
{
    case 10:                /* Zoom */

        scapp_gi (hwnd , 0 , 0 , GI_CUBIGCROSS) ;
        zoom = 1 ;         /* ... das grosse Kreuz erscheint als Cursor */

        return 0 ;

    case 20:                /* Optimale Groesse */

        scdap_gi (hwnd) ;  /* ... es koennte ein "Spezial-Cursor"      */
        plx = - 4. ;       /* (z. B. das grosse Kreuz) noch aktiv      */
        ply = - 4. ;       /* sein, scdap_gi schaltet diesen ab      */
        p2x = 4. ;
        p2y = 4. ;
        InvalidateRect (hwnd , NULL , TRUE) ;

        return 0 ;

    case 30:                /* Viewport gross/klein */

        vpbig = (vpbig == 1) ? 0 : 1 ;
        SendMessage (hwnd , WM_COMMAND , 20 , 0) ;

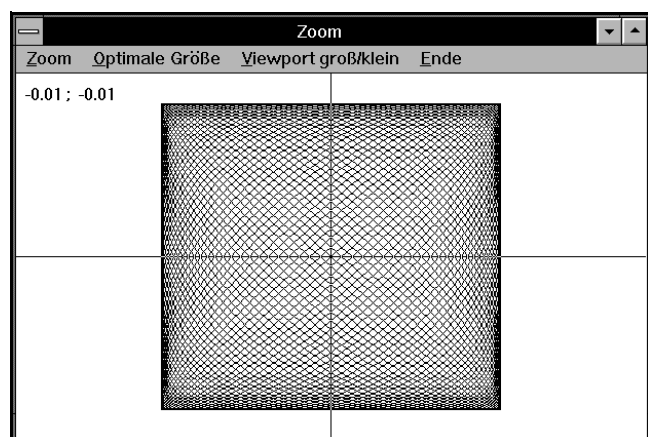
        return 0 ;

    case 40:                /* Ende */

        SendMessage (hwnd , WM_CLOSE , 0 , 0) ;

        return 0 ;
}
```

- ♦ Nach Auswahl des Menü-Angebots **Zoom** wird **scapp\_gi** aufgerufen, und ein "spezieller GIW-Cursor" erscheint (hier das "große Kreuz", das von Viewport-Grenze bis Viewport-Grenze reicht, nebenstehende Abbildung). Der Cursor erscheint hier an der als Argumente 2 und

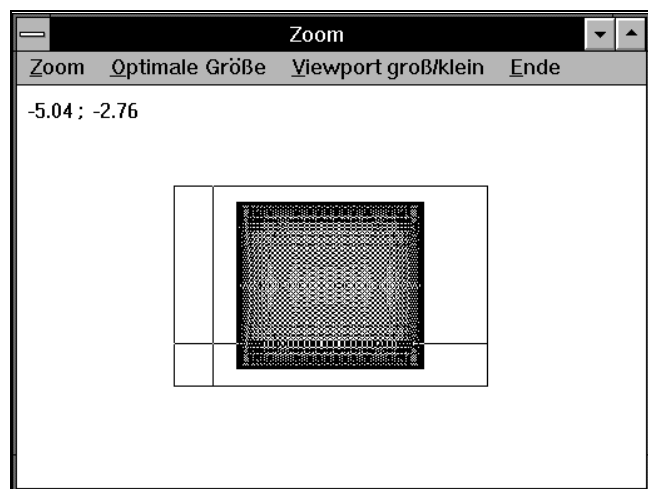


3 übergebenen Position **(0;0)**. Weil das Viewport-Koordinatensystem in die Viewport-mitte gelegt wurde, erscheint der Cursor an dieser Stelle.

Da der Parameter **zoom** den Wert **1** erhält, wird nun die bei Auswertung der Botschaft WM\_MOUSEMOVE angesiedelte Funktion **scupd\_gi** auch aufgerufen. Diese überzeichnet jeweils den "alten" Cursor und zeichnet ihn an der aktuellen Position neu.

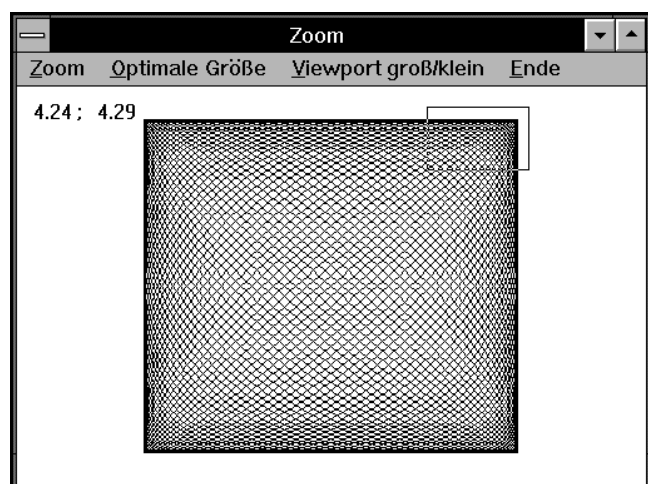
- ♦ Die Menü-Auswahl **Optimale Größe** setzt (nach Zoom-Operationen) die Werte, mit denen die "User Coordinates" der Viewport-Ecken definiert werden, auf die Anfangswerte zurück und erzwingt das Neuzeichnen (**InvalidateRect** löst WM\_PAINT aus). Vorher wird mit **scdap\_gi** ein eventuell aktiver "spezieller GIW-Cursor" abgeschaltet.
- ♦ Das Menü-Angebot **Viewport groß/klein** wurde eingefügt, um zu demonstrieren, daß die hier vorgestellten GIW-Funktionen auch dann funktionieren, wenn der "Current Viewport" nicht mit der "Client Area" identisch ist.

Die nebenstehende Abbildung zeigt den kleineren Viewport, der nach der Wahl dieses Menü-Angebots erscheint, nachdem danach noch das Angebot **Zoom** gewählt wurde. Man sieht zusätzlich zur Graphik den speziellen GIW-Cursor ("großes Kreuz"), dessen Ausdehnung sich auf den "Current Viewport" beschränkt. Die links oben angegebenen Koordinaten entsprechen der aktuellen Cursor-Position, bezogen auf die "User Coordinates", die für den Viewport eingestellt wurden.



Nach der Wahl von **Zoom** kann das "große Kreuz" verschoben werden, nach Drücken der linken Maustaste wird nun die Funktion **rpick\_gi** aufgerufen. Diese verändert bei ihrem ersten Aufruf (nach Aufruf von **scapp\_gi**) den Cursor auf die Form "Rechteck" (nebenstehende Abbildung), ein Eckpunkt ist der gerade "gepickte" Punkt, der andere verändert sich wieder bei der Mausbewegung (realisiert durch den Aufruf von **scupd\_gi** bei der Auswertung der Botschaft WM\_MOUSEMOVE).

Das "Picken des Rechteckbereichs" wird im Programm **zoom.c** mit der Auswertung der Botschaft WM\_LBUTTONDOWN erledigt, die in **zoom.c** folgendermaßen codiert ist:

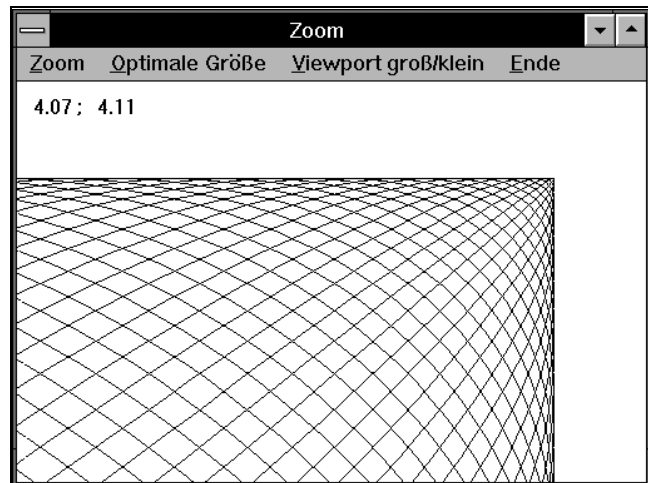


```
case WM_LBUTTONDOWN:
```

```
    if (zoom)
    {
        if (rpick_gi (hwnd , lParam , &plx , &ply , &p2x , &p2y))
            InvalidateRect (hwnd , NULL , TRUE) ; /* ... wird erst beim
                                                    zweiten Aufruf von rpick_gi ausgeführt */
    }
    else
    {
        upick_gi (hwnd , lParam , &xu , &yu) ;
        sprintf (str , "x = %g, y = %g" , xu , yu) ;
        MessageBox (hwnd , str , "Gepickter Punkt" , MB_OK) ;
    }
}
```

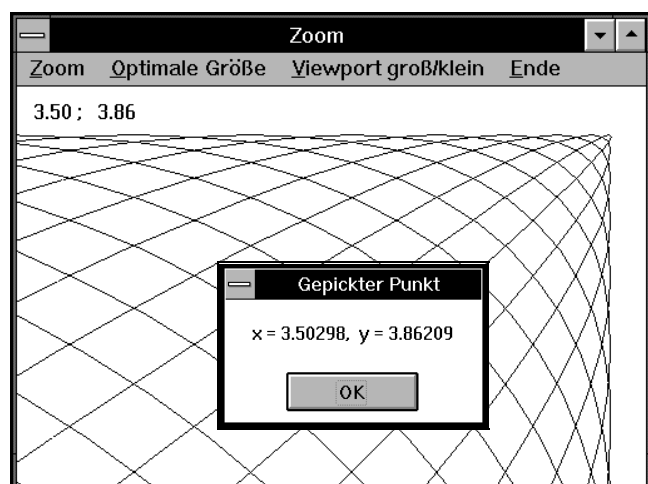
- ♦ Beim ersten Aufruf liefert **rpick\_gi** den Return-Wert **0**, so daß außer den Aktivitäten innerhalb dieser Funktion (Umstellen des Cursors) keine weiteren Aktionen ausgelöst werden. Erst beim zweiten Aufruf liefert die Funktion den Return-Wert **1**, und mit **InvalidateRect** wird das Neuzeichnen veranlaßt.

Die von **rpick\_gi** abgelieferten "User Coordinates" des gewählten Rechtecks werden bei der Bearbeitung der Botschaft WM\_PAINT für die Definition der Eckpunkte mittels **stuci\_gi** verwendet, so daß ein Zoom realisiert wird. Die nebenstehende Abbildung zeigt das Ergebnis "nach dem zweiten Mausepick".



Natürlich hätte man bei der Zoom-Aktion den Windows-Cursor (hier der Pfeil) für das Fenster auch abschalten können. Weil er aber eigentlich nicht stört, kann er durchaus auch sichtbar bleiben.

- ♦ Wenn die Botschaft WM\_LBUTTONDOWN ohne vorherige Auswahl des Menü-Angebots **Zoom** ausgelöst wird, ist der oben dargestellte "else-Zweig" für die Abarbeitung zuständig: Die Funktion **upick\_gi** liefert die "User Coordinates" des "gepickten Punktes", die in diesem Fall in einer Message-Box angezeigt werden (nebenstehende Abbildung).



### 3 Transformationen und Projektionen

Häufig ist es sinnvoll, die Koordinaten der Punkte, die die zu zeichnenden Elemente definieren, vor dem Zeichnen einer Transformation (Verschiebung, Drehung, Spiegelung, Skalierung) zu unterwerfen. So kann man die Darstellung eines Objekts in einer speziellen Lage des Koordinatensystems programmieren und die Koordinaten vor der tatsächlichen Zeichenaktion transformieren. Besonders erleichtert wird so auch die mehrfache Darstellung des gleichen Objekts in unterschiedlichen Lagen.

Die Koordinaten von dreidimensionalen Objekten müssen in jedem Fall vor der graphischen Darstellung auf eine zweidimensionale Ebene projiziert werden.

Die Lösung beider Probleme (Transformation bzw. Projektion) wird von speziellen GIW-Funktionen unterstützt, deren Anwendung in diesem Kapitel beschrieben wird. Vorangestellt werden jeweils die theoretischen Grundlagen, die als Basis für die GIW-Funktionen dienen.

#### 3.1 Homogene Koordinaten

Speziell für die Probleme der projektiven Geometrie wird mit erheblichem Vorteil die Darstellung eines Punktes im Raum durch vier Angaben (an Stelle der drei kartesischen Koordinaten) beschrieben. Diese sogenannten **homogenen Koordinaten** stehen mit den kartesischen Koordinaten in einem einfachen Zusammenhang:

$$\text{Homogene Koordinaten} \rightarrow \begin{bmatrix} x \\ y \\ z \\ \lambda \end{bmatrix} \Rightarrow \begin{bmatrix} x/\lambda \\ y/\lambda \\ z/\lambda \end{bmatrix} \leftarrow \text{Kartesische Koordinaten}$$

Dabei kann  $\lambda$  einen beliebigen Wert ungleich Null haben, für den Spezialfall  $\lambda = 1$  sind die ersten drei Komponenten in der Darstellung eines Punktes mit homogenen Koordinaten mit den kartesischen Koordinaten identisch.

Der Vorteil, der sich aus der Verwendung homogener Koordinaten ergibt, wird sich in der Möglichkeit der einheitlichen Darstellung unterschiedlicher Transformationen zeigen, wodurch sich nacheinander auszuführende Transformationen mathematisch sehr übersichtlich verknüpfen lassen. Bei der Projektion von Raumpunkten in eine Darstellungsebene ergeben sich die Abbildungen automatisch in (ebenen) homogenen Koordinaten.

Folgende Vorstellung kann mit der sicher etwas ungewöhnlichen Beschreibung eines Punktes durch vier Angaben verknüpft werden: Der darzustellende Punkt definiert gemeinsam mit dem Nullpunkt des Koordinatensystems eine Gerade. Wenn  $\lambda$  die Werte von  $-\infty$  bis  $+\infty$  durchläuft, dann werden alle Punkte auf dieser Geraden beschrieben (und im Vorgriff auf das Thema "Projektion": Wenn diese Gerade die "Blickrichtung" definiert, werden alle Punkte der Geraden auf den gleichen Punkt der "Projektionsebene" abgebildet). Aber selbst für die ebenen Probleme allein ergeben sich Vorteile durch die Verwendung homogener Koordinaten.

### 3.2 Ebene Transformationen

Zwei verschiedene Transformationen werden betrachtet:

- ♦ Wenn sich ein Objekt bezüglich eines **festen Koordinatensystems** bewegt, dann ändern sich die Koordinaten seiner Punkte nach den Regeln der **geometrischen Transformation**.
- ♦ Wenn das Koordinatensystem selbst bewegt wird, dann ändern sich die Koordinaten der Punkte eines sich nicht mitbewegenden Objekts nach den Regeln der **Koordinatentransformation**.

Es genügt, das "Schicksal" eines Punktes bei einer Transformation zu untersuchen. In allen Fällen wird im folgenden die "alte" Lage des Punktes durch die Koordinaten  $x$  und  $y$  beschrieben, die Lage nach der Transformation durch die Koordinaten  $x'$  und  $y'$ .

Es werden folgende vier sehr einfache Transformationen beschrieben (Translation, Rotation um den Nullpunkt, Skalierung bezüglich des Nullpunktes, Spiegelung an den Koordinatenachsen), aus denen sich durch geeignete Verknüpfungen beliebige andere Transformationen zusammensetzen lassen:

- ♦ Bei einer **Translation** bewegen sich alle Punkte des Objekts (geometrische Transformation) bzw. des Koordinatensystems (Koordinatentransformation) auf kongruenten Bahnen. Bei der **geometrischen Translation** mögen die Koordinaten aller Punkte des Objekts die Veränderungen  $t_x$  und  $t_y$  erfahren, so daß die neue Lage eines Punktes durch

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

beschrieben wird.

- ♦ Bei einer **Rotation** drehen sich das Objekt (geometrische Transformation) bzw. das Koordinatensystem (Koordinatentransformation) um einen bestimmten Winkel **um eine vorzugebende Achse**, beim ebenen Problem also um einen vorzugebenden Punkt. Mit einer kleinen Skizze macht man sich leicht klar, daß bei einer **Rotation eines Punktes um den Nullpunkt (geometrische Rotation)** um den (entgegen dem Uhrzeigersinn positiv gezählten) Winkel  $\varphi$  folgende Transformation gilt:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \cos \varphi - y \sin \varphi \\ x \sin \varphi + y \cos \varphi \end{bmatrix} = \begin{bmatrix} \cos \varphi & -\sin \varphi \\ \sin \varphi & \cos \varphi \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}.$$

- ♦ Bei der **Skalierung** werden die Abmessungen des Objekts (geometrische Transformation) bzw. die Skaleneinteilung der Koordinatenachsen (Koordinatentransformation) vergrößert (Skalierungsfaktoren größer als 1) bzw. verkleinert. Die Skalierung bezieht sich immer auf einen zu definierenden Punkt, der dann selbst seine Lage nicht

verändert, während alle anderen Punkte ihren Abstand vom Bezugspunkt vergrößern oder verkleinern. Bei der **geometrischen Skalierung bezüglich des Nullpunktes** mit den Skalierungsfaktoren  $s_x$  und  $s_y$  (jeweils in Richtung der Koordinatenachsen) berechnet sich die Lage des neuen Punktes nach

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x x \\ s_y y \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} .$$

- ♦ Bei einer **Spiegelung eines Objektes an der y-Achse (geometrische Transformation)** ändern alle  $x$ -Koordinaten der Punkte ihr Vorzeichen:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} -x \\ y \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} .$$

Dementsprechend gilt für die **Spiegelung eines Objektes an der x-Achse (geometrische Transformation)**:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ -y \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} .$$

Da sich die Translation als einzige Transformation in kartesischen Koordinaten nicht durch eine Transformationsmatrix beschreiben läßt, kann hier erstmals mit Vorteil von den homogenen Koordinaten Gebrauch gemacht werden. Die kartesischen Koordinaten werden um  $\lambda = 1$  ergänzt, und alle Transformationen lassen sich einheitlich durch Transformationsmatrizen beschreiben. Während für die Translation das Aufschreiben der Beziehung als Multiplikation "Matrix \* Vektor" überhaupt erst möglich wird, werden die Transformationsmatrizen für die Rotation, die Skalierung und die Spiegelung um eine einfache Zeile bzw. Spalte ergänzt ("erändert"). Diese Formeln werden nachfolgend zusammengestellt.

### 3.2.1 Ebene geometrische Transformation mit homogenen Koordinaten

Die Formeln beschreiben die **Bewegung eines Punktes im festen Koordinatensystem!**

**Translation um  $t_x$  und  $t_y$ :**

$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \bar{T}_G \bar{x}$$

**Rotation um den Nullpunkt mit dem Winkel  $\varphi$  entgegen dem Uhrzeigersinn:**

$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \varphi & -\sin \varphi & 0 \\ \sin \varphi & \cos \varphi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \bar{R}_G \bar{x}$$

**Skalierung bezüglich des Nullpunktes um  $s_x$  und  $s_y$ :**

$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \bar{S}_G \bar{x}$$

**Spiegelung an der y-Achse:**

$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \bar{M}_{y,G} \bar{x}$$

**Spiegelung an der x-Achse:**

$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \bar{M}_{x,G} \bar{x}$$

### 3.2.2 Ebene Koordinatentransformation mit homogenen Koordinaten

Die Formeln gelten bei **Bewegung des Koordinatensystems (Objekte bleiben in Ruhe)!**

**Translation um  $t_x$  und  $t_y$ :**

$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & -t_x \\ 0 & 1 & -t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \bar{T}_K \bar{x} = \bar{T}_G^{-1} \bar{x}$$

**Rotation um den Nullpunkt mit dem Winkel  $\varphi$  entgegen dem Uhrzeigersinn:**

$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \varphi & \sin \varphi & 0 \\ -\sin \varphi & \cos \varphi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \bar{R}_K \bar{x} = \bar{R}_G^{-1} \bar{x}$$

**Skalierung der Einheiten bezüglich des Nullpunktes um  $s_x$  und  $s_y$ :**

$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1/s_x & 0 & 0 \\ 0 & 1/s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \bar{S}_K \bar{x} = \bar{S}_G^{-1} \bar{x}$$

**Spiegelung an der y-Achse:**

$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \bar{M}_{y,K} \bar{x} = \bar{M}_{y,G}^{-1} \bar{x}$$

**Spiegelung an der x-Achse:**

$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \bar{M}_{x,K} \bar{x} = \bar{M}_{x,G}^{-1} \bar{x}$$



Die angegebenen Formeln zur Koordinatentransformation leiten sich aus ähnlich einfachen Überlegungen her, wie sie für die geometrischen Transformationen diskutiert wurden. Bemerkenswert ist, daß alle Transformationsmatrizen der Koordinatentransformationen die Inversen zu den entsprechenden Transformationsmatrizen der geometrischen Transformationen sind (und natürlich umgekehrt). Dies ist allerdings leicht einzusehen, weil eine Koordinatentransformation mit den gleichen Parametern wie eine entsprechende geometrische Transformation ein "Nachführen des Koordinatensystems" bedeutet, so daß der ursprüngliche Zustand wieder hergestellt wird.

In der GIW-Library sind nur Funktionen verfügbar, die die geometrischen Transformationen realisieren, nach den oben angegebenen Formeln sind sie allerdings leicht auch für eventuell auszuführende Koordinatentransformationen nutzbar.

### 3.2.3 Verknüpfung von Transformationen

Aus den Elementartransformationen, für die in den Abschnitten 3.2.1 und 3.2.2 die Formeln angegeben wurden, können beliebige Transformationen durch das Ausführen von mehreren Transformationen nacheinander realisiert werden. Dabei ist zu beachten, daß eine **Transformation immer durch Linksmultiplikation mit der entsprechenden Transformationsmatrix realisiert wird**, so daß schließlich ein Produkt mehrerer Transformationsmatrizen die Gesamt-Transformation beschreibt, bei der die Matrix der letzten Transformation links steht.

Dies soll an einem einfachen Beispiel gezeigt werden: Es ist die geometrische Transformationsmatrix zu bestimmen, mit der die Rotation eines Punktes  $(x, y)$  um einen Winkel  $\varphi$  (im Uhrzeigersinn) um den **beliebigen Drehpunkt**  $(x_M, y_M)$  erzeugt wird. Da in den Elementartransformationen nur der Fall "Rotation um den Nullpunkt" vorgesehen ist, wird folgendermaßen vorgegangen:

Zunächst wird der Ursprung des Koordinatensystems in den Punkt  $(x_M, y_M)$  verschoben (**Koordinatentranslation 1** mit  $t_x = x_M$  und  $t_y = y_M$ ), danach kann die **geometrische Rotation** um den Nullpunkt ausgeführt werden, schließlich muß die Verschiebung des Koordinatensystems rückgängig gemacht werden (entweder **Koordinatentranslation 2** mit  $t_x = -x_M$  und  $t_y = -y_M$  oder **geometrische Translation** mit  $t_x = x_M$  und  $t_y = y_M$ ). Folgende Matrizen sind also zu multiplizieren (man beachte, daß in der Matrix für die Koordinatentranslation die Elemente  $-t_x$  und  $-t_y$  stehen):

$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \bar{T}_{K,2} \bar{R}_G \bar{T}_{K,1} \bar{x} = \bar{T}_{Gesamt} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \bar{T}_{Gesamt} \bar{x}$$

mit

$$\bar{T}_{Gesamt} = \begin{bmatrix} 1 & 0 & x_M \\ 0 & 1 & y_M \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos \varphi & -\sin \varphi & 0 \\ \sin \varphi & \cos \varphi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_M \\ 0 & 1 & -y_M \\ 0 & 0 & 1 \end{bmatrix} .$$

Man beachte, daß eine Vertauschung der Reihenfolge zweier Transformationen in der Regel auf eine ganz andere Gesamt-Transformation führt. In der mathematischen Beschreibung verknüpfter Transformationen mit homogenen Koordinaten wird dies durch die Matrizen-Produkte der Transformationsmatrizen deutlich: Im Gegensatz zum Produkt skalarer Größen ist das Matrix-Produkt nicht kommutativ.

### 3.3 Die "t...-Funktionen" des GIW

Die sogenannten "**t...-Funktionen**" (alle zugehörigen Funktionsnamen beginnen mit **t**) des GIW lassen sich in zwei Gruppen einteilen:

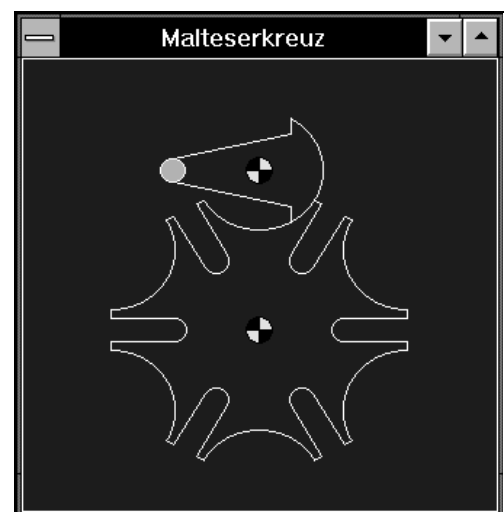
- ♦ Die "**vorbereitenden t...-Funktionen**" definieren bzw. ändern die gültige Ebene Transformationsmatrix.
- ♦ Die "**zeichnenden t...-Funktionen**" arbeiten wie die im Abschnitt 2.2 beschriebenen "u...-Funktionen" mit "User Coordinates", beziehen sich auf das gültige (mit **stuca\_gi** bzw. **stuci\_gi** eingestellte) Koordinatensystem, wenden aber vor der eigentlichen Zeichenaktion die gültige Ebene Transformationsmatrix auf die übergebenen Koordinaten an.

Nicht zu allen "u...-Funktionen" existieren auch die entsprechenden "zeichnenden t...-Funktionen", konsequent kann das Prinzip der Vorab-Transformation ohnehin nur für die Argumente realisiert werden, die Punkt-Koordinaten beschreiben ("Move to", "Line to", "Polygon"). Die "t...-Funktionen", mit denen Kreise bzw. Kreisbögen mit Mittelpunkt und Radius beschrieben werden, wenden auf den Radius z. B. nur die eingestellte Skalierung an.

Nachfolgend werden das Prinzip des Arbeitens mit den "t...-Funktionen" und die Verwendung der wichtigsten Funktionen am Beispiel des Programms **malteser.c** (Zeichnen eines symbolischen Malteserkreuz-Getriebes) dargestellt, das schrittweise entwickelt wird.

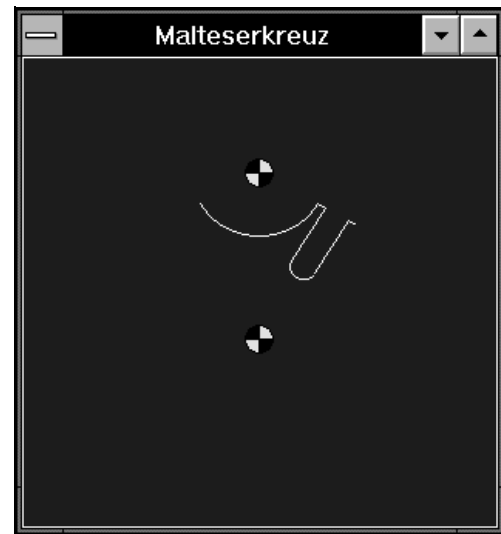
Die nebenstehende Skizze zeigt das Schema des Getriebes (dieses Bild wird schließlich vom Programm **malteser.c** erzeugt), das nach folgendem Prinzip arbeitet. Die "Kurbel" (im Bild oben) dreht sich normalerweise mit konstanter Winkelgeschwindigkeit. Der Stift am Ende der Kurbel greift bei jeder Umdrehung in einen Schlitz des "Malteserkreuzes" ein und dreht dieses weiter. Wenn der Stift den Schlitz wieder verläßt, kommt das Kreuz zum Stillstand, bis der Stift bei der folgenden Kurbelumdrehung in den nächsten Schlitz eingreift.

Das Malteserkreuz der dargestellten Getriebe-Variante besteht aus 6 jeweils um 60° versetzten "Schwalbenschwänzen", die jeweils durch einen Schlitz mit halbkreisförmigem Grund getrennt sind. Es bietet



Schema eines Malteserkreuz-Getriebes

sich also an, nur ein Sechstel des Bildes zu programmieren und mit geeigneten Rotationstransformationen wiederholt zu erzeugen. Die nebenstehende Abbildung zeigt das zu programmierende Sechstel (und die beiden Drehzapfen), dieses Bild wird von der Version **maltesr0.c** erzeugt. Schon für das Erzeugen dieses Teilbildes werden mit Vorteil Rotationstransformationen eingesetzt:

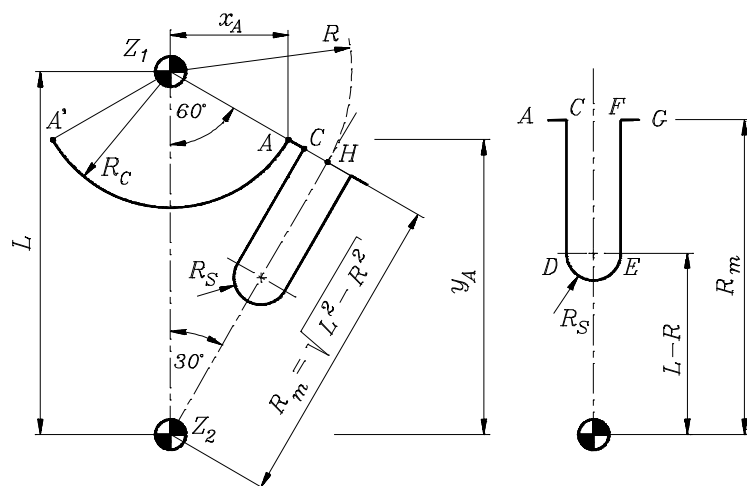


Ein Sechstel des Malteserkreuzes

- ♦ Vor der ersten Verwendung einer "zeichnenden t...-Funktion" muß die ebene Transformationsmatrix initialisiert werden. Dies geschieht hier mit **tinit\_gi**, damit wird eine Einheitsmatrix vorgegeben (keine Transformation), so daß die "zeichnenden t...-Funktionen" sich zunächst wie die entsprechenden "u...-Funktionen" verhalten.
- ♦ Die Funktion **trota\_gi** realisiert die (im Abschnitt 3.2.3 als Beispiel demonstrierte) "Rotation um einen vorzugebenden Punkt". Sie **arbeitet inkrementell**, fügt also die Rotation (durch entsprechende Matrix-Multiplikation) einer bereits eingestellten Transformation hinzu.

Um den nachfolgenden Ausschnitt aus dem Programmcode von **maltesr0.c** besser zu verstehen, zeigt die nebenstehende Skizze die typischen Abmessungen, die die Geometrie des Malteserkreuzes bestimmen:

Die Mittelpunkte der beiden Drehzapfen  $Z_1$  und  $Z_2$  bilden mit dem Punkt  $H$  ein rechtwinkliges Dreieck. Die eingezeichneten Winkel und die Abmessungen  $L$ ,  $R$ ,  $R_C$  und  $R_S$  definieren die aus Kreisbögen und Geraden aufgebaute Kontur.



Die Geometrie des Malteserkreuzes, der Schlitz wird in der rechts dargestellten vertikalen Lage programmiert

Die Abmessungen werden als (globale) Konstanten definiert oder erhalten ihre Werte beim Bearbeiten der Botschaft WM\_CREATE. Dort werden auch sämtliche "Pens" und "Brushes" erzeugt, die irgendwann benötigt werden, diese werden beim Bearbeiten von WM\_DESTROY wieder gelöscht. Auf diese Einzelheiten wird hier nicht eingegangen.

Die Zeichenaktion wird bei der Bearbeitung der Botschaft WM\_PAINT ausgeführt. Zwischen **gstrt\_gi** und **gstop\_gi** wird mit **vfrec\_gi** ein blauer Hintergrund erzeugt, mit **stuci\_gi** werden **isotrope** "User Coordinates" passend zu den maximalen Abmessungen bei Einhaltung eines

kleinen Randes eingestellt (der Koordinatenursprung liegt in der Mitte des unteren Drehzapfens). Vor dem Aufruf von **DrawCross** (Zeichnen des Malteserkreuzes) wird mit **tinit\_gi** die ebene Transformationsmatrix initialisiert (Einheitsmatrix = keine Transformation):

```
case WM_PAINT :

    hdc = gstrt_gi (hwnd , cxClient , cyClient) ;

    SelectObject (hdc , hPenWhite) ;
    SelectObject (hdc , hBrushBlue) ;

    vfrec_gi (hdc , 0 , 0 , cxClient - 1 , cyClient - 1) ;
    stuci_gi (- L , - L , L , L + R + Rs , 5.) ;

    tinit_gi () ;                                /* Transformation initialisieren */
    DrawCross (hdc) ;                            /* Malteserkreuz zeichnen */

    DrawPivot (hdc , 0.) ;
    DrawPivot (hdc , L) ;

    gstop_gi (hwnd) ;

    return 0 ;
```

Mit dem Aufruf von **DrawPivot** wird jeweils ein Drehzapfen gezeichnet. Dafür werden ausschließlich "u...-Funktionen" verwendet, die die eingestellte Transformation nicht berücksichtigen. Die eigentlich interessante Zeichenaktion wird durch folgende Sequenz von "t...-Funktionen" in **DrawCross** realisiert:

```
/* Zeichnen eines Sechstels des Malteserkreuzes: */
tdarc_gi (hdc , 0. , L , Rc , - xA , yA , xA , yA) ;
tmove_gi (hdc , xA , yA) ;
trota_gi (0. , 0. , - Pi / 6.) ; /* ... um 30° drehen */
tline_gi (hdc , - Rs , Rm) ;
tline_gi (hdc , - Rs , L - R) ;
tdarc_gi (hdc , 0. , L - R , Rs , - Rs , L - R , Rs , L - R) ;
tmove_gi (hdc , Rs , L - R) ;
tline_gi (hdc , Rs , Rm) ;
trota_gi (0. , 0. , - Pi / 6.) ; /* ... noch einmal 30° */
tline_gi (hdc , - xA , yA) ;
```

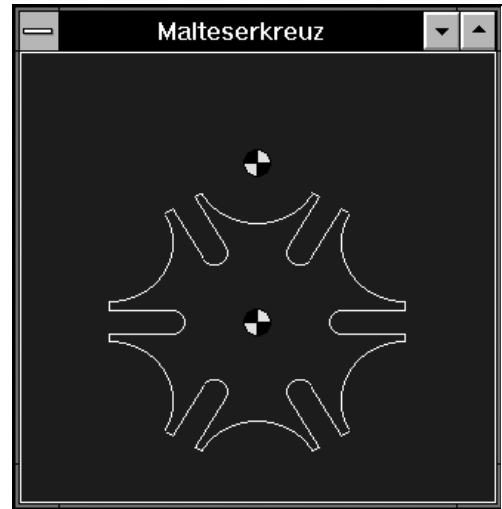
- ♦ Mit **tdarc\_gi** wird der Kreisbogen  $A'-A$  mit dem Mittelpunkt bei  $(0,L)$  gezeichnet, weil noch keine Transformation eingestellt wurde, genau mit den angegebenen Parametern. Danach wird der Punkt  $A$  mit **tmove\_gi** zur "Current Position" (Ausgangspunkt für das Zeichnen der Geraden  $A-C$ ).
- ♦ Bevor die Gerade  $A-C$  gezeichnet wird, stellt **trota\_gi** eine Rotation um  $30^\circ$  um den Nullpunkt ein (negativer Winkel, weil in Uhrzeigerrichtung gedreht wird). Damit können alle Koordinaten für das Zeichnen des Schlitzes bis zum Punkt  $F$  der wesentlich einfacheren vertikalen Lage (wie im Bild auf der vorigen Seite rechts dargestellt) entnommen werden.
- ♦ Das Zeichnen der letzten Geraden  $F-G$  würde die (etwas umständliche) Berechnung der Länge dieses Geradenstücks erfordern. Da wir uns schon vor der Berechnung der Länge des Geradenstücks  $A-C$  "gedrückt" haben, bietet sich der gleiche Trick hier ein zweites Mal an: Mit **trota\_gi** wird "um  $30^\circ$  weitergedreht" (jetzt wird also eine  $60^\circ$ -Rotation vor dem Zeichnen ausgeführt, weil **trota\_gi** inkrementell arbeitet), und für den Zielpunkt des letzten Geradenstücks werden die Koordinaten des Punktes  $A'$  noch einmal verwendet, denn bei  $G$  muß sich ja das nächste Sechstel anschließen.

Da die gesamte eingestellte Transformation jetzt eine  $60^\circ$ -Rotation um den Nullpunkt ist, kann sich unmittelbar das Zeichnen des nächsten Sechstels **mit exakt den gleichen Funktionen, aufgerufen mit unveränderten Argumenten**, anschließen.

Die oben angegebene Sequenz von Zeichenbefehlen muß also nur in eine sechsfach abzuarbeitende Schleife gelegt werden, um das komplette Malteserkreuz zu erzeugen (in **maltesr0.c** sind diese Zeilen bereits vorgesehen, es müssen nur die sie einschließenden Kommentarsymbole entfernt werden):

```
void DrawCross (HDC hdc)
{
    int i ;

    for (i = 0 ; i < 6 ; i++)
    {
        /* Zeichnen eines Sechstels des Malteserkreuzes: ... */
    }
}
```



Das komplette Malteserkreuz

Die Programmversion **maltesr1.c** zeichnet (neben den beiden Drehzapfen zur Orientierung) nur die Kurbel, deren Abmessungen aus der nebenstehenden Skizze zu entnehmen sind.

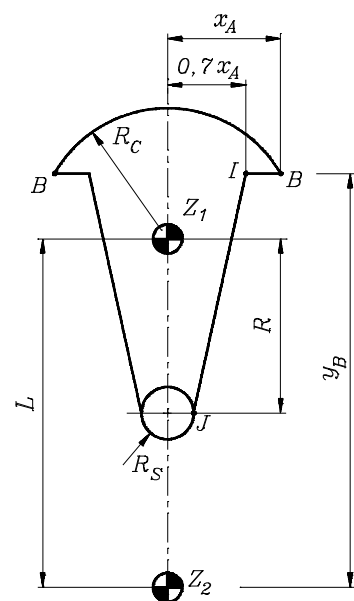
Die Bearbeitung der Botschaft WM\_PAINT ist nur in einer Zeile gegenüber der Programmversion **maltesr0.c** geändert: An Stelle von **DrawCross** wird die Funktion **DrawCrank** aufgerufen.

Natürlich wird die Zeichnung in **DrawCrank** in der (nebenstehend zu sehenden) vertikalen Stellung programmiert. Um die "t...-Funktion" **tmiry\_gi** zu demonstrieren, werden die beiden Geradenstücke, die rechts bzw. links spiegelbildlich paarweise auftreten, in einer zweimal zu durchlaufenden Schleife angeordnet, an deren Ende die Transformation "Spiegeln an der y-Achse" eingeschaltet wird:

```
void DrawCrank (HDC hdc)
{
    int i ;

    for (i = 0 ; i < 2 ; i++)
    {
        tmove_gi (hdc , xA , yB) ;
        tline_gi (hdc , xA * .7 , yB) ;
        tline_gi (hdc , Rs , L - R) ;
        /* trota_gi (0. , L , - OmT) ; */ /* Bis zur y-Achse drehen, ... */
        tmiry_gi ( ) ; /* Spiegeln an der y-Achse, ... */
        /* trota_gi (0. , L , OmT) ; */ /* "Zurueckdrehen" */

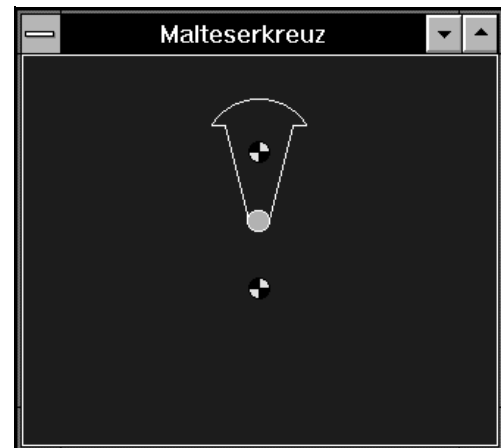
        tdarc_gi (hdc , 0. , L , Rc , xA , yB , - xA , yB) ;
        tcirc_gi (hdc , 0. , L - R , Rs) ;
    }
}
```



Geometrie der Kurbel

Die nebenstehende Abbildung zeigt das Bild, das mit **maltesr1.c** erzeugt wird.

Wenn die gesamte Kurbel einer Transformation unterworfen werden soll (um sie z. B. in horizontaler Lage darzustellen), funktioniert natürlich die einfache Transformation "Spiegeln an der y-Achse" nicht mehr. Dann muß die gleiche Strategie verwendet werden, die im Abschnitt 3.2.3 schon an einem anderen Beispiel demonstriert wurde, hier: "Transformieren, so daß wieder die y-Achse die Spiegelachse ist" ---> "Spiegeln" ---> "Rücktransformieren". In **maltesr1.c** sind die Anweisungen ("herauskommentiert") dafür bereits vorgesehen:



Bei der Bearbeitung von WM\_PAINT wird eine Rotationstransformation (90° im Uhrzeigersinn) um den Drehpunkt der Kurbel vor dem Aufruf von **DrawCrank** eingefügt:

```
tinit_gi () ;                /* Transformation initialisieren */
OmT = - Pi * .5 ;
trot_gi (0. , L , OmT) ;     /* Transformation setzen      */
DrawCrank (hdc) ;           /* Kurbel zeichnen          */
```

Der Winkel **OmT** ist als globale Variable auch in **DrawCrank** verfügbar, so daß die oben genannte Transformationssequenz dort aktiviert werden kann:

```
trot_gi (0. , L , - OmT) ;    /* Bis zur y-Achse drehen, ... */
tmir_gi () ;                 /* Spiegeln an der y-Achse, ... */
trot_gi (0. , L , OmT) ;     /* "Zurueckdrehen"            */
```

Die nebenstehende Abbildung zeigt die Darstellung der Kurbel nach dieser kleinen Änderung.

Die so programmierte Darstellung funktioniert natürlich für jeden beliebigen Winkel. Dies wird in der Version **maltesr2.c** ausgenutzt, um zu zeigen, wie man eine einfache Animation mit den "t...-Funktionen" realisieren kann. Dabei wird folgende Strategie verfolgt:

In **WinMain** wird ein "Timer" angefordert (bei der Bearbeitung der Botschaft WM\_DESTROY wird er wieder freigegeben), der mit "50/1000 Sekunden" auf das kürzeste mögliche Intervall zur Erzeugung von WM\_TIMER-Botschaften eingestellt wird (Wert wird automatisch auf 54,925/1000 Sekunden korrigiert). Das Programm erhält also maximal 18,2 Botschaften WM\_TIMER pro Sekunde, das ist weniger als beim Kinofilm, aber immerhin wird der Eindruck der Bewegung damit vermittelt. Als Winkelinkrement wird (willkürlich) **dOmT = Pi/15** festgelegt, das sind 12° (wenn es der PC schafft, müßte die Kurbel mit etwa 36,4 Umdrehungen pro Minute rotieren).

Die Botschaft WM\_PAINT bereitet die Zeichenaktionen mit dem Initialisieren und dem Setzen der Transformation für die Anfangslage der Kurbel nur noch vor:



Kurbel in transformierter Lage

```

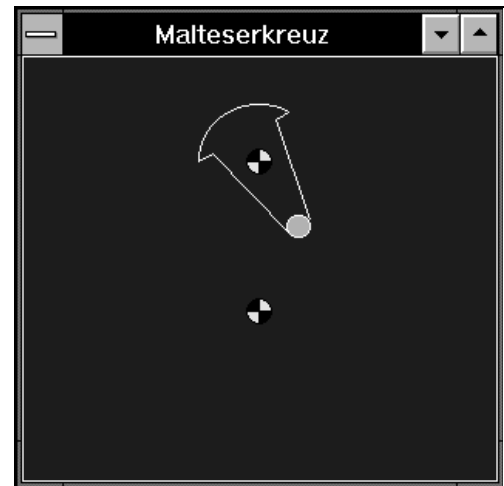
OmT      = - Pi * .5 ;
dOmT     =  Pi / 15. ;

tinit_gi ( ) ;                /* Transformation initialisieren */
trota_gi (0. , L , OmT) ;    /* Transformation fuer Kurbel setzen */

```

Gezeichnet wird dagegen nur noch bei der Auswertung der Botschaft WM\_TIMER, jeweils zweimal.

Zunächst wird mit der Hintergrundfarbe ("Pen" und "Brush") das alte Bild überzeichnet, danach wird der Winkel **OmT** (wird nur für die Transformationen in der Funktion **DrawCrank** benötigt) für die Rotations-Transformation um das Inkrement **dOmT** vergrößert (wenn eine volle Umdrehung absolviert worden ist, wird **OmT** wieder um  $2\pi$  verkleinert). Mit **trota\_gi** wird eine **zusätzliche** Rotation um **dOmT** eingestellt, und das Bild wird neu gezeichnet:



```

case WM_TIMER:

    hdc = GetDC (hwnd) ;

    SelectObject (hdc , hPenBlue) ; /* ... fuer Ueberzeichnen */
    SelectObject (hdc , hBrushBlue) ; /* ... fuer Ueberzeichnen */
    DrawCrank (hdc) ; /* Kurbel ueberzeichnen */

    OmT += dOmT ;
    if (OmT > Pi * 1.5) OmT -= Pi * 2. ;

    SelectObject (hdc , hPenWhite) ; /* ... fuer Neuzeichnen */
    SelectObject (hdc , hBrushCyan) ; /* ... fuer Neuzeichnen */
    trota_gi (0. , L , dOmT) ; /* ... zusaetzhliche Drehung */
    DrawCrank (hdc) ; /* Kurbel neu zeichnen */

    ReleaseDC (hwnd , hdc) ;

    return 0 ;

```

Die Kurbel dreht sich

Das Programm **malteser.c** stellt schließlich sowohl die Kurbel als auch das Malteserkreuz in der Bewegung dar. Dabei sind zwei Probleme zu lösen:

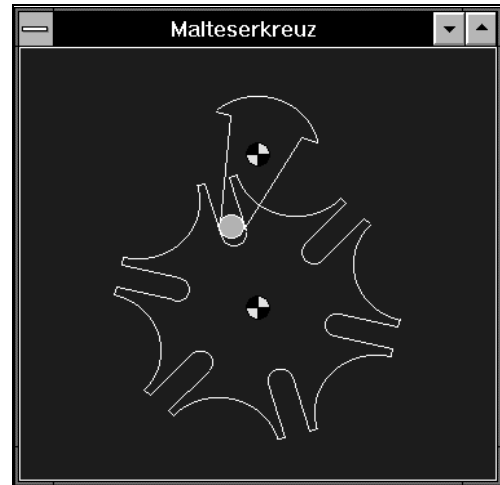
- ♦ Die Synchronisation beider Bewegungen kann nur gelingen, wenn das kinematische Bewegungsgesetz des Malteserkreuz-Getriebes beachtet wird. Das Bewegungsgesetz kann man z. B. in "Dankert/Dankert: Technische Mechanik, computerunterstützt" auf Seite 449 finden. Es findet sich im Programm **malteser.c** in der Zeile, in der der Winkel **Phi**, mit dem die Rotation des Malteserkreuzes beschrieben wird, aus dem Winkel **OmT**, der die Stellung der Kurbel beschreibt, berechnet wird. Bei der Programmierung ist zu beachten, daß dieser Zusammenhang nur gilt, wenn die Kurbel mit dem Malteserkreuz im Eingriff steht, ansonsten bleibt **Phi** ungeändert.
- ♦ Es müssen zwei unterschiedliche Transformationen verwaltet werden (für Kurbel bzw. Malteserkreuz), jede Transformation wird einmal für das Zeichnen und (bei der folgenden WM\_TIMER-Botschaft) für das Überzeichnen der alten Lage benötigt. Um die Transformationsmatrizen nicht stets neu berechnen zu müssen, werden die Funktionen **tgttm\_gi** und **tsttm\_gi** verwendet.

Bei der Auswertung von WM\_PAINT werden die Anfangswerte für die Transformationsmatrizen gesetzt und mit **tgttm\_gi** auf die Felder **tm\_cross** bzw. **tm\_crank** gesichert:

```
static double  tm_cross [9] , tm_crank [9] ; /* ... */

tinit_gi ( ) ;                      /* Transformation initialisieren */
tgttm_gi (tm_cross) ;               /* = Start-Transformation fuer Kreuz */
trota_gi (0. , L , OmT) ;          /* Transformation fuer Kurbel setzen */
tgttm_gi (tm_crank) ;              /* ... und sichern */
```

Bei der Auswertung der Botschaft WM\_TIMER wird dann jeweils die passende Transformationsmatrix gesetzt (mit **tsttm\_gi**), die natürlich entsprechend der Änderungen der Winkel verändert (und mit **tgttm\_gi** nach jeder Änderung gesichert) werden muß. Für die Kurbel wird die Änderung wie in **maltesr2.c** inkrementell mit **trota\_gi** realisiert, die Transformation für das Malteserkreuz wird mit **ttabs\_gi** "gesetzt" (ohne Berücksichtigung vorheriger Transformationen, mit **ttabs\_gi** können gleichzeitig eine Skalierung bezüglich des Nullpunktes, eine Rotation um den Nullpunkt und eine Translation als neue "Initialisierung" der Transformationsmatrix erzeugt werden):



Kurbel und Kreuz rotieren synchron

```
case WM_TIMER:

    hdc = GetDC (hwnd) ;

    SelectObject (hdc , hPenBlue) ; /* ... fuer Ueberzeichnen */
    SelectObject (hdc , hBrushBlue) ; /* ... fuer Ueberzeichnen */
    tsttm_gi (tm_crank) ;           /* Transformation fuer Kurbel */
    DrawCrank (hdc) ;               /* Kurbel ueberzeichnen */

    OmT += dOmT ;
    if (OmT > Pi * 1.5) OmT -= Pi * 2. ;

    if (OmT > - Pi / 3. && OmT < Pi / 3.) /* Kurbel im Eingriff */
        Phi = - atan2 (sin (OmT) , L / R - cos (OmT)) - Pi / 6. ;
    else /* Kreuz in Ruhe */
        SelectObject (hdc , hPenWhite) ;

    tsttm_gi (tm_cross) ;           /* Transformation fuer Kreuz */
    DrawCross (hdc) ;               /* Kreuz ueberzeichnen */

    SelectObject (hdc , hPenWhite) ; /* ... fuer Neuzeichnen */
    SelectObject (hdc , hBrushCyan) ; /* ... fuer Neuzeichnen */
    tsttm_gi (tm_crank) ;           /* Transformation fuer Kurbel */
    trota_gi (0. , L , dOmT) ;      /* mit zusaetzlicher Drehung */
    DrawCrank (hdc) ;               /* Kurbel neu zeichnen */
    tgttm_gi (tm_crank) ;           /* Transformation sichern */

    SelectObject (hdc , hPenWhite) ;
    ttabs_gi (0. , 0. , Phi , 1. , 1. ) ;
    /* Transformation fuer Kreuz */
    DrawCross (hdc) ;               /* Kreuz neu zeichnen */
    tgttm_gi (tm_cross) ;           /* Transformation sichern */

    ReleaseDC (hwnd , hdc) ;

    return 0 ;
```

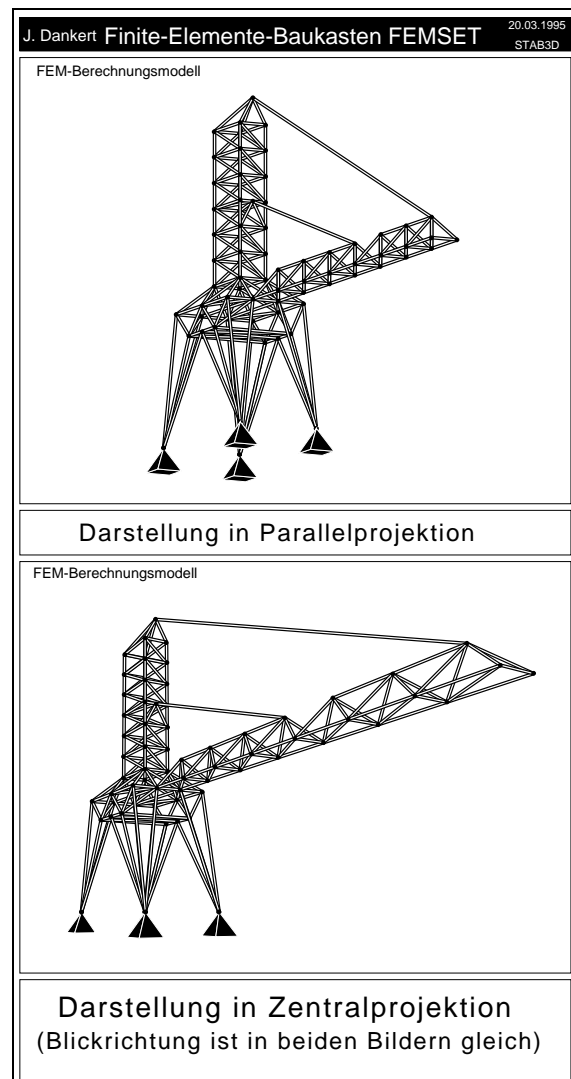


### 3.4 Projektionen

Das Problem, dreidimensionale Objekte auf eine zweidimensionale Zeichenfläche abzubilden, wird durch **Projektion** der dreidimensionalen Punkte auf eine Ebene nach fest zu definierenden Regeln gelöst:

- ♦ Bei der **Zentralprojektion** wird von einem **Projektionszentrum** ("Eye Point") zu jedem Punkt des Körpers ein **Sehstrahl** gezogen. Der Schnittpunkt des Sehstrahls mit der Ebene, auf der die zweidimensionale Abbildung entstehen soll (**Projektionsebene**), ist die Abbildung des 3D-Punktes in der Zeichenebene.
- ♦ Die **Parallelprojektion** kann als Sonderfall der Zentralprojektion angesehen werden, bei der das Projektionszentrum im Unendlichen liegt, so daß alle Sehstrahlen parallel verlaufen.

Während die Zentralprojektion (besonders mit nicht zu großen Entfernungen des Projektionszentrums vom Objekt, vgl. nebenstehendes Bild, "Eye Point" etwa in realer Augenhöhe) den räumlichen Eindruck besonders gut vermittelt, hat die Parallelprojektion unter anderem die angenehme Eigenschaft, vertikale Linien in der Projektionsebene auch vertikal darzustellen.



#### 3.4.1 Allgemeine Theorie der Zentralprojektion

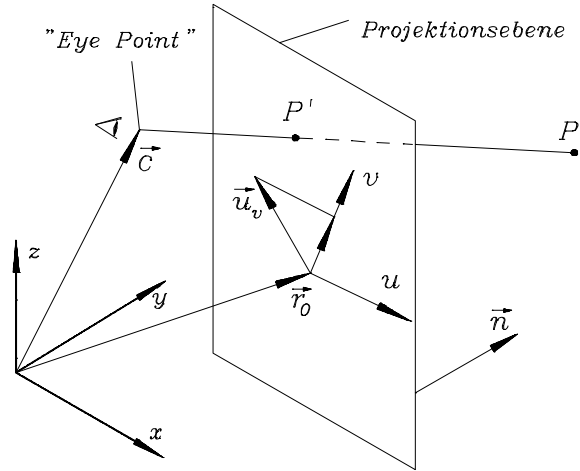
Eine **Zentralprojektion** wird definiert mit

- ♦ dem "**Eye Point**" (Projektionszentrum), beschrieben durch einen Vektor  $\vec{c}$ ,
- ♦ der **Projektionsebene**, die durch einen **Referenzpunkt** (Vektor  $\vec{r}_0$ ) und einen **Normalenvektor**  $\vec{n}$  beschrieben wird, und
- ♦ einem sogenannten "**Up Vector**"  $\vec{u}$ , zur Definition des ebenen  $u$ - $v$ -Koordinatensystems der Bildebene.

Die Vektoren  $\vec{c}$ ,  $\vec{r}_0$ ,  $\vec{n}$ ,  $\vec{u}_v$  werden in einem dreidimensionalen raumfesten  $x$ - $y$ - $z$ -Koordinatensystem ("World Coordinates") beschrieben, für den Vektor  $\vec{n}$  wird vereinbart, daß er zu der Seite der Projektionsebene zeigt, auf der der "Eye Point" nicht liegt.

Das zweidimensionale Koordinatensystem der Bildebene definiert sich wie folgt:

- Der Ursprung fällt mit dem Referenzpunkt zusammen.
- Die Richtung der  $v$ -Achse wird durch die Projektion des "Up Vectors" auf die Projektionsebene festgelegt.
- Die Richtung der  $u$ -Achse wird so festgelegt, daß  $u$ ,  $v$  und  $\vec{n}$  in dieser Reihenfolge ein Linkssystem definieren ("Eye Point" befindet sich vor dem Bildschirm,  $\vec{n}$  zeigt in den Bildschirm hinein und das  $u$ - $v$ -Koordinatensystem liegt in der Bildebene wie die ebenen "User Coordinates", wenn die Projektion des  $\vec{u}_v$ -Vektors auf dem Bildschirm vertikale Richtung hat).



Allgemeine Definition der Zentralprojektion

Ein beliebiger Körperpunkt  $P$ , der im dreidimensionalen  $x$ - $y$ - $z$ -Koordinatensystem durch einen (in der Skizze nicht gezeichneten) Vektor  $\vec{p}$  beschrieben wird und sowohl vor oder hinter und auch in der Projektionsebene liegen darf, wird auf die Projektionsebene abgebildet, indem der Schnittpunkt  $P'$  (nachfolgend durch den Vektor  $\vec{p}'$  beschrieben) berechnet wird, den der **Sehstrahl**, der vom "Eye Point" zum Punkt  $P$  gezogen wird, mit der Projektionsebene hat.

Da die Koordinaten des Punktes  $P'$  für den Zeichenvorgang im ebenen  $u$ - $v$ -Koordinatensystem benötigt werden, werden zunächst zwei Vektoren  $\vec{u}$  und  $\vec{v}$  in Richtung der  $u$ - bzw.  $v$ -Achse bestimmt, die dann zu den Einheitsvektoren  $\vec{u}_e$  und  $\vec{v}_e$  des Bildebenen-Koordinatensystems normiert werden:

Der Normalenvektor  $\vec{n}$  wird durch Division durch seinen Betrag zum Normalen-Einheitsvektor

$$\vec{n}_e = \frac{\vec{n}}{|\vec{n}|} . \quad (3.4.1)$$

Der "Up Vector"  $\vec{u}_v$  kann als Summe des Vektors  $\vec{v}$  und eines Vektors  $k \vec{n}_e$  (mit zunächst noch unbestimmten Faktor  $k$ ) aufgeschrieben werden:

$$\vec{u}_v = \vec{v} + k \vec{n}_e . \quad (3.4.2)$$

Multiplikation dieser Beziehung mit  $\vec{n}_e$  führt wegen  $\vec{n}_e \cdot \vec{v} = 0$  (Vektoren stehen senkrecht aufeinander) und mit  $\vec{n}_e \cdot \vec{n}_e = 1$  auf

$$k = \vec{n}_e \cdot \vec{u}_v , \quad (3.4.3)$$

was in die Beziehung (3.4.2) eingesetzt werden kann. Aus dem daraus berechneten Vektor

$$\vec{v} = \vec{u}_v - (\vec{n}_e \cdot \vec{u}_v) \cdot \vec{n}_e \quad (3.4.4)$$

entsteht schließlich der Einheitsvektor in Richtung der  $\mathbf{v}$ -Achse:

$$\vec{v}_e = \frac{\vec{v}}{|\vec{v}|} . \quad (3.4.5)$$

Der Einheitsvektor in Richtung der  $\mathbf{u}$ -Achse kann nun einfach aus dem Vektorprodukt

$$\vec{u}_e = \vec{n}_e \times \vec{v}_e \quad (3.4.6)$$

berechnet werden.

Mit den (gesuchten) Koordinaten  $\mathbf{u}$  und  $\mathbf{v}$  des Punktes  $\mathbf{P}'$  kann der Vektor  $\vec{p}'$  zu diesem Punkt formal aufgeschrieben werden als

$$\vec{p}' = \vec{r}_0 + u \vec{u}_e + v \vec{v}_e . \quad (3.4.7)$$

Da der Endpunkt des Vektors  $\vec{c}'$  ("Eye Point") und die Punkte  $\mathbf{P}'$  und  $\mathbf{P}$  auf einer Geraden liegen (Sehstrahl), können sich die beiden Differenzvektoren  $(\vec{p}' - \vec{c}')$  und  $(\vec{p} - \vec{c}')$  nur um einen (zunächst ebenfalls noch unbekannten) Faktor unterscheiden:

$$(\vec{p}' - \vec{c}') \lambda = (\vec{p} - \vec{c}') . \quad (3.4.8)$$

Aus den Beziehungen (3.4.7) und (3.4.8) wird der Vektor  $\vec{p}'$  eliminiert (die Koordinaten des Punktes  $\mathbf{P}'$  im dreidimensionalen Koordinatensystem sind ohnehin nicht interessant), und es verbleibt mit

$$(\vec{r}_0 + u \vec{u}_e + v \vec{v}_e - \vec{c}') \lambda = \vec{p} - \vec{c}' \quad (3.4.9)$$

eine Vektorgleichung für die drei Unbekannten  $\mathbf{u}$ ,  $\mathbf{v}$  und  $\lambda$ . Die Vektoren werden durch ihre Komponenten dargestellt:

$$\vec{p} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} , \quad \vec{r}_0 = \begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix} , \quad \vec{c}' = \begin{bmatrix} c_x \\ c_y \\ c_z \end{bmatrix} , \quad \vec{u}_e = \begin{bmatrix} u_{ex} \\ u_{ey} \\ u_{ez} \end{bmatrix} , \quad \vec{v}_e = \begin{bmatrix} v_{ex} \\ v_{ey} \\ v_{ez} \end{bmatrix} , \quad (3.4.10)$$

für die Produkte der unbekannten Koordinaten  $\mathbf{u}$  und  $\mathbf{v}$  mit  $\lambda$  werden neue Unbekannte eingeführt:

$$\bar{u} = u \lambda , \quad \bar{v} = v \lambda . \quad (3.4.11)$$

Dann kann Gleichung (3.4.9) als lineares Gleichungssystem formuliert werden. Die Lösung von

$$\begin{bmatrix} u_{ex} & v_{ex} & x_0 - c_x \\ u_{ey} & v_{ey} & y_0 - c_y \\ u_{ez} & v_{ez} & z_0 - c_z \end{bmatrix} \begin{bmatrix} \bar{u} \\ \bar{v} \\ \lambda \end{bmatrix} = \begin{bmatrix} x - c_x \\ y - c_y \\ z - c_z \end{bmatrix} , \quad (3.4.12)$$

$$\bar{\mathbf{P}} \quad \bar{\mathbf{b}} = \vec{p} - \vec{c}'$$

liefert den Vektor der (ebenen) homogenen Koordinaten des Punktes  $\mathbf{P}'$  im Koordinatensystem der Bildebene:

$$\vec{b} = \begin{bmatrix} \bar{u} \\ \bar{v} \\ \lambda \end{bmatrix} \Rightarrow \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \bar{u}/\lambda \\ \bar{v}/\lambda \end{bmatrix} . \quad (3.4.13)$$

Die Projektion versagt für  $\lambda = 0$ , was folgende Gründe haben kann:

- ♦ Der zu projizierende Punkt  $P$  ist identisch mit dem "Eye Point" ( $\vec{p} = \vec{c}$ ), oder
- ♦ der Sehstrahl und der Normalenvektor der Projektionsebene stehen senkrecht aufeinander.

In beiden Fällen ist natürlich auch keine sinnvolle Projektion möglich. Formal liefert die Projektion auch Bildkoordinaten  $u$  und  $v$ , wenn der Körperpunkt  $P$  hinter dem "Eye Point" liegt. Diese Fälle, die sich durch ein negatives  $\lambda$  äußern, sollten aussortiert werden.

### 3.4.2 Allgemeine Theorie der Parallelprojektion

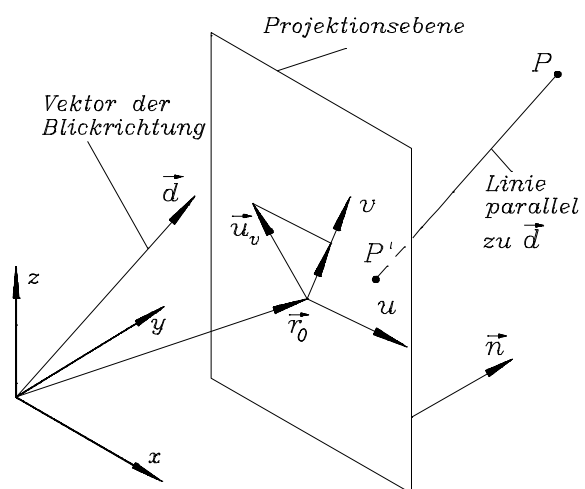
Eine **Parallelprojektion** wird definiert durch

- ♦ den "Vektor der Blickrichtung"  $\vec{d}$ ,
- ♦ die **Projektionsebene**, die durch einen **Referenzpunkt** (Vektor  $\vec{r}_0$ ) und einen **Normalenvektor**  $\vec{n}$  beschrieben wird, und
- ♦ einen sogenannten "**Up Vector**"  $\vec{u}_v$  zur Definition des ebenen  $u$ - $v$ -Koordinatensystems der Bildebene.

Die Vektoren  $\vec{d}$ ,  $\vec{r}_0$ ,  $\vec{n}$ ,  $\vec{u}_v$  werden in einem dreidimensionalen raumfesten  $x$ - $y$ - $z$ -Koordinatensystem beschrieben, für den Vektor  $\vec{n}$  wird vereinbart, daß er einen spitzen Winkel mit dem "Vektor der Blickrichtung" bildet.

Die Definition des zweidimensionalen  $u$ - $v$ -Koordinatensystems der Bildebene erfolgt exakt nach der Vorschrift, die für die Zentralprojektion beschrieben wurde, so daß die Formeln (3.4.1) bis (3.4.7) unverändert gelten.

Ein beliebiger Körperpunkt  $P$ , der im dreidimensionalen  $x$ - $y$ - $z$ -Koordinatensystem durch einen (in der Skizze nicht gezeichneten) Vektor  $\vec{p}$  beschrieben wird und sowohl vor oder hinter und auch in der Projektionsebene



Allgemeine Definition der Parallelprojektion

liegen darf, wird auf die Projektionsebene abgebildet, indem der Schnittpunkt  $P'$  (nachfolgend durch den Vektor  $\vec{p}'$  beschrieben) berechnet wird, den **eine Parallele zum Vektor der Blickrichtung** durch  $P$  mit der Projektionsebene hat.

Der Vektor zum Punkt  $P$  kann also als Summe des Vektors zum Punkt  $P'$  und dem mit einem (zunächst unbekannten) Faktor multiplizierten "Vektor der Blickrichtung"  $\vec{d}$  aufgeschrieben werden:

$$\vec{p} = \vec{p}' + \alpha \vec{d} . \quad (3.4.14)$$

Da auch Formel (3.4.7) ihre Gültigkeit behält, kann aus (3.4.7) und (3.4.14) der Vektor  $\vec{p}'$  (ähnlich zum Vorgehen bei der Zentralprojektion) eliminiert werden. Man erhält mit

$$\vec{r}_0 + u \vec{u}_e + v \vec{v}_e = \vec{p} - \alpha \vec{d} \quad (3.4.15)$$

eine Vektorgleichung für die drei Unbekannten  $u$ ,  $v$  und  $\alpha$ . Die Vektoren werden durch ihre Komponenten dargestellt:

$$\vec{p} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \quad \vec{r}_0 = \begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix}, \quad \vec{d} = \begin{bmatrix} d_x \\ d_y \\ d_z \end{bmatrix}, \quad \vec{u}_e = \begin{bmatrix} u_{ex} \\ u_{ey} \\ u_{ez} \end{bmatrix}, \quad \vec{v}_e = \begin{bmatrix} v_{ex} \\ v_{ey} \\ v_{ez} \end{bmatrix}, \quad (3.4.16)$$

und Gleichung (3.4.15) kann als lineares Gleichungssystem formuliert werden. Die Lösung von

$$\begin{bmatrix} u_{ex} & v_{ex} & d_x \\ u_{ey} & v_{ey} & d_y \\ u_{ez} & v_{ez} & d_z \end{bmatrix} \begin{bmatrix} u \\ v \\ \alpha \end{bmatrix} = \begin{bmatrix} x - x_0 \\ y - y_0 \\ z - z_0 \end{bmatrix}, \quad (3.4.17)$$

$$\vec{P}_p \quad \vec{b}_p = \vec{p} - \vec{r}_0$$

liefert unmittelbar die Koordinaten  $u$  und  $v$  des Punktes  $P'$  im Koordinatensystem der Bildebene (und den nicht interessierenden Parameter  $\alpha$ ). Man beachte, daß die Lösung von (3.4.17) im Gegensatz zur Lösung von (3.4.12) keine homogenen Koordinaten liefert, sondern direkt die kartesischen Koordinaten des ebenen  $u$ - $v$ -Koordinatensystems.

### 3.4.3 Empfehlungen zur Definition von Projektionen

Die volle Allgemeinheit, mit der die Zentralprojektion und die Parallelprojektion in den Abschnitten 3.4.1 und 3.4.2 beschrieben wurden, sollte nicht dazu verführen, beliebige Kombinationen der jeweils vier Vektoren, die eine Projektion definieren, zu verwenden. Folgende Empfehlungen können allgemein gegeben werden:

- ♦ Bei der **Zentralprojektion** wird der Fußpunkt des Lotes vom "Eye Point" auf die Projektionsebene als "Hauptpunkt" bezeichnet. Es ist empfehlenswert, den Referenzpunkt der Projektionsebene (Vektor  $\vec{r}_0$ ) mit dem Hauptpunkt zusammenfallen zu lassen. Da der Referenzpunkt der Ursprung des ebenen Koordinatensystems der

Bildebene ist, steht die Projektionsebene dann senkrecht zum Sehstrahl auf den Nullpunkt des  $u$ - $v$ -Koordinatensystems.

- ◆ Bei der **Parallelprojektion** sollte die Projektionsebene senkrecht zum "Vektor der Blickrichtung" gelegt werden. Dann ist der Normalenvektor, der die Projektionsebene definiert, parallel zum "Vektor der Blickrichtung".
- ◆ Im allgemeinen ist ein "Up Vector", der parallel zur  $z$ -Achse des dreidimensionalen Koordinatensystems liegt, eine gute Wahl (vermittelt den Eindruck, als würde der Beobachter auf einer zur  $x$ - $y$ -Ebene parallelen Ebene aufrecht stehen). Ein solcher "Up Vector" ist allerdings nicht möglich, wenn der "Eye Point" der Zentralprojektion selbst auf der  $z$ -Achse liegt, oder der "Vektor der Blickrichtung" der Parallelprojektion parallel zu  $z$ -Achse ist.
- ◆ Die in technischen Zeichnungen üblichen Darstellungen (Vorderansicht, Seitenansicht, Draufsicht) sind als Sonderfälle der Parallelprojektion zu realisieren.

### 3.4.4 Vereinfachte Definition der Projektionen im GIW

Die allgemeinen Definitionen von Zentralprojektion und Parallelprojektion mit jeweils vier Vektoren, wie sie in den Abschnitten 3.4.1 und 3.4.2 vorgestellt wurden, werden in der GIW-Toolbox verwaltet und können bei Bedarf für spezielle Untersuchungen genutzt werden. Wenn nicht wirklich gute Gründe gegen die nachfolgend beschriebenen vereinfachten Definitionen sprechen, die eine Projektion mit nur zwei Vektoren eindeutig beschreiben, dann sollten diese benutzt werden. Die GIW-Funktionen, die in den folgenden Abschnitten vorgestellt werden, arbeiten mit diesen sinnvollen Einschränkungen.

#### Definition einer Zentralprojektion in der GIW-Toolbox:

- ◆ Es werden nur der "**Eye Point**" (Projektionszentrum) und ein Punkt der Projektionsebene ("**Referenzpunkt**") im raumfesten  $x$ - $y$ - $z$ -Koordinatensystem ("World Coordinates") definiert.
- ◆ Die Projektionsebene wird von den GIW-Funktionen automatisch so gelegt, daß sie senkrecht zur Verbindungslinie vom "Eye Point" zum Referenzpunkt liegt (der Normalenvektor hat also die Richtung dieser Verbindungslinie, der Referenzpunkt ist gleichzeitig der sogenannte "Hauptpunkt" der Projektion).
- ◆ Als "Up Vector" wird i. a. automatisch die  $z$ -Achse gewählt (dies vermittelt den Eindruck, als würde der Beobachter auf einer zur  $x$ - $y$ -Ebene parallelen Ebene aufrecht stehen). Wenn der "Eye Point" selbst auf der  $z$ -Achse liegt, wird entweder die positive  $y$ -Achse ("Eye Point" liegt auf der positiven  $z$ -Achse, "Draufsicht") oder die negative  $y$ -Achse ("Eye Point" liegt auf der negativen  $z$ -Achse) zum "Up Vector".

Diese Definition mit zwei Vektoren ist nicht nur eindeutig, sie erfüllt auch die im vorigen Abschnitt gegebenen Empfehlungen. Eine entsprechende Aussage gilt auch für die folgende "Zwei-Vektor-Definition" der Parallelprojektion.

#### Definition einer Parallelprojektion in der GIW-Toolbox:

- ◆ Es werden nur der "**Vektor der Blickrichtung**" und ein Punkt der Projektionsebene ("**Referenzpunkt**") im raumfesten  $x$ - $y$ - $z$ -Koordinatensystem ("World Coordinates") definiert.
- ◆ Die Projektionsebene wird von den GIW-Funktionen automatisch so gelegt, daß sie senkrecht zum Vektor der Blickrichtung liegt (der Normalenvektor hat also die gleiche Richtung wie der Vektor der Blickrichtung).
- ◆ Als "Up Vector" wird i. a. automatisch die  $z$ -Achse gewählt (dies vermittelt den Eindruck, als würde der Beobachter auf einer zur  $x$ - $y$ -Ebene parallelen Ebene aufrecht stehen). Wenn der "Vektor der Blickrichtung" parallel zur  $z$ -Achse ist, wird entweder die positive  $y$ -Achse ("Vektor der Blickrichtung" hat negative  $z$ -Komponente, "Draufsicht") oder die negative  $y$ -Achse ("Vektor der Blickrichtung" hat positive  $z$ -Komponente) zum "Up Vector".

### 3.4.5 Die "pr...-Funktionen" des GIW

Die sogenannten "**pr...-Funktionen**" (alle zugehörigen Funktionsnamen beginnen mit **pr**) lassen sich in zwei Gruppen unterteilen:

- ◆ Die "**vorbereitenden pr...-Funktionen**" definieren bzw. ändern die gültige Projektion (es kann immer nur eine Projektion gültig sein, entweder eine Zentral- oder eine Parallelprojektion).
- ◆ Den "**zeichnenden pr...-Funktionen**" werden die Punkte in "World Coordinates" übergeben (bezogen auf ein festes  $x$ - $y$ - $z$ -Koordinatensystem), die vor der Zeichenaktion mit der aktuellen Projektion umgerechnet und damit zu zweidimensionalen "User Coordinates" werden.

#### Der erste Aufruf einer "pr...-Funktion" muß immer

```
prini_gi ( ) ;
```

für die Initialisierung aller Projektionsparameter sein. Ersatzweise kann auch

```
ptini_gi ( ) ;
```

aufgerufen werden, die selbst **prini\_gi** und die im Abschnitt 3.3 beschriebene Funktion **tinit\_gi** aufruft (also Projektionen **und** Transformationen initialisiert).

- ♦ Weil die Initialisierungs-Funktionen keinen Device Context benötigen, können sie bereits beim Erzeugen eines Fensters (Botschaft WM\_CREATE) oder sogar in **WinMain** aufgerufen werden.
- ♦ Die Funktion **prini\_gi** definiert eine Zentral- und eine Parallel-Projektion, die (bei nicht "zu großen" Objekten) sehr ähnliche Ansichten erzeugen, weil als Referenzpunkt der Projektionsebene für beide Projektionen der Nullpunkt des 3D-Koordinatensystems eingestellt wird und sich der "Eye Point"-Vektor (Zentralprojektion) vom "Blickrichtungs"-Vektor (Parallelprojektion) nur durch das Vorzeichen unterscheidet. Nach der Initialisierung ist zunächst die Parallelprojektion gültig ("vorsichtshalber", bei einer Zentralprojektion besteht immer die Gefahr, daß man mit dem "Eye Point" im darzustellenden Objekt liegt).

Das Programm **projec1.c** zeigt das Zusammenspiel der wichtigsten "**pr...-Funktionen**" am ganz einfachen Beispiel (dargestellt werden nur drei gerade Linien, die entlang der Koordinatenachsen des 3D-Systems verlaufen), gelistet werden hier nur die wesentlichen Passagen des Programms.

Die Botschaft WM\_CREATE wird benutzt, um die Projektions-Parameter zu initialisieren, die Werte werden erfragt und globalen Variablen zugewiesen (um sie für Änderungen über Dialoge verfügbar zu haben):

```
case WM_CREATE :

    prini_gi ( ) ;                /* ... initialisiert Projektionen      */
    prgte_gi (&xwe , &ywe , &zwe) ; /* ... liefert "Eye Point"              */
    prgtr_gi (&xwr , &ywr , &zwr) ; /* ... liefert Referenzpunkt            */
    prgtd_gi (&xwd , &ywd , &zwd) ; /* ... liefert "Blickrichtungs-Vektor" */

    hPenRed   = CreatePen (PS_SOLID , 1 , mkrrgb_gi (GI_RED)) ;
    hPenBlue  = CreatePen (PS_SOLID , 1 , mkrrgb_gi (GI_BLUE)) ;

    return 0 ;
```

Bei der Bearbeitung der Botschaft WM\_PAINT werden zunächst die eingestellten Standard-Parameter benutzt. Der aktuelle Projektionstyp wird mit der Funktion **prgtp\_gi** erfragt, die drei geraden Linien werden (mit unterschiedlichen Farben) mit **prmov\_gi** und **prlin\_gi** gezeichnet. Da **prmov\_gi** und **prlin\_gi** die übergebenen 3D-Koordinaten ("World Coordinates") in 2D-"User Coordinates" umrechnen, muß ein geeigneter Bereich für die "User Coordinates" eingestellt werden, was hier mit **stuci\_gi** erledigt wird (vgl. Abschnitt 2.2.2, wie man zu geeigneten Grenzwerten für den **stuci\_gi**-Aufruf kommt, wird später beschrieben):

```
case WM_PAINT :

    hdc = gstrt_gi (hwnd , cxClient , cyClient) ;

    if (prgtp_gi ( ) == 1)
        TextOut (hdc , 10 , 10 , "Zentralprojektion " , 20);
    else
        TextOut (hdc , 10 , 10 , "Parallelprojektion " , 20);

    stuci_gi (- 15. , - 5. , 15. , 10. , 0.) ;

    prmov_gi (hdc , -5. , 0. , 0.) ;
    prlin_gi (hdc , 10. , 0. , 0.) ;

    SelectObject (hdc , hPenRed) ;
    prmov_gi (hdc , 0. , -5. , 0.) ;
    prlin_gi (hdc , 0. , 10. , 0.) ;
```



```

SelectObject (hdc , hPenBlue) ;
prmov_gi (hdc , 0. , 0. , -5.) ;
prlin_gi (hdc , 0. , 0. , 10.) ;

gstop_gi (hwnd) ;

return 0 ;

```

Die "zeichnenden pr...-Funktionen" (im Programm **projec1.c** sind dies **prmov\_gi** und **prlin\_gi**) arbeiten wie die "u...-Funktionen" mit ähnlichen Namen (z. B.: **umove\_gi** bzw. **uline\_gi**) mit folgenden Unterschieden:

- ♦ Den "zeichnenden pr...-Funktionen" werden 3D-Punkte übergeben ("World Coordinates"), die von den Funktionen auf "User Coordinates" (unter Benutzung der aktuellen Projektion) umgerechnet werden. Das kann fehlschlagen (z. B.: Punkt befindet sich "hinter dem Eye Point"), dann wird die Aktion nicht ausgeführt, und ...
- ♦ die "pr...-Funktion" signalisiert den Mißerfolg mit dem Return-Wert **0** (bei Erfolg ist der Return-Wert **1**). Im Programm **projec1.c** werden die Return-Werte der "zeichnen-den pr...-Funktionen" nicht ausgewertet.

Die Parameter der Projektionen und der Projektionstyp können über Dialoge geändert werden. Von der Auswertung der Änderungen werden nur zwei Beispiele gelistet:

```

case WM_COMMAND:
    switch (wParam)
    {
        case 100:
            /* ... aendert Projektionstyp: */

            if (prgtp_gi () == GI_CENTRAL) projn_gi (GI_PARALLEL) ;
            else projn_gi (GI_CENTRAL) ;

            InvalidateRect (hwnd , NULL , TRUE) ;

            return 0 ;

        case 200:
            /* "Eye Point" aendern: */

            if (DialogBox (hActInstance , "EYEPOINT" , hwnd ,
                MakeProcInstance (DialogProcE , hActInstance)))
            {
                if (!prste_gi (xwe , ywe , zwe))
                    MessageBox (hwnd , "Setzen der Projektion ist misslungen!" ,
                        "Sorry" , MB_ICONINFORMATION | MB_OK) ;
                else
                    InvalidateRect (hwnd , NULL , TRUE) ;
            }

            return 0 ;      /* ... */
    }

```

- ♦ Bei Wahl des Menüangebots **Projektionstyp ändern** (WM\_COMMAND mit wParam = 100) wird der aktuelle Projektionstyp mit **prgtp\_gi** erfragt, und mit **projn\_gi** wird der jeweils andere Projektionstyp eingestellt.
- ♦ Nach Änderung der Koordinaten des "Eye Points" über das Menüangebot **Eye Point** (WM\_COMMAND mit wParam = 200) wird mit **prste\_gi** der neue "Eye Point" gesetzt und gegebenenfalls auf "Zentralprojektion" umgeschaltet. Diese Aktion kann fehlschlagen (z. B.: "Eye Point" und Referenzpunkt sind identisch), dann bleibt die alte Einstellung komplett erhalten.

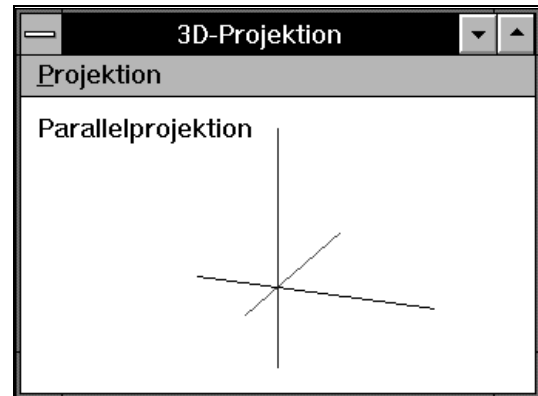
- ♦ Die (hier nicht gelisteten) Änderungen von "Blickrichtungs"-Vektor (mit **prstd\_gi**) und Referenzpunkt (mit **prstr\_gi**) werden nach identischen Strategien ausgeführt.

Die nebenstehende Abbildung zeigt die Zeichnung mit der Standard-Einstellung, wie sie nach dem Programmstart verwendet wird.

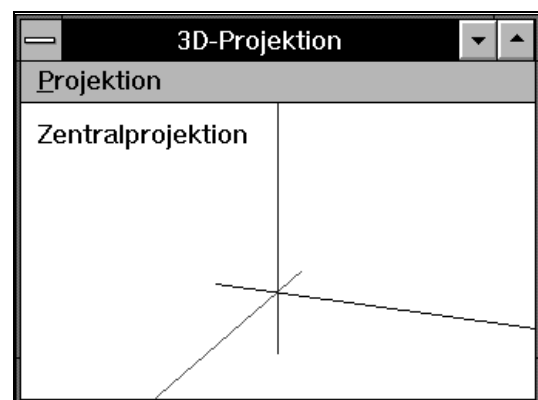
Nach Umschaltung auf "Zentralprojektion" zeigt das Bild keine sichtbaren Veränderungen, weil "in die gleiche Richtung und aus genügend großer Entfernung" geblickt wird, denn die Standard-Einstellung für den "Eye Point" ist die "negative Blickrichtung" (2000;-5000;2000).

Wenn allerdings der "Eye Point" auf (2;-5;2) geändert wird (das ist immer noch die gleiche Richtung, aber der Betrachter ist deutlich näher am Objekt), ändert sich die Darstellung (untere Abbildung). Man sollte bei der Zentralprojektion unbedingt vermeiden, "zu nah" an das Objekt heranzugehen.

Eine entsprechende Änderung des "Blickrichtungs"-Vektors der Parallelprojektion hätte dagegen überhaupt keine Auswirkung, weil sich der Betrachter bei dieser Projektion immer "im Unendlichen" befindet (der Vektor gibt nur eine Richtung vor, sein absoluter Betrag hat auf die Darstellung keinen Einfluß).



"Blickrichtung" (-2000;5000;-2000)



"Eye Point" bei (2;-5;2)

### 3.5 3D-Transformationen

Nachfolgend werden die Transformationsformeln für die geometrische Transformation und die Koordinatentransformation eines 3D-Punktes angegeben. Es gelten alle Aussagen, die im Abschnitt 3.2 (ebene Transformationen) gemacht wurden, zusätzlich ist zu beachten:

- ♦ Die Rotation bedeutet im Raum immer "Rotation um eine vorgegebene Achse" (das gilt natürlich auch für die Ebene, dort ist von der Rotationsachse aber immer nur ein Punkt zu sehen). Als "elementare Rotationen" werden die Rotationen um die drei Koordinatenachsen betrachtet (im Gegensatz zu einer "Elementar-Rotation" in der Ebene, der Rotation um den Nullpunkt). Rotationen um beliebige Achsen sind durch "Verknüpfung von Transformationen" (vgl. Abschnitt 3.2.3) zu realisieren.
- ♦ Spiegelung ist im Raum "Spiegelung an einer Ebene", Elementartransformationen spiegeln an  $x$ - $y$ -Ebene,  $x$ - $z$ -Ebene und  $y$ - $z$ -Ebene. Die besonders einfachen Transformationsmatrizen dafür werden nicht angegeben. Sie unterscheiden sich von der Einheitsmatrix jeweils um genau ein Minuszeichen auf der Hauptdiagonalen.

### 3.5.1 Geometrische Transformation mit homogenen Koordinaten

**Translation um  $t_x$ ,  $t_y$  und  $t_z$ :**

$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \bar{T}_G \bar{x}$$

**Skalierung bezüglich des Nullpunktes um  $s_x$ ,  $s_y$  und  $s_z$ :**

$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \bar{S}_G \bar{x}$$

**Rotation um die x-Achse mit dem Winkel  $\varphi_x$ :**

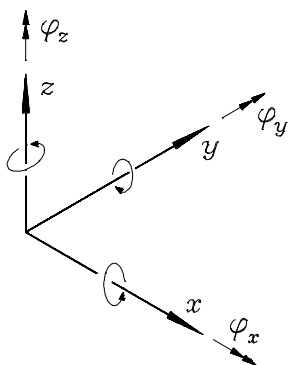
$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \varphi_x & -\sin \varphi_x & 0 \\ 0 & \sin \varphi_x & \cos \varphi_x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \bar{R}_{Gx} \bar{x}$$

**Rotation um die y-Achse mit dem Winkel  $\varphi_y$ :**

$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \varphi_y & 0 & \sin \varphi_y & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \varphi_y & 0 & \cos \varphi_y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \bar{R}_{Gy} \bar{x}$$

**Rotation um die z-Achse mit dem Winkel  $\varphi_z$ :**

$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \varphi_z & -\sin \varphi_z & 0 & 0 \\ \sin \varphi_z & \cos \varphi_z & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \bar{R}_{Gz} \bar{x}$$



Definition positiver Drehwinkel

Mit diesen Formeln wird die **Bewegung eines Punktes im festen Koordinatensystem** beschrieben.

### 3.5.2 Koordinatentransformation mit homogenen Koordinaten

**Translation um  $t_x$ ,  $t_y$  und  $t_z$ :**

$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & -t_x \\ 0 & 1 & 0 & -t_y \\ 0 & 0 & 1 & -t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \bar{T}_K \bar{x}$$

**Skalierung bezüglich des Nullpunktes um  $s_x$ ,  $s_y$  und  $s_z$ :**

$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1/s_x & 0 & 0 & 0 \\ 0 & 1/s_y & 0 & 0 \\ 0 & 0 & 1/s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \bar{S}_K \bar{x}$$

**Rotation um die x-Achse mit dem Winkel  $\varphi_x$ :**

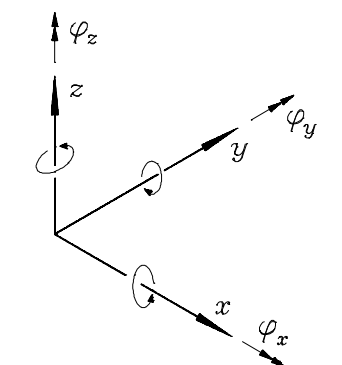
$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \varphi_x & \sin \varphi_x & 0 \\ 0 & -\sin \varphi_x & \cos \varphi_x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \bar{R}_{Kx} \bar{x}$$

**Rotation um die y-Achse mit dem Winkel  $\varphi_y$ :**

$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \varphi_y & 0 & -\sin \varphi_y & 0 \\ 0 & 1 & 0 & 0 \\ \sin \varphi_y & 0 & \cos \varphi_y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \bar{R}_{Ky} \bar{x}$$

**Rotation um die z-Achse mit dem Winkel  $\varphi_z$ :**

$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \varphi_z & \sin \varphi_z & 0 & 0 \\ -\sin \varphi_z & \cos \varphi_z & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \bar{R}_{Kz} \bar{x}$$



Definition positiver Drehwinkel

Mit diesen Formeln werden die Veränderungen der Koordinaten eines festen Punktes bei **Bewegung des Koordinatensystem** beschrieben.

### 3.5.3 "t3...-Funktionen" und "pt...-Funktionen"

Die "t3...-Funktionen" (alle Funktionsnamen beginnen mit **t3**) sind die 3D-Versionen der im Abschnitt 3.3 vorgestellten "vorbereitenden t...-Funktionen" (es wird eine Transformation definiert oder geändert). Die "pt...-Funktionen" (alle zugehörigen Funktionsnamen beginnen mit **pt**) sind die 3D-Versionen der im Abschnitt 3.3 vorgestellten "zeichnenden t...-Funktionen" und der zugehörigen Hilfsfunktionen (vor der eigentlichen Zeichenaktion werden die Koordinaten einer Transformation unterworfen). Allerdings sind "zeichnende 3D-Funktionen" natürlich nur sinnvoll, wenn außerdem eine Projektion auf die Zeichenfläche ausgeführt wird (**pt** steht für "Projektion und Transformation").

Das Prinzip des Arbeitens mit den Funktionen, die eine Transformation setzen, ändern und auswerten, ist völlig analog zur Arbeitsweise, die im Abschnitt 3.3 vorgestellt wurde:

- ♦ Vor der ersten Verwendung einer "t3...- bzw. pt...-Funktion" müssen alle Projektions- und Transformationsparameter initialisiert werden. Dies erledigt die Funktion **ptini\_gi** (erledigt die Arbeit von **tinit\_gi** und **prini\_gi**).
- ♦ Bei der Initialisierung wird die "Einheits-Transformation" eingestellt, so daß alle "pt...-Funktionen" zunächst nicht anders arbeiten als die entsprechenden "pr...-Funktionen", die im Abschnitt 3.4.5 vorgestellt wurden.
- ♦ Mit den "t3...-Funktionen" wird die 3D-Transformation modifiziert. Die jeweils gültige Transformation wird für die Punkte, die den "pt...-Funktionen" übergeben werden, vor der eigentlichen Zeichenaktion ausgeführt. Man beachte, daß (wie bei den 2D-"t...-Funktionen") nicht die Datenstruktur, die das Objekt beschreibt, geändert wird. Die Koordinaten werden **nur für die Darstellung modifiziert**.

Das Programm **projec2.c** ist eine Modifikation des Programms **projec1.c** aus dem Abschnitt 3.4.5, allerdings wurden alle "pr...-Funktionen" durch die entsprechenden "pt...-Funktionen" ersetzt. Zusätzlich wurde die Möglichkeit vorgesehen, die 3D-Transformation dadurch zu ändern, daß (inkrementell) Translationen in Richtung der 3 Koordinatenachsen und Rotationen um diese Achsen über die Tastatur eingebracht werden können, so daß das dargestellte Objekt beliebig verschoben und gedreht werden kann.

Die Auswertung der Botschaft WM\_CREATE unterscheidet sich von **projec1.c** nur dadurch, daß in **projec2.c** mit **ptini\_gi** initialisiert wird:

```
case WM_CREATE :
    ptini_gi ( ) ;                               /* Initialisieren und */
    prgte_gi (&xwe , &ywe , &zwe) ;             /* Erfragen der      */
    prgtr_gi (&xwr , &ywr , &zwr) ;             /* Projektions-      */
    prgtd_gi (&xwd , &ywd , &zwd) ;             /* Parameter         */

    hPenRed   = CreatePen (PS_SOLID , 1 , mkr gb_gi (GI_RED)) ;
    hPenBlue  = CreatePen (PS_SOLID , 1 , mkr gb_gi (GI_BLUE)) ;

    return 0 ;
```

Bei der Auswertung der Botschaft WM\_PAINT wurde die Textausgabe (Hinweis auf den eingestellten Projektionstyp) herausgenommen, ansonsten wurden die "Move to"- und die "Line to"-Funktionen **prmov\_gi** und **prlin\_gi**, die "nur projizieren" durch die "transformierenden und projizierenden" Funktionen **ptmov\_gi** bzw. **ptlin\_gi** ersetzt:

```

case WM_PAINT :

    hdc = gstrt_gi (hwnd , cxClient , cyClient) ;

    stuci_gi (- 15. , - 5. , 15. , 10. , 0.) ;

    ptmov_gi (hdc , -5. , 0. , 0.) ;
    ptlin_gi (hdc , 10. , 0. , 0.) ;

    SelectObject (hdc , hPenRed) ;
    ptmov_gi (hdc , 0. , -5. , 0.) ;
    ptlin_gi (hdc , 0. , 10. , 0.) ;

    SelectObject (hdc , hPenBlue) ;
    ptmov_gi (hdc , 0. , 0. , -5.) ;
    ptlin_gi (hdc , 0. , 0. , 10.) ;

    gstop_gi (hwnd) ;

    return 0 ;

```

Nach dem Programmstart sieht man das gleiche Bild wie nach dem Start von **projec1.c**. Durch Drücken der Tasten **x**, **y**, **z** bzw. **X**, **Y**, **Z** wird eine Rotationstransformation um die jeweilige Achse um  $10^\circ$  bzw.  $-10^\circ$  (zusätzlich zur aktuellen Transformation) eingestellt. Über das Menüangebot **Transformation** kann dies geändert werden: Wenn die Option **Translation** gewählt wird, bewirkt nachfolgendes Drücken einer der genannten Tasten, daß eine Translation um **1** bzw. **-1** in Richtung der jeweiligen Achse (zusätzlich) eingestellt wird.

Die Modifikation der aktuellen Transformation wurde als Auswertung der Botschaft WM\_CHAR programmiert. In Abhängigkeit von einem Schalter **rot** (Voreinstellung: **1**), der über das genannte Menüangebot umgestellt werden kann, werden entweder eine zusätzliche Rotation mit **t3rot\_gi** oder eine zusätzliche Translation mit **t3trn\_gi** eingestellt:

```

case WM_CHAR :

    switch (wParam)
    {
        case 'x' : rot ? t3rot_gi (pi / 18. , GI_AXISX) :
                        t3trn_gi (1. , 0. , 0.) ;
                        break ;
        case 'X' : rot ? t3rot_gi (- pi / 18. , GI_AXISX) :
                        t3trn_gi (- 1. , 0. , 0.) ;
                        break ;
        case 'y' : rot ? t3rot_gi (pi / 18. , GI_AXISY) :
                        t3trn_gi (0. , 1. , 0.) ;
                        break ;
        case 'Y' : rot ? t3rot_gi (- pi / 18. , GI_AXISY) :
                        t3trn_gi (0. , - 1. , 0.) ;
                        break ;
        case 'z' : rot ? t3rot_gi (pi / 18. , GI_AXISZ) :
                        t3trn_gi (0. , 0. , 1.) ;
                        break ;
        case 'Z' : rot ? t3rot_gi (- pi / 18. , GI_AXISZ) :
                        t3trn_gi (0. , 0. , - 1.) ;
                        break ;
    }

    InvalidateRect (hwnd , NULL , TRUE) ;

    return 0 ;

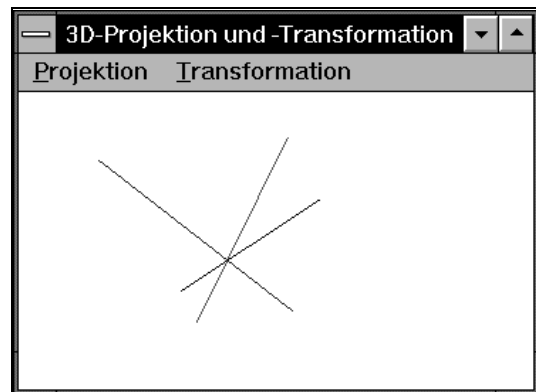
```

Da alle Transformationen "addiert" werden, kann das Objekt in jede beliebige Lage im Raum gebracht werden.

Die nebenstehende Abbildung zeigt ein Bild, das sich nach Ausführung mehrerer Rotationen und Translationen ergab.

Wenn man "den Finger auf einer Taste läßt", ergibt sich der Eindruck einer einfachen Animation (empfehlenswert, wenn "Rotation" eingestellt ist, sonst ist das Objekt sehr schnell verschwunden).

Natürlich läßt sich jede beliebige Ansicht des Objekts auch durch Änderung der Projektions-Parameter erreichen (Bewegung des Betrachters, nicht des Objekts), aber das ist eine andere Denkweise für den Programm-Benutzer.



Programm projec2.c

### 3.6 Darstellung von "Drahtmodellen"

Das Programm **draht1.c** 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. Dazu werden die GIW-Funktionen **flini\_gi** (Initialisieren des "File-Namen-Dialogs", muß einmal aufgerufen werden), **flodl\_gi** (erledigt den typischen Windows-Dialog für die Auswahl eines File-Namens) und **rddfl\_gi** (liest die Datei) verwendet.

Die Funktion **rddfl\_gi** ist spezialisiert auf das Lesen von Dateien, die sehr einfache Objekte durch "Elemente" und "Knoten" beschreiben. "Knoten" sind Punkte, die durch jeweils 3 Koordinaten in einem kartesischen Koordinatensystem definiert werden, zu jedem "Element" soll zunächst die gleiche Anzahl von Knoten gehören. In der nebenstehenden Box ist ein Beispiel einer solchen Datei zu sehen, die das Drahtmodell eines Würfels mit der Kantenlänge 5 beschreibt (Mittelpunkt des Würfels liegt im Ursprung des Koordinatensystems):

♦ 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).

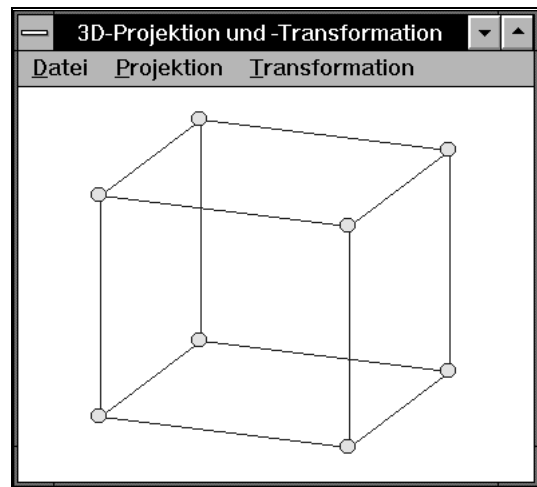
♦ In den folgenden 8 Zeilen stehen die Koordinaten der 8 Knoten, von links nach rechts:  $x$ ,  $y$ ,  $z$ . Durch die Reihenfolge der Zeilen wird eine "Knoten-Numerierung" festgelegt (in diesem Fall gibt es die Knoten **1 ... 8**), auf die sich die nachfolgenden Daten beziehen.

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

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

Die nebenstehende Darstellung zeigt das durch die Datei **d\_wuerfl.dat** beschriebene Drahtmodell, gezeichnet vom Programm **draht1.c**. Die Knoten werden durch "Marker" besonders hervorgehoben (um den Aufruf der Funktion **ptmrk\_gi** zu demonstrieren). Wie beim Programm **projec2.c** (Abschnitt 3.5.3) können beliebige Projektionen eingestellt werden, und über die Tastatur kann man das Objekt bewegen (Rotationen oder Translationen werden zur jeweils aktuellen Transformationsmatrix hinzugefügt, die von den "zeichnenden pt...-Funktionen" ausgewertet werden).



Programm **draht1.c** mit Datei **d\_wuerfl.dat**

Nachfolgend werden nur die Passagen des Programms **draht1.c** beschrieben, in denen bisher nicht vorgestellte GIW-Funktionen auftauchen. Beim Programmstart wird versucht, mit **rddfl\_gi** eine Datei **d\_wuerfl.dat** zu lesen:

```
case WM_CREATE :

    ptini_gi ( ) ;                                /* Initialisieren und */
    prgte_gi (&xwe , &ywe , &zwe) ;                /* Erfragen der      */
    prgtr_gi (&xwr , &ywr , &zwr) ;                /* Projektions-      */
    prgtd_gi (&xwd , &ywd , &zwd) ;                /* Parameter         */

    hPenBlue      = CreatePen (PS_SOLID , 1 , mkrrgb_gi (GI_BLUE)) ;
    hBrushYellow  = CreateSolidBrush (mkrrgb_gi (GI_YELLOW)) ;

    model = rddfl_gi (hwnd , "d_wuerfl.dat" ,
                     &ne , &nk , &ke , &xy_p , &re_p) ;

    /* ... versucht Datei "d_wuerfel.dat" zu lesen, bei Erfolg
       werden mit ptmx3_gi die Extremwerte der "User Coordinates"
       berechnet, die sich unter Anwendung der aktuellen
       Transformation und der aktuellen Projektion auf alle
       Punkte ergeben wurden: */

    if (model) ptmx3_gi (nk , xy_p , &xumin , &xumax ,
                        &yumin , &yumax) ;

    flini_gi (hwnd) ;                             /* ... als Vorbeitung */
                                                /* fuer flodl_gi      */

    return 0 ;
```

- ♦ Wenn das Lesen einer Datei mit **rddfl\_gi** erfolgreich ist, werden (hier als **ne**, **nk** und **ke**) die drei Werte aus der ersten Zeile der Datei abgeliefert und ab Adresse **xy\_p** stehen die Koordinaten der Knoten (dicht gepackt in der Reihenfolge  $x_1, y_1, z_1, x_2, y_2, z_2, \dots$ ). Die Element-Informationen wurden in **ne** Strukturen des Typs

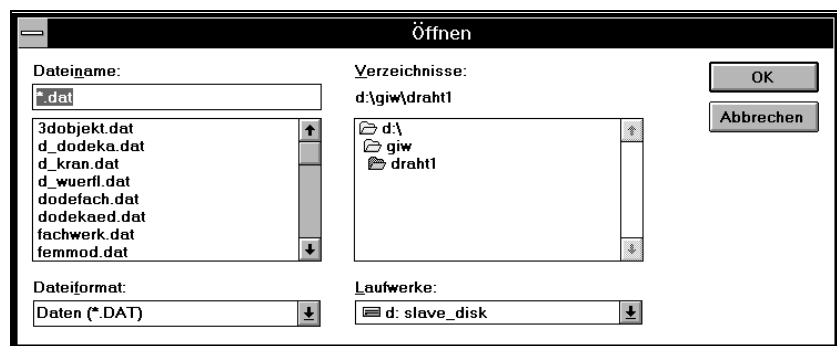
```
typedef struct GI_ELEM_tag
{
    int          nop ;
    int          *param ;
    struct GI_ELEM_tag *next ;
} GI_ELEM ;
```



(Definition dieses Typs in **giw.h**) abgelegt, die zu einer verketteten Liste verbunden sind. Der Pointer auf die erste Struktur wird von **rddfl\_gi** hier als **re\_p** abgeliefert. Die Komponente **nop** ist die Anzahl der Parameter, die das Element beschreiben (für das hier betrachtete Modell sind das die beiden Knotennummern, dort steht zunächst immer der Wert **2**), die Komponente **param** pointert auf ein Feld mit den **nop** Parametern, die Komponente **next** pointert auf die Folgestruktur der Liste.

- ♦ Die Funktion **ptmx3\_gi** kann ein Koordinatenfeld ("World Coordinates") in der beschriebenen Anordnung übernehmen und ermittelt den "Platzbedarf" in der Zeichenebene, wenn alle Punkte sichtbar sein sollen.
- ♦ Die Funktion **flini\_gi** dient nur zur Vorbereitung (Initialisierung) eines eventuellen späteren **flodl\_gi**-Aufrufs, der nachfolgend beschrieben wird.

Über das Menüangebot **Datei** kann eine andere Modell-Beschreibungs-Datei eingelesen werden. Bei der Auswertung der WM\_COMMAND-Botschaft wird zunächst mit der Funktion **flodl\_gi** die Dialog-Box geöffnet, die als Standard-Dialog in Windows für die Eingabe eines Datei-Namens dient. Wenn der Dialog vom Benutzer nicht abgebrochen wird, liefert **flodl\_gi** auf der Position 2 den kompletten Pfadnamen der Datei ab, mit dem dann **rddfl\_gi** aufgerufen wird:



```
case WM_COMMAND:
    switch (wParam)
    {
        case 10:
            if (flodl_gi (hwnd, pathnm, filename))
            {
                /* ... startet den typischen "Windows-Dialog"
                   fuer die Eingabe eines File-Namens */

                /* Wenn vom Programm bereits ein Berechnungsmodell
                   erfolgreich eingelesen worden ist, wurde
                   Speicherplatz fuer die Knoten-Koordinaten und
                   die Elementbeschreibungen dynamisch in rddfl_gi
                   angefordert, der nun freigegeben wird: */

                if (model)
                {
                    free (xy_p);
                    frell_gi (km_p); /* ... loescht die gesamte */
                }                  /* verkettete Liste */

                model = rddfl_gi (hwnd, pathnm,
                                &ne, &nk, &ke, &xy_p, &km_p);

                /* ... liest die Datei "pathnm" (Name wurde vom
                   flodl_gi-Dialog geliefert), legt dabei
```

```

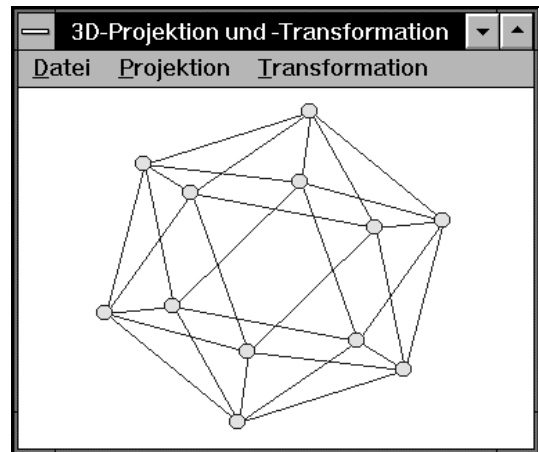
        ausreichenden Speicherplatz fuer die Koordinaten
        ab xy_p und fuer eine verkettete Liste mit den
        Elementbeschreibungen (Root-Pointer re_p) an. */

        if (model) ptmx3_gi (nk , xy_p , &xumin , &xumax ,
                             &yumin , &yumax) ;

        InvalidateRect (hwnd , NULL , TRUE) ;
    }

```

Da das "alte" Modell bei dieser Aktion verschwindet, wird mit **InvalidateRect** eine WM\_PAINT-Botschaft ausgelöst, um das Fenster des Programms zu aktualisieren. Wenn auf diesem Weg z. B. die Datei **d\_ikosa.dat** eingelesen wird (beschreibt das Drahtmodell eines "Ikosaeders", das ist ein von 20 gleichseitigen Dreiecken begrenzter Körper), zeigt sich das nebenstehende Bild.



Programm draht1.c mit Datei d\_ikosa.dat

Bei der Auswertung der Botschaft WM\_PAINT wird nur dann gezeichnet, wenn eine Modell-Beschreibung eingelesen wurde. Die Festlegung der "User Coordinates" mit **stuci\_gi** erfolgt so, daß das Modell "in das Fenster paßt" (nach dem Einlesen, das kann sich durch Transformationen, die über die Tastatur durch Drücken der Tasten **x**, **y**, **z** bzw. **X**, **Y**, **Z** erzeugt werden, natürlich ändern).

Das Zeichnen der geraden Linien erfolgt in einer Schleife über alle Elemente (Abarbeiten der verketteten Liste) mit den bereits behandelten Funktionen **ptmov\_gi** und **ptlin\_gi**. Für das Zeichnen der Knoten (ebenfalls in einer Schleife) wird die Funktion **ptmrk\_gi** verwendet, die wie die im Abschnitt 2.2.5 beschriebene Funktion **umark\_gi** ("Marker" zeichnen) arbeitet, allerdings wird **ptmrk\_gi** ein 3D-Punkt übergeben ("World Coordinates"), der vor der Zeichenaktion der aktuellen Transformation unterworfen und dann auf die Zeichenebene projiziert wird:

```

case WM_PAINT :

    hdc = gstrt_gi (hwnd , cxClient , cyClient) ;

    if (model)
    {
        stuci_gi (xumin , yumin , xumax , yumax , 10.) ;

        /* ... definiert "User Coordinates" nach dem mit
           ptmx3_gi ermittelten Bedarf, sieht "10% Rand" vor */

        SelectObject (hdc , hPenBlue) ;
        SelectObject (hdc , hBrushYellow) ;

        elem_p = re_p ;

        while (elem_p) /* ... alle Elemente */
        {
            k1 = *(elem_p->param) - 1 ; /* ... zugehoerige */
            k2 = *(elem_p->param + 1) - 1 ; /* Knotennummern */

            coord_p = xy_p + k1*3 ; /* Pointer auf x-Koordinate */

```

```

    ptmov_gi (hdc , *coord_p , *(coord_p+1) , *(coord_p+2)) ;

    coord_p = xy_p + k2 * 3 ;
    ptlin_gi (hdc , *coord_p , *(coord_p+1) , *(coord_p+2)) ;

    elem_p = elem_p->next ;
}

for (i = 0 ; i < nk ; i++)          /* ... alle Knoten */
{
    coord_p = xy_p + i * 3 ;

    ptmrk_gi (hdc , GI_MKFCIRCLE , 1. ,
               *coord_p , *(coord_p+1) , *(coord_p+2) , 0) ;

    /* ... zeichnet Marker (gefüllten Kreis) in
       Standard-Groesse */
}

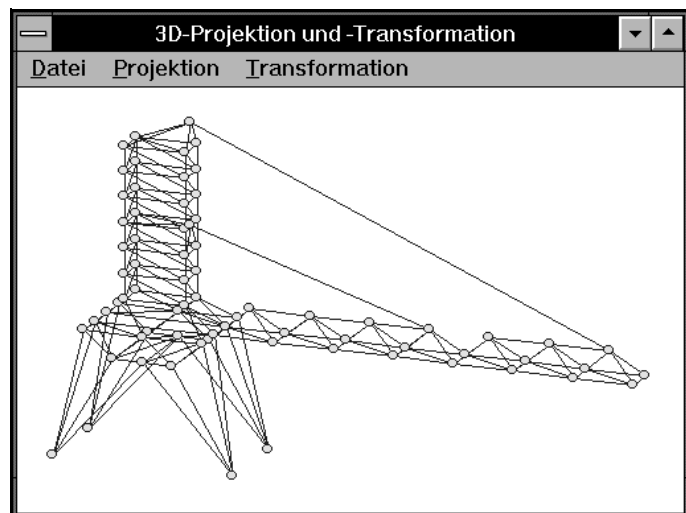
gstop_gi (hwnd) ;
return 0 ;

```

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 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.

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 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.c 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).

### 3.7 "Breite zweifarbige Linien"

Für die Visualisierung der Information, welche von zwei sich kreuzenden Geraden vor der anderen liegt, kann man "breite zweifarbige Linien" verwendet, die mit den GIW-Funktionen **uwidl\_gi** bzw. **ptwidl\_gi** gezeichnet werden können. Eine Linie besteht dabei gewissermaßen aus "drei nebeneinanderliegenden Linien", von denen die mittlere eine andere Farbe hat.

Nachfolgend wird nur die Funktion **ptwidl\_gi** beschrieben, die übrigens nach der Transformation der Koordinaten und Projektion auf die Zeichenebene selbst **uwidl\_gi** aufruft. Es müssen die Koordinaten für zwei 3D-Punkte ("World Coordinates") übergeben werden (**ptwidl\_gi** erledigt sowohl "Movo to" als auch "Line to"), außerdem die Gesamtbreite der Linie (Pixel), die Breite der "mittleren Linie" (Pixel) und die Farbwerte (COLORREF) für den Randbereich und den mittleren Bereich.

Im Programm **escher.c** ist gegenüber dem Programm **draht1.c** (Abschnitt 3.6) nur bei der Bearbeitung der WM\_PAINT-Botschaft die Schleife geändert worden, mit der die Linien (Elemente) des Objekts gezeichnet werden. An die Stelle von **ptmov\_gi** und **ptlin\_gi** ist der Aufruf von **ptwidl\_gi** getreten. Es wird eine 5 Pixel breite schwarze Linie gezeichnet, die von einer 1 Pixel breiten gelben Linie überlagert wird (da **ptwidl\_gi** die vorher eingestellten GDI-Objekte nach dem Zeichnen wieder in den Device Context einsetzt, wird der vor dem **ptwidl\_gi**-Aufruf ausgewählte Zeichenstift für die nach dem **ptwidl\_gi** noch zu zeichnenden "Marker" wie im Programm **stab1.c** verwendet):

```
elem_p = re_p ;

while (elem_p)
{
    k1 = *(elem_p->param)      - 1 ;          /* ... alle Elemente */
    k2 = *(elem_p->param + 1) - 1 ;          /* ... zugehoerige */
                                           /* Knotennummern */

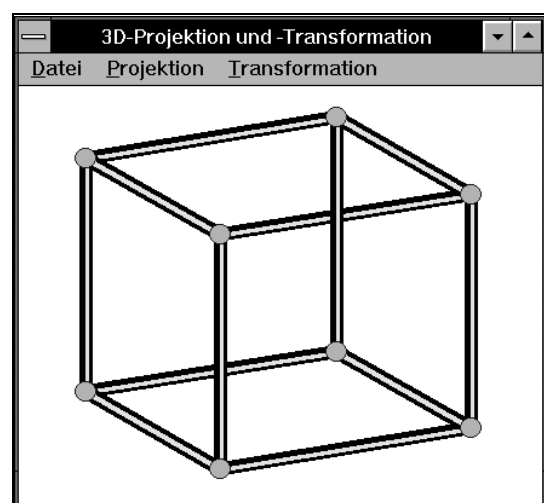
    coord1_p = xy_p + k1 * 3 ;
    coord2_p = xy_p + k2 * 3 ;

    ptwidl_gi (hdc , *coord1_p , *(coord1_p+1) , *(coord1_p+2) ,
               *coord2_p , *(coord2_p+1) , *(coord2_p+2) ,
               5 , mkrb_gi (GI_BLACK) , 1 , mkrb_gi (GI_YELLOW)) ;

    elem_p = elem_p->next ;
}
```

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).

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 be-



Typisches "Escher-Bild"

schreiben, 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 3.9 werden im folgenden Abschnitt einige allgemeine Bemerkungen zum Problem der verdeckten Kanten und Flächen vorangestellt.

### 3.8 "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 GIW-Funktionen 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). Um diesem Mangel abzuhelpen, 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.

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 3.10 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 Antwortverhalten 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 3.6 und 3.7) 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 3.7 wurde der dafür in der GIW-Toolbox vorgesehene spezielle Linientyp vorgestellt, im folgenden Abschnitt wird die Lösung des Überdeckungsproblems bei Stabmodellen mit dem "geordneten Zeichnen breiter zweifarbiger Linien" demonstriert.

### 3.9 Darstellung von "Stabmodellen"

Zur Realisierung des "Zeichnens der Objekte in der Reihenfolge ihres Abstands vom Betrachter" ist das Anlegen einer sortierten Liste der Objekte erforderlich. Dies wird durch mehrere GIW-Funktionen unterstützt, wobei dem Programmierer zwei verschiedene Strategien angeboten werden:

- ♦ **Die Objekte werden in einer "doppelt verketteten linearen Liste" angeordnet** (vgl. z. B. "J. Dankert: C und C++ für UNIX, DOS und MS-Windows 3.1/95/NT", Kapitel 7). Dies wird von der Funktion **insol\_gi** realisiert, die jeweils ein Objekt in die Liste einbringt. Der Vorteil dieser Variante ist der recht einfache Umgang mit einer solchen Liste. Diese Strategie wird in diesem Abschnitt im Programm **stab1.c** verwendet.
- ♦ **Die Objekte werden in einem "binären Baum" angeordnet** (vgl. z. B. "J. Dankert: C und C++ für UNIX, DOS und MS-Windows 3.1/95/NT", Kapitel 8). Dies wird von der Funktion **insot\_gi** realisiert, die jeweils ein Objekt in den Baum einfügt. Der Vorteil dieser Variante ist ein wesentlich schnellerer Aufbau einer solchen Anordnung, bei einer großen Anzahl zu zeichnender Elemente ist diese Strategie unbedingt zu bevorzugen. Die Abarbeitung einer Baumstruktur ist aber wohl nur mit "rekursiver Programmieretechnik" vernünftig realisierbar, dies wird im Abschnitt 3.10 im Programm **polyarea.c** demonstriert.

Die Objekte, die mit **insol\_gi** in die verkettete Liste bzw. mit **insot\_gi** in den binären Baum eingefügt werden, müssen vom Typ **GI\_OBJ** sein. Der Typ dieser Struktur wird in **giw.h** definiert:

```
typedef struct GI_OBJ_tag
{
    int          type      ;
    void         *data     ;
    double       dist      ;
    struct GI_OBJ_tag *left ;
    struct GI_OBJ_tag *right;
} GI_OBJ ;
```

Mit der Komponente **type** kann der Programmierer (willkürlich) den Typ des zu zeichnenden Objekts festlegen. Im Programm **stab1.c** werden die Typen **1** (für die Stäbe) und **2** (für die Knoten) verwendet. Der **void**-Pointer **data** zeigt auf die eigentliche Beschreibung dieses Objekts, im Programm **stab1.c** werden dort entweder ein Pointer auf eine **GI\_ELEM**-Struktur (für die Stäbe) oder ein **double**-Pointer auf die Koordinaten des Knotens eingetragen. Die Komponente **dist** ist die "Entfernung des Objekts vom Betrachter" und damit das Sortierkriterium. Die beiden Pointer **left** und **right** zeigen auf "Vorgänger" oder "Nachfolger" in der verketteten Liste bzw. linken und rechten Nachfolger im binären Baum, in jedem Fall zeigt **left** auf ein Objekt, das weiter vom Betrachter entfernt ist (oder ist der NULL-Pointer), **right** zeigt auf ein Objekt mit kürzerer Entfernung (oder ist der NULL-Pointer).

Ein Maß für die Entfernung des Objekts vom Betrachter wird mit der Funktion **ptdis\_gi** ermittelt, der ein geeigneter Punkt des Objekts ("World Coordinates") zu übergeben ist. Dieser Punkt wird der aktuellen Transformation unterworfen. Wenn "Zentralprojektion" eingestellt ist, wird das (immer positive) Quadrat der Distanz des Objekt-Punktes vom "Eye

Point" abgeliefert (um das Berechnen der Quadratwurzel zu vermeiden). Bei "Parallelprojektion" befindet sich der Betrachter theoretisch "im Unendlichen", als "Standpunkt des Betrachters" könnte irgendein Punkt auf der Projektionsebenen-Normalen gewählt werden, die durch den Objekt-Punkt geht. Dann aber ist auch der (vorzeichenbehaftete) Abstand des Objekt-Punktes von der Projektionsebene geeignet. Dieser wird (bis auf einen für alle berechneten Abstände gleichen Faktor) als Ergebnis von **ptdis\_gi** geliefert.

Der Programmierer braucht allerdings **ptdis\_gi** nicht selbst aufzurufen, die Funktionen **insol\_gi** bzw. **insot\_gi** erledigen das. Im aufrufenden Programm ist nur der Pointer auf das "Anchor-Element" ("Root-Element") der verketteten Liste bzw. des binären Baums zu verwalten. Im Programm **stab1.c** wird folgende Strategie realisiert:

- ♦ Ein Schalter **proj\_chngd** ist der Indikator dafür, ob sich Projektion oder Transformation geändert haben (wird mit **1** initialisiert und bei jeder Änderung wieder auf **1** gesetzt).
- ♦ Bei der Bearbeitung der Botschaft WM\_PAINT wird immer dann, wenn **proj\_chngd** den Wert **1** hat, eine geordnete verkettete Liste aller zu zeichnenden Objekte (Elemente und Knoten) erzeugt. Anschließend wird die Liste abgearbeitet, indem alle Objekte in der Reihenfolge gezeichnet werden, wie sie in der Liste verzeichnet sind.

Hinweis: 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". Hier soll jedoch nur das Einbringen von unterschiedlichen Objekt-Typen in die Liste demonstriert werden.

Der Aufbau der Liste ist in **stab1.c** folgendermaßen programmiert:

```
if (proj_chngd)
{
    desot_gi (robj_p) ;                /* ... loescht eine bereits */
    robj_p = NULL ;                   /* existierende Liste      */

    elem_p = re_p ;

    while (elem_p)                    /* ... ueber alle Elemente */
    {
        k1 = *(elem_p->param) - 1 ;
        k2 = *(elem_p->param + 1) - 1 ;

        coord1_p = xy_p + k1*3 ;      /* Pointer auf x-Koordinate */
        coord2_p = xy_p + k2*3 ;      /* Pointer auf x-Koordinate */

        insol_gi (&robj_p , 1 , elem_p ,
                  (*(coord1_p) + *(coord2_p)) / 2 ,
                  (*(coord1_p+1) + *(coord2_p+1)) / 2 ,
                  (*(coord1_p+2) + *(coord2_p+2)) / 2) ;

        /* ... fuegt ein Element (Objekt-Typ 1, Typ wird
           willkuerlich festgelegt) in die verkettete Liste ein,
           elem_p pointert auf die Beschreibung in der GI_ELEM-
           Struktur, es folgen die Koordinaten des Element-
           Mittelpunkts, mit dem in insol_gi die "Distanz vom
           Beobachter" berechnet und als Sortier-Kriterium
           verwendet wird. Auf robj_p wird der Pointer auf das
           "Anchor-Element" der Liste abgeliefert. */

        elem_p = elem_p->next ;
    }
}
```



```

for (i = 0 ; i < nk ; i++)      /* ... ueber alle Knoten */
{
    coord1_p = xy_p + i * 3 ;

    insol_gi (&robj_p , 2 , coord1_p ,
              *(coord1_p) , *(coord1_p+1) , *(coord1_p+2)) ;

    /* ... fuegt einen Knoten (Objekt-Typ 2) in die
       verkettete Liste ein, coord1_p pointer auf die
       Koordinaten, die dieses Objekt beschreiben, diese
       werden auch als "Objekt-Punkt" (fuer das Sortier-
       kriterium) uebergeben. */
}
proj_chngd = 0 ;
}

```

- ♦ Man beachte, daß als erstes Argument an **insol\_gi** die Adresse des Pointers **robj\_p** ("Pointer auf den Pointer") übergeben werden muß, weil sich das "Anchor-Element" bei jedem **insol\_gi**-Aufruf ändern kann (neues Objekt kann "Spitzenposition" übernehmen, beim Einfügen des ersten Objekts in jedem Fall).

Bei der anschließenden Abarbeitung der Liste wird über die **type**-Komponente entschieden, was für ein Objekt gezeichnet werden soll:

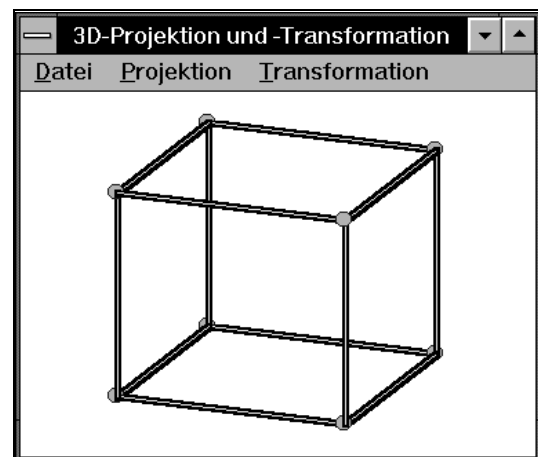
```

gi_obj_p = robj_p ;                /* In der folgenden Schleife */
while (gi_obj_p)                   /* wird die gesamte verkettete */
{                                   /* Liste abgearbeitet: */
    if (gi_obj_p->type == 1)        /* ... ist es ein Element */
    {
        elem_p = (GI_ELEM *) gi_obj_p->data ;
        k1      = *(elem_p->param) - 1 ;
        k2      = *(elem_p->param + 1) - 1 ;
        coord1_p = xy_p + k1 * 3 ;
        coord2_p = xy_p + k2 * 3 ;
        ptwdl_gi (hdc , *coord1_p , *(coord1_p+1) , *(coord1_p+2) ,
                  *coord2_p , *(coord2_p+1) , *(coord2_p+2) ,
                  5 , mkr gb_gi (GI_BLACK) , 1 , mkr gb_gi (GI_YELLOW)) ;
    }
    else if (gi_obj_p->type == 2)   /* ... ist es ein Knoten */
    {
        coord1_p = (double *) gi_obj_p->data ;
        ptmrk_gi (hdc , GI_MKFCIRCLE , 1. ,
                  *coord1_p , *(coord1_p+1) , *(coord1_p+2) , 0) ;
    }
    gi_obj_p = gi_obj_p->right ;
}

```

- ♦ Der Zugriff auf die beschreibenden Daten der Objekte erfolgt über den **void**-Pointer **data**, der auf den passenden Typ "gecastet" werden muß.

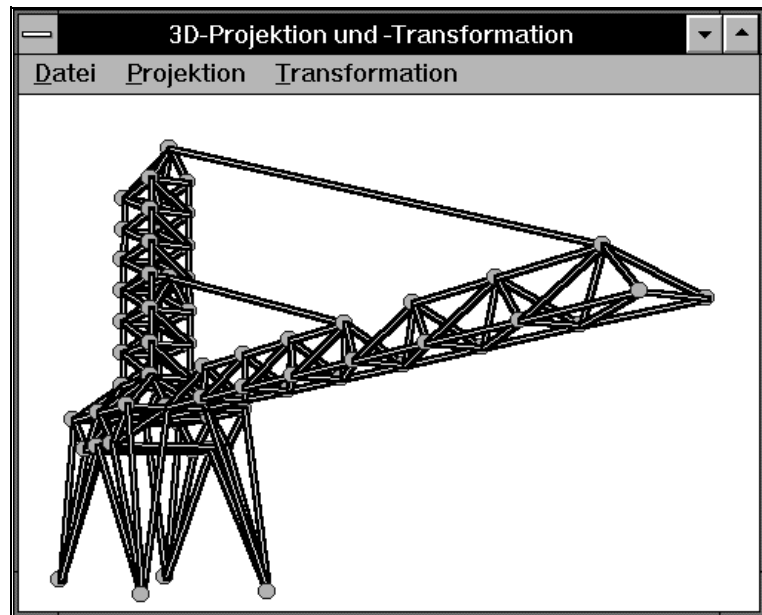
Bis auf das bereits genannte Problem der "Durchdringung" der zu zeichnenden Objekte zeigt sich der aus Stäben gebildete Würfel nach dem Programmstart (und nach jeder Änderung von Transformation und Projektion) in korrekter "Tiefen-Anordnung" der Stäbe (nebenstehende Abbildung).



Programm stab1.c mit Datei d\_wuerfl.dat

Das mit der Datei **d\_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.

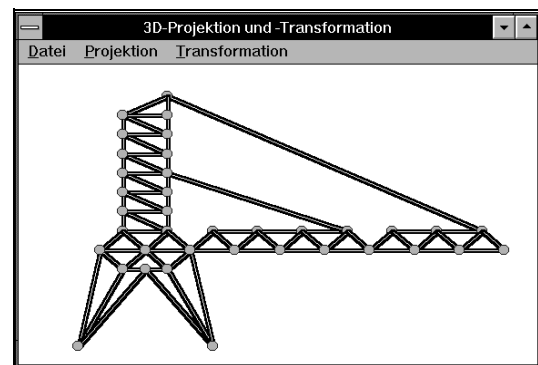
Um allerdings Details erkennen zu können, müßte man (neben der bereits gegebenen Möglichkeit unterschiedlicher Standpunkte, man kann unter den Kran und auch in ihn "hineinkriechen") noch die Möglichkeit des "Zooms" ergänzen. Das allerdings ist mit den im Abschnitt 2.3 vorgestellten Funktionen problemlos realisierbar, weil die Darstellung letztendlich (nach 3D-Transformation und Projektion) mit "User Coordinates" gezeichnet wird, für die die Zoom-Funktionen ausgelegt sind.



Datei **d\_kran.dat**, Zentralprojektion, "Eye Point" bei (25;-10;-4)

Für "technische Ansichten" wie die nebenstehend zu sehende "Seitenansicht" 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.

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



Datei **d\_kran.dat**, Seitenansicht  
(Parallelprojektion)

Abschließend soll noch auf die "Aufräum-Funktionen" im Programm **stab1.c** aufmerksam gemacht werden. Die Liste der Element-Beschreibungen, die beim Einlesen der Daten von **rddfl\_gi** angelegt wird, wurde bereits im Programm **draht1.c** mit der Funktion **frell\_gi** gelöscht. Für das Löschen der Liste, die von **insol\_gi** angelegt wird, ist die Funktion **desot\_gi** zuständig. Beiden Funktionen ist jeweils der Pointer auf das "Anchor-Element" der Liste zu übergeben.

Für die Freigabe des (ebenfalls von **rddfl\_gi** angelegten) dynamischen Speicherbereichs für die Koordinaten kann die C-Funktion **free** (in **stab1.c** mit dem Pointer auf das Koordinatenfeld **xy\_p**) aufgerufen werden.

### 3.10 3D-Polygonflächen

Für das Zeichnen von "gefüllten 3D-Polygonflächen" ist die Funktion **ptfpl\_gi.c** verfügbar. Sie erwartet als Argumente (neben einem Handle auf den Device Context) die Anzahl der Punkte *n* und drei Felder mit den *x*-, *y*- und *z*-Koordinaten der Punkte ("World Coordinates"). Die Punkte werden 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 Funktion **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 z. B. die Seitenflächen eines Quaders) tatsächlich Ebenen sind oder aber tolerierbar im Rahmen der angestrebten Abbildung von Ebenen abweichen.

Das Programm **polyarea.c** dient zur Demonstration der Möglichkeiten, die von folgenden GIW-Funktionen angeboten werden:

- ♦ Mit der bereits in den Programmen **draht1.c** und **stab1.c** verwendeten Funktion **rddfl\_gi** werden Modelle mit Elementen eingelesen, die nicht alle durch die gleiche Anzahl von Parametern beschrieben werden (Polygone können unterschiedliche Anzahl von Punkten haben). Dafür muß der dritte Wert in der ersten Zeile der Datei, der ansonsten die Anzahl der Parameter pro Element angibt, den Wert **0** (oder einen negativen Wert) haben. Den Elementbeschreibungen wird in jeder Zeile dann eine Zahl vorangestellt, die angibt, wieviel Parameter für dieses Element folgen.  
  
Neben den Knotennummern wird als zusätzlicher Element-Parameter ein Wert für die Farbe gespeichert (und von **rddfl\_gi** gelesen und in der GI\_ELEM-Struktur abgelegt). Als Farben-Indikatoren werden die ganzzahligen Werte **0...7** verwendet, die mit der Funktion **mkrgb\_gi** einen COLORREF-Wert für eine Grundfarbe erzeugen.
- ♦ Mit **insot\_gi** werden alle zu zeichnenden Objekte (3D-Polygonflächen) in einem "binären Baum" gespeichert. Diese Aktion ist auf exakt die gleiche Weise zu programmieren wie das Anlegen der verketteten Liste, das im Programm **stab1.c** (Abschnitt 3.9) vorgestellt wurde. Als Koordinaten des Bezugspunktes, mit dem der Abstand des Objekts vom Betrachter berechnet werden soll, werden die Mittelwerte der Koordinaten der Polygonpunkte verwendet.
- ♦ Für das Zeichnen der im binären Baum verankerten Objekte wird die rekursiv arbeitende Funktion **DrawRecursive** geschrieben.
- ♦ Das Löschen des gesamten binären Baums wird (wie das Löschen der verketteten Liste, vgl. Programm **stab1.c** im Abschnitt 3.9) von der Funktion **desot\_gi** erledigt. Diese Funktion überprüft selbst, ob eine verkettete Liste oder ein binärer Baum vorliegt, und startet den passenden Algorithmus.

Wie im Programm **stab1.c** wird ein Indikator **proj\_chngd** benutzt, der anzeigt, ob sich Transformation oder Projektion geändert haben. Wenn dies der Fall ist, wird bei der Bearbeitung der Botschaft WM\_PAINT der binäre Baum mit den zu zeichnenden Objekten neu angelegt:

```

if (proj_chngd)
{
    desot_gi (robj_p) ;
    robj_p = NULL ;

    elem_p = re_p ;

    while (elem_p)                                /* ... ueber alle Elemente */
    {
        xc = 0. ;
        yc = 0. ;
        zc = 0. ;
        n = elem_p->nop - 1 ;                      /* Anzahl der Elementknoten */

        for (i = 0 ; i < n ; i++)
        {
            k = *(elem_p->param + i) - 1 ;
            coord_p = xy_p + k * 3 ;

            xc += *(coord_p) ;
            yc += *(coord_p + 1) ;
            zc += *(coord_p + 2) ;
        }

        xc = xc / n ;
        yc = yc / n ;
        zc = zc / n ;                             /* Mittelwerte der Koordinaten */
                                                /* der Polygon-Punkte */

        insot_gi (&robj_p , 1 , elem_p , xc , yc , zc) ;

        /* ... fuegt ein Objekt sortiert in den binaeren Baum ein. */

        elem_p = elem_p->next ;
    }

    proj_chngd = 0 ;
}

DrawRecursive (robj_p) ;

```

Gezeichnet werden die Objekte in der Funktion **DrawRecursive**. Die Strategie der rekursiven Abarbeitung des binären Baums ist relativ einfach:

- ◆ Aufgerufen wird die Funktion mit dem Pointer auf das "Anchor-Element" **robj\_p** des binären Baums (siehe oben).
- ◆ Die erste Aktion in **DrawRecursive** ist der rekursive (Selbst-)Aufruf mit dem Pointer auf den linken Nachfolger (falls existent), weil linke Nachfolger weiter entfernte Objekte beschreiben. Dies kann unter Umständen ein sehr tiefer "rekursiver Abstieg" werden.
- ◆ Danach erst wird die eigentliche Zeichenaktion für das Objekt ausgeführt, mit dem **DrawRecursive** aufgerufen wurde, um im Anschluß daran wieder mit rekursivem Aufruf den Abstieg zu den rechten Nachfolgern zu starten.
- ◆ Um die für das Zeichnen benötigten Parameter nicht von Aufruf zu Aufruf "durchreichen" zu müssen, wurden **hdc** (Handle auf Device Context), die "Brushes" mit den Füllfarben **hBrush[8]** und der Pointer auf die Koordinaten **xy\_p** global vereinbart.
- ◆ Auch die drei in **DrawRecursive** benötigten Arbeitsfelder sind global für alle rekursiv erzeugten Instanzen verfügbar.

```

/* Arbeitsfelder werden global vereinbart (moeglich, weil Fuellen der Felder
   und Uebergeben an ptfpl_gi ohne Unterbrechung durch einen rekursiven Aufruf
   der Funktion erfolgt), um bei tiefer Rekursion keinen "Stack Overflow" zu
   riskieren. Als "Unikate" koennen sie grosszuegig angelegt werden: */

double  xpoly[40] , ypoly[40] , zpoly[40] ;

void DrawRecursive (GI_OBJ *gi_obj_p)
{
    int      k , n , i ;
    GI_ELEM  *elem_p ;
    double   *coord_p ;

    if (gi_obj_p->left) DrawRecursive (gi_obj_p->left) ;

        /* ... zeichnet zuerst alle weiter entfernten Objekte, ... */

    elem_p = (GI_ELEM *) gi_obj_p->data ;
    n = elem_p->nop - 1 ;

    for (i = 0 ; i < n ; i++)
    {
        k = *(elem_p->param + i) - 1 ;
        coord_p = xy_p + k * 3 ;

        xpoly[i] = *(coord_p) ;
        ypoly[i] = *(coord_p + 1) ;
        zpoly[i] = *(coord_p + 2) ;
    }

        /* ... danach mit einem */
        /* "Brush", der waehrend */
        /* der WM_CREATE-Aktion */
        /* erzeugt wird (und */
        /* global vereinbart */
        /* ist), ... */

    SelectObject (hdc , hBrush[*(elem_p->param + n)]) ;

    ptfpl_gi (hdc , n , xpoly , ypoly , zpoly) ;

        /* ... das aktuelle Objekt, bevor (wieder rekursiv) alle
           naeherliegenden Objekte gezeichnet werden: */

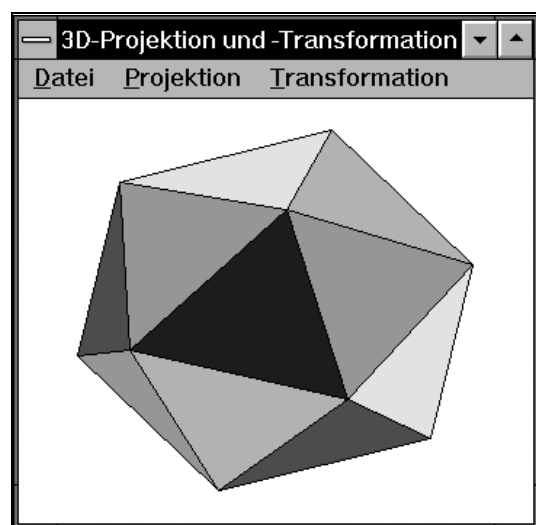
    if (gi_obj_p->right) DrawRecursive (gi_obj_p->right) ;
}

```

Die nebenstehende Abbildung zeigt das Bild, das vom Programm **polyarea.c** mit der Datei **p\_ikosa.dat** erzeugt wird, ein Ikosaeder ("platonischer Körper", begrenzt von 20 gleichseitigen Dreiecken, wurde bereits im Abschnitt 3.6 als "Drahtmodell" gezeigt), von dem alle 20 Flächen gezeichnet wurden, allerdings in der "richtigen Reihenfolge", so daß nur die 10 sichtbaren auch tatsächlich zu sehen sind.

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.

Die Strategie, mit "gefüllten Flächen" die unsichtbaren Flächen und Kanten zu überzeichnen, kann auch verwendet werden, um ein reines "Hidden



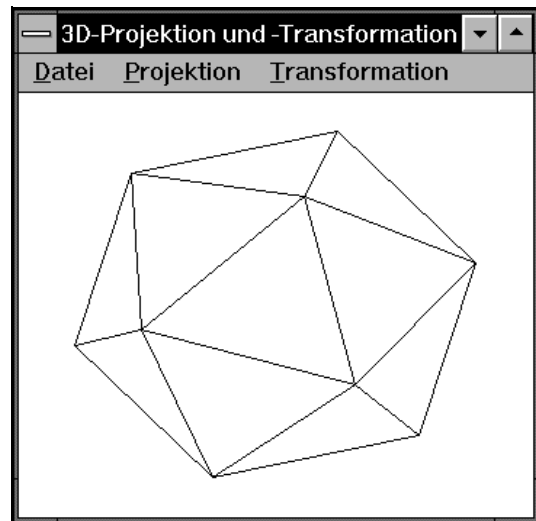
Programm **polyarea.c**: Ikosaeder (Datei **p\_ikosa.dat**) als Flächenmodell

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.

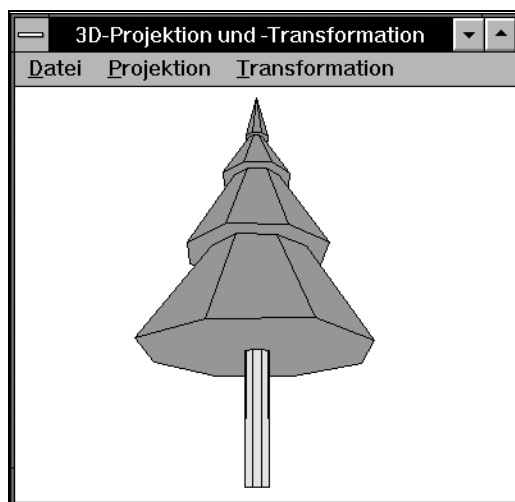
Der einfache Algorithmus des Programms **polyarea.c**, der die Flächen nach dem Abstand des durch die Mittelwerte der Eckpunkte festgelegten Punktes vom Betrachter sortiert, arbeitet für einen einfachen Körper wie ein Ikosaeder fehlerfrei. Bei auch nur etwas komplizierteren Flächen kann man sich nicht 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 **stab1.c** (Abschnitt 3.9) dargestellt werden.

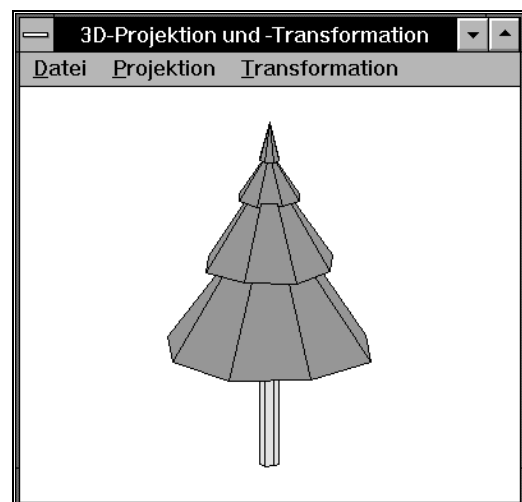
Die beiden Abbildungen unten sollen das demonstrieren. Das dargestellte Objekt (Datei **p\_baum.dat**) wird durch insgesamt 73 Dreiecke, Vierecke und Achtecke begrenzt. In fast allen Ansichten bei beliebigen Transformationen ist die Darstellung korrekt (z. B. wie im linken Bild). Für ganz wenige Transformationen 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. Die Frage, wie fein die Unterteilung der Flächen sein muß, kann nicht allgemein beantwortet werden, hängt schließlich auch davon ab, ob man eher einen Fehler tolerieren oder eher größere Antwortzeiten akzeptieren kann.



"Hidden Line"-Bild des Ikosaeders  
(Programm **polyarea.c**, Datei **p\_ikosal.dat**)



Korrekte Darstellung



Ansicht mit kleinem Fehler

## 4 Header-Datei und GIW-Funktionen

Die in alle Programme, die GIW-Funktionen benutzen, einzubindende Header-Datei wird im Abschnitt 4.2 komplett gelistet, von den GIW-Funktionen werden (**alphabetisch geordnet**) im Abschnitt 4.3 jeweils der Anfangskommentar (mit der Beschreibung der Parameter) und die Kopfzeile angegeben. Im Abschnitt 4.1 sind die Funktionen nach ihrer Funktionalität zusammengefaßt.

### 4.1 GIW-Funktionen

Die im Abschnitt 4.3 näher beschriebenen Funktionen können in folgende Gruppen eingeordnet werden:

#### Start/Stopp:

<b>gstrt_gi</b>	...	initialisiert alle GIW-Parameter, besorgt "Handle of Device Context",
<b>gstop_gi</b>	...	ruft <b>EndPaint</b> , "räumt im GIW auf".

#### Viewport und Koordinaten definieren:

<b>stcvp_gi</b>	...	definiert "Current Viewport" mit Geräte-Koordinatensystem und "Clipping"-Bereich,
<b>stuca_gi</b>	...	definiert anisotrope Koordinaten mit <b>double</b> -Werten für Viewport,
<b>stuci_gi</b>	...	definiert isotrope Koordinaten mit <b>double</b> -Werten für Viewport.

#### Graphik-Ausgabe mit Geräte-Koordinaten ("Viewport Coordinates"):

<b>vmove_gi</b>	...	bewegt den imaginären Zeichenstift ohne Zeichenaktion,
<b>vline_gi</b>	...	zeichnet eine Gerade von der aktuellen Position bis zu einer vorzuziehenden Position,
<b>vrect_gi</b>	...	zeichnet ein Rechteck,
<b>vfrec_gi</b>	...	zeichnet ein gefülltes Rechteck,
<b>vfram_gi</b>	...	zeichnet Rahmen um den Viewport.

#### Graphik-Ausgabe mit double-Koordinaten ("User Coordinates"):

<b>umove_gi</b>	...	bewegt den imaginären Zeichenstift ohne Zeichenaktion,
<b>uline_gi</b>	...	zeichnet eine Gerade von der aktuellen Position bis zu einer vorzuziehenden Position,
<b>udarc_gi</b>	...	zeichnet einen Kreisbogen,
<b>ufrec_gi</b>	...	zeichnet ein gefülltes Rechteck,
<b>ufcir_gi</b>	...	zeichnet einen gefüllten Kreis,
<b>ufell_gi</b>	...	zeichnet eine gefüllte Ellipse,
<b>ufel2_gi</b>	...	zeichnet eine gefüllte Ellipse,
<b>ufpol_gi</b>	...	zeichnet ein gefülltes Polygon,
<b>ufsec_gi</b>	...	zeichnet einen gefüllten Kreissektor,

<b>ufpie_gi</b>	...	zeichnet einen gefüllten elliptischen Sektor,
<b>ufpi2_gi</b>	...	zeichnet einen gefüllten elliptischen Sektor,
<b>umark_gi</b>	...	zeichnet einen "Marker".

### **"Vorbereitende t...-Funktionen" (Ändern der ebenen Transformation):**

<b>tinit_gi</b>	...	initialisiert alle Transformationsparameter,
<b>ttran_gi</b>	...	setzt zusätzliche Translation,
<b>trota_gi</b>	...	stellt zusätzliche Rotation um einen vorzugebenden Punkt ein,
<b>tmirx_gi</b>	...	stellt zusätzliche Spiegelung an der $x$ -Achse ein,
<b>tmiry_gi</b>	...	stellt zusätzliche Spiegelung an der $y$ -Achse ein,
<b>ttabs_gi</b>	...	setzt gleichzeitig Translation, Rotation und Skalierung,
<b>tmamu_gi</b>	...	multipliziert ebene Transformationsmatrix mit 3*3-Matrix (von links),
<b>tgttm_gi</b>	...	liefert die gültige ebene Transformationsmatrix ab,
<b>tsttm_gi</b>	...	setzt vorzugebende Matrix als ebene Transformationsmatrix,
<b>tru2t_gi</b>	...	transformiert Punktkoordinaten mit aktueller ebener Transformation.

### **"Zeichnende t...-Funktionen" (werten ebene Transformation aus):**

<b>tmove_gi</b>	...	bewegt den imaginären Zeichenstift (ohne zu zeichnen),
<b>tline_gi</b>	...	zeichnet eine gerade Linie,
<b>tfcir_gi</b>	...	zeichnet einen gefüllten Kreis,
<b>tdarc_gi</b>	...	zeichnet einen Kreisbogen.

### **"Vorbereitende pr...-Funktionen" (Ändern der Projektion):**

<b>prini_gi</b>	...	initialisiert alle Projektionsparameter,
<b>projn_gi</b>	...	stellt Zentralprojektion oder Parallelprojektion ein,
<b>prstc_gi</b>	...	definiert neue Zentralprojektion,
<b>prstp_gi</b>	...	definiert neue Parallelprojektion,
<b>prstd_gi</b>	...	definiert neuen "Blickrichtungs"-Vektor,
<b>prste_gi</b>	...	definiert neuen "Eye Point",
<b>prstr_gi</b>	...	definiert neuen Referenzpunkt der Projektionsebene,
<b>prw2u_gi</b>	...	projiziert 3D-Punkt auf die Zeichenebene.

### **"Zeichnende pr...-Funktionen" (projizieren "World Coordinates" auf Zeichenebene):**

<b>prmov_gi</b>	...	bewegt den imaginären Zeichenstift (ohne zu zeichnen),
<b>prlin_gi</b>	...	zeichnet eine gerade Linie,
<b>prfpl_gi</b>	...	zeichnet ein gefülltes Polygon.

### **"Fragende pr...-Funktionen" (liefern Informationen über Projektionen):**

<b>prgtp_gi</b>	...	liefert aktuelle Projektions-Einstellung (Parallelprojektion oder Zentralprojektion),
<b>prgtd_gi</b>	...	liefert "Blickrichtungs"-Vektor der Parallelprojektion,
<b>prgte_gi</b>	...	liefert "Eye Point" der Zentralprojektion,
<b>prgtr_gi</b>	...	liefert Referenzpunkt der Zeichenebene.



## **"Vorbereitende pt...- und t3...-Funktionen" (Initialisieren und Ändern der 3D-Transformation und der Projektion):**

<b>ptini_gi</b>	...	initialisiert alle Transformations- und Projektionsparameter,
<b>t3rot_gi</b>	...	stellt zusätzliche Rotation um eine Koordinatenachse ein,
<b>t3trn_gi</b>	...	setzt zusätzliche Translation,
<b>t3mam_gi</b>	...	multipliziert 3D-Transformationsmatrix mit 4*4-Matrix (von links),
<b>t3gtm_gi</b>	...	liefert die gültige 3D-Transformationsmatrix ab,
<b>t3stm_gi</b>	...	setzt vorzugebende Matrix als 3D-Transformationsmatrix,
<b>t3w2w_gi</b>	...	transformiert 3D-Punkt mit aktueller Transformation,
<b>ptw2u_gi</b>	...	projiziert 3D-Punkt auf die Zeichenebene.

## **"Zeichnende pt...-Funktionen" (werten 3D-Transformation aus und projizieren "World Coordinates" auf Zeichenebene):**

<b>ptmov_gi</b>	...	bewegt den imaginären Zeichenstift (ohne zu zeichnen),
<b>ptlin_gi</b>	...	zeichnet eine gerade Linie,
<b>ptfpl_gi</b>	...	zeichnet ein gefülltes Polygon,
<b>ptedl_gi</b>	...	zeichnet "breite zweifarbige Linie",
<b>ptmrk_gi</b>	...	zeichnet einen "Marker".

## **"Fragende pt...-Funktionen":**

<b>ptdis_gi</b>	...	liefert ein Maß für den Abstand eines Punktes vom Betrachter,
<b>ptmx3_gi</b>	...	ermittelt Platzbedarf in der Zeichenebene.

## **"Spezielle Cursor", Punkte und Rechteckbereiche picken:**

<b>scapp_gi</b>	...	läßt speziellen GIW-Cursor erscheinen,
<b>scdap_gi</b>	...	läßt speziellen GIW-Cursor verschwinden,
<b>scupd_gi</b>	...	aktualisiert die Position des speziellen GIW_Cursors,
<b>upick_gi</b>	...	liefert Mausposition in "User Coordinates",
<b>rpick_gi</b>	...	liefert zwei Punkte, die einen Rechteckbereich definieren (verändert nach Eingabe des ersten Punktes Cursorform auf "Rechteck").

## **Hilfsfunktionen:**

<b>rddfl_gi</b>	...	liest Datei mit Beschreibung eines 3D-Objektes,
<b>frell_gi</b>	...	gibt Speicherplatz einer in rddfl_gi angelegten GI_ELEM-Liste frei,
<b>insol_gi</b>	...	legt geordnete doppelt verkettete Liste mit GI_OBJ-Strukturen an,
<b>insot_gi</b>	...	legt geordneten bnären Baum mit GI_OBJ-Strukturen an,
<b>desot_gi</b>	...	gibt Speicherplatz einer GI_OBJ-Liste bzw. eines GI_OBJ-Baums frei,
<b>flini_gi</b>	...	Initialisierungs-Funktion, muss vor flodl_gi aufgerufen werden,
<b>flodl_gi</b>	...	startet Windows-Dialog für Eingabe eines File-Namens,
<b>mkrbg_gi</b>	...	liefert GDI-Farben-Parameter für 8 Grundfarben.

## 4.2 Header-Datei giw.h

```

/*****
 * Header-Datei fuer die Einbindung der Library des Graphics-
 * Interface "GIW"
 *
 * Autor: J. Dankert
 *****/

#include <windows.h>

#define DOUBLE_EPS 1.e-12
#define MARDIV_GI 20 /* Zur Festlegung des Standard-Marker-Offsets: */
/* Offset = "1 Logical Inch" / MARDIV_GI */

#define GI_BLACK 0
#define GI_BLUE 1
#define GI_GREEN 2
#define GI_CYAN 3
#define GI_RED 4
#define GI_MAGENTA 5
#define GI_YELLOW 6
#define GI_WHITE 7
#define GI_TRANSPARENT -1
#define GI_TOPLEFT 1
#define GI_BASELEFT 2
#define GI_BOTTOMLEFT 3
#define GI_TOPCENTER 4
#define GI_BASECENTER 5
#define GI_BOTTOMCENTER 6
#define GI_TOPRIGHT 7
#define GI_BASERIGHT 8
#define GI_BOTTOMRIGHT 9
#define GI_XYBOTTOMLEFT 0
#define GI_XYCENTER 1
#define GI_XYTOPLEFT 2
#define GI_XYTOPRIGHT 3
#define GI_XYBOTTOMRIGHT 4
#define GI_MKCIRCLE 1
#define GI_MKSQUARE 2
#define GI_MKCROSS 3
#define GI_MKVBAR 4
#define GI_MKHBAR 5
#define GI_MKFCIRCLE 6
#define GI_MKFSQUARE 7
#define GI_UPLEFT 1
#define GI_LEFT 2
#define GI_DOWNLEFT 3
#define GI_UP 4
#define GI_CENTER 5
#define GI_DOWN 6
#define GI_UPRIGHT 7
#define GI_RIGHT 8
#define GI_DOWNRIGHT 9
#define GI_CUBIGCROSS 1
#define GI_CUSMALLCROSS 2
#define GI_CURECTANGLE 3
#define GI_CENTRAL 1
#define GI_PARALLEL 2
#define GI_AXISX 0
#define GI_AXISY 1
#define GI_AXISZ 2

```

```

typedef struct GI_OBJ_tag          /* ... fuer verkettete Listen */
{
    int          type ;           /* und binaere Baeume, die */
    void         *data ;          /* insol_gi und insot_gi */
    double       dist ;           /* angelegt werden */
    struct GI_OBJ_tag *left ;
    struct GI_OBJ_tag *right ;

}   GI_OBJ ;

typedef struct GI_ELEM_tag          /* ... fuer die Liste der */
{
    int          nop ;           /* Elemente, die von rddfl_gi */
    int          *param ;        /* angelegt wird */
    struct GI_ELEM_tag *next ;

}   GI_ELEM ;

/***** Prototypen der Interface-Routinen der GIW-TOOLBOX: *****/

void    desot_gi (GI_OBJ *root_p) ;
void    flini_gi (HWND hwnd) ;
int     flodl_gi (HWND hwnd, char *pathnm , char *filenm) ;
void    frell_gi (GI_ELEM *kroot_p) ;
void    gstop_gi (HWND hwnd) ;
HDC     gstrt_gi (HWND hwnd , int cxClient , int cyClient) ;

int     insol_gi (GI_OBJ **root_pp , int    type , void    *data ,
double   xw      , double yw      , double zw) ;
int     insot_gi (GI_OBJ **root_pp , int    type , void    *data ,
double   xw      , double yw      , double zw) ;
COLORREF mkrrgb_gi (int color) ;
int     prfpl_gi (HDC hdc , int    npoin , double xw [] ,
double   yw [] , double zw [] ) ;
void    prgtd_gi (double *xwd , double *ywd , double *zwd) ;
void    prgte_gi (double *xwe , double *ywe , double *zwe) ;
int     prgtp_gi () ;
void    prgtr_gi (double *xwr , double *ywr , double *zwr) ;
void    prini_gi () ;
int     prlin_gi (HDC hdc , double xw , double yw , double zw) ;
int     prmov_gi (HDC hdc , double xw , double yw , double zw) ;
void    projn_gi (int ptyp) ;
int     prstc_gi (double xwe , double ywe , double zwe ,
double   xwr , double ywr , double zwr) ;
int     prstd_gi (double xwd , double ywd , double zwd) ;
int     prste_gi (double xwe , double ywe , double zwe) ;
int     prstp_gi (double xwd , double ywd , double zwd ,
double   xwr , double ywr , double zwr) ;
int     prstr_gi (double xwr , double ywr , double zwr) ;
int     prw2u_gi (double xw , double yw , double zw ,
double   *xu , double *yu) ;
double  ptdis_gi (double xw , double yw , double zw) ;
int     ptfpl_gi (HDC hdc , int npoin ,
double   xw [] , double yw [] , double zw [] ) ;
void    ptini_gi () ;
int     ptmov_gi (HDC hdc , double xw , double yw , double zw) ;
int     ptmrk_gi (HDC     hdc , int    mtype , double msize ,
double   xw , double yw , double zw , int offset) ;
int     ptmx3_gi (int    npoin , double xyzw [] ,
double   *xumin , double *xumax ,
double   *yumin , double *yumax) ;
int     ptlin_gi (HDC hdc , double xw , double yw , double zw) ;
int     ptw2u_gi (double xw , double yw , double zw ,
double   *xu , double *yu) ;
int     ptwdl_gi (HDC hdc , double x1w , double y1w , double z1w ,
double   x2w , double y2w , double z2w ,
int      wpix , COLORREF wcol ,
int      ipix , COLORREF icol) ;

```

```

int      rddfl_gi (HWND hwnd , char *pathnm , int      *ne_p , int  *nk_p ,
                  int  *ke_p , double **xy_pp , GI_ELEM **km_pp) ;
int      rpick_gi (HWND hwnd , LONG lParam , double *xlu , double *ylu ,
                  double *x2u , double *y2u) ;
void     scapp_gi (HWND hwnd , int xv , int yv , int ctype) ;
void     scdap_gi (HWND hwnd) ;
void     scupd_gi (HWND hwnd , LONG lParam) ;
void     stcvp_gi (HDC  hdc , int pulx , int puly ,
                  int width , int height , int cstype) ;
void     stuca_gi (double p1xu , double p1yu ,
                  double p2xu , double p2yu , double pmarg) ;
void     stuci_gi (double p1xu , double p1yu ,
                  double p2xu , double p2yu , double pmarg) ;
void     t3gtm_gi (double *tm) ;
void     t3mam_gi (double *tnew) ;
void     t3rot_gi (double phi , int axis) ;
void     t3stm_gi (double *tm) ;
void     t3trn_gi (double tx , double ty , double tz) ;
void     t3w2w_gi (double xw , double yw , double zw ,
                  double *txw , double *tyw , double *tzw) ;
void     tdarc_gi (HDC  hdc , double xc , double yc , double r ,
                  double xs , double ys , double xe , double ye) ;
void     tfcir_gi (HDC  hdc , double xc , double yc , double r) ;
void     tgttm_gi (double *tm) ;
void     tinit_gi () ;
void     tline_gi (HDC  hdc , double xu , double yu) ;
void     tmamu_gi (double *tnew) ;
void     tmirx_gi () ;
void     tmiry_gi () ;
void     tmove_gi (HDC  hdc , double xu , double yu) ;
void     trota_gi (double xm , double ym , double phi) ;
void     tru2t_gi (double xu , double yu , double *txu , double *tyu) ;
void     tsttm_gi (double *tm) ;
void     ttabs_gi (double tx , double ty , double phi ,
                  double sx , double sy) ;
void     ttran_gi (double tx , double ty) ;
void     udarc_gi (HDC  hdc , double xc , double yc , double r ,
                  double xs , double ys , double xe , double ye) ;
void     ufcir_gi (HDC  hdc , double xc , double yc , double r) ;
void     ufell_gi (HDC  hdc , double xu1 , double yu1 ,
                  double xu2 , double yu2) ;
void     ufel2_gi (HDC  hdc , double xc , double yc ,
                  double a , double b) ;
void     ufpie_gi (HDC  hdc , double x1 , double y1 , double x2 , double y2 ,
                  double xs , double ys , double xe , double ye) ;
void     ufpi2_gi (HDC  hdc , double xc , double yc , double a , double b ,
                  double xs , double ys , double xe , double ye) ;
int      ufpol_gi (HDC  hdc , int npoin , double xu [] , double yu []) ;
void     ufrec_gi (HDC  hdc , double xu1 , double yu1 ,
                  double xu2 , double yu2) ;
void     ufsec_gi (HDC  hdc , double xc , double yc , double r ,
                  double xs , double ys , double xe , double ye) ;
void     uline_gi (HDC  hdc , double xu , double yu) ;
void     umark_gi (HDC  hdc , int  mtype , double msize ,
                  double xu , double yu , int offset) ;
void     umove_gi (HDC  hdc , double xu , double yu) ;
int      umpos_gi (LONG lParam , double *xu , double *yu) ;
void     uwidl_gi (HDC  hdc , double xu1 , double yu1 ,
                  double xu2 , double yu2 ,
                  int wpix , COLORREF wcol ,
                  int ipix , COLORREF icol) ;
int      upick_gi (HWND hwnd , LONG lParam , double *xu , double *yu) ;
void     vfram_gi (HDC  hdc , int offset) ;
void     vfrec_gi (HDC  hdc , int xv1 , int yv1 , int xv2 , int yv2) ;
void     vline_gi (HDC  hdc , int xv , int yv) ;
void     vmove_gi (HDC  hdc , int xv , int yv) ;
void     vrect_gi (HDC  hdc , int xv1 , int yv1 , int xv2 , int yv2) ;

```

### 4.3 GIW-Funktionen in alphabetischer Reihenfolge

```

/*****
 * ### Graphics Interface ###
 *
 * Freigeben der GI_OBJ-Liste bzw. des GI_OBJ-Trees
 * =====
 *
 * Der dynamische Speicherplatz, der beim Anlegen einer
 * doppelt verketteten Liste mit insol_gi bzw. eines
 * binären Baumes mit insot_gi fuer die GI_OBJ-Strukturen
 * und deren Komponenten angefordert wurde, wird
 * komplett freigegeben.
 *
 * Eingabe: root_p - Pointer auf das "Anchor-Element"
 *             der verketteten Liste bzw. des
 *             binären Baumes (muss den Wert
 *             haben, der auf der Argument-
 *             Position 1 beim letzten insol_gi-
 *             bzw. insot_gi-Aufruf abgeliefert
 *             wurde)
 *
 * Autor: J. Dankert
 *****/

```

```
void desot_gi (GI_OBJ *root_p)
```

```

/*****
 * ### Graphics Interface ###
 *
 * Initialisieren der OPENFILENAME-Struktur
 * =====
 *
 * Eine OPENFILENAME-Struktur wird fuer den typischen
 * Windows-Dialog zum Oeffnen eines Files benoetigt, der
 * von der GIW-Funktion flodl_gi gestartet wird.
 *
 * Die Funktion flini_gi muss vor spaeteren flodl_gi-
 * Aufrufen einmal zum Initialisieren der OPENFILENAME-
 * Struktur aufgerufen werden.
 *
 * Eingabe: hwnd - "Handle of window" des "Parent
 *             windows", aus dem der Dialog
 *             gestartet werden soll
 *
 * Autor: J. Dankert
 *****/

```

```
void flini_gi (HWND hwnd)
```

```

/*****
* ### Graphics Interface ###
*
* Dialog zum Oeffnen eines Files
* =====
*
* Funktion startet den typischen Windows-Dialog zum
* Festlegen des Namens eines zu oeffnenden Files (vor
* dem ersten flodl_gi-Aufruf muss flinit_gi aufgerufen
* werden).
*
* Abgeliefert werden (bei einem Return-Wert ungleich 0)
* der ausgewaehlte File-Name und der komplette Pfadname
* dieses Files.
*
* Eingabe: hwnd - "Handle of window" des "Parent
*               windows", aus dem der Dialog
*               gestartet werden soll
*
* Ausgabe: pathnm - Kompletter Pfadname des Files
*               (Argument sollte vorsichtshalber
*               Platz fuer _MAX_PATH Characters
*               bereitstellen)
*           filename - File-Name (Argument sollte
*               vorsichtshalber Platz fuer
*               _MAX_FNAME + _MAX_EXT Characters
*               bereitstellen)
*
* Return-Wert: 1 ---> Dialog erfolgreich
*              0 ---> Dialog wurde abgebrochen
*
* Autor: J. Dankert
*****/
int flodl_gi (HWND hwnd, char *pathnm , char *filename)

```

```

/*****
* ### Graphics Interface ###
*
* Freigeben der verketteten Element-Liste
* =====
*
* Der dynamische Speicherplatz, der beim Anlegen einer
* Element-Liste mit rddfl_gi fuer die GI_ELEM-Strukturen
* und deren Komponenten angefordert wurde, wird
* komplett freigegeben.
*
* Eingabe: kroot_p - Pointer auf das "Anchor-Element"
*               der verketteten Liste (wurde auf
*               der letzten Argument-Position
*               beim rddfl_gi-Aufruf abgeliefert)
*
* Autor: J. Dankert
*****/
void frell_gi (GI_ELEM *kroot_p)

```

```

/*****
* ### Graphics Interface ###
*
* Beenden der Graphikausgabe
* =====
*
* ... im Fenster hwnd, die mit gstrt_gi gestartet wurde.
* Die Funktionen gstrt_gi und gstop_gi sollten aus-
* schliesslich innerhalb der Bearbeitung der Message
* WM_PAINT verwendet werden und die Aufrufe der GIW-
* Zeichenroutinen einrahmen.
*
* Die Funktion gstop_gi ruft die Windows-Funktion EndPaint
* auf (gstrt_gi ruft BeginPaint).
*
* Eingabe: hwnd      - Handle des Fensters, in das ge-
*                  zeichnet wurde
*
* Autor: J. Dankert
*****/

```

```
void gstop_gi (HWND hwnd)
```

```

/*****
* ### Graphics Interface ###
*
* Starten der Graphikausgabe
* =====
*
* Eine Graphikausgabe-Aktion bei der Bearbeitung der
* Message WM_PAINT sollte von Aufrufen von
*
*          gstrt_gi      bzw.      gstop_gi
*
* "eingerahmt" werden. Der Funktion gstrt_gi werden
* Fenster-Handle und Fenster-Abmessungen uebergeben, die
* wesentlichen Zeichenparameter werden initialisiert.
* Abgeliefert wird als Return-Wert ein "Handle of Device
* Context", der allen Zeichen-Funktionen als
* Identifikator uebergeben wird.
*
* Nach gstrt_gi ist das gesamte Fenster mit dem "Current
* Viewport" identisch, das (Geraete-)Viewport-Koordinaten-
* system liegt mit seinem Ursprung in der linken oberen
* Ecke, die Achsen zeigen nach rechts bzw. unten.
*
* Eingabe: hwnd      - Handle des Fensters, in das ge-
*                  zeichnet werden soll (Client Area)
*          cxClient   ,
*          cyClient   - Horizontale und vertikale Pixel-
*                  Anzahl der "Client Area" (Werte,
*                  die z. B. bei WM_SIZE als
*                  LOWORD (lParam) und HIWORD (lParam)
*                  bereitgestellt werden
*
* Return-Wert: "Handle of Device Context", wurde von
*              BeginPaint erzeugt und abgeliefert
*
* Autor: J. Dankert
*****/

```

```
HDC gstrt_gi (HWND hwnd , int cxClient , int cyClient)
```

```

/*****
* ### Graphics Interface ###
*
* Einfuegen eines Objektes in die geordnete verkettete
* Liste
* =====
*
* Es wird eine Struktur des Typs GI_OBJ (Typ-Definition
* in giw.h) erzeugt und "geordnet" in eine doppelt
* verkettete Liste eingefuegt. Die Funktion dient dazu,
* Zeichnungs-Elemente nach ihrem "Abstand vom Betrachter"
* zu ordnen (die am weitesten entfernten Elemente stehen
* an der Spitze der Liste).
*
* Die Entfernung des Zeichnungs-Elements vom Betrachter
* wird durch den Punkt bestimmt, dessen "World
* Coordinates" als xw, yw, zw uebergeben werden. Berechnet
* wird der Abstand in Abhaengigkeit von der gueltigen
* 3D-Transformation und Projektion. Dazu wird in insol_gi
* die Funktion ptdis_gi aufgerufen.
*
* Angeliefert werden muessen ausserdem der "Pointer auf
* den Pointer" des "Anchor-Elements" der Liste (wenn
* die Liste noch nicht existiert, muss dieser Pointer
* den Wert NULL haben, wird dann in insol_gi geaendert)
* und die Argumente type und data (Pointer), die in die
* von insol_gi anzulegende Struktur uebertragen werden
* (mit diesen Groessen identifiziert der Programmierer
* das Graphik-Element).
*
* Eingabe: *root_pp - Pointer auf "Anchor-Pointer"
*               der Liste ("Anchor-Pointer kann
*               sich in insol_gi aendern)
*           type      - Identifikator f#r das Element
*               (wird vom Programmierer will-
*               kuerlich vergeben)
*           data      - Pointer auf die beschreibenden
*               Daten des Elements (Datenstruktur
*               wird vom Programmierer definiert)
*           xw,
*           yw,
*           zw        - Charakteristischer Punkt ("World
*               Coordinates"), mit dem die
*               Entfernung des Elements vom
*               Betrachter berechnet wird
*               (Entfernung wird als Komponente
*               dist in die GI_OBJ-Struktur
*               eingetragen)
*
* Ausgabe  *root_pp - Eventuell geaenderter "Anchor-
*               Pointer" der Liste (beim Anlegen
*               der Liste oder dann, wenn das
*               neue Element an die Spitze der
*               Liste gesetzt wird)
*
* Return-Wert:  1 ---> Erfolg
*               0 ---> Misserfolg (z. B.: Kein
*               Speicherplatz)
*
* Autor: J. Dankert
*****/

int insol_gi (GI_OBJ **root_pp , int    type , void  *data ,
              double xw      , double yw  , double zw)

```



```

/*****
* ### Graphics Interface ###
*
* Einfuegen eines Objektes in den geordneten
* binaeren Baum
* =====
*
* Es wird eine Struktur des Typs GI_OBJ (Typ-Definition
* in giw.h) erzeugt und "geordnet" in einen "binaeren
* Baum eingefuegt. Die Funktion dient dazu, Zeichnungs-
* Elemente nach ihrem "Abstand vom Betrachter" zu ordnen
* (ein "linker Nachfolger" ist immer weiter entfernt als
* das Objekt, das auf ihn zeigt).
*
* Die Entfernung des Zeichnungs-Elements vom Betrachter
* wird durch den Punkt bestimmt, dessen "World
* Coordinates" als xw, yw, zw uebergeben werden. Berechnet
* wird der Abstand in Abhaengigkeit von der gueltigen
* 3D-Transformation und Projektion. Dazu wird in insot_gi
* die Funktion ptdis_gi aufgerufen.
*
* Angeliefert werden muessen ausserdem der "Pointer auf
* den Pointer" des "Anchor-Elements" deBaums (wenn
* der Baum noch nicht existiert, muss dieser Pointer
* den Wert NULL haben, wird dann in insot_gi geaendert)
* und die Argumente type und data (Pointer), die in die
* von insot_gi anzulegende Struktur uebertragen werden
* (mit diesen Groessen identifiziert der Programmierer
* das Graphik-Element).
*
* Eingabe:  *root_pp  -  Pointer auf "Anchor-Pointer"
*                  des Baums ("Anchor-Pointer kann
*                  sich in insot_gi aendern)
*           type      -  Identifikator f"r das Element
*                  (wird vom Programmierer will-
*                  kuerlich vergeben)
*           data      -  Pointer auf die beschreibenden
*                  Daten des Elements (Datenstruktur
*                  wird vom Programmierer definiert)
*           xw,
*           yw,
*           zw        -  Charakteristischer Punkt ("World
*                  Coordinates"), mit dem die
*                  Entfernung des Elements vom
*                  Betrachter berechnet wird
*                  (Entfernung wird als Komponente
*                  dist in die GI_OBJ-Struktur
*                  eingetragen)
*
* Ausgabe   *root_pp  -  Eventuell geaenderter "Anchor-
*                  Pointer" des Baums (beim Anlegen
*                  des Baums)
*
* Return-Wert:  1  --->  Erfolg
*               0  --->  Misserfolg (z. B.: Kein
*                  Speicherplatz)
*
* Autor: J. Dankert
*****/
int insot_gi (GI_OBJ **root_pp , int    type , void    *data ,
             double xw      , double yw  , double zw)

```

```

/*****
* ### Graphics Interface ###
*
* Umrechnen der einfachen GIW-Farbnummer in COLORREF-Wert,
* der RGB-Werte repräsentiert
* =====
*
* Der Rueckgabewert der Funktion mkrbg_gi entspricht
* dem Ergebnis, das von dem in windows.h definierten
* Makro RGB abgeliefert wird und z. B. von der Windows-
* Funktion CreatePen als Eingabewert erwartet wird.
*
* Es kann eine von 8 Grundfarben umgerechnet werden:
*
* Eingabe: color =  GI_BLACK    (0)  --->  Schwarz
*                =  GI_BLUE    (1)  --->  Blau
*                =  GI_GREEN   (2)  --->  Gruen
*                =  GI_CYAN    (3)  --->  Cyan
*                =  GI_RED     (4)  --->  Rot
*                =  GI_MAGENTA (5)  --->  Magenta
*                =  GI_YELLOW  (6)  --->  Gelb
*                =  GI_WHITE   (7)  --->  Weiss
*
* Wenn der Wert von color ausserhalb dieses Bereichs
* liegt, wird bei negativem Wert der absolute Betrag, bei
* Werten groesser als 7 der Wert (color modulo 8)
* verwendet.
*
* Autor: J. Dankert
*****/

```

```
COLORREF mkrbg_gi (int color)
```

```

/*****
* ### Graphics Interface ###
*
* Zeichnen eines gefuellten 3D-Polygons, dessen Punkte
* vorher auf die Bildebene projiziert werden
* =====
*
* Der Rand des Polygons wird mit der Farbe des aktuellen
* Zeichenstiftes" gezeichnet, ausgefuellt wird das
* Polygon mit der Farbe des aktuellen "Brushs".
*
* Eingabe: hdc      - "Handle of Device Context"
*          npoin    - Anzahl der Punkte (das Polygon wird
*                  in jedem Fall geschlossen, fuer ein
*                  n-seitiges Polygon muessen also n
*                  Punkte eingegeben werden)
*          xw []    - Feld mit npoin xw-Koordinaten
*          yw []    - Feld mit npoin yw-Koordinaten
*          zw []    - Feld mit npoin zw-Koordinaten
*                  ("World Coordinates")
*
* Retrun-Wert:  1  -->  Projektion gelungen
*              2  -->  Projektion misslungen (es wurde
*                  nichts gezeichnet), z. B.:
*                  Ein Polygonpunkt lag hinter dem
*                  "Eye Point"
*
* Autor: J. Dankert
*****/

```

```
int prfpl_gi (HDC hdc , int      npoin , double xw [] ,
              double yw [] , double zw [])
```

```

/*****
* ### Graphics Interface ###
*
* Koordinaten der aktuellen "Blickrichtung"
* (Parallelprojektion) in "World Coordinates"
* =====
*
* Die "Blickrichtung" wird durch einen beliebigen Vektor
* definiert (das "Auge" befindet sich im Unendlichen in
* der Richtung des NEGATIVEN "Blickrichtungs"-Vektors).
* Die Projektionsebene liegt bei der Parallelprojektion
* immer senkrecht zum "Blickrichtungs"-Vektor.
*
* Eingabe: Keine
*
* Ausgabe: xwd -
*          ywd -
*          zwd - "World Coordinates" der "Blickrichtung"
*
* Autor: J. Dankert
*****/

void prgtd_gi (double *xwd , double *ywd , double *zwd)

/*****
* ### Graphics Interface ###
*
* Koordinaten des aktuellen "Eye Points" in
* "World Coordinates"
* =====
*
* Der "Eye Point" ("Projektionszentrum") definiert gemein-
* sam mit dem Referenzpunkt der Projektionsebene (vgl.
* prgtr_gi) die ZENTRALPROJEKTION. Die Blickrichtung
* ist bei der Zentralprojektion immer die Richtung vom
* "Eye Point" zum Referenzpunkt der Projektionsebene.
* Die Projektionsebene steht senkrecht auf dieser Blick-
* richtung.
*
* Eingabe: Keine
*
* Ausgabe: xwe -
*          ywe -
*          zwe - "World Coordinates" des "Eye Points"
*
* Autor: J. Dankert
*****/

void prgte_gi (double *xwe , double *ywe , double *zwe)

/*****
* ### Graphics Interface ###
*
* Aktuell eingestellte Transformation
* =====
*
* Return-Wert: 1 ---> Zentralprojektion
*              2 ---> Parallelprojektion
*
* Autor: J. Dankert
*****/

int prgtp_gi ()

```

```

/*****
* ### Graphics Interface ###
*
* Koordinaten des aktuellen Referenzpunktes
* in "World Coordinates"
* =====
*
* Der Referenzpunkt wird sowohl fuer die Definition der
* Zentralprojektion als auch fuer die Definition der
* Parallelprojektion verwendet. Es ist in jedem Fall
* ein Punkt der Projektionsebene, die durch den Referenz-
* punkt und die "Blickrichtung" definiert wird (Projek-
* tionsebene liegt senkrecht zur "Blickrichtung").
*
* Bei der ZENTRALPROJEKTION ist die "Blickrichtung" die
* Verbindungslinie vom "Eye Point" (Projektionszentrum,
* vgl. prgte_gi) zum Referenzpunkt.
*
* Bei der PARALLELPROJEKTION wird die "Blickrichtung"
* durch den "Blickrichtungs"-Vektor (vgl. prgtd_gi)
* festgelegt.
*
* Der Referenzpunkt wird in beiden Projektionen auch als
* Ursprung des (ebenen) Bildkoordinatensystems verwendet.
*
* Eingabe: Keine
*
* Ausgabe: xwr -
*          ywr -
*          zwr - "World Coordinates" des Referenzpunktes
*
* Autor: J. Dankert
*****/
void prgtr_gi (double *xwr , double *ywr , double *zwr)

```

```

/*****
* ### Graphics Interface ###
*
* Initialisieren der Projektions-Parameter
* =====
*
* Die Parameter der 3D-Projektion werden wie folgt
* initialisiert:
*
* * "Eye Point" der Zentralprojektion liegt bei
*   (2000 ; -5000 ; 2000),
*
* * "Blickrichtungsvektor" für Parallelprojektion ist
*   (-2000 ; 5000 ; 2000),
*
* * Referenzpunkt der Projektionsebene fuer beide
*   Projektionen ist der Nullpunkt des 3D-Koordinaten-
*   systems ("World Coordinates"),
*
* * Up-Vector zur Definition des 2D-Koordinatensystems
*   in der Zeichenebene zeigt in Richtung der positiven
*   z-Achse des 3D-Koordinatensystems,
*
* * als aktuelle Projektion wird "Parallelprojektion"
*   eingestellt.
*
* Eingabe: Keine
*
* Autor: J. Dankert
*****/

```

```
void prini_gi ()
```

```

/*****
* ### Graphics Interface ###
*
* Zeichnen einer geraden Linie von der "Current Position"
* zu einem Punkt, der vorher auf die Bildebene
* transformiert wird
* =====
*
* Eingabe: hdc - "Handle of Device Context"
*          xw ,
*          yw ,
*          zw - "World Coordinates" des Zielpunktes
*
* Return-Wert: 0 ---> Projektion unmöglich, keine
*                  Aktion ausgefuehrt
*               1 ---> Aktion ausgefuehrt
*
* Autor: J. Dankert
*****/

```

```
int prlin_gi (HDC hdc , double xw , double yw , double zw)
```

```

/*****
* ### Graphics Interface ###
*
* Bewegen des "Zeichenstiftes" zu einem Punkt, der
* vorher auf die Bildebene transformiert wird
* =====
*
* Eingabe:  hdc  -  "Handle of Device Context", der von
*                gstrt_gi geliefert wird
*           xw  ,
*           yw  ,
*           zw  -  "World Coordinates" des Zielpunktes
*
* Return-Wert:  0  --->  Projektion unmöglich, keine
*                   Aktion ausgefuehrt
*               1  --->  Aktion ausgefuehrt
*
* Autor: J. Dankert
*****/

```

```
int prmov_gi (HDC hdc , double xw , double yw , double zw)
```

```

/*****
* ### Graphics Interface ###
*
* Einstellen: Zentralprojektion oder Parallelprojektion
* =====
*
* Es wird eingestellt, ob die nachfolgenden graphischen
* Darstellungen dreidimensionaler Objekte in Zentral-
* oder Parallel-Projektion ausgefuehrt werden.
*
* Als "Rerenzpunkt", "Eye Point" bzw. Blickrichtungsvektor
* werden die durch die Intialisierung vorgegebenen
* Standardwerte oder die durch die Routinen prstc_gi,
* prstp_gi, prste_gi, prstr_gi und prstd_gi eingestellten
* Werte verwendet.
*
* Eingabe:  ptyp = GI_CENTRAL  --->  Zentralprojektion
*           = GI_PARALLEL  --->  Parallelprojektion
*
* Autor: J. Dankert
*****/

```

```
void projn_gi (int ptyp)
```

```

/*****
* ### Graphics Interface ###
*
* Definition einer Zentralprojektion
* =====
*
* Eine Zentralprojektion wird definiert durch das
* Projektionszentrum ("Eye Point") xwe,ywe,zwe und einen
* "Referenzpunkt" xwr,ywr,zwr der Projektionsebene.
*
* Der Referenzpunkt wird als "Hauptpunkt" der Projektion
* (Sehstrahl vom "Eye Point" zum "Hauptpunkt" steht
* senkrecht auf der Projektionsebene) und als Ursprung des
* in der Projektionsebene liegenden (zweidimensionalen)
* "Bildebenen-Koordinatensystems u,v" verwendet.
*
* Die Richtung der v-Achse des Bildebenen-Koordinaten-
* system entspricht der Projektion eines zur z-Achse
* ("World Coordinates") parallelen Vektors im Hauptpunkt
* auf die Bildebene. Wenn der "Eye Point" auf der z-Achse
* liegt (Projektionsebene ist senkrecht zur z-Achse), wird
* ein Vektor parallel zur y-Achse ("World Coordinates")
* zur Definition der Richtung der v-Achse benutzt.
*
* Die u-Achse des Bildebenen-Koordinatensystems steht
* senkrecht auf der v-Achse und ist so gerichtet, daß
* u-Achse, v-Achse und ein vom Referenzpunkt auf den
* "Eye Point" gerichteter Vektor in dieser Reihenfolge
* ein Rechtssystem bilden.
*
* Eingabe:  xwe -
*           ywe -
*           zwe - "World Coordinates" des "Eye Points"
*           xwr -
*           ywr -
*           zwr - "World Coordinates" des Referenzpunktes
*                der Projektionsebene
*
* Return-Wert:  1 --> Projektion wurde akzeptiert
*               0 --> Projektion wurde nicht akzeptiert
*
* Autor: J. Dankert
*****/

int prstc_gi (double xwe , double ywe , double zwe ,
              double xwr , double ywr , double zwr)

```

```

/*****
* ### Graphics Interface ###
*
* Setzen des Blickrichtungs-Vektors der Parallelprojektion
* =====
*
* Eine Parallelprojektion wird definiert durch die
* "Blickrichtung" (Vektor xwd,ywd,zwd) und einen
* "Referenzpunkt" xwr,ywr,zwr der Projektionsebene.
*
* Die Projektionsebene liegt senkrecht zur Blickrichtung,
* der Referenzpunkt wird als Ursprung des in der
* Projektionsebene liegenden (zweidimensionalen)
* "Bildebenen-Koordinatensystems u,v" verwendet.
*
* Es wird eine neue Parallelprojektion mit einer neuen
* "Blickrichtung" bei unverändertem Referenzpunkt
* definiert. Da die Blickrichtung senkrecht zur
* Projektionsebene steht ändert sich trotzdem die Lage
* der Projektionsebene.
*
* Wenn die neue Projektion akzeptiert werden kann, wird in
* jedem Fall "Parallelprojektion" fuer weitere Darstel-
* lungen eingestellt, auch wenn vorher "Zentral-
* projektion" eingestellt war.
*
* Eingabe:  xwd -
*           ywd -
*           zwd - "World Coordinates" des Vektors der
*                Blickrichtung
*
* Return-Wert:  1 --> Projektion wurde akzeptiert
*               0 --> Projektion wurde nicht akzeptiert
*
* Autor: J. Dankert
*****/
int prstd_gi (double xwd , double ywd , double zwd)

```



```

/*****
* ### Graphics Interface ###
*
* Setzen des "Eye Points" der Zentralprojektion
* =====
*
* Eine Zentralprojektion wird definiert durch das
* Projektionszentrum ("Eye Point") xwe,ywe,zwe und einen
* "Referenzpunkt" xwr,ywr,zwr der Projektionsebene.
*
* Der Referenzpunkt wird als "Hauptpunkt" der Projektion
* (Sehstrahl vom "Eye Point" zum "Hauptpunkt" steht
* senkrecht auf der Projektionsebene) und als Ursprung des
* in der Projektionsebene liegenden (zweidimensionalen)
* "Bildebenen-Koordinatensystems u,v" verwendet.
*
* Es wird eine neue Zentralprojektion mit einem neuen
* "Eye Point" bei unverändertem Referenzpunkt definiert.
* Da der Sehstrahl senkrecht auf der Projektionsebene
* steht ändert sich trotzdem die Lage der Projektions-
* ebene.
*
* Wenn die neue Projektion akzeptiert werden kann, wird in
* jedem Fall "Zentralprojektion" fuer weitere Darstel-
* lungen eingestellt, auch wenn vorher "Parallel-
* projektion" eingestellt war.
*
* Eingabe:  xwe -
*           ywe -
*           zwe - "World Coordinates" des "Eye Points"
*
* Return-Wert:  1 --> Projektion wurde akzeptiert
*              0 --> Projektion wurde nicht akzeptiert
*
* Autor: J. Dankert
*****/
int prste_gi (double xwe , double ywe , double zwe)

```

```

/*****
* ### Graphics Interface ###
*
* Definition einer Parallelprojektion
* =====
*
* Eine Parallelprojektion wird definiert durch die
* "Blickrichtung" (Vektor xwd,ywd,zwd) und einen
* "Referenzpunkt" xwr,ywr,zwr der Projektionsebene.
*
* Die Projektionsebene liegt senkrecht zur Blickrichtung,
* der Referenzpunkt wird als Ursprung des in der
* Projektionsebene liegenden (zweidimensionalen)
* "Bildebenen-Koordinatensystems u,v" verwendet.
*
* Die Richtung der v-Achse des Bildebenen-Koordinaten-
* system entspricht der Projektion eines zur z-Achse
* ("World Coordinates") parallelen Vektors im Referenz-
* punkt auf die Bildebene. Wenn die Blickrichtung parallel
* zur z-Achse verläuft (Projektionsebene ist senkrecht
* zur z-Achse), wird ein Vektor parallel zur y-Achse
* ("World Coordinates") zur Definition der Richtung der
* v-Achse benutzt.
*
* Die u-Achse des Bildebenen-Koordinatensystems steht
* senkrecht auf der v-Achse und ist so gerichtet, dass
* u-Achse, v-Achse und der Vektor der Blickrichtung in
* dieser Reihenfolge ein Linkssystem bilden.
*
* Eingabe:  xwd -
*           ywd -
*           zwd - "World Coordinates" des Vektors der
*                Blickrichtung
*           xwr -
*           ywr -
*           zwr - "World Coordinates" des Referenzpunktes
*                der Projektionsebene
*
* Return-Wert:  1 --> Projektion wurde akzeptiert
*               0 --> Projektion wurde nicht akzeptiert
*
* Autor: J. Dankert
*****/

int prstp_gi (double xwd , double ywd , double zwd ,
              double xwr , double ywr , double zwr)

```

```

/*****
* ### Graphics Interface ###
*
* Setzen des Referenzpunktes der Projektionsebene
* =====
*
* Eine Zentralprojektion wird definiert durch das
* Projektionszentrum ("Eye Point") xwe,ywe,zwe und einen
* "Referenzpunkt" xwr,ywr,zwr der Projektionsebene
* (vgl. Funktion prstc_gi).
*
* Eine Parallelprojektion wird definiert durch die
* "Blickrichtung" (Vektor xwd,ywd,zwd) und einen
* "Referenzpunkt" xwr,ywr,zwr der Projektionsebene
* (vgl. Funktion prstp_gi).
*
* Der Referenzpunkt wird bei der Zentralprojektion als
* "Hauptpunkt" (Sehstrahl vom "Eye Point" zum "Hauptpunkt"
* steht senkrecht auf der Projektionsebene) und bei beiden
* Projektionen als Ursprung des in der Projektionsebene
* liegenden (zweidimensionalen) "Bildebenen-
* Koordinatensystems u,v" verwendet.
*
* Es wird eine neue Projektion mit einem neuen Referenz-
* punkt bei unverändertem "Eye Point" (Zentralprojektion)
* bzw. unveränderter "Blickrichtung" (Parallelprojektion)
* definiert.
*
* Eingabe:  xwr -
*           ywr -
*           zwr - "World Coordinates" des Referenz-
*                 punktes der Projektionsebene
*
* Return-Wert:  1 --> Projektion wurde akzeptiert
*               0 --> Projektion wurde nicht akzeptiert
*
* Autor: J. Dankert
*****/

```

```
int prstr_gi (double xwr , double ywr , double zwr)
```

```

/*****
* ### Graphics Interface ###
*
* Transformation eines in "World Coordinates" gegebenen
* Punktes mit der aktuellen Projektionsmatrix in das
* Bildkoordinatensystem "User Coordinates"
* =====
*
* Eingabe:  xw      - x-Koordinate ("World Coordinates")
*           yw      - y-Koordinate ("World Coordinates")
*           zw      - z-Koordinate ("World Coordinates")
*
* Ausgabe:  xu      - Transformierte x-Koordinate
*           yu      - Transformierte y-Koordinate
*
* Return-Wert:  0 --> Projektion misslungen
*               1 --> Projektion geglueckt
*
* Autor: J. Dankert
*****/

```

```
int prw2u_gi (double xw , double yw , double zw ,
              double *xu , double *yu)
```

```

/*****
* ### Graphics Interface ###
*
* Berechnen der Distanz eines Punktes (nach Transfor-
* tion) vom "Eye Point" bzw. von der Projektionsebene
* =====
*
* Die Funktion ptdis_gi liefert nur ein vergleichendes
* Mass fuer den Abstand eines Punktes ("World
* Coordinates") vom Betrachter, der fuer das Sortieren
* (z. B. fuer "Hidden Line"- und "Hidden Surface"-
* Darstellungen geeignet ist:
*
* * Bei eingestellter Zentralprojektion wird das Quadrat
*   des Abstandes vom "Eye Point" abgeliefert.
*
* * Bei eingestellter Parallelprojektion wird ein
*   vorzeichenbehafteter Wert geliefert, der der mit
*   einem konstanten Faktor multiplizierte Abstand
*   des Punktes von der Projektionsebene ist.
*
* Eingabe:  xw ,
*           yw ,
*           zw  - "World Coordinates" des Punktes
*
* Return-Wert:  Mass fuer den Abstand vom Betrachter
*
* Autor: J. Dankert
*****/
double ptdis_gi (double xw , double yw , double zw)

```

```

/*****
* ### Graphics Interface ###
*
* Zeichnen eines gefuellten Polygons, dessen Eckpunkte in
* "World Coordinates" gegeben sind, die vor dem Zeichnen
* der aktuellen Transformation unterworfen und danach
* auf "User Coordinates" projiziert werden
* =====
*
* Der Rand des Polygons wird mit der Farbe des aktuellen
* Zeichenstiftes" gezeichnet, ausgefuehlt wird das
* Polygon mit der Farbe des aktuellen "Brushs".
*
* Eingabe: hdc      - "Handle of Device Context"
*          npoin    - Anzahl der Punkte
*          xw []    - Feld mit npoin xw-Koordinaten
*          yw []    - Feld mit npoin yw-Koordinaten
*          zw []    - Feld mit npoin zw-Koordinaten
*
* Return-Wert:  0 ---> Projektion unmoeglich, keine
*                  Aktion ausgefuehrt
*               1 ---> Aktion ausgefuehrt
*
* Autor: J. Dankert
*****/
int ptfpl_gi (HDC hdc      , int npoin ,
              double xw [] , double yw [] , double zw [])

```

```

/*****
* ### Graphics Interface ###
*
* Initialisieren der Transformations- und
* Projektions-Parameter
* =====
*
* Die Funktion ptini_gi erledigt die Aufgaben von tinit_gi
* (Initialisieren der 2D- und der 3D-Transformation) und
* prini_gi (Initialisieren der Projektions-Parameter).
*
* Eingabe: Keine
*
* Autor: J. Dankert
*****/

```

```
void ptini_gi ()
```

```

/*****
* ### Graphics Interface ###
*
* Zeichnen einer geraden Linie von der "Current Position"
* zu einem Punkt, der vorher vorher der aktuellen
* 3D-Transformation unterworfen und danach auf die
* Bildebene transformiert wird
* =====
*
* Eingabe: hdc - "Handle of Device Context"
*          xw ,
*          yw ,
*          zw - "World Coordinates" des Zielpunktes
*
* Return-Wert: 0 ---> Projektion unmöglich, keine
*                Aktion ausgeführt
*              1 ---> Aktion ausgeführt
*
* Autor: J. Dankert
*****/

```

```
int ptlin_gi (HDC hdc , double xw , double yw , double zw)
```

```

/*****
* ### Graphics Interface ###
*
* Bewegen des "Zeichenstiftes" zu einem Punkt, der
* vorher der aktuellen 3D-Transformation unterworfen
* und danach auf die Bildebene transformiert wird
* =====
*
* Eingabe: hdc - "Handle of Device Context"
*          xw ,
*          yw ,
*          zw - "World Coordinates" des Zielpunktes
*
* Return-Wert: 0 ---> Projektion unmöglich, keine
*                Aktion ausgeführt
*              1 ---> Aktion ausgeführt
*
* Autor: J. Dankert
*****/

```

```
int ptmov_gi (HDC hdc , double xw , double yw , double zw)
```

```

/*****
* ### Graphics Interface ###
*
* Zeichnen eines Markers an einem Punkt, der vorher der
* aktuellen 3D-Transformation unterworfen und danach
* auf die Bildebene transformiert wird
* =====
*
* Es wird ein vordefiniertes Symbol (mtype) gezeichnet.
*
* Eine Aenderung der Fenstergroesse beeinflusst die
* Groesse der Marker nicht.
*
* Der Marker wird mit der Farbe des aktuellen "Zeichen-
* stiftes" gezeichnet, bei den Markertypen
* GI_MKFCIRCLE (6) und GI_MKFSQUARE (7) wird nur der
* Rand mit dieser Farbe gezeichnet, ausgefuellt werden
* diese beiden Marker mit der Farbe des aktuellen
* "Brushes".
*
* Die Funktion ptmrk_gi aendert die "Current Position",
* in Abhaengigkeit vom Parameter offset liegt sie im
* Zentrum des Markers (sinnvoll, wenn von dort aus die
* folgende Linie startet) oder auf dem Rand des den Marker
* umschliessenden Quadrats (sinnvoll, wenn der Marker
* anschliessend beschriftet werden soll).
*
* Eingabe: hdc      - "Handle of Device Context", der von
*                  gstrt_gi geliefert wird
*          mtype    - Markertyp:
*          = GI_MKFCIRCLE (1) --> Kreis
*          = GI_MKSQUARE (2) --> Quadrat
*          = GI_MKCROSS  (3) --> Kreuz
*          = GI_MKVBAR   (4) --> Vertikale Linie
*          = GI_MKHBAR   (5) --> Horizont. Linie
*          = GI_MKFCIRCLE (6) --> Gefuellt. Kreis
*          = GI_MKFSQUARE (7) --> Gef. Quadrat
*          msize     - Faktor zur Steuerung der Markergroesse
*          xw ,
*          yw ,
*          zw      - "World Coordinates" des Zielpunktes
*          offset   - "Current position" NACH dem Zeichnen
*                    des Markers:
*          = GI_UPLEFT  (1) --> links oben
*          = GI_LEFT    (2) --> links
*          = GI_DOWNLEFT (3) --> links unten
*          = GI_UP      (4) --> oben
*          = GI_CENTER  (5) --> Makermittle
*                    (auch bei offset=0)
*          = GI_DOWN    (6) --> unten
*          = GI_UPRIGHT (7) --> rechts oben
*          = GI_RIGHT   (8) --> rechts
*          = GI_DOWNRIGHT (9) --> rechts unten
*
* Return-Wert:  0 ---> Projektion unmoeglich, keine
*                  Aktion ausgefuehrt
*               1 ---> Aktion ausgefuehrt
*
* Autor: J. Dankert
*****/

int ptmrk_gi (HDC      hdc , int      mtype , double msize ,
              double xw , double yw , double zw , int offset)

```

```

/*****
* ### Graphics Interface ###
*
* Ermittlung der Maxima und Minima der "User Coordinates",
* die sich bei Transformation und Projektion von Punkten
* ergeben, die in "World Coordinates" gegeben sind
* =====
*
* Es werden in einem eindimensionalen Feld npoin Koordi-
* natentripel in der Reihenfolge x1,y1,z1,x2,y2,z2,...
* erwartet (insgesamt npoin*3 Werte, "World Coordinates").
*
* Es werden die Minima und Maxima der "User Coordinates"
* berechnet, die sich bei Anwendung der aktuellen
* Transformation und der aktuellen Projektion ergeben.
*
* Eingabe: npoin - Anzahl der Punkte
*          xyzw[] - Feld mit npoin*3 Koordinaten
*                  ("World Coordinates")
*
* Ausgabe: xumin,
*          xumax,
*          yumin,
*          yumax - Extremwerte der "User Coordinates"
*
* Return-Wert: 0 ---> Es konnte kein Punkt erfolgreich
*                transformiert werden
*              1 ---> Extremwerte wurden berechnet
*
* Autor: J. Dankert
*****/

int ptmx3_gi (int    npoin , double xyzw [] ,
              double *xumin , double *xumax ,
              double *yumin , double *yumax)

/*****
* ### Graphics Interface ###
*
* Transformation eines in "World Coordinates" gegebenen
* Punktes mit der aktuellen 3D-Transformationsmatrix und
* der aktuellen Projektionsmatrix in das
* Bildkoordinatensystem "User Coordinates"
* =====
*
* Eingabe: xw      - x-Koordinate ("World Coordinates")
*          yw      - y-Koordinate ("World Coordinates")
*          zw      - z-Koordinate ("World Coordinates")
*
* Ausgabe: xu      - Transformierte x-Koordinate
*          yu      - Transformierte y-Koordinate
*
* Return-Wert: 0 --> Projektion misslungen
*              1 --> Projektion geglueckt
*
* Autor: J. Dankert
*****/

int ptw2u_gi (double xw , double yw , double zw ,
              double *xu , double *yu)

```

```

/*****
* ### Graphics Interface ###
*
* Zeichnen einer geraden "breiten und zweifarbigen" Linie
* mit "World Coordinates", die vorher vorher der aktuellen
* 3D-Transformation unterworfen und danach auf die
* Bildebene transformiert werden
* =====
*
* Zwischen zwei mit "World Coordinates" gegebenen Punkten
* wird eine wpix breite Linie mit der Farbe wcol
* gezeichnet, die von einer zweiten (sinnvollerweise nicht
* so breiten) Linie mit der Breite ipix und der Farbe
* icol ueberlagert wird.
*
* Eingabe:  hdc    - "Handle of Device Context"
*           xlw ,
*           ylw ,
*           zlw    - "World Coordinates" des Startpunktes
*           x2w ,
*           y2w ,
*           z2w    - "World Coordinates" des Endpunktes
*           wpix   - Gesamtbreite der Linie
*           wcol   - "Randfarbe" der Linie
*           ipix   - Breite der "Mittellinie"
*           icol   - Farbe der Mittellinie
*
* Return-Wert:  0 ---> Projektion unmoeglich, keine
*                Aktion ausgefuehrt
*                1 ---> Aktion ausgefuehrt
*
* Autor: J. Dankert
*****/

int ptwdl_gi (HDC hdc , double xlw , double ylw , double zlw ,
              double x2w , double y2w , double z2w ,
              int  wpix , COLORREF wcol ,
              int  ipix , COLORREF icol)

```



```

/*****
* ### Graphics Interface ###
*
* Lesen eines Daten-Files mit Beschreibung eines
* 3D-Objektes
* =====
*
* Es wird eine Datei gelesen, die ein 3D-Objekt durch
* "Elemente" und "Knotenkoordinaten" beschreibt:
*
* * In der ersten Zeile muessen 3 int-Werte stehen:
*   "Anzahl der Elemente", "Anzahl der Knoten" und
*   "Anzahl der ein Element beschreibenden int-Werte".
*   Diese Werte werden von rddfl_gi als *ne_p, *nk_p
*   und *ke_p abgeliefert.
*
* * Ab Zeile 2 folgen *nk_p Zeilen mit jeweils einem
*   Koordinaten-Tripel ("World Coordinates" der Knoten),
*   diese werden in einem double-Feld dicht gepackt in
*   der Reihenfolge x1,y1,z1,x2,y2,z2,... abgeliefert
*   (das Feld wird in rddfl_gi dynamisch erzeugt, der
*   Pointer auf das erste Element wird als *xy_pp
*   abgeliefert).
*
* * Danach folgen in jeweils einer Zeile die Element-
*   beschreibungen, das sind jeweils *ke_p int-Werte.
*   Diese werden von rddfl_gi in einer verketteten Liste
*   von GI_ELEM-Strukturen abgeliefert (es wird der
*   Pointer auf das "Anchor-Element" der Liste
*   abgeliefert).
*
* 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
* fuer die Beschreibung dieses Elements folgen.
*
* Eingabe: hwnd    - "Handle of window", das bei Fehlern
*                  das "Parent window" fuer die Ausgabe
*                  einer "Message Box" ist
*            pathnm - Name des zu lesenden Files
*
* Ausgabe: *ne_p   - Anzahl der Elemente
*            *nk_p   - Anzahl der Knoten
*            *ke_p   - Anzahl der int-Werte fuer die
*                  Beschreibung eines Elements
*            *xy_pp  - Pointer auf Koordinatenfeld (Argument
*                  muss Pointer auf double-Pointer sein)
*            *km_pp  - Pointer auf "Anchor-Element" einer
*                  Liste mit GI_ELEM-Strukturen (Argument
*                  muss Pointer auf GI_ELEM-Pointer sein)
*
* Return-Wert:  1 ---> Erfolg
*               0 ---> Misserfolg
*
* Autor: J. Dankert
*****/

int rddfl_gi (HWND hwnd , char *pathnm , int      *ne_p , int      *nk_p ,
             int  *ke_p , double **xy_pp , GI_ELEM **km_pp)

```

```

/*****
* ### Graphics Interface ###
*
* Definieren eines Rechteckbereichs durch Mauspick
* ("User Coordinates")
* =====
*
* Diese Funktion setzt voraus, dass vorher mit dem Aufruf
* von scapp_gi ein spezieller Cursor des Typs
* GI_CUBIGCROSS (1) oder GI_CUSMALLCROSS (2) sichtbar
* gemacht und mit der Funktion scupd_gi.c "verfolgt"
* wurde.
*
* Die Funktion rpick_gi sollte als Reaktion auf die
* Message WM_LBUTTONDOWN oder die Message WM_RBUTTONDOWN
* aufgerufen werden. Sie liefert beim ersten Aufruf
* den gepickten Punkt und den Return-Wert 0 und aendert
* den speziellen Cursor auf den Typ GI_CURECTANGLE (3).
* Beim zweiten Aufruf werden die beiden Punkte des
* ersten und zweiten Aufrufs und der Return-Wert 1
* abgeliefert, und der spezielle Cursor verschwindet.
*
* Eingabe:  hwnd   - "Handle of Window", in dem gearbeitet
*                wird
*           lParam  - LONG-Wert, der die Cursor-Position
*                   als "LOWORD" und "HIWORD" enthaelt,
*                   wie er als Parameter von der Windows-
*                   Message geliefert wird
*
* Ausgabe:  xlu   ,
*           ylu   - Punkt ("User Coordinates"), der beim
*                   ersten Aufruf von rpick_gi erzeugt
*                   wird
*           x2u   ,
*           y2u   - Punkt ("User Coordinates"), der beim
*                   zweiten Aufruf von rpick_gi erzeugt
*                   wird (undefiniert beim ersten
*                   rpick_gi-Aufruf)
*
* Return-Wert:  0 nach dem ersten rpick_gi-Aufruf,
*               1 nach dem zweiten rpick_gi-Aufruf
*
* Autor: J. Dankert
*****/
int rpick_gi (HWND hwnd , LONG lParam , double *xlu , double *ylu ,
              double *x2u , double *y2u)

```

```

/*****
* ### Graphics Interface ###
*
* Zeichnen eines speziellen Cursors im "Current Viewport"
* =====
*
* Im "Current Viewport" erscheint ein spezieller Cursor
* (grosses oder kleines Kreuz), der mit dem Aufruf von
* scupd_gi seine Position aendern und mit dem Aufruf von
* rpick_gi seine Form in ein Rechteck umwandeln kann.
* Mit upick_gi kann ein Punkt, mit rpick_gi kann ein durch
* zwei Punkte definiertes Rechteck gepickt werden (und
* der spezielle Cursor verschwindet wieder). Der spezielle
* Cursor kann auch durch den Aufruf von scdap_gi zum
* Verschwinden gebracht werden.
*
* Eingabe:  hwnd  - "Handle of Window", in dem gearbeitet
*              wird
*           xv   ,
*           yv   - Geraete-Koordinaten, die sich auf das
*                   Viewport-Koordinatensystem beziehen,
*                   Punkt, an dem der Cursor erscheinen
*                   soll
*           ctype - Cursor-Typ, moegliche Varianten:
*                   GI_CUBIGCROSS   (1) -> Kreuz ueber
*                                       ges. Viewport
*                   GI_CUSMALLCROSS (2) -> Kleines Kreuz
*
* Autor: J. Dankert
*****/

```

```
void scapp_gi (HWND hwnd , int xv , int yv , int ctype)
```

```

/*****
* ### Graphics Interface ###
*
* Abschalten eines speziellen Cursors
* =====
*
* Ein mit scapp_gi erzeugter spezieller Cursor kann mit
* scdap_gi gezielt abgeschaltet werden (nach Picken von
* Punkten mit upick_gi oder rpick_gi wird der spezielle
* Cursor automatisch abgeschaltet).
*
* Eingabe:  hwnd  - "Handle of Window", in dem gearbeitet
*              wird
*
* Autor: J. Dankert
*****/

```

```
void scdap_gi (HWND hwnd)
```

```

/*****
* ### Graphics Interface ###
*
* Aktualisieren der Position eines speziellen Cursors
* =====
*
* Diese Funktion setzt voraus, dass vorher mit dem Aufruf
* von scapp_gi ein spezieller Cursor des Typs
* GI_CUBIGCROSS (1) oder GI_CUSMALLCROSS (2) sichtbar
* gemacht und eventuell mit der Funktion rpick_gi auf
* den Typ GI_CURECTANGLE (3) geaendert wurde.
*
* Die Funktion scupd_gi sollte als Reaktion auf die
* Message WM_MOUSEMOVE aufgerufen werden. Sie aktualisiert
* die Position des speziellen Cursors (wenn kein
* spezieller Cursor aktiv ist, fuehrt scupd_gi keine
* Aktion aus).
*
* Eingabe: hwnd - "Handle of Window", in dem gearbeitet
*            wird
*            lParam - LONG-Wert, der die Cursor-Position
*                   als "LOWORD" und "HIWORD" enthaelt,
*                   wie er als Parameter von der Windows-
*                   Message geliefert wird
*
* Autor: J. Dankert
*****/
void scupd_gi (HWND hwnd , LONG lParam)

```

```

/*****
* ### Graphics Interface ###
*
* Festlegen des "Current Viewport"
* =====
*
* Mit dem "Current Viewport" werden der Ursprung eines
* Pixel-Koordinatensystems und ein Clipping-Bereich
* definiert. Mit den Koordinaten (pulx,puly) wird die
* LINKE OBERE ECKE des Viewports in Geraete-Koordinaten
* bezueglich der linken oberen Ecke der "Client Area"
* (mit Achsen, die nach rechts bzw. unten zeigen)
* beschrieben, width und height sind Breite bzw. Hoehe
* des Viewports (ebenfalls in Geraete-Koordinaten).
*
* Die Punkte (pulx,puly) und (pulx+width,puly+height)
* definieren den Clipping-Bereich. Das Koordinatensystem
* liegt in Abhaengigkeit vom Parameter cstype entweder in
* einer Ecke dieses Rechtecks oder in seiner Mitte.
*
* Auf das Viewport-Koordinatensystem beziehen sich alle
* Ausgaben der "v-Routinen" (vmove_gi, vline_gi, ...),
* die Koordinatenangaben in Geraete-Koordinaten erwarten.
*
* Eingabe: hdc      - "Handle of Device Context"
*          pulx ,
*          puly   - Linke obere Ecke des Viewports
*          width ,
*          height - Breite und Hoehe des Viewports
*          cstype = 0 --> Koordinatenursprung in linker
*                        unterer Ecke des Viewports,
*                        x nach rechts, y nach oben
*          = 1 --> Koordinatenursprung in der
*                        Viewportmitte,
*                        x nach rechts, y nach oben
*          = 2 --> Koordinatenursprung in linker
*                        oberer Ecke des Viewports,
*                        x nach rechts, y nach unten
*          = 3 --> Koordinatenursprung in rechter
*                        oberer Ecke des Viewports,
*                        x nach links, y nach unten
*          = 4 --> Koordinatenursprung in rechter
*                        unterer Ecke des Viewports,
*                        x nach links, y nach oben
*
* Fuer den Parameter cstype ist die Verwendung folgender
* (in giw.h definierter) Konstanten moeglich:
*
*          GI_XYBOTTOMLEFT   = 0
*          GI_XYCENTER       = 1
*          GI_XYTOPLEFT      = 2
*          GI_XYTOPRIGHT     = 3
*          GI_XYBOTTOMRIGHT  = 4
*
* Autor: J. Dankert
*****/

void stcvp_gi (HDC hdc , int pulx , int puly ,
              int width , int height , int cstype)

```

```

/*****
* ### Graphics Interface ###
*
* Definieren der "User Coordinates" (anisotrop)
* =====
*
* Fuer den "Current Viewport" werden "User Coordinates"
* durch Angabe der Koordinatenwerte fuer die beiden
* Punkte P1 und P2 festgelegt. Diese definieren ein Recht-
* eck, in dem sie sich diagonal als linker unterer (P1)
* bzw. rechter oberer Eckpunkt (P2) gegenueberliegen.
*
* Die Differenz der beiden x-Koordinaten wird auf die
* gesamte Breite des Viewports, die Differenz der y-Koor-
* dinaten auf die gesamte Hoehe des Viewports abgebildet
* (wenn fuer das Argument pmarg der Wert 0. angegeben
* wird), so dass in der Regel unterschiedliche Skalie-
* rungen fuer die beiden Richtungen gelten (anisotrope
* Skalierung).
*
* Der Richtungssinn der Achsen stellt sich automatisch
* so ein, dass sie von kleineren Koordinatenwerten zu
* groesseren zeigen. Es sind also alle vier Kombinationen
* realisierbar.
*
* Mit dem Parameter pmarg kann ein "Rand" vorgesehen
* werden: Die Distanz der "User Coordinates" der Punkte
* P1 und P2 wird fuer die kleinere Viewport-Abmessung auf
* jeder Seite um pmarg Prozent vergroessert, so dass P1
* und P2 nicht mehr in den Viewport-Ecken liegen. Fuer die
* groessere Viewport-Abmessung wird ein Rand gleicher
* Breite eingestellt. Man beachte, dass in diese Raender
* nur dann nicht heineingezeichnet wird, wenn die
* Graphik-Ausgabe innerhalb des durch P1 und P2 definier-
* ten Rechtecks bleibt, weil das "Clipping" ausschliess-
* lich durch die Viewport-Grenzen gesteuert wird.
*
* Eingabe:  plxu  -  Koordinate des Punktes P1 in
*                horizontaler Richtung
*            plyu  -  Koordinate des Punktes P1 in
*                vertikaler Richtung
*            p2xu  -  Koordinate des Punktes P2 in
*                horizontaler Richtung
*            p2yu  -  Koordinate des Punktes P2 in
*                vertikaler Richtung
*            pmarg -  Festlegung der Randbreite (Angabe
*                in Prozent)
*
* Autor: J. Dankert
*****/
void stuca_gi (double plxu , double plyu ,
               double p2xu , double p2yu , double pmarg)

```

```

/*****
* ### Graphics Interface ###
*
* Definieren der "User Coordinates" (gleiche Skalierung
* in beiden Richtungen)
* =====
*
* Fuer den "Current Viewport" werden "User Coordinates"
* durch Angabe der Koordinatenwerte fuer die beiden
* Punkte P1 und P2 festgelegt. Diese definieren ein Recht-
* eck, in dem sie sich diagonal als linker unterer (P1)
* bzw. rechter oberer Eckpunkt (P2) gegenueberliegen.
*
* Dieses Rechteck wird so in den "Current Viewport"
* eingepasst, dass sich in beiden Richtungen gleiche
* Skalierungen ergeben. In einer Richtung wird die
* Viewport-Abmessung voll ausgenutzt (P1 und P2 liegen
* auf den Viewport-Grenzen), in der anderen Richtung
* sind P1 und P2 jeweils gleich weit von den Viewport-
* Raendern entfernt (isotrope Skalierung).
*
* Der Richtungssinn der Achsen stellt sich automatisch
* so ein, dass sie von kleineren Koordinatenwerten zu
* groesseren zeigen. Es sind also alle vier Kombinationen
* realisierbar.
*
* Mit dem Parameter pmarg kann ein "Rand" vorgesehen
* werden: Die Distanz der "User Coordinates" der Punkte
* P1 und P2 wird fuer die "unguenstigere" Richtung auf
* jeder Seite um pmarg Prozent vergroessert, so dass P1
* und P2 nicht mehr auf den Viewport-Raendern liegen. Fuer
* die andere Viewport-Abmessung werden die Raender so
* ausgelegt, dass bei gleicher Skalierung der Koordinaten
* P1 und P2 jeweils in gleicher Entfernung vom Viewport-
* Rand liegen. Man beachte, dass in diese Raender
* nur dann nicht heineingezeichnet wird, wenn die
* Graphik-Ausgabe innerhalb des durch P1 und P2 definier-
* ten Rechtecks bleibt, weil das "Clipping" ausschliess-
* lich durch die Viewport-Grenzen gesteuert wird.
*
* Eingabe:  plxu  -  Koordinate des Punktes P1 in
*                horizontaler Richtung
*            plyu  -  Koordinate des Punktes P1 in
*                vertikaler Richtung
*            p2xu  -  Koordinate des Punktes P2 in
*                horizontaler Richtung
*            p2yu  -  Koordinate des Punktes P2 in
*                vertikaler Richtung
*            pmarg -  Festlegung der Randbreite (Angabe
*                in Prozent)
*
* Autor: J. Dankert
*****/

void stuci_gi (double plxu , double plyu ,
               double p2xu , double p2yu , double pmarg)

```

```

/*****
* ### Graphics Interface ###
*
* Abliefern der gueltigen 3D-Transformationsmatrix
* =====
*
* Als Argument muss ein Array mit mindestens 16 double-
* Elementen uebergeben werden, auf dem die aktuelle
* 3D-Transformationsmatrix (zeilenweise gespeichert)
* abgeliefert wird.
*
* Eingabe: Keine
*
* Ausgabe: tm3 - 16 Elemente der gueltigen 3D-Transfor-
*           mationsmatrix
*
* Autor: J. Dankert
*****/

void t3gtm_gi (double *tm)

/*****
* ### Graphics Interface ###
*
* Multiplikation der 3D-Transformationsmatrix mit
* einer 4*4-Matrix (von links)
* =====
*
* Mit dieser Funktion kann eine beliebige zusaetzliche
* 3D-Transformation eingefuegt werden. Sie kann z. B.
* zur schnelleren Erledigung einer Transformation
* benutzt werden, die sich nur sehr aufwendig aus den
* Elementar-Transformationen zusammensetzen laesst.
*
* Eingabe: tnew - Matrix der zusaetzlichen Transfor-
*           mation (bezieht sich auf
*           HOMOGENE Koordinaten!).
*
* Autor: J. Dankert
*****/

void t3mam_gi (double *tnew)

```



```

/*****
* ### Graphics Interface ###
*
* Rotation: Aenderung der aktuellen
* 3D-Transformationsmatrix
* =====
*
* Es wird eine zusaetzliche Rotation um die Koordinaten-
* achse axis in die Transformationsmatrix hineingenommen.
*
* Eingabe:  phi -   Winkel (positiv nach "Rechte-Hand-
*                Regel": Daumen in Richtung der
*                positiven Koordinatenachse, dann
*                zeigen die Finger die Richtung
*                positiver Winkel an), Bogenmaass!!
*                axis - Koordinatenachse, um die die
*                Drehung erfolgt, akzeptiert werden
*                GI_AXISX, GI_AXISY und GI_AXISZ
*
* Autor: J. Dankert
*****/

void t3rot_gi (double phi , int axis)

/*****
* ### Graphics Interface ###
*
* Setzen der 3D-Transformationsmatrix
* =====
*
* Als Argument wird ein Array mit den 16 (zeilenweise
* gespeicherten) Elementen der 3D-Transformationsmatrix
* erwartet (fuer die Transformation mit homogenen
* Koordinaten), so wie diese Matrix z. B. von t3gtm_gi
* abgeliefert wird.
*
* Eingabe:  tm - 16 Elemente der neuen 3D-Transformations-
*                matrix
*
* Ausgabe:  Keine
*
* Autor: J. Dankert
*****/

void t3stm_gi (double *tm)

/*****
* ### Graphics Interface ###
*
* Translation: Aenderung der aktuellen
* Transformationsmatrix
* =====
*
* Es wird eine zusaetzliche Translation um die Strecken
* tx, ty, tz in die Transformationsmatrix hineingenommen.
*
* Eingabe:  tx -   Translation in x-Richtung
*            ty -   Translation in y-Richtung
*            tz -   Translation in z-Richtung
*
* Autor: J. Dankert
*****/

void t3trn_gi (double tx , double ty , double tz)

```

```

/*****
* ### Graphics Interface ###
*
* Transformation eines in "World Coordinates" gegebenen
* Punktes mit der aktuellen 3D-Transformationsmatrix
* =====
*
* Eingabe:  xw,
*           yw,
*           zw  - "World Coordinates" des umzurechnenden
*                Punktes
*
* Ausgabe:  txu,
*           tyu,
*           tzu  - Transformierte Koordinaten
*
* Autor: J. Dankert
*****/

void t3w2w_gi (double xw , double yw , double zw ,
               double *txw , double *tyw , double *tzw)

/*****
* ### Graphics Interface ###
*
* Zeichnen eines Kreisbogens ("User Coordinates")
* der vorher der aktuellen Transformation unterworfen wird
* =====
*
* Ein Kreisbogen wird definiert durch
*
* * die Koordinaten xc,yc des Kreis-Mittelpunktes,
*
* * den Radius r,
*
* * einen Startpunkt, der sich als Schnittpunkt des
*   Kreises mit einer Geraden ergibt, die durch den
*   Mittelpunkt des Kreises und den Punkt xs,ys geht.
*   Vom Startpunkt wird der Bogen entgegen dem
*   Uhrzeigersinn gezeichnet bis zu ...
*
* * einem Endpunkt, der sich als Schnittpunkt des
*   Kreises mit einer Geraden ergibt, die durch den
*   Mittelpunkt des Kreises und den Punkt xe,ye geht.
*
* Eingabe:  hdc  - "Handle of Device Context"
*           xc ,
*           yc  - Mittelpunkt ("User Coordinates") des
*                Kreises
*           r   - Radius
*           xs ,
*           ys  - Punkt ("User Coordinates"), mit dem
*                die Gerade definiert wird, die den
*                Startpunkt des Bogens festlegt
*           xe ,
*           ye  - Punkt ("User Coordinates"), mit dem
*                die Gerade definiert wird, die den
*                Endpunkt des Bogens festlegt
*
* Autor: J. Dankert
*****/

void tdarc_gi (HDC hdc , double xc , double yc , double r ,
               double xs , double ys , double xe , double ye)

```

```

/*****
* ### Graphics Interface ###
*
* Zeichnen eines "gefüllten Kreises" ("User Coordinates")
* der vorher der aktuellen Transformation unterworfen wird
* =====
*
* Ein Kreisbogen wird definiert durch die Koordinaten
* xc,yc des Mittelpunktes und den Radius r. Die
* Mittelpunkt-Koordinaten werden vor dem Zeichnen der
* aktuellen Transformation unterworfen, der Radius wird
* nur skaliert.
*
* Der Rand des Kreises wird mit der "Farbe des aktuellen
* Zeichenstiftes" gezeichnet, ausgefüllt wird er
* mit der Farbe des aktuellen "Brushs".
*
* Eingabe:  hdc  - "Handle of Device Context"
*           xc   , - Mittelpunkt ("User Coordinates") des
*           yc   - Kreises
*           r    - Radius
*
* Ausgabe:  Keine
*
* Autor: J. Dankert
*****/

```

```
void tfcir_gi (HDC hdc , double xc , double yc , double r)
```

```

/*****
* ### Graphics Interface ###
*
* Abliefern der gueltigen 2D-Transformationsmatrix
* =====
*
* Als Argument muss ein Array mit mindestens 9 double-
* Elementen uebergeben werden, auf dem die aktuelle
* 2D-Transformationsmatrix (zeilenweise gespeichert)
* abgeliefert wird.
*
* Eingabe:  Keine
*
* Ausgabe:  tm - 9 Elemente der gueltigen 2D-Transfor-
*           mationsmatrix
*
* Autor: J. Dankert
*****/

```

```
void tgttm_gi (double *tm)
```

```

/*****
* ### Graphics Interface ###
*
* Initialisieren der Transformations-Parameter
* =====
*
* Es werden alle Transformations-Parameter (2D und 3D)
* initialisiert. Als Transformationsmatrizen werden
* Einheitsmatrizen eingetragen, so dass sich alle
* "zeichnenden t...-Funktionen und pt...-Funktionen"
* nach einem tinit_gi-Aufruf wie die Funktionen verhalten,
* die keine Transformation auswerten.
*
* Eingabe: Keine
*
* Autor: J. Dankert
*****/

void tinit_gi ()

/*****
* ### Graphics Interface ###
*
* Zeichnen einer geraden Linie von der "Current Position"
* zu einem in "User Coordinates" gegebenen Punkt,
* der vorher der aktuellen Transformation unterworfen wird
* =====
*
* Die Funktion tline_gi startet an der "Current Position"
* des imaginaeren Zeichenstiftes, die von einer vorange-
* gangenen Zeichenaktion (z. B. vom vorherigen Aufruf von
* tline_gi) oder von einer "Move"-Routine (wie z. B.
* vmove_gi oder tmove_gi) hinterlassen wurde.
*
* Hinweis: Wenn beim Aufruf von tline_gi keine "Current
* Position" existiert, wird keine Zeichenaktion
* ausgefuehrt, der eigentliche Zielpunkt wird
* aber als "Current Position" fuer nachfolgende
* Aktionen registriert.
*
* Eingabe: hdc - "Handle of Device Context"
*          xu ,
*          yu - Koordinaten, die sich auf das mit
*              stusa_gi oder stuci_gi definierte
*              User-Koordinatensystem beziehen
*
* Ausgabe: Keine
*
* Autor: J. Dankert
*****/

void tline_gi (HDC hdc , double xu , double yu)

```

```

/*****
 * ### Graphics Interface ###
 *
 * Multiplikation der ebenen Transformationsmatrix mit
 * einer 3*3-Matrix (von links)
 * =====
 *
 * Mit dieser Funktion kann eine beliebige zusaetzliche
 * Ebene Transformation eingefuegt werden. Sie kann z. B.
 * zur schnelleren Erledigung einer Transformation
 * benutzt werden, die sich nur sehr aufwendig aus den
 * Elementar-Transformationen zusammensetzen laesst.
 *
 * Eingabe:  tnew   - Matrix der zusaetzlichen Transfor-
 *                  mation (bezieht sich auf Ebene
 *                  HOMOGENE Koordinaten!).
 *
 * Autor: J. Dankert
 *****/

```

```
void tmamu_gi (double *tnew)
```

```

/*****
 * ### Graphics Interface ###
 *
 * Spiegelung an der x-Achse: Aenderung der aktuellen
 * Transformationsmatrix
 * =====
 *
 * Es wird eine zusaetzliche Spiegelung an der x-Achse
 * in die Transformationsmatrix hineingenommen.
 *
 * Eingabe:  Keine
 *
 * Autor: J. Dankert
 *****/

```

```
void tmirx_gi ()
```

```

/*****
 * ### Graphics Interface ###
 *
 * Spiegelung an der y-Achse: Aenderung der aktuellen
 * Transformationsmatrix
 * =====
 *
 * Es wird eine zusaetzliche Spiegelung an der y-Achse
 * in die Transformationsmatrix hineingenommen.
 *
 * Eingabe:  Keine
 *
 * Autor: J. Dankert
 *****/

```

```
void tmiry_gi ()
```

```

/*****
* ### Graphics Interface ###
*
* Bewegen des "Zeichenstiftes" zu einem in
* "User Coordinates" gegebenen Punkt, der vorher
* der aktuellen Transformation unterworfen wird
* =====
*
* Die Funktion tmove_gi fuehrt keine Zeichenaktion aus,
* es wird nur die "Current Position" des imaginaeren
* Zeichenstiftes geaendert.
*
* Hinweis: Die "Move"-Funktionen im GIW fuehren keine
* Aktion mit dem Windows-GDI aus, die "Current
* Position" wird nur im GIW gespeichert
* und fuer eine nachfolgende Zeichenaktion,
* die eine "Current Postion" erfordert,
* verwendet.
*
* Eingabe:  hdc   - "Handle of Device Context"
*           xu    ,
*           yu    - Koordinaten, die sich auf das mit
*                   stuca_gi oder stuci_gi definierte
*                   User-Koordinatensystem beziehen
*
* Autor: J. Dankert
*****/

```

```
void tmove_gi (HDC hdc , double xu , double yu)
```

```

/*****
* ### Graphics Interface ###
*
* Rotation: Aenderung der aktuellen
* Transformationsmatrix
* =====
*
* Es wird eine zusaetzliche Rotation um den Punkt xm,ym
* in die Transformationsmatrix hineingenommen. Der
* Punkt xm,ym wird vorher der noch gueltigen
* Transformation unterworfen.
*
* Eingabe:  xm    - x-Koordinate des Rotations-
*                 Mittelpunktes
*           ym    - y-Koordinate des Rotations-
*                 Mittelpunktes
*           phi    - Winkel (positiv entgegen dem
*                 Uhrzeigersinn), Bogenmass!
*
* Autor: J. Dankert
*****/

```

```
void trota_gi (double xm , double ym , double phi)
```

```

/*****
* ### Graphics Interface ###
*
* Transformation eines in "User Coordinates" gegebenen
* Punktes mit der aktuellen Transformationsmatrix
* =====
*
* Eingabe:  xu,
*           yu  -  "User Coordinates" des umzurechnenden
*                 Punktes
*
* Ausgabe:  txu,
*           tyu  -  Transformierte Koordinaten
*
* Autor: J. Dankert
*****/

void tru2t_gi (double xu , double yu , double *txu , double *tyu)

/*****
* ### Graphics Interface ###
*
* Setzen der 2D-Transformationsmatrix
* =====
*
* Als Argument wird ein Array mit den 9 (zeilenweise
* gespeicherten) Elementen der 2D-Transformationsmatrix
* erwartet (fuer die Transformation mit homogenen
* Koordinaten), so wie diese Matrix z. B. von tgttm_gi
* abgeliefert wird.
*
* Eingabe:  tm - 9 Elemente der neuen 2D-Transformations-
*               matrix
*
* Ausgabe:  Keine
*
* Autor: J. Dankert
*****/

void tsttm_gi (double *tm)

/*****
* ### Graphics Interface ###
*
* Transformation (absolut): Skalierung bezueglich des
* Nullpunktes mit sx,sy, Drehung um den Nullpunkt um den
* Winkel phi, anschliessend Translation um tx,ty
* =====
*
* Alle vorab vorgenommenen Transformationen bleiben
* unberuecksichtigt.
*
* Eingabe:  tx  -  Translation in x-Richtung
*           ty  -  Translation in y-Richtung
*           phi -  Winkel (positiv entgegen dem
*                 Uhrzeigersinn), Bogenmass!
*           sx  -  Skalierung in x-Richtung
*           sy  -  Skalierung in y-Richtung
*
* Ausgabe:  Keine
*
* Autor: J. Dankert
*****/

void ttabs_gi (double tx , double ty , double phi ,
               double sx , double sy)

```

```

/*****
* ### Graphics Interface ###
*
* Translation: Aenderung der aktuellen
* Transformationsmatrix
* =====
*
* Es wird eine zusaetzliche Translation um die Strecken
* tx und ty in die Transformationsmatrix hineingenommen.
*
* Eingabe: tx - Translation in x-Richtung
*          ty - Translation in y-Richtung
*
* Autor: J. Dankert
*****/

void ttran_gi (double tx , double ty)

```

```

/*****
* ### Graphics Interface ###
*
* Zeichnen eines Kreisbogens ("User Coordinates")
* =====
*
* Ein Kreisbogen wird definiert durch
*
* * die Koordinaten xc,yc des Kreis-Mittelpunktes,
*
* * den Radius r,
*
* * einen Startpunkt, der sich als Schnittpunkt des
*   Kreises mit einer Geraden ergibt, die durch den
*   Mittelpunkt des Kreises und den Punkt xs,ys geht.
*   Vom Startpunkt wird der Bogen entgegen dem
*   Uhrzeigersinn gezeichnet bis zu ...
*
* * einem Endpunkt, der sich als Schnittpunkt des
*   Kreises mit einer Geraden ergibt, die durch den
*   Mittelpunkt des Kreises und den Punkt xe,ye geht.
*
* Eingabe: hdc - "Handle of Device Context"
*          xc ,
*          yc - Mittelpunkt ("User Coordinates") des
*             Kreises
*          r - Radius
*          xs ,
*          ys - Punkt ("User Coordinates"), mit dem
*             die Gerade definiert wird, die den
*             Startpunkt des Bogens festlegt
*          xe ,
*          ye - Punkt ("User Coordinates"), mit dem
*             die Gerade definiert wird, die den
*             Endpunkt des Bogens festlegt
*
* Autor: J. Dankert
*****/

void udarc_gi (HDC hdc , double xc , double yc , double r ,
              double xs , double ys , double xe , double ye)

```



```

/*****
* ### Graphics Interface ###
*
* Zeichnen eines gefuellten Kreises, dessen Abmessungen
* in "User Coordinates" gegeben sind
* =====
*
* Der Rand des Kreises wird mit der "Farbe des aktuellen
* Zeichenstiftes" gezeichnet, ausgefullt wird er
* mit der Farbe des aktuellen "Brushs".
*
* Ein Kreis wird definiert durch
*
* * die Koordinaten xc,yc des Mittelpunktes,
*
* * den Radius r.
*
* Eingabe:  hdc  - "Handle of Device Context"
*           xc,
*           yc,  - Mittelpunkt ("User Coordinates")
*           r    - Radius
*
* Autor: J. Dankert
*****/
void ufcir_gi (HDC hdc , double xc , double yc , double r)

```

```

/*****
* ### Graphics Interface ###
*
* Zeichnen einer gefuellten Ellipse, deren Abmessungen
* in "User Coordinates" gegeben sind (vgl. auch
* Funktion ufell_gi)
* =====
*
* Der Rand der Ellipse wird mit der "Farbe des aktuellen
* Zeichenstiftes" gezeichnet, ausgefullt wird sie
* mit der Farbe des aktuellen "Brushs".
*
* Eingabe:  hdc  - "Handle of Device Context"
*           xc,
*           yc  - Koordinaten des Mittelpunkts der
*               Ellipse ("User Coordinates")
*           a   - Halbachse in x-Richtung
*           b   - Halbachse in y-Richtung
*
* Autor: J. Dankert
*****/
void ufel2_gi (HDC hdc , double xc , double yc ,
              double a , double b)

```

```

/*****
* ### Graphics Interface ###
*
* Zeichnen einer gefuellten Ellipse, deren umschliessendes
* Rechteck in "User Coordinates" gegeben ist (vgl. auch
* Funktion ufel2_gi)
* =====
*
* Der Rand der Ellipse wird mit der "Farbe des aktuellen
* Zeichenstiftes" gezeichnet, ausgefüllt wird sie
* mit der Farbe des aktuellen "Brushs".
*
* Eingabe:  hdc  -  "Handle of Device Context"
*           xul,
*           yul,
*           xu2,
*           yu2  -  User-Koordinaten, die sich auf das
*                   mit stuci_gi oder stuca_gi definierte
*                   Koordinatensystem beziehen und zwei
*                   diagonal entgegengesetzte Punkte des
*                   umschliessenden Rechtecks beschreiben
*
* Autor: J. Dankert
*****/

void ufell_gi (HDC hdc , double xul , double yul ,
              double xu2 , double yu2)

```

```

/*****
* ### Graphics Interface ###
*
* Zeichnen eines elliptischen Sektors ("User Coordinates")
* (siehe auch Funktion ufpi2_gi)
* =====
*
* Ein elliptischer Sektor wird definiert durch
*
* * den Mittelpunkt xc,yc der kompletten Ellipse,
*
* * die beiden Halbachsen a und b der kompletten
  Ellipse,
*
* * einen Startpunkt, der sich als Schnittpunkt der
  Ellipse mit einer Geraden ergibt, die durch den
  Mittelpunkt der Ellipse und den Punkt xs,ys geht.
  Vom Startpunkt wird der Bogen entgegen dem
  Uhrzeigersinn gezeichnet bis zu ...
*
* * einem Endpunkt, der sich als Schnittpunkt der
  Ellipse mit einer Geraden ergibt, die durch den
  Mittelpunkt der Ellipse und den Punkt xe,ye geht,
*
* * die beiden Radien zum Startpunkt bzw. Endpunkt,
  die durch die vier Punkte eindeutig bestimmt sind.
*
* Eingabe:  hdc  - "Handle of Device Context"
*           xc   ,
*           yc   - Mittelpunkt ("User Coordinates")
*                  der Ellipse
*           a    - Halbachse in x-Richtung
*           b    - Halbachse in y-Richtung
*           xs   ,
*           ys   - Punkt ("User Coordinates"), mit dem
*                  die Gerade definiert wird, die den
*                  Startpunkt des Bogens festlegt
*           xe   ,
*           ye   - Punkt ("User Coordinates"), mit dem
*                  die Gerade definiert wird, die den
*                  Endpunkt des Bogens festlegt
*
* Autor: J. Dankert
*****/

void ufpi2_gi (HDC hdc , double xc , double yc , double a , double b ,
              double xs , double ys , double xe , double ye)

```

```

/*****
* ### Graphics Interface ###
*
* Zeichnen eines elliptischen Sektors ("User Coordinates")
* (siehe auch Funktion ufpi2_gi)
* =====
*
* Ein elliptischer Sektor wird definiert durch
*
* * zwei Punkte, gegeben durch die beiden Koordinaten-
*   paare x1,y1 und x2,y2, die ein Rechteck definieren
*   (liegen in diesem Rechteck auf einer Diagonalen),
*   das fuer die komplette Ellipse das umschliessende
*   Rechteck waere,
*
* * einen Startpunkt, der sich als Schnittpunkt der
*   Ellipse mit einer Geraden ergibt, die durch den
*   Mittelpunkt der Ellipse und den Punkt xs,ys geht.
*   Vom Startpunkt wird der Bogen entgegen dem
*   Uhrzeigersinn gezeichnet bis zu ...
*
* * einem Endpunkt, der sich als Schnittpunkt der
*   Ellipse mit einer Geraden ergibt, die durch den
*   Mittelpunkt der Ellipse und den Punkt xe,ye geht,
*
* * die beiden Radien zum Startpunkt bzw. Endpunkt,
*   die durch die vier Punkte eindeutig bestimmt sind.
*
* Eingabe:  hdc  -  "Handle of Device Context"
*           x1  ,
*           y1  ,
*           x2  ,
*           y2  -  Punkte ("User Coordinates"), die das
*                   umschliessende Rechteck der Ellipse
*                   definieren
*           xs  ,
*           ys  -  Punkt ("User Coordinates"), mit dem
*                   die Gerade definiert wird, die den
*                   Startpunkt des Bogens festlegt
*           xe  ,
*           ye  -  Punkt ("User Coordinates"), mit dem
*                   die Gerade definiert wird, die den
*                   Endpunkt des Bogens festlegt
*
* Autor: J. Dankert
*****/

void ufpie_gi (HDC hdc , double x1 , double y1 , double x2 , double y2 ,
              double xs , double ys , double xe , double ye)

```

```

/*****
* ### Graphics Interface ###
*
* Zeichnen eines gefuellten Polygons, dessen Eckpunkte in
* "User Coordinates" gegeben sind
* =====
*
* Der Rand des Polygons wird mit der Farbe des aktuellen
* Zeichenstiftes" gezeichnet, ausgefullt wird das
* Polygon mit der Farbe des aktuellen "Brushs".
*
* Das Polygon wird automatisch geschlossen, es wird ein
* "gefuelltes npoin-Eck" gezeichnet.
*
* Eingabe: hdc      - "Handle of Device Context"
*          npoin    - Anzahl der Punkte
*          xu []    - Feld mit npoin xu-Koordinaten
*          yu []    - Feld mit npoin yu-Koordinaten
*
* Return-Wert:  1 ---> Erfolg, Polygon wurde gezeichnet
*               0 ---> Misserfolg, keine Zeichenaktion
*                  (z. B., wenn der erforderliche
*                  dynamisch angeforderte Speicher-
*                  platz nicht verfuegbar war)
*
* Autor: J. Dankert
*****/
int ufpol_gi (HDC hdc , int npoin , double xu [] , double yu [])

```

```

/*****
* ### Graphics Interface ###
*
* Zeichnen eines gefuellten Rechtecks, dessen Eckpunkte in
* "User Coordinates" gegeben sind
* =====
*
* Der Rand des Rechtecks wird mit der "Farbe des aktuellen
* Zeichenstiftes" gezeichnet, ausgefullt wird
* es mit der Farbe des aktuellen "Brushs".
*
* Eingabe:  hdc  - "Handle of Device Context"
*           xul,
*           yul,
*           xu2,
*           yu2  - User-Koordinaten, die sich auf das
*                  mit stuci_gi oder stuca_gi definierte
*                  Koordinatensystem beziehen und zwei
*                  diagonal entgegengesetzte Punkte des
*                  Rechtecks beschreiben
*
* Ausgabe: Keine
*
* Autor: J. Dankert
*****/
void ufrec_gi (HDC hdc , double xul , double yul ,
              double xu2 , double yu2)

```

```

/*****
* ### Graphics Interface ###
*
* Zeichnen eines "gefüllten" Kreissektors
* ("User Coordinates")
* =====
*
* Ein Kreissektor wird definiert durch
*
* * die Koordinaten xc,yc des Kreis-Mittelpunktes,
*
* * den Radius r,
*
* * einen Startpunkt, der sich als Schnittpunkt des
*   Kreises mit einer Geraden ergibt, die durch den
*   Mittelpunkt des Kreises und den Punkt xs,ys geht.
*   Vom Startpunkt wird der Bogen entgegen dem
*   Uhrzeigersinn gezeichnet bis zu ...
*
* * einem Endpunkt, der sich als Schnittpunkt des
*   Kreises mit einer Geraden ergibt, die durch den
*   Mittelpunkt des Kreises und den Punkt xe,ye geht.
*
* Eingabe:  hdc  -  "Handle of Device Context"
*           xc   ,
*           yc   -  Mittelpunkt ("User Coordinates") des
*                   Kreises
*           r    -  Radius
*           xs   ,
*           ys   -  Punkt ("User Coordinates"), mit dem
*                   die Gerade definiert wird, die den
*                   Startpunkt des Bogens festlegt
*           xe   ,
*           ye   -  Punkt ("User Coordinates"), mit dem
*                   die Gerade definiert wird, die den
*                   Endpunkt des Bogens festlegt
*
* Autor: J. Dankert
*****/

void ufsec_gi (HDC hdc , double xc , double yc , double r ,
              double xs , double ys , double xe , double ye)

```

```

/*****
* ### Graphics Interface ###
*
* Zeichnen einer geraden Linie von der "Current Position"
* zu einem in "User Coordinates" gegebenen Punkt
* =====
*
* Die Funktion uline_gi startet an der "Current Position"
* des imaginaeren Zeichenstiftes, die von einer vorange-
* gangenen Zeichenaktion (z. B. vom vorherigen Aufruf von
* uline_gi) oder von einer "Move"-Routine (wie z. B.
* vmove_gi oder umove_gi) hinterlassen wurde.
*
* Hinweis: Wenn beim Aufruf von uline_gi keine "Current
* Position" existiert, wird keine Zeichenaktion
* ausgefuehrt, der eigentliche Zielpunkt wird
* aber als "Current Position" fuer nachfolgende
* Aktionen registriert.
*
* Eingabe:  hdc  - "Handle of Device Context"
*           xu   ,
*           yu   - Koordinaten, die sich auf das mit
*                   stuca_gi oder stuci_gi definierte
*                   User-Koordinatensystem beziehen
*
* Autor: J. Dankert
*****/
void uline_gi (HDC hdc , double xu , double yu)

```

```

/*****
* ### Graphics Interface ###
*
* Zeichnen eines "Markers" an einem in
* "User Coordinates" gegebenen Punkt
* =====
*
* Es wird ein vordefiniertes Symbol (mtype) gezeichnet.
*
* Eine Aenderung der Fenstergroesse beeinflusst die
* Groesse der Marker nicht.
*
* Der Marker wird mit der Farbe des aktuellen "Zeichen-
* stiftes" gezeichnet, bei den Markertypen
* GI_MKFCIRCLE (6) und GI_MKFSQUARE (7) wird nur der
* Rand mit dieser Farbe gezeichnet, ausgefuellt werden
* diese beiden Marker mit der Farbe des aktuellen
* "Brushes".
*
* Die Funktion umark_gi aendert die "Current Position",
* in Abhaengigkeit vom Parameter offset liegt sie im
* Zentrum des Markers (sinnvoll, wenn von dort aus die
* folgende Linie startet) oder auf dem Rand des den Marker
* umschliessenden Quadrats (sinnvoll, wenn der Marker
* anschliessend beschriftet werden soll).
*
* Eingabe: hdc      - "Handle of Device Context"
*           mtype    - Markertyp:
*
*           = GI_MKFCIRCLE (1) --> Kreis
*           = GI_MKSQUARE (2) --> Quadrat
*           = GI_MKCROSS (3) --> Kreuz
*           = GI_MKVBAR (4) --> Vertikale Linie
*           = GI_MKHBAR (5) --> Horizont. Linie
*           = GI_MKFCIRCLE (6) --> Gefuellt. Kreis
*           = GI_MKFSQUARE (7) --> Gef. Quadrat
*
*           msize    - Faktor zur Steuerung der Markergroesse
*           xu,      - Koordinaten des Marker-Mittelpunkts
*           yu      - "Current position" NACH dem Zeichnen
*           offset   - des Markers:
*
*           = GI_UPLEFT (1) --> links oben
*           = GI_LEFT (2) --> links
*           = GI_DOWNLEFT (3) --> links unten
*           = GI_UP (4) --> oben
*           = GI_CENTER (5) --> Makermittle
*           (auch bei offset=0)
*           = GI_DOWN (6) --> unten
*           = GI_UPRIGHT (7) --> rechts oben
*           = GI_RIGHT (8) --> rechts
*           = GI_DOWNRIGHT (9) --> rechts unten
*
* Autor: J. Dankert
*****/

void umark_gi (HDC hdc , int mtype , double msize ,
              double xu , double yu , int offset)

```



```

/*****
* ### Graphics Interface ###
*
* Bewegen des "Zeichenstiftes" zu einem in
* "User Coordinates" gegebenen Punkt
* =====
*
* Die Funktion umove_gi fuehrt keine Zeichenaktion aus,
* es wird nur die "Current Position" des imaginaeren
* Zeichenstiftes geaendert.
*
* Hinweis: Die "Move"-Funktionen im GIW fuehren keine
* Aktion mit dem Windows-GDI aus, die "Current
* Position" wird nur im GIW gespeichert
* und fuer eine nachfolgende Zeichenaktion,
* die eine "Current Postion" erfordert,
* verwendet.
*
* Eingabe: hdc - "Handle of Device Context"
*          xu ,
*          yu - Koordinaten, die sich auf das mit
*              stuca_gi oder stuci_gi definierte
*              User-Koordinatensystem beziehen
*
* Autor: J. Dankert
*****/

```

```
void umove_gi (HDC hdc , double xu , double yu)
```

```

/*****
* ### Graphics Interface ###
*
* Punkteingabe ("User Coordinates") durch Mausepick
* =====
*
* Die Funktion upick_gi kann als Reaktion auf die
* Message WM_LBUTTONDOWN oder die Message WM_RBUTTONDOWN
* aufgerufen werden. Sie liefert die "User Coordinates"
* des gepickten Punktes. Sie kann jedoch auch nur zur
* Berechnung der "User Coordinates" aus einer in einem
* Doppelwort verschluesselten Mausposition genutzt werden.
*
* Wenn vor dem Aufruf von upick_gi mit dem Aufruf
* von scapp_gi ein spezieller Cursor des Typs
* GI_CUBIGCROSS (1) oder GI_CUSMALLCROSS (2) sichtbar
* gemacht und mit der Funktion scupd_gi "verfolgt"
* wurde, wird dieser in upick_gi abgeschaltet.
*
* Eingabe: hwnd - "Handle of Window", in dem gearbeitet
*           wird
*           lParam - LONG-Wert, der die Cursor-Position
*                   als "LOWORD" und "HIWORD" enthaelt,
*                   wie er als Parameter von der Windows-
*                   Message geliefert wird
*
* Ausgabe: xu ,
*           yu - Punkt ("User Coordinates")
*
* Return-Wert = 0 ---> Punkt liegt ausserhalb des
*                  "Current Viewport"
*               = 1 ---> Punkt liegt innerhalb des
*                  "Current Viewport"
*
* Autor: J. Dankert
*****/

```

```
int upick_gi (HWND hwnd , LONG lParam , double *xu , double *yu)
```

```

/*****
* ### Graphics Interface ###
*
* Zeichnen einer geraden "breiten und zweifarbigen" Linie
* mit "User Coordinates"
* =====
*
* Zwischen zwei mit "User Coordinates" gegebenen Punkten
* wird eine wpix breite Linie mit der Farbe wcol
* gezeichnet, die von einer zweiten (sinnvollerweise nicht
* so breiten) Linie mit der Breit ipix und der Farbe
* icol ueberlagert wird.
*
* Eingabe:  hdc      - "Handle of Device Context", der von
*                  gstrt_gi geliefert wird
*           xlu  ,
*           ylu  - "User Coordinates" des Startpunktes
*           x2   ,
*           y2u  - "User Coordinates" des Endpunktes
*           wpix - Gesamtbreite der Linie
*           wcol - "Randfarbe" der Linie
*           ipix - Breite der "Mittellinie"
*           icol - Farbe der Mittellinie
*
* Autor: J. Dankert
*****/

void uwidl_gi (HDC hdc , double xul , double yul ,
              double xu2 , double yu2 ,
              int wpix , COLORREF wcol ,
              int ipix , COLORREF icol)

/*****
* ### Graphics Interface ###
*
* Zeichnen einer gefuellten Ellipse, deren umschliessendes
* Rechteck in "Viewport Coordinates" gegeben ist
* =====
*
* Der Rand des Ellipse wird mit der "Farbe des aktuellen
* Zeichenstiftes" gezeichnet, ausgefuellt wird sie
* mit der Farbe des aktuellen "Brushes".
*
* Eingabe:  hdc      - "Handle of drawing context", der von
*                  gstrt_gi geliefert wird
*           xv1,
*           yv1,
*           xv2,
*           yv2 - Pixel-Koordinaten, die sich auf das
*                  Viewport-Koordinatensystem beziehen
*                  (vgl. "Set current viewport" stcvp_gi
*                  und gstrt_gi), und zwei diagonal
*                  entgegengesetzte Punkte des
*                  umschliessenden Rechtecks beschreiben
*
* Autor: J. Dankert
*****/

void vfell_gi (HDC hdc , int xv1 , int yv1 , int xv2 , int yv2)

```

```

/*****
* ### Graphics Interface ###
*
* Zeichnen eines Rahmens um den "Current Viewport"
* =====
*
* Es wird ein Rahmen um den "Current Viewport" gezeichnet,
* bei offset=0 mit Linien, die die minimalen und maximalen
* Geraete-Koordinaten des Viewports darstellen, bei
* offset>0 werden die Rahmenecken in beiden Richtungen
* um offset Geraeteeinheiten nach innen versetzt.
*
* Eingabe:  hdc      - "Handle of Device Context"
*           offset   - Offset von den Viewport-Ecken
*
* Autor: J. Dankert
*****/

void vfram_gi (HDC hdc , int offset)

```

```

/*****
* ### Graphics Interface ###
*
* Zeichnen eines gefuellten Rechtecks, dessen Eckpunkte in
* "Viewport Coordinates" gegeben sind
* =====
*
* Der Rand des Rechtecks wird mit der "Farbe des aktuellen
* Zeichenstiftes" gezeichnet, ausgefuehlt wird
* es mit der Farbe des aktuellen "Brushs".
*
* Eingabe:  hdc      - "Handle of Device Context"
*           xv1,
*           yv1,
*           xv2,
*           yv2      - Geraete-Koordinaten, die sich auf das
*                       Viewport-Koordinatensystem beziehen
*                       (vgl. "Set current viewport" stcvp_gi
*                       und gstrt_gi) und zwei diagonal
*                       entgegengesetzte Punkte des Rechtecks
*                       beschreiben
*
* Autor: J. Dankert
*****/

void vfrec_gi (HDC hdc , int xv1 , int yv1 , int xv2 , int yv2)

```

```

/*****
* ### Graphics Interface ###
*
* Zeichnen einer geraden Linie zu einem in
* "Viewport Coordinates" gegebenen Punkt
* =====
*
* Die Funktion vline_gi startet an der "Current Position"
* des imaginaeren Zeichenstiftes, die von einer vorange-
* gangenen Zeichenaktion (z. B. vom vorherigen Aufruf von
* vline_gi) oder von einer "Move"-Routine (wie z. B.
* vmove_gi oder umove_gi) hinterlassen wurde.
*
* Hinweis: Wenn beim Aufruf von vline_gi keine "Current
* Position" existiert, wird keine Zeichenaktion
* ausgefuehrt, der eigentliche Zielpunkt wird
* aber als "Current Position" fuer nachfolgende
* Aktionen registriert.
*
* Eingabe: hdc - "Handle of Device Context"
*          xv ,
*          yv - Geraete-Koordinaten, die sich auf das
*               Viewport-Koordinatensystem beziehen
*               (vgl. "Set Current Viewport" stcvp_gi
*               und gstrt_gi)
*
* Autor: J. Dankert
*****/

```

```
void vline_gi (HDC hdc , int xv , int yv)
```

```

/*****
* ### Graphics Interface ###
*
* Bewegen des "Zeichenstiftes" zu einem in
* "Viewport Coordinates" gegebenen Punkt
* =====
*
* Die Funktion vmove_gi fuehrt keine Zeichenaktion aus,
* es wird nur die "Current Position" des imaginaeren
* Zeichenstiftes geaendert.
*
* Hinweis: Die "Move"-Funktionen in GIW fuehren keine
* Aktion mit dem Windows-GDI aus, die "Current
* Position" wird nur im GIW gespeichert
* und fuer eine nachfolgende Zeichenaktion,
* die eine "Current Position" erfordert,
* verwendet.
*
* Eingabe: hdc - "Handle of Device Context"
*          xv ,
*          yv - Geraete-Koordinaten, die sich auf das
*               Viewport-Koordinatensystem beziehen
*               (vgl. "Set Current Viewport" stcvp_gi
*               und gstrt_gi)
*
* Autor: J. Dankert
*****/

```

```
void vmove_gi (HDC hdc , int xv , int yv)
```

```

/*****
* ### Graphics Interface ###
*
* Zeichnen eines Rechtecks, dessen Eckpunkte in
* "Viewport Coordinates" gegeben sind
* =====
*
* Eingabe:  hdc   - "Handle of Device Context"
*           xv1,
*           yv1,
*           xv2,
*           yv2   - Geraete-Koordinaten, die sich auf das
*                   Viewport-Koordinatensystem beziehen
*                   (vgl. "Set Current Viewport" stcvp_gi
*                   und gstrt_gi) und zwei diagonal
*                   entgegengesetzte Punkte des Rechtecks
*                   beschreiben
*
* Autor: J. Dankert
*****/

void vrect_gi (HDC hdc , int xv1 , int yv1 , int xv2 , int yv2)

```