2.2.3 "Gefüllte Flächen" mit "User coordinates"

Neben den in den Abschnitten 2.2.1 und 2.2.2 vorgestellten Methoden **CGI::umove_gi** und **CGI::uline_gi** gibt es in der Klasse **CGI** noch zahlreiche weitere Methoden, die mit "User coordinates" arbeiten. In diesem Abschnitt werden die Methoden (sämtlich zum Zeichnen "gefüllter Flächen") **CGI::ufrec_gi** (Rechteck mit Seitenlinien, die parallel zu den Viewport-Rändern verlaufen, definiert durch zwei Punkte), **CGI::ufell_gi** (Ellipse, definiert durch zwei Punkte des umschließenden Rechtecks), **CGI::ufpol_gi** (geschlossenes Polygon) und **CGI::ufpie_gi** (elliptischer Sektor) vorgestellt, die in der Header-Datei **cgiw.h** mit folgenden Prototypen vertreten sind:

```
void  ufrec_gi  (double xu1 , double yu1 , double xu2 , double yu2) ;
void  ufell_gi  (double xu1 , double yu1 , double xu2 , double yu2) ;
void  ufpie_gi  (double x1  , double y1  , double x2  , double y2   ,
                 double xs  , double ys  , double xe  , double ye)  ;
void  ufpol_gi  (int npoin  , double xu [] , double yu []) ;
```

- Ein Rechteck wird durch die Angabe der Koordinaten zweier Punkte definiert, die sich (beliebig) diagonal gegenüberliegen müssen.
- Eine Ellipse wird wie ein Rechteck definiert, das als das "umschließende Rechteck" der Ellipse interpretiert wird.
- 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).
- Ein geschlossenes Polygon wird durch die Anzahl der Punkte und zwei Felder definiert, die die Koordinaten der Punkte enthalten.

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 **CGBas**-Methode **mkrgb_gi** etwas einfacher geschehen. Diese Methode ist auch in allen anderen **CGIW**-Klassen verfügbar, da in der "Erbfolge" alle **CGIW**-Klassen von **CGBas** "abstammen".

Für die Grundfarben sind in **CGBas** die folgenden "Enumerations"-Konstanten definiert:

```
enum GI_COLORS     {GI_TRANSPARENT = - 1 ,
                    GI_BLACK   ,
                    GI_BLUE    ,
                    GI_GREEN   ,
                    GI_CYAN    ,
                    GI_RED     ,
                    GI_MAGENTA ,
                    GI_YELLOW  ,
                    GI_WHITE   } ;
```

Von dem Programm **fillcol.cpp**, das die Verwendung der oben genannten Methoden zum Zeichnen gefüllter Flächen mit "User coordinates", das Ausfüllen des Viewport-Hintergrunds mit **CGI::vback_gi** (wird anschließend genauer erläutert) und die Verwendung der Methode **mkrgb_gi** demonstriert, wird nachfolgend nur die Methode **CMainFrame::OnPaint** gelistet:

```
void CMainFrame::OnPaint ()
{
   double   poly_x [4] = {   0. ,   6. , 5. ,   4. } ,
            poly_y [4] = { - 3. , - 2. , 2. , - 1. } ;

   CPaintDC  dc (this) ;
   CGI       gi (this , &dc) ;

   int  cxClient = gi.gtcxv_gi () ;
   int  cyClient = gi.gtcyv_gi () ;

   CBrush  Brush1 (gi.mkrgb_gi (gi.GI_BLUE)) ;
   CBrush  Brush2 (gi.mkrgb_gi (gi.GI_CYAN)) ;
   CPen    Pen    (PS_SOLID , 1 , gi.mkrgb_gi (gi.GI_BLACK)) ;

   gi.stcvp_gi (2 , 2 , cxClient / 2 - 4 , cyClient - 4 , gi.GI_XYBOTTOMLEFT) ;
            // ... definiert "Current viewport" in linker Fensterhaelfte

   gi.vback_gi (Brush1 , Pen) ;
            // ... fuellt Viewport mit Farbe, zeichnet Rahmen

   gi.stuca_gi (-1. , -4. , 11. , 5.) ;
            // ... 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)

   CBrush* pBrushOld = dc.SelectObject (&Brush2) ;

   gi.ufrec_gi (0. , 0. , 4. , 4.) ;
            // ... zeichnet ein i. a. "zum Rechteck verzerrtes Quadrat"
   gi.ufell_gi (6. , - 2. , 10. , 2.) ;
            // ... zeichnet einen i. a. "zur Ellipse verzerrten Kreis"
   gi.ufell_gi (8. , 3. , 14. , 6.) ;
            // ... zeichnet Ellipse, die am Viewport-Rand "geclippt" wird
   gi.ufpol_gi (4 , poly_x , poly_y) ;
            // ... zeichnet geschlossenes Polygon mit 4 Punkten
   gi.ufpie_gi (4. , 0. , 8. , 4. , 7. , 4. , 4. , 3.) ;
            // ... zeichnet einen i. a. verzerrten Kreissektor

   gi.stcvp_gi (cxClient / 2 + 2 , 2 ,
                cxClient / 2 - 4 , cyClient - 4 , gi.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
   // ausgeloest werden. Lediglich die fuer den Viewport festgelegten
   // "User coordinates" fuehren auf ein abweichendes Ergebnis:

   gi.vback_gi (CBrush (gi.mkrgb_gi (gi.GI_YELLOW)) , Pen) ;
            // ... fuellt Viewport mit Farbe, zeichnet Rahmen

   gi.stuci_gi (-1. , -4. , 11. , 5.) ;

   dc.SelectObject (&Brush1) ;

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

   dc.SelectObject (pBrushOld) ;
}
```

Für das farbige Ausfüllen der Viewports wurde die Methode **CGI::vback_gi** verwendet, die in **cgiw.h** mit folgendem Prototypen vertreten ist:

```
void  vback_gi  (CBrush &brush , CPen &pen , int offset = 0) ;
```

Als Argumente müssen also Referenzen auf eine **CBrush**-Instanz, die zum Ausfüllen des Viewport-Hintergrunds verwendet wird, und auf eine **CPen**-Instanz (für das Zeichnen des Randes) übergeben werden. Das dritte Argument ist optional. Sinnvoll sind für **offset** positive Werte, die dafür verwendet werden, um das zu zeichnende "gefüllte Rechteck" an jeder Ecke in beiden Richtungen um **offset** Geräteeinheiten kleiner als die Viewport-Abmessungen zu zeichnen.

Im Programm **fillcol.cpp** werden zwei typische Varianten demonstriert, **CGI::vback_gi** aufzurufen. Beim ersten Aufruf wurden vorher konstruierte Instanzen von **CBrush** und **CPen** verwendet:

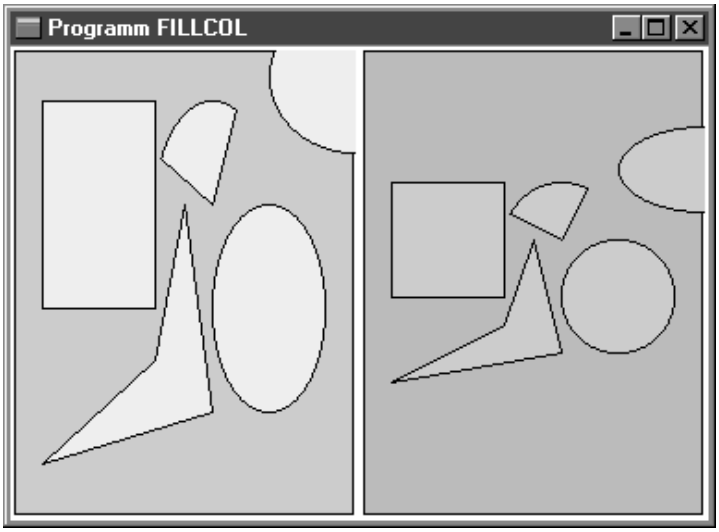
```
CBrush  Brush1 (gi.mkrgb_gi (gi.GI_BLUE)) ;
CPen    Pen    (PS_SOLID , 1 , gi.mkrgb_gi (gi.GI_BLACK)) ;
gi.vback_gi (Brush1 , Pen) ;
```

Die Objekte **Brush1** und **Pen** waren verfügbar, weil sie auch für das Zeichnen der "gefüllten Flächen" verwendet werden. Der "Yellow-Brush" für den Hintergrund des rechten Viewports muß speziell dafür konstruiert werden. Deshalb wird hier die "flüchtige Variante" der Objekt-Erzeugung (nur für den Funktionsaufruf) gewählt:

```
gi.vback_gi (CBrush (gi.mkrgb_gi (gi.GI_YELLOW)) , Pen) ;
```

In allen Fällen wurde der **COLORREF**-Wert für die Farbe mit **mkrgb_gi** erzeugt. Dort könnte natürlich auch das **RGB**-Makro stehen.

Die Funktion **CGI::vback_gi** verändert den "Brush" und den "Pen", die im "Device context" eingesetzt sind, nur vorübergehend. Nach dem Funktionsaufruf ist der vor dem Funktionsaufruf geltende Zustand wieder hergestellt.



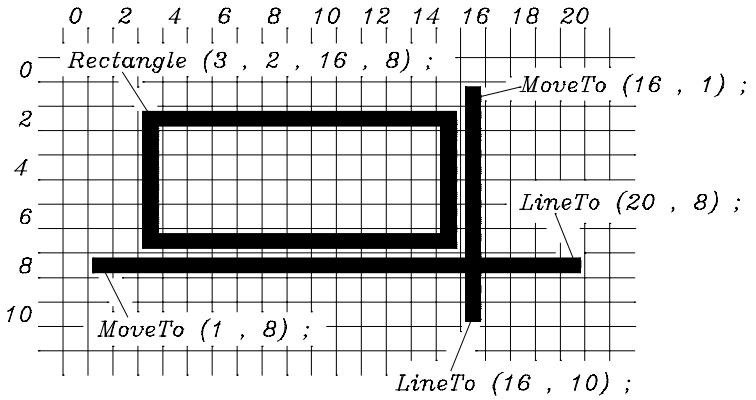
Programm fillcol.cpp: Anisotrope und isotrope Skalierung

Die nebenstehende Abbildung zeigt die Ausgabe des Programms **fillcol.cpp**: 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 CDC-Methode **Rectangle**. Es fällt auf, daß der Punkt links oben exakt die Koordinaten hat, die als Argumente übergeben werden, während die Koordinaten des Punktes rechts unten jeweils um 1 kleiner sind als die Argumente. Der hier am Beispiel des Rechtecks gezeigte Effekt gilt für alle CDC-Methoden, 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 Methoden als "innerhalb des angegebenen Rechtecks zeichnend" vorstellen.



Interpretation der Argumente im Windows-GDI

Dieses Gitter-Koordinaten-Modell versagt natürlich für die CDC-Methoden **MoveTo** und **LineTo**, die genau auf die Pixelpositionen zeichnen (wie natürlich die Methode **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 Methoden verwendet werden. Korrigieren kann der Programmierer diesen Schönheitsfehler nur, wenn er ein Koordinatensystem verwendet, das mit Geräte-Koordinaten arbeitet.

Die CGI-Methoden 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 in der Klasse CGI 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).

In der Klasse CGI werden deshalb für das Zeichnen einiger spezieller Figuren zusätzliche Methoden bereitgestellt (z. B. **ufsec_gi** für das Zeichnen eines "gefüllten Kreissektors", ...), obwohl dies mit den allgemeineren Methoden (z. B. **ufpie_gi**) auch realisierbar wäre, Diese sollten für diese Spezialfälle bevorzugt werden, weil sie 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 Methode **CGI::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 größer. Die Anzahl der Geräteeinheiten, die einem "Logical Inch" entspricht, wird durch die in **cgiw.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 **cgiw.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).

Der Prototyp der Methode **CGI::umark_gi** in der Datei **cgiw.h**

```
void  umark_gi  (int    mtype , double msize ,
                 double xu    , double yu    , int offset = 0) ;
```

zeigt, daß 4 oder 5 Argumente übergeben werden müssen. Für den "Markertyp" **mtype** können "Enumeration"-Konstanten (definiert in der Klasse **CGI**) angegeben werden:

```
enum GI_MARKER     {GI_MKNOMARKER ,
                    GI_MKCIRCLE   ,    // Kreis                 (1)
                    GI_MKSQUARE   ,    // Quadrat               (2)
                    GI_MKCROSS    ,    // Kreuz                 (3)
                    GI_MKVBAR     ,    // Vertikale Linie       (4)
                    GI_MKHBAR     ,    // Horizontale Linie     (5)
                    GI_MKFCIRCLE  ,    // Gefuellter Kreis      (6)
                    GI_MKFSQUARE  } ;  // Gefuelltes Quadrat    (7)
```

Im Beispiel-Programm **marker.cpp** werden die numerischen Werte verwendet.

Die Größe des Markers wird über **msize** gesteuert und beträgt jeweils das "msize-fache" der oben beschriebenen Standardgröße.

Mit **xu** und **yu** werden die "User coordinates" des Marker-Mittelpunkts festgelegt.

Das letzte Argument ist optional. Es legt die "Current position" für die nachfolgende Zeichenaktion fest. Auch dafür können "Enumeration"-Konstanten (definiert in der Klasse **CGI**) verwendet werden:

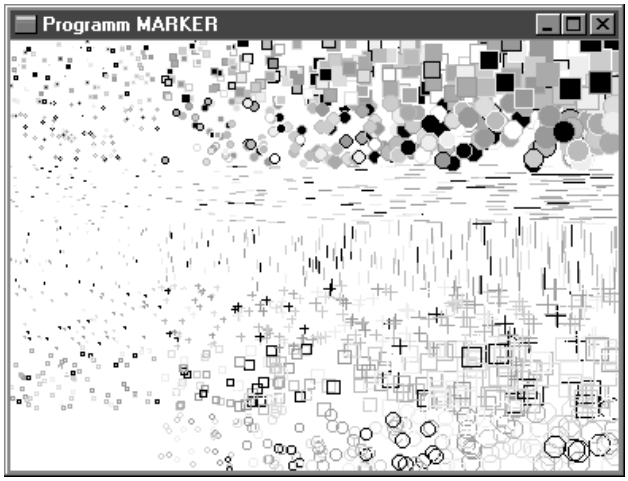
```
enum GI_OFFSET     {GI_NOOFFSET  ,     // ... Markermitte       (0)
                    GI_UPLEFT    ,     // ... links oben        (1)
                    GI_LEFT      ,     // ... links             (2)
                    GI_DOWNLEFT  ,     // ... links unten       (3)
                    GI_UP        ,     // ... oben              (4)
                    GI_CENTER    ,     // ... Markermitte       (5)
                    GI_DOWN      ,     // ... unten             (6)
                    GI_UPRIGHT   ,     // ... rechts oben       (7)
                    GI_RIGHT     ,     // ... rechts            (8)
                    GI_DOWNRIGHT } ;   // ... rechts unten      (9)
```

Das Beispiel-Programm **marker.cpp** demonstriert die Verwendung der Methode **CGI::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 Groesse 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.cpp** wird nachfolgend die Methode **CMainFrame::OnPaint** gelistet:

```
void CMainFrame::OnPaint ()
{
   int       i , mtype , nmark = 2000 ;
   double    msize , xs , ys , rand1 , rand2 ;
   CPen      pen   [8] ;
   CBrush    brush [8] ;
   CPaintDC  dc (this) ;
   CGI       gi (this , &dc) ;
   int  cxClient = gi.gtcxv_gi () ;
   int  cyClient = gi.gtcyv_gi () ;
   for (i = 0 ; i < 8 ; i++)
   {
       pen  [i].CreatePen        (PS_SOLID , 1 , gi.mkrgb_gi (i)) ;
       brush[i].CreateSolidBrush (gi.mkrgb_gi (i)) ;
   }
       // ... stellt 8 "Zeichenstifte" und 8 "Brushs" (fuer die "gefuellten
       //     Markertypen") bereit, die mit den von mkrgb_gi gelieferten
       //     8 Grundfarben arbeiten
   for (i = 1 ; i <= nmark ; i++)
   {
      rand1 = double (rand ()) / RAND_MAX ;
      rand2 = double (rand ()) / RAND_MAX ;
       // ... und 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
      dc.SelectObject (&pen   [int ((rand1 * 8.))]) ;
      dc.SelectObject (&brush [int ((rand2 * 8.))]) ;
```

```
      gi.umark_gi (mtype , msize , xs , ys , 0) ;
   }
   for (i = 0 ; i < 8 ; i++)
   {
      pen  [i].DeleteObject () ;
      brush[i].DeleteObject () ;
   }
}
```



Die nebenstehende Abbildung zeigt die Ausgabe des Programms **marker.cpp**. 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.

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:

- **CGI::umpos_gi** liefert die Mausposition in "User coordinates" und die Information, ob die Position im "Current viewport" liegt.
- **CGI::upick_gi** liefert die Mausposition in "User coordinates" und schaltet einen eventuell vorher eingeschalteten "speziellen CGI-Cursor" ab.
- **CGI::scapp_gi** läßt einen "speziellen CGI-Cursor" erscheinen (z. B. ein über den gesamten Viewport ausgedehntes Kreuz), dieser wird von CGI-Methoden gezeichnet, ist also kein Cursor im Sinne der "Windows-Cursor".
- **CGI::scupd_gi** aktualisiert die Position eines mit **CGI::scapp_gi** erzeugten "speziellen CGI-Cursors" (wird sinnvollerweise als Reaktion auf die Botschaft WM_MOUSEMOVE aufgerufen).
- **CGI::scdap_gi** läßt einen mit **CGI::scapp_gi** erzeugten "speziellen CGI-Cursor" wieder verschwinden.
- **CGI::rpick_gi** ist speziell für das Definieren eines Rechteckbereichs zuständig, setzt den vorherigen Aufruf von **CGI::scapp_gi** voraus. Die Methode **CGI::rpick_gi** "merkt sich" beim ersten Aufruf (z. B. als Reaktion auf WM_LBUTTONDOWN) die Koordinaten des aktuellen Punktes und ändert den "speziellen CGI-Cursor" auf die Form "Rechteck". Beim nächsten Aufruf von **CGI::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. Der spezielle CGI-Cursor (Rechteck) wird abgeschaltet.

2.3.1 "Langlebende" CGI-Instanzen

In allen Beispiel-Programmen der vorherigen Abschnitte wurde ein **CGI**-Objekt bei der Bearbeitung der Botschaft WM_PAINT in **OnPaint** nach dem Anfordern eines "Device contextes" (z. B. als **CPaintDC**-Objekt) erzeugt und hauchte am Ende von **OnPaint** sein Leben wieder aus. Im Zusammenhang mit der Bearbeitung der Maus-Botschaften sind nun auch Zeichenaktionen während der Bearbeitung anderer Botschaften angesagt, für die ein neuer "Device context" (z. B. als **CClientDC**-Objekt) angefordert werden muß.

Es wäre aufwendig und auch wenig sinnvoll, nach jedem Anfordern eines "Device contextes" auch ein neues **CGI**-Objekt zu konstruieren, zumal wesentliche Informationen via **CGI**-Objekt von einer Zeichenaktion zur folgenden "konserviert" werden können. Deshalb sollte in solchen Fällen unbedingt mit einem **CGI**-Objekt gearbeitet werden, dessen Lebensdauer nicht auf die Arbeitszeit einer speziellen Funktion beschränkt ist.

Man kann ein **CGI**-Objekt als globale Variable erzeugen oder z. B. als Mitglied einer Klasse, deren Objekte eine ausreichend lange Lebenszeit haben. Die zu beachtenden Besonderheiten sind in beiden Fällen gleich. In den nachfolgenden Beispiel-Programmen wird die letztgenannte Variante realisiert:

- In der Fensterklasse **CMainFrame** wird ein **CGI**-Objekt "private" angesiedelt und hat damit die völlig ausreichende Lebensdauer wie das Hauptfenster-Objekt:

```
class CMainFrame : public CFrameWnd
{
      private:
            CGI  m_gi ;
      // ...
}
```

  Beim Erzeugen einer Instanz der **CMainFrame**-Klasse (in **InitInstance**) wird also auch das zur Klasse gehörende **CGI**-Objekt konstruiert, kann aber nur mit dem Standard-Konstruktor initialisiert werden, weil zu diesem Zeitpunkt noch kein "Device context" existiert.

- Der Standard-Konstruktor setzt im **CGI**-Objekt als Pointer auf den "Device context" einen NULL-Pointer ein, der von allen "zeichnenden **CGI**-Methoden" beachtet wird. Wenn der Programmierer versehentlich eine Zeichenaktion mit einem solchen **CGI**-Objekt ausführen lassen will, wird dies abgelehnt.

  Um ein solches "nicht komplettes" **CGI**-Objekt "zeichenfähig" zu machen, muß es initialisiert werden, wenn der Pointer auf den "Device context" verfügbar ist. Die dafür verfügbare Methode **CGI::init_gi** ist mehrfach überladen. Es stehen etwa die gleichen Varianten wie für den ebenfalls mehrfach überladenen Konstruktor dieser Klasse zur Verfügung. Im Programm **ucpick.cpp** des folgenden Abschnitts wird das **CGI**-Objekt folgendermaßen initialisiert:

```
void CMainFrame::OnPaint ()
{
   CPaintDC  dc (this) ;
   CRect           rect  ;
   GetClientRect (&rect) ;
   int  cxClient = rect.Width  () ;
   int  cyClient = rect.Height () ;
   m_gi.init_gi (&dc , cxClient , cyClient) ;     // ...
```

Das Programm **zoom.cpp** im Abschnitt 2.3.3 demonstriert die etwas einfachere Variante, bei der (wie bei der entsprechenden Konstruktor-Variante) die **CGI**-Methode sich über den Pointer auf das Fenster die notwendigen Informationen (Abmessungen) selbst beschafft:

```
void CMainFrame::OnPaint ()
{
   CPaintDC  dc (this) ;
   m_gi.init_gi (this , &dc) ;
   int  cxClient = m_gi.gtcxv_gi () ;
   int  cyClient = m_gi.gtcyv_gi () ;    // ...
```

- Nach dem Aufruf von **CGI::init_gi** können alle **CGI**-Zeichenfunktionen ausgeführt werden. Weil am Ende von **OnPaint** das **CPaintDC**-Objekt stirbt, ein Pointer auf dieses Objekt aber im ("weiterlebenden") **CGI**-Objekt noch verzeichnet ist, sollte man diesen unbedingt (zur eigenen Sicherheit) mit **CGI::remdc_gi** entfernen:

```
   m_gi.remdc_gi () ;
}  // Ende der Funktion OnPaint
```

Damit ist das **CGI**-Objekt wieder für Zeichenaktionen untauglich, für andere Aktionen aber durchaus noch nützlich. Im Programm **ucpick.cpp** werden z. B. während der Bearbeitung der Botschaft WM_MOUSEMOVE mit **CGI::umpos_gi** die aktuellen Cursor-Koordinaten auf "User coordinates" umgerechnet. Dies funktioniert, weil die aktuellen Viewport-Abmessungen und die Definition der "User coordinates" noch gespeichert sind und z. B. bei Änderung der Fenstergröße über den dadurch erneuten Aufruf von **OnPaint** und damit auch von **CGI::init_gi** aktualisiert werden.

- Wenn (wie im Programm **zoom.cpp**) während der Bearbeitung anderer Botschaften auch mit **CGI**-Funktionen gezeichnet werden soll, braucht keine komplette Initialisierung des **CGI**-Objektes erfolgen, es muß nur der Pointer auf den für das Zeichnen anzufordernden "Device context" eingesetzt werden. Dies erledigt **CGI::setdc_gi**, wird erst im Abschnitt 3.4 (Programm **malteser.cpp**) demonstriert, aus dem dieser Auszug (Auswertung der Botschaft ON_TIMER) stammt:

```
void CMainFrame::OnTimer (UINT nIDTimer)
{
   CClientDC  dc (this) ;          // "Device context" ...
   m_gi.setdc_gi (&dc)  ;          // in CGI-Objekt einsetzen
   // ... Aufruf von CGI-Zeichenfunktionen ...
   m_gi.remdc_gi () ;
}
```

- Beim Arbeiten mit den **CGI**-Methoden, die einen "Spezialcursor" zeichnen, wird es für den Programmierer noch etwas einfacher. Er übergibt an diese Funktionen den Pointer auf das aktuelle Window, und die **CGI**-Methoden besorgen sich selbst einen Pointer auf einen "Device context", folgender Ausschnitt aus **zoom.cpp** zeigt die Funktion **OnMouseMove**, mit der die Botschaft WM_MOUSEMOVE bearbeitet wird. Die Methode **CGI::scupdt_gi** ("special cursor update") kann den Spezialcursor zeichnen, weil sie den Pointer auf das aktuelle Fenster (**this**-Pointer) empfängt:

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

2.3.2 Cursorbewegung verfolgen, Punkte picken

Das Programm **ucpick.cpp** demonstriert das Arbeiten mit den Methoden **CGI::init_gi**, **CGI::remdc_gi** und **CGI::umpos_gi** in Viewports unterschiedlicher Größe. Beim Start des Programms ist die gesamte "Client area" gleichzeitig der "Current viewport". Über das Menüangebot **Viewport groß/klein** kann ein wesentlich kleinerer Viewport eingestellt werden. Um dies sichtbar zu machen, wird der Viewport mit Farbe gefüllt, und es wird die mathematische Funktion

$$x = 4 \cos 21\,t \quad ; \quad y = 4 \sin 20\,t$$

hineingezeichnet, deren Werte den Bereich

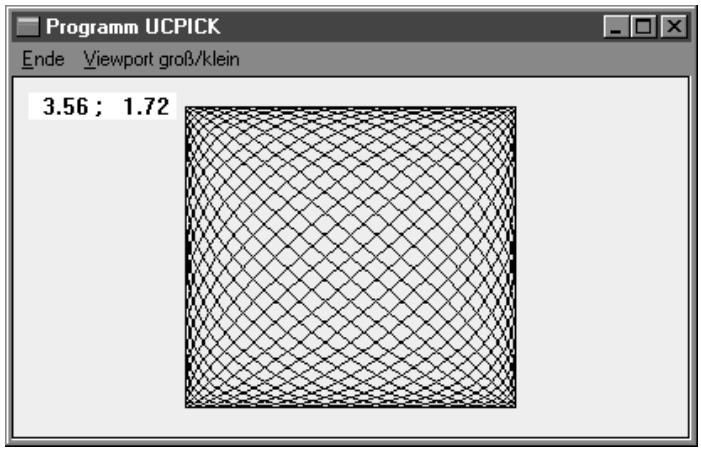
$$-4 \le x \le +4 \quad ; \quad -4 \le y \le +4$$

überstreichen, für den mit **CGI::stuci_gi** die "User coordinates" isotrop (10% Rand) eingestellt werden. Damit hat man eine gewisse Vorstellung von den Koordinaten, die mit dem Cursor überstrichen werden, einen anderen Zweck erfüllt die dargestellte Funktion in diesem Programm nicht.

Eine **CGI**-Instanz wird (wie im Abschnitt 2.3.1 beschrieben) in der Klasse **CMainFrame** angesiedelt und beim Erzeugen des Hauptfensters mit dem Standard-Konstruktor konstruiert. In **OnPaint** muß das Objekt deshalb mit **CGI::init_gi** initialisiert werden. Danach kann gezeichnet werden:

```
void CMainFrame::OnPaint ()
{
   CPaintDC  dc (this) ;
   CRect           rect  ;
   GetClientRect (&rect) ;
   int  cxClient = rect.Width  () ;
   int  cyClient = rect.Height () ;
   m_gi.init_gi (&dc , cxClient , cyClient) ;
   if (m_vpbig)
   {
      m_gi.stcvp_gi (0 , 0 , cxClient , cyClient , m_gi.GI_XYCENTER) ;
   }
   else
   {
      m_gi.stcvp_gi (cxClient / 4 , cyClient / 4 ,
                     cxClient / 2 , cyClient / 2 , m_gi.GI_XYCENTER) ;
   }
   m_gi.vback_gi (CBrush (m_gi.mkrgb_gi (m_gi.GI_CYAN)) ,
                  CPen   (PS_SOLID , 1 , m_gi.mkrgb_gi (m_gi.GI_BLACK))) ;
   // ... fuellt Viewport-Hintergrund mit Farbe, zeichnet Rahmen
   m_gi.stuci_gi (-4. , -4. , 4. , 4. , 10.) ;
   double  t  = 0. ;
   double  dt = GI_PI * 2. / NSTEPS ;
   for (int i = 0 ; i <= NSTEPS ; i++ , t += dt)
   {
      m_gi.uline_gi (4. * cos (21. * t) , 4. * sin (20. * t)) ;
   }
   m_gi.remdc_gi () ;   // ... dient der eigenen Sicherheit: Pointer auf
                        //     auf "Device context" wird aus der (weiter-
                        //     lebenden) CGI-Instanz entfernt, damit nicht
                        //     versehentlich eine zeichnende CGI-Methode
                        //     diesen "Device context", der gerade stirbt,
                        //     benutzt.
}
```

Die nebenstehende Abbildung zeigt das Hauptfenster des Programms nach dem Programmstart: Der "Current viewport" ist mit der "Client area" des Hauptfensters identisch, die isotrope Skalierung sorgt dafür, daß die Funktion den ("breiten") Viewport rechts und links nicht ausfüllt.



Programm ucpick.cpp: Hauptfenster nach dem Programmstart

Links oben ist die aktuelle Cursorposition in "User coordinates" zu sehen, die sich bei Maus-Bewegungen ständig ändert. Dies wird realisiert durch Auswertung der Botschaft WM_MOUSEMOVE in der Methode **CMainFrame::OnMouseMove**. Diese empfängt die aktuellen Cursor-Koordinaten in einem **CPoint**-Objekt, das der **CGI**-Methode **umpos_gi** übergeben wird, die daraus die "User coordinates" berechnet, wenn der Punkt im "Current viewport" liegt. Sie ist in der Klasse **CGI** mit folgendem Prototypen verzeichnet:

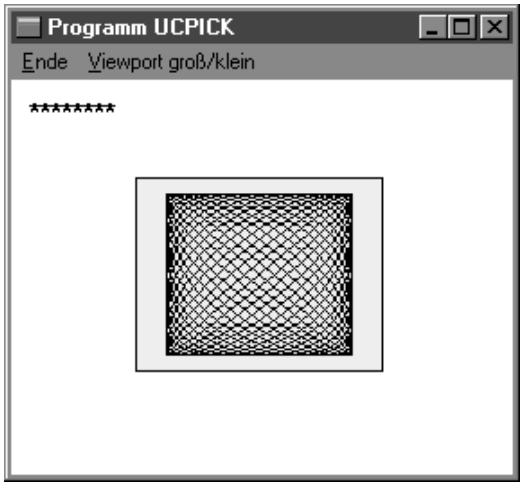
```
int  umpos_gi  (POINT point , double *xu_p , double *yu_p) ;
```

Als Return-Wert wird eine 1 abgeliefert, wenn der Punkt im "Current viewport" liegt, ansonsten eine 0. Dies alles wird in **CMainFrame::OnMouseMove** ausgewertet, schließlich wird ein "Device context" als **CClientDC**-Objekt erzeugt, mit dem der Text in die linke obere Fensterecke geschrieben wird:

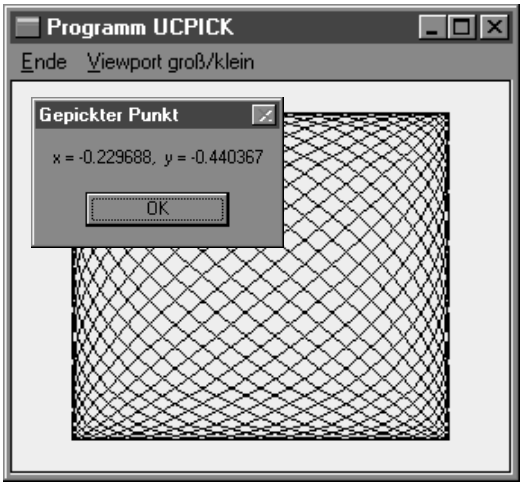
```
void CMainFrame::OnMouseMove (UINT nFlags , CPoint point)
{
   char    str[20] ;
   double  xu , yu ;
   if (m_gi.umpos_gi (point , &xu , &yu))         // Punkt im
   {                                              // "Current viewport" ?
       sprintf (str , "%6.2lf ; %6.2lf " , xu , yu) ;
   }
   else
   {
       sprintf (str , "********           ") ;
   }
   CClientDC dc (this) ;                    // ... aktualisiert staendig die
   dc.TextOut (10 , 10 , CString (str));    //     Cursor-Position ("User
}                                           //     coordinates")
```

Wenn der kleinere Viewport eingestellt ist, kann die aktuelle Cursor-Position außerhalb des Viewports liegen. Die Abbildung auf der folgenden Seite oben links zeigt diese Situation.

Das Programm wertet auch die Botschaft WM_LBUTTONDOWN (linke Maustaste gedrückt) aus. Die Methode **CMainFrame::OnLButtonDown** empfängt wie **CMainFrame::OnMouseMove** ein **CPoint**-Objekt, gibt es an **CGI::umpos_gi** zur Auswertung und zeigt die "User coordinates" des gepickten Punkts in einer Message-Box an (Abbildung oben rechts auf der folgenden Seite).



Cursor außerhalb des Viewports

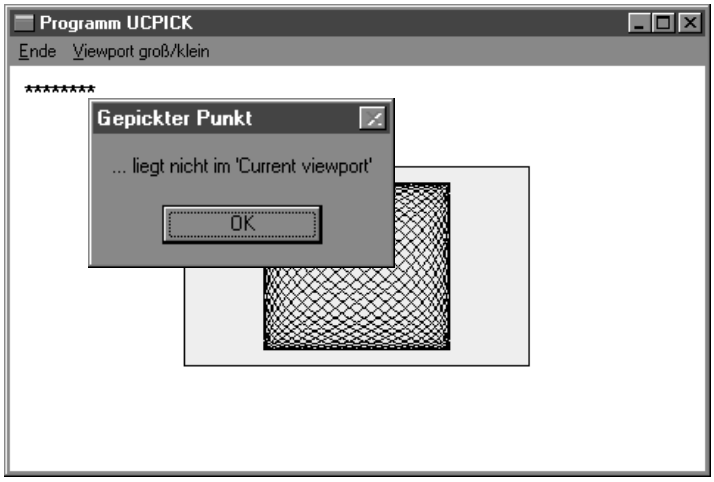


"User coordinates" des gepickten Punktes

Die Methode **CMainFrame::OnLButtonDown** ist im Programm **ucpick.cpp** folgendermaßen codiert:

```
void CMainFrame::OnLButtonDown (UINT nFlags , CPoint point)
{
   char    str[40] ;
   double  xu , yu ;
   if (m_gi.umpos_gi (point , &xu , &yu))         // Punkt im
   {                                              // "Current viewport" ?
       sprintf (str , "x = %g,  y = %g" , xu , yu) ;
   }
   else
   {
       sprintf (str , "... liegt nicht im 'Current viewport'") ;
   }
   MessageBox (str , "Gepickter Punkt") ;
}
```

Auch bei dieser Aktion kann, wenn der kleinere Viewport zu sehen ist, der gepickte Punkt außerhalb des "Current viewport" liegen, so daß von **CGI::umpos_gi** keine "User coordinates" berechnet werden können. Dann reagiert das Programm mit der entsprechenden Ausschrift (nebenstehende Abbildung).



2.3.3 Zoom

Die Methode **CMainFrame::OnPaint** des Programms **zoom.cpp** entspricht weitgehend der entsprechenden Funktion des Programms **ucpick.cpp** aus dem vorigen Abschnitt:

```
void CMainFrame::OnPaint ()
{
   CPaintDC  dc (this) ;
   m_gi.init_gi (this , &dc) ;
   int  cxClient = m_gi.gtcxv_gi () ;
   int  cyClient = m_gi.gtcyv_gi () ;
   if (m_vpbig)
   {
      m_gi.stcvp_gi (0 , 0 , cxClient , cyClient , m_gi.GI_XYCENTER) ;
   }
   else
   {
      m_gi.stcvp_gi (cxClient / 4 , cyClient / 4 ,
                     cxClient / 2 , cyClient / 2 , m_gi.GI_XYCENTER) ;
   }
   m_gi.vback_gi (CBrush (m_gi.mkrgb_gi (m_gi.GI_YELLOW)) ,
                  CPen   (PS_SOLID , 1 , m_gi.mkrgb_gi (m_gi.GI_BLACK))) ;
   // ... fuellt Viewport-Hintergrund mit Farbe, zeichnet Rahmen
   m_gi.stuci_gi (min (m_p1x , m_p2x) , min (m_p1y , m_p2y) ,
                  max (m_p1x , m_p2x) , max (m_p1y , m_p2y) , 10.) ;
   double  t  = 0. ;
   double  dt = GI_PI * 2. / NSTEPS ;
   for (int i = 0 ; i <= NSTEPS ; i++ , t += dt)
   {
      m_gi.uline_gi (4. * cos (17. * t) * cos (t) ,
                     4. * cos (17. * t) * sin (t)) ;
   }
   m_zoom  = 0 ;
   m_gi.remdc_gi () ;
}
```

Die wichtigsten Unterschiede zum Programm **ucpick.cpp** sind durch Fettdruck markiert:

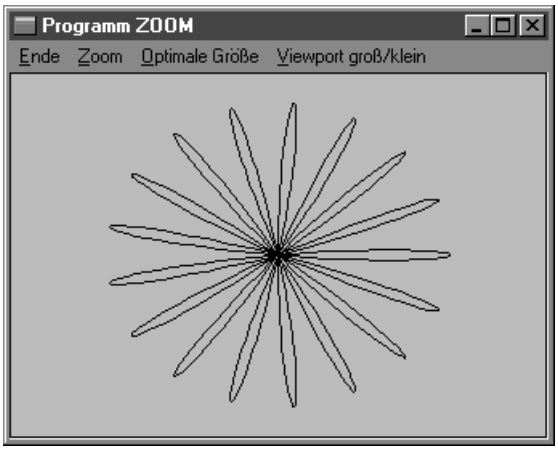
- Die Initialisierung des bereits konstruierten **CGI**-Objekts wird mit der etwas einfacheren Variante von **CGI::init_gi** erledigt (vgl. Abschnitt 2.3.1).
- Die "User coordinates" werden mit **CGI::stuci_gi** mit **variablen Eckpunkt-Koordinaten** definiert, die zur Klasse **CMainFrame** gehören, im Konstruktor von **CMainFrame** initialisiert und bei den Zoom-Aktionen geändert werden.
- Es wird die mathematische Funktion

  $$r = 4 \cos(17 \varphi)$$

  (Polarkoordinaten) im Bereich

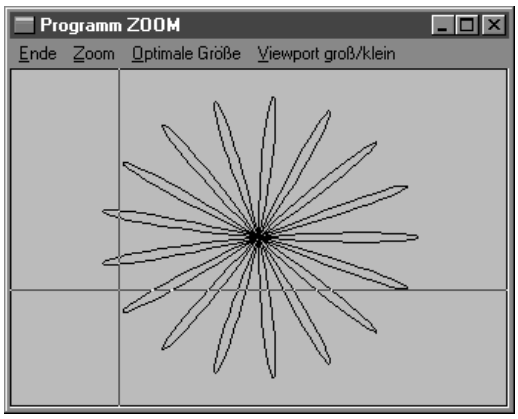
  $$0 \leq \varphi \leq 2 \pi$$

  gezeichnet. Die nebenstehende Abbildung zeigt das Hauptfenster nach dem Programmstart.



Hauptfenster nach dem Start von zoom.cpp

Über das Menü-Angebot **Zoom** wird die Festlegung eines rechteckigen Ausschnitts der Zeichnung ermöglicht, der dann vergrößert (den gesamten Viewport füllend, der beim Programmstart mit der "Client area" identisch ist) dargestellt werden soll. Realisiert wird dies, indem die "User coordinates" der beiden Punkte, die den neuen Zeichnungsausschnitt markieren, für die nachfolgende Zeichenaktion als die Punkte bereitgehalten werden, mit denen (über **CGI::stuci_gi**) die neuen "User coordinates" definiert werden.



Hauptfenster nach der Menü-Auswahl "Zoom"

Die nebenstehende Abbildung zeigt das Hauptfenster nach der Wahl von **Zoom**. Die dadurch ausgelöste WM_COMMAND-Botschaft wird im Programm **zoom.cpp** von der Methode **CMainFrame::OnZoom** bearbeitet:

```
void CMainFrame::OnZoom  ()
{
   m_gi.scapp_gi (this , 0 , 0 , m_gi.GI_CUBIGCROSS) ;
   m_zoom  = 1 ;
}
```

Die Methode **CGI::scapp_p**, die den "Spezialcursor" (in diesem Fall das "große Kreuz") erscheinen läßt, ist in **cgiw.h** mit folgenden Protytypen vertreten:

```
void  scapp_gi  (CWnd *wnd_p , double xu , double yu , int ctype) ;
void  scapp_gi  (CWnd *wnd_p , int    xv , int    yv , int ctype) ;
```

Der Pointer auf das aktuelle Window ermöglicht **scapp_gi** die Zeichenaktion ("Device context" anfordern), die beiden Koordinaten legen fest, an welchem Punkt der "Spezial-Cursor" erscheinen soll (im Programm **zoom.cpp** werden "User coordinates" angegeben, die überladene Variante erwartet "Viewport coordinates"). Für den letzten Paramter dürfen die Typen **GI_CUBIGCROSS** (großes Kreuz) oder **GI_CUSMALLCROSS** (kleines Kreuz) angegeben werden.

Der (in der Klasse **CMainFrame** angesiedelte) Parameter m_zoom wird auf den Wert 1 gesetzt, was im Programm **zoom.cpp** ab sofort die Auswertung der Botschaft WM_MOUSE-MOVE veranlaßt, die an **CMainFrame::OnMouseMove** geleitet wird:

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

CGI::scupd_gi mit dem Prototyp

```
void  scupd_gi  (CWnd *wnd_p , POINT point) ;
```

empfängt (neben dem Pointer auf das aktuelle Fenster) den mit der Botschaft WM_MOUSE-MOVE gelieferten Punkt, überzeichnet den alten "Spezialcursor" und zeichnet ihn an der aktuellen Position neu.

Die Zeichenaktionen in **CGI::scapp_gi** und **CGI::scupd_gi** werden übrigens mit dem "ROP2"-Code **R2_NOT** erledigt, der eine gesetzte Zeichenfarbe ignoriert und mit der zum Hintergrund inversen Farbe zeichnet. Die Linen des "Spezial-Cursors" auf dem gelben Hintergrund erscheinen also blau, wo die schwarzen Linien der Funktionsdarstellung gekreuzt werden, sind sie weiß.

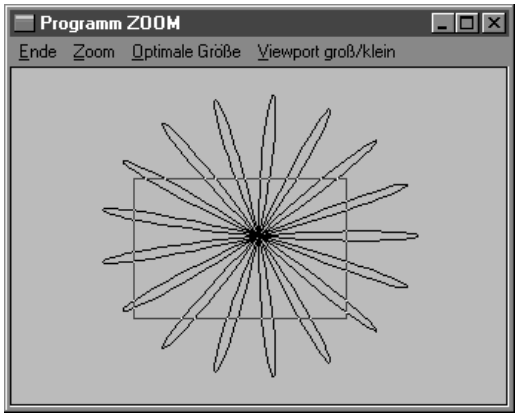
Wenn der Parameter **m_zoom** den Wert 1 hat, wird auch die Botschaft WM_LBUTTON-DOWN ausgewertet. die an **CMainFrame::OnLButtonDown** geleitet wird:

```
void CMainFrame::OnLButtonDown (UINT nFlags , CPoint point)
{
   if (m_zoom)
   {
      if (m_gi.rpick_gi (this   , point  ,
                         &m_p1x , &m_p1y , &m_p2x , &m_p2y))
      {
         Invalidate () ;  // ... wird erst beim zweiten Aufruf
                          //     von rpick_gi ausgefuehrt
      }
   }
}
```

Die Methode **CGI::rpick_gi** mit dem Prototyp

```
int  rpick_gi  (CWnd *wnd_p , POINT  point , double *x1u_p , double *y1u_p ,
                                             double *x2u_p , double *y2u_p) ;
```

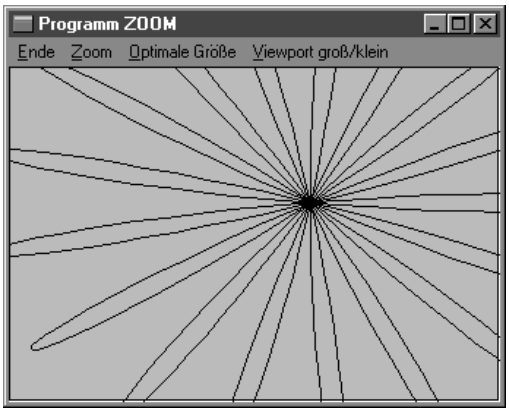
empfängt den mit der Botschaft WM_LBUTTONDOWN ankommenden Punkt, läßt den aktuellen "Spezialcursor" verschwinden und führt **beim ersten Aufruf** noch folgende Aktionen aus: Der Typ des "Spezialcursors" wird auf **GI_CURECTANGLE** (Rechteck) umgestellt, dieser Cursor wird sofort gezeichnet und danach ständig von **CMainFrame::OnMouseMove** aktualisiert (diese Situation zeigt die nebenstehende Abbildung). Abgeliefert werden von **CGI::rpick_gi** die "User coordinates" des übergebenen Punktes (*x1u_p und *y1u_p) und der Return-Wert 0.



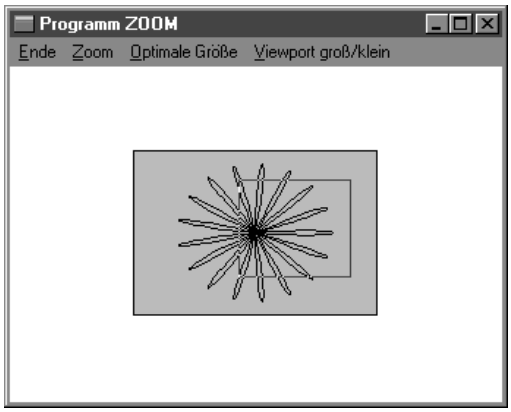
Der "Rechteck-Cursor" wird ständig aktualisiert

Beim zweiten Aufruf von **CGI::rpick_gi** (im Programm **zoom.cpp** nach einem erneuten Drücken der linken Maustaste, das kann man auch anders organiseren) wird der "Spezialcursor" abgeschaltet, und **rpick_gi** liefert die "User coordinates" **beider** Punkte und den Return-Wert 1.

Der Return-Wert 1 wird im Programm **zoom.cpp** zum Anlaß genommen, über **Invalidate** eine WM_PAINT-Botschaft auszulösen, so daß neu gezeichnet wird. Weil die von **rpick_gi** abgelieferten "User coordinates" der beiden gepickten Punkte im **CMainFrame**-Objekt auf den Komponenten m_p1x , m_p1y , m_p2x , m_p2y abgelegt wurden, die in **OnPaint** für die Definition neuer "User coordinates" verwendet werden, liegen die gepickten Punkte dann in den Viewport-Ecken, so daß die "gezoomte" Zeichnung dargestellt wird (Abbildung oben links auf der folgenden Seite).



Darstellung nach der "Zoom"-Aktion



"Zoom"-Aktion in einem kleinen Viewport

Natürlich sind alle Aktionen auch in einem "kleinen" Viewport möglich, der nicht die gesamte "Client area" ausfüllt. Die Abbildung oben rechts zeigt dies nach dem Umschalten über **Viewport groß/klein** in dem Zustand des Verfolgens des Rechteck-Cursors.

Die "Zoom-Aktionen" können beliebig wiederholt werden. Über das Menü-Angebot **Optimale Größe** kann schließlich alles rückgängig gemacht werden. Die zugehörige WM_COMMAND-Botschaft wird an **CMainFrame::OnOptimal** geleitet:

```
void CMainFrame::OnOptimal ()
{
   m_gi.scdap_gi (this) ;      // Es koennte ein spezieller Cursor
                               // angeschaltet sein
   m_p1x   = -4. ;
   m_p1y   = -4. ;
   m_p2x   =  4. ;
   m_p2y   =  4. ;
   Invalidate () ;
}
```

Vorsorglich wird **CGI::scdap_gi** (Abschalten eines eventuell sichtbaren "Spezialcursors") aufgerufen. Dieser Methode wird nur der Pointer auf das aktuelle Window übergeben. Danach werden die für die Definition der "User coordinates" zuständigen Koordinatenwerte auf die Anfangswerte zurückgesetzt, und mit **Invalidate** wird das Neuzeichnen initiiert.

Die in **zoom.cpp** realisierte Variante der Definition des Rechtecks mit zweimaligem Klick mit der linken Maustaste ist eine Möglichkeit. Allgemein gilt das "Aufziehen" eines Rechtecks als elegantere Lösung: Die linke Maustaste wird gedrückt und legt damit den ersten Punkt fest, danach wird bei gedrückter Taste das Rechteck "aufgezogen", und der zweite Punkt wird durch das Lösen der linken Maustaste festgelegt. In diesem Fall muß also auch die Botschaft WM_LBUTTONUP ausgewertet werden. Diese Variante ist in den Beispiel-Programmen **stab2.cpp** und **mtharea2.cpp** realisiert.