1 Vorbemerkungen

Die Klassen-Bibliothek **CGIW** ist für die Windows-C^{++}-Programmierung mit dem Entwicklungssystem Microsoft-Visual-C^{++} konzipiert (C-Programmierer können die Library **GIW.LIB** benutzen, für die eine gesonderte Dokumentation verfügbar ist). Die **CGIW**-Klassen verstehen sich als Ergänzungen zu den "Microsoft Foundation Classes" (MFC), benutzen diese und können gemischt mit ihnen verwendet werden.

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

1.1 Funktionalität der CGIW-Klassen

Neben einigen kleineren Hilfen für den Programmierer bieten die **CGIW**-Klassen für folgende Bereiche Unterstützung an:

- Der Programmierer kann sogenannte **"User coordinates"** definieren, die Punkte in der Ebene mit **double**-Werten darstellen. Der Ursprung des "User coordinates"-Systems und die Richtungen der Achsen können beliebig festgelegt werden, die Achsen können wahlweise isotrop oder anisotrop skaliert werden. Die wichtigsten Zeichenfunktionen werden für den Aufruf mit "User coordinates" bereitgestellt.
- Mit den sogenannten **"t...-Routinen"** werden Zeichenoperationen ausgeführt, bei denen die übergebenen Koordinaten vorab einer ebenen Transformation unterworfen werden. Die Transformationen (Translation, Rotation, Spiegelung, Skalierung) können initialisiert, gesetzt und inkrementiert werden.
- Es kann in sogenannten **"World coordinates"** (3D-Koordinaten) gearbeiten werden. Eine beliebige Projektion (Zentral- oder Parallelprojektion) muß vorab definiert werden. Danach werden alle 3D-Punkte, die den "zeichnenden pr...-Routinen" übergeben werden, mit der aktuellen Projektion in Punkte der 2D-Zeichenfläche umgerechnet.
- Es kann auch eine **3D-Transformation** definiert werden, die von den sogenannten "pt...-Routinen" ausgewertet wird, die die ihnen übergebenen 3D-Punkte zunächst dieser Transformation und danach der aktuellen Projektion unterwerfen.
- Mit einigen speziellen Routinen wird das **Picken von Punkten und das Zoomen** unterstützt, wobei die Punkte bzw. Rechteckbereiche in "User coordinates" übergeben werden.
- Einige Klassen unterstützen das Verwalten und Verarbeiten von 3D-Modellen. Speziell wird das geordnete Einfügen von 3D-Objekten in einen binären Baum unter-

stützt, um über eine geeignete Reihenfolge der Zeichenaktionen die korrekte Überdeckung der Elemente in Abhängigkeit von Projektionszentrum bzw. Blickrichtung zu realisieren.

1.2 Verfügbarkeit der Klassen-Bibliothek CGIW

Der gesamte Code der Klassen-Bibliothek **CGIW** wird als Quelltext zur Verfügung gestellt. Es ist empfehlenswert, daß sich der Programmierer, der die Bibliothek benutzen will, sich die Library, die er in seine Programme einbinden möchte, selbst erzeugt. Auf diese Weise wird mit Sicherheit Kompatibilität mit der MS-Visual-C++-Version hergestellt, die er verwendet.

Die Klassen-Bibliothek **CGIW** und alle in diesem Manual beschriebenen Beispiel-Programme wurden mit folgenden MS-Visual-C++-Versionen erfolgreich getestet:

- MS-Visual-C++ 1.5 unter Windows 3.1 (wer nur diese Version besitzt, kann sie auch unter Windows 95 erfolgreich betreiben, auch für diese Variante wurde die **CGIW**-Library getestet),
- MS-Visual-C++ 4.0 unter Windows 95 und Windows NT,
- MS-Visual-C++ 5.0 unter Windows 95 und Windows NT (alle in diesem Manual beschriebenen Beispiele sind auch mit der "Einsteiger-Edition" zu realisieren).

Über die Internet-Adresse, über die dieses Manual und der **CGIW**-Quellcode zu beziehen sind, können auch "readme"-Dateien bezogen werden, in denen genau beschrieben wird, wie aus der "Visual workbench" (MS-Visual-C++ 1.5) bzw. dem "Developer studio" (MS-Visual-C++ 4.0 und höher) die Libraries erzeugt und die Beispiel-Programme compilert und mit der Library zu einem ausführbaren Programm gelinkt werden können. Auf folgendes ist dabei besonders zu achten:

- Sowohl die Library als auch die Beispiel-Programme sollten unbedingt mit den gleichen Compiler-Optionen "Debug" bzw. "Release" erzeugt werden.
- Sowohl beim Erzeugen der Library als auch beim Erzeugen der ausführbaren Programme muß "Use Microsoft Foundation Classes" bzw. "MFC in einer Statischen Bibliothek oder einer gemeinsam genutzten DLL verwenden" aktiviert sein.
- Beim Arbeiten mit MS-Visual-C++ 1.5 muß darauf geachtet werden, daß das gleiche "Memory model" für die Library und die Beispiel-Programme verwendet wird (empfohlen wird "Large", Standard-Einstellung ist "Medium"). Bei den größeren Programmen kann die zusätzliche Option **-K** für den "Resource compiler" erforderlich sein, worauf man allerdings gegebenenfalls durch eine "intelligente Fehlermeldung" aufmerksam gemacht wird.

Das leidige Problem der "16-Bit-Welt" mit unterschiedlichen Speichermodellen und "Near- und Far-Pointern" ist glücklicherweise ein Auslaufmodell. Es wird in der CGIW-Klassen-Bibliothek nicht besonders berücksichtigt (im Gegensatz zur GIW-Library für die C-Programmierung, bei der dieses Problem noch ausführlich behandelt wurde). Damit sind bei der Arbeit mit MS-Visual-C++ 1.5 natürlich einige Grenzen gesetzt, auf die bei der Behandlung der speicherintensiven 3D-Modelle besonders aufmerksam gemacht wird.

2 2D-Koordinatensysteme

Das Windows-GDI ("Graphics Device Interface") 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 Methoden der **CGIW**-Klassen arbeiten intern ausschließlich mit dem GDI-Koordinatensystem **MM_TEXT**, stellen aber dem Programmierer Methoden für das Arbeiten mit unterschiedlichen Koordinatensystemen zur Verfügung:

- Die Methoden, die mit sogenannten **"Viewport coordinates"** arbeiten (die Namen dieser Methoden beginnen mit **v**), erwarten **int**-Argumente, die als "Geräte-Einheiten" interpretiert werden (für die Bildschirm-Ausgabe: Pixel).
- Die Methoden, die mit den sogenannten **"User coordinates"** arbeiten (die Namen dieser Methoden beginnen mit **u**), 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.

2.1 Das CGIW-Viewport-Konzept

2.1.1 Arbeiten mit der Klasse CGI, Viewports definieren

Zunächst wird in den Beispiel-Programmen jeweils genau eine Instanz der Klasse **CGI** erzeugt. Bei der Konstruktion eines **CGI**-Objektes werden dem Konstruktor (in der Regel, andere Konstruktoren werden später behandelt) der **Pointer auf den "Device context"** und die Abmessungen der Zeichenfläche in Geräte-Koordinaten übergeben (bzw. der Pointer auf das aktuelle Window, so daß sich der Konstruktor die Abmessungen der "Client area" selbst besorgen kann).

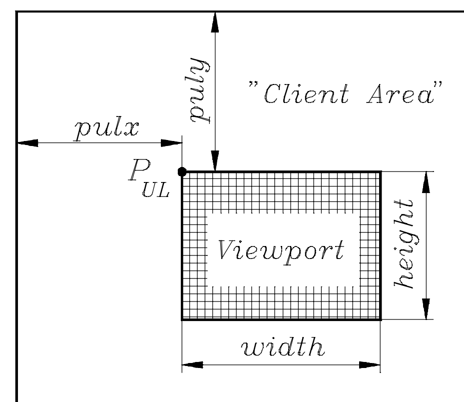
Zunächst wird der typische Anwendungsfall für das Erzeugen eines **GDI**-Objektes besprochen, die Bearbeitung der Botschaft **WM_PAINT**, bei der der "Device context"-Pointer z. B. als Objekt der Klasse **CPaintDC** erzeugt wird, die Abmessungen der Zeichenfläche können mit **GetClientRect** ermittelt werden (in einem mit dem "App Wizard" erzeugten Programmgerüst wird der "Device context"-Pointer als Parameter der Methode **OnDraw** der Ansichtsklasse übergeben).

Der Konstruktor legt die gesamte Zeichenfläche als "Current viewport" fest (das Geräte-Koordinatensystem liegt in der linken oberen Ecke des Viewports), mit der Methode **CGI::stcvp_gi** können ein beliebiges rechteckiges Teilgebiet der Zeichenfläche zum "Current viewport" erklärt und ein anderes Viewport-Koordinatensystem eingestellt werden.

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

- durch die Angabe der **Geräte-Koordinaten seiner linken oberen Ecke** (bezüglich der linken oberen Ecke der Zeichenfläche) und
- die Angabe seiner **Breite und Höhe** (ebenfalls in Geräte-Koordinaten) definiert wird.
- Der Rechteck-Bereich des Viewports kann wahlweise die Ausgabe der nachfolgenden Zeichenaktionen begrenzen (Standard-Einstellung) und wird damit zum **"Clipping-Bereich"**.
- Für die **CGI**-Methoden, die mit "Viewport Coordinates" arbeiten, wird bei der Definition des Viewports mit **CGI::stcvp_gi** eins von **5 möglichen Koordinatensystemen** festgelegt.

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 **CGI::stcvp_gi** übergeben werden, sie werden als **private**-Daten in der Klasse gespeichert.

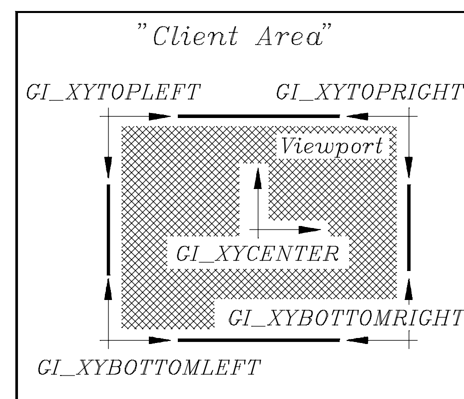


Viewport in der Zeichenfläche

Das fünfte Argument, das **CGI::stcvp_gi** übergeben werden kann (Typ des Viewport-Koordinatensystems), ist optional, weil in der Klassen-Deklaration für diesen Parameter der Default-Wert **GI_XYTOPLEFT** (Koordinaten-Ursprung in der linken oberen Ecke des Viewports) vorgesehen ist.

Es stehen fünf Viewport-Koordinatensysteme zur Wahl, der Koordinaten-Ursprung liegt entweder in einer Ecke des Viewports oder in der Viewport-Mitte. In der Klasse **CGI** sind dafür sinnvolle Enumeration-Konstanten definiert. Die Abbildung unten rechts zeigt die Lage und die Richtung der Achsen für die Koordinatensysteme.

Der Pointer auf den "Device context" ermöglicht mit den **CDC**-Methoden der "Microsoft Foundation Classes" die Änderung aller gewünschten Einstellungen (Farben, Linientypen, ...) und das Zeichnen mit den dafür verfügbaren **CDC**-Methoden. Alternativ dazu können die hier vorgestellten **CGI**-Methoden verwendet 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:



Viewport-Koordinatensysteme

Sowohl die CDC-Methode

```
BOOL CDC::LineTo (int x , int y)
```

als auch die CGI-Methode

```
void CGI::vline_gi (int xv , int xv)
```

erwarten zwei **int**-Werte, die Koordinaten des Zielpunktes. Die **CDC**-Methode **LineTo** bezieht diese auf das im Konstruktor der Klasse **CGI** eingestellte Koordinatensystem **MM_TEXT**, während **CGI::vline_gi** die Koordinaten auf das mit dem Aufruf von **CGI::stcvp_gi** eingestellte Koordinatensystem bezieht.

2.1.2 Programm viewport.cpp

Das Beispiel-Programm **viewport.cpp** demonstriert die typische Bearbeitung der Botschaft **WM_PAINT** (in der Methode **CMainFrame::OnPaint**) mit den CGI-Methoden und das CGI-Viewport-Konzept. Da alle in diesem Manual beschriebenen Programme die gleiche einfache Grundstruktur haben, wird dieses erste Programm etwas ausführlicher beschrieben:

Die (nicht mit dem "App wizard" erzeugten) Programme bestehen in der Regel aus einer Programm-Datei (mit der Extension **.cpp**, hier: **viewport.cpp**), einer Header-Datei (mit der Extension **.h**, hier: **viewport.h**) und gegebenenfalls einer (mit "App studio" erzeugten) Ressourcen-Datei (mit der Extension **.rc**, hier: **viewport.rc**), zu der dann auch noch die Datei **resource.h** gehört (nicht alle Beispiel-Programme benutzen Ressourcen, dann entfallen die beiden letztgenannten Dateien).

In der Header-Datei werden immer mindestens zwei Klassen deklariert, eine von **CWinApp** abgeleitete "Applications class" (hier: **class CViewportApp : public CWinApp**) und eine Klasse für das Hauptfenster (hier: **class CMainFrame : public CFrameWnd**). Gegebenenfalls finden sich in der Header-Datei weitere Klassen-Deklarationen (z. B. für Dialog-Klassen oder Klassen für das Graphik-Modell). Nachfolgend wird die Header-Datei **viewport.h** gelistet:

```
/*****************************************************************
 *  Demonstrationsprogramm VIEWPORT.CPP, Header-Datei VIEWPORT.H  *
 *  Autor: J. Dankert                                             *
 *****************************************************************/
#include "..\class\cgiw.h"
class CViewportApp : public CWinApp
{
   public:
       virtual BOOL InitInstance () ;
} ;
class CMainFrame : public CFrameWnd
{
   private:
       int  m_vpclip ;
   public:
       CMainFrame () ;
   protected:
       afx_msg void OnPaint   () ;
       afx_msg void OnClipOn  () ;
       afx_msg void OnClipOff () ;
       DECLARE_MESSAGE_MAP    ()
} ;
```

- Alle Programme, die mit den **CGIW**-Klassen arbeiten, müssen die Header-Datei **cgiw.h** einbinden. Die include-Anweisung wird in der Header-Datei des Beispiel-Programms plaziert, weil häufig in den dort deklarierten Klassen bereits auf Deklarationen aus **cgiw.h** Bezug genommen wird. Weil **cgiw.h** die MFC-Header-Datei **afxwin.h** inkludieren muß, braucht diese nicht noch einmal zusätzlich in die Beispiel-Programme eingebunden zu werden.
- In der "Applications class" **CViewportApp** wird nur die virtuelle **CWinApp**-Methode **InitInstance** überschrieben, mit der das Hauptfenster erzeugt und sichtbar gemacht wird. Dies geschieht dadurch, daß eine globale Instanz von **CViewportApp** erzeugt wird (**theApp**). Dieses Programmstück ist für alle Beispiel-Programme dieses Manuals gleich (man findet es hier in der Datei **viewport.cpp**):

```
CViewportApp  theApp ;
BOOL CViewportApp::InitInstance ()
{
   m_pMainWnd = new CMainFrame ;
   m_pMainWnd->ShowWindow (m_nCmdShow) ;
   m_pMainWnd->UpdateWindow () ;
   return TRUE ;
}
```

- Im Gegensatz zur "Applications class" enthält die "Hauptfenster-Klasse" die wichtigsten Daten und Methoden, die individuell auf das zu behandelnde Problem zugeschnitten sind. Hier sind es die Komponente **m_vpclip**, mit der gesteuert wird, ob die Viewports mit oder ohne "Clipping" arbeiten, natürlich der Konstruktor **CMainFram::CMainFrame** und die Methoden, die als Reaktionen auf Botschaften an das Fenster ausgeführt werden sollen.

Die Zuordnung der Botschaften zu den Methoden erfolgt nach dem Konzept der "Message maps" (ausführlich erläutert z. B. im Abschnitt 13.3 im Teil 4 von "J. Dankert: C und C++ für UNIX, DOS und MS-Windows 3.1/95/NT"). Deshalb muß in der Deklaration der Fensterklasse das Makro **DECLARE_MESSAGE_MAP ()** angesiedelt werden. Der Code der "Message maps" wird von den entsprechenden Makros erzeugt, die in der **cpp**-Datei untergebracht werden, in **viewport.cpp** findet man:

```
BEGIN_MESSAGE_MAP (CMainFrame , CFrameWnd)
   ON_WM_PAINT ()
   ON_COMMAND  (ID_ENDE     , OnClose)
   ON_COMMAND  (ID_CLIP_ON  , OnClipOn)
   ON_COMMAND  (ID_CLIP_OFF , OnClipOff)
END_MESSAGE_MAP ()
```

Hier soll also auf die Botschat **WM_PAINT**, für die eine Methode mit dem Namen **OnPaint** zu schreiben ist, und drei **WM_COMMAND**-Botschaften reagiert werden. Den (in **resource.h** zu findenden) Identifikatoren (hier **ID_ENDE**, **ID_CLIP_ON** und **ID_CLIP_OFF**), die zu entsprechenden Menü-Angeboten gehören, werden Methoden zugeordnet (hier gewählt: **OnClose**, **OnClipOn** und **OnClipOff**).

Der Konstruktor sollte alle Datenelemente der Klasse initialisieren und mit dem Aufruf von **Create** das Fenster erzeugen, das im Konstruktor der "Applications class" dann (mit **ShowWindow**) auf den Bildschirm gebracht wird. Neben den Default-Argumenten werden an **Create** die Fenster-Überschrift (hier: "Programm VIEWPORT") und die Zuordnung der Menü-Ressource übergeben (hier: **IDR_MENU1**, siehe ebenfalls **resource.h**, dieser Identifikator wurde im "App studio" festgelegt):

```
CMainFrame::CMainFrame ()
{
   m_vpclip = 1 ;
   Create (NULL , "Programm VIEWPORT" , WS_OVERLAPPEDWINDOW ,
           rectDefault , NULL , MAKEINTRESOURCE (IDR_MENU1)) ;
}
```

Während für das Beenden des Programms (Reaktion auf die über das Menü ausgelöste **WM_COMMAND**-Botschaft mit dem Identifikator **ID_ENDE**) die von **CWnd** geerbte Methode **OnClose** aufgerufen wird, müssen die Methoden **OnClipOn** und **OnClipOff** (Namen wurden willkürlich gewählt) in der Klasse **CMainFrame** implementiert werden. Sie schalten jeweils die Variable **m_vpclip** um (von 0 auf 1 bzw. umgekehrt) und lösen über **Invalidate** die Botschaft **WM_PAINT** aus:

```
void CMainFrame::OnClipOn ()
{
   m_vpclip = 1  ;
   Invalidate () ;
}
void CMainFrame::OnClipOff ()
{
   m_vpclip = 0  ;
   Invalidate () ;
}
```

WM_PAINT wird außerdem u. a. beim Programmstart und bei jeder Änderung der Fenstergröße ausgelöst. Die zugehörige Methode **CMainFrame::OnPaint** erledigt die wesentliche Arbeit des Programms.

2.1.3 CMainFrame::OnPaint im Programm viewport.cpp

Die ersten ausführbaren Anweisungen von **CMainfram::OnPaint** sind typisch für das Bearbeiten der **WM_PAINT**-Botschaft unter Benutzung der **CGIW**-Klassen: Nach dem Erzeugen eines "Device contextes" (Instanz der Klasse **CPaintDC**, hier: **dc**, dem Konstruktor wird der **this**-Pointer der **CMainFrame**-Instanz übergeben, die den "Device context" anfordert) wird eine Instanz der Klasse **CGI** erzeugt (hier: **gi**). Beide Objekte "sterben" automatisch beim Verlassen von **OnPaint**.

Weil die **CGI**-Instanz hier mit einem Konstruktor erzeugt wird, der die Abmessungen des Fensters erwartet, müssen diese vorab ermittelt werden, was mit **GetClientRect** erledigt wird:

```
CPaintDC  dc (this) ;

CRect           rect  ;
GetClientRect (&rect) ;
int  cxClient = rect.Width  () ;
int  cyClient = rect.Height () ;

CGI gi (&dc , cxClient , cyClient , m_vpclip) ;
```

Nach dem Erzeugen eines CGI-Objektes **gi** ist zunächst die gesamte "Client area" auch "Current viewport". In einer doppelten Schleife werden danach Viewports erzeugt, die jeweils nur ein Achtel der "Client area" erfassen und für die zunächst (das bei der Konstruktion von **gi** angegebene) "Clipping an den Viewporträndern" eingestellt ist. Mit **CGI::stcvp_gi** werden die Viewports erzeugt, für die (5. Argument) mit **gi.GI_XYCENTER** das Koordinatensystem in die Viewport-Mitte gelegt wird. Die mit **vmove_gi** und **vline_gi** ausgeführten Zeichenaktionen beziehen sich dann auf dieses Koordinatensystem.

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, der Kreis selbst wird nicht gezeichnet). In zwei Zeilen werden jeweils vier Viewports mit der gleichen Zeichnung gefüllt, dabei ist das Verhältnis des Kreisdurchmessers zur kleineren Viewport-Abmessung in den Viewports der oberen Zeile kleiner als 1, so daß die Zeichnungen in die Viewports "passen", in der unteren Viewportzeile ist dieses Verhältnis größer als 1, so daß das Clipping an den Viewport-Rändern deutlich wird.

Nachfolgend wird der Quellcode der zum Programm **viewport.cpp** gehörenden Methode **CMainFrame::OnPaint** komplett gelistet:

```
void CMainFrame::OnPaint ()
{
   int       nViewpx = 4 , nViewpy = 2 , nPoints = 15 ,
             Radius , ix , iy , i , j , xs , ys ;
   double    QuotDiamLowDist = 0.9 , dPhi ;

   CPaintDC  dc (this) ;

   CRect           rect  ;
   GetClientRect (&rect) ;
   int  cxClient = rect.Width  () ;
   int  cyClient = rect.Height () ;

   CGI gi (&dc , cxClient , cyClient , m_vpclip) ;

   for (iy = 0 ; iy < nViewpy ; iy++)
   {
     for (ix = 0 ; ix < nViewpx ; ix++)
     {
        gi.stcvp_gi (int (cxClient * ix / nViewpx) ,
                     int (cyClient * iy / nViewpy) ,
                     int (cxClient / nViewpx) ,
                     int (cyClient / nViewpy) , gi.GI_XYCENTER) ;
        // ... definiert den "Current viewport" mit einem (Pixel-)Viewport-
        //     Koordinatensystem in Viewportmitte
        Radius = (cxClient / nViewpx > cyClient / nViewpy) ?
                   cyClient / nViewpy : cxClient / nViewpx ;
        Radius = int (Radius * QuotDiamLowDist / 2) ;
        dPhi   = GI_PI * 2. / nPoints ;                 // pi * 2 / nPoints

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

           for (j = i + 1 ; j < nPoints ; j++)
           {
              gi.vmove_gi (xs , ys) ;
              gi.vline_gi (int (Radius * cos (dPhi * j) + .5)  ,
                           int (Radius * sin (dPhi * j) + .5)) ;
           }
        }
     }
     QuotDiamLowDist = 1.5 ;
   }
}
```

Man beachte, daß den mit **CGI::vmove_gi** und **CGI::vline_gi** ausgeführten Zeichenaktionen für jeden Viewport die gleichen Koordinaten übergeben werden. Die Transformation der Koordinaten auf die zum eingestellten Viewport gehörenden Positionen in der "Client area" wird innerhalb der **CGI**-Methoden vorgenommen.

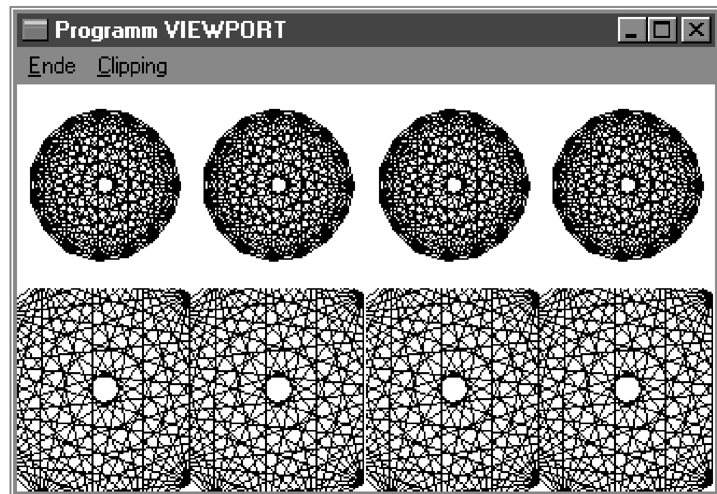
Die ersten vier Parameter der Methode **CGI::stcvp_gi** sind die Pixel-Koordinaten des in der linken oberen Ecke der "Client area" angesiedelten **MM_TEXT**-Koordinatensystems und die

Breite bzw. Höhe des Viewports. Der in der Header-Datei **cgiw.h** zu findende Prototyp benutzt die im Bild des Abschnitts 2.2.1 angegebenen Bezeichnungen:

```
void stcvp_gi (int pulx  , int puly   ,
               int width , int height , int cstype = GI_XYTOPLEFT) ;
```

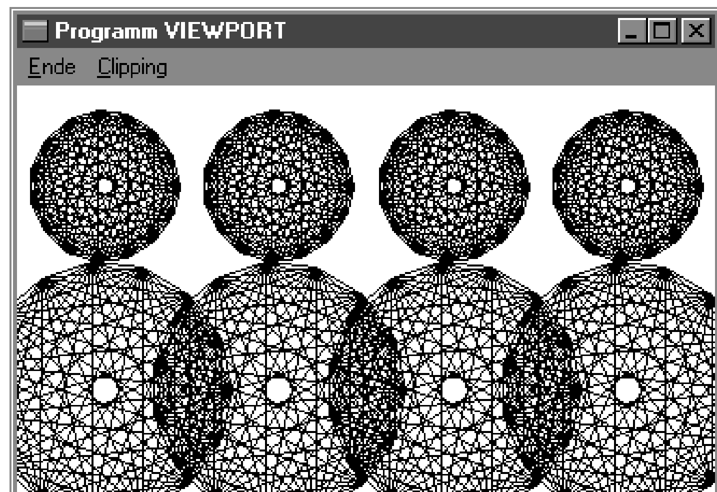
Wenn der 5. Parameter weggelassen wird, liegt das Viewport-Koordinatensystem in der linken oberen Ecke (im Programm **viewport.cpp** wird auf dieser Position als Argument **gi.GI_XYCENTER** übergeben).

Das nebenstehende Bild zeigt den Zustand nach dem Programmstart: Die vier Rosetten in der oberen Reihe passen jeweils in ihren Viewport, während die Rosetten in der unteren Reihe zu groß sind (das Verhältnis von Rosetten-Durchmesser zur kleineren Viewport-Abmessung wird vor dem Zeichnen dieser Rosetten am Ende der Schleife auf **1.5** gesetzt), so daß an den Rändern der Viewports "geclippt" wird.



An den Viewport-Rändern wird "geclippt"

Über das Popup-Menü-Angebot **Clipping** kann **Clipping aus** gewählt werden. Das CGI-Objekt **gi** wird dann in **CMainFrame::OnPaint** mit dieser Option konstruiert, und die Ausgabe des Programms zeigt das Zeichnen über die Viewportränder hinaus (nebenstehendes Bild).



"Clipping" ausgeschaltet

Dieses "Clipping"-Verhalten der Viewports gilt nicht nur für die hier verwendeten "v...-Routinen" (Zeichnen mit "Viewport coordinates") sondern auch für das Zeichnen mit allen anderen "zeichnenden Routinen" (mit "User coordinates", "World coordinates", auch bei eingestellten Transformationen) der **CGIW**-Library.

2.2 Problembezogene Koordinatensysteme

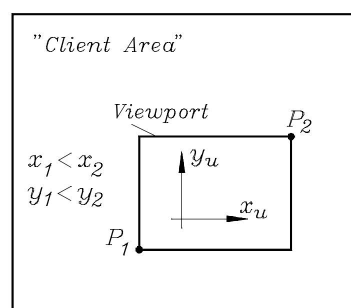
Als Ergänzung zu den (ausschließlich mit **int**-Werten arbeitenden) Koordinatensystemen des Windows-GDI und den im vorigen Abschnitt vorgestellten "Viewport coordinates" der Klasse **CGI** werden problembezogene Koordinaten definiert, in der **CGIW**-Library bezeichnet als

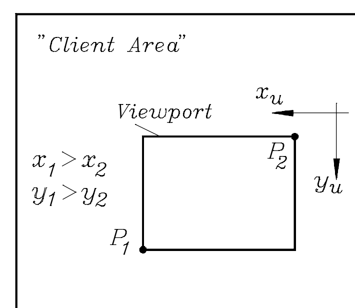
"User coordinates":

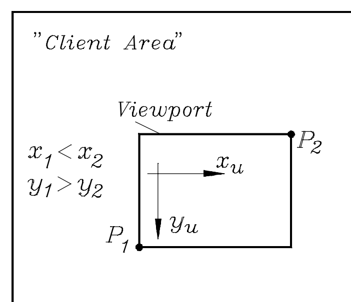
Für den "Current viewport" (festgelegt durch **CGI::stcvp_gi** bzw. **CGI::CGI**) wird ein Koordinatensystem durch Angabe der Koordinaten zweier Punkte $\boldsymbol{P_1}$ und $\boldsymbol{P_2}$ definiert. $\boldsymbol{P_1}$ ist stets die **LINKE UNTERE**, $\boldsymbol{P_2}$ die **RECHTE OBERE** Ecke 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").

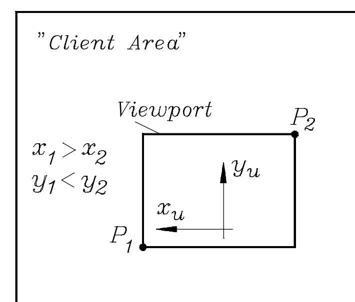
- ♦ Die "User coordinates" werden mit **double**-Werten definiert, alle Zeichenroutinen, die mit diesem Koordinatensystem arbeiten (die Namen dieser CGI-Methoden beginnen mit **u**), erwarten **double**-Koordinaten.
- ♦ Für $\boldsymbol{P_1}$ und $\boldsymbol{P_2}$ dürfen beliebige Werte festgelegt werden. 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 $\boldsymbol{P_1}$ und $\boldsymbol{P_2}$ alle vier Kombinationen realisiert werden (wenn die beiden $\boldsymbol{P_1}$-Koordinaten kleiner sind als die entsprechenden $\boldsymbol{P_2}$-Koordinaten, zeigt die $\boldsymbol{x}$-Achse nach rechts, und die $\boldsymbol{y}$-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...-Routinen" 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 $\boldsymbol{P_1}$ und $\boldsymbol{P_2}$) außerhalb des "Current viewport" liegt.









2.2.1 "User coordinates" mit anisotroper Skalierung

Mit der **CGI**-Methode **stuca_gi**, der 4 oder 5 **double**-Argumente übergeben werden können, 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. Ihr Prototyp ist in **cgiw.h** folgendermaßen festgelegt:

```
void stuca_gi (double p1xu , double p1yu ,
               double p2xu , double p2yu , double pmarg = 0.) ;
```

Diese anisotrope Skalierung 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 der anisotropen Skalierung beide Abmessungen des Viewports voll ausgenutzt, **CGI::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, oder er wird weggelassen. 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" nur durch die Viewport-Grenzen festgelegt wird.

Das Beispiel-Programm **usrcoor1.cpp** demonstriert das Arbeiten mit "User coordinates" (Methode **CGI::stuca_gi** für das Definieren der Koordinaten und die Methoden **CGI::umove_gi** und **CGI::uline_gi**, die mit diesen Koordinaten arbeiten). Es wird die mathematische Funktion

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

dargestellt. Dabei wird in 8 Viewports gearbeitet, so daß alle Varianten der Koordinatenrichtungen und auch das Vorgeben einer Rand-Einstellung gezeigt werden können. Im nachfolgend gelisteten Quellcode der zum Programm **usrcoor1.cpp** gehörenden Methode **CMainFram::OnPaint** wird das **CGI**-Objekt mit einem anderen Konstruktor als im Programm **viewport.cpp** erzeugt:

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

Diesem Konstruktor wird der Pointer auf das aktuelle Fenster übergeben (1. Argument), so daß er sich die Fensterabmessungen selbst besorgen kann (über ein 3. Argument kann auch mit diesem Konstruktor das "Clipping" eingestellt werden, hier wird das Default-Argument **GI_CLIPPING** durch Weglassen akzeptiert). Der Gebrauch dieses Konstruktors ist für den Programmierer etwas einfacher. Wenn trotzdem (wie im Programm **usrcoor1.cpp**) die Abmessungen der "Client area" benötigt werden, kann man diese über die **CGI-inline**-Methoden **gtcxv_gi** bzw. **gtcyv_gi** erfragen.

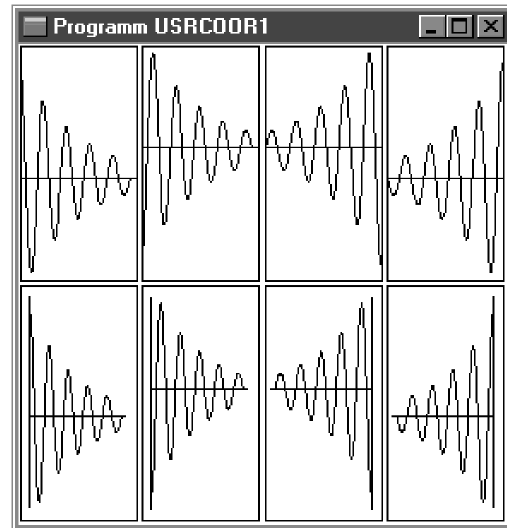
CMainFrame::OnPaint des Programms **usrcoor1.cpp**:

```
void CMainFrame::OnPaint ()
{
   int       nViewpx = 4  , nViewpy = 2 , ix , iy , width , height;
   double    pmarg   = 0. , x ;
   CPaintDC  dc (this) ;
   CGI       gi (this , &dc) ;          // ... ermittelt selbstaendig die
                                        //     Abmessungen der "Client area"
   int  cxClient = gi.gtcxv_gi () ;
   int  cyClient = gi.gtcyv_gi () ;
   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++)
     {
        gi.stcvp_gi ((int) (cxClient * ix / nViewpx) + 1 ,
                     (int) (cyClient * iy / nViewpy) + 1 ,
                     width , height , gi.GI_XYBOTTOMLEFT) ;
        // ... definiert den "Current viewport" mit einem Viewport-
        //     Koordinatensystem in der linken unteren Ecke
        gi.vrect_gi (0 , 0 , width - 1, height - 1) ;
        // ... zeichnet einen Rahmen um den Viewport
        if (ix == 0) gi.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), bei ix=0 (obere
        //     Reihe) gilt pmarg=0. (kein Rand), danach pmarg=10.
        if (ix == 1) gi.stuca_gi ( 0. ,  4.5 , 10.5 , -3.5 , pmarg) ;
        if (ix == 2) gi.stuca_gi (10. ,  4.5 ,  0.  , -3.5 , pmarg) ;
        if (ix == 3) gi.stuca_gi (10. , -3.5 ,  0.  ,  4.5 , pmarg) ;
        // ... und in den vier Viewports einer Reihe sind alle vier
        //     Kombinationen fuer die Richtungen der beiden
        //     Koordinatenachsen realisiert
        gi.umove_gi ( 0.  ,  4.5) ;
        gi.uline_gi ( 0.  , -3.5) ;   // Horizontale Linie als x-Achse
        gi.umove_gi ( 0.  ,  0. ) ;
        gi.uline_gi (10.5 ,  0. ) ;   // Vertikale Linie als y-Achse
        gi.umove_gi ( 0.  ,  4. ) ;   // Startpunkt fuer Funktionsgraph
        for (x = 0.05 ; x <= 10. ; x += 0.05)
        {
          //                      -x/5
          //    Funktion  y = 4 e   cos 3x :
          gi.uline_gi (x , 4. * exp (-.2 * x) * cos (3. * x)) ;
        }
     }
     pmarg = 10. ;                    // ... untere Reihe mit 10% Rand
   }
}
```

Die mit **CGI::stuca_gi** eingestellten Koordinaten für die Eckpunkte wurden hier (mit der Kenntnis der Grenzen der Funktionswerte für das dargestellte Intervall) fest einprogrammiert, was natürlich i. a. weder gut noch sinnvoll möglich ist. Unter den **CGIW**-Methoden finden sich auch mehrere, die den Programmierer bei der Ermittlung sinnvoller Grenzwerte unterstützen, die in nachfolgenden Beispiel-Programmen besprochen werden.

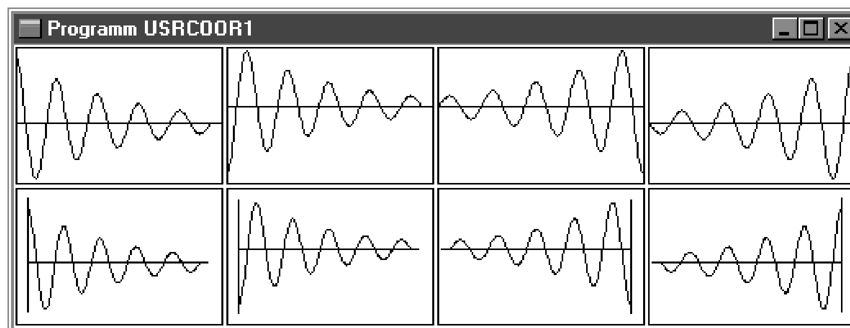
Die dargestellten Funktionen in den Viewports folgen in beiden Richtungen jeder Veränderung der Größe der Fenster-Abmessungen, die auch die Viewport-Abmessungen verändern.

Die nebenstehende Abbildung zeigt die Ausgabe des Programms **usrcoor1.cpp**: 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 $\boldsymbol{P}_1$ und $\boldsymbol{P}_2$, mit denen die Koordinaten definiert werden, mit den Viewport-Ecken identisch sind.



Funktions-Darstellung in "hohen" Viewports

Die Zeichnungen in den Viewports passen sich jeder Änderung der Zeichenfläche in beiden Richtungen an, so daß die Viewportfläche jeweils optimal ausgenutzt wird. Die nachfolgende Darstellung in einem Window, das wesentlich breiter als hoch ist, zeigt die entsprechend verzerrten Funktionen:

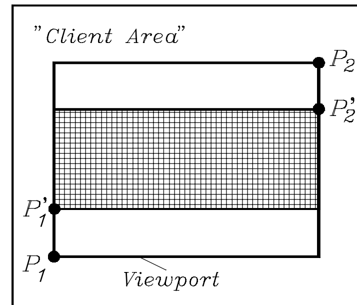


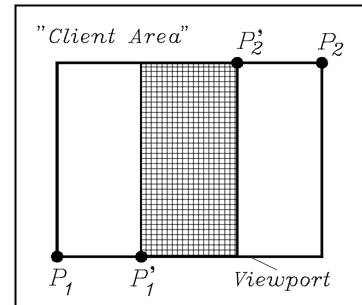
Darstellung der Funktionen in "breiten" Viewports

2.2.2 "User coordinates" mit isotroper Skalierung

Die **CGI**-Methode **stuci_gi** erwartet wie die im vorigen Abschnitt vorgestellte Methode **CGI::stuca_gi** 4 oder 5 **double**-Argumente, die auch die gleiche Bedeutung haben. Die beiden Punkte $\boldsymbol{P}_1$ und $\boldsymbol{P}_2$, 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 Plazierung des Zeichenbereichs im Viewport in Abhängigkeit davon, welche Richtung durch den von den "User coordinates" abzubildenden Bereich besser auszufüllen ist.



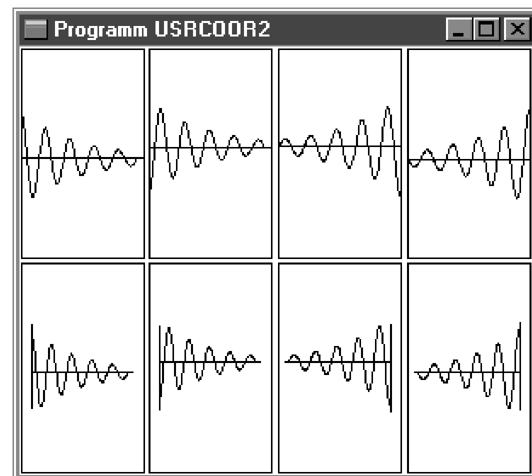


Mögliche Verschiebungen der Eckpunkte bei isotroper Skalierung

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.

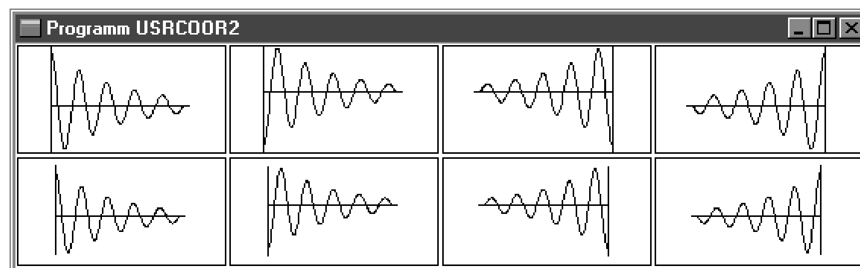
Das Beispiel-Programm **usrcoor2.c** unterscheidet sich vom Programm **usrcoor1.cpp**, das im vorigen Abschnitt vorgestellt wurde, nur dadurch, daß die "User coordinates" isotrop mit **CGI::stuci_gi** definiert werden. Deshalb wird hier kein Listung angegeben.

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, so daß die Funktionen unverzerrt dargestellt werden.



Isotrope Skalierung in "hohen" Viewports: Viewport-Bereich bleibt oben und unten zum Teil ungenutzt

Auch bei der Ausgabe des Programms **usrcoor2.cpp** enthält die obere Viewport-Reihe "randlose" Viewports, während in den Viewports der unteren Reihe ein Rand von 10% eingestellt ist, und es sind jeweils die vier möglichen Richtungen der Koordinaten verwendet worden.



Isotrope Skalierung in "breiten" Viewports