

7 Grafik

7.1 Der Device-Context

Das so genannte Graphic Device Interface (GDI) ist die Grafikschnittstelle von Windows. Vom Prinzip her ist GDI eine hardwareunabhängige Schnittstelle, d.h. die Applikationen brauchen sich nicht um die Ansteuerung der eigentlichen Hardware kümmern; dies wird von den Treibern durchgeführt.

Windows unterstützt sowohl rasterorientierte als auch vektororientierte Ausgabegeräte. In die erste Klasse fallen etwa alle derzeit üblichen VGA-Karten bzw. deren Derivate sowie Nadel-, Laser- und Tintenstrahldrucker; Beispiele für die zweite Klasse sind Plotter.

Die konkrete Realisierung erfolgt über einen so genannten Device-Context (DC). Dies ist eine Struktur, die als Bindeglied zwischen Hardware und Applikation angesehen werden kann.

Um Grafikausgaben durchführen zu können, muss eine Applikation einen Device Context anlegen und mit dem gewünschten Ausgabemedium (meist ein Fenster oder der aktive Drucker) verbinden. Danach können API-Funktionen zum Zeichnen von geometrischen Objekten, Texten, etc. aufgerufen werden, denen ein Handle auf den Device-Context übergeben wird. Nach erfolgter Ausgabe muss ein Device-Context wieder geschlossen werden. Der Device-Context stellt also für die Anwendung ein „universales“ Ausgabegerät dar.

Die MFC-Bibliothek stellt nun für die bequemere Handhabung eine CDC-Klasse zur Verfügung, die einen Handle auf einen Device-Context kapselt. Die Grafikoperationen selbst sind als Methoden implementiert. Alle vom API bekannten Funktionen zum Zeichnen, Selektieren von GDI-Objekten, Palettenverarbeitung, Clipping, Skalieren von Viewports und Windows sind vorhanden.

Ein CDC-Objekt enthält eigentlich zwei DC-Handles, nämlich `m_hDC` und `m_hAttribDC`. Normalerweise verweisen beide auf denselben DC. Für spezielle Anwendungen - etwa für eine Druckvorschau - kann es sinnvoll sein, einen DC für die Grafikausgaben und den anderen für die Abfrage von Device-Informationen zu verwenden. Für den zweiten DC wird man dann im allgemeinen nur einen Information-Context anfordern (`CreateIC()`).

Ein CDC-Objekt muss wie immer in zwei Schritten erzeugt werden: Zuerst wird es per Konstruktor instanziiert und danach wird mit der Methode `CreateDC()` der eigentliche DC-Handle erzeugt.

```
virtual BOOL CDC::CreateDC(LPCSTR lpszDriverName, LPCSTR lpszDeviceName,
LPCSTR lpszOutput, const void FAR *lpInitData)
```

Dies ist die allgemeinste Funktion zum Anlegen eines DC.

| | |
|-----------------------|--|
| lpszDriverName | Dateiname des Treibers (z.B. "HPPCL5E") |
| lpszDeviceName | Bezeichnung des Gerätes, das der Treiber öffnen soll (z.B. "HP Laserjet 4P/4MP") |
| lpszOutput | Bezeichnung des Ausgabeports (z.B. "LPT1:") |
| lpInitData | Zeiger auf einen DEVMODE-Struct mit speziellen Initialisierungsdaten. NULL, falls Defaultwerte gewünscht sind. |
| Returnwert | TRUE, falls erfolgreich, sonst FALSE |

Im Destruktor wird automatisch die Methode `DeleteDC()` aufgerufen, so dass ein expliziter Aufruf nicht unbedingt nötig ist.

Der DC muss aber nicht immer auf solch komplizierte Weise erzeugt werden: Da DCs zumeist für Fenster benötigt werden, existieren folgende von CDC abgeleitete Klassen, die mit speziellen Konstruktoren und Destruktoren gleich einen Fenster-DC erzeugen und entfernen:

`CClientDC` wird verwendet, um einen DC für die Client-Area eines Fensters anzulegen. Der Konstruktor ruft `::GetDC()` auf und benötigt als Parameter den Zeiger auf das gewünschte Fenster (`this`, falls es das aktuelle Fenster ist). Der Destruktor ruft `::ReleaseDC()` auf.

`CWindowDC` wird verwendet, um einen DC für ein ganzes Fenster anzulegen. Der Konstruktor ruft `::GetWindowDC()` auf und benötigt als Parameter den Zeiger auf das gewünschte Fenster (`this`, falls es das aktuelle Fenster ist). Der Destruktor ruft `::ReleaseDC()` auf. Der `CWindowDC` wird nur in Spezialfällen benötigt, z.B. wenn die Non-Client-Area eines Fensters gezeichnet werden soll.

`CPaintDC` wird für die Behandlung der Nachricht `WM_PAINT` benötigt. Der Konstruktor ruft `::BeginPaint()` auf und benötigt als Parameter den Zeiger auf das gewünschte Fenster (`this`, falls es das aktuelle Fenster ist). Der Destruktor ruft `::EndPaint()` auf.

`CMetaFileDC` schreibt die Grafikausgaben in eine Datei, ein sog. MetaFile (.WMF). Der DC muss explizit mit der Methode `Create()` erzeugt und mit `Close()` geschlossen werden.

Das folgende Beispiel zeigt die Anwendung des `CClientDC` in der Behandlung von `WM_LBUTTONDOWN`.

```
void CMyView::OnLButtonDown(UINT nFlags, CPoint point)
{
    CClientDC dc(this);
    dc.Ellipse(point.x-20, point.y-20, point.x+20, point.y+20);
}
```

Bei Verlassen der Funktion wird automatisch der Destruktor von `CClientDC` aufgerufen, der den Device-Context wieder freigibt. Insgesamt also ist die Handhabung der Klasse `CDC` und ihren Derivaten wesentlich bequemer als die entsprechenden API-Funktionen.

7.2 GDI-Objekte

Für die Ausgabe von Grafikelementen können natürlich auch noch etliche zusätzliche Zeichenattribute verwendet werden. So können etwa bei Linien noch die Stärke, der Linientyp und die Farbe bestimmt werden. Für Textausgaben ist die Schriftart, die Schriftausrichtung, die Farbe sowie die „Hintergrundfarbe“ der Schrift wichtig.

Um nun etwa für die Ausgabe einer Linie die Übergabe etlicher Parameter an die Funktion `CDC::LineTo()` zu verhindern, werden die Zeichenattribute im DC extra gespeichert. Zum Setzen dieser Attribute existieren eigene Funktionen im CDC-Objekt. Es müssen also erst die Attribute in den Device-Context gesetzt werden und nachher die gewünschten Grafikausgaben durchgeführt werden.

Um eine hohe Geschwindigkeit bei der Ausgabe zu erreichen, ist daher auch generell empfehlenswert, die Grafikausgaben nach gemeinsamen Zeichenattributen zu „sortieren“, um häufiges Wechseln der Attribute zu verhindern.

Prinzipiell kann man in der Klasse `CDC` zwei Arten von Funktionen zu Setzen der Attribute unterscheiden:

Einerseits existieren zum Setzen der Hintergrundfarbe, der Textausrichtung, der Textfarbe, der aktuellen Koordinatensystems, etc. eigene Funktionen, die einfach nur mit bestimmten Parametern aufgerufen werden müssen.

Für manche andere Zeichenattribute muss aber zunächst ein Umweg begangen werden: Zum Setzen der Schriftart, der Füllung und Kontur von Objekten etwa müssen so genannte GDI-

Objekten erzeugt werden. Dies sind eigene Objekte, die separat erzeugt werden und nachher in den DC eingesetzt werden.

Sehen wir uns zunächst im Überblick die vorhandenen GDI-Objekte an:

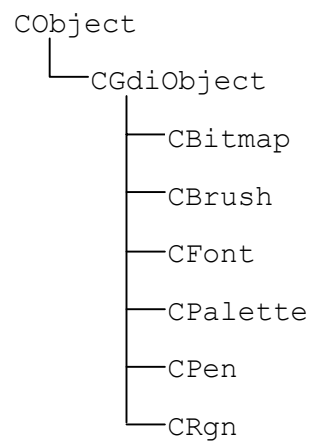


Abbildung 2 - Die GDI-Objekte

| | |
|------------|---|
| CGDIObject | Dies ist die Basisklasse aller GDI-Objekte. Diese Klasse speichert auch den eigentlichen Handle auf das zugeordnete „echte“ Windows-GDI-Objekt. Die hier definierten Funktionen dienen daher auch hauptsächlich der Bearbeitung dieses Handles. |
| CPen | Bestimmt die Farbe, Dicke und den Typ (durchgezogen, strichliert, etc.) aller Linien und Umrandungen von Objekten. |
| CBrush | Die Farbe und das Muster von gefüllten Objekten. Es ist auch möglich, transparente Brushes zu erzeugen. |
| CFont | Die Schriftart, die bei allen Textausgaben eingesetzt wird. |
| CBitmap | Eine Bitmap ist zumeist ein Bild, das entweder von den Ressourcen oder von einer Datei geladen wird. Im Prinzip ist aber eine Bitmap nur ein Speicherbereich. Es können auch Bitmaps zur Laufzeit erstellt werden, indem mit einem sogenannten Memory-Device-Context Grafikausgaben gemacht können. |
| CPalette | Für Grafikausgaben, die mehr als die fixen 16 Standardfarben benötigen, muß auf eine Palette zurückgegriffen werden. Da viele Bildschirmkarten nur |

| | |
|------|---|
| | in der Lage sind, 256 Farben (von meist vielen tausenden Farben) gleichzeitig darzustellen, muß eine Auswahl dieser Farben getroffen werden. Dabei entsteht auch das Problem, daß bei einer Änderung der Palette natürlich auch andere Applikationen betroffen sind. Das CPalette-Objekt enthält eine logische Palette mit den gewünschten Farben, die in den DC eingesetzt wird. Der DC kann dann auf Wunsch die gewünschten Farben so in die Systempalette einsetzen, daß möglichst wenige Farben anderer Applikationen betroffen sind. |
| CRgn | Eine „Region“ ist ein Bereich (entweder eine Ellipse oder ein Polygon), der als Maske in den DC eingesetzt werden kann. Alle weiteren Ausgabeoperationen werden dann am Rand dieser Maske abgeschnitten. |

All diese GDI-Objekte müssen wie immer in zwei Schritten erzeugt werden: Zuerst wird das C++-Objekt erzeugt, danach der zugehörige Windows-Handle. Eine Darstellung der konkreten Funktionen folgt ein paar Seiten später.

Ein DC enthält immer genau einen Pen, einen Brush, einen Font, eine Bitmap, eine Region und eine Palette. Darüber übrigen Zeichenattribute, z.B. die aktuelle Hintergrundfarbe oder die Textausrichtung, werden nicht als GDI-Objekte betrachtet und können wie bereits gesagt über eigene Funktionen gesetzt werden.

Um ein GDI-Objekt in den DC einzusetzen, muss die Funktion `CDC::SelectObject()` aufgerufen werden:

```
CPen *CDC::SelectObject(CPen *pPen)
CBrush *CDC::SelectObject(CBrush *pBrush)
virtual CFont *CDC::SelectObject(CFont *pFont)
CBitmap *CDC::SelectObject(CBitmap *pBitmap)
```

Setzt einen neuen Pen, Brush, Font, etc. in den Device Context ein und gibt einen Zeiger auf das bisher eingesetzte Objekt zurück.

Der Name dieser Funktion ist sehr unglücklich gewählt, da der Eindruck entsteht, es könne immer nur ein einziges Objekt in einen DC eingesetzt werden. Tatsächlich wird aber je nach Parameter eine andere Aktion durchgeführt, so dass eigentlich unterschiedliche Bezeichner (z.B. `SelectPen()`, `SelectBrush()`, `SelectFont()`, etc.) treffender gewesen wären. Interessanterweise ist dies für Paletten und Regions so getan worden:

```
int CDC::SelectObject(CRgn *pRgn)
virtual int SelectClipRgn(CRgn *pRgn)
```

Diese beiden Funktionen sind von der Funktionalität identisch: In beiden Fällen wird eine neue Region in den DC eingesetzt. **Achtung: Im Gegensatz zu den anderen Funktionen**

liefert er Returnwert nicht die bisherige Region zurück, sondern nur den Typ (COMPLEX, SIMPLE, etc.)

Der Rückgabewert all dieser Funktionen (außer derjenigen zum Einsetzen der Regions) ist ein Zeiger auf das zuvor im DC enthaltene GDI-Objekt. Dieser Zeiger ist aber nur ein temporärer Zeiger und darf nur vorübergehend verwendet werden! Die MFC-Bibliothek führt eine eigene Liste mit Objekten, die vom Benutzer nicht explizit angelegt wurden (also etwa Objekte, die als Returnwert geliefert werden). Sobald die Nachrichtenqueue leer ist, das Programm also „idle“ ist, werden diese Objekte wieder entfernt. Der Returnwert der Funktion `SelectObject()` sollte also nur kurzfristig in einer Variable gespeichert werden.

Die Destruktoren der GDI-Objekte geben den zugeordneten Handle automatisch wieder frei. Allerdings dürfen sie in keinen Device-Context mehr eingesetzt worden sein. Um das zu erreichen, können vor dem Einsetzen der neuen Objekte die alten in lokalen Variablen gesichert werden und anschließend wieder zurückselektiert werden:

7.2.1.1 Beispiel

```
void CMyView::DrawYellowCircle()
{
    CBrush brushRot(128, 0, 0);
    CPen penGelb(PS_SOLID, 1, RGB(255, 255, 0));
    CPen *pOldPen;
    CBrush *pOldBrush;

    CClientDC dc(this);

    // neue GDI-Objekte einsetzen und Zeiger auf die alten Objekte
    merken:
    pOldPen = dc.SelectObject(&penGelb);
    pOldBrush = dc.SelectObject(&brushRot);

    dc.Ellipse(100, 100, 200, 200);
    // ... eventuell weitere Zeichenoperationen

    // alte GDI-Objekte wieder einsetzen:
    dc.SelectObject(pOldPen);
    dc.SelectObject(pOldBrush);

    // Jetzt können die Destruktoren aufgerufen werden
}
```

Prinzipiell sollte eine Applikation so optimiert werden, dass möglichst selten Attribuswechsel durchgeführt werden müssen, d.h. Zeichenoperationen mit gleichen Attributen sollten hintereinander durchgeführt werden. Werden aber viele Zeichenattribute auf einmal gewechselt werden, ist es relativ mühsam, jedes Mal alle aktuellen Attribute zwischenspeichern. Eine Alternative bieten daher die Funktionen `SaveDC()` bzw. `RestoreDC()`:

```
virtual int CDC::SaveDC()    Sichert den aktuellen Zustand, d.h. sämtliche  
                             Attribute des Device-Context auf einen internen  
                             Stack.
```

Returnwert: 0 bei einem Fehler, sonst eine Zahl, die bei einem Aufruf von `RestoreDC()` verwendet werden kann.

```
virtual BOOL CDC::RestoreDC(int nSavedDC)
```

Holt die mit `SaveDC()` gesicherten Attribute wieder zurück.

nSavedDC: Die Nummer, die von dem korrespondierenden `SaveDC()`-Aufruf zurückgegeben wurde. Falls -1, werden die Attribute des unmittelbar letzten Aufrufs von `SaveDC()` zurückgeholt.

Returnwert: FALSE bei einem Fehler, sonst TRUE.

7.2.1.2 Beispiel

Das folgende Beispiel hat die Funktionalität des vorgehenden, nur werden die GDI-Objekte mit `SaveDC()` / `RestoreDC()` zurückselektiert. Beachten Sie aber, dass `SaveDC()` bzw. `RestoreDC()` sämtliche DC-Attribute speichern, und das ist eine ganze Menge. Die folgende Funktion ist daher etwas langsamer also vorhin gezeigte Variante. Sie benötigt auch etwas mehr Speicherplatz - was aber mittlerweile kein ernstzunehmendes Argument mehr darstellt.

```
void CMyView::DrawYellowCircle()  
{  
    CBrush brushRot(128, 0, 0);
```

```

CPen penGelb(PS_SOLID, 1, RGB(255, 255, 0);

CClientDC dc(this);

// aktuellen DC-Zustand sichern:
dc.SaveDC();

// neue GDI-Objekte einsetzen und Zeiger auf die alten Objekte
merken:
dc.SelectObject(&penGelb);
dc.SelectObject(&brushRot);

dc.Ellipse(100, 100, 200, 200);
// ... eventuell weitere Zeichenoperationen

// alten DC-Zustand wiederherstellen:
dc.RestoreDC(-1);

// Jetzt können die Destruktoren aufgerufen werden
}

```

7.3 CPen und CBrush

7.3.1.1 CPen

Ein Pen bestimmt die Stärke, Farbe und den Stil (durchgezogen, strichliert, etc.) aller Linien und aller Umrandungen von Objekten.

Ein Pen wird am besten per Konstruktor erzeugt.

```
CPen::CPen(int nPenStyle, int nWidth, COLORREF crColor)
```

Erzeugt einen Pen mit bestimmten Attributen. Schlägt die Erzeugung fehl, wird eine `CResourceException` erzeugt.

nPenStyle: Der Stil des Pen. Kann einer der folgenden Werte sein:

| | |
|----------|---------------------|
| PS_SOLID | durchgezogene Linie |
| PS_DASH | strichlierte Linie |
| PS_DOT | punktierte Linie |

| | |
|----------------|---|
| PS_DASHDOT | Punkt-Strich |
| PS_DASHDOTDOT | Strich-Punkt-Punkt |
| PS_NULL | transparente Linie. Bei diesem Wert wird die Farbe ignoriert. |
| PS_INSIDEFRAME | zeichnet die Kontur von Objekten stets innerhalb des umgebenden Rechtecks |

nWidth: Die Breite der Linie, in logischen Koordinaten. Ist dieser Wert jedoch 0, ist die Linie immer genau ein Pixel breit, egal, welches logisches Koordinatensystem derzeit eingestellt ist.

crColor: Die Farbe des Pen

Daneben existiert auch noch ein Defaultkonstruktor:

```
CPen::CPen()           Erzeugt das Objekt, ohne jedoch das zugehörige
                       GDI-Objekt zu erzeugen. Dieses kann
                       nachträglich mit CPen::Attach() mit dem Pen
                       verbunden werden oder mit CPen::CreatePen()
                       erzeugt werden.
```

Die folgenden beiden Beispiele zeigen, wie ein Pen mit Fehlerabfrage erzeugt werden kann:

Methode 1: Erzeugung in zwei Schritten:

```
CPen DickerGelberStift;
if (!DickerGelberStift.CreatePen(PS_SOLID, 10, RGB(255, 255, 0)))
    MessageBox("CreatePen() failed");
```

Methode 2: Erzeugung per Konstruktor mit Exception-Handling:

```
try
{
    CPen DickerGelberStift(PS_SOLID, 10, RGB(255, 255, 0));
}
catch(CResourceException *rex)
{
    MessageBox("CreatePen() failed");
    rex->Delete();
}
```

7.3.1.2 CBrush

Der Brush bestimmt die Füllung eines Objektes. Dies kann eine Farbe, ein Muster oder auch eine transparente Füllung sein.

Neben dem Defaultkonstruktor gibt es noch einige Konstruktoren, mit denen Brushes für Farben oder Muster erstellt werden können:

`CBrush()` Der Defaultkonstruktor. Dieser Konstruktor tut nichts, der Brush muß daher nachträglich erzeugt oder mit einem `HBRUSH`-Handle verbunden werden.

`CBrush (COLORREF crColor)` Dieser Konstruktor erzeugt einen Brush in einer bestimmten Farbe.

`CBrush (int nIndex, COLORREF crColor)`

Erzeugt einen Brush mit einem einfachen Muster.

nIndex: bestimmt das Muster. kann eine der folgenden Konstanten sein:

`HS_BDIAGONAL, HS_CROSS,`
`HS_DIAGCROSS, HS_FDIAGONAL,`
`HS_HORIZONTAL, HS_VERTICAL`

crColor: Bestimmt die Vordergrundfarbe des Musters. Für die Hintergrundfarbe wird die aktuelle Hintergrundfarbe des DC verwendet.

`CBrush (CBitmap *pBitmap)` Erzeugt einen Brush mit einem frei bestimmbareren Muster. `pBitmap` ist ein Zeiger auf eine Bitmap, die das Muster enthält. Diese Bitmap wird beim Füllen einfach nebeneinander gestellt.

Allerdings sind mit diesen Konstruktoren noch nicht alle Möglichkeiten der Brush-Erzeugung erschöpft. Brushes können wie bereits erwähnt auch durchsichtig sein. Die allgemeinste Funktion zur Erzeugung von Brushes ist folgende:

```
BOOL CBrush::CreateBrushIndirect(LPLOGBRUSH lpLogBrush);
```

Mit dieser Funktion ist es möglich, alle Arten von Brushes zu erzeugen. `lpLogBrush` ist ein Zeiger auf die Struktur `LOGBRUSH`, die vorher ausgefüllt werden sollte. `LOGBRUSH` enthält diverse Komponenten, über die der Brush genau definiert werden kann. Allerdings decken die

oben angeführten Konstruktoren ohnehin die meisten Fälle ab. Es folgt daher anhand eines Beispiels nur die Erzeugung eines transparenten Brushes:

```
void CMyView::DrawSomething()
{
    CClientDC dc(this);
    CBrush brTrans, *pOldBrush
    LOGBRUSH logTrans;

    // Transparenten Brush erzeugen:
    logTrans.lbStyle = BS_HOLLOW;
        // definiert einen transparenten Brush
        // statt BS_HOLLOW kann auch BS_NULL verwendet werden
        //(ist dasselbe)

    brTrans.CreateBrushIndirect(&logTrans); // Brush erzeugen

    // jetzt kann der Brush verwendet werden
    pOldBrush = dc.SelectObject(&brTrans);

    // Kreis mit durchsichtiger Fläche zeichnen:
    dc.Ellipse(20, 20, 100, 100);

    dc.SelectObject(pOldBrush);
}
```

7.4 Die Hintergrundfarbe

Die Hintergrundfarbe wird beispielsweise beim Zeichnen von Text verwendet. Es ist jedoch nicht die Hintergrundfarbe des aktuellen Fensters! Diese kann nur über den Messagehandler `CWnd::OnEraseBkgnd()` definiert werden.

```
virtual COLORREF CDC::SetBkColor(COLORREF crColor)
```

Setzt die aktuelle Hintergrundfarbe. Der Returnwert ist die bisher gesetzte Hintergrundfarbe.

```
int CDC::SetBkMode (int nBkMode)
```

Setzt den Typ der Hintergrundfarbe. `nBkMode` kann einer der folgenden Werte sein:

| | |
|--------------------------|-------------------------------------|
| <code>OPAQUE</code> | Hintergrundfarbe ist undurchsichtig |
| <code>TRANSPARENT</code> | Hintergrundfarbe ist durchsichtig |

Der Returnwert ist der bisher gesetzte Modus.

```
COLORREF CDC::GetBkColor() const
int CDC::GetBkMode() const
```

Diese beiden Funktionen ermitteln die aktuelle Hintergrundfarbe bzw. den aktuellen Hintergrundmodus.

7.5 Bitmaps

Bitmaps sind einfach Byte-Arrays, die ein bestimmtes Bitmuster enthalten, das als Bild interpretiert wird. Für die Anzeige von Bitmaps muss tief in die Trickkiste gegriffen werden: Neben dem `CBitmap`-Objekt muss ein so genannter Memory-Device-Context angefordert werden. Ein Memory-Device-Context ist ein Device-Context, der schlicht und einfach mit dem Speicher als Gerät verbunden ist. Die „Zeichenfläche“ ist ein vom Programmierer definierter Speicherbereich - nämlich die Bitmap. Bitmaps werden also unter Windows als Zeichenflächen im Speicher betrachtet.

Um also eine Bitmap am Bildschirm anzuzeigen, werden zwei DCs angefordert: Einer für das Fenster und einer für den Speicher, der dieselben Eigenschaften hat wie der Fenster-DC. Danach wird die gewünschte Bitmap in den Memory-DC eingesetzt, d.h. dem Memory-DC wird eine „Zeichenfläche“ zugewiesen.

Zuerst aber muss die eigentliche Bitmap erzeugt oder geladen werden.

7.5.1.1 Bitmap laden oder erzeugen

Das Objekt `CBitmap` besitzt nur einen Defaultkonstruktor, es muss also daher in zwei Schritten erzeugt werden: Erst das C++-Objekt, dann der Handle.

Eine Bitmap kann entweder aus den Ressourcen geladen werden oder programmgesteuert erzeugt werden.

```
BOOL CBitmap::LoadBitmap(UINT nIDResource)
```

```
BOOL CBitmap::LoadBitmap(LPSTR lpszResourceName)
```

Lädt eine per AppStudio erstellte Bitmap aus den Ressourcen. `nIDResource` ist die ID der Bitmap; optional kann auch ein String übergeben werden (das war aber nur in früher üblich). Für `nIDResource` können auch vordefinierte Konstanten verwendet werden, um Standardbitmaps zu laden. Eine Auflistung findet sich in der Referenz.

```
BOOL CBitmap::CreateBitmap(int nWidth, int nHeight, UINT nPlanes,
UINT nBitCount, const void FAR *lpBits);
```

Erzeugt eine neue Bitmap und initialisiert sie mit einem bestimmten Bitmuster. Diese Funktion wird dann verwendet, wenn Bitmaps aus externen Bilddateien (z.B. PCX- oder TGA-Dateien) erzeugt werden.

| | |
|------------------------|---|
| nWidth, nHeight | Höhe und Breite der Bitmap, in Pixel |
| nPlanes | Anzahl der Farbebenen in der Bitmap |
| nBitCount | Anzahl der Bits pro Pixel. Für eine Farbbitmap muß entweder dieser oder der vorige Parameter den Wert 1 erhalten. |
| lpBits | Zeiger auf ein Byte-Array, das die Bitmap-Daten enthält. Wird NULL übergeben, bleibt die Bitmap uninitialized. |

7.5.1.2 Größe der Bitmap feststellen

Oft wird die Größe der Bitmap benötigt, bevor die Bitmap angezeigt werden kann. Dazu kann eine Funktion aus der Basisklasse verwendet werden:

```
CGdiObject::GetObject(int nCount, LPVOID lpObject)
```

Diese Funktion ermittelt bestimmte Attribute eines GDI-Objekts. Welche dies sind, hängt vom Objekt selbst ab.

| | |
|-----------------|--|
| nCount | Die Größe der Struktur, in Bytes |
| lpObject | Zeiger auf eine Struktur, die die Attribute erhalten soll. Je nach GDI-Objekt ist dies: BITMAP - für Bitmaps LOGFONT - Für Fonts LOGPEN - Für Pens LOGBRUSH - für Brushes int (!) - für Paletten |

Bemerkung: Im Falle von Fonts sind diese Informationen jedoch nicht besonders hilfreich, da nur die logischen Font-Merkmale eingetragen werden. Beim Einsetzen des Font in den DC wählt der Fontmapper jedoch unter Umständen einen ganz anderen Font aus. Dessen Merkmale können nur mit `CDC::GetTextMetrics()` ermittelt werden.

Für Bitmaps wird die Struktur BITMAP verwendet. Diese enthält folgende Komponenten:

```
typedef struct tagBITMAP
{
    int bmType;    // Typ; 0 für logische Bitmaps
    int bmWidth;  // Breite
    int bmHeight; // Höhe
    int bmWidthBytes; // Anzahl der Bytes pro Zeile
    BYTE bytePlanes; // Anzahl der Farbebenen
    BYTE bmBitsPixel, // Anzahl der Bits pro Farbpixel
    void FAR *bmBits; // Zeiger auf die eigentliche Bildinformation
} BITMAP;
```

Die Größe der Bitmap kann daher folgendermaßen festgestellt werden:

```
CBitmap Bitmap;
BITMAP bmInfo;
Bitmap.LoadBitmap(IDB_SOMEBITMAP);

Bitmap.GetObject(sizeof(bmInfo), &bmInfo);

int width = bmInfo.bmWidth;
int height = bmInfo.bmHeight;
```

Anmerkung: In der Referenz werden Sie die Funktion `CBitmap::GetDimension()` finden - diese Funktion liefert aber nicht die Größe der Bitmap zurück, sondern nur deren "originale" Größe in cm.

7.5.1.3 Der Memory-DC

Um nun die Bitmap sichtbar zu machen, muss sie in einen Memory-DC eingesetzt werden und anschließend in den Fenster-DC kopiert werden.

Die Erzeugung des Memory-DC erfolgt mit der Funktion

```
virtual BOOL CDC::CreateCompatibleDC(CDC *pDC);
```

Diese Funktion erzeugt einen Device-Context, dessen Zeichenfläche ein bestimmter Speicherbereich (d.h. eine Bitmap) ist. Direkt nach Erzeugung wird eine Bitmap von 1x1 Pixel Größe eingesetzt. Der neue Device-Context hat dieselbe Farbtiefe wie der DC, der als Parameter übergeben wird.

Danach kann die Bitmap mit `CDC::SelectObject()` in den Memory-DC eingesetzt werden. Danach ist die Bitmap gewissermaßen die Zeichenfläche des Memory-DC. Rein

theoretisch können also auch Zeichenoperationen durchgeführt werden, mit denen die Bitmap zur Laufzeit verändert werden kann.

Mit der Funktion `BitBlt()` (lies: „Bitblit“, heißt soviel wie „Bit block transfer“) kann die Bitmap ganz oder teilweise von einem DC in den anderen übertragen werden:

```
BOOL CDC::BitBlt(int x, int y, int nWidth, int nHeight, CDC *pSrcDC, int xSrc, int ySrc, DWORD dwRop);
```

Kopiert einen bestimmten Bereich vom übergebenen DC zu diesem DC.

| | |
|------------------------|--|
| x,y | Koordinaten der linken oberen Ecke der Zielfläche |
| nWidth, nHeight | Breite und Höhe des zu kopierenden Rechtecks |
| pSrcDC | Zeiger auf den DC, aus dessen Zeichenfläche herauskopiert werden soll |
| xSrc, ySrc | Koordinaten der linken oberen Ecke des Quellbereiches |
| dwRop | Gibt die Rasteroperation an, die beim Kopieren durchgeführt werden soll. Die wichtigsten Werte sind: SRCCOPY - überschreibt den Zielbereich SRCINVERT - kopiert mit XOR-Operator |

7.5.1.4 Beispiel

Die folgende Funktion kopiert eine benutzerdefinierte Bitmap in die Fenstermitte:

```
void CMainWindow::DrawLittleBitmap()
{
    // Eine kleine Bitmap in die Mitte zeichnen:
    CClientDC dc(this);
    CDC memdc;
    CRect rect;

    CBitmap Bitmap;
    CBitmap *pOldBmp;
    BITMAP bmInfo;

    // Größe der Client-area feststellen:
    GetClientRect(rect);

    // Bitmap laden und Dimensionen feststellen:
```

```

Bitmap.LoadBitmap(IDB_MYBITMAP);
Bitmap.GetObject(sizeof (bmInfo), &bmInfo);

// Memory-DC anfordern und Bitmap hineinselektieren
memdc.CreateCompatibleDC(&dc);

pOldBmp = memdc.SelectObject(&Bitmap); // alte Bitmap merken

// Bitmap in die Mitte der Client-Area kopieren:
dc.BitBlt((rect.right-bmInfo.bmWidth)/2,
          (rect.bottom-bmInfo.bmHeight)/2,
          bmInfo.bmWidth,
          bmInfo.bmHeight, // Zielbereich
          &memdc, // Ursprungs-DC
          0, 0, // Ursprungskoordinaten
          SRCCOPY); // Funktion: einfach drüberkopieren

memdc.SelectObject(pOldBmp); // und Wiederherstellung des DC

// DCs und Bitmap werden automatisch im Destruktor zerstört
}

```

Als Erweiterung zu `BitBlt()` gibt es die Funktion `StretchBlt()`:

```

BOOL CDC::StretchBlt(int x, int y, int nWidth, int nHeight,
CDC *pSrcDC, int xSrc, int ySrc,
int nSrcWidth, int nSrcHeight, DWORD dwRop);

```

Verhält sich wie `BitBlt()`, nur kann der zu kopierende Bereich auch verkleinert oder vergrößert werden.

| | |
|------------------------------|---|
| x, y | Koordinaten der linken oberen Ecke der Zielfläche |
| nWidth, nHeight | Breite und Höhe des zu kopierenden Rechtecks |
| pSrcDC | Zeiger auf den DC, aus dessen Zeichenfläche herauskopiert werden soll |
| xSrc, ySrc | Koordinaten der linken oberen Ecke des Quellbereiches |
| nSrcWidth, nSrcHeight | Breite und Höhe des Quellbereiches. Ist diese Größe unterschiedlich von der des Zielbereiches, wird die Bitmap entsprechend vergrößert oder verkleinert in den Zielbereich kopiert. |
| dwRop | Gibt die Rasteroperation an, die beim Kopieren durchgeführt werden soll. Die wichtigsten Werte sind: |

| |
|--|
| SRCCOPY - überschreibt den Zielbereich SRCINVERT - kopiert mit XOR-Operator |
|--|

Natürlich müssen die Bit-Transferoperationen nicht nur für die Anzeige von Bitmaps verwendet werden - sie wird generell für den Transfer von Bildschirmhalten verwendet werden, auch innerhalb eines DC. Das SDK-Tool „Zoomin“ führt dies beispielsweise durch.

7.6 Textausgaben

Für Textausgaben gibt es eine ganze Reihe von Funktionen, aus denen im Folgenden die wichtigsten aufgelistet sind:

```
virtual BOOL CDC::TextOut(int x, int y, const CString &str);
virtual BOOL CDC::TextOut(int x, int y, LPSTR lpszString, int nCount);
```

Gibt einen einzeiligen Text an den aktuellen Koordinaten aus. Dank Function-Overloading gibt es zwei Möglichkeiten, den Text zu übergeben.

Die Bedeutung der Startkoordinaten wird von der Funktion `SetTextAlign()` definiert:

```
UINT CDC::SetTextAlign(UINT nFlags);
```

Definiert die Bedeutung der Startkoordinaten in der Funktion `TextOut()`.

`nFlags` ist eine ODER-Verknüpfung aus nachfolgend angeführten Werten. Insgesamt sind es drei Gruppen von Werten, wobei aus jeder Gruppe ein Wert ausgewählt werden muss:

Erste Gruppe: Die horizontale Ausrichtung

| | |
|-----------|------------------------------------|
| TA_CENTER | Der Text wird zentriert ausgegeben |
| TA_LEFT | Linksbündige Ausgabe |
| TA_RIGHT | Rechtsbündige Ausgabe |

Die zweite Gruppe bestimmt die vertikale Ausrichtung:

| | |
|-------------|---|
| TA_TOP | Ausrichtung an der Oberkante der Schriftart |
| TA_BOTTOM | Ausrichtung an der untersten Kante der Schriftart |
| TA_BASELINE | Ausrichtung an der Baseline der Schriftart, also auf jener Ebene, auf der die Großbuchstaben aufliegen. |

Die letzte Gruppe bestimmt, ob die aktuelle Position aktualisiert werden soll:

| | |
|---------------|--|
| TA_UPDATECP | Nach der Ausgabe des Textes wird die aktuelle Position auf das Ende des Textes gesetzt. Damit kann Text hintereinander in einer Zeile ausgegeben werden. Achtung! Wird dieser Wert gesetzt, werden die x/y-Koordinaten in der Funktion <code>TextOut()</code> ignoriert, statt dessen wird die aktuelle Position genommen. |
| TA_NOUPDATECP | Die aktuelle Position wird nicht verändert. |

Hier ein Beispiel für eine zentrierte Textausgabe. Der Text wird an der x-Koordinate 100 zentriert ausgegeben:

```
CClientDC dc(this);
dc.SetTextAlign(TA_CENTER | TA_TOP | TA_NOUPDATECP);
dc.TextOut(100, 100, CString("Der Worte sind genug gewechselt - laßt mich
auch endlich Taten sehen!"));
```

Mit dem Wert `TA_UPDATECP` kann der gleiche Text etwas „umständlicher“ auch linksbündig folgendermaßen zusammengestückt werden:

```
CClientDC dc(this);
dc.SetTextAlign(TA_LEFT | TA_TOP | TA_UPDATECP);
dc.MoveTo(0, 100);
// Achtung: Die Koordinaten in TextOut() werden ignoriert:
dc.TextOut(999, 999, CString("Der Worte sind genug gewechselt"));
dc.TextOut(999, 999, CString(" - aßt mich auch endlich Taten sehen!"));
```

Die Ausrichtungsart `TA_BASELINE` ist vor allem dann interessant, wenn verschiedene Schriftarten nebeneinander ausgegeben werden müssen (siehe Beispiel im nächsten Abschnitt)

Eine leistungsfähigere Funktion ist die folgende:

```
CDC::DrawText(LPCSTR lpszString, int nCount, LPRECT lpRect, UINT
nFormat);
```

Gibt einen mehrzeiligen Text in ein bestimmtes Rechteck aus.

| | |
|-------------------------|--|
| <code>lpszString</code> | Der auszugebende Text, Die Zeilen müssen mit <code>\n</code> getrennt sein |
| <code>nCount</code> | Die Anzahl der Buchstaben des Textes |
| <code>lpRect</code> | Zeiger auf ein Rechteck oder auf eine <code>CRect</code> -Klasse, in dem der Text ausgegeben werden soll. |
| <code>nFormat</code> | Eine Kombination folgender Werte: <code>DT_BOTTOM</code> Ausrichtung am unteren Rand der Textzeile (nur mit <code>DT_SINGLELINE</code>) <code>DT_TOP</code> Ausrichtung am oberen Rand der Textzeile (nur mit <code>DT_SINGLELINE</code>) <code>DT_SINGLELINE</code> Falls gesetzt, muß der Text einzeilig sein. <code>DT_LEFT</code> Linksbündig <code>DT_CENTER</code> |

| | |
|--|---|
| | Zentriert DT_RIGHT Rechtsbündig DT_TABSTOP Setzt die Anzahl der Spaces pro Tab; muß im High-Byte von nFormat stehen DT_NOPREFIX Das Zeichen „&“ wird so ausgegeben; fehlt diese Option, werden Buchstaben, denen dieses Zeichen voransteht, unterstrichen. DT_VCENTER vertikal zentrierter Text (nur bei DT_SINGLELINE) DT_WORDBREAK automatischer Wortumbruch am Zeilenende DT_EXTERNALLEADING Die external leadings des Font werden in die Berechnung der Zeilenhöhe miteinbezogen. |
|--|---|

7.6.1.1 Fonts

Für die Auswahl der Schriftart muss ein Font angefordert werden und in einen Device-Context eingesetzt werden. Wenn ein Device-Context angefordert wird, ist automatisch der System-Font eingesetzt, der auch in den Menüs erscheint. Dieser Font ist übrigens der einzige Font, der mit Sicherheit immer vorhanden ist.

Fonts werden über das GDI-Objekt CFont angesprochen. Ein Font kann folgendermaßen erzeugt werden:

```
CFont::CreateFont(int nHeight, int nWidth, int nEscapement,
int nOrientation, int nWeight, BYTE bItalic,
BYTE bUnderline, BYTE bStrikeOut, BYTE nCharSet,
BYTE nOutPrecision, BYTE nClipPrecision,
BYTE nQuality, BYTE nPitchAndFamily,
LPCSTR lpszFacename)
```

Erstellt einen „logischen“ Font. Die einzelnen Parameter sind als „Wünsche“ bezüglich der Font-Charakteristika anzusehen, die aber nicht unbedingt erfüllt werden müssen. Da die Beschreibung der einzelnen Werte viele Seiten füllen würde, seien Sie hier auf die Dokumentation verwiesen.

Übrigens waren viele dieser Werte vor der Windows-Version 3.1 wichtig, da es damals nur Rasterfonts gab, die nicht frei skalierbar waren. Die Einführung der TrueType-Fonts hat viele Probleme aus der Welt geschaffen.

Wurde der Font einmal erzeugt, kann er mittels `SelectObject` in den Device-Context eingesetzt werden. Danach können mit dem DC Textausgaben durchgeführt werden:

```
void CMyView::OnDraw(CDC *pDC)
{
    CFont fontNormal, *pOldFont;
```

```

fontNormal.CreateFont(-12, 0, 0, 0, FW_BOLD,
    TRUE, FALSE, FALSE, ANSI_CHARSET,
    OUT_TT_PRECIS, CLIP_DEFAULT_PRECIS, PROOF_QUALITY,
    VARIABLE_PITCH | FF_SWISS, "Arial");

pOldFont = pDC->SelectObject(&fontNormal);
pDC->SetTextAlign(TA_TOP |TA_NOUPDATECP |TA_LEFT);
pDC->TextOut(0, 0, CString("Das ist Arial,12 Punkt, Fett und
Kursiv "));
pDC->SelectObject(pOldFont);
}

```

Achtung: Die Funktion `CreateFont()` erzeugt nur einen „logischen“ Font, d.h. im `CFont`-Objekt werden eigentlich nur bestimmte gewünschte Anforderungen an den Font gespeichert. Wenn der Font in den Device-Context mittels `CDC::SelectObject()` eingesetzt wird, sucht der sog. Font-Mapper in der vorhandenen Fontliste den passendsten Font heraus. Existiert kein Font, der den Anforderungen entspricht, wird ein möglichst „ähnlicher“ Font ausgewählt. Man sollte sich also niemals darauf verlassen, dass der in den DC eingesetzte Font auch wirklich den angeforderten Wünschen entspricht.

Nun ist es aber häufig nötig, die konkreten Dimensionen der Schriftart zu erfahren, etwa dann, wenn Grafiken beschriftet oder die Höhen von Zeilen, die aus mehreren Fonts bestehen, errechnet werden müssen.

In diesen Fällen hilft die Funktion `GetTextMetrics()` weiter:

```

BOOL CDC::GetTextMetrics(LPTEXTMETRIC lpTextMetric);

```

Füllt den übergebenen `TEXTMETRIC`-Struct mit den Maßen der aktuellen Schriftart im Device-Context.

Die Bedeutung der wichtigsten Komponenten des `TEXTMETRIC`-Structs können am besten grafisch dargestellt werden. Die einzelnen Dimensionen werden in logischen Koordinaten angeliefert. Die Bedeutung der restlichen Komponenten kann der Referenz entnommen werden.

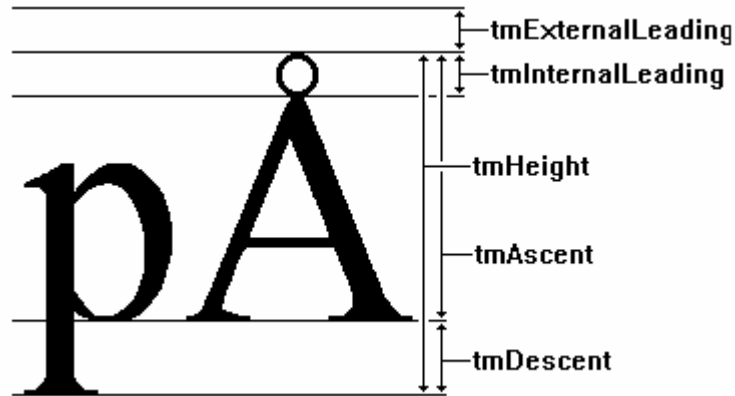


Abbildung 3 - Wichtige Komponenten der Struktur TEXTMETRIC

Allerdings ist oft auch wichtig, die Breite eines Textes zu erfahren, etwa dann, wenn Text im Blocksatz ausgegeben werden muss. Die gebräuchlichste Funktion ist:

```
CSize CDC::GetTextExtent(LPCSTR lpszString, int nCount) const;
```

Ermittelt Breite und Höhe eines beliebigen Strings anhand des momentan im DC befindlichen Fonts.

7.6.1.2 Textfarbe

Die Textfarbe kann mit folgender Funktion gesetzt werden:

```
virtual COLORREF CDC::SetTextColor(COLORREF crColor);
```

Setzt die aktuelle Textfarbe. Der Returnwert ist die bisher gesetzte Textfarbe.

```
COLORREF CDC::GetTextColor() const;
```

Ermittelt die momentan eingesetzte Textfarbe

7.7 Invalidate und Update

Stellen wir uns ein Programm für die Erstellung von Datenflussdiagrammen vor: Mit Mausklicks kann der Anwender beliebig viele Rechtecke erzeugen, die durch Pfeile miteinander verbunden sein sollen. In Programmen dieser Art müssen also sowohl bei der Behandlung von `OnLButtonDown()` als auch in `OnDraw()` Zeichenoperationen