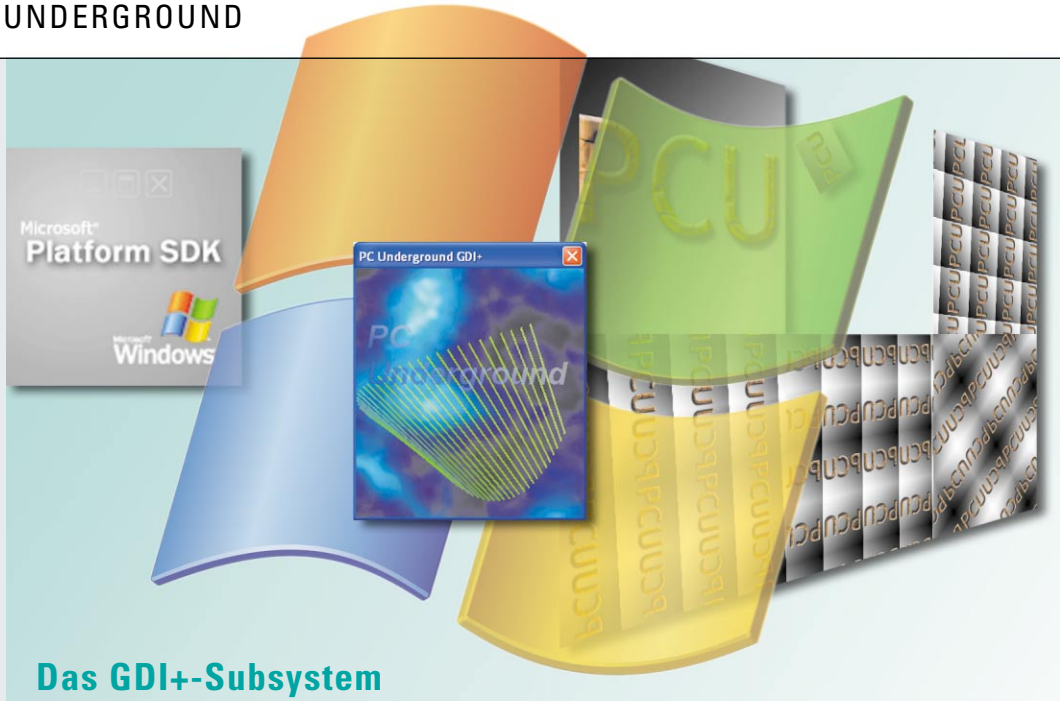


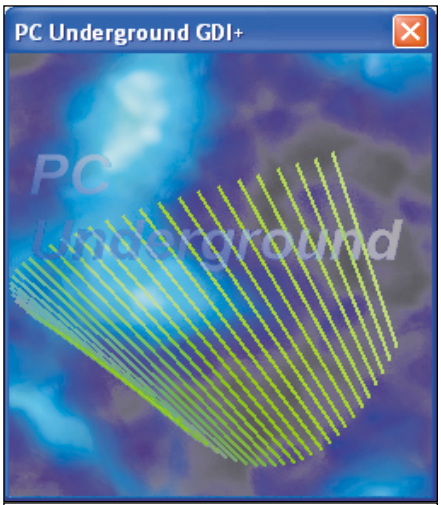
Das GDI+-Subsystem

Schnell zur Schönschrift

GDI+, das neue Subsystem von Windows XP, vereinfacht und beschleunigt die Programmierung von Bildschirm- und Druckerausgaben. Im Artikel lesen Sie, wie Sie GDI+ in eigenen Applikationen mit Gewinn einzusetzen.

Carsten Dachsbacher

Das Window-GDI+-Subsystem von Windows XP und Windows Server 2003 ist für die Darstellung bzw. Ausgabe auf dem Bildschirm und Druckern verantwortlich. Sie greifen hierbei auf die Nachfolge API zu GDI (Windows Graphics Device Interface) über C++-Klassen zu, die bereits in älteren Windows-Versionen zur Verfügung standen. Zwecks Abwärtskompatibilität unterstützen Windows XP/Server 2003 auch das bisherige GDI Interface, doch GDI+ können Sie leichter und schneller für neue Applikationen verwenden. Die Dienste von GDI+ lassen sich in drei große Bereiche unterteilen.



GDI+: Unser Beispielprogramm stellt eine Dialogbox mit einer Animation dar, die Sie mit wenigen Zeilen programmieren.

- Der erste Punkt ist 2D-Vektor-Grafik. Damit ist das Zeichnen von Primitiven, wie Linien, Kurven etc., gemeint, die durch eine Menge von Punkten in einem Koordinatensystem definiert sind, z.B. wird eine Linie durch Start- und Endpunkt definiert. GDI+ stellt Klassen zur Verfügung, die die Information über die Primitive selbst enthalten, Klassen, die speichern, wie die Primitive gezeichnet werden sollen, und Klassen, die das Zeichnen an sich übernehmen. So zeichnen Sie z.B. ein Rechteck mit der *Rect*-Klasse, speichern Dimension und Position per *Pen*-Klasse und Sie legen Linienfarbe, -dicke und -stil per *Graphics*-Klasse fest.
- Bildbearbeitung ist der zweite Aufgabenbereich, der durch GDI+ abgedeckt wird. Darunter fällt die Verwaltung und das Zeichnen von Bitmaps, wofür eine Reihe von Klassen und Methoden zur Verfügung stehen. Außerdem wird die Funktionalität zum Laden und Speichern von Bitmaps in verschiedenen Formaten, wie z.B. JPG, PNG, BMP, bereitgestellt.
- Der dritte und letzte Bereich ist die Typographie, also die Darstellung von Text mit verschiedenen Schriftarten, -größen und Attributen. Hierzu zählt z.B. auch die Textdarstellung mit Subpixel-Antialiasing für TFT-Bildschirme.

Die Klassenstruktur

Das C++-Interface von GDI+ besteht aus etwa 40 Klassen, zahlreichen *Enumerations* und wenigen *Structures*. Weiterhin gibt es noch eine kleine Zahl von Funktionen, die nicht *Member* einer Klasse sind. Die Klasse *Graphics* ist gleichsam der Kernpunkt des GDI+-Interface: Hier finden Sie die Methoden, die Linien, Kurven, Bilder und Text zeichnen. Wie bereits erwähnt, ist die Graphics Klasse auf die Informationen aus anderen Klassen angewiesen, um z.B. Linien mit einer bestimmten Farbe (Pen) zu zeichnen. Es gibt Klassen, die primär als Datencontainer dienen, wie z.B. die *Rect-*, *Point-* und *Size*-Klasse, und für verschiedene Zwecke eingesetzt werden. Andere sind spezielle Hilfsklassen, wie z.B. die *BitmapData*-Klasse, die Bildattribute und -daten für die *Bitmap*-Klasse speichert, die wiederum Methoden zur Bildmanipulation bereitstellt.
Zu GDI+ gehören aber auch einige Funktionen, die nicht Bestandteil einer Klasse sind. Die beiden wichtigsten Funktionen sind *GdiplusStartup*, die Sie vor einem anderen GDI+-Befehl aufrufen müssen, und *GdiplusShutdown*, die Sie verwenden, wenn Sie alle GDI+-Aufrufe abgeschlossen haben.

GDI+ Step by Step

Schrittweise demonstrieren wir Ihnen die Verwendung der wichtigsten GDI+-Funktionen anhand eines einfachen Beispiels. Unser Beispielprogramm öffnet eine modale Dialogbox, deren *Window Procedure* beim Empfang einer *WM_PAINT* Nachricht unsere GDI+-Kommandos abarbeitet, um eine einfache Animation darzustellen.
Mit einem Timer – gesetzt beim Empfang der *WM_INITDIALOG* Nachricht – wird das Neuzeichnen des Dialoginhalts regelmäßig mit *InvalidateRect(...)* erzwungen. Beim Start des Programms initialisieren Sie zunächst GDI+ und erzeugen die Dialogbox:

```
GdiplusStartupInput gdipStartupInput;
```

Das zurückgelieferte *gdiplusToken* übergeben Sie bei Programmende wieder:

```
GdiplusShutdown( gdiplusToken );
```

Widmen Sie sich also nun der *WM_PAINT*-Behandlung. Um auf einen *Device* Kontext zu zeichnen, müssen Sie dazu ein Graphics Objekt erzeugen. Dieses Objekt speichert alle Attribute für ein Device und die Attribute der Primitive, die Sie zeichnen:

```
Graphics graphics( hdc );
```

Da Sie, um Flackern zu vermeiden, nicht direkt in den Dialog zeichnen wollen, legen Sie ein Bitmap mit der richtigen Größe an und verwenden dieses später zum Zeichnen – dazu benötigen Sie ein Graphics Objekt, das mit dem Bitmap assoziiert ist:

```
RECT rect;.....
```

Zum Einstieg zeichnen Sie eine Linie in das Bitmap. Dazu benötigen Sie einen *Pen*, der Farbe und Strichstärke der Linie speichert. Anschließend können Sie mit der *Graphics::DrawLine*-Methode die Linie von *(0,0)* nach *(77,44)* zeichnen:

```
Pen p(Color(alpha,red,green,blue));
graph->DrawLine( &pen, 0, 0, 77, 44 );
```

Um das Ergebnis zu sehen, müssen Sie das Bitmap noch auf den *Device Context* der Dialogbox kopieren.
Um ein Bitmap auf ein Graphics Objekt zu zeichnen (und um nichts anderes handelt es sich hier), verwenden Sie folgenden Aufruf, wobei das *Rectangle* den zu zeichnenden Bereich angibt:

```
graphics.DrawImage( &bmp, rect );
```

Abschließend deklarieren Sie den Client-Bereich der Dialogbox, d.h. die Region des Bildschirms, die Sie aufgrund der *WM_PAINT*-Nachricht aktualisieren sollten:

```
ValidateRect( hDlg, &rect );
```

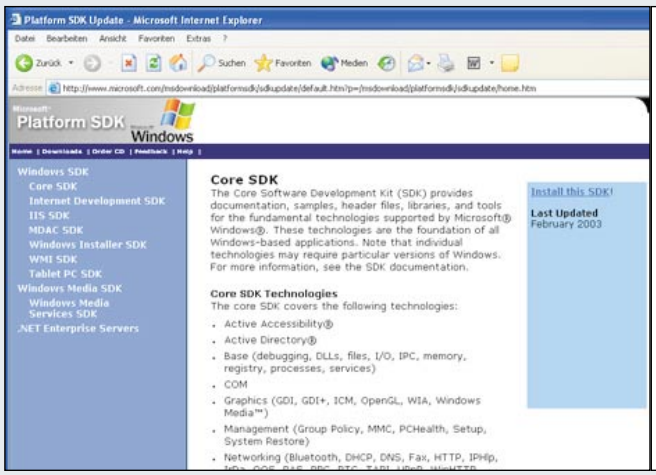
Das Beispielprogramm setzt noch weitere Grafikfunktionen ein: Wie Sie vielleicht bemerkt haben, haben wir den Hintergrund des Bitmaps gar nicht gelöscht. Der Inhalt ist also undefiniert. Zum Löschen des Bildhintergrundes zeichnen Sie ein Rechteck in der gewünschten Farbe. Um Flächen einzufärben, verwenden Sie keinen *Pen*, sondern einen so genannten *Brush*. Ein *Brush* ist ein Füllmuster, das eine einzelne Farbe, einen Farbverlauf oder eine Textur enthalten kann.
Um den Hintergrund einfarbig zu kolorieren, legen Sie einen *SolidBrush* an und rufen damit die *FillRectangle* Methode des Graphics Objektes auf:

```
SolidBrush brush( Color(255,0,0,0) );
graph->FillRectangle( &brush, rect );
```

Wenn Sie den Hintergrund mit einem Farbverlauf füllen möchten, ist das für GDI+ kein Problem. Sie tauschen lediglich den *Brush* aus und verwenden statt einem *SolidBrush* einen *LinearGradientBrush*. Das folgende Beispiel

Installation des Platform SDK und der Redistributables

Um das Beispielprogramm zu kompilieren, benötigen Sie die GDI+-Headerdateien und Libraries. Diese sind Bestandteil des Microsoft SDK (Platform SDK). Sie finden online eine Anleitung zur Installation: **www.microsoft.com/msdownload/platformsdk/sdkupdate**.
Auf der Webseite finden Sie mehrere SDKs zur Auswahl. GDI+ ist Teil des Core SDK, das Sie auf Ihrem Rechner installieren können, indem Sie den Instruktionen folgen. Anschließend müssen Sie nur noch die *Include* und *Library* Pfade Ihres C++-Compilers anpassen. Bei Visual C++ 6.0 finden Sie diese Einstellungen unter *Tools/Optionen*, bei Visual Studio .NET unter *Extras/Optionen* im Unterpunkt *Projects/VC++ Verzeichnisse*.
Wenn Sie in einer Ihrer Applikationen GDI+ verwenden und diese auf einem älteren System wie Windows 2000 laufen lassen wollen, müssen Sie die notwendigen DLLs mit Ihrem Programm mitliefern. Die dazu notwendigen Dateien sind im *Platform SDK Redistributable: GDI+ RTM* Paket enthalten, das Sie unter folgender URL finden können: **www.microsoft.com/downloads/**.



***Platform SDK*: Mit der Online Installation erhalten Sie Headerdateien und Bibliotheken, um GDI+-Applikationen zu entwickeln.**

(Quellcode Heft-CD) erzeugt einen Farbverlauf von halb-transparentem Gelb zu opakem Blau, der diagonal (45 Grad Rotation) verläuft:

```
LinearGradientBrush ...
```

Die dritte hier vorgestellte Variante ist der *TextureBrush*. Dem Konstruktor dieser Klasse übergeben Sie ein Image Objekt. Ein Image Objekt stellt Methoden zur Verfügung, um Bitmaps und Vektorgrafik zu laden und zu speichern. Die Bilddaten werden dabei im Objekt gehalten und mit einer Reihe von Methoden können Sie auf Attribute zugreifen. Die Image Klasse kann auch Bilddateien verschiedenster Formate wie *BMP, ICON, GIF, JPEG, PNG, TIFF* lesen. Dazu verwenden Sie diese Klasse auch, wenn Sie wie im Beispiel einen *TextureBrush* anlegen. Laden Sie damit eine Bilddatei und übergeben Sie das Image dem *Brush*-Konstruktor:

```
Image image( L"./data/image.jpg" );
```

Der Wert des zweiten Parameters des Konstruktor (Quellcode auf der Heft-CD) gibt an, dass das Bild beim Füllen einer Fläche gekachelt wird, wobei verschiedene Optionen ein abwechselungsreiches Bild ergeben.
Sowohl der *LinearGradientBrush* als auch der *TextureBrush* bieten Transformationen an, d.h. der *Brush* kann verschoben, skaliert oder rotiert werden. Dazu speichern diese Klassen intern eine Transformationsmatrix, die Sie direkt angeben oder modifizieren können. Die Matrix definiert eine affine Abbildung, speichert diese aber in einer 3x3-Matrix, von der die dritte Zeile immer die Werte *(0,0,1)* enthält. Mit der Methode *ResetTransform* setzen Sie die Abbildung zurück. Anschließend können Sie z.B. die *RotateTransform(30.0f, MatrixOrderPrepend)*, eine Rotationsabbildung, erzeugen und mit der bereits gespeicherten verknüpfen. Der zweite Parameter gibt an, dass die Multiplikation der Rotationsmatrix links an die bereits gespeicherten stattfindet. Beachten Sie, dass die Matrixmultiplikation nicht kommutativ ist und dass sich Abbildungen anschaulich von rechts nach links aufbauen. Um die Matrix am rechten Rand zu multiplizieren, verwenden Sie den Parameter *MatrixOrderAppend*.
Die letzte hier vorgestellte Funktionalität des Graphics Objektes ist die Textausgabe. Als erstes benötigen Sie ein *FontFamily* Objekt, das eine Gruppe von Schriftarten mit demselbem Aussehen aber unterschiedlichem Stil repräsentiert:

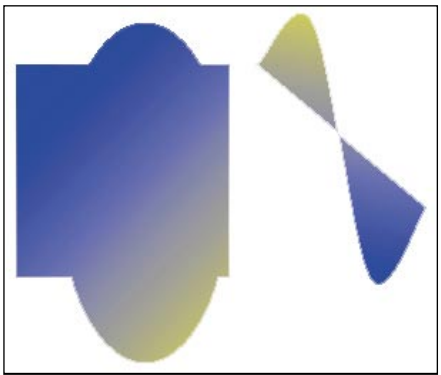
```
FontFamily fontFamily( L"Arial" );
```

Jetzt benötigen Sie noch ein Font Objekt, das die Familie wie Schriftgröße, Art und Stil beinhaltet, um den Text auszugeben. So erhalten Sie einen Arial-Font der Größe 35 in kursiver Fettschrift:

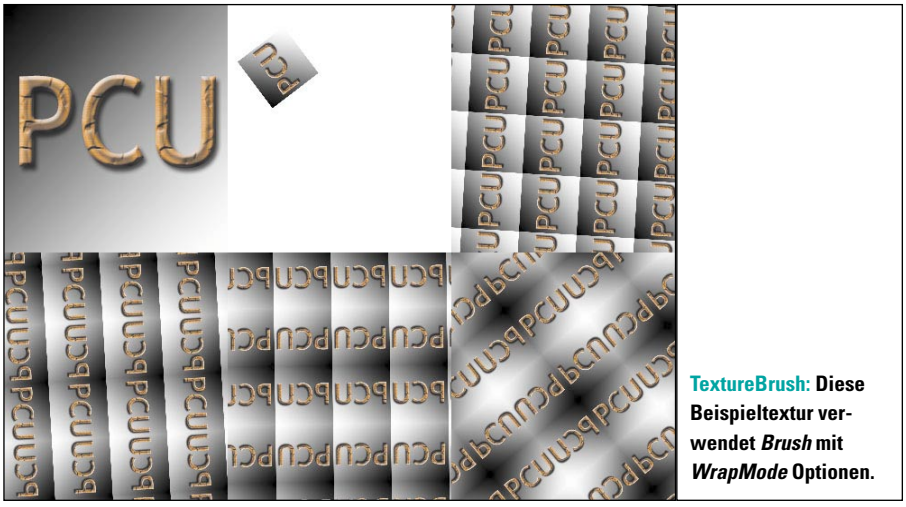
```
Font font( &fontFamily, 35,
  FontStyleItalic | FontStyleBold,
  UnitPixel );
```

Schließlich speichern Sie noch die Position in einer *PointF* Klasse und geben per Graphics Objekt den Text aus:

```
PointF position( 5.0f, 55.0f );...
```



Neue Fähigkeiten: *GradientBrush* mit Farbverlauf und *Cardinal Splines* mit *GraphicsPath*



TextureBrush: Diese Beispieltextur verwendet *Brush* mit *WrapMode* Optionen.

Speichern mit der Image Klasse

Wie erwähnt, laden und speichern Sie per *Image* Bilddateien in verschiedensten Formaten. Als erstes legen Sie ein *IStream* Objekt an, welches Sie in den Bibliotheken des Platform SDK finden (Kasten), um Daten im Hauptspeicher ablegen und auslesen zu können.

```
IStream *stream;...
```

Als nächstes wählen Sie den *Encoder* mit einer Hilfsfunktion aus der MSDN Library, um dessen entsprechende *Class ID* auszulesen:

```
CLSID jpegClsid;....
```

Diese – im Source Code in der MSDN Library enthaltene – Funktion enumeriert alle installierten Encoder und sucht anhand des übergebenen Bezeichner-Strings die entsprechende Class ID heraus. Diese übergeben Sie später an die Bitmap oder Image Klasse für das Encoding des Bildes. Nachdem Sie die Class ID bekommen haben, legen Sie noch die Kompressionsparameter für das JPEG-Encoding fest. Diese speichern Sie in der *EncoderParameters* Klasse (ein Container für *EncodeParameter* Objekte). Jeder Eintrag besteht aus einem Identifier (z.B. *EncoderQuality*), dem Typ der Variablen (*EncoderParameterValueTypeLong*), Anzahl der Werte und der Adresse des Wertes:

```
EncoderParameters enc;....
```

Damit können Sie das Bitmap bereits kodieren lassen und die resultierenden Daten in den Stream schreiben:

```
graph->Save( stream,
&jpegClsid, &encoderParameters );
```

Per *Seek*-Kommando erfragen Sie die Größe der Bilddatei, die sich im Speicher des Streams befindet:

```
ULARGE_INTEGER compressedSize;....
```

Eine Methode, die Daten zu speichern, ist, Speicher außerhalb dieses Streams zu allokieren, die Daten dort hin zu kopieren und mit beliebigen Dateifunktionen zu schreiben:

```
FILE *f = fopen( „bild.jpg", „wb" );..
```

Mit den GDI+-Funktionen der Image Klasse können Sie somit auch sehr einfach Bildformate konvertieren, indem Sie Bilddateien laden und mit der gerade vorgestellten Methode in einem anderen Format speichern. Oder Sie verwenden den obigen Code, um Fensterinhalte als Bilddateien abzulegen. : et