Windows verschönern mit DirectDraw Overlays

Fenster in allen Farben

Individualisieren Sie Ihr Windows System mit einem selbst programmierten, animiertem Desktop-Hintergrund. Hier erfahren Sie, wie Sie Video Overlays verwenden und Ihre Programme über den System Tray erreichen.

Carsten Dachsbacher

In dieser Ausgabe zeigen wir Ihnen, wie Sie Ihr Windows-System mit einem Desktop-Hintergrund animieren. Es gibt prinzipiell zwei Wege, dies zu erreichen. Der erste nutzt die so genannten Windows Hooks, um sich in das System einzuhängen. So fangen Sie die Nachrichten ab, die das Zeichnen des Desktops anstoßen. Der Desktop ist per Definition auch ein Windows-Fenster.

Der andere Weg verwendet Overlays. Dabei handelt es sich um 2D-Bitmaps, die die Grafikkarte beliebig auf dem Bildschirm platzieren kann. Sie können sie so verwenden, dass Sie nur bestimmte Farben auf dem Bildschirm wie die Farbe Ihres Desktop-Hintergrunds ersetzen. Auf den Overlays können Sie dann beliebige Grafikeffekte darstellen.

Direct Draw und Video Overlays

Im ersten Schritt initialisieren Sie DirectDraw wie andere DirectX-Komponenten: Sie erzeugen eine Instanz des DirectDraw Objektes.

```
LPDIRECTDRAW7 pDirectDraw;
DirectDrawCreateEx( NULL, ....
```

Mit *SetCooperativeLevel(...)* bestimmen Sie das Top-Level-Verhalten Ihres Programms. Die Konstante *DDSCL_NORMAL* bedeutet, dass es sich um eine herkömmliche Anwendung mit einem Fenster handelt. Andere Werte deuten z.B. auf Vollbild- oder Exclusive-Level-Anwendungen hin. Achten Sie bei DirectX-Funktionen immer auf die Rückgabewerte und prüfen Sie eventuell auftretende Fehler. Um mit DirectDraw etwas darzustellen, benötigen Sie immer eine so genannte *Primary Surface*. Surfaces sind Speicherbereiche, in denen Bitmaps verschiedenste Formate und optional mit Hintergrund- oder Tiefenpuffern repräsentieren.

Die Primary Surface ist diejenige, die die für den Benutzer sichtbare Bitmap enthält. Sie beschreiben die Eigenschaften der Surface mit der *ddsd*-Struktur:

```
DDSURFACEDESC2 ddsd;
```

Das *dwFlags*-Feld gibt an, welche Struktureinträge Sie ausfüllen. Mit dem *dwCaps*-Feld legen Sie fest, dass es sich um eine Primary Surface handelt, die Sie so erzeugen:

```
pDirectDraw->CreateSurface(
  &ddsd, &pDDSurfacePrimary, NULL );
```

Overlay Surfaces

Als nächstes benötigen Sie noch eine Surface für das Overlay. Bevor Sie diese anlegen, überprüfen Sie, ob die Grafikhardware Overlays unterstützt. Dazu lassen Sie sich die Device Caps

von Direct Draw geben. In dieser Struktur sind alle Fähigkeiten der Hardware beschrieben:

```
DDCAPS  ddCaps;
INIT_DDSTRUCT( ddCaps );
pDirectDraw->GetCaps(&ddCaps,NULL):
```

Overlays werden unterstützt, wenn das folgende Flag gesetzt ist: *ddCaps.dwCaps & DDCAPS_OVERLAY*. Wenn das der Fall ist, legen Sie die Overlay Surface an. Beginnen Sie wieder damit, die gewünschten Eigenschaften der Surface zu beschreiben:

```
DDSURFACEDESC2 ddsdOv;
INIT_DDSTRUCT( ddsdOv );
```

Die Bedeutung der einzelnen Flags beschreibt detailliert die MSDN Hilfe (**http://msdn.microsoft.com**). Zusammengefasst: Sie beschreiben eine Overlay Surface per Hintergrundpuffer, der im Video-Speicher liegt, mit einer exemplarischen Größe von 320x240 Pixel. Jetzt müssen Sie noch für das Pixelformat entscheiden, ob die Surface die Farbwerte im *RGB*- oder *YUV*-Format enthält und mit welcher Genauigkeit sie gespeichert werden. Die meisten Grafikkarten unterstützen 32-Bit-RGBA-Overlays, die Sie auch im Folgenden anlegen. Das Beispielprogramm beherrscht weitere Pixelformate, um alle Grafikkarten bedienen zu können:

```
DDPIXELFORMAT overlayFormat =
  { sizeof( DDPIXELFORMAT ),
    DDPF_RGB, 0, 32,  0x00FF0000,
 0x0000FF00, 0x000000FF, 0 };
```

Darstellung der Overlays

Als letztes programmieren Sie den aufwendigsten Teil, indem Sie die Overlay Surface darstellen. Dabei geben Sie an, welchen Ausschnitt des Overlays Sie wo auf dem Bildschirm darstellen wollen. Dabei müssen Sie eventuelle Beschränkungen wie Größe und Position des Overlays berücksichtigen. Zudem wird manchmal eine leichte Skalierung bzw. Streckung des Overlays gefordert. Alle Anforderungen sind in den Device Caps des Direct Draw Objektes enthalten, die Sie erfüllen müssen.
Zunächst positionieren Sie die Overlays, die zwei *RECT*-Strukturen definieren. Dann vergrößern Sie das Overlay auf den ganzen Bildschirm:

```
RECT rs = { 0, 1, 320, 240 };
```

Wenn Sie das Overlay in Originalgröße darstellen wollen, müssen Sie die minimale Skalierung einhalten, wozu Sie diese Einträge verwenden:

```
DWORD scale = max( 1000,
  ddCaps.dwMinOverlayStretch );
```

Die Größen für das Overlay beschränken Sie, indem Sie den darzustellenden Ausschnitt zuschneiden:

```
DWORD s = ddCaps.dwAlignSizeSrc;
if (ddCaps.dwCaps&DDCAPS_ALIGNSIZESRC
    && s )
  rs.right -= rs.right % s;
```

Das Ziel-Rechteck vergrößern Sie so, dass es ein Vielfaches des vorgegebenen Alignments wird:

```
DWORD s = ddCaps.dwAlignSizeDest;

if(ddCaps.dwCaps&DDCAPS_ALIGNSIZEDEST
   && s )
 rd.right=((rd.right+s-1)/s)*s;
```

Damit stellen Sie das Overlay dar, wie es nach folgendem Funktionsaufruf das Listing der Heft-CD fortführt:

```
DDOVERLAYFX ovfx; DWORD dwUpdateFlags;
```

Das Overlay bedeckt Ihren gesamten Bildschirm, so dass Sie nichts anderes mehr sehen. Da Sie aber nur Teile wie den Desktophintergrund einfärben wollten, verwenden Sie das so genannte *Color Keying*.
Das Verfahren ersetzt nur eine bestimmte Farbe oder einen bestimmten Farbbereich per Overlay auf dem Bildschirm. So färben Sie Desktop- und Fensterhintergründe oder Schrift mit Farbwerten ein.
Die *DDColorMatch*-Funktion passt den angegebenen Farbwert (hier *(0,0,0)*: schwarz) an das Pixelformat der Primary Surface an. Jetzt teilen Sie der Primary Surface den Color Key Wert mit und aktivieren die Funktionalität, indem Sie die *dwUpdateFlags* vor dem Aufruf der *UpdateOverlay*-Methode modifizieren:

```
DDCOLORKEY colorKey;
colorKey.dwColorSpaceLowValue  =
colorKey.dwColorSpaceHighValue =
  DDColorMatch( pDDSurfacePrimary,
                RGB( 0, 0, 0 ) );

pDDSurfacePrimary->SetColorKey(
  DDCKEY_DESTOVERLAY, &colorKey );

if ( ddCaps.dwCKeyCaps &
  DDCKEYCAPS_DESTOVERLAY )
  dwUpdateFlags |= DDOVER_KEYDEST;
```

Update der Overlay Surface

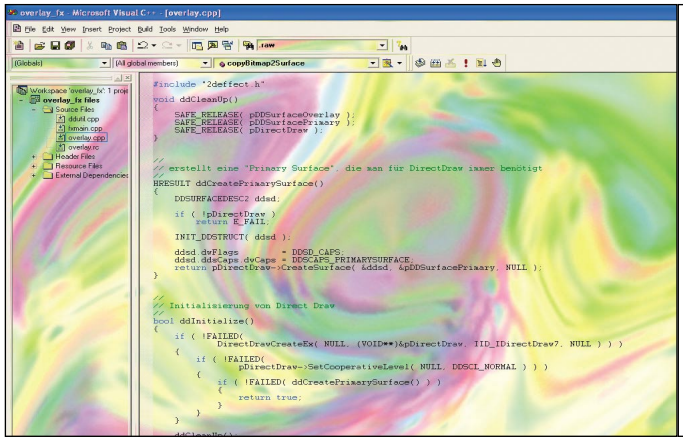
Zuletzt aktualisieren die Overlay Surface. Unserer Beispielprogramm berechnet einen 2D-Grafikeffekt in einem 32-Bit-RGB-Puffer, der in die Overlay Surface zu kopieren ist. Wenn das Pixelformat der Surface von dem des Puffers abweicht, müssen Sie das Format konvertieren, wie dies der Quelltext des Beispielprogramms vorführt.
Um auf den Inhalt einer Surface zuzugreifen, verwenden Sie die *Lock(...)*-Methode. Sie erhalten dann eine Struktur, welche die Auflösung des Bitmaps und die Bytes pro Bitmapzeile beschreiben:

```
pSurface->Lock( NULL, &ddsd,
 DDLOCK_SURFACEMEMORYPTR|DDLOCK_WAIT,
 NULL );
```

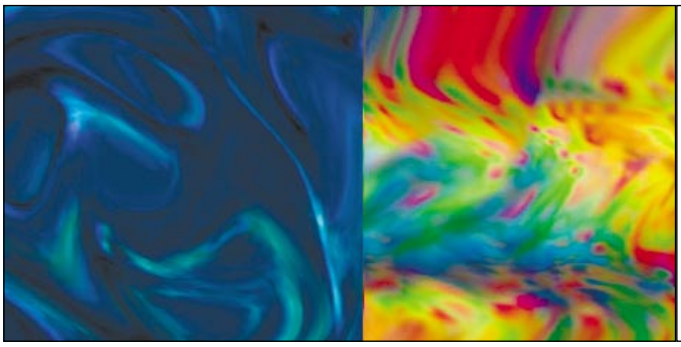
Dorthin kopieren Sie das Bild des Grafikeffektes und beenden den Schreibzugriff mit:

```
Surf = (BYTE*)ddsd.lpSurface;
```

Notifiy Icon: Unsere Applikation in der Tray Bar, mit Kontextmenü und Tooltip.



Color Keying: Das Overlay erzeugt animierte Fensterhintergründe; auch Schrift kann es einfärben.



2D Effekt: Diese Effekte stellt unser Programm auf Ihrem Desktop dar.

```
DWORD *src = ...;
for( y = 0; y < 240; y++ )
{
 memcpy(pSurf,src,sizeof(DWORD)*320);
 pSurf += ddsd.lPitch;
 src += 320;
}

pSurface->Unlock(NULL);
```

System Tray

Anwendungen, die im Hintergrund arbeiten und keine ständige Benutzerinteraktion voraussetzen, brauchen kein ständig geöffnetes Anwendungsfenster. Die notwendigen Funktionen sollten aber trotzdem bequem erreichbar sein, wozu sich der so genannte System Tray anbietet: Dort steuert der Benutzer-Programme per Icon.

Sie programmieren diese so genannten *Notify Icons* vergleichsweise leicht nach Anleitung. Das Konzept der Notify Icons sieht vor, dass Benutzer-Interaktionen wie ein Mausklick auf das Icon als Nachricht an ein Windowsfenster geschickt werden. Das bedeutet, Sie müssen wie gewohnt ein Fenster und eine Nachrichten-Callback-Funktion programmieren, ohne das Fenster darstellen zu müssen. Erzeugen Sie zuerst ein Fenster wie im Beispiel eine Dialog-Box aus dem Ressource File:

```
HWND hWnd = CreateDialog( hInst,
  MAKEINTRESOURCE( IDD_DLG_DIALOG ),
  NULL, (DLGPROC)messageHandler );
```

Als nächstes legen Sie eine *NOTIFYICONDATA*-Struktur an, die die notwendigen Informationen für das Notify Icon enthält. Allerdings gibt es zwei Versionen dieser Struktur, die unterschiedliche Informationen und so verschiedene Strukturgrößen aufweisen. Welche der beiden Varianten Sie verwenden, hängt von der Version der Windows eigenen *Shell32.dll* ab. Um die Versionsnummer abzufragen, bieten die meisten Windows-DLLs wie *Comctl32.dll, Shdocvw.dll* und *Shlwapi.dll* die Methode *DllGetVersion* an. Sie laden also diese DLL, holen die Adresse der *DllGetVersion*-Methode und fragen mit ihrer Hilfe die Versionsnummer ab. Einen Beispielcode finden Sie in unserem Programm und der MSDN-Hilfe von MS Visual C++. So definieren Sie die kürzere Version der Struktur, die auch schon alle Felder enthält:

```
typedef struct _NOTIFYICONDATA {
    DWORD cbSize;
```

Zuerst tragen Sie in Ihre Struktur die tatsächliche Größe ein:

```
NOTIFYICONDATA niData;
```

Nun geben Sie an, welche Felder Sie mit validen Werten füllen. Sie geben ein Icon, eine Tooltip-Nachricht und eine Fensternachricht an:

```
niData.uFlags =
NIF_ICON | NIF_MESSAGE | NIF_TIP;

// beliebige ID
niData.uID = TRAY_ICON_ID;
```

Nun geben Sie an, welches Windows-Fenster (bzw. dessen Message-Handler), die Nachrichten vom Notify Icon empfängt. Diese Callback-Funktion rufen Sie mit einer ausgewählten Nachricht *TWM_TRAYMSG* auf, deren Wert sich zwischen *WM_APP* und *0xBFFF* befindet, sobald ein Mouse Event im Bereich des Notify Icons auftritt.

```
niData.hWnd = hWnd;
```

Dann bleibt also noch der Tooltip-Text und das Icon, die Sie in die Struktur eintragen:

```
lstrcpyn( niData.szTip, _T( „Text“ ),
 sizeof(niData.szTip)/sizeof(TCHAR));
```

Zuletzt schicken Sie die Nachricht an das System ab, die das Traybar Icon hinzufügt:

```
Shell_NotifyIcon (NIM_ADD, &niData);
```

Jetzt geben Sie noch die Ressourcen für das Icon frei:

```
if ( niData.hIcon &&  DestroyIcon...
```

Die Nachrichten, die aufgrund des Notify Icon gesendet werden, bearbeiten Sie also in der Message Handler Funktion der oben angelegten Dialog-Box:

```
INT_PTR CALLBACK messageHandler(
  HWND hWnd, UINT msg,
  WPARAM wP, LPARAM lP )...
```

Ein doppelter Mausklick soll die Dialog-Box darstellen. Ein Klick der rechten Maustaste ruft die *createContextMenu(...)*-Methode auf, die ein Kontext-Menü aufbaut und darstellt:

```
void createContextMenu
     (HWND hWnd)
{
 HMENU hMenu =CreatePopupMenu();

  if ( hMenu )
  {
InsertMenu( hMenu, -1,
  MF_BYPOSITION, TWM_BEISPIEL,
  _T(„Beispieleintrag“));

SetForegroundWindow( hWnd );...
```

Die *CreatePopupMenu()*-Methode erzeugt ein zunächst leeres Drop-Down-Menü. Mit der *InsertMenu(...)*-Methode fügen Sie Menüpunkte hinzu. Die beiden letzten Parameter geben die Konstante für die Nachricht an, die im Falle eines Anklickens gesendet wird, sowie den Text des Eintrags. Nach dem Aufbau des Menüs sorgt *SetForegroundWindow(...)* dafür, dass es sichtbar wird. *GetCursorPos(...)* gibt die Position des Mauszeigers aus, und dementsprechend platzieren Sie das Drop-Down-Menü. Dies und die Eingabenbehandlung leistet der *TrackPopupMenu(...)*-Befehl, dem Sie auch das Window-Handle des Fensters übergeben, das die Nachrichten empfangen soll. Nachdem das Menü verlassen wird, geben Sie die Ressourcen mit *DestroyMenu(hMenu)* wieder frei. Da Sie hier wieder das Window-Handle der Dialog-Box angegeben haben, müssen Sie den Message Handler noch um folgendes erweitern:

```
switch ( msg )
{
  ...
  case WM_COMMAND:
trayMsg = LOWORD( wParam );

switch ( trayMsg )
{ case TWM_BEISPIEL:
// Beispieleintrag wurde gewählt
    break; }
return 1;
  ...
}
```

Damit haben Sie schon alle Werkzeuge in der Hand, die Sie für Notify Icons benötigen. Wenn Sie das Programm verlassen, müssen Sie das Icon aus der Liste löschen:

```
niData.uFlags = 0;
Shell_NotifyIcon(NIM_DELETE,&niData);
```

: et