GDI Objects

Ron Gery
Microsoft Developer Network Technology Group

Created: March 20, 1992

Abstract

This article discusses how to create, select, and delete graphics device interface (GDI) objects such as pens, brushes, fonts, bitmaps, palettes, and regions. Sprinkled throughout are general guidelines for using objects efficiently and for making basic use decisions.

Creation

Each type of object has a routine or a set of routines that is used to create that object.

Pens are created with the **CreatePen** and the **CreatePenIndirect** functions. An application can use either function to define three pen attributes: style, width, and color. The background mode during output determines the color (if any) of the gaps in any nonsolid pen. The PS_INSIDEFRAME style allows dithered wide pens and a different mechanism for aligning the pen on the outside of filled primitives.

Brushes are created with the **CreateSolidBrush**, **CreatePatternBrush**, **CreateHatchBrush**, **CreateDIBPatternBrush**, and **CreateBrushIndirect** functions. Unlike other objects, brushes have distinct types that are not simply attributes. Hatch brushes are special because they use the current background mode (set with the **SetBkMode** function) for output.

Fonts are created with the **CreateFont** and **CreateFontIndirect** functions. An application can use either function to specify the 14 attributes that define the desired size, shape, and style of the logical font.

Bitmaps are created with the **CreateBitmap**, **CreateBitmapIndirect**, **CreateCompatibleBitmap**, and **CreateDIBitmap** functions. An application can use all four functions to specify the dimensions of the bitmap. An application uses the **CreateBitmap** and **CreateBitmapIndirect** functions to create a bitmap of any color format. The **CreateCompatibleBitmap** and **CreateDIBitmap** functions use the color format of the device context. A device supports two bitmap formats: monochrome and device-specific color. The monochrome format is the same for all devices. Using an output device context (DC) creates a bitmap with the native color format; using a memory DC creates a bitmap that matches the color format of the bitmap currently selected into that DC. (The DC ś color format changes based on the color format of the currently selected bitmap.)

Palette objects are created with the **CreatePalette** function. Unlike pens, brushes, fonts, and bitmaps, the logical palette created with this function can be altered later with the **SetPaletteEntries** function or, when appropriate, with the **AnimatePalette** function.

Regions can be created with the **CreateRectRgn**, **CreateRectRgnIndirect**, **CreateRoundRectRgn**, **CreateEllipticRgn**, **CreateEllipticRgnIndirect**, **CreatePolygonRgn**, and **CreatePolyPolygonRgn** functions. Internally, the region object that each function creates is composed of a union of rectangles with no vertical overlap. Regions created based on nonrectangular primitives simulate the complex shape with a series of rectangles, roughly corresponding to the scanlines that would be used to paint the primitive. As a result, an elliptical region is stored as many short rectangles (a bit fewer than the height of the ellipse), which leads to more cumbersome and slower region calculations and clipping. Coordinates used for creating regions are not specified in logical units as they are for other objects. The graphics device interface (GDI) uses them without transformation. GDI translates coordinates for clip regions to be relative to the upper-left corner of a window when applicable. Region objects can be altered with the **CombineRgn** and **OffsetRgn** functions.

What Happens During Selection

Selecting a logical object into a DC involves converting the logical object into a physical object that the device driver uses for output. This process is called *realization*. The principle is the same for all objects,

but the actual operation is different for each object type. When an application changes the logical device mapping of a DC (by changing the mapping mode or the window or viewport definition), the system re-realizes the currently selected pen and font before they are used the next time. Changing the DC's coordinate mapping scheme alters the physical interpretation of the logical pen's width and the logical font's height and width by essentially reselecting the two objects.

Pens are the simplest of objects. An application can use three attributes to define a logical pen—width, style, and color. Of these, the width and the color are converted from logical values to physical values. The width is converted based on the current mapping mode (a width of 0 results in a pen with a one-pixel width regardless of mapping mode), and the color is mapped to the closest color the device can represent. The physical color is a solid color (that is, it has no dithering). If the pen style is set to PS_INSIDEFRAME and the physical width is not 1, however, the pen color can be dithered. The pen style is recorded in the physical object, but the information is not relevant until the pen is actually used for drawing.

Logical brushes have several components that must be realized to make a physical brush. If the brush is solid, a physical representation must be calculated by the device driver; it can be a dithered color (represented as a bitmap with multiple colors that when viewed by the human eye approximates a solid color that cannot be shown as a single pixel on the device), or it can be a solid color. Pattern brush realization involves copying the bitmap that defines the pattern and, for color patterns, ensuring that the color format is compatible with the device. Usually, the device driver also builds a monochrome version of a color pattern for use with monochrome bitmaps. With device-independent bitmap (DIB) patterns, GDI converts the DIB into a device-dependent bitmap using **SetDIBits** before it passes a normal pattern brush to the device driver. The selection of a DIB pattern brush with a two-color DIB and DIB_RGB_COLORS into a monochrome DC is a special case; GDI forces the color table to have black as index 0 and white as index 1 to maintain foreground and background information. The device driver turns hatched brushes into pattern brushes using the specified hatch scheme; the foreground and background colors at the time of selection are used for the pattern. All brush types can be represented at the device-driver level as bitmaps (usually 8-by-8) that are repeatedly blted as appropriate. To allow proper alignment of these bitmaps, GDI realizes each physical brush with a brush origin. The default origin is (0,0) and can be changed with the **SetBrushOrg** function (discussed in more detail below).

The GDI component known as the font mapper examines every physical font in the system to find the one that most closely matches the requested logical font. The mapper penalizes any font property that does not match. The physical font chosen is the one with the smallest penalty. The possible physical fonts that are available are raster, vector, TrueType™ fonts installed in the system, and device fonts built into or downloaded to the output device. The logical values for height and width of the font are converted to physical units based on the current mapping mode before the font mapper examines them.

Selecting a bitmap into a memory DC involves nothing more than performing some error checking and setting a few pointers. If the bitmap is compatible with the DC and is not currently selected elsewhere, the bits are locked in memory and the appropriate fields are set in the DC. Most GDI functions treat a memory DC with a selected bitmap as a regular device DC; only the device driver acts differently, based on whether the output destination is memory or the actual device. The color format of the bitmap defines the color format of the memory DC. When a memory DC is created with **CreateCompatibleDC**, the default monochrome bitmap is selected into it, and the color format of the DC is monochrome. When an appropriate color bitmap (one whose color resolution matches that of the device) is selected into the DC, the color format of the DC changes to reflect this event. This behavior affects the result of the **CreateCompatibleBitmap** function, which creates a monochrome bitmap for a monochrome DC and a color bitmap for a color DC.

Palettes are not automatically realized during the selection process. The **RealizePalette** function must be explicitly called to realize a selected palette. If a palette is realized on a nonpalette device, nothing happens. On a palette device, the logical palette is color-matched to the hardware palette to get the best possible matching. Subsequent references to a color in the logical palette are mapped to the appropriate hardware palette color.

Nothing is actually realized when a clip region is selected into a DC. A copy of the region is made and placed in the DC. This new clip region is then intersected with the current visible region (computed by the system and defining how much of the window is visible on the screen), and the DC is ready for drawing. Calling **SelectObject** with a region is equivalent to using the **SelectClipRgn** function.

Memory Usage

The amount of memory each object type consumes in GDI's heap and in the global memory heap depends

on the type of the object. Figures below apply to Windows version 3.1 and might change in future versions.

The following table describes memory used for storing logical objects.

Object type	GDI heap use (in bytes)	Global memory use (in bytes)
pen	10 + sizeof(LOGPEN)	0
brush	10 + sizeof(LOGBRUSH) + 6	0
pattern brush	same as brush + copy of bitmap	
font	10 + sizeof(LOGFONT)	0
bitmap	10 + 18	32 + room for bits
palette	10 + 10	4 + (10 * num entries)
rectangular region	10 + 26	0
solid complex region	rect region + (6 * (num scans –1))	0
region with hole	region + (2 * num scans with hole)	0

When an object is selected into a DC, it may have corresponding physical (realized) information that is stored globally and in GDI's heap. The table below details that use. The size of realized versions of objects that devices maintain is determined by the device.

Object type	GDI heap use (in bytes)	Global memory use
pen	10 + 8 + device info	0
brush	10 + 14 + device info	0
font	55 (per realization)	font data (per physical font)
bitmap	0	0
palette	0	0
region	intersection of region with visible region	0

As a result of the font caching scheme, several variables determine how much memory a realized font uses. If two logical fonts are mapped to the same physical font, only one copy of the actual font is maintained. For TrueType fonts, glyph data is loaded only upon request, so the size of the physical font grows (memory permitted) as more characters are needed. When the font can grow no larger, characters are cached to use the available space. The font data stored for a single physical font ranges from 48 bytes for a hardware font to 120K for a large bitmapped font.

Physical pens and brushes are not deleted from the system until the corresponding object is deleted. The physical object that corresponds to a selected logical object is locked in GDI's heap. (It is unlocked upon deselection.) Similarly, a font "instance" is cached in the system to maintain a realization of a specific logical font on a specific device with a specific coordinate mapping. When the logical font is deleted, all of its instances are removed as well.

When the clip region intersects with the visible region, the resulting intersection is roughly the same size as the initial clip region. This is always the case when the DC belongs to the topmost window and the clip region is within the window's boundary.

Creating vs. Recreating

If an application uses an object repeatedly, should the object be created once and cached by the application, or should the application recreate the object every time it is needed and delete it when that part of the drawing is complete? Creating on demand is simpler and saves memory in GDI ś heap (objects do not remain allocated for long). Caching the objects within an application involves more work, but it can greatly increase the speed of object selection and realization, especially for fonts and sometimes for palettes.

The speed gains are possible because GDI caches physical objects. Although realizing a new logical pen or brush simply involves calling the device driver, realizing a logical font involves a cumbersome comparison of the logical font with each physical font available in the system. An application that wants to minimize font-mapping time should cache logical font handles that are expected to be used again. All previous font-mapping information is lost when a logical font handle is deleted; a recreated logical font must be realized from scratch.

Applications should cache palette objects for two reasons (both of which apply only on palette devices). Most importantly, because bitmaps on palette devices are stored based on a specific logical bitmap, using a different palette alters the bitmap ś coloration and meaning. The second reason is a speed issue; the foreground realization of a palette is cached by GDI and is not calculated after the first realization. A new foreground realization must be computed from scratch for a newly created palette (or a palette altered by the **SetPaletteEntries** function or unrealized with the **UnrealizeObject** function).

Stock Objects

During initialization, GDI creates a number of predefined objects that any application can use. These objects are called *stock objects.* With the exception of regions and bitmaps, every object type has at least one defined stock object. An application calls the **GetStockObject** function to get a handle to a stock object, and the returned handle is then used as a standard object handle. The only difference is that no new memory is used because no new object is created. Also, because the system owns the stock objects, an application is not responsible for deleting the object after use. Calling the **DeleteObject** function with a stock object does nothing.

Several stock fonts are defined in the system, the most useful being SYSTEM_FONT. This font is the default selected into a DC and is used for drawing the text in menus and title bars. Because this object defines only a logical font, the physical font that is actually used depends on the mapping mode and on the resolution of the device. A screen DC with a mapping mode of MM_TEXT has the system font as the physical font, but if the mapping mode is changed or if a different device is used, the physical font is no longer guaranteed to be the same. A change of behavior for Windows version 3.1 is that a stock font is never affected by the current mapping mode; it is always realized as if MM_TEXT were being used. Note that a font created by an application as a copy of a stock font does not have this immunity to scaling.

No stock bitmap in the system is accessible by means of the **GetStockObject** function, but GDI uses a default one-by-one monochrome bitmap as a stock object. This default bitmap is selected into a memory DC during creation of that DC. The bitmap ś handle can be obtained by selecting a bitmap into a freshly minted memory DC; the return value from the **SelectObject** function is the stock bitmap.

Error Handling

The two common types of errors associated with objects are failure to create and failure to select. Both are most commonly associated with low-memory conditions.

During the creation process, GDI allocates a block of memory to store the logical object information. When the heap is full, applications cannot create any more objects until some space is freed. Bitmap creation tends to fail not because GDI ś heap is full but because available global memory is insufficient for storing the bits themselves. Palettes also have a block of global memory that must be allocated by GDI to hold the palette information. The standard procedure for handling a failed object creation is to use a corresponding stock object in its place, although a failed bitmap creation is usually more limiting. An application usually warns the user that memory is low when an object creation or selection fails.

Out-of-memory conditions can also occur when a physical object is being realized. Realization also involves GDI allocating heap memory, and realizing fonts usually involves global memory as well. If the object was realized in the past for the same DC, new allocation is unnecessary (see the "Creating vs. Recreating" section). If a call to **SelectObject** returns an error (0), no new object is selected into the DC, and the

previously selected object is not deselected.

Another possible error applies only to bitmaps. Attempting to select a bitmap with a color format that does not match the color format of the DC results in an error. Monochrome bitmaps can be selected into any memory DC, but color bitmaps can be selected only into a memory DC of a device that has the same color format. Additionally, bitmaps can be selected only into memory DCs; they cannot be selected into a DC connected to an actual output device or into metafile DCs.

Some object selections do not fail. Selecting a default object (WHITE_BRUSH, BLACK_PEN, SYSTEM_FONT, or DEFAULT_PALETTE stock objects) into a screen DC or into a screen-compatible memory DC does not fail when the mapping mode is set to MM_TEXT. Also, a bitmap with a color format matching a memory DC always successfully selects into that DC. Palette selection has no memory requirements and always succeeds.

Deletion

All applications should delete objects when they are no longer needed. To delete an object properly, first deselect it from any DC into which it was previously selected. To deselect an object, an application must select a different object of the same type into the DC. Common practice is to track the original object that was selected into the DC and select it back when all work is accomplished with the new object. When a region is selected into a DC with the **SelectObject** or **SelectClipRgn** function, GDI makes a copy of the object for the DC, and the original region can be deleted at will.

```
hNewPen = CreatePen(1, 1, RGB(255, 0, 0));
if (hNewPen)
      hOldPen = SelectObject(hDC, hNewPen);
else
      hOldPen = NULL;              // no selection
      .                            // drawing operations
      .                            // (could be with old pen)
      .
if (hOldPen)
   SelectObject(hDC, hOldPen);   // deselect hNewPen (if selected)
if (hNewPen)
   DeleteObject(hDC, hNewPen);   // delete pen if created
```

An alternative method is to select in a stock object returned from the **GetStockObject** function. This approach is useful when it is not convenient to track the original object. A DC is considered "clean" of application-owned objects when all currently selected objects are stock objects. The three exceptions to the stock object rule are fonts (only the SYSTEM_FONT object should be used for this purpose); bitmaps, which do not have a stock object defined (the one-by-one monochrome stock bitmap is a constant object that is the default bitmap of a memory DC); and regions, which have no stock object and have no need for one.

```
hNewPen = CreatePen(1, 1, RGB(255, 0, 0));
if (hNewPen)
{
    if (SelectObject(hDC, hNewPen))
    {
     .
     .                              // drawing operations
     .
        SelectObject(hDC, GetStockObject(BLACK_PEN));
    }
    DeleteObject(hDC, hNewPen);
}
```

The rumor that an application should never delete a stock object is far from the truth. Calling the **DeleteObject** function with a stock object does nothing. Consequently, an application need not ensure that an object being deleted is not a stock object.

UNREALIZEOBJECT

The **UnrealizeObject** function affects only brushes and palettes. As its name implies, the **UnrealizeObject** function lets an application force GDI to re-realize an object from scratch when the object is next realized in a DC.

The **UnrealizeObject** function lets an application reset the origin of the brush. When a patterned, hatched, or dithered brush is used, the device driver handles it as an eight-by-eight bitmap. During use,

the driver aligns a point in the bitmap, known as the brush origin, to the upper-left corner of the DC. The default brush origin is (0,0). If an application wants to change the brush origin, it uses the **SetBrushOrg** function. This function does not change the origin of the current brush; it sets the origin of the brush for the next time that the brush is realized. The origin of a brush that has never been selected into a DC can be set as follows:

```
// Create the brush.
hBrush = CreatePatternBrush(.....);
// Set the origin as needed.
SetBrushOrg(hDC, X, Y);
// Select (and realize) the brush with the chosen origin.
SelectObject(hDC, hBrush);
```

If, on the other hand, the brush is currently selected into a DC, calling the **SetBrushOrg** function alone accomplishes nothing. Because the new origin does not take effect until the brush is realized anew, the application must force this re-realization by using the **UnrealizeObject** function before the brush is reselected into a DC. The following sample code changes the origin of a brush that is initially selected into a DC:

```
// Deselect the brush from the DC.
hBrush = SelectObject(hDC, GetStockObject(BLACK_BRUSH));
// Set a new origin.
SetBrushOrg(hDC, X, Y);
// Unrealize the brush to force re-realization.
UnrealizeObject(hBrush);
// Select (and hence re-realize) the brush.
SelectObject(hDC, hBrush);
```

The **UnrealizeObject** function can also be called for a palette object, although the effect is a bit more subtle. (As is common with the palette functions, nothing happens on a nonpalette device.) The function forces the palette to be realized from scratch the next time the palette is realized, thereby ignoring any previous mapping. This is useful in situations in which an application expects that the palette will realize differently the next time around, perhaps matching more effectively with a new system palette and not forcing a system palette change. Any bitmaps created with the original realization of the palette are no longer guaranteed to be valid.

Special Cases

Palette objects are selected into DCs using the **SelectPalette** function. The reason for this additional, seemingly identical, function is that palette selection has an additional parameter that defines whether the palette is being selected as a foreground or as a background palette, which affects palette realization on palette devices. Calling the **SelectObject** function with a palette returns an error. Palettes are deleted using the **DeleteObject** function.

A clip region can be selected into a DC by calling either the **SelectClipRgn** or the **SelectObject** function. Both functions perform identically with the exception of selecting a NULL handle in place of a region. **SelectClipRgn** can be used to clear the current clipping state by calling the function as follows:

```
SelectClipRgn(hDC, NULL);
```

This is not the same as selecting an empty region. Substituting the **SelectObject** function but maintaining the parameters in the code sample above results in an error.