

Appendix B. Programming The Windows GUI

B.1. Introduction

Many of the examples given in the text contain Windows operating system drawing functions. In this appendix, we give a brief introduction to using these functions. Quite complicated results can be achieved with a few simple functions, and the purpose of this section is to introduce you to some of the most useful functions.

B.2. Device Contexts

In the Windows operating system, all functions are associated with a device context. The device context is actually an object-oriented concept that was introduced into the Windows operating system before object oriented programming was widely used. For MFC components, the **CDC** class encapsulates the functionality of the device context, making drawing operations completely object oriented. From time to time, however, it is necessary to make a call to a non-object-oriented function.

The device context, for the most part, allows a single drawing routine to be used for several different purposes. For example, the same drawing routine can be used to draw the component window, print a copy of the component window, and to copy the component window drawing into the clipboard.

B.2.1. Window Device Contexts

A device context to the component window is provided as an operand of the *OnDraw* function. This parameter is named “*pdw*” and is a pointer to an object of type **CDC**. This

pointer must be used for all drawing functions called by the *OnDraw* routine. The **CDC** address can be passed to other subroutines, but it *cannot* be saved in the support object for use by other routines.

If you need to draw to the component window from some place other than the *OnDraw* routine, you will first need to obtain the address of the device context for the component window. This is done by executing the following statement within any support-class function.

```
CDC * MyDc = CWnd::GetDC();
```

When you are finished with the device context, you must give it back. If you fail to do this, you can gobble up system resources and eventually lock up your system. To give the device context back, you execute the following statement.

```
CWnd::ReleaseDC(MyDc);
```

In the *OnDraw* function, the *rcBounds* argument will give the size of the window. In most cases the *Top* and *Left* members of this rectangle will be zero, but this may not always be the case. To obtain this rectangle outside the *OnDraw* routine, execute the following statement.

```
CRect MyRect;  
CWnd::GetClientRect(&MyRect);
```

B.2.2. Memory Device Contexts

A memory device context allows you to draw into a bitmap that is not visible to the user. This can be done for a number of different reasons, but the most common reason is for smoothing animation. In such a case, you want to make sure that your bitmap is compatible with the frame-buffer memory, and that your device context is compatible with the device context of the component window. To draw into a bitmap that will eventually be copied into the component window, execute the statements of Figure B.1. We assume that these appear in the *OnDraw* function.

```
CBitmap XMap,*OldMap;  
// create a bitmap with the same properties as the frame buffer  
XMap.CreateCompatibleBitmap(pdc,rcBounds.Width(),rcBounds.Height());  
CDC WorkDC;  
CDC *Mydc=&WorkDC;  
// create a device context with the same properties as the component  
// window device context  
Mydc->CreateCompatibleDC(pdc);  
// associate the bitmap with the device context.  
OldMap = Mydc->SelectObject(&XMap);
```

Figure B.1. Creating a Memory Device Context.

After executing the code of Figure B.1, all drawing with the device context *Mydc* will be directed into the bitmap *Xmap*.

Once the device context and bitmap are no longer required, they must be destroyed using a procedure similar to that of Figure B.2.

```
// select out the bitmap  
Mydc->SelectObject(OldMap);  
// delete the bitmap  
XMap.DeleteObject();  
// delete the device context  
Mydc->DeleteDC();
```

Figure B.2. Deleting a Memory Device Context.

B.2.3. Metafile Device Contexts

A metafile is a recorded list of drawing commands that can be replayed to create a drawing in several different locations. Metafiles are used to copy drawings to the clipboard, save drawings to disk, and for various other purposes. To create a device context for a metafile, execute the following two statements

```
HDC MfHdc = CreateMetaFile(NULL);  
CDC * MyDc = CDC::FromHandle(MfHdc);
```

After executing these two statements, you can use the device context *MyDc* to draw into the metafile. You can use the same drawing routine that you use for window and memory device contexts. Once you are finished drawing you need to create a metafile handle for the metafile you have just created. To do this, execute the following statement.

```
HMETAFILE MHandle = CloseMetaFile(MfHdc);
```

The metafile handle can now be used by various metafile processing functions. This statement destroys the metafile device context.

B.2.4. Printer Device Contexts

A printer device context works like other device contexts, but is used to create output for a printer. There are a few additional function calls that must be used with a printer device context.

Once you have the device context, you must start the printing process by calling the *StartDoc* function. The *StartDoc* function has one argument, which must be a pointer to a **DOCINFO** structure (this structure is documented in the MSDN library.)

To start each page you must call the *StartPage* function. At the end of the page you must call the *EndPage* function, and at the end of the document, you must call the *EndDoc* function. Figure B.3 shows how to print a one page document with a large X on it.

```
MyDc->StartDoc(&MyDoc);  
MyDc->StartPage();  
MyDc->MoveTo(0,0);  
MyDc->LineTo(3000,3000);  
MyDc->MoveTo(3000,0);  
MyDc->LineTo(0,3000);  
MyDc->EndPage();  
MyDc->EndDoc();
```

Figure B.3. Printing an X.

On a 600 DPI printer, the code of Figure B.3 will print an X that is 5 inches on a side.

B.3. Pens and Brushes

In addition to being associated with a particular device, the device context also contains a collection of drawing resources. We have seen one of these already. The memory device context owns the bitmap into which the drawings are made. Two additional types of drawing resources are pens and brushes. The pen determines how lines are drawn, while the brush determines the fill color for closed figures like rectangles and ellipses.

Figure B.4 shows how to create a green brush. After executing these statements, all lines drawn using the device context *Mydc* will be drawn using green lines of minimum

width. To use a larger width, the second argument of *CreatePen* should be set to something larger than zero. The third argument determines the color of the pen, while the first argument is a system-defined constant that specifies solid lines (as opposed to dotted or dashed lines.)

```
CPen MyPen;  
MyPen.CreatePen(PS_SOLID, 0, RGB(0, 255, 0));  
CPen * OldPen = Mydc->SelectObject(&MyPen);
```

Figure B.4. Creating a Green Pen.

If you create a pen, you must destroy it when you are done using it. Figure B.5 shows how to destroy the green pen that we created in Figure B.4. Before the pen can be destroyed, it must be selected out of the device context.

```
Mydc->SelectObject(OldPen);  
MyPen.DeleteObject();
```

Figure B.5. Deleting The Green Pen.

Creating a brush is quite similar to creating a pen. Unlike a pen, a brush does not have a width, or a style specification. Figure B.6 shows how to create a brush, and Figure B.7 shows how to delete it when you are finished with it. When this brush is selected into the *Mydc* device context, all rectangles and ellipses will be filled with the magenta color. To prevent filling of these shapes, use the stock object *HOLLOW_BRUSH*. (See *GetStockObject* in the C++ documentation.)

```
CBrush MyBrush;  
MyBrush.CreateSolidBrush(RGB(255, , 255));  
CBrush * OldBrush = Mydc->SelectObject(&MyBrush);
```

Figure B.6. Creating a Magenta Brush.

```
Mydc->SelectObject(OldBrush);  
MyBrush.DeleteObject();
```

Figure B.7. Deleting The Magenta Brush.

The main use of pens and brushes is to set the line and fill colors for drawing shapes and lines. There are other uses for these objects, and these objects have many functions that we have not described. See the C++ documentation for more information.

B.4. Useful Drawing Functions

Line drawing is performed using the **CDC** functions *LineTo(NewX,NewY)* and *MoveTo(NewX,NewY)*. Every device context has a current pen position. The *LineTo* function moves the pen from the current position to the position defined by *(NewX,NewY)*, and draws a line between these two points. The function *MoveTo* moves the pen in the same way, but does not draw a line.

The **CDC** function *Rectangle* is used to draw a rectangle with the current pen, and fill it using the current brush. *Rectangle* has four parameters, *Left*, *Top*, *Right*, and *Bottom*, which define the bounds of the rectangle.

The **CDC** function *Ellipse* works exactly like the *Rectangle* function, but draws the inscribed ellipse of the rectangle rather than the rectangle itself.

Other **CDC** functions can be used to draw Polygons, Polylines (non-closed polygons) perform Bitblt operations, and to fill rectangular areas with color. One additional function that has been used in a few examples in this book is the *FloodFill* function. An example of a call to this function is **cdc->FloodFill(2,2,RGB(0,0,0))** This function call begins a flood fill operation at point (2,2) in the component window, and replaces all black pixels in the flood-fill area with pixels of the current brush color.

Another function that is widely used in the examples is the `FillRect` function which is generally used to paint background areas a single color. To use the `FillRect` function, one must create a brush, or obtain a stock brush (see *GetStockObject* in the C++ documentation). Figure B.8 shows how to fill an area with a green brush.

```
CBrush MyBrush;  
MyBrush.CreateSolidBrush(RGB(0,255,0));  
Mydc->FillRect(&rcBounds,&MyBrush);  
MyBrush.DeleteObject();
```

Figure B.8. Doing a Background Fill.

There are a few text functions that are used in the examples, the most important of which is *TextOut*. *TextOut* has three parameters, the first two are the coordinates of the upper left corner of the rectangle that will contain the text, and the third is the text itself in the form of a null-terminated string of characters. All text drawn with the same Y coordinate will have the same base line.

If you want to know the size of the rectangle required to contain a particular text string, use the **CDC** *GetTextExtent* function. You can draw text in different fonts by selecting a font resource into the drawing context. You normally obtain the font resource from the stock font property. You can change the text foreground and background colors by using the **CDC** *SetTextColor*, and *SetBkColor* functions. The *SetBkColor* applies only to the background of the rectangle in which text is drawn.

B.5. Scaling and Scrolling

The capability of the Simple Graphical Editor would be greatly enhanced if it had the ability to scale and scroll drawings. Both of these things can be done fairly simply using

standard Windows operating system functions. In this section we will show how to perform some of the basic operations.

B.5.1. Scroll-Bar Management

For scrolling features to be useful, it is first necessary to enable the scroll bars in the component window. This is done by adding a *PreCreateWindow* function to our support class. This function is added using the first tab of the class wizard menu. In this function, we can add a vertical scroll bar, a horizontal scroll bar, or both. Figure B.9 (which is taken from Chapter 10) shows how to add both a horizontal and a vertical scroll bar. To create a single scroll bar, *OR* just one of the constants into `cs.style`.

```
BOOL CTextViewerCtrl::PreCreateWindow(CREATESTRUCT& cs)
{
    cs.style |= WS_HSCROLL | WS_VSCROLL;
    return COleControl::PreCreateWindow(cs);
}
```

Figure B.9 Adding Scroll Bars to a Control Window.

Once the scroll bars are added, it is up to your program to determine how they will be used. A scroll bar is a UI Widget that does nothing more than pass operating system messages to your component. You must determine how the scroll bars will affect the display, and you must handle all scrolling operations yourself. The first thing you must do is set the range of each scroll bar. The default range is 0-0, which will not permit the user to move the slider. The scroll bar ranges are set using the following functions.

```
CWnd::SetScrollRange(SB_VERT, 0, VMax);
CWnd::SetScrollRange(SB_HORZ, 0, HMax);
```

The first operand of the `SetScrollRange` function determines which scroll bar will be affected. The second operand is the minimum value, and the third is the maximum value. Either value may be negative. The size of the range determines the number of allowable slider positions. These functions are normally called from a spot where the size of the document being displayed is known.

The next step is to set the scroll bar position. Never assume that the position will be set for you automatically. Regardless of how a new scroll bar position is determined, you should always set the position yourself. You do this using the following two function calls.

```
CWnd::SetScrollPos(SB_HORZ,HPos);  
CWnd::SetScrollPos(SB_VERT,VPos);
```

The first operand of the `SetScrollPos` function determines which scroll bar will be affected. The second gives the absolute position of the slider. The second operand must be an integer in the current range of the scroll bar. You normally start by setting both scroll bar sliders to their minimum position, and then change the position as scroll-bar messages are processed.

To process scroll bar messages, you must add an *OnVScroll* function and an *OnHScroll* function to your support class. (If you have only one scroll bar, you should add only the function you need.) These two functions are added using the first tab of the class wizard dialog box. The functions themselves will not appear in the list of available functions, you must add handlers for the Windows operating system messages *WM_HSCROLL* and *WM_VSCROLL*.

The first operand of the *OnVScroll* and *OnHScroll* functions is a code telling you what operation the user has performed on the scroll bar. The name of this operand is *nSBCode*. You should code a switch statement to decode this operand. There is a set of predefined constants that you can use to make this job somewhat easier. Some of the codes are different for horizontal and vertical scroll bars. Figure B.10 gives a complete list of the codes and their meanings.

Code	Meaning
SB_TOP	The user has moved the slider of the vertical scroll bar all the way to the top.
SB_LEFT	The user has moved the slider of the horizontal scroll bar all the way to the left.
SB_BOTTOM	The user has moved the slider of the vertical scroll bar all the way to the bottom.
SB_RIGHT	The user has moved the slider of the horizontal scroll bar all the way to the right.
SB_LINEDOWN	The user has clicked the button at the bottom of the vertical scroll bar.
SB_LINERIGHT	The user has clicked the button at the right end of the horizontal scroll bar.
SB_LINEUP	The user has clicked the button at the top of the vertical scroll bar.
SB_LINELEFT	The user has clicked the button at the left end of the horizontal scroll bar.
SB_PAGEDOWN	The user has clicked in the slider area of the vertical scroll bar, but below the slider.
SB_PAGERIGHT	The user has clicked in the slider area of the horizontal scroll bar, but to the right of the slider.
SB_PAGEUP	The user has clicked in the slider area of the vertical scroll bar, but above the slider.
SB_PAGELEFT	The user has clicked in the slider area of the horizontal scroll bar, but to the left of the slider.
SB_THUMBPOSITION	The user has dragged the slider of either scroll bar to a new position. The new position is given in the second operand of the function, <i>nPos</i> .
SB_THUMBTRACK	The user is in the process of dragging the slider of either scroll bar. This message allows you to synchronize the display with the movement of the scroll bar. The current position of the slider is in the second operand of the function, <i>nPos</i> .
SB_ENDSCROLL	Signals the end of a set of <i>SB_THUMBTRACK</i> events. Will normally be followed by an <i>SB_THUMBPOSITION</i> event.

Figure B.10. Scroll Bar Message Codes.

It is best to keep track of the current scroll bar position using a tracking variable, but you can also query the scroll bar for its current position. Your response to most of the message codes, will be to change the value of your tracking variable, reset the scroll bar position using the `SetScrollPos` function, and then invalidating the display to force a redraw with the new scroll bar position. Handling the *SB_THUMBTRACK* message

usually requires special drawing techniques to avoid flicker. Such techniques are beyond the scope of this appendix.

The amount of change in the scroll-bar position for the codes *SB_LINEUP*, *SB_LINEDOWN*, *SB_LINERIGHT*, *SB_LINELEFT*, *SB_PAGEUP*, *SB_PAGEDOWN*, *SB_PAGELEFT*, and *SB_PAGERIGHT* must be determined by you. Normally you will adjust the slider position by a constant value for each of these codes. It is sometimes necessary to experiment with the values to achieve a pleasing result.

B.5.2. Window and Viewport origin

If your document is relatively small, you can do all of your scrolling using the **CDC** *SetWindowOrg* function. Normally, the upper left corner of the component window is at the coordinates (0,0). The *SetWindowOrg* function changes the coordinates of the upper left to whatever coordinates you specify. If you set the coordinates to values larger than zero, this has the effect of moving a different portion of your document into the visible portion of the window. After calling *SetWindowOrg*, you draw your document normally. The apparent movement of the document is handled automatically.

Suppose you have a component window that is 200 pixels wide and 100 pixels high, and suppose your document is 400 pixels wide and 200 pixels high. If you draw this document without calling *SetWindowOrg*, only the upper left quadrant of the document will be visible. Figure B.11 illustrates the four quadrants of the document, and the *SetWindowOrg* call that must be used to make each quadrant visible.

<code>pdcc->SetWindowOrg(0,0)</code>	<code>pdcc->SetWindowOrg(200,0)</code>
<code>pdcc->SetWindowOrg(0,100)</code>	<code>pdcc->SetWindowOrg(200,100)</code>

Figure B.11. The Effect of SetWindowOrg.

If you use tracking variables for your scroll bars, the tracking variables can be used as *SetWindowOrg* operands.

B.5.3. Window and Viewport extent

Just as the origin can be used for scrolling, the window and viewport extents can be used for scaling. The window and viewport extents provide a pair of numbers that are used to scale output to a device context. The ratio of the extents is important, the absolute value of the extents is not. (To avoid overflow problems, both values should be as small as possible.) Normally the window and viewport extents are the same. When they are not the same, the coordinates of all drawn objects are multiplied by the viewport extent and then divided by the window extent. A different scale can be used for the X and Y directions. Let's suppose we want to double the size of our drawing, to make it appear 200% of its original size. We would make the following function calls before drawing our document.

```
pdC->SetWindowExt(1,1);  
pdC->SetViewportExt(2,2);
```

If instead we wanted to draw our document smaller, at 50% of its original size, we would use the following function calls.

```
pdC->SetWindowExt(2,2);  
pdC->SetViewportExt(1,1);
```

Scaling is especially important when drawing graphical objects to a printer. When doing this, use the function calls of Figure B.12. (Assume that *MyDc* is a printer device context.) The *GetDeviceCaps* function calls are used to determine the horizontal and vertical resolution of the printer. The display has a logical resolution of 100 pixels per inch. If you wish, you can use similar function calls to determine the precise resolution of the display, but things will look right if you use 100.

```
long PixX = MyDc->GetDeviceCaps(LOGPIXELSX);  
long PixY = MyDc->GetDeviceCaps(LOGPIXELSY);  
// Start a document and a page  
MyDc->StartDoc(&MyDoc);  
MyDc->StartPage();  
// This stuff MUST DEFINITELY follow StartDoc and StartPage  
// MM_ANISOTROPIC allows independent scaling of X and Y dimensions.  
MyDc->SetMapMode(MM_ANISOTROPIC);  
MyDc->SetWindowExt(100,100);  
MyDc->SetViewportExt(PixX,PixY);
```

Figure B.12. Scaling Print Output.

Instead of using the window and viewport extents, you could perform all scaling operations yourself. This is fairly simple for graphical objects, but is nightmarishly difficult for text. If your document contains text, don't try to scale it yourself.

B.6. Bitmaps, Icons and other Resources

The *Cards* and *LED* examples used bitmap resources to draw their displays. To use a bitmap resource, first draw the bitmap using the Visual C++ bitmap editor, or some other tool, and save it as a .bmp file. (This is done behind-the-scenes in the Visual C++ editor.) Then include the bitmap in your component as a resource. (It is fairly obvious how to do this.) Bitmaps drawn in the Visual C++ editor get added to your resources automatically.

Once you have your bitmap stored as a resource, you need to load it into memory and then map it into your component window. This is done using a memory device context. First, you load the bitmap from your resources using the following statement.

```
CBitmap MyMap;  
MyMap.LoadBitmap(IDB_MYMAP);
```

The operand *IDB_MYMAP* is the name of the resource as it appears in the resource editor. Once the bitmap has been loaded into memory, it must be selected into a device context. The device context must be compatible with the display. This is done using the following statements.

```
CDC MyDc  
MyDc.CreateCompatibleDC(pdc);  
CBitmap *OldMap = MyDc->SelectObject(&MyMap);
```

To draw the bitmap at position (X, Y) of your component window use the following function call. In this function call, *Width* and *Height* are the width and height of the bitmap in pixels.

```
pdC->BitBlt(X,Y,Width,Height,&MyDc,0,0,SRCCOPY);
```

Before exiting your *OnDraw* function, you should select the bitmap out of the *MyDc* context using the following function call.

```
MyDc.SelectObject(OldBitmap);
```

The destructor of the **CBitmap** class will take care of removing the bitmap from memory.

There are several standard resources that are created when you use the MFC ActiveX wizard. The first is the bitmap that will be used to represent your component in the Visual Basic tool box. The default is the letters OCX in purple on a gray background. If you make more than one ActiveX component, these bitmaps can quickly become confusing. You should edit this bitmap to identify your component. You don't need to be creative, a few meaningful letters will do. You can enlarge the bitmap if it is too small. Note that the color of the edge of the bitmap will be treated as the background color and replaced by gray when the bitmap appears in the Visual Basic toolbox.

The next standard resource is the about-box icon. This is the standard MFC building blocks icon. This icon will appear in your about box, and wherever the Windows operating system needs an Icon to identify your component. (This happens only rarely.) You should edit this icon to provide some useful information.

Next is the about-box dialog. You should change the copyright notice, and provide the long name for your component. You can provide additional lines of text as necessary. The property page dialog is discussed in Appendix C.

The String Table resource gives the names that will be used for your component in the registry. It is seldom necessary to change the defaults. The intent of the string table is to provide translated names for components that are distributed in other countries.

The Version resource is discussed in Appendix D.

B.7. Conclusion

This appendix has given a brief outline of Windows GUI programming which will be sufficient for most simple ActiveX components. As you gain experience you will want to enhance your knowledge of the GUI drawing functions. The Visual C++ documentation contains an enormous amount of material that will help you broaden your knowledge of this subject.