Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

Tech Tips

October 01, 2002
URL:http://www.drdobbs.com/tech-tips/184416581

Font Creation and Rounding Differences

Philip Hamer
phiham@hotmail.com

Recently I had the problem of creating a font given, among various other attributes, the point size. In many applications, you may never need to do this. Microsoft provides the **ChooseFont()** API function (and its MFC wrapper counterpart CFontDialog) that gives the user a dialog, allows selection of point size, and automatically fills in the information that is needed in its LOGFONT struct. The important field in this structure is the lfHeight value, which is the logical height. You can then pass this to **CreateFont()** or **CreateFontIndirect()** to create a font.

But what if you don't have the logical height? As explained in the LOGFONT documentation and in Microsoft Knowledge Base article Q74299, you can, for the MM_TEXT mapping mode, get the logical height of a font using this formula:

```
Height = -MulDiv(PointSize, GetDeviceCaps(hDC, LOGPIXELSY), 72)
```

The MulDiv function is useful for multiplying two possibly large 32-bit values and then dividing by another 32-bit value without causing overflow. However, a side effect is that it also rounds the result to the nearest integer. So the aforementioned formula is not necessarily equal to:

```
Height = -PointSize * GetDeviceCaps(hDC, LOGPIXELSY) / 72
```

since **Height** would represent some integral type that will receive the truncated result of the arithmetic — no rounding, just "chopping off."

I was using MFC, so I figured why not let MFC do all this. The CFont class provides the CreatePointFontIndirect() method: Give it a LOGFONT structure (with 10 times the desired point size in the lfHeight field) and it converts it to logical units. You can even pass in a device context and get logical units even if you're not in MM_TEXT mode. Wow, MFC is amazing! Or is it? Turns out that **CFont::CreatePointFontIndirect()** doesn't use MulDiv, so the value is truncated. Worse yet, **ChooseFont()** seems to do the rounding like MulDiv does. So if you create a font with **CFont::CreatePointFontIndirect()** or any similar nonrounding method, display text with that font, and allow the user to change the size of that text with **ChooseFont()** (or CFontDialog), then even if the user selects the same point size, the text may change because the underlying **lfHeight** value may be different. Included in Listing 1 is a better routine, similar to **CFont::CreatePointFontIndirect()**, except that it is in C to be more generic and it uses MulDiv. Moral of the story: Use **MulDiv** for font creation.

How to Relocate GetOpenFileName Dialogs

Stephen Schumacher
Steve@GHSports.com

The **GetOpenFileName** common dialog box (and its near clone **GetSaveFileName**) can be customized by enabling a hook function to process messages intended for the dialog box. One use for this hook function is to intercept the WM_INITDIALOG message in order to rename some of the controls appearing in the dialog, such as the IDOK button. Another potential use is to move the file dialog box to the center of its parent window or any place else besides the upper left corner of its parent's client area (which is the default location). Use of this hook function is documented in the Microsoft Developer Network (MSDN), available on the Web.

The unwary programmer trying to actually use this feature will run into two problems, one poorly documented, the other undocumented. First, according to Paul DiLascia from *Microsoft Systems Journal* (January 1998): Windows "doesn't hook the dialog directly, but instead creates an empty child dialog, which is the hooked window...So if you try to set the text of a child button with SetDlgItemText(IDOK), it fails because the child dialog has no OK button — or indeed any controls at all." So instead of using the dialog window handle passed to the hook function, you have to be careful to use its parent window

handle (returned from **GetParent**) when trying to affect the file dialog box.

The other problem pertains to trying to relocate the dialog box using SetWindowPos or MoveWindow. One would assume that this action should take place in response to the WM_INITDIALOG message — as done in the article "Centering the Open & Save Common Dialogs Using Callbacks" at www.mvps.org/vbnet and other examples on the Web. However, the file dialog later resizes itself, presumably to match flag options specifying which controls to include in the dialog. As a result, the dimensions found from doing GetWindowRect at WM_INITDIALOG time are not accurate, so any attempt at centering in fact produces a dialog box that is slightly out of position. Relocating the dialog box in response to the WM_SIZE message produces the desired result.

But there is an additional and worse problem related to relocating a file dialog box in response to WM_INITDIALOG. Windows 95 has a bug (fixed in later Windows operating systems) so that any effort to relocate using a WM_INITDIALOG hook results in a vertically truncated or expanded file dialog box, with dimensions strangely correlated to the target y-coordinate. Sometimes the dialog box is so severely truncated that only its title bar appears on the screen. The solution, again, is to do SetWindowPos or MoveWindow only when handling the hooked WM_SIZE message, which entirely sidesteps this problem.

Listing 2 presents a sample hook procedure for relocating a file dialog box to the center of the screen's work area (excluding the system taskbar and any application toolbars). The sample is written in the beautiful language Modula-2, which is immanently suited for professional Windows programming!

Querying Multiple IDispatch Method Identifiers
Matthew Wilson
matthew@synesis.com.au

The **IDispatch** interface is at the foundation of COM scripting, supporting Visual Basic, VBScript, JavaScript, and the like. The interface is used to execute arbitrarily named methods that may also have named arguments. Support for this type of flexibility is based around translating such names into identifiers (known as DISPIDs) that are then used to call the methods.

The sixth member of the **IDispatch** interface, **GetIDsOfNames**, is used to elicit the method DISPID and zero or more argument DISPIDs for a particular method. For example, consider the following IDL definition:

```
[id(1), helpstring("Retrieves system message for given error")]
HRESULT TranslateError( [in] long err,
[in, defaultvalue(0)] long flags,
[in, defaultvalue(0)] BSTR strModule,
[out, retval] BSTR *pbstr);

[id(2), helpstring("Restarts system according to given flags")]
HRESULT Restart([in] long flags);
```

Here we might wish to query the **TranslateError** method and argument IDs, as in:

```
LPCOLESTR rgszNames[] =
{
L"TranslateError"
, L"strModule"
, L"flags"
, L"err"
};
const cNames = sizeof(rgszNames) / sizeof(rgszNames[0]);
DISPID rgdispid[cNames];

hr = pitf->GetIDsOfNames( IID_NULL,
(LPOLESTR*)rgszNames,
cNames,
LOCALE_SYSTEM_DEFAULT,
rgdispid);
```

However, when a method has no arguments or has a well-known array of arguments, what is required is not the DISPID of a method and its arguments, but rather the DISPIDs of several methods. Since GetIDsOfNames can only return the DISPID of one method per call, I wrote the function **Dispatch_GetMethodIDs** (see Listing 3) to assist in making such multiple method DISPID queries.

The function has a very similar set of arguments to **GetIDsOfNames**, with the addition of the first parameter that is the interface whose Dispatch members are to be queried.

punk is of type **IUnknown*** rather than **IDispatch***, allowing for cleaner client code where the **IDispatch** interface has not already been obtained. This may be less efficient when **IDispatch** is already available, but the cost of this extra query will likely be insignificant compared to the other operations on the interface since **IDispatch** is a comparatively costly interface to use. **riid** and **lcid** are passed straight through to each call to **GetIDsOfNames**. **rgszNames** is an array of one or more method names, whose DISPIDs are placed into the corresponding elements of **rgdispid**. **cNames** is the dimension of the **rgszNames** and **rgdispid** arrays.

rgszNames is of type LPCOLESTR[], as opposed to **GetIDsOfNames** LPOLESTR[], since it is entirely an in parameter, and we can only assume that the original definition of the **GetIDsOfNames** was an oversight. This type makes it easier to formulate these arrays in client code.

The function shadows the semantics of **GetIDsOfNames**, in so far as when a DISPID is not available, the elicitation stops at that point and the function returns DISP_E_UNKNOWNNAME, thereby providing maximum information to client code.

Partial Array Expansion in the Microsoft Visual Studio Debugger
Gigi Sayfan
the_gigi@hotmail.com

When you look at an array in the watch window, it appears by name only with an expandable '+' sign. If you click on the '+', the entire array is unfurled. If your array is very large (several thousand entries), the expansion could take a long time. During development, you often need to check only a limited number of entries to make sure the array contains sensible data. One solution is to put separate entries in the watch window:

```
array[0]
array[1]
array[2]
array[3]
```

However, the Microsoft Visual Studio Debugger allows you to do a partial array expansion as follows:

```
array, 4
```

Now you will get the regular '+', but when you click on it, only four entries are displayed.

Sometimes you might like to see a range such as items 8-11. Here you need to specify array+8, 4. Note that the indices in the watch window are still 0 based. It's up to you to do the math. Unfortunately, the expansion trick doesn't work for objects.

George Frazier is a software engineer in the System Level Design group at Cadence Design Systems and has been programming for Windows since 1991. He can be reached at georgefrazier@yahoo.com.

Listing 1: Using MulDiv for font creation

```
/*
* Initialize lfHeight to 10 times the desired point size. Pass a HDC to use
* to convert to logical units, or NULL to use a screen DC.
*/
HFONT CreatePointFontIndirect(const LOGFONT *pLF, HDC hDC)
{
             LOGFONT lf = *pLF;
             POINT pt, ptOrg = {0,0};
             BOOL bScreenDC = (hDC == NULL);

             if (bScreenDC)
                          hDC = GetDC(NULL);  /* Get screen DC */

             pt.y = MulDiv(lf.lfHeight, GetDeviceCaps(hDC, LOGPIXELSY), 720);
             DPtoLP(hDC, &pt, 1);
             DPtoLP(hDC, &ptOrg, 1);
             lf.lfHeight = -abs(pt.y - ptOrg.y);

             if (bScreenDC)
                          ReleaseDC(NULL, hDC);

             return CreateFontIndirect(&lf);
}
```

Listing 2: A sample hook procedure

```
/*
* Initialize lfHeight to 10 times the desired point size. Pass a HDC to use
* to convert to logical units, or NULL to use a screen DC.
*/
HFONT CreatePointFontIndirect(const LOGFONT *pLF, HDC hDC)
{
             LOGFONT lf = *pLF;
             POINT pt, ptOrg = {0,0};
             BOOL bScreenDC = (hDC == NULL);

             if (bScreenDC)
                          hDC = GetDC(NULL);  /* Get screen DC */

             pt.y = MulDiv(lf.lfHeight, GetDeviceCaps(hDC, LOGPIXELSY), 720);
             DPtoLP(hDC, &pt, 1);
             DPtoLP(hDC, &ptOrg, 1);
             lf.lfHeight = -abs(pt.y - ptOrg.y);

             if (bScreenDC)
                          ReleaseDC(NULL, hDC);

             return CreateFontIndirect(&lf);
}
```

Listing 3: Making multiple method DISPID queries

```
SYFNCOMDECL Dispatch_GetMethodIDs(  LPUNKNOWN   punk,
                                   REFIID      riid,
                                   LPCOLESTR   rgszNames[],
                                   UINT        cNames,
                                   LCID        lcid,
                                   DISPID      rgdispid[])
{
    HRESULT hr;
    UINT    i;
```

```
    if( punk == NULL ||
        rgszNames == NULL ||
        rgdispid == NULL)
    {
        hr = E_POINTER;
    }
    else if(cNames < 1)
    {
        hr = E_INVALIDARG;
    }
    else
    {
        LPDISPATCH  pdisp;

        hr = punk->QueryInterface(IID_IDispatch, (void**)&pdisp);

        if(SUCCEEDED(hr))
        {
            BOOL    bUnknownDispid = false;

            for(hr = S_FALSE, i = 0; i < cNames; ++i)
            {
                hr = pdisp->GetIDsOfNames(  riid,
                                            (LPOLESTR*)rgszNames + i,
                                            1,
                                            lcid,
                                            rgdispid + i);

                if(SUCCEEDED(hr))
                {
                    continue;
                }
                else if(hr == DISP_E_UNKNOWNNAME)
                {
                    rgdispid[i]     =   DISPID_UNKNOWN;
                    bUnknownDispid  =   true;
                }
                else
                {
                    break;
                }
            }

            if(bUnknownDispid)
            {
                hr = DISP_E_UNKNOWNNAME;
            }

            pdisp->Release();
        }
    }

    return hr;
}
```

Terms of Service | Privacy Statement | Copyright © 2020 UBM Tech, All rights reserved.