June 1997

MICROSOFT SYSTEMS JOURNAL

More Fun with MFC: DIBs, Palettes, Subclassing, and a Gamut of Goodies, Part III

Paul DiLascia

Quick View is one of the new shell extensions supported in Windows 95 and Windows NT 4.0. If you right-click a file in Windows 95 and your system has a viewer installed for that file type, one of the commands in the context menu that appears is Quick View.

This article assumes you're familiar with C++, MFC, COM

Code for this article: FileViewer.exe (46KB)

Paul DiLascia is a freelance software consultant specializing in training and software development in C++ and Windows. He is the author of Windows++: Writing Reusable Code in C++ *(Addison-Wesley, 1992).*

This article is the last in a saga that began with "More Fun with MFC: DIBs, Palettes, Subclassing, and a Gamut of Reusable Goodies" in the January 1997 issue of MSJ and continued in the March 1997 issue. If you recall, I was hired by Acme Corporation, a multiquadrillion- dollar conglomerate with offices throughout the solar system, to write a Quick View file viewer for their XYZ documents. **Figure 1** shows the updated Master Plan I'm using to approach this feat. In my previous articles, I showed you how to build a super DIBVIEW program for viewing device-independent bitmaps (DIBs). I chose DIBs because the real Acme XYZ format is proprietary, but uses DIBs. I showed you how to implement a CDib class for handling DIBs, and a universal palette-message handling class, CPalMsgHandler, that handles palette messages automatically using my very own homebrewed hook scheme for doing multiple subclassing in MFC (it really is cool—you should check it out). I also built some debugging tools and a CFontUI that does the UI for growing, shrinking, or setting a font.

GOAL: Earn money to pay mortgage.

Subgoal: Write a Quick View file viewer for Acme Corp's XYZ documents.

PLAN (work from simple to complex):

1. First, write a normal MFC doc/view program to view XYZ documents. (More Fun with MFC, Parts 1 and 2)

2. Next, write a simple file viewer for a simple document type, like text, in order to learn how viewers work.
3. Encapsulate the simple file viewer from Step 2 to a general-purpose mini framework that supports any kind of MFC doc/view.
4. Plop doc/view classes from XYZ viewer app in Step 1 into viewer framework from step 3.


PRAY

- Collect paycheck.


- Pay mortgage.

- Write about it.

Unfortunately, after spending all that time writing lots of fun code, I'm only up to step 1 of my Master Plan, and His Excellency the Maximum Leader at Acme has already left three messages on my satellite pager. So now it's time to finally convert DIBVIEW into what it was always intended to be: a Quick View file viewer.

Of course, those of you who read the prequels know I could never write anything disposable. The whole point of these articles is to illustrate the goodies-as-you-go philosophy of software construction, which dictates that whenever you solve some new programming problem you encapsulate the solution in reusable classes. As the Master Plan suggests, I didn't just build a one-off DIB viewer. I encapsulated my viewer code in a fully reusable, generic viewer framework into which you can plop your favorite MFC doc/view classes and prest0—instant viewer! So the next time someone hires me to write a file viewer, I can do it in five minutes and bill them for two weeks.

Quick View Review

Don't feel ignorant if you're wondering, "what's Quick View?" It's a relatively new thing. Quick View is one of the new shell extensions supported in Windows® 95 and Windows NT® 4.0. If you right-click a file in Windows 95 and your system has a viewer installed for that file type, one of the commands in the context menu is Quick View (see **Figure 2**). If you click it, Windows runs the program QUIKVIEW.EXE, which in turn invokes the appropriate viewer for the file type. The viewer itself is a COM object that implements an interface called IFileViewer. By the time you're done reading this article, you'll know more about IFileViewer than anyone should.

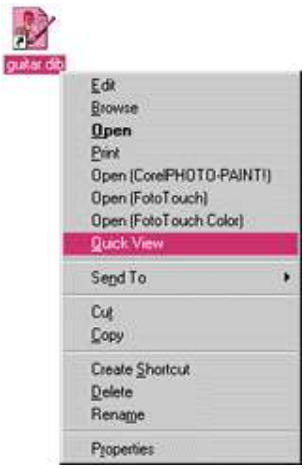

Figure 2: Quick View

Figure 3 shows Quick View displaying a Microsoft® Excel file. There's no fancy formatting, fonts, or colors—just the raw content. And you can only look, not touch. That is, you can view the spreadsheet, but you cannot edit it. The purpose of Quick View is to let users see what's in a file without having to open it in its native

app. Viewers are also useful for looking at binary information like EXEs and DLLs. You could use a viewer in a Web app where you want to distribute the viewing program freely, but you want to make users pay for the one that edits. Windows 95 comes with built-in viewers for file types like EXE, DLL, TXT, BMP, DOC (Word), and XLS (Microsoft Excel) files. There seems to be a bias toward file types from Microsoft products. Why that is, I couldn't possibly say.

Figure 3: Quick View for a spreadsheet

Windows lets you write your own viewers for your own file types, but it can be a bit challenging. There's COM to contend with, where the behavior of an object is never fully understood from the specification of its interfaces, but must be gleaned from trial-and-error or from other programs. Also, the built-in viewers are buggy, and the documentation is full of unhelpful tautologies like "IFileViewer::GetPinnedWindow retrieves the handle of the current pinned window." Yes, but what's a pinned window? In short, while writing a file viewer isn't as hard as, say, building a Mars lander, it's not trivial either. That's one of the reasons for building a framework: to write only one viewer once and for all time.

IFileViewer

To the user, Quick View looks like a single app. Behind the scenes, QUIKVIEW.EXE loads and unloads different viewers to view different kinds of files (See **Figure 4**).

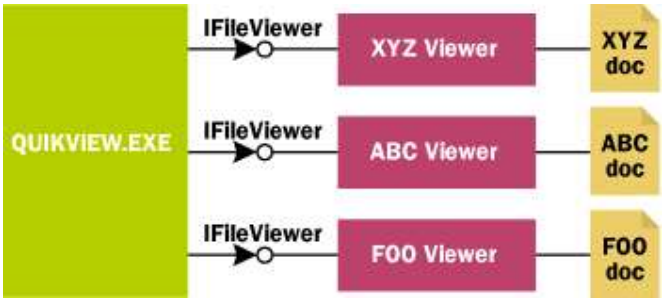

Figure 4: One App, Many Viewers

In technical terms, a viewer is a COM object that implements IFileViewer and IPersistFile. The best way to understand how it works is to take a journey through the code, starting when the user selects Quick View from the menu. I sprinkled my DIB viewer liberally with TRACE statements to help you see what's happening. **Figure 5** shows the commented output of the sequence I'm about to describe.

When the user picks Quick View from the context menu, Windows runs QUIKVIEW.EXE (in \Windows\System\Viewers), passing the file name on the command line. In this case it's guitar.dib. Quick View

looks up the DIB extension in the system registry and finds the GUID for my DIB viewer.

```
\HKEY_CLASSES_ROOT\QuickView\
  .DIB\{828C5D60-1B7E-11cf-82ED-444553540000} =
    "MSJ Bitmap viewer"
```

Quick View then calls CoCreateInstance (or CoGetClassObject) with the GUID. COM determines (again from the registry) that the viewer is implemented in C:\foo\mumble\ bletch\dibview.dll. It loads my DLL, calls DllGetClassObject to get an IClassFactory, then calls IClassFactory::CreateInstance to create the viewer object. Whew. So far, this is just standard COM stuff.

Assuming Quick View was able to create my viewer successfully, the first thing it does is call my viewer's QueryInterface to get an IPersistFile pointer. Remember, file viewers must implement IFileViewer and IPersistFile. Quick View then calls IPersistFile::Load, which I implemented to open and read the DIB into memory. If the Load goes OK, Quick View calls my viewer's QueryInterface again to get an IFileViewer pointer. IFileViewer is my kind of interface—it has just three functions:

```
interface IFileViewer {
  // (IUnknown assumed)
  HRESULT ShowInitialize(LPFILEVIEWERSITE lpfsi);
  HRESULT Show(LPFVSHOWINFO pvsi);
  HRESULT PrintTo(LPSTR pszDriver, BOOL fSuppressUI);
};
```

After Load, Quick View next calls my IFileViewer::ShowInitialize. Then, if that succeeds, it calls IFileViewer::Show. The reason IFileViewer uses two functions for showing is so you can fail without painting the screen. Your implementation of ShowInitialize should create an invisible main window, allocate storage, and in general do everything necessary to run—except actually display the window. If anything fails, you can return an error code and Quick View will give the user a friendly message like, "Sorry Charlie, you're out of luck," and quit without calling your Show function. If, on the other hand, everything is hunky-dory, you should return NOERROR from ShowInitialize, whereupon Quick View calls IFileViewer::Show.

When Quick View calls Show, the, er, show is yours. Typically, your Show function will call ::ShowWindow or ::UpdateWindow to show your main window, then enter a Get/DispatchMessage loop. Your Show function runs continuously, processing messages until the user either exits or views another file, whereupon the Show is over—you should exit your message loop and return. To summarize, Quick View calls the following functions in the following order.

```
IPersistFile::Load
IFileViewer::ShowInitialize
IFileViewer::Show
```

Viewing Another File

So far everything seems simple enough, but as usual things are more complicated than they seem. The user interface guidelines for file viewers (*User Interface Services: Shell: File Viewers* in the Platform SDK) dictates that file viewers should let users open another file using either File Open or drag-and-drop. Let's say your DIB viewer is already running and the user drops another file on it. If the file is another DIB file, no problem—just open it. But what if it's some other kind of file, like a spreadsheet? DIBVIEW displays bitmaps—it

has no idea how to read spreadsheets. What do you do?

This is where things get tricky. When Quick View calls your IFileViewer::Show function, it passes a structure:

```
struct FVSHOWINFO {
  DWORD cbSize;
  HWND hwndOwner;
  int iShow;
  DWORD dwFlags;
  RECT rect;
  LPUNKNOWN punkRel;
  OLECHAR strNewFile[MAX_PATH];
};
```

FVSHOWINFO is an in/out parameter; that is, Quick View uses it to pass information in to your viewer, and you also use it to pass information back out to Quick View. In the situation where the user wants to open a foreign file, you must perform the following actions:

- Copy the name of the file to strNewFile and set FVSIF_NEWFILE in dwFlags.
- Copy the coordinates of your window rectangle to FVSHOWINFO::rect and set FVSIF_RECT in dwFlags.
- Do a QueryInterface(IUnknown) on your viewer object and store the result in punkRel.
- Quit your message loop (but don't destroy or hide your main window) and return from IFileViewer::Show.

The idea here is to keep your window displayed until Quick View successfully loads and shows the new (spreadsheet) viewer. When you return from Show—without destroying your main window—and Quick View sees FVSIF_ NEWFILE, it fires up the viewer for strNewFile and calls its Load, ShowInitialize, and Show functions, passing strNewFile as the filename and the same FVSHOWINFO information you returned. The new viewer Shows itself, careful to position its window using the coordinates you passed in FVSHOWINFO::rect.

At this point, both viewer windows are displayed, but the spreadsheet viewer is directly on top of yours, with the same size and position it looks to the user as if there's just one program that knows how to open both DIBs and spreadsheets. Pretty cool. Assuming everything went OK, the new viewer is responsible for Releasing the punkRel it receives. Unless something is rotten in the universe, this will be the final Release on your viewer object. Your ref count drops to zero and your viewer destroys itself and its main window. Bye-bye, viewer. Hello, viewer.

The sequence of operations is tricky; the new viewer must Release the old viewer after it shows itself, but before it enters its message loop. And of course, everything I just said about what the new viewer must do applies to your viewer too, because it could be the new viewer if the user started out viewing a spreadsheet, then opened a DIB. In other words, if your Show function sees FVSIF_RECT flags set in FVSHOWINFO::dwFlags, it should show its main window at the position specified in FVSHOWINFO::rect. If Quick View passes you an IUnknown pointer in FVSHOWINFO::punkRel, you should Release it after showing your main window.

What happens if something goes wrong with the second viewer? What if its Load or ShowInitialize fails and returns an error? Or what if it's not a spreadsheet but an XYZ file and there's no viewer registered for XYZ files? No problem. Quick View displays a friendly

message like "Sorry Jane, today is not your day," then Releases the new viewer (if there is one) without calling its Show function. Quick View now calls your Show function again, this time with FVSIF_NEWFAILED set. Since you never destroyed your main window, your Show function doesn't have to do anything except re-enter its message loop. Your viewer keeps running as if nothing happened. By doing this little open-a-new-file dance, Quick View communicates with multiple independent viewers to create the illusion of a single running app.

Pinhead Interface: IFileViewerSite

There's one more Quick View feature I have to describe before moving on to DIBVIEW. File viewers are supposed to have a command and toolbar button that lets users pin the viewer window. For some reason, the people who wrote the spec (see *User Interface Services: Shell: File Viewers* in the SDK docs) call this command View | Replace Window and the button looks like **Figure 6**. It's beyond me why anyone chose this command name and icon instead of View|Pin Window or View|Keep Visible and the near universal pushpin icon. I've long suspected some of the folk in Redmond commute from another galaxy, and I'm thinking of calling in the X-Files. In any case, I will say "pinning" the window, not "replacing" it, since that's what the interface functions are named and that's what's actually going on.

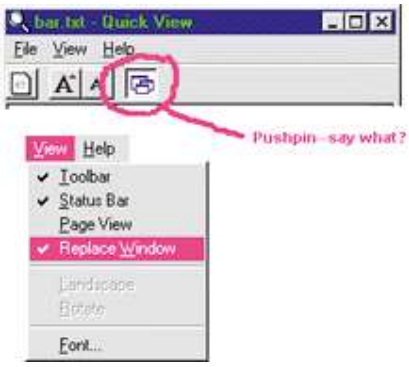

Figure 6: View|Replace Window

Whatever you call it, when the user pushes the pin button or invokes the menu command, Quick View views subsequent files in the same window. Normally, if the user invokes Quick View from a context menu, Quick View launches another viewer instance. But if a window is pinned, Quick View opens the file in that window. Quick View accomplishes this by sending your pinned viewer a WM_DROPFILES message. To your viewer, it looks exactly as if the user has dropped a new file on its main window. There's no difference; your code goes through the same logic. In other words, once you implement WM_DROPFILES, you don't have to do anything else to make pinned windows work.

At least, not as far as opening the file goes. You still have to implement the Pin|View Replace command and keep track of which window is pinned. To do this, you need IFileViewerSite. In COM, whenever you have an IFoo interface, chances are good there's an IFooSite interface to go with it. The site interface is just a callback—a way for a COM object to talk back to the COM object controlling it. In this case, IFileViewerSite lets your viewer call the Quick View program that invoked it. You never have to implement IFileViewerSite; you only use the one passed to your ShowInitialize function. IFileViewerSite is even simpler than IFileViewer:

```
interface IFileViewerSite {
  HRESULT SetPinnedWindow(HWND hwnd);
  HRESULT GetPinnedWindow(HWND *phwnd);
};
```

When the user pushes the pin button or invokes the menu command, you're supposed to call IFileViewerSite::SetPinnedWindow(hYourMainWnd). Your main window is then pinned. When the user unpins the window, you call SetPinnedWindow(NULL). You'd think something so simple would be trivial—but not in Windows! Despite all appearances to the contrary, implementing pinned windows correctly requires plenty of Ibuprofen because IFileViewerSite doesn't quite work the way a normal thinking person would expect.

First, there can be only one pinned window at any time. If the user has six viewers running, only one of them can be pinned. What does this mean for you? It means that when the user presses the pin/unpin button, you can't just call SetPinnedWindow to toggle your pinned state. When you call SetPinnedWindow, you must examine the return code to see if it succeeded because SetPinnedWindow will fail if another window is pinned. Alternatively, you can call GetPinnedWindow to check if another window is pinned before attempting to pin yours. Either way, if another window is pinned, you have two options: do nothing (ignore the user's request), or force-pin your window by calling SetPinnedWindow(NULL) to clear the pinned window, then SetPinnedWindow(hYourMainWnd). Mannerly UI etiquette dictates the latter approach; if the user wants to pin the window, why presume to make him unpin the old one first? But the built-in Windows viewers take the former approach, forcing you to unpin the pinned window, so that's how I implemented mine. (To be fair, using the force-pin method leads to a problem: how does the formerly pinned window get advised when it becomes unpinned? IFileViewer would need another function.)

The second pinning problem has to do with initializing the pinned state. When Quick View starts your viewer for the first time and calls your IFileViewer::Show function, one of the flags it can pass is FVSIF_PINNED. If this flag is set, it means you should start out pinned. But if you simply call IFileViewerSite::SetPinnedWindow(hYourMainWnd), the call will fail because the previous viewer is still technically pinned. Even after you call punkRel->Release() to release the old viewer, Quick View still remembers the now-invalid HWND. The only way to start out pinned is to do the force-pin thing by calling SetPinnedWindow(NULL), then SetPinnedWindow(hYourWnd). I learned this only from reading sample source code.

The last pinning problem is a bug in the Windows 95 viewers. Say Jane User is running your DIB viewer pinned, then drags a DLL file into it. You do the new file dance described earlier, also setting FVSIF_PINNED to tell the new viewer it should start up pinned. Well, the pin button for the new viewer—if it's one of the viewers that comes with Windows 95—comes up in the wrong state; it should be down, but it's up—even though the window really is pinned! I pulled my hair out over this one because I assumed the bug had to be in my code. Silly me. Once I realized it was in the Windows viewer, it was pretty easy to guess what's going on: the built-in viewer must use an internal flag to determine whether the button's checked state is down (TRUE) or up (FALSE), and apparently initializes the flag to FALSE

regardless of FVSIF_PINNED. And yet the window is, in fact, pinned! So the Windows viewer is smart enough to detect FVSIF_PINNED and call SetPinnedWindow, but not smart enough to update the state of its pin button. The moral is: when determining the up/down state of your pin/unpin button, don't rely on flags or BOOLs; always test directly whether the HWND returned by IFileViewerSite::GetPinnedWindow is in fact your viewer's HWND.

```
void CApp::OnUpdatePinWindow(CCmdUI* pCmdUI)
{
  pCmdUI->Enable(m_ipSite != NULL);
  // GetPinned is a helper that calls
  // IFileViewerSite::GetPinnedWindow
  pCmdUI->SetCheck(GetPinned());
}
```

Part of what makes building a viewer so frustrating is the poor documentation. The official reference, *Programming the Windows 95 User Interface*, Chapter 10, "File Viewers," gives only the most superficial overview and none of the stuff you really need to make it work—like any of the pinning semantics just described. For that, you have to learn by trial and error or from reading sample source code. The best is the FILEVIEW program in \MSDEV\ SAMPLES\SDK\WIN95, which I used as my main reference. The MFCVIEW program that comes with Visual C++ 4.1 is a joke—its Show function doesn't even start a message loop, and when you run it, it crashes! Obviously, the code was never finished, but somehow it was released anyway.

My File Viewer Framework

As I predicted, you now know more about IFileViewer than you could ever wish to. Sheesh, how complicated could an interface be that has only three functions? Now you know. Fortunately, it's time for some fun. Let's step back a moment to look at the big picture, as I did when I got the call from Acme.

As is so often the case in COM, much of the file viewer implementation is the same from one viewer to the next. The only thing that's different is how they actually load and display files. If the file is a spreadsheet, you have to parse the format and display the numbers. If the file is a DIB, you have to load the DIB and blt it to the screen. Everything else—all that pinned window voodoo, dragging and dropping, the new file dance—is the same for every viewer. So my first thought was, why not encapsulate it in reusable classes?

Beyond that, I wanted to make maximum use of MFC. A file viewer is not an application; it's a DLL. But it looks and acts like an application. It creates a main window and runs a message loop. So why not use CWinApp, CFrameWnd, and all the rest? That way, I could use message maps to handle commands, ON_COMMAND_UPDATE_UI to update menu items and toolbar buttons, and I'd get automatic status line prompts, tooltips, and all the other neat stuff MFC does for free. This might seem obvious, but as you'll soon find out, it isn't easy. There are places where MFC assumes any program with a message loop is an EXE, not a DLL.

Using CWinApp and CFrameWnd would gain a lot of features to keep Maximum Leader happy, with no effort. But why not take it a step further? Why not use MFC to do the loading and showing too? MFC already has a doc/view model for viewing files. Why not adopt this model to build a generic viewer? IPersistFile::Load would map to CWinApp:: OpenDocumentFile, IFileViewer:: ShowInitialize is like

CWinApp::InitInstance, and IFileViewer::Show is like ShowWindow/UpdateWindow, followed by a call to CWinApp::Run to run the MFC message loop. If I wrote classes that mapped file viewer operations into standard MFC operations, it should be possible to use any MFC doc/view classes. It would become almost trivial to convert any MFC doc/view app into a Quick View viewer—just insert the doc/view classes into your project and compile.

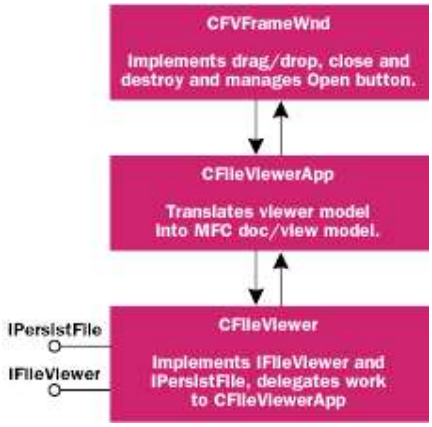

Figure 7 MFC File Viewer Framework

Figure 7 shows an overview of what I came up with. There are three classes. CFileViewer implements IFileViewer and IPersistFile generically, delegating all real work to CFileViewerApp. CFileViewerApp is a CWinApp-derived class with new virtual functions like OnLoad and OnShowInitialize. You can override them, of course, but in most cases you shouldn't have to because the default implementations map IFileViewer operations onto existing MFC operations. For example, OnLoad creates a CDocument object and opens it. The third class, CFVFrameWnd, is a CFrameWnd-derived main window class that handles main window functions common to all viewers, such as drag and drop.

Figure 8 and **Figure 9** summarize what the functions in each class do. Use them as a quick reference guide while I now describe the classes in detail.

CFileViewer

CFileViewer is a CCmdTarget-based COM object that implements IFileViewer and IPersistFile. CFileViewer's basic philosophy is: if I can implement it without knowing anything about the app, do so; otherwise pass to CFileViewerApp. For example, CFileViewer implements AddRef, Release, and QueryInterface for both IFileViewer and IPersistFile in the usual MFC way—by passing them to the outer IUnknown, which is the CCmdTarget from which CFileViewer is derived. CFileViewer implements most IPersistFile functions by returning E_NOTIMPL (not implemented), taking advantage of the fact that Quick View never calls them. CFileViewer also manages a simple state machine to double-check that Quick View calls its functions in the expected order: Load, ShowInitialize, then Show. If not, it returns an error.

Wherever CFileViewer has to do something application-specific, it passes the buck to CFileViewerApp. It passes IPersistFile::Load and IFileViewer::ShowInitialize, Show, and PrintTo to new virtual CFileViewerApp functions OnShowInitialize, OnShow, and OnPrintTo. But first, CFileViewer does some of the boring prep work for these functions. For example, it saves the FVSHOWINFO and IFileViewerSite passed from Quick View, and even AddRefs and Releases the IFileViewerSite so you don't

have to.

Though they're not part of the CFileViewer class, FileView.cpp also implements the standard COM entry points DllGetClassObject and DllCanUnloadNow. These just pass the work to the MFC versions AfxDllGetClassObject and AfxDllCanUnloadNow, which implements them using MFC's knowledge about the class factories defined in DECLARE/IMPLEMENT_OLECREATE. Like any standard MFC COM object, CFileViewer uses these macros to set up its class factory so MFC knows how many viewer objects are alive and thus whether it's safe to unload.

MFC/COM gurus in the audience may be wondering: if CFileViewer is part of the framework, how can different viewers have different GUIDs? Remember, every viewer must have its own GUID—the "unique" in globally unique ID. But how can this be if the GUID is defined in CFileViewer, which is part of the framework? Simple. There's no IMPLEMENT_OLECREATE statement in FileView.cpp. It's in DibView.cpp, which is part of the app:

```
// In DibView.cpp—or your own app module
IMPLEMENT_OLECREATE(CFileViewer,
                    "My Viewer", ... )
```

This macro instantiates a COleObjectFactory called CFileViewer::factory and a GUID called CFileViewer::guid. Just because these variables are CFileViewer members doesn't mean I have to define them in FileView.cpp! By leaving them out, they become undefined symbols, which you must supply by coding the IMPLEMENT_OLECREATE statement somewhere in your viewer app. I suggest doing so in whichever file implements your CFileViewerApp-derived class. This is how each viewer has its own GUID. Pretty sneaky, you have to admit.

CFileViewerApp

So much for CFileViewer. CFileViewerApp is more interesting—it's the real brains of the family. This CWinApp-derived class has virtual functions OnLoad, OnShowInitialize, and OnShow, which CFileViewer calls to implement the application-specific parts of the corresponding IPersistFile/IFileViewer functions. CFileViewerApp has the unenviable task of translating viewer-speak into MFC-speak—that is, implementing viewer operations using doc/view. Unenviable because, as you'll see, MFC was never designed for building file viewers.

The first problem comes when you try to load the document. Normally, MFC loads the doc and creates the frame and view—all in one fell swoop—in CWinApp::OnFileNew or OnFileOpen. If you trace the logic all the way down into the bowels of MFC, you'll discover that control eventually reaches CSingleDocTemplate::OpenDocumentFile (for SDI apps, including viewers), which creates the main frame and the document at the same time. Actually, MFC first creates a new CDocument object, then the frame, then loads the doc. But file viewers don't work that way.

A file viewer is supposed to load its file in IPersistFile::Load, then create the window in IFileViewer::ShowInitialize. So CFileViewerApp must separate document loading from frame/view construction. To do this, CFileViewerApp:: OnLoad calls CDocTemplate::CreateNewDocument and CDocument::OnOpenDocument to create and load the

document without creating a frame window. Later, CFileViewerApp::OnShowInitialize calls CDocTemplate::CreateNewFrame to manually create the frame/view and hook them up to the doc. It sets CWinApp::m_nCmdShow = SW_HIDE to create the frame hidden, as per the rules of IFileViewer. Only when Quick View calls Show, which arrives via CFileViewer to CFileViewerApp::OnShow, does CFileViewerApp show the frame and initialize it with a call to CDocTemplate::InitialUpdateFrame. So CFileViewerApp actually spreads the normally unified doc/view/frame construction across three functions.

The next modification is in OpenDocumentFile. This CWinApp function is the central place where MFC opens a new document. CFileViewerApp's implementation first checks to see if it can open the file. If so, it opens the file the normal way by passing the call to CWinApp. If not—if the user is attempting to view a foreign file type—CFileViewerApp::OpenDocumentFile does the new-file dance: it copies the file name to FVSHOWINFO::strNewFile, sets FVSIF_NEWFILE, and posts a quit message to terminate the message loop. This causes CFileViewerApp::OnShow to return control to CFileViewer's Show function, and from there back to Quick View.

To determine if a new file is openable, I wrote a helper function, CanOpenDocument, that searches the list of doc templates for one whose extension matches the filename. In theory, there's no reason CFileViewerApp can't support multiple document types if you add more doc templates in your InitInstance, but I've never tried it. Windows does it this way; the built-in viewers are actually one viewer object that knows how to open many files. You can tell by looking at the GUIDs for the file extensions listed in the registry under \HKEY_CLASSES_ROOT\QuickView\.xxx. Many of the entries have the same GUID, which points to the same DLL, \windows\system\viewers\sccview.dll.

When Quick View calls Show, CFileViewer passes it to CFileViewerApp::OnShow, which processes all the flags, shows the main window, and calls CWinApp::Run, MFC's message loop function. This is another big problem area. Actually there are two problems. First, CWinThread::Run assumes it's time to quit the app when it receives a WM_ QUIT, so it calls ExitInstance.

```
// in CWinThread::Run()
.
.
.
if (!PumpMessage())
  return ExitInstance();
.
.
.
```

That's fine for a normal EXE app, but not a viewer COM object. The way IFileViewer::Show is supposed to work, opening a new file should terminate the run loop and return control to Quick View without exiting the app. Go directly to home, do not pass ExitInstance. The second problem is that MFC doesn't let you re-enter CWinThread::Run after you've entered it once, which is exactly what happens when the user tries to open a file type Quick View doesn't recognize: it calls your Show function again with FVSIF_ NEWFAILED so you can restart your Run loop.

To fake MFC out, I had to override CWinApp::Run. Essentially, I just copied the whole function (actually in CWinThread, from which CWinApp derives) to CFileViewerApp and changed the parts I didn't like. Hey,

that's what virtual functions are for! I removed the call to ExitInstance, replacing

```
return ExitInstance();
```

with simply

```
return;
```

and I removed the reentry check by setting a variable called m_nDisablePumpCount to zero. See the comments in the source code for details.

One of the things that makes writing a file viewer so confusing at first is that it looks like an app and smells like an app, but it's not really an app. It's really a DLL that runs a message loop. QUIKVIEW.EXE is the app. It sits there, loading and unloading different viewers as needed to view different files, passing information in FVSHOWINFO from one viewer to another, always maintaining the illusion of a single program. So it's understandable that MFC would get a little confused in places.

There are some other fine things CFileViewerApp does to make your life easy. First, CFileViewerApp uses programmer-friendly BOOLs instead of HRESULTs to communicate errors. If a function succeeds it returns TRUE, otherwise it returns FALSE. There's an HRESULT member m_hr you can set to specify the exact COM error, which CFileViewer will pass back to Quick View. Second, CFileViewerApp provides an ON_COMMAND handler OnPinWindow to handle the Pin command, and an ON_UPDATE_COMMAND_UI handler that updates the checked state of its button. These functions encapsulate all the messy pin/unpin logic I described earlier. All you have to do is use them in your app's message map.

Finally, CFileViewerApp also overrides CWinApp::OnFileOpen to call ShellExecute to launch the file in its native app. For example, if the file viewed is a DIB, CFileViewerApp launches Paint. Again, all you have to do is add the function to your message map.

CFVFrameWnd

Last but not least, CFVFrameWnd is the smallest and simplest of the three classes. It implements three message handlers: OnClose, OnDestroy, and OnDropFiles. CFVFrameWnd overrides these MFC message handlers to suit the peculiarities of file viewers. The most significant is OnClose. The normal MFC logic in CFrameWnd::OnClose is over 70 lines. Buried deep within, you'll find the lines

```
// don't exit if there are
// outstanding component objects
if (!AfxOleCanExitApp()) {
   return;
}
```

These lines were intended for EXE apps that support automation and embedding through command-line switches passed to WinMain. If the user exits such an app while there are objects alive, MFC figures it better not post a quit message, or the app will terminate, leaving the objects stranded. So it returns out of CFrameWnd::OnClose having only hidden the main window. Later, when the last object is Released, MFC quits. This fails miserably for viewers; when the user Exits your viewer, you really do want to terminate the

run loop so you can return control from IFileViewer::Show to Quick View. There's no danger of the code going away because it's in a DLL, not an EXE. DllCanUnloadNow makes sure the DLL will stay loaded as long as it has any objects alive. So when the users clicks Exit, you really do want to post a quit message. If you let CFrameWnd handle the WM_CLOSE, your viewer won't ever leave its message loop! The window disappears—because MFC hides it—but it's still there, running invisibly, and the message loop keeps on going and going and going. The solution is to override OnClose.

```
void CFVFrameWnd::OnClose()
{
  // will destroy main main (this) win too
  m_pApp->CloseAllDocuments(FALSE);
  AfxPostQuitMessage(0); // end message loop
}
```

Aside from working, this code is a lot simpler than the 70 or more lines in the MFC version.

The other MFC overrides in CFVFrameWnd are more straightforward. CFVFrameWnd::OnDestroy unpins the main window if it's pinned, before calling CFrameWnd::OnDestroy to destroy the window as normal. CFrameWnd::OnDropFiles ignores multiple files beyond the first one dropped since viewers are SDI apps and can't open multiple files at the same time.

Finally, CFVFrameWnd manages the toolbar Open button. The first button in the Quick View toolbar is supposed to open the file for editing. If you're viewing a DIB, pressing the button launches Paint. If you're viewing a spreadsheet, the viewer would launch the spreadsheet program. To be really cool, the bitmap for this icon should be the shell icon used for files of that type. CFVFrameWnd does it with a new virtual function, OnSetOpenButtonIcon, which calls SHGetFileInfo to get the icon, then draws it into the toolbar. For more details, consult the source code. You don't have to do anything to get this feature. CFVFrameWnd assumes the ID of the toolbar is AFX_IDW_TOOLBAR (the MFC default). If you use a different ID, just change CFVFrameWnd::m_uIDToolBar in your CMainFrame constructor, or set it to zero to turn off the feature.

When I first implemented the Open button feature, I discovered an annoying bug. No matter what I did, there was a row of pixels missing at the bottom of the icon! I assumed it was because MFC makes the toolbar bitmap 15 pixels high by default, whereas icons are 16 pixels high. No problem, just change it. But then the buttons grew taller by one pixel. A minor thing—what's a pixel among friends, and who would even notice? But when I took screen captures of my viewer and the Windows viewer and viewed the toolbars magnified with Paint, I could plainly see that the Windows viewer was somehow able to get all 16 pixels of bitmap without making the buttons taller! Needless to say, I was mystified. And darned if I wasn't going to figure out how they did it.

After several hours of spelunking, I finally discovered that the magic solution is to call

```
m_wndToolBar.SetSizes(CSize(16+7,16+6), CSize(16,16));
```

from your CMainFrame's OnCreate handler. I wasted so many brain cells on this one I don't even want to discuss it—just read the documentation for CToolBar::SetSize, then look at how MFC calls it in CToolBar::LoadToolBar and you'll understand the bug in MFC. In my humble opinion, all apps should have 16-pixel high toolbar

bitmaps so the buttons can display a full 16X16 icon. Now you know how.

Building Your Own Viewer

That was a lot of code to grok in one sitting, but the results are well worth it. Now writing a viewer is practically trivial. **Figure 10** shows DIBVIEW converted to a file viewer using my framework. Not shown are the doc/view classes and all the other goodies from the previous episode. That's because they haven't changed! I literally did not change a single line—they're the same files with the same date/time stamps as the standalone DIBVIEW program. And as you can see from **Figure 10**, the modifications to CApp and CMainFrame are tiny. Most important, since it's the same code, the Quick View version of DIBVIEW has all the same neat features—like zooming in and out, selecting fonts and displaying the information in the BITMAPINFOHEADER (see **Figure 11**).

Figure 11 DIBVIEW

Here's how to create your own viewer starting from DIBVIEW:

1. Modify your InitInstance function to use your own doc/view classes instead of CDIBDoc and CDIBView. Replace doc.cpp and view.cpp with your own files and, of course, get rid of the other files DIBVIEW uses like MsgHook, PalHook, FontUI, and so on. To reduce code size, you might want to delete all your doc/view code that deals with editing.
2. Modify the IMPLEMENT_OLECREATE statement in DibView.cpp to use your own GUID. The GUIDGEN program that comes with Visual C++ can generate an IMPLEMENT_OLECREATE statement with your very own GUID.
3. Delete the CPalMsgHandler::Install call from CMainFrame::OnCreate, unless you use palettes.
4. Edit your menus and toolbars to add pin/unpin and any other viewer commands. If your viewer displays text, you should really implement the font bigger/font smaller buttons—my CFontUI class from the Goodies article makes it easy.
5. Last, but not least, don't forget to modify the REG file in **Figure 12**. Just change "\..\..\dibview.dll" to the path name of your own DLL, and change my GUID everywhere to yours. By convention, viewers go in \WINDOWS\SYSTEM\ VIEWERS, but you can install your DLL anywhere as long as the registry points to the right path name.

I'm sure I forgot some minor details, but that's pretty much it. Just to prove to myself that it all works, I built a Scribble file viewer by using the doc/view classes from the MFC Scribble program. I added ScribDoc.* and ScribVw.* to my project, removed unnecessary #includes of Scribble.h, edited the menus, and

compiled. I had a bona fide Quick View Scribble viewer in about five minutes! **Figure 13** shows it running. Gee, do you think the Redmondtonians will add it to the tutorial?

Figure 13 Scribble Viewer

Debugging Tips

Before parting, I'd like to share some important debugging tips I discovered along the way. First, **Figure 14** shows the proper way to set up your Debug settings in Visual C++ to launch your viewer with F5. Also, use TRACE everywhere! It's the best way I know to learn how Quick View calls your viewer. I used my TRACEFN macro from Part I to do call stack indenting. If you want to use it, just #include Debug.h and add Debug.cpp to your project. These files are not shown in the listing, but are included in the downloadable source code.

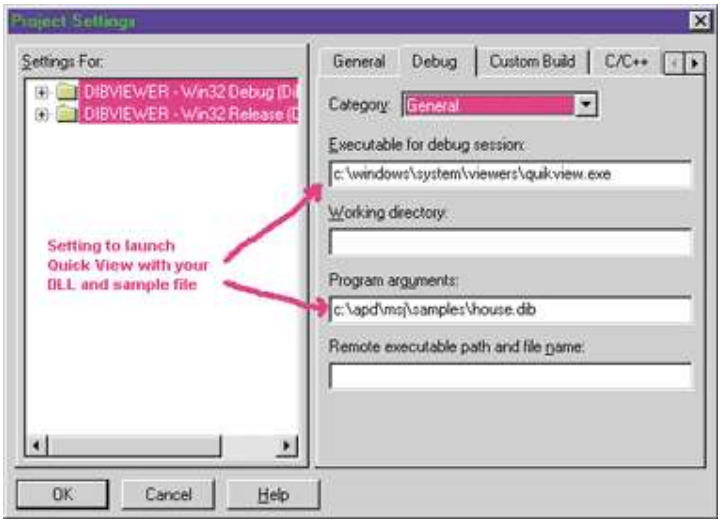

Figure 14 Setting up your Debug sessions

I added another overload of DbgName (described in Part I) to display GUIDs as human-readable text like "IPersistFile" instead of 128-bit hex gobbledygook. To use it, you need the aforementioned Debug module, and you have to set a variable _pDbgInterfaceNames to a table of interfaces you want to print. Check out DibView.cpp in **Figure 10** to see how it works.

Back up your registry entries before loading your REG file! This is particularly important if you're writing a replacement viewer for a file type that already has a Windows 95 viewer, like TXT or BMP. The easiest thing to

do is select \HKEY_CLASSES_ROOT\QuickView in the Registry Editor and export it to a file like restore.reg. Then if you mess up, you can run restore.reg. As an added precaution, during development I used the file type XXX. Once everything worked, I changed XXX to the real BMP extension.

If you build your own viewer, you will run into the following mysterious problem: you compile your code, run it, exit, edit, and compile again—only to have the linker complain it can't open the DLL for output. If you try again a little while later, it works OK. What's going on? It's the delayed-Release feature in Quick View. Even though you exited your viewer, Quick View waits a while before Releasing it in the hopes you might view another file. Your DLL is still loaded in memory, so the file is locked and the linker can't overwrite it. There are two solutions: view another file of a different type (Quick View will release your viewer immediately), or press Control-Alt-Del, select Quick View from the task list, and select End Task. Go ahead, it's harmless—and it feels good.

Conclusion

Writing a Quick View viewer can cause a lot of consternation, but CFileViewer, CFileViewerApp, and CFVFrameWnd make it easy, especially if you already have MFC doc/view classes. The miniframework doesn't do everything. For example, it doesn't implement PrintTo or the Page View option some viewers support to display a whole page in the window, sort of like print preview. But, by using CWinApp, CFrameWnd, and doc/view, my framework leverages MFC's benefits. MFC has its problems, but it's a testament to the sound design of doc/view that something like the viewer framework could ever work at all.

There's only one minor fly in the ointment. Because the viewer uses MFC, it's a bit on the hefty side, which is contrary to the spirit of viewers. Viewers should be as fast and lightweight as possible—that's the "Quick" in Quick View. The release build of DibView seems fast enough to me—about five seconds for the first open, then one second after that. But then, all I have is a lowly 486/66 brain-transplanted to a P/133. Nor do I have Windows NT installed, so I can't say whether my framework runs on it. What can I say? I'm a retro kind of guy.

From the June 1997 issue of Microsoft Systems Journal. Get it at your local newsstand, or better yet, subscribe.