

10. Displays

10.1. Prerequisites and Objectives

Before starting this chapter you should have:

1. A knowledge of programming in C++.
2. A knowledge of component interface design.

After completing this chapter you will have:

1. A knowledge of how to create an object viewer from an existing graphical editor.
2. Familiarity with the principles of object viewers and file viewers, and a knowledge of the differences between them.
4. A knowledge of the Viewer-Component design methodology..

10.2. Introduction

A Display component permits the viewing of an entity without permitting the entity to be edited or changed. There are three types of viewers, File Viewers, Object Viewers, and Visualization components. Examples of File Viewers include the Adobe Acrobat PDF file viewer, and the Microsoft PowerPoint viewer. These components allow PDF files and PowerPoint files to be viewed in a web page, but no editing facilities are provided.

Object viewers are similar to file viewers in that they permit the internal state of an object to be viewed without providing the ability to manipulate the object. Unlike file viewers, object viewers cannot be used in isolation, because they depend on other

components for object creation. An object viewer can be combined with a serialization component to create a file viewer program.

Visualization components are fundamentally different from File and Object viewers. Both File and Object viewers are designed in much the same way as Editor components. In fact, many File and Object viewers are based on existing editors, with the editing facilities omitted or disabled. Visualization components, on the other hand, are based on Model components. In most cases File and Object viewers provide only a static view of their internal objects, while Visualization components typically provide a more dynamic display.

In the following sections we provide examples of all three types of Display components. The SGE Object Viewer and the SGE Debug Viewers are both object viewers. The Text File Viewer is a file viewer, and the Quicksort Visualizer is a Visualization Component. (Visualization is a major research area in its own right, and beyond the scope of this book.)

10.3. The Methodology

Creating a viewer based on an existing Editor component is quite simple. One simply copies the Editor component, and strips out all features that permit the user to change the underlying file or object. Saving files, if available, must also be disabled. We will demonstrate this procedure by creating an object viewer based on the simple graphics editor.

The design methodology for Displays is different depending on what kind of display is being created. If an object viewer is being designed as a complement for a graphical

editor, one can simply make a copy of the editor and disable the editing features. This is essentially what was done for the SGE object viewer.

When we don't have an editor to start with, the procedure is a bit more systematic. The methodology for object and file viewers is somewhat different than that for visualizers, so we will cover these topics separately. The first step in creating an object or file viewer is to specify the visible elements of the file or object. In many cases it is undesirable or impossible to display every detail of a file or an object. In such cases, one must determine what portions of the data are most important, and must specify those portions to be the visible portions of the file or object.

The next step is to define a display format. In most cases, the display format will be dictated somewhat by the structure of the object or file. Text files should be displayed as text, graphical objects should be displayed graphically. It is only in rare cases that we would deviate from the "natural" method of viewing a file. (As an example of such a case, we present the SGE Debug Viewer, which presents graphical objects in text form for debugging.)

The final design step is to define the transformations that will convert data from its internal form into visible objects on the screen. In many cases, this is the most demanding part of the design. The first two steps are relatively simple, because it will generally be obvious which portions of the object or file should be displayed, and the display format will generally also be obvious. The real work of creating the component is in designing and implementing the data transformations.

Visualizers differ from other viewers in two important ways. The first is in the source of the input data. Unlike object and file viewers, which accept data from the outside

world, the data used by many visualizers is entirely self-contained. For example, a visualizer that shows the opening of a flower bud may display successive frames of data from an internal cache. It is also possible to go to the other extreme with data that streams in continuously over a long period of time. For example, a visualizer might be used to display a graph of temperature data for the latest 24 hour period. The data for such a component will probably be presented as periodic observations of the current temperature.

The second step in designing a visualizer is to determine the visual format of the data. Consider the example of the temperature graph. One could present a jagged line representing the temperature fluctuations, or one could represent different temperatures as different colors and present the observations as a multi-colored strip.

The transformation routines used to convert data from internal form to a visible display are the final and most complex step. Most effective visualizers use some form of animation, which must be managed very carefully. Visualization is a science in its own right and is, for the most part, beyond the scope of this book. We will present a simple example to illustrate the principles of Visualizer design.

10.4. The Simple Graphical Editor Object Viewer

Our first example is an object viewer for the **CGraphicList** object. This component has one property, *ModelHandle*, and one method *Redraw*. Figure 10.1 gives the property definition for the *ModelHandle* property while Figure 10.2 gives the method description for the *Redraw* method. The implementations of these are straightforward and are omitted.

The component consists of little more than a drawing routine. Since virtually all of the drawing functionality is embedded in the **CGraphicList** object, this routine is almost trivial. The implementation of the *OnDraw* function is given in Figure 10.3. It is virtually identical to that of the Simple Graphical Editor.

The *ModelHandle* method is used to pass addresses of **CGraphicList** objects to the viewer. The value of this property is retained by the component. If other components change the model, the *Redraw* method must be called to redraw the component window. Alternatively a new value can be assigned to the *ModelHandle* property. This does not cause the current model to be deleted.

Property Design Table		
Name	Type	Function
ModelHandle	Long Integer Maximum=N/A Minimum=N/A Default=None	Provides the address of the modeling object cast to a long integer. NULL values are ignored. Assigning a value to this property causes a redraw of the control window.

Figure 10.1. ModelHandle for the CGraphicList Viewer.

Method Description			
Name	Redraw		
Return Value	Void		
Description	Redraws the control window. If <i>ModelHandle</i> is null, the window is filled with a white rectangle.		
Arguments	Name	Type	Description
Void			

Figure 10.2. Method Description for the CGraphicList Viewer.

```
void CSGEViewCtrl::OnDraw(
    CDC* pdc, const CRect& rcBounds, const CRect& rcInvalid)
{
    if (Model != NULL)
    {
        // we don't have a handle-radius, so make one up
        Model->Draw(pdc,rcBounds,rcInvalid,4);
    }
    else
    {
        // blank window if no model
        pdc->FillRect(rcBounds,
            CBrush::FromHandle((HBRUSH)GetStockObject(WHITE_BRUSH)));
    }
}
```

Figure 10.3. OnDraw Implementation for the CGraphicList Viewer.

10.5. The Simple Graphical Editor Debug Viewer

Our next example is a component that is used only for debugging purposes. It is an object viewer that takes a **CGraphicList** object and displays its contents in human readable form. It is intended to be used for debugging applications that modify the **CGraphicList** object. Adding this component to a debugging program for such a component allows the programmer to monitor the contents of a **CGraphicList** object owned by another component.

This component has much in common with the File Handler component described in Chapter 9. It requires each object in the **CGraphicList** model to provide its own debugging information. The **CGraphicList** object will use this facility to accumulate information about its subobjects. The **CGraphicList** object will accumulate this information into a single string, add information about itself, and return the entire string as the debugging information for the model. To implement this facility, each class in the **CGraphicList** model will have a function named *GetDebugInfo*. This function will be used to provide a human-readable dump of the corresponding objects' contents. There are

many classes in the **CGraphicList** model, and the *GetDebugInfo* functions are quite similar to one another. For that reason we will provide the implementations of only a few of these functions.

Figure 10.4 gives the implementation of the **CGraphicList** *GetDebugInfo* function.

This function has the responsibility for collecting information about all sub-objects in the model.

```
CString CGraphicList::GetDebugInfo()
{
    // Dump the entire model for debugging purposes
    CString rv;
    CString rv2;
    // Dump the main modeling object
    rv.Format("%8.8x GRAPHICLIST Dirty=%d,Count=%d,Head=%8.8x,\
                Tail=%8.8x,SelCount=%d,Selection=%8.8x\r\n",
                this,Dirty,Count,Head,Tail,SelCount,Selection);
    // Dump all graphical objects
    rv += "\r\n-----Graphical Objects-----\r\n";
    CGraphicObject * Temp;
    for (Temp = Head ; Temp != NULL ; Temp=Temp->Next)
    {
        rv2 = Temp->GetDebugInfo();
        rv += rv2;
    }
    // Dump the selection list
    rv += "\r\n-----Selection List-----\r\n";
    CSelectItem * STemp;
    for (STemp = Selection ; STemp != NULL ; STemp = STemp->Next)
    {
        rv2 = STemp->GetDebugInfo();
        rv += rv2;
    }
    // Dump the UNDO/REDO lists
    rv += "\r\n-----Undo/Redo Lists-----\r\n";
    rv2 = UndoList.GetDebugInfo();
    rv += rv2;
    rv += "\r\n";
    rv2 = RedoList.GetDebugInfo();
    rv += rv2;
    return rv;
}
```

Figure 10.4. The CGraphicList GetModelInfo Function.

The *GetModelInfo* function of the **CGraphicList** model collects information about the graphical objects, the selection, and the Undo/Redo lists. The debug information for each object shows the address of the object in hex to allow visual coordination between the objects and the dump of other objects that contain pointers. Figure 10.5 shows the debug information for the **CSelectItem** class. The dump of the *Item* address can be coordinated with the addresses of the graphical objects shown in the dump.

```
CString CSelectItem::GetDebugInfo()  
{  
    CString rv;  
    rv.Format("%8.8x Select Item: Item=%8.8x, Next=%8.8x\r\n",  
             this, Item, Next);  
    return rv;  
}
```

Figure 10.5. The CSelectItem GetDebugInfo Function.

Figure 10.6 shows the implementation of the **CUndoList** *GetDebugInfo* function. This function calls the base class function for most of its information. This function is provided to permit **CUndoList** objects to be distinguished from **CRedoList** objects. The base class function is shown in Figure 10.7. This function also dumps the contents of the Undo Command chain, by calling the *GetDebugInfo* function of each object in the chain. This object dumps the contents of its Undo object chain by calling the *GetDebugInfo* function of every object in its Undo object chain. Figure 10.8 gives an example of a *GetDebugInfo* function for an Undo object.

```
CString CUndoList::GetDebugInfo()
{
    // Dump object for debugging
    CString rv;
    CString rv2;
    rv.Format("%8.8x UNDOLIST ",this); // \r\n intentionally omitted.
    // Get the rest of the information from the base class
    rv2 = CUndoListBase::GetDebugInfo();
    return rv+rv2;
}
```

Figure 10.6. The CundoList GetDebugInfo Function.

```
CString CUndoListBase::GetDebugInfo()
{
    // dump contents for debugging
    CString rv;
    CString rv2;
    // Meant to be concatenated to the end of an existing line
    rv.Format("Count=%d, Limit=%d, Head=%8.8x, Tail=%8.8x\r\n",
             ,Count,Limit,Head,Tail);
    // dump contents of command list
    for (CUndoCommand * Temp = Head ; Temp != NULL ; Temp=Temp->Next)
    {
        rv2 = Temp->GetDebugInfo();
        rv += rv2;
    }
    return rv;
}
```

Figure 10.7. The CundoListBase GetDebugInfo Function.

```
CString CUndoAdd::GetDebugInfo()
{
    // Dump information for debugging
    CString rv;
    CString rv2;
    rv.Format("      %8.8x UNDOADD Obj=%8.8x\r\n",this,Obj);
    rv2.Format("          Links: Next=%8.8x\r\n",Next);
    return rv+rv2;
}
```

Figure 10.8. The CundoAdd GetDebugInfo Function.

Like the **CGraphicList** viewer, the Debug Viewer has one method and one property. The names and descriptions of these are identical to those of Figure 10.1 and Figure 10.2, so we will omit their descriptions and implementations. Most of the work done in the

Debug Viewer is encapsulated in the **CGraphicList** *GetDebugInfo* function. For the implementation we will subclass the standard Windows operating system Text Box control. This control gives us the ability to display multi-line text, and to scroll through the text using horizontal and vertical scroll bars.

The first step in creating a subclassed control is done when creating the project.

Figure 10.9 shows the second page of the MFC ActiveX wizard. At the bottom of this page is a combo-box labeled “Which window class, if any, should this control subclass?” We have selected “EDIT” from the list of available controls which will make our control a special case of the *EDIT* control, which is also known as the *Text Box* control.

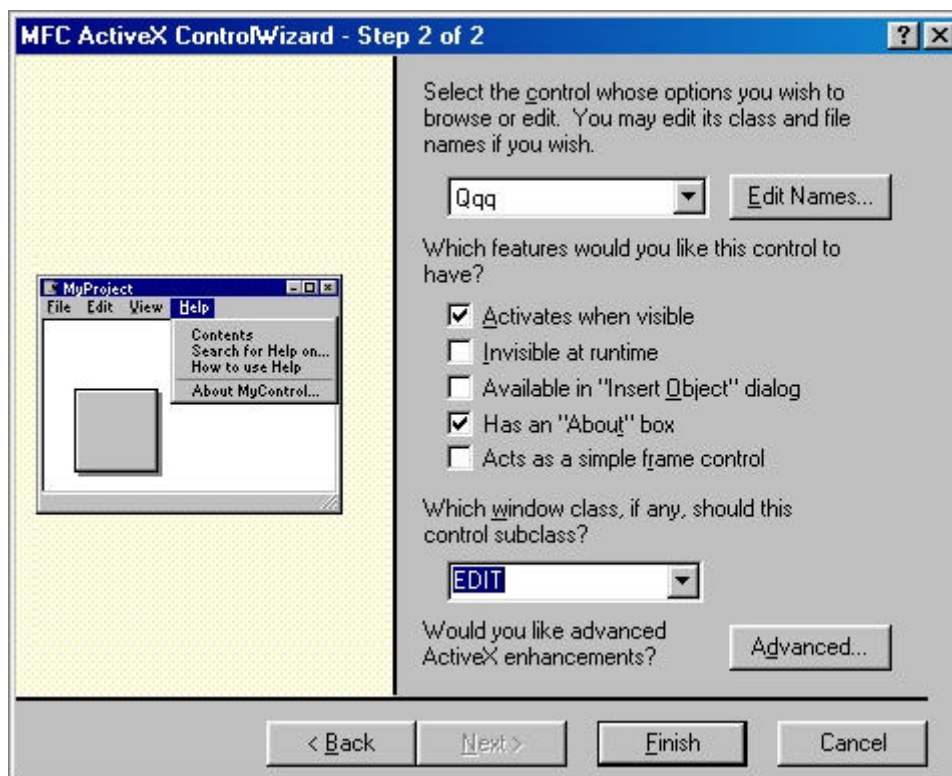


Figure 10.9. Subclassing a Control.

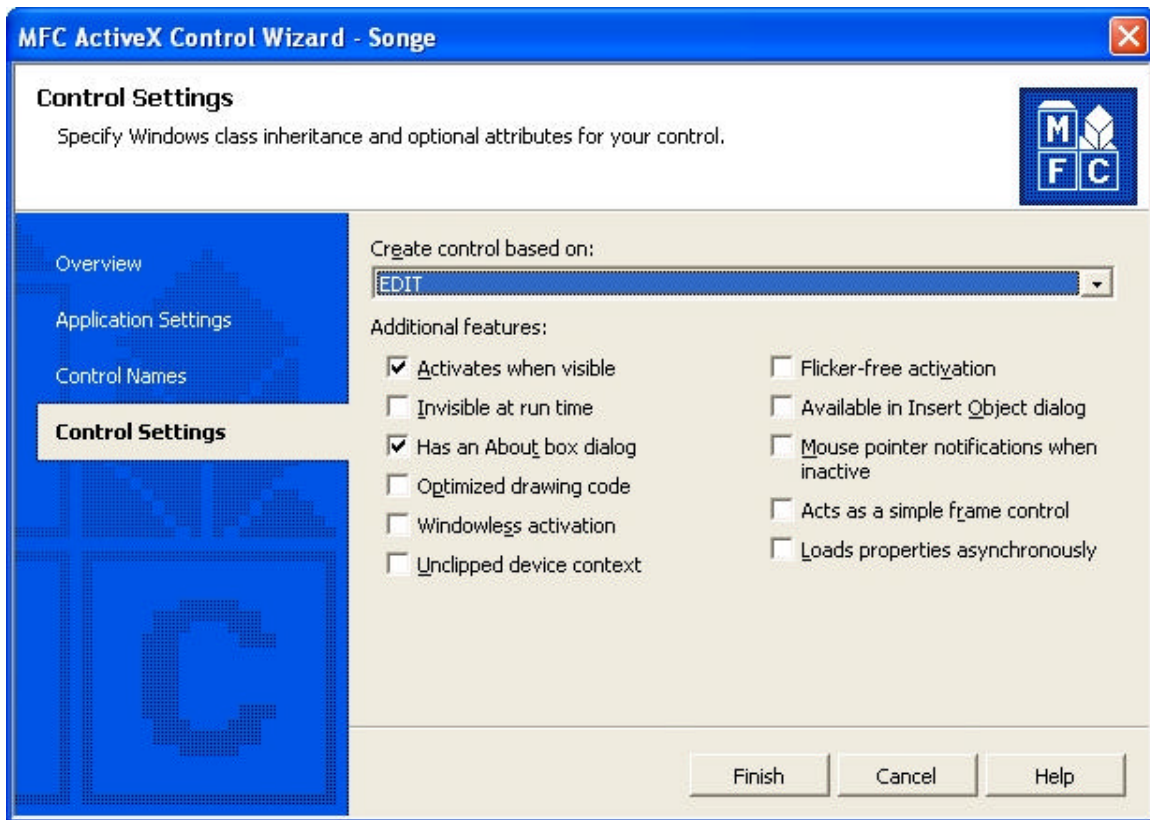


Figure 10.10. Subclassing in VS.NET.

There is one important additional function that is added to your component when subclassing a Windows operating system control. That is the *PreCreateWindow* function. This function can be modified to change the properties of the control. When we subclass a windows control, our component ends up being a special case of that control. When we set standard Windows operating system properties in the *PreCreateWindow* function, we are setting the properties of our own window. We wish our component to be a multi-line Text Box and we wish it to have vertical and horizontal scroll bars. None of these properties are default, so we must set them in the *PreCreateWindow* function. Figure 10.11 shows how this is done.

```
BOOL CSGEDebugCtrl::PreCreateWindow(CREATESTRUCT& cs)
{
    cs.lpszClass = _T("EDIT");
    // required additions to make EDIT control MULTILINE and to add
    // scroll bars. The other two lines are provided by the wizard.
    cs.style |= ES_MULTILINE | WS_HSCROLL | WS_VSCROLL;
    return COleControl::PreCreateWindow(cs);
}
```

Figure 10.11. The PreCreateWindow Function.

Because our control is a subclassed Text Box, the standard Text Box drawing routine will eventually be used to draw the control window. The *OnDraw* routine of our component can add steps before and after the standard Text Box drawing routine is called. To invoke the standard drawing routine the *DoSuperclassPaint* function must be called at some point in the *OnDraw* routine. In our case, we will simply make sure that the Text Box contains the proper text before it is drawn. We will use the *GetDebugInfo* function of the **CGraphicList** class to obtain the Text Box contents, and we will insert this text into the text box using the *SendMessage* Function. If there is no model, we will delete all text from the text box using the same function. See Chapter3 for more discussion of the *SendMessage* function. The design of the *OnDraw* function completes the design of the Debug Viewer.

```

void CSGEDebugCtrl::OnDraw(
    CDC* pdc, const CRect& rcBounds, const CRect& rcInvalid)
{
    // Remember we are a subclassed multi-line text box
    if (Model != NULL)
    {
        // dump the model into Txt, and feed Txt to the TEXT control
        CString Txt = Model->GetDebugInfo();
        SendMessage(WM_SETTEXT, 0, (long)((LPCTSTR)Txt));
    }
    else
    {
        // If no model, blank the text control
        SendMessage(WM_SETTEXT, 0, (long)("");
    }
    // required function call for subclassed controls
    DoSuperclassPaint(pdc, rcBounds);
}

```

Figure 10.12. The Debug Viewer OnDraw Routine.

10.6. The Text File Viewer

Next we will give an example of a File Viewer. To keep things simple we will create a viewer for simple text files. This viewer will allow us to scroll through a text file of virtually any size, regardless of whether it fits completely in memory, but will provide no editing capabilities. To permit very large files to be viewed, we will read in only that portion of the file that is currently visible in the control window. We will use the default system font to display the text, and will not allow the user to change fonts. Each line in the text file will be displayed on a separate line of the display. We will not process special characters like “tab.”

The main problem that we must deal with is locating lines in the text file. Because lines in a text file are of variable length, the position of a particular line can be determined only by reading all characters that precede it. Because the file is not read completely into memory, this would require an enormous amount of I/O when scrolling long distances through the file. To solve this problem, we will create a line indexing

array. When a file is first opened, we will read the entire file, and note the starting position and length of each line. We will accumulate this information into an array that will allow us instant access to any line in the file. (For exceptionally large files, we may want to keep less information than this, but that is a different component.)

The functionality of our line-list array will be encapsulated in a single class, **CLineList**, which is shown in Figure 10.13. Initially, data will be accumulated in a linked list, and then the linked-list will be reorganized into an array.

```
class CLineList
{
public:
    CLineElement * Array;
    long Count;

    BOOL ArrayDone;
    CLineLink * Tail;
    CLineLink * Head;

    CLineList();
    virtual ~CLineList();

    void AddLine(long NewPos, long NewLen);
    void MakeArray(void);
    long GetCount(void);
    long GetLength(long LineNumber);
    long GetPosition(long LineNumber);
};
```

Figure 10.13. The CLineList Class.

In Figure 10.13 the data items *Array* and *Count* are used to maintain the line-list array, while the data items *Head* and *Tail* are used for the initial accumulation of information. The *AddLine* function is used to accumulate data into the linked list, while the *MakeArray* function is used to reorganize the linked list into an array. Access to the array is given by the functions *GetCount*, *GetLength*, and *GetPosition*. The class

CLineLink, shown in Figure 10.14, is used to create the linked list. The definition of this class is straightforward.

```
class CLineLink
{
public:
    CLineLink * Next;

    long Length;
    long Position;

    CLineLink();
    CLineLink(long NewPos, long NewLen);
    virtual ~CLineLink();
};
```

Figure 10.14. The CLineLink Class.

The line-list array is an array of **CLineElement** objects, the class definition of which is given in Figure 10.15. The definition of this class is reasonably straightforward, except for the overloaded assignment operator, which is used to simplify the creation of the array from a linked list of **CLineLink** objects.

```
class CLineElement
{
public:
    long Length;
    long Position;

    CLineElement();
    virtual ~CLineElement();

    CLineElement & operator=(const CLineLink &x);
};
```

Figure 10.15. The CLineElement Class.

Figure 10.16 shows how line information is accumulated into a linked list with the *AddLine* function. The *ArrayDone* variable of the *CLineList* class is used to determine whether an array has been built from the accumulated data. If it has, then accumulating

new data is not allowed. The file must be read through completely when it is first opened.

The *AddLine* function maintains the value of the *Count* data item.

Figure 10.17 shows how data is moved from the linked list to the array, using the *MakeArray* function. Making an array is illegal if there is already an array, or if no data has been accumulated. The linked list is destroyed as data is moved into the array.

```
void CLineList::AddLine(long NewPos, long NewLen)
{
    if (ArrayDone)
    {
        return;
    }
    CLineLink * Temp = new CLineLink(NewPos,NewLen);
    if (Head == NULL)
    {
        Head = Temp;
    }
    else
    {
        Tail->Next = Temp;
    }
    Tail = Temp;
    Count++;
}
```

Figure 10.16. The AddLine Function of the CLineList Class.

```
void CLinkedList::MakeArray()
{
    if (ArrayDone)
    {
        return;
    }
    if (Count <= 0)
    {
        return;
    }
    Array = new CLineElement[Count];
    long i = 0;
    CLineLink * Temp;
    for (Temp = Head, *Temp2=NULL ; Temp != NULL ; Temp=Temp2,i++)
    {
        Temp2 = Temp->Next;
        Array[i] = *Temp;
        delete Temp;
    }
    Head = NULL;
    Tail = NULL;
    ArrayDone = TRUE;
}
```

Figure 10.17. The MakeArray Function.

The *GetPosition* and *GetLength* functions are given in Figure 10.18. These functions will return a -1 if there is no array, or if the array bounds have been violated.

```

long CLineList::GetPosition(long LineNumber)
{
    if (!ArrayDone)
    {
        return -1;
    }
    if (LineNumber < 0)
    {
        return -1;
    }
    if (LineNumber >= Count)
    {
        return -1;
    }
    return Array[LineNumber].Position;
}

long CLineList::GetLength(long LineNumber)
{
    if (!ArrayDone)
    {
        return 0;
    }
    if (LineNumber < 0)
    {
        return 0;
    }
    if (LineNumber >= Count)
    {
        return 0;
    }
    return Array[LineNumber].Length;
}

```

Figure 10.18. GetPosition and GetLength Functions.

To allow the user to specify a file to be viewed, we provide a property called *InputFile*. The property table for this property is given in Figure 10.19. This property is not available at design time. When a new value is assigned to *InputFile*, the existing file, if any, is closed and its line list array is deleted. The *SetInputFile* function is used to create the new line list array. The file is kept open while being viewed to prevent any other program from writing new data to the file and invalidating the line list array. Figure 10.20 shows the implementation of the *SetInputFile* function.

Property Design Table		
Name	Type	Function
InputFile	String Default=Empty	Supplies the name of the text file to be viewed. A full path name is usually supplied. Restrictions: Run Time Only

Figure 10.19. The InputFile Property.

```

void CTextViewerCtrl::SetInputFile(LPCTSTR lpszNewValue)
{
    m_inputFile = lpszNewValue; // Record file name
    if (FileOpen) // Close existing file, if any
    {
        CloseHandle(FileHandle);
        FileOpen = FALSE;
    }
    if (LL != NULL) // delete existing line list, if any
    {
        delete LL;
        LL = NULL;
    }
    LL = new CLineList; // create new line list
    FileHandle = CreateFile(m_inputFile,GENERIC_READ,FILE_SHARE_READ,
        NULL,OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,NULL);
    if (FileHandle == INVALID_HANDLE_VALUE)
    {
        ... // Report open error
    }
    else // file opened correctly
    {
        FileOpen = TRUE;
        char * Buffer = new char[10000]; // create I/O buffer
        long CharCount = 0;
        long Pos = 0;
        unsigned long BytesRead;
        // Read first chunk of file
        BOOL Success = ReadFile(FileHandle,Buffer,10000,&BytesRead,NULL);
        if (!Success)
        {
            ... // report I/O Error
        }
        while (BytesRead > 0) // Process chunks until EOF
        {
            unsigned long i;
            char * tc;
            // examine each character
            for (i=0,tc=Buffer ; i<BytesRead ; i++,tc++)
            {
                CharCount++; // length of current line
                if (*tc == '\n') // If EOL found, record line pos & len
                {
                    LL->AddLine(Pos,CharCount);
                }
            }
        }
    }
}

```

```

        Pos += CharCount;
        CharCount = 0;
    }
}
// get next chunk
Success = ReadFile(FileHandle, Buffer, 10000, &BytesRead, NULL);
if (!Success)
{
    ... // Report I/O Error
}
}
if (CharCount > 0) // process an unterminated line, if any
{
    LL->AddLine(Pos, CharCount);
}
delete [] Buffer; // release I/O buffer
LL->MakeArray();
}
// Set scrolling parameters
VMax = LL->GetCount()-1;
VPos = 0;
HPos = 0;
CWnd::SetScrollRange(SB_VERT, 0, VMax);
CWnd::SetScrollPos(SB_HORZ, HPos);
CWnd::SetScrollPos(SB_VERT, VPos);
CWnd::Invalidate();
}

```

Figure 10.20. The SetInputFile Function.

The *SetInputFile* function first closes any existing file and deletes the line list object for the file. It then opens the new file and creates a new line list object. It reads the file in 10,000 byte chunks, and processes each chunk looking for line-feed characters. Characters are examined one at a time and counted. When a line-feed is found, the position and length of the line is added to the line list object. Once all data has been read, the file is left open to prevent other programs from changing it. If there is an unterminated line at the end of the file, the position and length of this line are recorded at this point. Finally, a new array is created by the line list object, and scrolling parameters are set for both scroll bars.

The viewer component allows the user to scroll data up and down as well as to the right and left. This function is not performed automatically by the Windows operating

system, but must be explicitly programmed into our component. The first thing we must do is add horizontal and vertical scroll bars to our component window. Figure 10.21 shows how this is done. (Use the first tab of the class wizard to add this function to a component.)

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

Figure 10.21. Adding Scroll Bars to a Control Window.

Next, we need to add functions to process the messages created by the scroll bars. The scroll bars do not control the window directly. Their only function is to send messages to the component. The handling of these messages is entirely up to the component. We add two variables, *VPos* and *HPos*, to the support class to keep track of the current scrolling position. When we receive a message from one of the scroll bars, we will change the value of either *VPos* or *HPos* and redraw the window. The drawing routine will use the *VPos* and *HPos* variables to position text within the window, and to determine which lines to read from the file. Messages from the vertical scroll bar are handled by the function *OnVScroll*. Rather than providing code for this function, we will list the scroll-bar messages processed by the routine, and the action taken for each. This information is presented in Figure 10.22. The horizontal scrolling actions are virtually identical to the vertical ones.

Message	Action
SB_BOTTOM	Set <i>VPos</i> to its maximum value.
SB_TOP	Set <i>VPos</i> to zero.
SB_LINEDOWN	If <i>VPos</i> is less than the maximum, increment its value by one.
SB_LINEUP	If <i>VPos</i> is greater than zero, decrement its value by one.
SB_PAGEDOWN	Increment <i>VPos</i> by the number of lines in the window minus one. If <i>VPos</i> is greater than the maximum, set it to the maximum.
SB_PAGEUP	Decrement <i>VPos</i> by the number of lines in the window minus one. If <i>VPos</i> is less than zero, set it to zero.

Figure 10.22. Vertical Scrolling Actions.

The *OnDraw* routine is given in Figure 10.23. This function first determines the pixel height and width of the window by calling the operating system function *GetClientRect*. Next, the *GetTextExtent* function is used on a sample string to determine the height of a line and the average width of a character in the current font. This information is used to compute the height of the window in lines and the width of the window in average-width characters. This information is saved in the support class variables *VPage* and *HPage*, which will be used by the horizontal and vertical scrolling routines. (The first drawing of the window will precede the first use of the scroll-bars.)

Vertical scrolling is accomplished by using the *VPos* variable as the starting index for reading lines from the current file. A sufficient number of lines will be read to fill the display. Each line is displayed as it is read. Horizontal scrolling is done using the standard operating system function *SetWindowOrg*. Each line will be drawn at *x*-position zero. By setting the window origin to a value larger than zero, different parts of the line will be clipped. The horizontal page size used by the scroll bar will be set by the *OnDraw* routine after the length of each line has been determined.

```

void CTextViewerCtrl::OnDraw(
    CDC* pdc, const CRect& rcBounds, const CRect& rcInvalid)
{
    pdc->FillRect(rcBounds,
        CBrush::FromHandle((HBRUSH)GetStockObject(WHITE_BRUSH)));
    if (FileOpen)
    {
        RECT DispRect;
        CWnd::GetClientRect(&DispRect);
        long DispHeight = DispRect.right - DispRect.left;
        long DispWidth = DispRect.bottom - DispRect.top;
        CSize TextSize =
            pdc->GetTextExtent("ABCDEFGHIJKLMNOPQRSTUVWXYZ"
                "abcdefghijklmnopqrstuvwxyz");

        long LineHeight = TextSize.cy;
        long CharWid = TextSize.cx / 52;
        DispLines = DispHeight / LineHeight;
        DispChars = DispWidth / CharWid;
        VPage = DispLines - 1;
        HPage = DispChars - 1;
        pdc->SaveDC();
        long OldHMax = HMax;
        HMax = 0;
        pdc->SetWindowOrg(HPos*DispChars,0);
        for (long i=0,DrawPos=0 ; i<DispLines && i+VPos < LL->GetCount();
            i++,DrawPos+=LineHeight)
        {
            long Pos = LL->GetPosition(i+VPos);
            unsigned long Len = LL->GetLength(i+VPos);
            if ((unsigned long)HMax < Len-1)
            {
                HMax = Len-1;
            }
            if (Pos != -1 && Len != 0)
            {
                unsigned long BytesRead;
                char * Buffer = new char [Len+1];
                SetFilePointer(FileHandle,Pos,NULL,FILE_BEGIN);
                BOOL rv = ReadFile(FileHandle,Buffer,Len,&BytesRead,NULL);
                if (rv && BytesRead == Len)
                {
                    Buffer[Len] = '\\0';
                    pdc->TextOut(0,DrawPos,Buffer);
                }
                delete Buffer;
            }
        }
        if (HMax != OldHMax)
        {
            CWnd::SetScrollRange(SB_HORZ,0,HMax);
        }
        pdc->RestoreDC(0);
    }
}

```

Figure 10.23. The Text Viewer OnDraw Function.

The complete code for the text file visualizer is available on the CD.

10.7. The Quicksort Visualizer

The Quicksort visualizer is based on a generic class that can be used to produce a visual animation of virtually any sorting algorithm. The control displays the list of items to be sorted as an array of vertical bars, with the height of each bar representing the value of the item in the list. Figure 10.24 shows the visualization of an unsorted list. When the list is sorted, the bars will be arranged in a stair-step fashion. During the course of the sort, the bars will be moved, and their color will be changed to illustrate the role that they are playing in the sort algorithm.

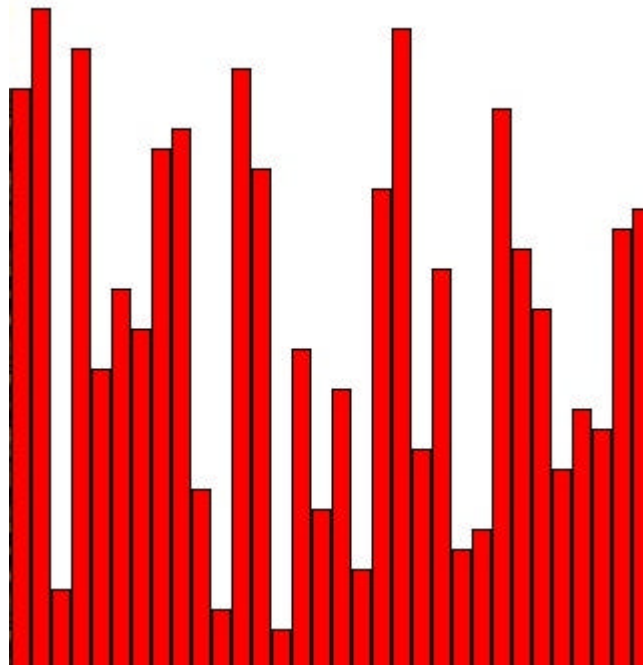


Figure 10.24. An Unsorted List.

Figure 10.25 shows the sort operation in progress. The green bar is already in its correct position. The yellow bar is the pivot point, the magenta bars are those bars that have been determined to be larger than the pivot point, while the blue-green bars are

those that have been determined to be smaller than the pivot point. The blue-green bar to the far left has just been compared to the pivot point, and is about to be swapped with the leftmost magenta bar. The red bars are the bars that have not yet been compared to the pivot point.

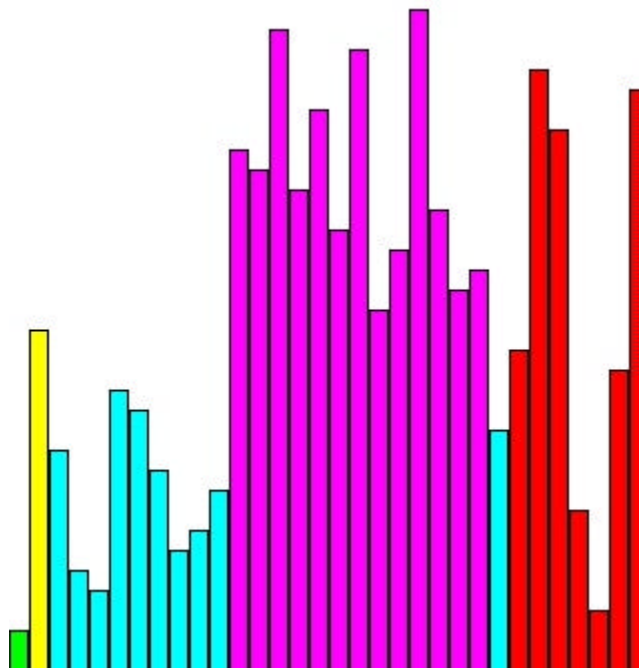


Figure 10.25. QuickSort in Progress.

The QuickSort Visualizer has several methods and properties that control the animation. The user can select the number of bars, and set the speed of the animation. The *Sort* method is used to begin the animation. The user can randomize the list to repeat the animation, or can presort the list to demonstrate the difference in speed between a presorted list and a randomized list.

Internally, the animation is built using a “standard” sort animation object, called **CSortAnimator**. This object has been used to create many different sort animations. This object controls the drawing of the list, and provides functions for changing bar colors and for moving bars around in the list. Figure 10.26 gives the definition of this class. The

general methodology for using this class is to first take a sort algorithm, “right out of the book,” and then modify it slightly to conform to the requirements of the class. In many respects, the objects of this class act like arrays of integers, however there are some differences. Figure 10.27 shows a textbook version of an iterative Quicksort algorithm, with the required changes shown in red.

```

class CSortAnimator
{
public:
    // Bar Dimensions in pixels
    // BarHeight[I] = BarNumber[I]*BarIncrement+MinBarHeight
    long BarIncrement;
    long MinBarHeight;
    long BarWidth;
    // Bar array
    long Size;
    long * Values;
    ColorType * Colors;
    long GetSize(void);
    long SetBarCount(long Bars);
    long SetBarColor(long BarNumber,ColorType Color);
    // Temp Array
    long TempCount;
    long * Temps;
    ColorType * TempColors;
    long GetTempCount(void);
    long SetTempCount(long NewTempCount);
    long SetTempColor(long TempID,ColorType Color);
    // Sorting
    long TempSize(long TempID); // bar size in temp storage
    long BarSize(long Bar); // retrieve bar size
    long operator[](long x); // retrieve bar size
    long MoveTempToBar(long TempID, long Bar); // temp storage mgmt
    long MoveBarToTemp(long Bar,long TempID); // temp storage mgmt
    long SwapBars(long BarA,long BarB);
    // Drawing
    long Draw(CDC *Mydc, const CRect& rcBounds, const CRect& rcInvalid);
    // Initialization
    long Randomize(void);
    long Initialize(void);
    // Color Decoding
    COLORREF GetColor(ColorType Color); // internal utility function
    // creation and Destruction
    CSortAnimator();
    virtual ~CSortAnimator();
};

```

Figure 10.26. The CSortAnimator Class.

```

void CQuickSortCtrl::Sort()
{
    long SFirst[32],SLast[32],SSize=0; // private stack
    long n = L.GetSize();
    SFirst[SSize] = 0; // push entire list on stack
    SLast[SSize] = n-1;
    SSize++;
    while (SSize>0) // while stack not empty
    {
        SSize--; // pop a sub-list off the stack
        long First = SFirst[SSize];
        long Last = SLast[SSize];
        while (First < Last) // split the sublist
        {
            long SP,i; // SP is the split point
            for (i=First+1,SP=First ; i<=Last ; i++)
            {
                // L is the CSortAnimator Object
                if (L[i] < L[First]) // L[First] is the pivot point
                {
                    SP++;
                    L.SwapBars(i,SP);
                }
            }
            L.SwapBars(First,SP);
            // Stack Shortest, process longest immediately
            // Empty lists must be stacked
            if ((Last-SP) > (SP-First))
            {
                SFirst[SSize] = SP+1;
                SLast[SSize] = Last;
                SSize++;
                Last = SP-1;
            }
            else
            {
                SFirst[SSize] = First;
                SLast[SSize] = SP-1;
                SSize++;
                First = SP+1;
            }
        }
    }
}

```

Figure 10.27. Adapted Quicksort.

Although the Sort function of Figure 10.27 provides a complete animation of the Quicksort algorithm, it will happen much too fast for the user to see. Furthermore, the color-changes that indicate the status of each bar have not yet been inserted. The color of

each bar limited to a fixed set of colors given by the **ColorType** enumeration of Figure 10.28. The enumeration is used to simplify the programming of the sort algorithm, and to provide a *None* color to indicate that a bar is currently invisible. This color enumeration is used with the *SetBarColor* function to further enhance the animation.

```
enum ColorType
{
    None = 0,
    Red = 1,
    Green = 2,
    Blue = 3,
    Yellow = 4,
    Cyan = 5,
    Magenta = 6,
    Black = 7,
    White = 8
};
```

Figure 10.28. The ColorType Enumeration.

To slow down the animation so that the user can see it, the component provides a *Sync* function that will redraw the component window, and then wait for a user-specified number of milliseconds. Although this function is not part of the **CSortAnimator** class, it is a standard part of all sort animations created using the class. This is done by embedding the *Sync* function and the **CSortAnimator** class in a project template, and using the template to generate several animations. Because the *Sync* function simply calls a series of operating system functions, we will not present the code for it. For interested readers, the complete code can be found on the accompanying CD.

The completely instrumented version of the Quicksort algorithm is given in Figure 10.29, with the additional code illustrated in red.

```

void CQuickSortCtrl::Sort()
{
    long SFirst[32],SLast[32],SSize=0;
    long n = L.GetSize();
    SFirst[SSize] = 0;
    SLast[SSize] = n-1;
    SSize++;
    for (long x = 0 ; x<n ; x++)
    {
        L.SetBarColor(x,White);
    }
    Sync();
    while (SSize>0)
    {
        SSize--;
        long First = SFirst[SSize];
        long Last = SLast[SSize];
        while (First < Last)
        {
            for (x=First ; x<=Last ; x++)
            {
                L.SetBarColor(x,Red);
            }
            Sync();
            long SP,i;
            L.SetBarColor(First,Yellow);
            Sync();
            for (i=First+1,SP=First ; i<=Last ; i++)
            {
                if (L[i] < L[First])
                {
                    L.SetBarColor(i,Cyan);
                    Sync();
                    SP++;
                    L.SwapBars(i,SP);
                    Sync();
                }
                else
                {
                    L.SetBarColor(i,Magenta);
                    Sync();
                }
            }
            L.SwapBars(First,SP);
            Sync();
            L.SetBarColor(SP,Green);
            Sync();
            if ((Last-SP) > (SP-First))
            {
                for (x=SP+1 ; x<=Last ; x++)
                {
                    L.SetBarColor(x,White);
                }
                Sync();
                SFirst[SSize] = SP+1;
                SLast[SSize] = Last;
                SSize++;
            }
        }
    }
}

```

```

        Last = SP-1;
    }
    else
    {
        for (x=First ; x<=SP-1 ; x++)
        {
            L.SetBarColor(x,White);
        }
        Sync();
        SFirst[SSize] = First;
        SLast[SSize] = SP-1;
        SSize++;
        First = SP+1;
    }
}
if (First == Last)
{
    L.SetBarColor(First,Green);
    Sync();
}
}
Sync();
}
}

```

Figure 10.29. The Fully Instrumented Quicksort Algorithm.

Because the **CSortAnimator** class is used for many different sort animations, it contains features that are not used by the Quicksort animation. There is a facility for demonstrating the use of temporary variables to hold list elements. (This is especially useful for Merge Sort, and Radix Sort.) The bars representing temporary variables are drawn horizontally across the top of the control window. There is also a facility for presorting the list, and another for randomizing a list.

The implementation of most of the **CSortAnimator** functions should be obvious. To illustrate the general methodology, we present the implementation of two of these functions in Figure 10.30.

```
long CSortAnimator::SwapBars(long BarA, long BarB)
{
    if (BarA >= 0 && BarA < Size && BarB >= 0 && BarB < Size)
    {
        long Temp = Values[BarA]; // Swap Bars and colors
        Values[BarA] = Values[BarB];
        Values[BarB] = Temp;
        ColorType CTemp = Colors[BarA];
        Colors[BarA] = Colors[BarB];
        Colors[BarB] = CTemp;
        return 1;
    }
    else
    {
        return 0; // at least one index is illegal
    }
}

long CSortAnimator::operator[](long x)
{
    if (x >= 0 && x < Size) // return bar size if index legal
    {
        return Values[x];
    }
    else
    {
        return 0;
    }
}
```

Figure 10.30. Two CSortAnimator Functions.

For the most part, the properties and methods of the component simply provide access to the functions of the internal **CSortAnimator** object. The exception is the *Interval* property, which is used to set the delay used by the *Sync* function. The property design table is given in Figure 10.31, and the method descriptions are given in Figure 10.32.

Property Design Table		
Name	Type	Function
Interval	Long Integer Maximum=10000 Minimum=0 Default=500	Delay in milliseconds for each discrete operation performed by the animation.
BarIncrement	Long Integer Maximum=None Minimum=None Default=20	The difference in height between two successive bars when the list is sorted. Units are in pixels.
BarWidth	Long Integer Maximum=None Minimum=None Default=20	The width in pixels of each bar.
MinBarHeight	Long Integer Maximum=None Minimum=None Default=20	The height in pixels of the shortest bar.
BarCount	Long Integer Maximum=None Minimum=None Default=32	The number of bars in the list. Setting this property causes the list to be presorted.

Figure 10.31. Property Design Table for the Quicksort Animator.

Method Description			
Name	Initialize		
Return Value	Void		
Description	Presorts the list and redraws the window. Bar color is set to red for each bar. The control window is redrawn.		
Arguments	Name	Type	Description
Void			

Method Description			
Name	Randomize		
Return Value	Void		
Description	Randomizes the list, and sets the bar color of each bar to red. The control window is redrawn.		
Arguments	Name	Type	Description
Void			

Method Description			
Name	Sort		
Return Value	Void		
Description	Starts the sort animation process. This method is not interruptible, and will not return until the animation is complete.		
Arguments	Name	Type	Description
Void			

Figure 10.32. Method Descriptions for the Quicksort Animator.

10.8. A Review of the Methodology

The design methodology for Displays is different depending on what kind of display is being created. If an object viewer is being designed as a complement for a graphical editor, one can simply make a copy of the editor and disable the editing features. This is essentially what was done for the SGE object viewer.

An object viewer that is not associated with an interactive editor must be more carefully designed. Such Displays are quite similar to file viewers, the only difference being in the source of the input data. The first step in the design of either type of viewer is to specify the visible elements of the component. Once the visible elements have been specified, along with any required scaling and scrolling features, one must design the

transformation routines that will convert the object or file into the visible display. In some cases, this is nothing more than a simple display of the data contained in the object or file. In most cases, however, some computation will be required to convert the source data into a form suitable for display. In fact, the major portion of the design of a Display consists of transforming input data into visible form.

The design of a visualizer is similar to that of an object or file viewer, but there are some additional aspects that must be considered. The design of the visible elements is generally quite complicated and is, for the most part, beyond the scope of this book. The transformation of data into visible form is extremely complicated, in many cases representing a major development effort. (This is true even for the simple example presented above.) The input data to a visualizer is likely to be handled somewhat differently than for object or file viewers. In some cases the data will be entirely contained within the component. In other cases the data will be presented piecemeal as it becomes available. (Imagine a historical display of temperature data.) In other cases the data may be streamed, with the component firing events to request new data. The design of a visualizer may involve each of these aspects.

The first step in designing a Visualizer is to determine the source, type, and input style of the data being visualized. The next step is to select (or design) the visualization method used to represent the data. The final, and most complicated step is to design the routines that transform the input data into the visible display. In some cases, the visualizer will perform some sort of simulation based on the input data. In such cases, it is necessary to design the timing behavior of the component. One must determine how often the view will be updated and how the information will be presented. Will smooth

animations be provided, or just successive views of the data? It is also necessary to determine what transformations will be performed on the input data for each successive update of the display. These questions can be quite complicated, and are beyond the scope of this book.

10.9. Conclusion

Display components are extremely useful for many different applications. There are three broad categories of Displays, file viewers, object viewers, and visualizers. File and object viewers are designed in much the same way. If an editor exists for the file or object, then it can be used as the basis for a viewer, otherwise the transformation of data into visible form is the major part of the design of the viewer. Visualizers are usually more complicated than viewers. They can be used for a number of purposes. In fact, the visualization of data is an important field in its own right. The visualizer component is an important way to package a visualization algorithm for public use. One should expect to see new and better visualizer components appearing in the future.

10.10. Exercises

1. Create a visualizer (or a pair of visualizers) that will demonstrate the difference between linear search and binary search.
2. Create a visualizer that will demonstrate how binary fan-in works. (Binary fan-in is a parallel algorithm for adding up a list of numbers.)
3. Create a viewer for INI files. See Chapter 17 for the format. Display only one section at a time, with the title at the top.

4. Create a viewer for BMP files. See the Visual C++ documentation for the format of these files.