

8. Background Editors

8.1. 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 background editing, what it is, and why it is used.
2. The ability to create background editors for complex objects..
4. A knowledge of the Background Editor design methodology..

8.2. Introduction

A background editor is an editor with no visible interface. This style of editor is sufficiently different from a graphical editor that it could be considered a separate kind of component. In a background editor, all editing operations must be performed using properties and methods. These facilities can also be built in to a graphical editor, depending on the aims of the project it may be preferable to do so. If one wishes to create a third-party product that will be sold to developers for creating their own programs, then it is advantageous to implement as many features as possible in a single component. This will make the product more attractive simplify the distribution of the third party software. If, on the other hand, the aim is to create a versatile set of tools, and distribute only those functions that are actually required by an application, then it is advantageous to separate all independent functions into separate components.

Developing a Background Editor based on an existing graphical editor is reasonably easy, because much of the functionality will already be implemented. Some of the required methods may already be in place. For example, the delete function will normally already be implemented as a method. To complete the implementation, it is necessary to provide methods for those operations that were performed graphically in the graphical editor. In this chapter we will illustrate the process of creating a Background Editor by presenting the design of a background editor for the simple graphical editor.

8.3. The Methodology

The methodology for creating the Background Editor is straightforward. We first make a list of those operations that are performed through the graphical interface. These are (typically) drawing, selecting, moving, and resizing objects. Because objects are not visible, there must be some facility for enumerating individual objects and their parameters. This is normally the function of an Accessor component, and we will assume the existence of such in the remainder of this chapter. By assuming the existence of an Accessor, we can confine ourselves to implementing the graphical editing functions as component methods.

An attractive alternative is to build the Accessor facility directly into the background editor, and integrate its function with the editing functions such as move, resize, and delete. Of course, all of this functionality could also be built into the Editor itself.

8.4. The Simple Graphical Editor Background Editor

Like the Simple Graphical Editor (SGE), the Background Editor will have an internal object model, but will have no serialization functions. This implies that there must be

some means for accessing the internal model and passing it to other components. We will implement a property named *ModelHandle* that can be used to obtain the address of the internal object model, or to replace the internal object model. Figure 8.1 shows the property design table entry for this property. We also provide a *ReleaseModel* method that is identical to that of the SGE. For brevity we will omit the formal description and definition of properties and methods that are identical to those of the SGE. These include the *Delete*, *Undo*, *Redo*, and *NewDrawing* methods, and the *LineColor*, *FillColor*, and *Dirty* properties.

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. When a new value is assigned to this property, the existing modeling object is destroyed and replaced with the new object. NULL values are ignored.

Figure 8.1. The Model Handle Description.

The **CGraphicList** object will be used for the internal object model of the Background Editor. This will permit internal object models to be shared by these two components, and will greatly reduce the development time for the Background Editor. During the development of this component, we may discover that certain convenient functions are missing from the **CGraphicList** model. Because this model is now shared between two projects, we have chosen to treat the development of the **CGraphicList** object as a separate project. The functions of the **CGraphicList** object and its aggregates will be made available to both projects as a compile-time function library. (This approach should be used for all objects that are used by more than one

project.) Since our objects are physically shared between components, we must make sure to recompile *both* components when the internal object model changes.

The first new methods that we will add are the AddRectangle and the AddCircle methods. The definitions of these methods are given in Figure 8.2 and Figure 8.3, and their implementations are given in Figure 8.3 and Figure 8.4.

Method Description			
Name	AddRectangle		
Return Value	Void		
Description	Adds a rectangle with the given Left, Top, Height, and Width parameters. The current line and fill colors are used to create the new object. If there is no existing object model, this method does nothing.		
Arguments	Name	Type	Description
1	NewLeft	Long Integer	X coordinate of the upper left corner of the new rectangle.
2	NewTop	Long Integer	Y coordinate of the upper left corner of the new rectangle.
3	NewWidth	Long Integer	Width of the new rectangle.
4	NewHeight	Long Integer	Height of the new rectangle.

Figure 8.2. AddRectangle Method Description.

Method Description			
Name	AddCircle		
Return Value	Void		
Description	Adds a circle with the given Center, and Radius. The current line and fill colors are used to create the new object. If there is no existing object model, this method does nothing.		
Arguments	Name	Type	Description
1	NewCenterX	Long Integer	X Coordinate of the center of the new circle.
2	NewCenterY	Long Integer	Y Coordinate of the center of the new circle.
3	NewRadius	Long Integer	Radius of the new circle.

Figure 8.3. AddCircle Method Description.

The implementations of *AddRectangle* and *AddCircle* require two new functions to be added to the *CGraphicList* model. These are *NewRectangle* and *NewCircle*. The existing functions were designed under the assumption that they would be given two mouse-points that defined the upper left and lower right of a rectangle containing the new object. We need two new functions that allow the object parameters to be supplied directly. Figure 8.4 and Figure 8.5 show the implementations of the *AddRectangle* and *AddCircle* methods, while Figure 8.6 and Figure 8.7 show the implementation of the two new functions. These functions make use of the existing Undo facility. These routines also use special constructors for the **CCircle** and **CRectangle** objects. These constructors were omitted from the figures of Chapter 7 to simplify the presentation.

```
// Add a rectangle
void CSGPEditCtrl::AddRectangle(long NewLeft, long NewTop,
                                long NewWidth, long NewHeight)
{
    if (Model != NULL)
    {
        COLORREF FillC;
        COLORREF LineC;
        OleTranslateColor(m_lineColor, NULL, &LineC);
        OleTranslateColor(m_fillColor, NULL, &FillC);
        Model->NewRectangle(NewLeft, NewTop, NewWidth, NewHeight,
                             LineC, FillC);
    }
}
```

Figure 8.4. AddRectangle Implementation.

```
// Add a circle
void CSGEPEditCtrl::AddCircle(long NewCenterX, long NewCenterY,
                               long NewRadius)
{
    if (Model != NULL)
    {
        COLORREF FillC;
        COLORREF LineC;
        OleTranslateColor(m_lineColor, NULL, &LineC);
        OleTranslateColor(m_fillColor, NULL, &FillC);
        Model->NewCircle(NewCenterX, NewCenterY, NewRadius, LineC, FillC);
    }
}
```

Figure 8.5. AddCircle Implementation.

```
void CGraphicList::NewRectangle(long NewLeft, long NewTop,
                                long NewWidth, long NewHeight, COLORREF LineC, COLORREF FillC)
{
    // Create a new rectangle from a list of its internal parameters
    CRectangle * Temp = new CRectangle(NewLeft, NewTop, NewWidth,
                                         NewHeight, LineC, FillC);

    // Link new rectangle into model's object chain
    if (Head == NULL)
    {
        Head = Temp;
    }
    else
    {
        Tail->Next = Temp;
    }
    Temp->Prev = Tail;
    Tail = Temp;
    Count++;
    // Create an ADD OBJECT Undo and add it to the Undo chain
    CUndoAdd * TempUndo = new CUndoAdd(Temp);
    UndoList.PushCommand(TempUndo);
    // Redo list must be cleared
    RedoList.Clear();
    // Model has been modified
    Dirty = TRUE;
}
```

Figure 8.6. NewRectangle Implementation.

```

void CGraphicList::NewCircle(long NewCenterX, long NewCenterY,
    long NewRadius, COLORREF LineC, COLORREF FillC)
{
    // Create a new circle from its list of internal parameters
    CCircle * Temp = new CCircle(NewCenterX,NewCenterY,
        NewRadius,LineC,FillC);
    // Link the circle into the model's object chain
    if (Head == NULL)
    {
        Head = Temp;
    }
    else
    {
        Tail->Next = Temp;
    }
    Temp->Prev = Tail;
    Tail = Temp;
    Count++;
    // Create an ADD OBJECT Undo and link it into the UNDO chain
    CUndoAdd * TempUndo = new CUndoAdd(Temp);
    UndoList.PushCommand(TempUndo);
    // Clear the REDO List
    RedoList.Clear();
    // The model has been modified
    Dirty = TRUE;
}

```

Figure 8.7. NewCircle Implementation.

Next, it is necessary to implement the methods that handle the selection. For the moment we will ignore the problem of creating the selection, and concentrate on the functions that manipulate it. The first method we will add is the *ClearSelect* method, which is a wrapper for the *ClearSelect* function of the **CGraphicList** model. The description of this method is given in Figure 8.8, and its implementation is given in Figure 8.9.

Method Description			
Name	ClearSelect		
Return Value	Void		
Description	Deselects all selected objects. If there is no internal object model, then this method does nothing.		
Arguments	Name	Type	Description
Void			

Figure 8.8. ClearSelect Method Description.

```

// deselect everything
void CSGEPEditCtrl::ClearSelect()
{
    if (Model != NULL)
    {
        Model->ClearSelect();
    }
}

```

Figure 8.9 ClearSelect Implementation.

Next we add the *MoveSelect* and *ScaleSelect* methods, which are also wrappers for the corresponding **CGraphicList** methods. The method descriptions for these are shown in Figure 8.10 and Figure 8.11, and their implementations are shown in Figure 8.12 and Figure 8.13.

Method Description			
Name	MoveSelect		
Return Value	Void		
Description	Moves the selection the distance specified by the NewX and NewY parameters. These parameters are deltas, not absolute coordinates. Their values can be negative. The selection rectangle does not change size during this operation.		
Arguments	Name	Type	Description
1	NewX	Long Integer	The distance, in the X direction, that the upper left corner of the selection rectangle will move.
2	NewY	Long Integer	The distance, in the Y direction, that the upper left corner of the selection rectangle will move.

Figure 8.10. MoveSelect Method Description.

Method Description			
Name	ScaleSelect		
Return Value	Void		
Description	Changes the size of the selection rectangle, using the NewWidth and NewHeight parameters. These parameters are absolute sizes. The upper left corner of the selection does not move.		
Arguments	Name	Type	Description
1	NewWidth	Long Integer	The new width of the selection rectangle.
2	NewHeight	Long Integer	The new height of the selection rectangle.

Figure 8.11. ScaleSelect Method Description.

```

// move selection, NewX and NewY are deltas, not absolute postions
void CSGEPEditCtrl::MoveSelect(long NewX, long NewY)
{
    if (Model != NULL)
    {
        Model->MoveSelection(NewX, NewY);
    }
}

```

Figure 8.12. MoveSelect Implementation.

```

// Scale selection
void CSGEPeditCtrl::ScaleSelect(long NewWidth, long NewHeight)
{
    RECT R1,R2;
    if (Model != NULL)
    {
        Model->GetSelRect(R1);
        R2 = R1;
        R2.right = R2.left + NewWidth;
        R2.bottom = R2.top + NewHeight;
        Model->ScaleSelect(R1,R2);
    }
}

```

Figure 8.13. ScaleSelect Implementation.

Finally it is necessary to provide a method for creating a selection. It is also necessary to provide some sort of enumeration and query facilities to allow the user of the component to determine what drawing objects are contained in the internal model. While it is simple enough to implement such a facility, these functions are more properly part of an Accessor component. Therefore, we will make the following assumptions.

We will assume that an Accessor component for the **CGraphicList** object has already been created, and that the Accessor is able to enumerate the objects contained in a **CGraphicList** model. We will assume further that the Accessor can provide the address of each of these objects on demand, and that this pointer will be cast to a long integer. Finally, we will assume that the Accessor component and the Background Editor share the same internal object model. In most cases, it may be more convenient to implement the enumeration and query features directly in the Background Editor. However, background editing and accessing are two separate functions, and in our examples we wish to encapsulate separate functions as separate components.

To create a selection, it is necessary to use an accessor to enumerate the sub-objects of the graphical model, and pass the addresses of the selected objects to the background

editor. We will use the *AddToSelect* method for this purpose. Figure 8.14 shows the method description of the *AddToSelect* method. This method has one argument, *Handle*, which is assumed to contain the address of a drawing object. This drawing object must be contained in the Background Editor's internal model. Since it is unwise to assume that the address is valid, the *Handle* pointer is validated by looking it up in the component's list of graphical drawing objects. If the pointer is invalid, it is ignored. To validate the *Handle* address, a new function, *IsLegalObject*, has been added to the **CGraphicList** model. Figure 8.15 shows the implementation of the *AddToSelect* method, and Figure 8.14 shows the implementation of the new function *IsLegalObject*.

This completes the implementation of the background editor.

Method Description			
Name	AddToSelect		
Return Value	Void		
Description	Converts the Handle parameter into a drawing object pointer, validates the pointer, and if valid, adds the object to the selection chain. To be valid, the object pointer must exist in the chain of drawing objects of the internal object model.		
Arguments	Name	Type	Description
1	Handle	Long Integer	Pointer to a graphical drawing object, cast to a long integer.

Figure 8.14. AddToSelect Method Description.

```
// get model handle from Accessor, send to this function
// Add handle to selection
long CSGEPeditCtrl::AddToSelect(long Handle)
{
    if (Model != NULL)
    {
        CGraphicObject * Addr = (CGraphicObject *)Handle;
        if (Model->IsLegalObject(Addr))
        {
            Model->AddToSelect(Addr);
            return 0;
        }
        return 1;
    }
    return 0;
}
```

Figure 8.15. AddToSelect Implementation.

```
BOOL CGraphicList::IsLegalObject(CGraphicObject *Obj)
{
    // Look up object in graphical object list.
    // Return TRUE if found, FALSE otherwise.
    for (CGraphicObject * Temp = Head; Temp != NULL ; Temp=Temp->Next)
    {
        if (Temp == Obj)
        {
            return TRUE;
        }
    }
    return FALSE;
}
```

Figure 8.16. IsLegalObject Implementation.

8.5. Conclusion

A Background Editor is usually much simpler than its graphical counterpart, but they have a number of uses. For example, a background editor could be used to implement a tutorial program for its graphical counterpart. A single internal model can be shared between a background editor and the graphical editor. The background editor could be used to create a sample drawing which could then be completed by the student.

In the same vein, a background editor could also be used to create a project wizard for its graphical counterpart. The project wizard could receive a number of design parameters from a user, and then create a partially completed project based on those parameters.

In many cases, the background editing and enumeration facilities will be incorporated into the graphical editor itself.

8.6. Exercises

1. Modify the Background Editor by adding enumeration and query methods and properties. Use the modified component to test adding, moving, scaling, recoloring, and deleting graphical objects. Pass the internal object model to the Simple Graphical Editor for viewing.
2. Incorporate the Background Editor functions into the Simple Graphical Editor, including the enumeration and query facility described in the preceding exercise.