Handling graphical objects with the mouse

Michaël Van Canneyt

February 2, 2009

Abstract

Many applications require some kind of mouse handling to manipulate graphical objects. In vector drawing applications, report designing applications, RAD environments - or even a CD-Cover designing program - moving, resizing, selecting is usually done with the mouse. In this article, the basics of such mouse handling is explained.

1 Introduction

In any graphical designing application, the handling of the mouse is of course very important: the mouse - rather than the keyboard - is used extensively to handle the various objects that make up the design. The CD-Cover application as presented in the a previous issue did not handle the mouse well: the mouse could only be used to drop new objects on the various pages, and to select a single object. More was not possible.

In this article will show how more extensive mouse handling can be designed so that

- Objects can be moved
- Objects can be resized
- Multiple objects can be selected by dragging a rectangle about them.
- The initial size of a newly dropped object can be specified by dragging a rectangle that outlines the object.

And as a corollary, a mechanism to align and resize groups of objects will be presented.

These - generally applicable - techniques will make the CD-Cover application feel less like a toy application, and more like an application that can be put to daily use.

2 Mouse handling in the designer

In an application such as the CD-Cover designing application, the mouse is the most important designing instrument.

The mouse is used for several things: selecting one or more objects, moving or resizing objects, and of course placing new objects on the page under design. This happens the same in most applications:

- Selecting an object can be done by clicking it with the left mouse button. If more objects must be added to the selection, the shift key must be held while clicking the additional objects. An alternative is usually drawing a rectangle which encompasses all objects that must be selected.

- Moving the objects is done by dragging them till the desired location.
- Resizing is usually done by dragging the edges till the desired size is reached.
- In the previous article, a new object was placed on a page by clicking on the location where it should be placed, after which the object was created with default width and heights. This will be changed to a method where the outline of the object must be drawn, after which it will be created with the specified height and width.

All this must be accomplished with only 3 events: The `MouseDown`, `MouseUp` and `MouseMove` events. There are many ways in which a designing application can handle the mouse events - a not so trivial task as it may seem - and the one presented here is just one of the options. Unix users may be familiar with the `xfig` drawing program, which takes a quite different approach to this subject.

What most sources seen by the author seem to have in common is that these 3 events are filled with all the code needed to perform all operations. Here a different approach is taken: the code in the mouse event handlers is just a dispatch towards different methods, which handle the rest of the operation. The dispatch is done on the basis of the current operational state or mode of the designer. The current mode is indicated by the following enumerated type:

```
TEditorMode = (emNeutral,emRubberBand,emStartAdd,emAdding,
               emStartMoving,emMoving,emSizing);
```

The various modes should be sel-explaining, although the difference between `emStartAdd` and `emAdding` or likewise `emStartMoving` and `emMoving` will need some explanation. The current state of the designer is kept in the `TCoverEditor` class' `FEditorMode` field.

Additional state information is kept when the editor is resizing an object. This is done with the `TSizeMode` enumerated, and it's corresponding set type `TSizeModes`:

```
TSizeMode   = (smUp,smDown,smLeft,smRight);
TSizeModes  = Set of TSizeMode;
```

The various elements indicate which edge of an object is being dragged: a combination of 2 elements means that a corner is being dragged: for example the set `[smUp,smLeft]` means that the top-left corner is being dragged. This information is kept in the `FSizeModes` field.

3 Handling a click

Since most operations start with a click, and a click consists of a `MouseDown` followed by a `MouseUp` event, the `MouseDown` event will be examined first:

```
procedure TCoverEditor.DoPanelMouseDown(Sender: TObject;
                                        Button: TMouseButton;
                                        Shift: TShiftState;
                                        X, Y: Integer);

Var
  C : TCanvas;
  O   : TCoverObject;
```

```
begin
  C:=TCustomPanel(Sender).Canvas;
  Case EditorMode of
    emNeutral : begin
                If (Button=mbLeft) then
                  begin
                  if (FSizeModes=[]) then
                    begin
                    O:=GetObjectAt(X,Y);
                    If (O=Nil) or (Shift=[ssCtrl]) then
                      StartRubberBand(X,Y)
                    else if InSelection(O) then
                      StartMoving(X,Y)
                    end
                  else
                    StartSizing(X,Y);
                  end
                else
                  ClearSelection;
                end;
    emStartAdd : StartAdding(C,X,Y);
  end;
end;
```

If the editormode is `emStartAdd`, meaning that the user has selected an object type he wishes to add from the menu, then the click means the start of the outlining operation. In that case, the `StartAdding` method is called.

If the editor mode is `emNeutral` - the user is simply moving and then clicking - then a mouse down event (for the left mouse button) can mean one of 3 things:

1. The user will start drawing a rectangle to select objects.
2. The user clicks an object to select or move it.
3. The user will start sizing an object.

Deciding between these 4 cases is done by first checking the `FSizeModes` field: if it is not empty, then a sizing operation is started with a call to `StartSizing`. If the `FSizeModes` is empty, then the `GetObjectAt` method is called to decide whether an object was clicked. If it was not, or the `Ctrl` key was pressed, then it is assumed that a rubber band must be drawn to select a set of objects: `StartRubberBand` is called. In case a click occurred on an object, the `StartMoving` method is called.

The various `Start*` methods are quite simple: for the most part they store the position of the mouseclick, and change the editor mode for use in the mousemove or mouseup events:

```
procedure TCoverEditor.StartMoving(X,Y: Integer);

begin
  FEditorMode:=emStartMoving;
  FLastX:=X;
  FLastY:=Y;
end;

procedure TCoverEditor.StartSizing(X,Y: Integer);
```

```
begin
  FEditorMode:=emSizing;
  FLastX:=X;
  FLastY:=Y;
end;

procedure TCoverEditor.StartRubberBand(X,Y: Integer);

begin
  FEditorMode:=emRubberband;
  FLastFocusRect:=Rect(X, Y, X, Y);
end;
```

In the last method, the initial size (empty) of the focus rectangle is stored in the `FLastFocusRect` variable.

4 Handling mouse movement

In most cases, after the `MouseDown` event, a `MouseMove` event follows: even when just clicking an object, chances are that the user lets go of the mousebutton not on the exact spot where he clicked; some fuzzyness should therefor be taken into account: hence the difference between the `emStartMoving` and `emMoving` states.

The decision between moving and a mere click is made in the `MouseMove` event:

```
procedure TCoverEditor.DoPanelMouseMove(Sender: TObject;
                                        Shift: TShiftState;
                                        X,Y: Integer);
Var
  C : TCanvas;
  DX,DY : Integer;

begin
  C:=TCustomPanel(Sender).Canvas;
  Case EditorMode of
    emNeutral : If (SelectionCount=1) then
                  FSizeModes:=CheckResizeCursor(X,Y);
    emRubberband,
    emAdding : DrawRubberBand(C,X,Y);
    emStartMoving,
    emSizing,
    emMoving : begin
               DX:=X-FLastX;
               DY:=Y-FLastY;
               If EditorMode=emSizing then
                 SizeObject(C,DX,DY)
               else If (SelectionCount>0) then
                 begin
                 If (Abs(DX)>=2) or (Abs(DY)>=2) then
                   FEditorMode:=emMoving;
                 If (FEditorMode=emMoving) then
                   MoveObjects(C,DX,DY);
```

```
                end;
              if (EditorMode<>emStartMoving) then
                begin
                FLastX:=X;
                FLastY:=Y;
                end;
              end;
  end;
end;
```

If the state of the editor is neutral - the user is just moving the mouse - then a check is needed whether the cursor is above the edge of an object or not, and if it is, the cursor should be set to one of the stock sizing cursors. This is done in the `CheckResizeCursor` routine, which returns the new `FSizeModes` value - which was used in the `OnMouseDown` event to start a resize operation.

In the case of the `emRubberband` and `emAdding` states, a rubber band must be drawn while dragging the cursor: in the former state to select objects, in the latter to outline the size of the object to be added. This is done in the `drawRubberBand` method.

In the `emStartMoving`,`emMoving` and `emSizing` states, first the difference between the current mouse location and the last saved mouse location is calculated, because it is needed for all operations that follow:

1. In the `emSizing` state, the current object is resized with amount of mouse movement, using the `SizeObject` call.
2. In the other states (`emStartMoving` or `emMoving`) first a check is made to see if one or more objects are selected. If so, and the editormode is still `emStartMoving`, big enough, and the state is `emStartMoving`, then the state is switched to `emMoving`. If the state is `emMoving`, the object(s) in the selection are actually moved in the `Moveobjects` method

Finally, if the state is not `emStartMoving`, the last position is set to the current position. If the last position was always reset, then moving the mouse very slowly would never result in a start of the moving operation. As it is now, a 'fuzzy' click operation is recognized as a click, and does not result in the object being moved a couple of pixels.

The `CheckResizeCursor` operation - where a check is done if the cursor is over the edge of an object, and a resize operation could be started - is quite lengthy, because a lot of cases must be checked. The code starts by checking if the cursor is over the currently selected object: the `FLastObjectRect` rectangle contains the outline of the object. This rectangle is inflated with a couple of pixels, again to account for a certain fuzzyness. The bottom/right size of the rectangle is then inflated with 1 pixel, because the `ptInRect` function returns false if the point is on the bottom/right edges:

```
Function TCoverEditor.CheckResizeCursor
   (X,Y: Integer): TSizeModes;

Const
  w = 1; // sensitivity

  Function InInterval(Const Y,O : integer) : Boolean;

  begin
    Result:=(Y>=O-W) and (Y<=O+W);
```

```
  end;

Var
  R : TRect;

begin
  Result:=[];
  R:=FLastObjectRect;
  InflateRect(R,W,W);
  R.Bottom:=R.Bottom+1;
  R.Right:=R.Right+1;
  If ptInRect(FLastObjectRect,Point(X,Y)) then
    begin
    R:=FLastObjectRect;
    If InInterval(Y,R.Top) then
      begin
      Result:=[smUP];
      if InInterval(X,R.Right) then
        Result:=Result+[smRight]
      else if InInterval(X,R.Left) then
        Result:=Result+[smLeft]
      end
```

As can be seen here, after the code has decided that the mouse cursor is somewhere above the selected object, it continues to check whether the cursor is over the top edge, and if so, sets the result to the appropriate values. Note the use of the `InInterval` function, which takes a certain amount of fuzzyness into account. (the larger the constant `w`, the more fuzzyness is allowed)

A similar check is done if the cursor is over the bottom, right or left edges of an object:

```
    else If InInterval(Y,R.Bottom) then
      begin
      Result:=[smDown];
      if InInterval(X,R.Left) then
        Result:=Result+[smLeft]
      else if InInterval(X,R.Right) then
        Result:=Result+[smRight];
      end
    else if InInterval(X,R.Left) then
      Result:=[smLeft]
    else if InInterval(X,R.Right) then
      Result:=[smRight];
    end;
  Screen.Cursor:=CursorFromSizeMode(Result);
end;
```

As the last step, the current cursor is set to whatever value was determined during the various checks. The calculation is done in the `CursorFromSizeMode` function - the reader can check the details in the code on the disc accompagnying this issue.

The `SizeObject` function is rather simple. It checks which sides should be changed and adds the needed amount (in real units, not pixels) to the top/left position and/or the width/height of the object. Note that the opposite edge must be kept on its position, which requires some extra logic:

```
procedure TCoverEditor.SizeObject(ACanvas : TCanvas;
                                  DX,DY : Integer);

Var
  UDX,UDY : Double;
  DPI : Integer;
  O : TPoint;
  CO : TCoverObject;

begin
  DPI:=CurrentDPI;
  UDX:=ToUnits(DX,DPI);
  UDY:=ToUnits(DY,DPI);
  CO:=FSelection[0];
  If smLeft in FSizeModes then
    begin
    CO.Left:=CO.Left+UDX;
    CO.Width:=CO.Width-UDX;
    end;
  If smRight in FSizeModes then
    CO.Width:=CO.Width+UDX;
  if smUp in FSizeModes then
    begin
    CO.Top:=CO.Top+UDY;
    CO.Height:=CO.Height-UDY;
    end;
  If smDown in FSizeModes then
    CO.Height:=CO.Height+UDY;
  O:=GetPageOrigin(FCurrentPanel,FCurrentPage,DPI);
  FCurrentPage.DrawPage(ACanvas,O,DPI);
  DrawSelectedObjects(ACanvas,O,DPI);
end;
```

As the last step, all objects are redrawn as objects may be covered or uncovered as a result of the resizing.

Moving the selected objects is simpler, but also involves the redraw of all objects:

```
procedure TCoverEditor.MoveObjects(ACanvas : TCanvas;
                                   DX,DY : Integer);

Var
  UDX,UDY : Double;
  I,DPI : Integer;
  O : TPoint;

begin
  ClearFocusRect(ACanvas);
  DPI:=CurrentDPI;
  UDX:=ToUnits(DX,DPI);
  UDY:=ToUnits(DY,DPI);
  For I:=0 to FSelection.Count-1 do
    begin
    FSelection[i].Left:=FSelection[i].Left+UDX;
    FSelection[i].Top:=FSelection[i].Top+UDY;
```

```
    end;
  O:=GetPageOrigin(FCurrentPanel,FCurrentPage,DPI);
  FCurrentPage.DrawPage(ACanvas,O,DPI);
  DrawSelectedObjects(ACanvas,O,DPI);
end;
```

Note the `ClearFocusRect` call which is done at the start: the focus rectangle (drawn around the selected object) is cleared prior to drawing all objects.

The last operation initiated by the mousemove - drawing a rubber band - also starts by clearing the focus rectangle:

```
procedure TCoverEditor.DrawRubberBand(ACanvas : TCanvas;
                                      X,Y : Integer);

begin
  ClearFocusRect(ACanvas);
  FLastFocusRect.Right:=X;
  FLastFocusRect.Bottom:=Y;
  ACanvas.DrawFocusRect(FLastFocusRect);
end;
```

It then sets the new size of the focus rectangle, and draws it again.

Clearing the focus rectangle is done simply by drawing it:

```
procedure TCoverEditor.ClearFocusRect(ACanvas : TCanvas);

begin
  If (FLastFocusRect.Top<>FLastFocusRect.Bottom) and
     (FLastFocusRect.Right<>FLastFocusRect.Left) then
    ACanvas.DrawFocusRect(FLastFocusRect);
end;
```

A focus rectangle is drawn with the `DrawFocusRect` method of `TCanvas`: this method draws a rectangle in `XOR` mode: each pixel of the rectangle is XOR-ed with the pixel that was there prior to drawing. This has the effect that drawing a focus rectangle on the same location twice clears the rectangle !

5 The mouse button is released

Finally, the `MouseUp` message must be treated. This is actually a relatively simple routine: when the mouse button is released, this ends the operation that was in progress, meaning that the code simply should check which operation this was, and call the appropriate method:

```
procedure TCoverEditor.DoPanelMouseUp(Sender: TObject;
                                      Button: TMouseButton;
                                      Shift: TShiftState;
                                      X, Y: Integer);

Var
  C : TCanvas;
```

```
begin
  C:=TCustomPanel(Sender).Canvas;
  If (Button=mbLeft) then
    Case EditorMode of
    emAdding:
      begin
      AddObjectAt(FAddObjectClass,X,Y);
      FAddObjectClass:=Nil;
      end;
    emNeutral,
    emStartMoving:
      DoSelectObject(C,Button,Shift,X,Y);
    emSizing:
      FSizeModes:=[];
    emRubberBand:
      begin
      ClearFocusRect(C);
      SelectObjectsInRectangle(FLastFocusRect);
      FillChar(FLastFocusRect,SizeOf(Trect),0);
      end;
    end;
  FEditorMode:=emNeutral;
  Screen.Cursor:=crDefault;
end;
```

The actual work is done in the subroutines, after which the code resets the editor mode, and resets the cursor. The `AddObjectAt` routine was presented in the previous article. The `DoSelectObject` procedure is a very simple procedure:

```
procedure TCoverEditor.DoSelectObject(ACanvas : TCanvas;
                                      Button : TMouseButton;
                                      Shift : TShiftState;
                                      X,Y : Integer);

Var
  DPI : Integer;
  O   : TCoverObject;
  Ori : TPoint;

begin
  DPI:=CurrentDPI;
  ORI:=GetPageOrigin(FCurrentPanel,FCurrentPage,DPI);
  O:=FCurrentPage.GetObjectAt(ToUnits(X-Ori.X,DPI),
                              ToUnits(Y-Ori.Y,DPI));
  If not (ssShift in Shift) then
    ClearSelection
  else
    DrawSelectedObjects(ACanvas,Ori,DPI);
  If Assigned(O) then
    begin
    If Inselection(O) then
      RemoveFromSelection(O)
    else
      AddToSelection(O);
```

```
    DrawSelectedObjects(ACanvas,Ori,DPI);
    FLastObjectRect:=O.GetObjectRect(ACanvas,ORI,DPI);
    end;
end;
```

The first three lines search the displayed page for an object located at the cursor position. The actual searching is done using the `GetObjectAt` method of the `TCoverPage` class, which expects natural units, and hence requires a coordinate transformation. The rest of the routine is concerned with handling the actual selection, and drawing the focus indicators.

The `AddObjectsInRectangle` routine is a variation on the `DoSelectObject` routine and will not be repeated here.

6 Using the selection

Now that the mouse can be used to select one or more objects, these objects can be aligned or matched in size: a common operation in visual designer programs. These operations can be done indepenent of the designer: the designer just serves to select the objects, the actual resizing or aligning is done by the low-level data objects.

Here the operation was implemented in the selection list object, which is defined as follows:

```
THAlignAction = (haNone,haLeft,haCenter,haRight,haSpace,haCentp);
TVAlignAction = (vaNone,vaTop,vaCenter,vaBottom,vaSpace,vaCentp);

TSizeAdjust = (saNone,saLargest,saSmallest,saValue);

TSelectionList = Class(TFPList)
Public
  Procedure AlignObjects (Hor : THAlignAction;
                          Ver : TValignAction);
  Procedure AdjustSizes(Height : TSizeAdjust; HSize : Double;
                        Width : TSizeAdjust; WSize: Double);
  Property Page : TCoverPage Read FPage Write SetPage;
  Property CoverObjects[Index : Integer] :
             TCoverObject Read GetC; default;
end;
```

The `AlignObjects` aligns the objects in the list according to the 2 alignment specifications (one horizontal, one vertical: `THAlignAction` and `TVAlignAction`), while the `AdjustSizes` adjusts the sizes of the objects - again using a vertical and horizontal specification, this time specified with the `TSizeAdjust` enumerated type.

Both methods are a simple loop over all objects, adjusting the dimension or location of the objects. The `AdjustSizes` method looks as follows:

```
procedure TSelectionList.AdjustSizes(
          Height: TSizeAdjust; HSize: Double;
          Width: TSizeAdjust; WSize: Double);

Var
  i : Longint;
  C : TCoverObject;
begin
  If Count<1 then exit;
```

```
  C:=CoverObjects[0];
  If (Height in [saSmallest,saLargest]) then
    HSize:=C.Height;
  If (Width in [saSmallest,SaLargest]) Then
    WSize:=C.Width;
  For I:=1 to Count-1 do
    begin
    C:=CoverObjects[i];
    Case Height of
      saSmallest : If HSize>C.Height then HSize:=C.Height;
      saLargest : If HSize<C.Height then HSize:=C.Height;
    end;
    Case Width of
      saSmallest : If WSize>C.Width then WSize:=C.Width;
      saLargest : If WSize<C.Width then WSize:=C.Width
    end;
    end;
  For I:=0 to Count-1 do
    begin
    C:=CoverObjects[i];
    If (Height<>saNone) then C.Height:=HSize;
    If (Width<>saNone) then C.Width:=WSize;
    end;
end;
```

The parameters `HSize` and `WSize` are used to specify a fixed dimension, used when the resizing action is `saValue`. In the case of `saSmallest` and `saLargest`, first a loop is run to determine, respectively, the smallest or largest dimension of the objects in the list.

The `AlignObjects` method is slightly more involved, but is essentially also 2 loops over all objects. The interested reader can consult the source code on the disc.

Integrating these operations in the GUI of the designer application is quite straightforward: a menu entry is added to the 'Edit' menu, which displays a dialog that asks the parameters for the operation, and then the appropriate method is called. For the alignment operations, buttons can be put on a toolbar - one per alignment method. The result can be observed in figure 1 on page 12.

7 Conclusion

The mouse handling presented here can serve for any graphica designing application. Implementing it in the CD-Cover application improves the latter significantly, giving the application a more real-world feel. Common operations related to a selection of objects: are still missing: an undo function, copy & paste operations, both of which may be the subject of a future contribution.

Figure 1: The resize dialog