

## Capturing the Mouse

In order to allow the user to drag something, you need to keep track of whether the mouse is "down" or "up". It is "down" from the *MouseDown* event to the subsequent *MouseUp* event.

What if the user begins to drag something in your window, but then moves the mouse off of your window while the mouse is down, and release the mouse in some other window? As far as we have so far discussed, you would not get the *MouseUp* event—it would go to that other window. Unless we're talking about programming a drag-and-drop operation, that would not be what you want. Normally, you would want to receive any mouse messages from the mouse between the *MouseDown* and the subsequent *MouseUp*. This is called *capturing the mouse*. It is not automatic in Windows itself, but the Foundation Class Libraries do provide it for you automatically (unlike MFC or the Win API).

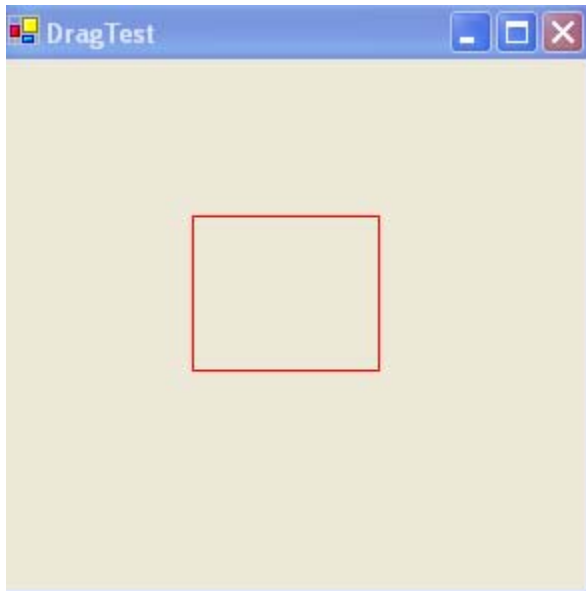
Explicitly: Every *MouseDown* event (except those corresponding to the second click of a double-click) automatically captures subsequent mouse input until there is a *MouseUp* event involving the same button.

You can test for whether the "mouse is down" by checking the *Capture* property of your form. Technically, *Capture* is a boolean member of the *Control* class, but *Form* is derived from *Control*, so your form has a *Capture* member.

## Dragging Example

Make a new Visual C# Windows Application called *DragTest*. We want to write code that outlines a red rectangle with one corner at (50,50), and lets the user drag the diagonally opposite corner

with the mouse. This is a useful thing to do because it is the basic interface that allows the user to select a rectangle.



To do this we start by adding two member variables of type *Point*:

```
private Point m_theAnchor; // this will always be (50,50)
private Point m_CurrentPoint;
```

These variables are initialized in the form constructor with *new Point()*;

The reason that we use *Point* variables instead of *Rectangle* variables is that a rectangle is not allowed to have a negative width or height. This will make it hard to program when the user drags the mouse above and to the left of the “anchor” point (50,50). In MFC or the Windows API, rectangles are specified by any two points (for diagonally opposite corners), so the “left” field could actually be greater than the “right” field of a rectangle, which was convenient in some situations, like this one, but made it easy to make mistakes. In the FCL, you can’t make those mistakes, and you still can program examples like this one, as we’ll see.

Handle the *Paint* event and write *Form1\_Paint* so that it outlines the rectangle:

```
private void Form1_Paint(object sender,
System.Windows.Forms.PaintEventArgs e)
{ // make a rectangle with m_theAnchor and m_CurrentPoint
as
  // two diagonally opposite corners
  int left = Math.Min(m_theAnchor.X, m_CurrentPoint.X);
  int width = Math.Abs(m_theAnchor.X - m_CurrentPoint.X);
  int top = Math.Min(m_theAnchor.Y, m_CurrentPoint.Y);
  int height = Math.Abs(m_theAnchor.Y - m_CurrentPoint.Y);
  Rectangle r = new Rectangle(left,top,width,height);
  Pen p = new Pen(Color.Red);
  e.Graphics.DrawRectangle(p,r);
}
```

Now create a handler for each of the three events *MouseDown*, *MouseUp*, and *MouseMove*. The following code updates *m\_CurrentPoint* at each *MouseMove* that occurs while the mouse is down, and at each *MouseUp*.

```
private void Form1_MouseDown(object sender,
System.Windows.Forms.MouseEventArgs e)
{ if(e.Button != MouseButton.Left)
  return;
  // FCL automatically sets the Capture property to true
before calling this method.
  m_theAnchor.X = e.X;
  m_theAnchor.Y = e.Y;
  m_CurrentPoint = m_theAnchor;
}

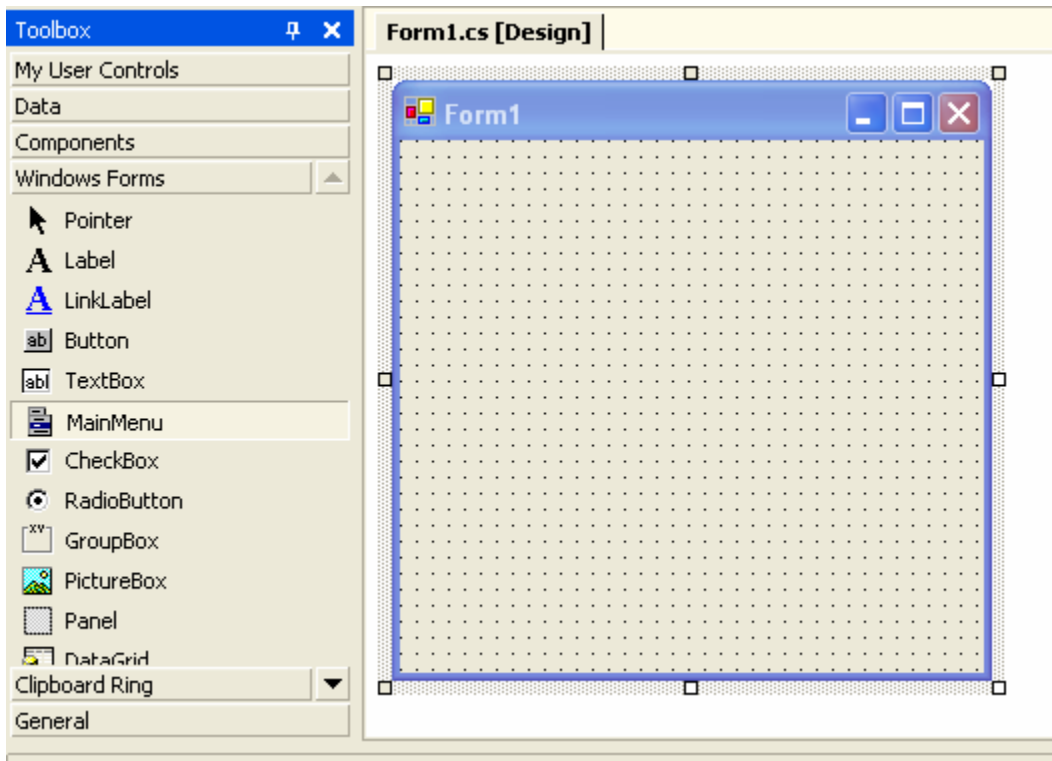
private void Form1_MouseMove(object sender,
System.Windows.Forms.MouseEventArgs e)
{ if(e.Button != MouseButton.Left)
  return;
  if(!Capture)
    return; // we're not dragging. The mouse is moving
while the button is up.
  m_CurrentPoint.X = e.X;
  m_CurrentPoint.Y = e.Y;
  Invalidate();
}
```

```
private void Form1_MouseUp(object sender,
System.Windows.Forms.MouseEventArgs e)
{   if(e.Button != MouseButton.Left)
        return;    // don't check Capture property, it's
already been set to false.
    m_CurrentPoint.X = e.X;
    m_CurrentPoint.Y = e.Y;
    Invalidate();
}
```

## Dragging using a Child Window

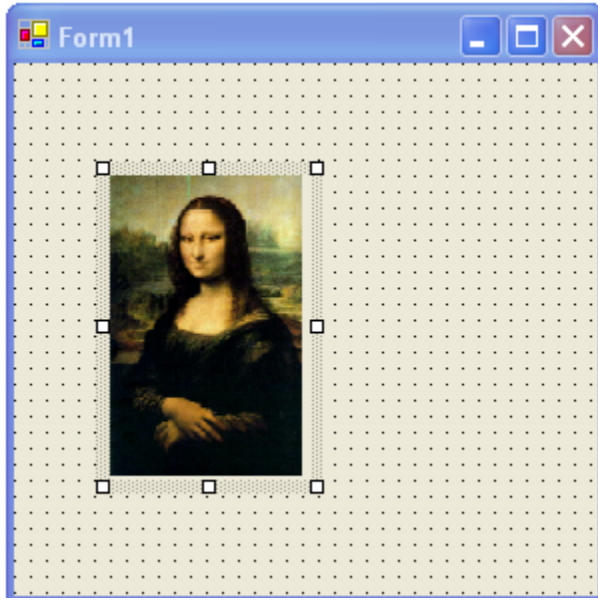
Often it is convenient to put whatever it is you want to drag into a small window of its own, and let Windows take care of the details of redrawing that window in a new location. The underlying function in the Win32 API is called *MoveWindow*. It makes use of the hardware on your graphics card to copy a rectangle from one place to another during the “vertical retrace interval”, so that your program doesn’t cause flicker. In the FCL, this function is invisibly invoked when you set the *Location* property of a window. The plan for this program is roughly as follows: create a child window, make it handle *MouseDown*, *MouseUp*, and *MouseMove*, and in *MouseMove*, if the *Capture* property is true (indicating that we’re dragging), change the *Location* property of the child window.

How do we create a child window? The easiest way is to use Visual Studio’s *Toolbox*. If you don’t see the *Toolbox*, choose *View / Toolbox* from the Visual Studio main menu.



If you don't see this, maybe it's because you have your code window on top instead of your design window. To see the toolbox as shown you must have your design window on top.

Later on we will work with all the different “controls” shown in the Toolbox. For now, drag and drop a “PictureBox” from the toolbox to your form. This is just a convenient and quick way to create a child window in which we can display an image. Bring up the property sheet of your picture box and look for the *Image* property. Browse for an image file, for example, a local copy of the file *monalisa.bmp* available on the course website. (First download the file to your local computer.) The default size of the picture box is pretty small—resize it with the mouse on the form design until it just holds the image.



Now if you build and run your program, you'll see the image displayed.

Next, add handlers to the PictureBox for *MouseDown*, *MouseUp*, and *MouseMove*. Be sure that you are adding them to the PictureBox and not to *Form1*.

We will need two member variables of type *Point*:

```
private Point m_mouseDownHere, m_lastLocation;
```

These should be initialized simply with *new Point()*. The idea is that *m\_lastLocation* is going to be set on *MouseDown*, and again at the end of each *MouseMove*, so that we can keep track of where the PictureBox was last displayed. The coordinates of *m\_lastLocation* are in the form's client coordinates. But we also need to keep track of where in the PictureBox the mouse cursor is located. That should stay fixed relative to the upper-left corner of the PictureBox, as we drag. The variable *m\_mouseDownHere* will give the location of the cursor in PictureBox client coordinates. That value will stay fixed during the entire dragging operation. In *MouseMove*, to compute the new location, we have to first figure

out how much the mouse has moved since the last display. The mouse is now at  $(e.X, e.Y)$ , where  $e$  is the *MouseEventArgs* object, and the mouse was at *m\_mouseDownHere*. So the difference of these two points is the vector by which we moved. If we add that to *m\_lastLocation*, we'll get the new location.

Here's the code:

```
private void pictureBox1_MouseDown(object sender,
System.Windows.Forms.MouseEventArgs e)
{ if(e.Button != MouseButton.Left)
    return;
  m_lastLocation = pictureBox1.Location;
  // original location of the window in Parent coordinates
  m_mouseDownHere.X = e.X; // child window coordinates of
the mouse
  m_mouseDownHere.Y = e.Y;
}

private void pictureBox1_MouseMove(object sender,
System.Windows.Forms.MouseEventArgs e)
{ if(pictureBox1.Capture == false)
    return;
  pictureBox1.Location = new Point(m_lastLocation.X + e.X
- m_mouseDownHere.X,
                                  m_lastLocation.Y + e.Y
- m_mouseDownHere.Y);
  m_lastLocation = pictureBox1.Location;
}

private void pictureBox1_MouseUp(object sender,
System.Windows.Forms.MouseEventArgs e)
{ pictureBox1.Location = new Point(m_lastLocation.X + e.X -
m_mouseDownHere.X,
                                  _lastLocation.Y + e.Y -
m_mouseDownHere.Y);
}
```

This is all the code it takes! Now you can drag the Mona Lisa around the screen, and there's no flicker at all. Observe that it was

the PictureBox that captured the mouse, when you clicked in the PictureBox. That's why you have to write *pictureBox1.Capture* in the *MouseMove* handler, not just *Capture*.

### *Technical Detail:*

To set the location, you have to assign the *PictureBox1.Location* property to a new point. You can't just make assignments to *pictureBox.Location.X* and *pictureBox.Location.Y*; you will get an error message if you try. Here is the reason: *Point* is a "value type", technically a "struct" rather than a "class". That means that it is "returned by value", so that when you access the property *pictureBox1.Location* you are actually getting a copy of the point. In other words, when you write *pictureBox1.Location*, there is an implicit *function call* to a function that "gets" the value of the *Location* property, and that "getter" function returns the location by value. If you then change the *.X* or *.Y* fields of this returned value, of course it has no effect on the actual *Location* point, which was copied to get the returned value. This is a somewhat technical point, but the more details of your programming language you understand, the better.

To complete your study of this program, examine the source code to see what the Design Editor wrote for you to create the PictureBox: First, it declared a member variable

```
private System.Windows.Forms.PictureBox pictureBox1;
```

and then it initialized it as follows:

```
System.Resources.ResourceManager resources = new
    System.Resources.ResourceManager(typeof(Form1));
this.pictureBox1 = new System.Windows.Forms.PictureBox();
this.SuspendLayout();

this.pictureBox1.Image =
((System.Drawing.Image)(resources.GetObject("pictureBox1.Im
age")));
this.pictureBox1.Location = new System.Drawing.Point(48,
56);
this.pictureBox1.Name = "pictureBox1";
this.pictureBox1.Size = new System.Drawing.Size(100, 152);
this.pictureBox1.TabIndex = 0;
this.pictureBox1.TabStop = false;
```

You can see that the filename *monalisa.bmp* hasn't been mentioned here. Instead, fetching the actual image is done by *resources.GetObject*. You can verify, though, that the .exe file will run even if it's placed in a new folder without the file *monalisa.bmp*. The image itself has been included in the .exe file. The word "resources" in Windows applies to various kinds of data that are not computer code but are nevertheless packed into the .exe file of a program. Your predecessors, who programmed in the Win32 API or in MFC, often had to work much harder to get programs to display images. Displaying the Mona Lisa here was no problem at all, just "incidental" to the problem of dragging something.

To finally make the point that we could be dragging *anything* this way, map the *Paint* event in the picture box. Let's put a caption on the image:

```
private void pictureBox1_Paint(object sender,
System.Windows.Forms.PaintEventArgs e)
{ Brush b = new SolidBrush(Color.White);
  Font f = new Font("Arial",8);
  Graphics g = e.Graphics;
  g.DrawString("Mona Lisa",f,b,25,85);
}
```



You wouldn't actually have needed to set the *Image* property of the picture box. You could have just left it without an image, and drawn anything you liked in there using GDI+ graphics.