

## Responding to the Mouse

The mouse has two buttons: left and right.  
Each button can be depressed and can be released.

Here, for reference are the definitions of three common terms for actions performed by a user with the mouse:

*Click:* to depress and then release a button without moving the mouse. [Technically, there may be a pixel or two of motion allowed between the depress and the release.]

*Drag:* to depress, move, and then release a mouse button.

*Double-click:* to click twice within the *double-click interval*.

The double-click interval can be changed by the user through the Windows Control Panel. People with disabilities, for example, may want it to be longer. It can also be changed under program control but I have never heard of this being done, and it doesn't seem like a good idea.

These actions generate hardware interrupts, which Windows processes by constructing messages, which in turn are wrapped by *Application.Run* into FCL events.

## **What window gets the mouse message?**

The *cursor location* is a point (pixel) on the screen. The cursor may be invisible but it always has a location. The cursor location is sometimes called the "hot spot" of the cursor. The cursor itself is indicated by an icon, whose appearance changes according to the function of the mouse at that point in the program.

Normally, the mouse messages will go to the window that contains the cursor location and is currently visible; or in FCL terminology, mouse events should be processed by the form whose window contains the cursor location. Technically, your form's mouse event handlers will normally get mouse events only if those events occur over the form's client area. Events that occur over the border, caption bar, menu bar, toolbar, scroll bars, minimize box, maximize box, or close box, will be handled by Windows rather than by your program.

There is one exception to this rule: when the user is dragging something, usually the programmer has "captured the mouse" by calling *SetCapture*. We will return to this issue later. As long as we're only talking about clicks (mouse-down and mouse-up, or double-click) and not about dragging, it should be true that mouse events are processed by the form whose window contains the cursor location.

## **Mouse Events in .NET**

Create a new Visual C# Windows Application project, say *MouseTest*. Right-click the form in the form designer, select

Properties, click the lightning bolt, and look for the mouse events. There are five events whose names begin with *Mouse*:

*MouseDown*

*MouseUp*

*MouseEnter*

*MouseLeave*

*MouseHover*

For now, we will concentrate on the most common mouse event, *MouseDown*. Add a handler for that event (by double-clicking its name in the list of events).

## A mouse programming example

To make a simple *MouseTest* program, let's add a *Rectangle* member variable to the form class, call it *m\_theRect*. We will make something happen when the user clicks in the rectangle, or more precisely when the left mouse button is depressed while the cursor location is in the rectangle. For example, change the color of the rectangle. We will keep the current color value in another member variable, *m\_color*.

```
private Rectangle m_theRect;  
private Color m_theColor;
```

Initialize these variables in the form class constructor:

```
m_theRect = new Rectangle(10,10,100,100);  
m_theColor = Color.RoyalBlue;
```

As we have done before, we add a handler for the *Paint* event, and use *FillRectangle* to paint *m\_theRect* using the color *m\_theColor*:

```
private void Form1_Paint(object sender,  
System.Windows.Forms.PaintEventArgs e)  
{ Brush b = new SolidBrush(m_theColor);
```

```
e.Graphics.FillRectangle(b,m_theRect);  
}
```

## Hit-testing

This refers to checking whether the mouse button has been pressed in a certain region or not. It is simplest if the region is a rectangle. The *Rectangle* class has a method *Contains*, which can take either a *Point* or two *int* arguments. The following short code sample illustrates simple hit-testing.

```
private void Form1_MouseDown(object sender,  
System.Windows.Forms.MouseEventArgs e)  
{ if(m_theRect.Contains(e.X,e.Y))  
  { // toggle the color  
    if(m_theColor == Color.RoyalBlue)  
      m_theColor = Color.Red;  
    else  
      m_theColor = Color.RoyalBlue;  
    Invalidate(m_theRect);  
  }  
}
```

This code sample also illustrates another important point: in responding to the mouse, we *do not do any drawing*. Instead, we only change some data, in this case, *m\_theColor*. We leave it up to *Form1\_Paint* to do the drawing. The key to this is *Invalidate*. Comment out the call to *Invalidate* and run the program. You will see that the color doesn't change when you click. But of course the variable *m\_theColor* has changed—you just don't see the changed data presented on the screen! Resize the window to partly obscure the rectangle and then reveal it again—then you'll see the change on the newly redrawn part of the window. Please review the discussion of *Invalidate* in the last lecture with this sample program running. You should re-read the discussion and re-run the program several times if necessary, until you fully understand what is going on. This point is extremely fundamental for successful Windows programming and you must put in the effort required to master the points in question. Just to summarize,

there are actually two related points: (1) Don't draw anywhere but in your *Paint* handler; (2) elsewhere, just change data and then cause a new *Paint* event to be generated by calling *Invalidate*.

## Handling Double Clicks

When you get the first click, you don't know if it is a single click, or the first click of a double click. Therefore, you have to design your program accordingly--you can't do something on double-click that will contradict what is done on a single click.

For example, a single click sometimes highlights ("selects") a filename and a double-click opens the file. No problem there; but it wouldn't work to have a single click open the file and a double click merely select it.

When the mouse is clicked twice within the double-click interval, you will, in .NET programming, get **two** *MouseDown* events, as you can verify by running the example program above. There is no difference, in that program, between a double click and two single clicks. This contrasts with the behavior of a similar program written in MFC or the Win32 API, where programmers had to explicitly process the double-click message in order to get this behavior.

In FCL programming, you have the opposite problem: suppose you want to detect a double click and do something in that case. For example, in our *MouseTest* program, we might want to have a double-click turn the rectangle green. That means we have to detect the second click of a double-click. That is done using the *Clicks* field of the *MouseEventArgs* class:

```
private void Form1_MouseDown(object sender,
System.Windows.Forms.MouseEventArgs e)
{ if(m_theRect.Contains(e.X,e.Y))
  { if(e.Clicks == 2)
```

```

        // a double-click
        m_theColor = Color.Green;
    else if(m_theColor == Color.RoyalBlue)
        m_theColor = Color.Red;
    else
        m_theColor = Color.RoyalBlue;
    Invalidate(m_theRect);
}
}
}

```

This program illustrates (by giving a bad example) the point that what happens on a double-click should be compatible with what happens on a single click. Although double-click turns the rectangle green, the first click already turns it red or blue, which is awkward.

### Which button was clicked?

The program we wrote above responds indiscriminately to a click of either the left or the right button. In MFC or the Windows API, you had to write a separate handler for the left or the right button. In FCL, there is only one *MouseDown* event, and you use the *Button* member of the *MouseEventArgs* class to determine which button was clicked. The type of that member is *MouseButtons*; this type has exactly six members:

*None, Left, Right, Middle, XButton1, XButton2.*

Some mice have a middle button; many mice nowadays have a “scroll wheel” which can also function as a middle button; and the Intellimouse Explorer has five buttons, so *XButton1* and *XButton2* can be used for that device.

To make our sample program respond only to the left mouse button, we can put this code at the beginning of our *MouseDown* handler:

```

if(e.Button != MouseButtons.Left)

```

```
return;
```