Chapter 1  
More Mouse

We have already seen how easy it is to use the mouse for simple left or right click processing. However, there are many more interesting facets to the use of the mouse. In this chapter we will explore the capabilities of the latest series of mice which have such features as a wheel and more than three buttons. We will also look at common mouse operations such as determining the object you are clicking on or dragging an object around the client area.

Mouse Buttons

A standard mouse usually has only two or possibly three buttons. Microsoft introduced the Intellimouse which has five buttons. The usual three buttons are on top and the additional two buttons are on the side. However, there is no reason that alternative mouse designs are not possible. In fact, trackballs are also popular and they use the same interface so that programs really aren’t aware of the exact physical configuration. As you have already seen, the MouseButtons enumeration is used to determine the button that was clicked. Table 7-1 show the complete enumeration which includes the two new buttons, XButton1 and XButton2.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>The left mouse button was pressed.</td>
</tr>
<tr>
<td>Middle</td>
<td>The middle mouse button was pressed.</td>
</tr>
<tr>
<td>None</td>
<td>No mouse button was pressed.</td>
</tr>
<tr>
<td>Right</td>
<td>The right mouse button was pressed.</td>
</tr>
<tr>
<td>XButton1</td>
<td>IntelliMouse</td>
</tr>
<tr>
<td>XButton2</td>
<td>IntelliMouse</td>
</tr>
</tbody>
</table>

When we single click the mouse we actually generate three mouse events as well as an additional higher level event, click. This is the order the mouse events are issued:

- MouseDown
- Click
- MouseClick
- MouseUp
Figure 7-1 shows a simple program to demonstrate the order these events are generated. If you run this program and click on any mouse button you will get the output of Figure 7-2. As you click the button down you will get the first event, the *MouseDown* event. When you let the button up the next three messages are generated in rapid succession in the order shown.

**MouseClick1 - Form1.cs**

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace MouseClick1
{
    public partial class Form1 : Form
    {
        private string s;
        public Form1()
        {
            InitializeComponent();
        }
        protected override void OnPaint(PaintEventArgs e)
        {
            e.Graphics.DrawString(s, Font, Brushes.Black, 10, 10);
        }
        private void Form1_MouseDown(object sender, MouseEventArgs e)
        {
            s += "MouseDown\n";
            Invalidate();
        }
        private void Form1_MouseClick(object sender, MouseEventArgs e)
        {
            s += "MouseClick\n";
            Invalidate();
        }
        private void Form1_MouseUp(object sender, MouseEventArgs e)
        {
            s += "MouseUp\n";
            Invalidate();
        }
        private void Form1_Click(object sender, EventArgs e)
        {
            s += "Click\n";
            Invalidate();
        }
        private void Form1_MouseDoubleClick(object sender, MouseEventArgs e)
        {
        }
    }
}
```
Double clicking the mouse generates two pairs of `MouseUp/Down` events, a `Click` event, a `MouseClick` event and a `MouseDoubleClick` event. The exact sequence is shown in Figure 7-3. The user can control the timing required to detect a double click through the control panel's mouse dialog. A user can also switch the left and right mouse buttons. This switch is invisible to programs. In other words, if the buttons are switched a left click generates events with the `MouseButton` enumeration indicating the right button.
The click event is not strictly a mouse event as it can also be triggered by other actions such as pressing the enter key when the associated control has focus. This action is only for some controls and even though your form is a control the click event is not generated when you hit enter.

**Hit Testing**

A task that confronts the programmer quite often is determining what graphic object the user is attempting to manipulate when the mouse is clicked in the client area. This may be a simple or very difficult problem to solve depending on the complexity of what is being displayed and how we want to allow it to be manipulated. In this section we will discuss only some simple concepts to get you going. Hit testing is the term we use for the steps required to associate the graphics object with the position in the client area that corresponds to the position the mouse is clicked.

When we click the mouse the position returned in the event arguments corresponds to the hot spot at the tip of the mouse cursor. Different cursors have slightly different positions used as the hot spot. This position is always in device units. In other words, regardless of any transformations we may apply when outputting graphical information, the coordinate is in pixels relative to the upper left hand corner of the client area. We can convert this coordinate to our logical coordinate system quite easily if necessary. The program shown in Figure 7-4 provides a graphical interface for a game of Tic-Tac-Toe. This version does not include an algorithm that allows the computer to play against you, but
rather it allows you to add the O's and X's by left or right clicking the mouse. The middle button is used to clear a cell. A sample of the output is shown in Figure 7-5.

```csharp
namespace HitTest1
{
    public partial class Form1 : Form
    {
        //dimensions
        private const float clientSize = 100;
        private const float lineLength = 80;
        private const float block = lineLength / 3;
        private const float offset = 10;
        private const float delta = 5;

        private enum CellSelection { N, O, X };  
        private CellSelection[,] grid = new CellSelection[3,3];

        public float scale;     //current scale factor

        public Form1()
        {
            InitializeComponent();
            ResizeRedraw = true;
        }
        protected override void OnPaint(PaintEventArgs e)
        {
            Graphics g = e.Graphics;
            ApplyTransform(g);

            //draw board
            g.DrawLine(Pens.Black, block, 0, block, lineLength);
            g.DrawLine(Pens.Black, 2*block, 0, 2*block, lineLength);
            g.DrawLine(Pens.Black, 0, block, lineLength, block);
            g.DrawLine(Pens.Black, 0, 2*block, lineLength, 2*block);

            for (int i = 0; i < 3; ++i)
                for (int j = 0; j < 3; ++j)
                    if (grid[i, j] == CellSelection.O) DrawO(i, j, g);
                    else if (grid[i, j] == CellSelection.X) DrawX(i, j, g);
        }
    }
}
```
The two dimensional array `grid` is used to store the state of each cell on the board. The values represent the three possible states of the cell. The enumeration `CellSelection` has the values shown in Table 7-2.
<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Empty cell, nothing displayed</td>
</tr>
<tr>
<td>O</td>
<td>An O is displayed</td>
</tr>
<tr>
<td>X</td>
<td>An X is displayed</td>
</tr>
</tbody>
</table>

The `OnPaint` method exploits many of the graphics capabilities we covered in previous chapters. The client area can be resized and the playing board scales accordingly. A world coordinate system 100 by 100 units is the basis of the graphics displayed. The actual application of the transformations is done in the method `ApplyTransform` which is called by `OnPaint` as well as the `MouseDown` event handler. The transformation parameters are controlled by the constants listed in Table 7-3.

**Table 1-3 - Transformation Constants**

<table>
<thead>
<tr>
<th>Constant</th>
<th>Determines</th>
</tr>
</thead>
<tbody>
<tr>
<td>clientSize</td>
<td>The size of the world coordinate space</td>
</tr>
<tr>
<td>lineLength</td>
<td>The length of the grid lines</td>
</tr>
<tr>
<td>block</td>
<td>The width and height of a cell (lineLength/3)</td>
</tr>
<tr>
<td>offset</td>
<td>The position of the upper left hand corner of the playing board</td>
</tr>
<tr>
<td>delta</td>
<td>The inside border of the X or O</td>
</tr>
</tbody>
</table>

`ApplyTransform` uses the techniques `ScaleTransform` and `TranslateTransform` using these values. `OnPaint` continues by drawing the four lines of the grid. The O's and X's are actually drawn by two methods, `DrawO` and `DrawX`. Two nested `for` loops are used to iterate through all nine board positions. `DrawO` and `DrawX` are straightforward although the calculation of the coordinate positions requires careful study, especially the use of the constant `delta` to reduce the size of the X or O to include a border. If we don’t allow for this then the X or O would totally fill the cell. It is important to remember that we are drawing in world coordinates where the constant `clientSize` is both the width and height of the client area.

The real action takes place in the `MouseDown` event handler. You will recall that all mouse positions are reported in device coordinates. In order to determine which cell, if any, the user has clicked in we need to determine the corresponding coordinate in the world coordinate space. This essentially means reversing the transformation. In other words we need to transform a point in device units to world units. In order to do this we need to create a `Graphics` object for our form and then apply the same transforms we use for the painting process. We can create a `Graphics` object with a call to `CreateGraphics` which is a method inherited from the `Form` class. Now you know why I placed the statements necessary to transform the `Graphics` object in a separate method.
To transform a coordinate in device units to world units the following statements are used:

```csharp
PointF[] p = { new Point(e.X, e.Y) };
g.TransformPoints(CoordinateSpace.World, CoordinateSpace.Device, p);
```

The `TransformPoints` method requires an array of points rather than a single point. The first statement sets up such an array with just one point in it. The result of the call to `TransformPoints` is that the resulting point in the array now corresponds to the position of the mouse click in the world coordinate system. All we need to do now is to determine which cell we are clicking in, if any,

The origin of the world coordinate system is not the same as the origin of the client area. We offset it by `offset` units. This means that if we click near the top or left edge of the client area we will have a coordinate that has a component that is less than zero. The first test we need to make is to ignore all mouse clicks with an X or Y value less than zero. These statements determine the cell we are clicking in:

```csharp
int i = (int)(p[0].X/block);
int j = (int)(p[0].Y/block);
```

If we click near the bottom or right edge of the client area the value we get is a non-existent cell. A simple test of the resulting `i` and `j` values eliminate these invalid clicks. We could also have made the same test on the point itself, but we would then have to compute the value to test against. It's easier to test `i` and `j`. The final step is to merely set the cell to the value corresponding to the button pushed and invalidating the form to cause a repaint.
The MouseMove Event

When you move the mouse the Windows operating system generates an event. The spacing between the coordinates reported by the MouseMove event depends on how fast you move the mouse and the speed of your system. You can't turn off mouse movement events, but you can just ignore them. Figure 7-6 shows a simple program that reports the current coordinate of the mouse in the upper left hand corner of the client area. The output is shown in Figure 7-7.

```csharp
namespace MouseMove1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        protected override void OnPaint(PaintEventArgs e)
        {
            Graphics g = e.Graphics;
        }
    }
}
```
Dragging an Object

A common task in graphical applications is to drag a graphic object around the client area. This is not as difficult as you might assume. The first task is to identify the object to be dragged. We do that using the hit testing techniques we discussed above or more advanced techniques is necessary. The next step is to determine the next position for the object. As you may have guessed, MouseMove events are used.

Two alternatives are possible while we are dragging the object:

1. Display the object in the new position continuously as we drag it.
2. Display the object only when we let up on the mouse button.

The first choice is what a user normally expects, but for complex objects it may result in excessive flicker. The first choice is normally used for graphics
application while the second choice is more appropriate when we are dragging text. For example, highlight some text in Microsoft Word and then hold down the left mouse button and drag the text to the new position. You will notice that the text doesn’t move until the mouse button is released. However, the cursor shape changes to indicate to the user that the text is being moved.

Figure 7-8 shows a program that displays a filled rectangle and allows it to be dragged around the client area by holding down the left button while within the boundaries of the rectangle.

```
DragRect1 - Form1.cs
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace DragRect1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        protected override void OnPaint(PaintEventArgs e)
        {
            Graphics g = e.Graphics;
            g.FillRectangle(Brushes.Red, rect);
        }
        protected override void OnMouseDown(MouseEventArgs e)
        {
            if (e.Button == MouseButtons.Left)
            {
                Point p = new Point(e.X, e.Y);
                if (rect.Contains(p))
                {
                    dragging = true;
                    offset = (Size)(rect.Location - new Size(e.X, e.Y));
                    Cursor = Cursors.Cross;
                }
            }
        }
        protected override void OnMouseUp(MouseEventArgs e)
        {
            if (e.Button == MouseButtons.Left)
            {
                dragging = false;
                Cursor = Cursors.Default;
            }
        }
    }
```
protected override void OnMouseMove(MouseEventArgs e)
{
    if (dragging)
    {
        Invalidate(rect);  //old position
        rect.Location = new Point(e.X, e.Y) + offset;
        Invalidate(rect);  //new position
    }
}

private Rectangle rect = new Rectangle(0, 0, 100, 100);
private bool dragging = false;
private Size offset;

Our first job is to hit test on the rectangle. If we know its current location we could easily check the X and Y coordinates of the mouse click to make sure they are within the X and Y bounds of the rectangle. This would require four comparisons. But hold on. There is an easier way. The Rectangle structure includes a method that does this for us. We merely call the Contains method passing it a Point object set to the coordinate of the mouse click.

The field rect is initialized to a new Rectangle object representing the width and height of our filled rectangle and a starting position in the upper left hand corner of the client area. The OnMouseDown method override is used to process the MouseDown event rather than using an event handler. This really is an arbitrary decision in the case of mouse events. The methods in the base class, Form, actually do nothing at all and are merely placeholders.

If the call to Contains returns true then we set dragging to true for future reference. The next step is to calculate the offset of the mouse click inside the rectangle. This is important since without this value we can't determine where the upper left hand corner of the rectangle should be after it is moved. We save this value in offset. This value is saved as a Size object. This makes calculations later on easier. This makes sense since the offset is not a Point but rather the difference between two points. This will actually be in a negative or left and up direction. This is the statement that does the job:

offset = (Size)(rect.Location - new Size(e.X, e.Y));

The final step is to set the cursor to a cross. The Cursors enumeration is used to set the Cursor property of the form. Figure 7-9 shows what things would look like with the left button held down.
The `OnMouseMove` method handles the `MouseMove` event. The primary purpose is to change the coordinate of the upper left hand corner of the rectangle. This is done with the following statement:

```csharp
rect.Location = new Point(e.X, e.Y) + offset;
```

Notice that all we have to do is to add the offset value we saved earlier to the current mouse position. We also need to ensure that the rectangle is redrawn in its new position if we wish to display the rectangle continuously as it is dragged. We could do this with a single call to `Invalidate`. However, that means that the entire client area would need to be redrawn. For this example it wouldn't make a lot of difference, but for complex graphics it might take time for the graphics memory to be updated on the display interface. As an alternative we can invalidate only the area that we know needs to be updated. This can be done with two calls to `Invalidate`. We merely pass the current and then new rectangle as the argument. Windows will then clip the output to the areas of the client area that do not need to be updated.

If we comment out the first call to `Invalidate` and drag the rectangle we get the effect shown in Figure 7-10. This results because the pixels occupied by the previous position of the rectangle are not erased and set to the background color of the form. You can also get a feel for the distance between `MouseMove` events by examining the output. If we comment out both calls to `Invalidate` then it appears that we can't drag the rectangle. Actually we are dragging the rectangle, but it is never repainted. In order to see the rectangle in its new position we would need to cause a repaint by minimizing and then restoring the form.
The OnMouseUp method is the easiest. We merely set dragging back to false and restore the cursor to the default value Cursors.Default.

**Mouse Captures**

When you run the program shown in Figure 7-6 you will see that the tracking of the coordinate stops as soon as the mouse cursor leaves the boundaries of the client area of your form. This does not happen with the example shown in Figure 7-8 while the mouse button is held down. This is because .NET automatically cause a mouse capture during the period the mouse is held down. You can see that the mouse cursor remains the cross and that you can actually drag an object out of the visible client area.

There is an important reason why you want the mouse to be captured. The default behavior is that Windows sends mouse events to the window the mouse is over. If the mouse was not captured then we would lose a MouseUp event if we moved out of our form. The program would think the mouse button was still down when in reality is was not. On the flip side it would also be possible to have a spurious MouseUp event if we depress a mouse button while outside the form and let it up after we move inside.

If you have programmed in earlier Windows programming environments you may have noticed that you had to explicitly make a function call to capture the mouse. Fortunately that is no longer required.

*The Wheel*
The wheel is a recent additional to mouse capabilities. It can be used for a variety of purposes. A number of controls have a built-in facility to use the wheel already and you don’t have to do any programming at all. If a control derives from ScrollableControl and the VScroll property is set to true, the mouse wheel scrolls through the control vertically. If VScroll is set to false and the HScroll property is set to true, the mouse wheel scrolls through horizontally. If you want to use the capabilities of the wheel in your programs in other ways keep reading.

Unlike the mouse position itself, the wheel is a relative device and there is no absolute positioning available. We can go in one direction or another. The concept of a detent is defined for the wheel even if the wheel itself does not exhibit this behavior. Some mice have wheels with a distinct clicking action and others rotate freely. The MouseWheel event is triggered when the mouse wheel is moved in either direction. The Delta property of MouseEventArgs provides the number of detents the wheel has rotated over. Unfortunately the value returned bears little correlation with reality. The value is either +120 or -120 unless you rotate the wheel extremely rapidly. In that case it is a multiple of 120. The best way to use the wheel is to merely detect if it is greater or less than zero and either increment or decrement the value being controlled.

Figure 7-11 shows a simple program that changes the shade of the background color of the form in response to the rotation of the mouse wheel. For each MouseWheel event the direction of the rotation is determined and the shade is adjusted up or down. The range of these values is constrained to be between 0, black, and 255, white. We create the appropriate shade of gray using the static method Color.FromArgb. The paint event handler merely displays the value of shade. Of course you won’t be able to see this if you make the background too dark. Figure 7-12 shows a sample of the output.

```csharp
MouseWheel - Form1.cs
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
namespace MouseWheel
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            BackColor = Color.FromArgb(128, 128, 128);
        }
        protected override void OnPaint(PaintEventArgs e)
        {
```
```csharp
protected override void OnMouseWheel MouseEventArgs(e)
{
    int i = e.Delta;
    if (i > 0) ++shade;
    else --shade;
    if (shade < 0) shade = 0;
    if (shade > 255) shade = 255;
    BackColor = Color.FromArgb(shade, shade, shade);
    Invalidate();
}
private int shade = 128;
```