6

GraphicsPaths and Regions

Back in the early chapters of this book, we used pens and brushes (in conjunction with the `Draw…` and `Fill…` methods of the `Graphics` object) to create abstract regular shapes on the page. We drew and filled lines, rectangles, regular curves, and ellipses.

In this chapter we'll take a look at two classes that take this a significant step further: the `GraphicsPath` and the `Region`. These classes enable us to work with more irregular shapes, and to group sets of shapes together for more effective processing.

So, in this short chapter we will cover:

- How we can use the `GraphicsPath` class to describe paths in GDI+
- How we can describe regions, using the `Region` class
- The similarities, differences, and relationships between `GraphicsPaths` and `Regions`

This chapter covers the basic features of `GraphicsPaths` and `Regions` – specifically, we focus on how we use these classes to compose outlines and shapes and perform some simple operations with them. In the next chapter, we'll see how these objects can be used in operations such as clipping, invalidation, and creating custom gradient brushes, to create some very useful effects.

What are GraphicsPaths and Regions?

The `GraphicsPath` and `Region` are two classes that contribute significantly to the power of GDI+ as a graphics-programming interface. These two classes are intimately related, as we'll see in this section, and yet quite distinct in their purposes and capabilities. Together, they allow us to build higher levels of abstraction, and thereby create simpler, cleaner code.

A **GraphicsPath** object represents a set of sub-paths or **figures**. Each figure is a connected path, composed of a series of line and curve segments and geometric figures. These segments and geometric figures can be described in any of the standard ways we've seen already in this book – we can use arcs, Bézier curves, ellipses, rectangles, and so on. To create a figure, we simply list the components of the figure in the right order – the `GraphicsPath` object computes the figure to be the outline that results when these component outlines are joined together in the order specified. (We'll see some examples of this in the next section.)

The `GraphicsPath` itself is the path composed of the ordered set of figures. Because the set of figures is ordered, and because each figure is composed of an ordered set of lines and curves, the `GraphicsPath` itself is a path with a beginning and an end.

The figures within a `GraphicsPath` object don't need to be connected to one another – they can be disconnected. In particular, this means that the figures don't have to be top-to-tail on the drawing surface: in fact, usually, they're *not* top-to-tail and thus the path described by the `GraphicsPath` object is discontinuous. To follow such a path from one end to the other, we would need to 'leap through space' from the end of one figure to the beginning of the next.

The **Region** class is similar to the `GraphicsPath` path class, but it relates to areas (or regions) instead of paths. Thus, we can use a `Region` object to represent a shape, or a set of shapes. A region can consist of geometric shapes (like rectangles and ellipses) and/or custom shapes whose outlines are composed of whatever lines and curves we care to specify. Like a `GraphicsPath`, the shapes that make up a `Region` don't necessarily have to be connected.

Once we've created and defined a `GraphicsPath` object, we can draw it to our drawing surface using a `Pen` object. After constructing a `Region` object, we can fill the region using a `Brush` object. In both cases, there's an important point here. We can use the `GraphicsPath` and `Region` classes to compose complex paths and shapes; but the task of rendering the completed path or shape to the screen can be achieved *in a single method call*. This makes the rendering process much tidier, and also allows us to organize our code more tidily.

Let's start to look at some real `GraphicsPath`s and `Region`s, so we can get a better idea of how they work.

The GraphicsPath Class

The `GraphicsPath` class encapsulates a connected series of lines and curves. You may recall that we used a `GraphicsPath` object back in Chapter 3, to demonstrate the effect created by a `PathGradientBrush`.

Essentially, we use a `GraphicsPath` object when we want to describe an outline. Once we've described the outline, we can do a number of things with it: we can draw it using a pen, or fill its interior using a brush, or use it to create clipping regions.

We'll see all these applications of our outline during this chapter and the next. Before we get to that, we should look at some simpler examples and get the basics under our belt.

Composing a GraphicsPath

As we noted earlier in the section, before we can do anything with a `GraphicsPath` path, we need to compose it. A `GraphicsPath` is composed of an ordered set of figures; and each figure is composed of an ordered set of line and curve segments, and geometric figures.

Once we've created a new `GraphicsPath` object, we compose the `GraphicsPath` path segment by segment, and figure by figure. The order in which we specify the segments and figures defines the order and direction of the path – and hence the start and end points of the path. Let's look at an example:

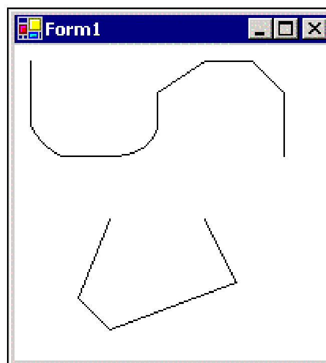
```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, this.ClientRectangle);

    // Create a new GraphicsPath object
    GraphicsPath gp = new GraphicsPath();

    // Create a figure
    gp.AddLine(10, 10, 10, 50);
    gp.AddBezier(10, 50, 10, 55, 25, 70, 30, 70);
    gp.AddLine(30, 70, 60, 70);
    gp.AddBezier(60, 70, 85, 70, 90, 55, 90, 50);
    gp.AddLine(90, 50, 90, 30);
    gp.AddLine(90, 30, 120, 10);
    gp.AddLine(120, 10, 150, 10);
    gp.AddLine(150, 10, 170, 30);
    gp.AddLine(170, 30, 170, 70);

    // Create another figure
    gp.StartFigure();
    gp.AddLine(60, 110, 40, 160);
    gp.AddLine(40, 160, 60, 180);
    gp.AddLine(60, 180, 140, 150);
    gp.AddLine(140, 150, 120, 110);

    // draw the path
    g.DrawPath(Pens.Black, gp);

    // Clean up
    gp.Dispose();
}
```

If you're trying these examples for yourself, remember that the `GraphicsPath` class is part of the `System.Drawing.Drawing2D` namespace, so you'll need a `using` directive to include it.

In this code we've created a path that contains two figures. We've composed the first figure by adding a series of line and Bézier curve segments to the `GraphicsPath` object, using the `AddLine` and `AddBezier` methods. It begins with a line-segment from the point (10, 10) to (10, 50), and then continues with a Bézier curve round to (30, 70), followed by another line segment, another Bézier curve, and then five more line segments – ending on the point (170, 70).

When we call the `StartFigure` method, we begin a second figure. By definition, this means that we have finished composing the first figure. As you can see, the second figure is composed of four lines, beginning at (60, 110) and ending at (120, 110).

The whole path looks like this. Its start point is the point (10, 10), and its end point is the point (120, 110). It's not a continuous line – in order to follow it from one end to the other we must jump from the end of the first figure to the start of the second.



Note that the `GraphicsPath` class itself doesn't provide any functionality for **rendering** the path we've created. If we want to render the path contained in a `GraphicsPath` object, then we call a method in the `Graphics` class – and pass the `GraphicsPath` object as an argument. (In fact, as we'll see later, the `Region` object is very similar in this respect.)

Drawing Multiple Figures in a GraphicsPath

In the example above, note how we composed the two figures. To begin describing the first figure, we just launched into a sequence of `Add…` methods, which sequentially build up the first figure. We must finish describing the first figure before we begin describing the second figure – because as soon as we call `StartFigure` to begin describing the second figure, GDI+ takes it as read that you've *finished* describing the first figure. There's no going back to it!

We can call `StartFigure` as often as we like. Each time we call it, it will end the existing figure and begin a new one.

Open Figures and Closed Figures

A figure is **closed** if its start point and end point are the same. If a figure is not closed, then we say that it's **open**. (It's just like a pen – if you don't close the pen, the sheep can get out.) In the example above, the first figure starts at (10, 10) and ends at (170, 70), and is therefore open. The second figure starts at (60, 110) and ends at (120, 110), and so is also open.

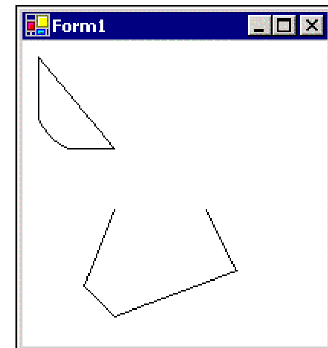
There are a number of ways we can close a figure. The obvious way is explicitly to add a line or curve segment at the end of the figure, whose end point is the same as the start point of the figure.

An easier way is to use the `CloseFigure` method when we've finished describing the figure, like this:

```
// Create a new GraphicsPath object
GraphicsPath gp = new GraphicsPath();

// Create a figure
gp.AddLine(10, 10, 10, 50);
gp.AddBezier(10, 50, 10, 55, 25, 70, 30, 70);
gp.AddLine(30, 70, 60, 70);
// Close this figure
gp.CloseFigure();
```

```
// Create another figure
gp.StartFigure();
gp.AddLine(60, 110, 40, 160);
gp.AddLine(40, 160, 60, 180);
gp.AddLine(60, 180, 140, 150);
gp.AddLine(140, 150, 120, 110);

// draw the path
g.DrawPath(Pens.Black, gp);

// Clean up
gp.Dispose();
```

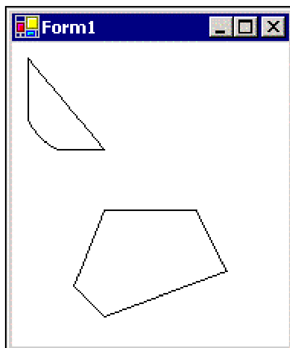
Now, the first figure is composed of a line from (10, 10) to (10, 50); a Bézier curve to (30, 70); a line to (60, 70); and a line that *closes* the figure by joining this last point back to the first. In fact, the `CloseFigure` method not only closes the existing figure; it also starts the next figure automatically, so that the call to `StartFigure` here (for the second figure) is in fact superfluous.



Alternatively, we can call the `CloseAllFigures` method, which closes all the figures in the path upto and including the current one:

```
// Create a new GraphicsPath object
GraphicsPath gp = new GraphicsPath();

// Create a figure
gp.AddLine(10, 10, 10, 50);
gp.AddBezier(10, 50, 10, 55, 25, 70, 30, 70);
gp.AddLine(30, 70, 60, 70);

// Create another figure
gp.StartFigure();
gp.AddLine(60, 110, 40, 160);
gp.AddLine(40, 160, 60, 180);
gp.AddLine(60, 180, 140, 150);
gp.AddLine(140, 150, 120, 110);

// Close all figures
gp.CloseAllFigures();

// draw the path
g.DrawPath(Pens.Black, gp);

// Clean up
gp.Dispose();
```

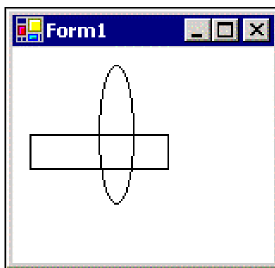
This code creates two open figures, and then closes them both off at the same time. Finally, it draws the two closed figures to the drawing surface.



We can also add geometric figures to the path. By nature, these figures are already closed, so the `CloseFigure` and `CloseAllFigures` methods have no effect on these figures. For example:

```
// Create a new GraphicsPath object
GraphicsPath gp = new GraphicsPath();

// Create a figure
gp.AddRectangle(new Rectangle(10, 50, 80, 20));

// Create another figure
gp.AddEllipse(50, 10, 20, 80);

// draw the path
g.DrawPath(Pens.Black, gp);

// Clean up
gp.Dispose();
```

In this example, the first figure is a rectangle and the second is an ellipse. Note that there's no need to use `StartFigure` to end the first figure and start the second, because the first figure is already closed (and therefore complete).



In short: if the first and last points of a figure are different, and we don't call the `CloseFigure` method for the figure, and we don't call the `CloseAllFigures` method, then the figure will be open.

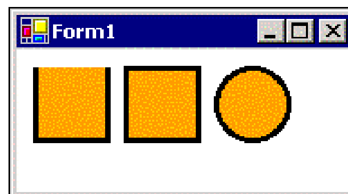
Filling the Area Bounded by a GraphicsPath

As we'll see a little later, we can use a `GraphicsPath` as the boundary of a `Region` – but only if all the figures in the `GraphicsPath` are closed.

Moreover, we can apply a `Brush` object to fill the area bounded by our `GraphicsPath` object, like this:

```
GraphicsPath gp = new GraphicsPath();

// create an open figure
gp.AddLine(10, 10, 10, 50);
gp.AddLine(10, 50, 50, 50);
gp.AddLine(50, 50, 50, 10);

// start a new figure
gp.StartFigure();
gp.AddLine(60, 10, 60, 50);
gp.AddLine(60, 50, 100, 50);
gp.AddLine(100, 50, 100, 10);
gp.CloseFigure();

// add a geometric shape (a rectangle) to the path
Rectangle r = new Rectangle(110, 10, 40, 40);
gp.AddEllipse(r);

// Fill the area 'bounded' by the path
g.FillPath(Brushes.Orange, gp);

// Draw the path
g.DrawPath(Pens.Black, gp);

// Clean up
gp.Dispose();
```

Of course, it doesn't make sense to fill an open area, like the one we've created here. In this case, the `FillPath` method works out the closed path that we would get if we called `CloseAllFigures` on our `GraphicsPath`. Then it uses the specified `Brush` to fill the area bounded by that closed path:



As you can see, GDI+ doesn't actually call `CloseAllFigures` on the `GraphicsPath` itself. It calls `CloseAllFigures` on an implicit copy of our `GraphicsPath`, and uses *that* path to perform the filling operation.

Absolute Coordinates and a Shorthand for Straight-line Figures

Note that the arguments to the `GraphicsPath.AddLine` method (and other `Add`… methods) are not relative – they represent the ***absolute*** coordinates of the start point and end point. Thus, the operation `gp.AddLine(50, 0, 0, 20)` will add a line that runs between the points (50, 0) and (0, 20) – regardless of what we've added to the figure so far.

This raises the interesting question of what happens when consecutive `Add`… method calls describe line or curve fragments that don't top-and-tail one another. For example, in the first example we created a 9-segment figure like this:

```
// Create a figure
gp.AddLine(10, 10, 10, 50);
gp.AddBezier(10, 50, 10, 55, 25, 70, 30, 70);
gp.AddLine(30, 70, 60, 70);
gp.AddBezier(60, 70, 85, 70, 90, 55, 90, 50);
gp.AddLine(90, 50, 90, 30);
gp.AddLine(90, 30, 120, 10);
gp.AddLine(120, 10, 150, 10);
gp.AddLine(150, 10, 170, 30);
gp.AddLine(170, 30, 170, 70);
```

Here, we've carefully specified the arguments of these `Add`… methods so that each one runs on from the previous one – so that segments *n* and *n*+1 meet at (10, 50), (30, 70), (60, 70), (90, 50), (90, 30), (120, 10), etc. What happens if we're not so careful?

In fact, if the end of one segment is not the same as the beginning of the next segment, then GDI+ assumes that you want to join them with a straight line segment. What this means is that, if we wish, we can specify figures with straight-line segments by omitting some of the `AddLine` operations (just so long as we don't omit any of the coordinate information).

For example, we can describe the above figure with three fewer `AddLine` operations than we've shown above:

```
    // Create a figure
    gp.AddLine(10, 10, 10, 50);
    gp.AddBezier(10, 50, 10, 55, 25, 70, 30, 70);
//  gp.AddLine(30, 70, 60, 70);
    gp.AddBezier(60, 70, 85, 70, 90, 55, 90, 50);
    gp.AddLine(90, 50, 90, 30);
//  gp.AddLine(90, 30, 120, 10);
    gp.AddLine(120, 10, 150, 10);
//  gp.AddLine(150, 10, 170, 30);
    gp.AddLine(170, 30, 170, 70);
```

GraphicsPath Properties and Methods

There are a number of methods and properties that give us quite a bit of power when working with paths. The following table gives an overview of the public **properties** of the `GraphicsPath` class, of which there are five:

Property	Description
FillMode	If we're filling the interior of the path, then this property dictates how it will be filled. Its values come from the FillMode enumeration (which has two values – Alternate and Winding). We can get or set the value of this property. See the additional note below.
PathData	This gets a PathData object that contains information on how the path is constructed. (The PathData object has two properties – Points and Types – which give the same arrays as those described in the PathPoints and PathTypes properties in this table.)
PathPoints	This gets the points in the path, in the form of an array.
PathTypes	Each point in a GraphicsPath has a type (for example, Start, Line, Bezier, CloseSubpath). This property gets the types of the points in the path, in the form of an array. Note that all the possible point types are listed in the PathPointType enumeration, which is part of the System.Drawing.Drawing2D enumeration.
PointCount	This gets the number of elements in the arrays returned by the PathPoints or PathTypes properties.

As we state in the table above, there are two possible values for the FillMode property – Alternate, and Winding. The exact way that these fill modes fill the interior of a path can depend on the orientation of the path – that is, the start and end points of the path and the resulting direction of the path.

For the purposes of writing custom controls, the use of intricate fill patterns is likely to be minimal, so we won't dwell on the details of this point here. However, it's worth noting that the fill behavior of some brushes can depend on the orientation of the path – and therefore can depend on the order in which you describe the individual line and curve segments and the individual figures.

Most of the **methods** in the GraphicsPath class facilitate construction of our path. These methods, which are all prefixed with the word Add…, allow us to add things like lines, curves, and geometrical figures to the path. We've already covered this process in some detail, so we will not tackle it further here, except to say that the full list of these methods includes:

AddArc	AddClosedCurve	AddLine	AddPie	AddRectangles
AddBezier	AddCurve	AddLines	AddPolygon	AddString
AddBeziers	AddEllipse	AddPath	AddRectangle	

The following table lists some of the other methods of the GraphicsPath class. This table is not intended to be a reference, but rather is a guide to the different ways that we can construct a path.

Method	Description
`ClearMarkers`	See below.
`Clone`	Creates a `GraphicsPath` object that contains a copy of the current path.
`CloseFigure`	Closes the current figure by adding a line between the ending point of the last segment and the starting point of the first segment. Automatically starts a new figure in the path.
`CloseAllFigures`	Closes all figures drawn so far, by adding lines between each figure's end point and its start point. Automatically starts a new figure in the path.
`Flatten`	Converts curves into a series of connected line segments.
`GetBounds`	Returns a rectangle that encloses the path.
`IsOutlineVisible`	Determines whether a particular point is contained in the outline of the path. Could be used for hit testing.
`IsVisible`	Determines whether a particular point is contained in the interior of the path.
`Reset`	Clears the path.
`Reverse`	Reverses the order of the points in the path.
`SetMarkers`	See below.
`StartFigure`	Starts a new figure in the path. Leaves the previous figure open.
`Transform`	Applies a transform matrix to the path.
`Warp`	Applies a warp transform to the path. A warp transform is defined by a rectangle and a parallelogram.

The `SetMarkers` and `ClearMarkers` methods relate (unsurprisingly) to a feature of GDI+ `GraphicsPaths`, called the ***marker****. We can use `SetMarkers` to insert a marker at a specific point in a path as we are constructing a path. Later, we can create a new path that consists of a* subset *of the original path that lies between two markers in that path. We can use the `ClearMarkers` method to clear any markers from an existing path. We use markers to separate groups of sub-paths.*

Finally, it's worth noting that the `GraphicsPath` object constructor is overloaded to accept an array of points, and/or an array of point-types, and/or a fill mode. If our `GraphicsPath` isn't very complicated, this approach works well. See the documentation at http://msdn.microsoft.com/library for more information.

The Region Class

As we've said, we can use a Region object to describe a region of our surface – that is, zero or more areas of real estate on our surface. When we work with a Region object, we generally begin by defining what area the Region covers, and then performing operations that use the defined region. (In this way, at least, we use a Region object in much the same way as we'd use a GraphicsPath object.) Once we've defined our region, we can perform a number of operations: we can fill it with a Brush object, or use it for hit testing or as a clipping region (see the next chapter).

It makes sense to begin by seeing how we might define a Region. Unlike the GraphicsPath class, the Region class doesn't have an army of Add… methods. In fact, while a GraphicsPath object has its figures, there is no equivalent concept in a Region object – while a Region object typically does consist of a number of disconnected enclosed areas, they're created in a very different way. In particular, we don't build up a Region incrementally, like we did a GraphicsPath.

Instead, we usually create a Region by using one of the overloaded Region class constructors. We'll look at some ways to create Regions here.

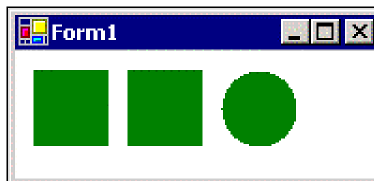
Creating a Region from a GraphicsPath

One way to create a Region is to use a GraphicsPath to define a path, and then use this path as the outline of our new Region object.

In this example, we'll use a familiar GraphicsPath object (one from earlier in the chapter) to create a region, and then we'll fill the resulting region with a Brush:

```
// Create a GraphicsPath
GraphicsPath gp = new GraphicsPath();

// create an open figure
gp.AddLine(10, 10, 10, 50);
gp.AddLine(10, 50, 50, 50);
gp.AddLine(50, 50, 50, 10);

// start a new figure
gp.StartFigure();
gp.AddLine(60, 10, 60, 50);
gp.AddLine(60, 50, 100, 50);
gp.AddLine(100, 50, 100, 10);
gp.CloseFigure();

// add a geometric shape (a rectangle) to the path
Rectangle r = new Rectangle(110, 10, 40, 40);
gp.AddEllipse(r);

// Create a Region whose boundary is the above GraphicsPath
Region reg = new Region(gp);

// Fill the Region
g.FillRegion(Brushes.Green, reg);

// Clean up
reg.Dispose();
gp.Dispose();
```

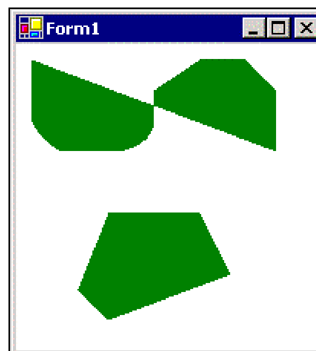
Here's the result. The shapes that we see are familiar, but the way we've filled them is different. The `Region` class constructor we've used here is one that accepts a `GraphicsPath` object; it takes a copy of the path, closes it, and then uses it to create a `Region`. As you can see, this region is composed of three disconnected areas: two 4-sided polygons and a circular ellipse.



So, `Regions` (like `GraphicsPaths`) don't need to be connected or contiguous. Moreover, `GraphicsPaths` give us a way to describe regions whose boundaries are irregular. For example:

```
// Create a GraphicsPath
GraphicsPath gp = new GraphicsPath();

// Create a figure
gp.AddLine(10, 10, 10, 50);
gp.AddBezier(10, 50, 10, 55, 25, 70, 30, 70);
gp.AddLine(30, 70, 60, 70);
gp.AddBezier(60, 70, 85, 70, 90, 55, 90, 50);
gp.AddLine(90, 50, 90, 30);
gp.AddLine(90, 30, 120, 10);
gp.AddLine(120, 10, 150, 10);
gp.AddLine(150, 10, 170, 30);
gp.AddLine(170, 30, 170, 70);

// Create another figure
gp.StartFigure();
gp.AddLine(60, 110, 40, 160);
gp.AddLine(40, 160, 60, 180);
gp.AddLine(60, 180, 140, 150);
gp.AddLine(140, 150, 120, 110);

// Create a Region whose boundary is the above GraphicsPath
Region reg = new Region(gp);

// Fill the Region
g.FillRegion(Brushes.Green, reg);

// Clean up
reg.Dispose();
gp.Dispose();
```

Here's another region whose outline is a familiar shape (it's outline is the `GraphicsPath` we drew at the beginning of this chapter). This time, the top figure is not entirely concave, so when GDI+ closes the figure in order to create the region, the resulting interior is in two pieces.

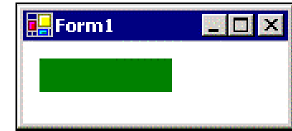


Creating a Region from a Rectangle

We can create a `Region` from a `Rectangle` object, in a very similar way:

```
// Create a Region whose boundary is the above Rectangle
Region reg = new Region(new Rectangle(10, 10, 80, 20));

// Fill the Region
g.FillRegion(Brushes.Green, reg);

// Clean up
reg.Dispose();
```

The result isn't very surprising; but it's handy to have an easy way to put together a rectangular `Region`, because many of the regions you're likely to need will be rectangular.



Creating a Region from Another Region

To create a `Region` from an existing region, the constructor doesn't accept the existing region object itself – instead it expects an array of data about the region. We can get the required data about such a `Region` from its `GetRegionData` method – this method returns the required data in the form of a `RegionData` object. We can then pass this object into the `Region` constructor, and hence create a new `Region` object just like the existing one:

```
Region r1 = new Region(new Rectangle(10, 10, 80, 20));
RegionData r1Data = r1.GetRegionData();
Region r2 = new Region(r1Data);
```

Alternatively, it's easier just to use the `Clone` method instead:

```
Region r1 = new Region(new Rectangle(10, 10, 80, 20));
Region r2 = r1.Clone();
```

Unions and Intersections of Regions

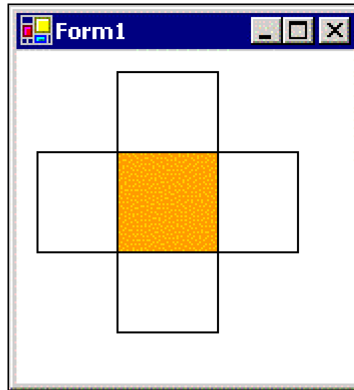
Once we've constructed a region, we can modify it. The `Region` object provides five methods that allow us to perform **set algebra** on the region. This involves specifying a second area (in the form of a `Rectangle`, a `RectangleF`, another `Region`, or the internal of a `GraphicsPath`), and performing an operation – this results in a new `Region` that is a subset of the two.

Here's an example. Here, we first set the `Region` object to represent the area inside the tall rectangle. Then we modify it – we reduce its size by taking only the intersection of the `Region` with the region described by `rect2`:

```
    // Create two rectangles
    Rectangle rect1 = new Rectangle(50, 10, 50, 130);
    Rectangle rect2 = new Rectangle(10, 50, 130, 50);
```

```
        // Set Region using first rectangle
        Region reg = new Region(rect1);

        // Modify Region to be intersection of itself with second rectangle
        reg.Intersect(rect2);

        // Draw the result so we can see it
        g.FillRegion(Brushes.Orange, reg);
        g.DrawRectangle(Pens.Black, rect1);
        g.DrawRectangle(Pens.Black, rect2);
```

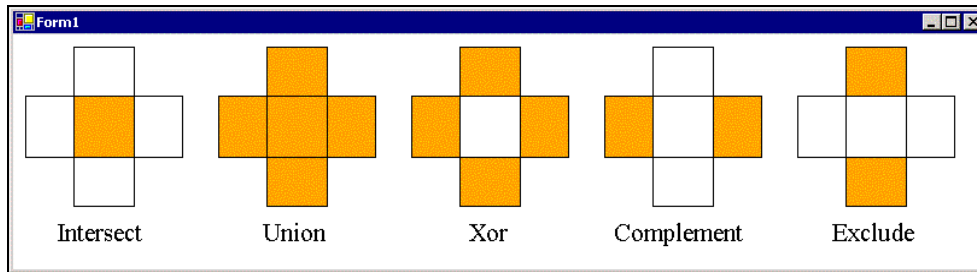
The resulting `Region` is shown in the shaded area here. The screenshot also shows the two rectangles.



All five methods work in a very similar way – they represent five different set algebra operations. The methods are listed below:

Operation	**Description**
`r1.Intersect(r2);`	Updates `r1` to be the intersection between `r1` and `r2` (that is, any portion that is in both `r1` and `r2`)
`r1.Union(r2);`	Updates `r1` to be the union of `r1` and `r2` (that is, any portion that is in either `r1` or `r2` or both)
`r1.Xor(r2);`	Updates `r1` to be the exclusive union of `r1` and `r2` (that is, any portion that is in either `r1` or `r2` but not both)
`r1.Complement(r2);`	Updates `r1` so that it *includes* areas that are contained within `r2`, but *excludes* any portion that was originally contained in `r1`
`r1.Exclude(r2);`	Updates `r1` to *exclude* any portion that is also contained within `r2`

Here's a screenshot that demonstrates all five. The code is quite similar to the code we've just seen, so we'll omit it here – though it is included within the support code for the book. In each of these, `r1` is the tall thin rectangle and `r2` is the short wide one:



Methods of the Region Class

There are a number of other methods that we can use to manipulate regions and query regions (and return information about them). Some of them are listed below. For a full list, see the documentation at http://msdn.microsoft.com/library. We will not cover them all in this chapter, but you will see some of them in use elsewhere in the book:

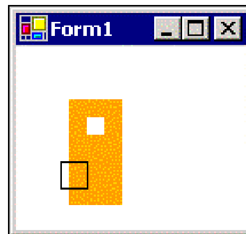
Method	Description
`Clone`	Returns a `Region` object that is a copy of the existing one.
`GetBounds`	Gets a `Rectangle` object that bounds this `Region` object on the specified drawing surface.
`GetRegionData`	Gets a `RegionData` object that contains data that defines the region. We can use the `RegionData` object to create a new `Region`.
`GetRegionScans`	Gets an array of `RectangleF` objects that approximate this `Region`.
`IsEmpty`	Returns a Boolean – `true` if the `Region` is empty on the specified drawing surface, `false` otherwise.
`IsInfinite`	Returns a Boolean – `true` if the `Region` has an infinite interior, `false` otherwise. See below.
`IsVisible`	Returns a Boolean – `true` if any part of the `Region` is visible when drawn on the drawing surface of the specified `Graphics` object; `false` otherwise. There are a number of overloads for this method, that test for visibility of an intersection of the `Region` with a given point or rectangle on the given drawing surface.
`MakeEmpty`	Initializes this `Region` so that its interior is empty.
`MakeInfinite`	Initializes this `Region` so that its interior is infinite. See below.
`Transform`	Applies a transform matrix to the `Region`.
`Translate`	Offsets the `Region` by an X and Y offset.

Let's finish off this section with a couple of examples that demonstrate some of these methods. We can illustrate the methods that return Booleans as follows:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, this.ClientRectangle);

    // Create a Region object, cut a rectangular hole in it, and fill it
    Region r = new Region(new Rectangle(30, 30, 30, 60));
    r.Exclude(new Rectangle(40, 40, 10, 10));
    g.FillRegion(Brushes.Orange, r);

    // Tell us about the Region
    Console.WriteLine("This Region: ");
    Console.WriteLine(r.IsInfinite(g) ? " - is infinite"
                                      : " - is finite");
    Console.WriteLine(r.IsEmpty(g) ? " - is empty"
                                   : " - is non-empty");

    PointF pf = new PointF(35.0f, 30.0f);
    Console.WriteLine((r.IsVisible(pf) ? " - includes"
                                       : " - excludes")
                      + " the point (35.0, 50.0)" );

    Rectangle rect = new Rectangle(25, 65, 15, 15);
    g.DrawRectangle(Pens.Black, rect);
    Console.WriteLine((r.IsVisible(rect) ? " - is visible"
                                         : " - is invisible")
                      + " in the rectangle shown" );

    r.Dispose();
}
```

When we run this, we get a list of output to the console window. We also see the region in the client window.

```
This Region:
 - is finite
 - is non-empty
 - includes the point (35.0, 50.0)
 - is visible in the rectangle shown
```



As you can see, the filled region is finite, in that it doesn't extend in all directions as we see it in the drawing surface. It's also non-empty – it contains at least one point. We've selected the point (35, 50) to see whether it's contained within the region – and the output tells us that it is. Finally, the output confirms that the region is visible through the rectangle shown – because the interior of the rectangle contains a part of the region.

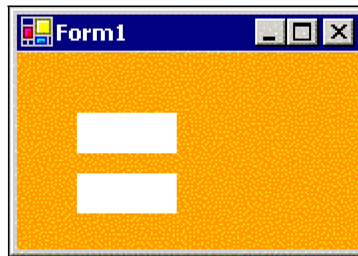
Infinite Regions

It's worth a note on infinite regions. It's sometimes useful to set a region to be equal to the entire drawing surface – extending infinitely in both directions. We then can use the set algebra operations (`Intersect`, `Union`, `Xor`, `Compliment`, and `Exclude`) to cut out bits of the infinite region and get our desired region.

This is useful, for example, we want to draw a background over the entire custom control, apart from certain rectangles or regions in which we want to draw other elements. In this case, making an infinite region (using the `MakeInfinite` method) and cutting rectangles out of it is a very convenient approach:

```
// Create a Region object and set it to be infinite
Region r = new Region();
r.MakeInfinite();

// Now cut a rectangular hole in it, and draw the result
r.Exclude(new Rectangle(30, 30, 50, 20));
r.Exclude(new Rectangle(30, 60, 50, 20));
g.FillRegion(Brushes.Orange, r);

r.Dispose();
```

Here, we've set the `Region` object to be the infinite space, before cutting out a couple of rectangles and filling what's left.



Summary

In this chapter, we covered the `GraphicsPath` and `Region` classes. A `GraphicsPath` object is used to represent a path. This path is composed of an ordered set of figures, and each figure is composed of an ordered set of line segments, curve segments, and geometric figures. To define the `GraphicsPath` object, we build up the path piece by piece. A `Region` object, by contrast, represents an area. We can't define a `Region` piece by piece – more commonly, we describe a `Region` by using set algebra operations (`Intersect`, `Union`, `Xor`, `Compliment`, and `Exclude`).

`GraphicsPaths` and `Regions` can both cope with being disconnected or discontinuous, and a `Region` can be infinite.

We saw how each of the figures in a GraphicsPath is either open or closed, and that we can call the CloseFigure method or CloseAllFigures method to close open figures. We can use a GraphicsPath as the outline of a Region – if the GraphicsPath contains any open figures, then GDI+ works out what the closed path would look like and then builds the Region from that.

We can draw a GraphicsPath's path using a Pen object, and we can fill a Region using a Brush object. (As a sort of shorthand, we can also fill the region whose boundary is a given GraphicsPath directly, by passing the GraphicsPath and a Brush to the Graphics.FillPath method.)

Both the GraphicsPath and Region classes have many utility methods and properties that give us a great deal of power and flexibility.

In the next chapter, we'll see how the GraphicsPath and Region classes can be very useful in the context of clipping. We're also going to take a look at invalidation.