10
Transformation

In chapter 9 we delved into advanced 2D graphics programming. In this chapter we will explore GDI+ transformations. A **transformation** is a process that changes graphics objects from one state to another. Rotation, scaling, reflection, translation, and shearing are some examples of transformation. Transformations can be applied not only to graphics shapes, curves, and images, but even to image colors.

In this chapter we will cover the following topics:

- The basics of transformation, including coordinate systems and matrices
- Global, local, and composite transformations
- Transformation functionality provided by the `Graphics` class
- Transformation concepts such as shearing, rotation, scaling, and translation
- The `Matrix` and `ColorMatrix` classes, and their role in transformation
- Matrix operations in image processing, including rotation, translation, shearing, and scaling
- Color transformation and recoloring
- Text transformation
- Composite transformations and the matrix order

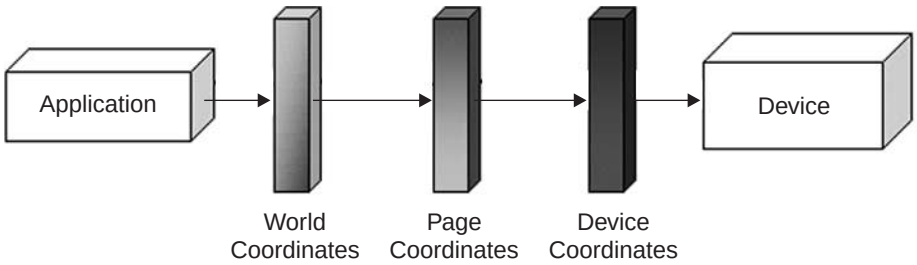

FIGURE 10.1: Steps in the transformation process

Any drawing process involves a source and a destination. The source of a drawing is the application that created it, and the destination is a display or printer device. For example, the process of drawing a simple rectangle starts with a command telling GDI+ to draw on the screen, followed by GDI+ iterating through multiple steps before it finally renders a rectangle on the screen. In the same way, transformation involves some steps before it actually renders the transformed object on a device. These steps are shown in Figure 10.1, which shows that GDI+ is responsible for converting world coordinates to page coordinates and device coordinates before it can render a transformed object.

10.1 Coordinate Systems

Before we discuss transformations, we need to understand coordinate systems. GDI+ defines three types of coordinate spaces: world, page, and device. When we ask GDI+ to draw a line from point A ($x1$, $y1$) to point B ($x2$, $y2$), these points are in the world coordinate system.

Before GDI+ draws a graphics shape on a surface, the shape goes through a few transformation stages (conversions). The first stage converts world coordinates to page coordinates. Page coordinates may or may not be the same as world coordinates, depending on the transformation. The process of converting world coordinates to page coordinates is called **world transformation**.

The second stage converts page coordinates to device coordinates. Device coordinates represent how a graphics shape will be displayed on a

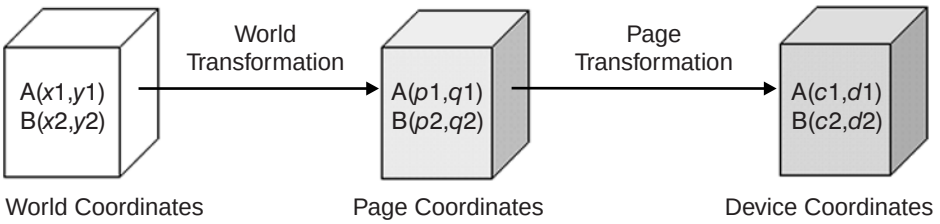

FIGURE 10.2: Transformation stages

device such as a monitor or printer. The process of converting page coordinates to device coordinates is called **page transformation**. Figure 10.2 shows the stages of conversion from world coordinates to device coordinates.

In GDI+, the default origin of all three coordinate systems is point (0, 0), which is at the upper left corner of the client area. When we draw a line from point A (0, 0) to point B (120, 80), the line starts 0 pixels from the upper left corner in the *x*-direction and 0 pixels from the upper left corner in the *y*-direction, and it will end 120 pixels over in the *x*-direction and 80 pixels down in the *y*-direction. The line from point A (0, 0) to point B (120, 80) is shown in Figure 10.3.

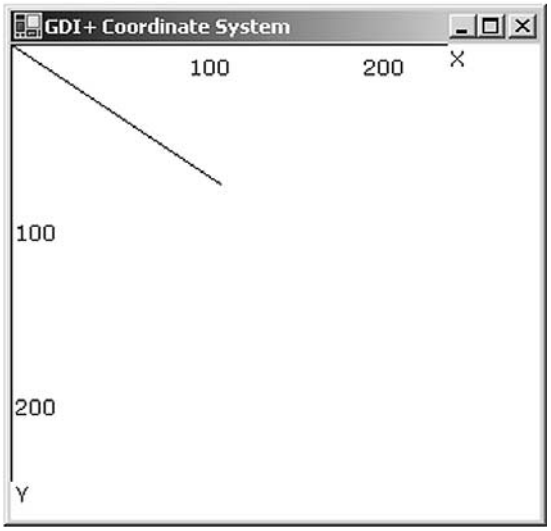

FIGURE 10.3: Drawing a line from point (0, 0) to point (120, 80)

Drawing this line programmatically is very simple. We must have a `Graphics` object associated with a surface (a form or a control). We can get a `Graphics` object in several ways. One way is to accept the implicit object provided by a form's paint event handler; another is to use the `CreateGraphics` method. Once we have a `Graphics` object, we call its draw and fill methods to draw and fill graphics objects. Listing 10.1 draws a line from starting point A (0, 0) to ending point B (120, 80). You can add this code to a form's paint event handler.

Listing 10.1: Drawing a line from point (0, 0) to point (120, 80)

```
Graphics g = e.Graphics;
Point A = new Point(0, 0);
Point B = new Point(120, 80);
g.DrawLine(Pens.Black, A, B);
```

Figure 10.3 shows the output from Listing 10.1. All three coordinate systems (world, page, and device) draw a line starting from point (0, 0) in the upper left corner of the client area to point (120, 80).

Now let's change to the page coordinate system. We draw a line from point A (0, 0) to point B (120, 80), but this time our origin is point (50, 40) instead of the upper left corner. We shift the page coordinates from point (0, 0) to point (50, 40). The `TranslateTransform` method of the `Graphics` class does this for us. We will discuss this method in more detail in the discussion that follows. For now, let's try the code in Listing 10.2.

Listing 10.2: Drawing a line from point (0, 0) to point (120, 80) with origin (50, 40)

```
Graphics g = e.Graphics;
g.TranslateTransform(50, 40);
Point A = new Point(0, 0);
Point B = new Point(120, 80);
g.DrawLine(Pens.Black, A, B);
```

Figure 10.4 shows the output from Listing 10.2. The page coordinate system now starts at point (50, 40), so the line starts at point (0, 0) and ends at point (120, 80). The world coordinates in this case are still (0, 0) and (120, 80), but the page and device coordinates are (50, 40) and (170, 120). The

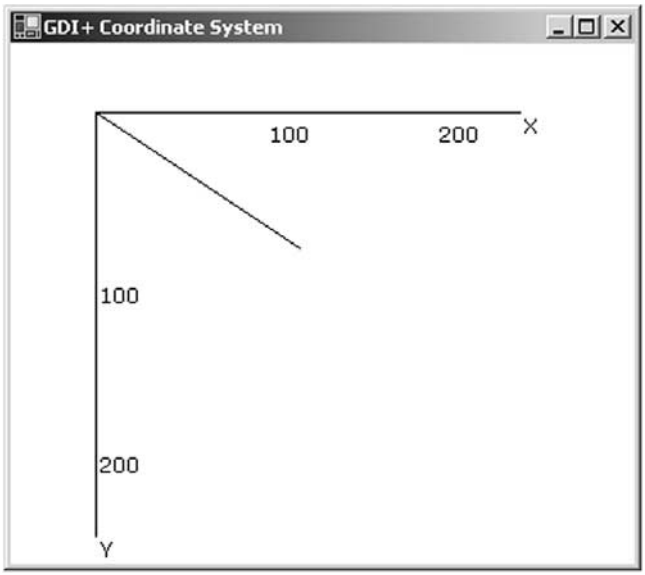

FIGURE 10.4: Drawing a line from point (0, 0) to point (120, 80) with origin (50, 40)

device coordinates in this case are the same as the page coordinates because the page unit is in the pixel (default) format.

What is the difference between page and device coordinates? Device coordinates determine what we actually see on the screen. They can be represented in many formats, including pixels, millimeters, and inches. If the device coordinates are in pixel format, the page coordinates and device coordinates will be the same (this is typically true for monitors, but not for printers).

The `PageUnit` property of the `Graphics` class is of type `GraphicsUnit` enumeration. In Listing 10.3 we set the `PageUnit` property to inches. Now graphics objects will be measured in inches, so we need to pass inches instead of pixels. If we draw a line from point (0, 0) to point (2, 1), the line ends 2 inches from the left side and 1 inch from the top of the client area in the page coordinate system. In this case the starting and ending points are (0, 0) and (2, 1) in both world and page coordinates, but the device coordinate system converts them to inches. Hence the starting and ending points

in the device coordinate system are (0, 0) and (192, 96), assuming a resolution of 96 dots per inch.

LISTING 10.3: Setting the device coordinate system to inches

```
g.PageUnit = GraphicsUnit.Inch;
g.DrawLine(Pens.Black, 0, 0, 2, 1);
```

Figure 10.5 shows the output from Listing 10.3. The default width of the pen is 1 page unit, which in this case gives us a pen 1 inch wide.

Now let's create a new pen with a different width. Listing 10.4 creates a pen that's 1 pixel wide (it does so by dividing the number of pixels we want—in this case 1—by the page resolution, which is given by `DpiX`). We draw the line again, this time specifying a red color.

LISTING 10.4: Using the `GraphicsUnit.Inch` option with a pixel width

```
Pen redPen = new Pen(Color.Red, 1/g.DpiX);
g.PageUnit = GraphicsUnit.Inch;
g.DrawLine(Pens.Black, 0, 0, 2, 1);
```

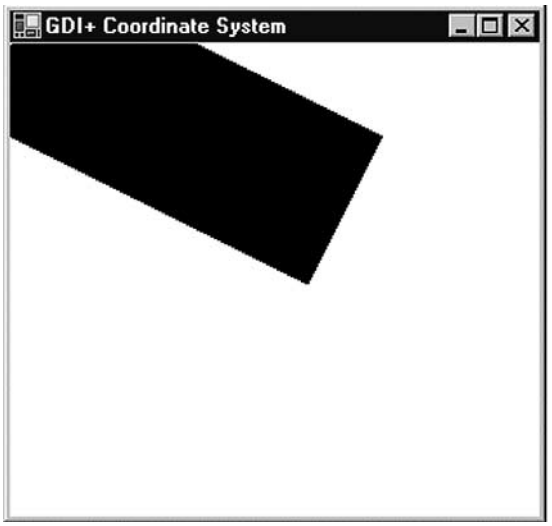

FIGURE 10.5: Drawing with the `GraphicsUnit.Inch` option

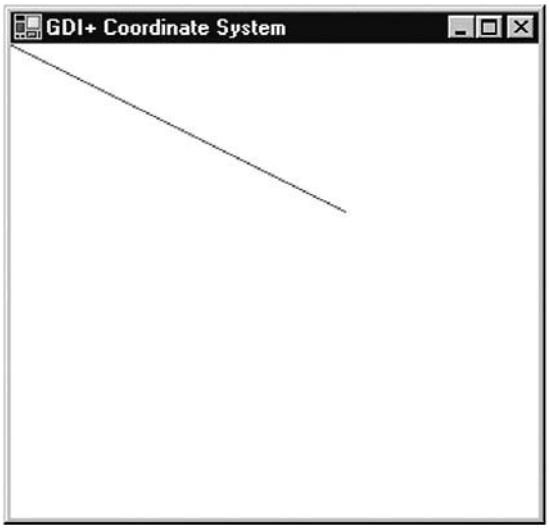

FIGURE 10.6: Drawing with the `GraphicsUnit.Inch` option and a pixel width

Figure 10.6 shows the output from Listing 10.4.

We can also combine the use of page and device coordinates. In Listing 10.5 we transform page coordinates to 1 inch from the left and 0.5 inch from the top of the upper left corner of the client area. Our new page coordinate system has starting and ending points of (1, 0.5) and (3, 1.5), but the device coordinate system converts them to pixels. Hence the starting and ending points in device coordinates are (96, 48) and (288, 144), assuming a resolution of 96 dots per inch.

LISTING 10.5: Combining page and device coordinates

```
Pen redPen = new Pen(Color.Red, 1/g.DpiX);
g.TranslateTransform(1, 0.5f);
g.PageUnit = GraphicsUnit.Inch;
g.DrawLine(redPen, 0, 0, 2, 1);
```

Figure 10.7 shows the output from Listing 10.5.

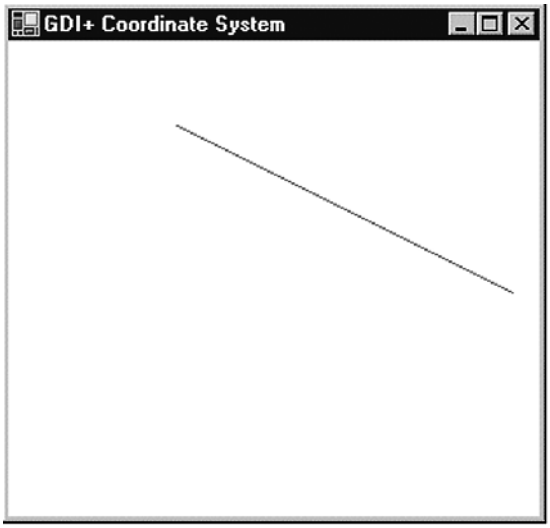

FIGURE 10.7: Combining page and device coordinates

10.2 Transformation Types

There are many types of transformations.

Translation is a transformation of the *xy* plane that moves a graphics object toward or away from the origin of the surface in the *x*- or *y*-direction. For example, moving an object from point A ($x1$, $y1$) to point B ($x2$, $y2$) is a translation operation in which an object is being moved ($y2 - y1$) points in the *y*-direction.

Rotation moves an object around a fixed angle around the center of the plane.

In the **reflection** transformation, an object moves to a position in the opposite direction from an axis, along a line perpendicular to the axis. The resulting object is the same distance from the axis as the original point, but in the opposite direction.

Simple transformations, including rotation, scaling, and reflection are called **linear transformations**. A linear transformation followed by translation is called an **affine transformation**.

The **shearing** transformation skews objects based on a shear factor. In the sample applications discussed throughout this chapter, will see how to use these transformations in GDI+.

So far we've looked at only simple transformations. Now let's discuss some more complex transformation-related functionality defined in the .NET Framework library.

What Can You Transform?

You have just seen the basics of transforming lines. We can also transform graphics objects such as points, curves, shapes, images, text, colors, and textures, as well as colors and images used in pens and brushes.

10.3 The `Matrix` Class and Transformation

Matrices play a vital role in the transformation process. A **matrix** is a multi-dimensional array of values in which each item in the array represents one value of the transformation operation, as we will see in the examples later in this chapter.

In GDI+, the `Matrix` class represents a 3×2 matrix that contains *x*, *y*, and *w* values in the first, second, and third columns, respectively.

NOTE

Before using the `Matrix` class in your applications, you need to add a reference to the `System.Drawing.Drawing2D` namespace.

We can create a `Matrix` object by using its overloaded constructors, which take an array of points (hold the matrix items) as arguments. The following code snippet creates three `Matrix` objects from different overloaded constructors. The first `Matrix` object has no values for its items. The second and third objects have integer and floating point values, respectively, for the first six items of the matrix.

```
Matrix M1 = new Matrix();
Matrix M2 = new Matrix(2, 1, 3, 1, 0, 4);
Matrix M3 =
  new Matrix(0.0f, 1.0f, -1.0f, 0.0f, 0.0f, 0.0f);
```

TABLE 10.1: `Matrix` properties

Property	Description
`Elements`	Returns an array containing matrix elements.
`IsIdentity`	Returns `true` if the matrix is an identity matrix; otherwise returns `false`.
`IsInvertible`	Returns `true` if a matrix is invertible; otherwise returns `false`.
`OffsetX`	Returns the *x* translation value of a matrix.
`OffsetY`	Returns the *y* translation value of a matrix.

The `Matrix` class provides properties for accessing and setting its member values. Table 10.1 describes these properties.

The `Matrix` class provides methods to invert, rotate, scale, and transform matrices. The `Invert` method is used to reverse a matrix if it is invertible. This method takes no parameters.

> **NOTE**
> The `Transform` property of the `Graphics` class is used to apply a transformation in the form of a `Matrix` object. We will discuss this property in more detail in Section 10.4.

Listing 10.6 uses the `Invert` method to invert a matrix. We create a `Matrix` object and read its original values. Then we call the `Invert` method and read the new values.

LISTING 10.6: Inverting a matrix

```
private void InvertMenu_Click(object sender,
    System.EventArgs e)
{
    string str = "Original values: ";
    // Create a Matrix object
    Matrix X = new Matrix(2, 1, 3, 1, 0, 4);
    // Write its values
    for(int i=0; i<X.Elements.Length; i++)
    {
```

```
            str += X.Elements[i].ToString();
            str += ", ";
        }
        str += "\n";
        str += "Inverted values: ";
        // Invert matrix
        X.Invert();
        float[] pts = X.Elements;
        // Read inverted matrix
        for(int i=0; i<pts.Length; i++)
        {
            str += pts[i].ToString();
            str += ", ";
        }
        // Display result
        MessageBox.Show(str);
    }
```

The `Multiply` method multiplies a new matrix against an existing matrix and stores the result in the first matrix. `Multiply` takes two arguments. The first is the new matrix by which you want to multiply the existing matrix, and the second is an optional `MatrixOrder` argument that indicates the order of multiplication.

The `MatrixOrder` enumeration has two values: `Append` and `Prepend`. `Append` specifies that the new operation is applied after the preceding operation; `Prepend` specifies that the new operation is applied before the preceding operation during cumulative operations. Listing 10.7 multiplies two matrices. We create two `Matrix` objects and use the `Multiply` method to multiply the second matrix by the first. Then we read and display the resultant matrix.

Listing 10.7: Multiplying two matrices

```
    private void MultiplyMenu_Click(object sender,
        System.EventArgs e)
    {
        string str = null;
        // Create two Matrix objects
        Matrix X =
            new Matrix(2.0f, 1.0f, 3.0f, 1.0f, 0.0f, 4.0f);
        Matrix Y =
            new Matrix(0.0f, 1.0f, -1.0f, 0.0f, 0.0f, 0.0f);
        // Multiply two matrices
        X.Multiply(Y, MatrixOrder.Append);
```

continues

```
    // Read the resultant matrix
    for(int i=0; i<X.Elements.Length; i++)
    {
        str += X.Elements[i].ToString();
        str += ", ";
    }
    // Display result
    MessageBox.Show(str);
}
```

The `Reset` method resets a matrix to the identity matrix (see Figure 10.21 for an example of an identity matrix). If we call the `Reset` method and then apply a matrix to transform an object, the result will be the original object.

The `Rotate` and `RotateAt` methods are used to rotate a matrix. The `Rotate` method rotates a matrix at a specified angle. This method takes two arguments: a floating point value specifying the angle, and (optionally) the matrix order. The `RotateAt` method is useful when you need to change the center of the rotation. Its first parameter is the angle; the second parameter (of type `float`) specifies the center of rotation. The third (optional) parameter is the matrix order.

Listing 10.8 simply creates a `Graphics` object using the `CreateGraphics` method and calls `DrawLine` and `FillRectangle` to draw a line and fill a rectangle, respectively.

Listing 10.8: Drawing a line and filling a rectangle

```
private void Rotate_Click(object sender,
    System.EventArgs e)
{
    // Create a Graphics object
    Graphics g = this.CreateGraphics();
    g.Clear(this.BackColor);
    // Draw a line
    g.DrawLine(new Pen(Color.Green, 3),
        new Point(120, 50),
        new Point(200, 50));
    // Fill a rectangle
    g.FillRectangle(Brushes.Blue,
        200, 100, 100, 60);
    // Dispose of object
    g.Dispose();
}
```

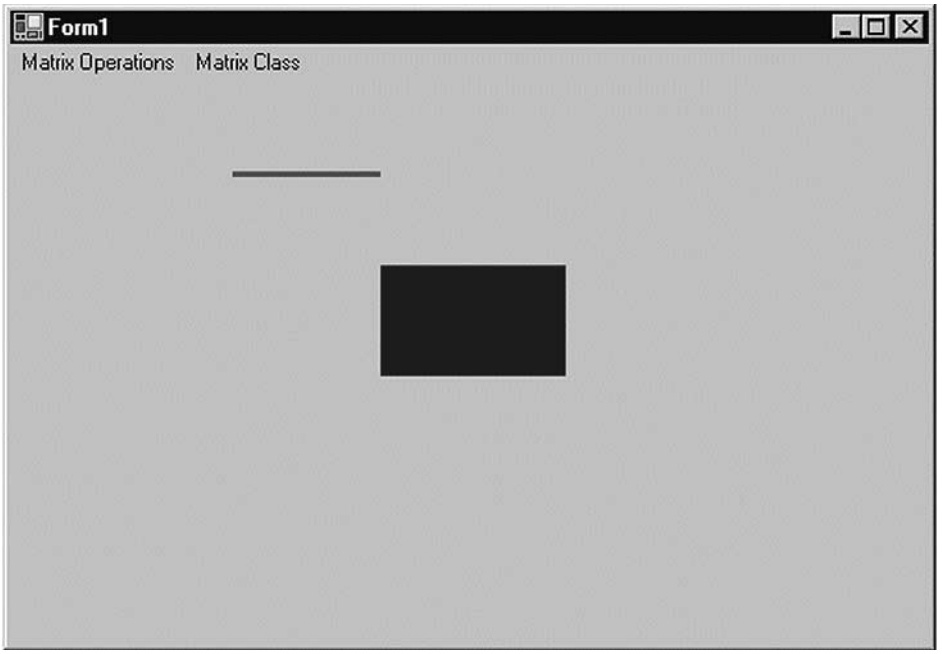

FIGURE 10.8: Drawing a line and filling a rectangle

Figure 10.8 shows the output from Listing 10.8.

Now let's rotate our graphics objects, using the `Matrix` object. In Listing 10.9 we create a `Matrix` object, call its `Rotate` method to rotate the matrix 45 degrees, and apply the `Matrix` object to the `Graphics` object by setting its `Transform` property.

LISTING 10.9: Rotating graphics objects

```
private void Rotate_Click(object sender,
    System.EventArgs e)
{
    // Create a Graphics object
    Graphics g = this.CreateGraphics();
    g.Clear(this.BackColor);
    // Create a Matrix object
    Matrix X = new Matrix();
    // Rotate by 45 degrees
    X.Rotate(45, MatrixOrder.Append);
    // Apply Matrix object to the Graphics object
```

continues

```
        // (i.e., to all the graphics items
        // drawn on the Graphics object)
        g.Transform = X;
        // Draw a line
        g.DrawLine(new Pen(Color.Green, 3),
            new Point(120, 50),
            new Point(200, 50));
        // Fill a rectangle
        g.FillRectangle(Brushes.Blue,
            200, 100, 100, 60);
        // Dispose of object
        g.Dispose();
    }
```

Figure 10.9 shows the new output. Both objects (line and rectangle) have been rotated 45 degrees.

Now let's replace `Rotate` with `RotateAt`, as in Listing 10.10.

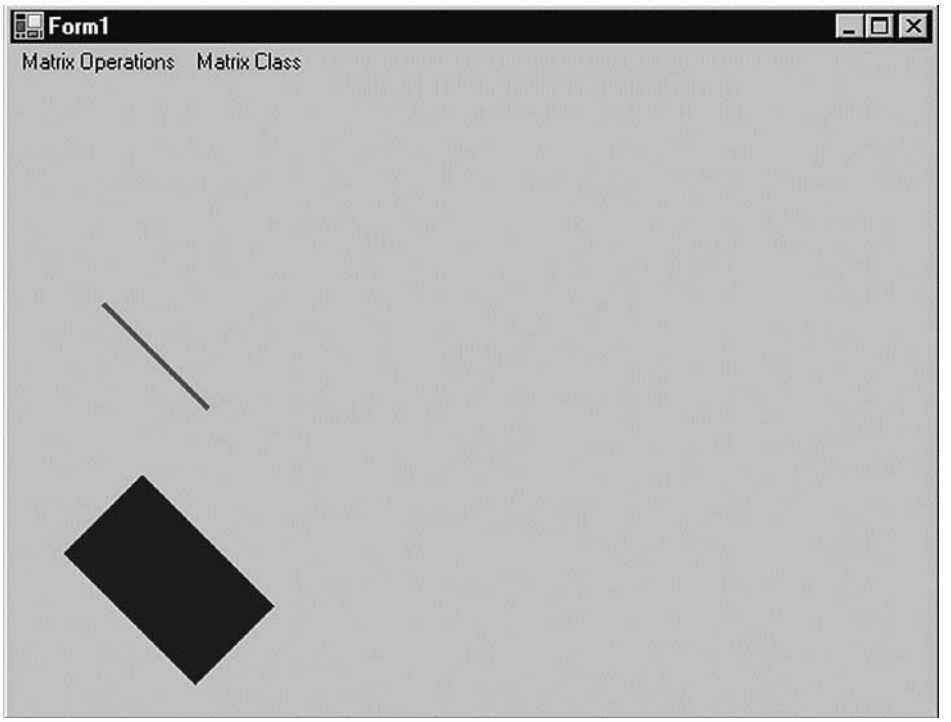

FIGURE 10.9: Rotating graphics objects

LISTING 10.10: Using the `RotateAt` method

```
private void RotateAtMenu_Click(object sender,
    System.EventArgs e)
{
    // Create a Graphics object
    Graphics g = this.CreateGraphics();
    g.Clear(this.BackColor);
    // Create a Matrix object
    Matrix X = new Matrix();
    // Create a point
    PointF pt = new PointF(180.0f, 50.0f);
    // Rotate by 45 degrees
    X.RotateAt(45, pt, MatrixOrder.Append);
    // Apply the Matrix object to the Graphics object
    // (i.e., to all the graphics items
    // drawn on the Graphics object)
    g.Transform = X;
    // Draw a line
    g.DrawLine(new Pen(Color.Green, 3),
        new Point(120, 50),
        new Point(200, 50));
    // Fill a rectangle
    g.FillRectangle(Brushes.Blue,
        200, 100, 100, 60);
    // Dispose of object
    g.Dispose();
}
```

This new code generates Figure 10.10.

If we call the `Reset` method in Listing 10.10 after `RotateAt` and before `g.Transform`, like this:

```
X.RotateAt(45, pt, MatrixOrder.Append);
// Reset the matrix
X.Reset();
// Apply the Matrix object to the Graphics object
// (i.e., to all the graphics items
// drawn on the Graphics object)
g.Transform = X;
```

the revised code generates Figure 10.11, which is the same as Figure 10.8. There is no rotation because the `Reset` method resets the transformation.

The `Scale` method scales a matrix in the *x*- and *y*-directions. This method takes two floating values (scale factors), for the *x*- and *y*-axes,

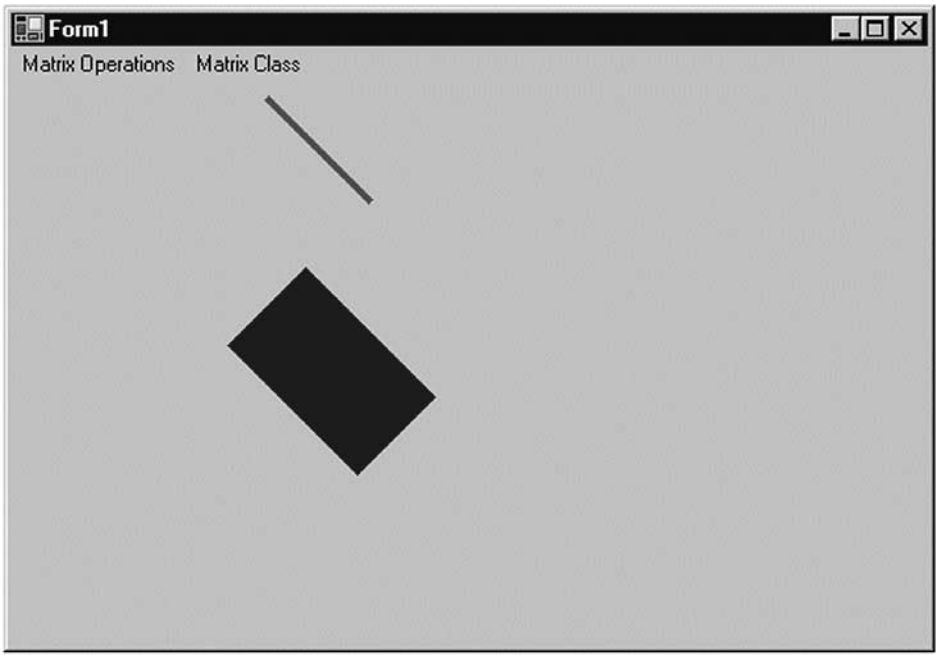

FIGURE 10.10: Using the `RotateAt` method

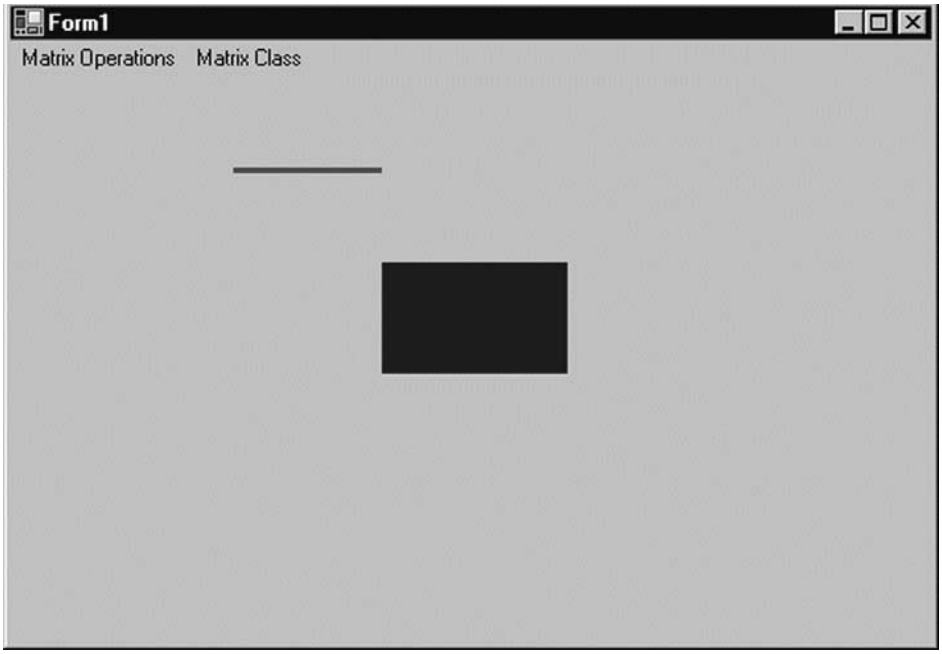

FIGURE 10.11: Resetting a transformation

respectively. In Listing 10.11 we draw a rectangle with a width of 20 and a height of 30. Then we create a `Matrix` object and scale it by calling its `Scale` method with arguments 3 and 4 in the *x*- and *y*-directions, respectively.

LISTING 10.11: Scaling graphics objects

```
private void Scale_Click(object sender,
    System.EventArgs e)
{
    // Create Graphics object
    Graphics g = this.CreateGraphics();
    g.Clear(this.BackColor);
    // Draw a filled rectangle with
    // width 20 and height 30
    g.FillRectangle(Brushes.Blue,
        20, 20, 20, 30);
    // Create Matrix object
    Matrix X = new Matrix();
    // Apply 3X scaling
    X.Scale(3, 4, MatrixOrder.Append);
    // Apply transformation on the form
    g.Transform = X;
    // Draw a filled rectangle with
    // width 20 and height 30
    g.FillRectangle(Brushes.Blue,
        20, 20, 20, 30);
    // Dispose of object
    g.Dispose();
}
```

Figure 10.12 shows the output from Listing 10.11. The first rectangle is the original rectangle; the second rectangle is the scaled rectangle, in which the *x* position (and width) is scaled by 3, and the *y* position (and height) is scaled by 4.

The `Shear` method provides a shearing transformation and takes two floating point arguments, which represent the horizontal and vertical shear factors, respectively. In Listing 10.12 we draw a filled rectangle with a hatch brush. Then we call the `Shear` method to shear the matrix by 2 in the vertical direction, and we use `Transform` to apply the `Matrix` object.

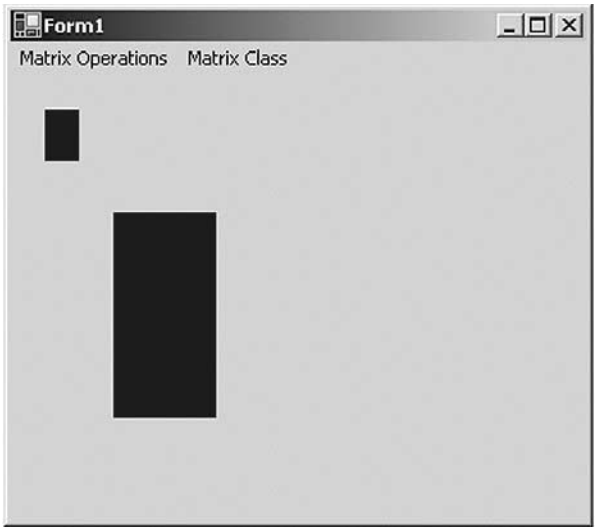

FIGURE 10.12: Scaling a rectangle

LISTING 10.12: Shearing graphics objects

```
private void Shear_Click(object sender,
    System.EventArgs e)
{
    // Create a Graphics object
    Graphics g = this.CreateGraphics();
    g.Clear(this.BackColor);
    // Create a brush
    HatchBrush hBrush = new HatchBrush
        (HatchStyle.DarkVertical,
        Color.Green, Color.Yellow);
    // Fill a rectangle
    g.FillRectangle(hBrush,
        100, 50, 100, 60);
    // Create a Matrix object
    Matrix X = new Matrix();
    // Shear
    X.Shear(2, 1);
    // Apply transformation
    g.Transform = X;
    // Fill rectangle
    g.FillRectangle(hBrush,
        10, 100, 100, 60);
    // Dispose of objects
    hBrush.Dispose();
    g.Dispose();
}
```

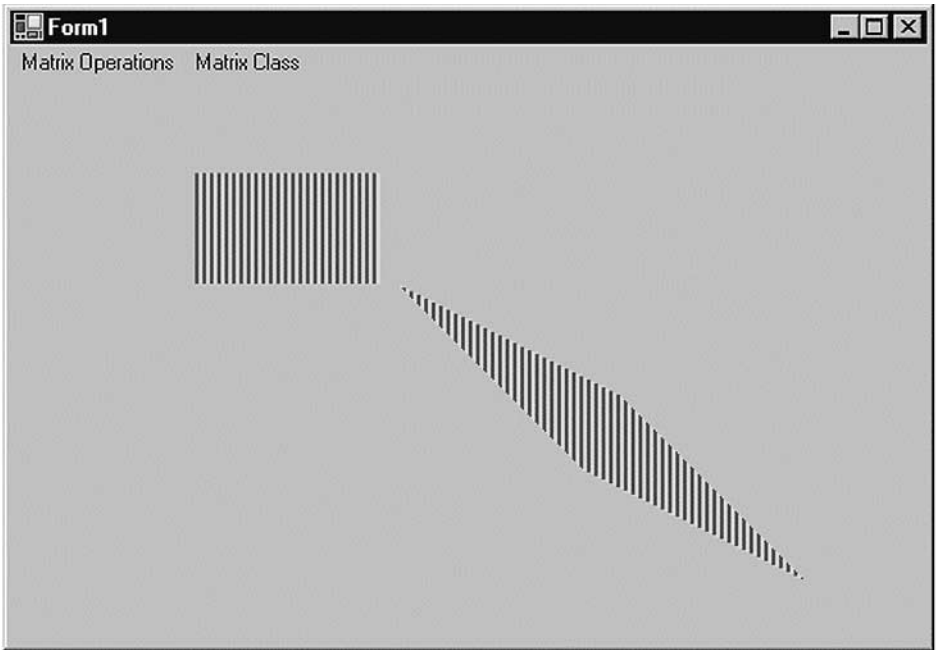

FIGURE 10.13: Shearing a rectangle

Figure 10.13 shows the output from Listing 10.12. The first rectangle in this figure is the original; the second is sheared.

The `Translate` method translates objects by the specified value. This method takes two floating point arguments, which represent the *x* and *y* offsets. For example, Listing 10.13 translates the original rectangle by 100 pixels each in the *x*- and *y*-directions.

LISTING 10.13: Translating graphics objects

```
private void Translate_Click(object sender,
    System.EventArgs e)
{
    // Create a Graphics obhect
    Graphics g = this.CreateGraphics();
    g.Clear(this.BackColor);
    // Draw a filled rectangle
    g.FillRectangle(Brushes.Blue,
        50, 50, 100, 60);
```

continues

```
        // Create a Matrix object
        Matrix X = new Matrix();
        // Translate by 100 in the x direction
        // and 100 in the y direction
        X.Translate(100, 100);
        // Apply transformation
        g.Transform = X;
        // Draw a filled rectangle after
        // translation
        g.FillRectangle(Brushes.Blue,
            50, 50, 100, 60);
        // Dispose of object
        g.Dispose();
    }
```

Here we draw two rectangles with a width of 100 and a height of 60. Both rectangles start at (50, 50), but the code generates Figure 10.14. Even though the rectangles were drawn with the same size and location, the second rectangle after translation is now located 100 points away in the *x*- and *y*-directions from the first rectangle.

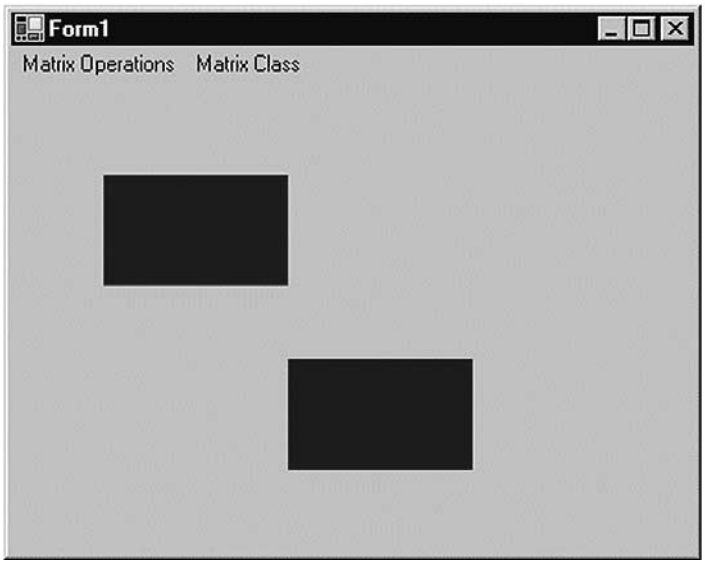

Figure 10.14: Translating a rectangle

10.4 The `Graphics` Class and Transformation

In Chapter 3 we saw that the `Graphics` class provides some transformation-related members. Before we move to other transformation-related classes, let's review the transformation functionality defined in the `Graphics` class, as described in Table 10.2. We will see how to use these members in the examples throughout this chapter.

The `Transform` property of the `Graphics` class represents the world transformation of a `Graphics` object. It is applied to all items of the object. For example, if you have a rectangle, an ellipse, and a line and set the

TABLE 10.2: Transformation-related members defined in the `Graphics` class

Member	Description
`MultiplyTransform`	Method that multiplies the world transformation of a `Graphics` object and a `Matrix` object. The `Matrix` object specifies the transformation action (scaling, rotation, or translation).
`ResetTransform`	Method that resets the world transformation matrix of a `Graphics` object to the identity matrix.
`RotateTransform`	Method that applies a specified rotation to the transformation matrix of a `Graphics` object.
`ScaleTransform`	Method that applies a specified scaling operation to the transformation matrix of a `Graphics` object by prepending it to the object's transformation matrix.
`Transform`	Property that represents the world transformation for a `Graphics` object. Both get and set.
`TransformPoints`	Method that transforms an array of points from one coordinate space to another using the current world and page transformations of a `Graphics` object.
`TranslateClip`	Method that translates the clipping region of a `Graphics` object by specified amounts in the horizontal and vertical directions.
`TranslateTransform`	Method that prepends the specified translation to the transformation matrix of a `Graphics` object.

Transform property of the Graphics object, it will be applied to all three items. The Transform property is a Matrix object. The following code snippet creates a Matrix object and sets the Transform property:

```
Matrix X = new Matrix();
X.Scale(2, 2, MatrixOrder.Append);
g.Transform = X;
```

The transformation methods provided by the Graphics class are MultiplyTransform, ResetTransform, RotateTransform, ScaleTransform, TransformPoints, TranslateClip, and TranslateTransform. The MultiplyTransform method multiplies a transformation matrix by the world transformation coordinates of a Graphics object. It takes an argument of Matrix type. The second argument, which specifies the order of multiplication operation, is optional. The following code snippet creates a Matrix object with the Translate transformation. The MultiplyTransform method multiplies the Matrix object by the world coordinates of the Graphics object, translating all graphics items drawn by the Graphics object.

```
Matrix X = new Matrix();
X. Translate(200.0F, 100.0F);
g.MultiplyTransform(X, MatrixOrder.Append);
```

RotateTransform rotates the world transform by a specified angle. This method takes a floating point argument, which represents the rotation angle, and an optional second argument of MatrixOrder. The following code snippet rotates the world transformation of the Graphics object by 45 degrees:

```
g.RotateTransform(45.0F, MatrixOrder.Append);
```

The ScaleTransform method scales the world transformation in the specified *x*- and *y*-directions. The first and second arguments of this method are *x*- and *y*-direction scaling factors, and the third optional argument is MatrixOrder. The following code snippet scales the world transformation by 2 in the *x*-direction and by 3 in the *y*-direction:

```
g.ScaleTransform(2.0F, 3.0F, MatrixOrder.Append);
```

The `TranslateClip` method translates the clipping region in the horizontal and vertical directions. The first argument of this method represents the translation in the *x*-direction, and the second argument represents the translation in the *y*-direction:

```
e.Graphics.TranslateClip(20.0f, 10.0f);
```

The `TranslateTransform` method translates the world transformation by the specified *x*- and *y*-values and takes an optional third argument of `MatrixOrder`:

```
g.TranslateTransform(100.0F, 0.0F, MatrixOrder.Append);
```

We will use all of these methods in our examples.

10.5 Global, Local, and Composite Transformations

Transformations can be divided into two categories based on their scope: global and local. In addition, there are composite transformations. A **global transformation** is applicable to all items of a `Graphics` object. The `Transform` property of the `Graphics` class is used to set global transformations.

A **composite transformation** is a sequence of transformations. For example, scaling followed by translation and rotation is a composite translation. The `MultiplyTransform`, `RotateTransform`, `ScaleTransform`, and `TranslateTransform` methods are used to generate composite transformations.

Listing 10.14 draws two ellipses and a rectangle, then calls `ScaleTransform`, `TranslateTransform`, and `RotateTransform` (a composite transformation). The items are drawn again after the composite transformation.

LISTING 10.14: Applying a composite transformation

```
private void GlobalTransformation_Click(object sender,
      System.EventArgs e)
{
      // Create a Graphics object
      Graphics g = this.CreateGraphics();
```

continues

```
        g.Clear(this.BackColor);
        // Create a blue pen with width of 2
        Pen bluePen = new Pen(Color.Blue, 2);
        Point pt1 = new Point(10, 10);
        Point pt2 = new Point(20, 20);
        Color [] lnColors = {Color.Black, Color.Red};
        Rectangle rect1 = new Rectangle(10, 10, 15, 15);
        // Create two linear gradient brushes
        LinearGradientBrush lgBrush1 = new LinearGradientBrush
              (rect1, Color.Blue, Color.Green,
              LinearGradientMode.BackwardDiagonal);
        LinearGradientBrush lgBrush = new LinearGradientBrush
              (pt1, pt2, Color.Red, Color.Green);
        // Set linear colors
        lgBrush.LinearColors = lnColors;
        // Set gamma correction
        lgBrush.GammaCorrection = true;
        // Fill and draw rectangle and ellipses
        g.FillRectangle(lgBrush, 150, 0, 50, 100);
        g.DrawEllipse(bluePen, 0, 0, 100, 50);
        g.FillEllipse(lgBrush1, 300, 0, 100, 100);
        // Apply scale transformation
        g.ScaleTransform(1, 0.5f);
        // Apply translate transformation
        g.TranslateTransform(50, 0, MatrixOrder.Append);
        // Apply rotate transformation
        g.RotateTransform(30.0f, MatrixOrder.Append);
        // Fill ellipse
        g.FillEllipse(lgBrush1, 300, 0, 100, 100);
        // Rotate again
        g.RotateTransform(15.0f, MatrixOrder.Append);
        // Fill rectangle
        g.FillRectangle(lgBrush, 150, 0, 50, 100);
        // Rotate again
        g.RotateTransform(15.0f, MatrixOrder.Append);
        // Draw ellipse
        g.DrawEllipse(bluePen, 0, 0, 100, 50);
        // Dispose of objects
        lgBrush1.Dispose();
        lgBrush.Dispose();
        bluePen.Dispose();
        g.Dispose();
    }
```

Figure 10.15 shows the output from Listing 10.14.

A **local transformation** is applicable to only a specific item of a `Graphics` object. The best example of local transformation is transforming a graphics path. The `Translate` method of the `GraphicsPath` class trans-

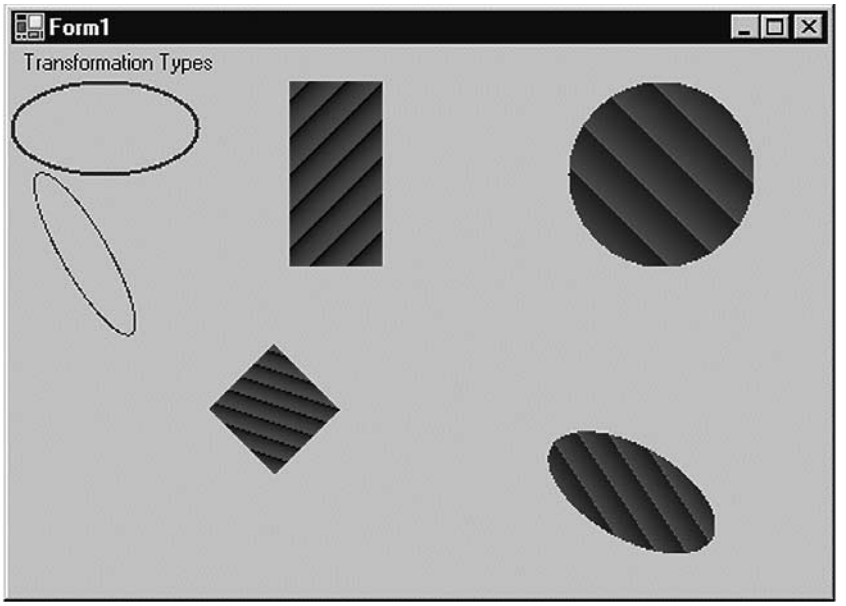

FIGURE 10.15: Composite transformation

lates only the items of a graphics path. Listing 10.15 translates a graphics path. We create a `Matrix` object and apply rotate and translate transformations to it.

LISTING 10.15: Translating graphics path items

```
private void LocalTransformation_Click(object sender,
    System.EventArgs e)
{
    // Create a Graphics object
    Graphics g = this.CreateGraphics();
    g.Clear(this.BackColor);
    // Create a GraphicsPath object
    GraphicsPath path = new GraphicsPath();
    // Add an ellipse and a line to the
    // graphics path
    path.AddEllipse(50, 50, 100, 150);
    path.AddLine(20, 20, 200, 20);
    // Create a blue pen with a width of 2
    Pen bluePen = new Pen(Color.Blue, 2);
```

continues

```
        // Create a Matrix object
        Matrix X = new Matrix();
        // Rotate 30 degrees
        X.Rotate(30);
        // Translate with 50 offset in x direction
        X.Translate(50.0f, 0);
        // Apply transformation on the path
        path.Transform(X);
        // Draw a rectangle, a line, and the path
        g.DrawRectangle(Pens.Green, 200, 50, 100, 100);
        g.DrawLine(Pens.Green, 30, 20, 200, 20);
        g.DrawPath(bluePen, path);
        // Dispose of objects
        bluePen.Dispose();
        path.Dispose();
        g.Dispose();
    }
```

Figure 10.16 shows the output from Listing 10.15. The transformation affects only graphics path items (the ellipse and the blue [dark] line).

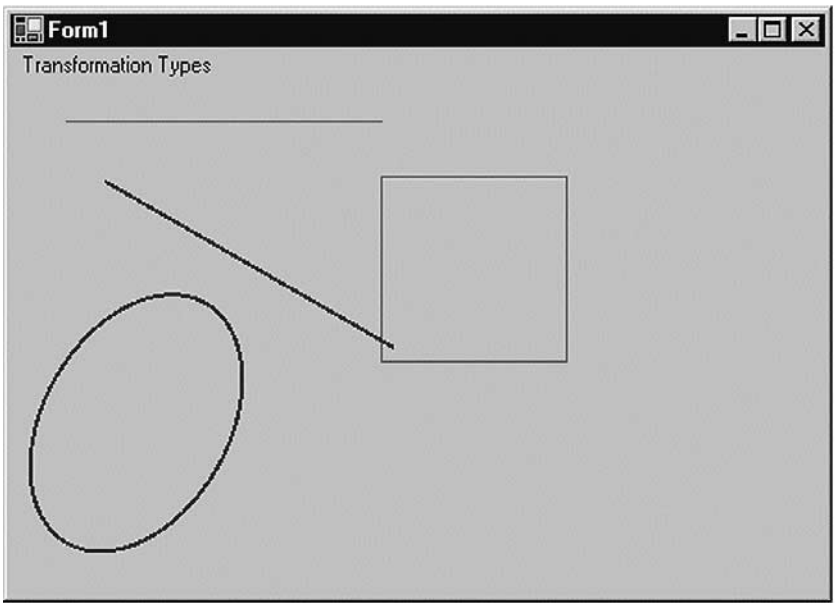

FIGURE 10.16: Local transformation

10.6 Image Transformation

Image transformation is exactly the same as any other transformation process. In this section we will see how to rotate, scale, translate, reflect, and shear images. We will create a `Matrix` object, set the transformation process by calling its methods, set the `Matrix` object as the `Transform` property or the transformation methods of the `Graphics` object, and call `DrawImage`.

Rotating images is similar to rotating other graphics. Listing 10.16 rotates an image. We create a `Graphics` object using the `CreateGraphics` method. Then we create a `Bitmap` object from a file and call the `DrawImage` method, which draws the image on the form. After that we create a `Matrix` object, call its `Rotate` method, rotate the image by 30 degrees, and apply the resulting matrix to the surface using the `Transform` property. Finally, we draw the image again using `DrawImage`.

Listing 10.16: Rotating images

```
private void RotationMenu_Click(object sender,
    System.EventArgs e)
{
    Graphics g = this.CreateGraphics();
    g.Clear(this.BackColor);
    Bitmap curBitmap = new Bitmap(@"roses.jpg");
    g.DrawImage(curBitmap, 0, 0, 200, 200);
    // Create a Matrix object, call its Rotate method,
    // and set it as Graphics.Transform
    Matrix X = new Matrix();
    X.Rotate(30);
    g.Transform = X;
    // Draw image
    g.DrawImage(curBitmap,
        new Rectangle(205, 0, 200, 200),
        0, 0, curBitmap.Width,
        curBitmap.Height,
        GraphicsUnit.Pixel) ;
    // Dispose of objects
    curBitmap.Dispose();
    g.Dispose();
}
```

Figure 10.17 shows the output from Listing 10.16. The first image is the original; the second image is rotated.

FIGURE 10.17: Rotating images

Now let's apply other transformations. Replacing the `Rotate` method in Listing 10.16 with the following line scales the image:

```
X.Scale(2, 1, MatrixOrder.Append);
```

The scaled image is shown in Figure 10.18.

Replacing the `Rotate` method in Listing 10.16 with the following line translates the image with 100 offset in the *x*- and *y*-directions:

```
X.Translate(100, 100);
```

The new output is shown in Figure 10.19.

Replacing the `Rotate` method in Listing 10.16 with the following line shears the image:

```
X.Shear(2, 1);
```

FIGURE 10.18: Scaling images

FIGURE 10.19: Translating images

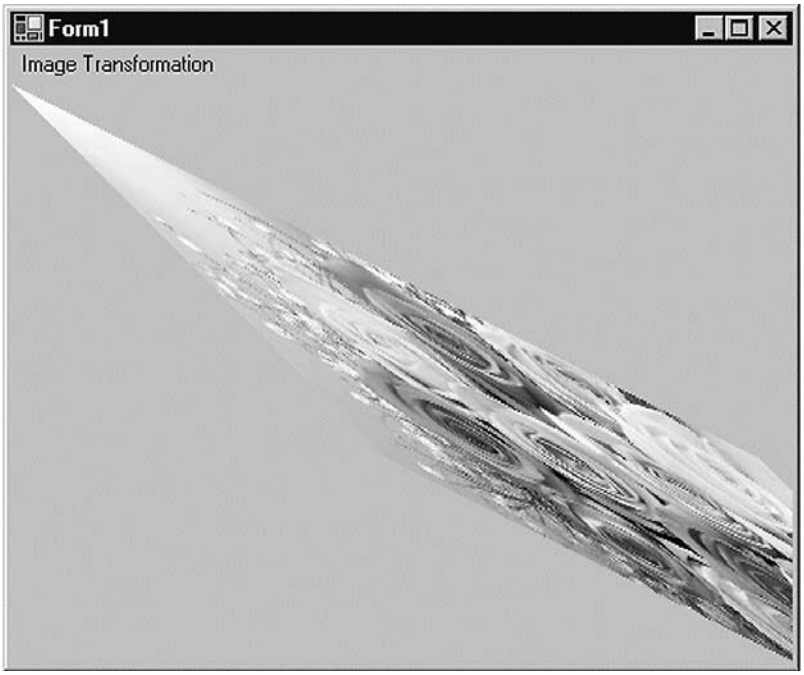

FIGURE 10.20: Shearing images

The new output is shown in Figure 10.20.

You have probably noticed that image transformation is really no different from the transformation of other graphics objects. We recommend that you download the source code samples from online to see the detailed code listings.

10.7 Color Transformation and the Color Matrix

So far we have seen the transformation of graphics shapes from one state to another, but have you ever thought about transforming colors? Why *would* you want to transform an image's colors? Suppose you wanted to provide grayscale effects, or needed to adjust the contrast, brightness, or even "redness" of an image. For example, images retrieved from video and still cameras often need correction. In these cases, a color matrix is very useful.

As we discussed in earlier chapters, the color of each pixel of a GDI+ image or bitmap is represented by a 32-bit number, of which 8 bits each are

used for the red, green, blue, and alpha components. Each of the four components is a number from 0 to 255. For red, green, and blue, 0 represents no intensity and 255 represents full intensity. For the alpha component, 0 represents transparent and 255 represents fully opaque. A color vector includes four items: A, R, G, and B. The minimum values for this vector are (0, 0, 0, 0), and the maximum values are (255, 255, 255, 255).

GDI+ allows the use of values between 0 and 1, where 0 represents the minimum intensity and 1 the maximum intensity. These values are used in a color matrix to represent the intensity and opacity of color components. For example, the color vector with minimum values is (0, 0, 0, 0), and the color vector with maximum values is (1, 1, 1, 1).

In a color transformation we can apply a color matrix on a color vector by multiplying a 4×4 matrix. However, a 4×4 matrix supports only linear transformations such as rotation and scaling. To perform nonlinear transformations such as translation, we must use a 5×5 matrix. The element of the fifth row and the fifth column of the matrix must be 1, and all of the other entries in the five columns must be 0.

The elements of the matrix are identified according to a zero-based index. The first element of the matrix is M[0][0], and the last element is M[4][4]. A 5×5 identity matrix is shown in Figure 10.21. In this matrix the elements M[0][0], M[1][1], M[2][2], and M[3][3] represent the red, blue, green, and alpha factors, respectively. The element M[4][4] means nothing, and it must always be 1.

Now if we want to double the intensity of the red component of a color, we simply set M[0][0] equal to 2. For example, the matrix shown in Figure 10.22 doubles the intensity of the red component, decreases the

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

FIGURE 10.21: An identity matrix

$$\begin{bmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

FIGURE 10.22: A matrix whose components have different intensities

intensity of the green component by half, triples the intensity of the blue component, and decreases the opacity of the color by half (making it semi-transparent).

In the matrix shown in Figure 10.22, we multiplied the intensity values. We can also add intensity values by using other matrix elements. For example, the matrix shown in Figure 10.23 will double the intensity of the red component and add 0.2 to each of the red, green, and blue component intensities.

10.7.1 The `ColorMatrix` Class

In this section we will discuss the `ColorMatrix` class. As you might guess from its name, this class defines a matrix of colors. In the preceding sections we discussed the `Matrix` class. The `ColorMatrix` class is not very different from the `Matrix` class. Whereas the `Matrix` class is used in general transformation to transform graphics shapes and images, the `ColorMatrix` class is specifically designed to transform colors. Before we see practical use of the color transformation, we will discuss the `ColorMatrix` class, its properties, and its methods.

$$\begin{bmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0.2 & 0.2 & 0.2 & 0 & 1 \end{bmatrix}$$

FIGURE 10.23: A color matrix with multiplication and addition

The `ColorMatrix` class constructor takes an array that contains the values of matrix items. The `Item` property of this class represents a cell of the matrix and can be used to get and set cell values. Besides the `Item` property, the `ColorMatrix` class provides 25 `MatrixXY` properties, which represent items of the matrix at row (x + 1) and column (y + 1). `MatrixXY` properties can be used to get and set an item's value.

Listing 10.17 creates a `ColorMatrix` object with item (4, 4) set to 0.5 (half opacity). Then it sets the values of item (3, 4) to 0.8 and item (1, 1) to 0.3.

LISTING 10.17: Creating a `ColorMatrix` object

```
float[][] ptsArray ={
   new float[] {1, 0, 0, 0, 0},
   new float[] {0, 1, 0, 0, 0},
   new float[] {0, 0, 1, 0, 0},
   new float[] {0, 0, 0, 0.5f, 0},
   new float[] {0, 0, 0, 0, 1}};
ColorMatrix clrMatrix = new ColorMatrix(ptsArray);
if( clrMatrix.Matrix34 <= 0.5)
{
    clrMatrix.Matrix34 = 0.8f;
         clrMatrix.Matrix11 = 0.3f;
}
```

Section 10.8 will describe how to apply color matrices to the transformation of colors.

10.8 Matrix Operations in Image Processing

Recoloring, the process of changing image colors, is a good example of color transformation. Recoloring includes changing colors, intensity, contrast, and brightness of an image. It can all be done via the `ImageAttributes` class and its methods.

The color matrix can be applied to an image via the `SetColorMatrix` method of the `ImageAttributes` class. The `ImageAttributes` object is used as a parameter when we call `DrawImage`.

10.8.1 Translating Colors

Translating colors increases or decreases color intensities by a set amount (not by multiplying them). Each color component (red, green, and blue) has

255 different intensity levels ranging from 0 to 255. For example, assume that the current intensity level for the red component of a color is 100. Changing its intensity level to 150 would imply translating by 50.

In a color matrix representation, the intensity varies from 0 to 1. The last row's first four elements represent the translation of red, green, blue, and alpha components of a color, as shown in Figure 10.22. Hence, adding a value to these elements will transform a color. For example, the t1, t2, t3, and t4 values in the following color matrix represent the red, green, blue, and alpha component translations, respectively:

```
Color Matrix = {
{1,  0,  0,  0, 0},
{0,  1,  0,  0, 0},
{0,  0,  1,  0, 0},
{0,  0,  0,  1, 0},
{t1, t2, t3, t4, 1}};
```

Listing 10.18 uses a `ColorMatrix` object to translate colors. We change the current intensity of the red component to 0.90. First we create a `Graphics` object using the `CreateGraphics` method, and we create a `Bitmap` object from a file. Next we create an array of `ColorMatrix` elements and create a `ColorMatrix` object from this array. Then we create an `ImageAttributes` object and set the color matrix using `SetColorMatrix`, which takes the `ColorMatrix` object as its first parameter. After all that, we draw two images. The first image has no effects; the second image shows the result of our color matrix transformation. Finally, we dispose of the objects.

LISTING 10.18: Using `ColorMatrix` to translate colors

```
private void TranslationMenu_Click(object sender,
    System.EventArgs e)
{
    // Create a Graphics object
    Graphics g = this.CreateGraphics();
    g.Clear(this.BackColor);
    // Create a Bitmap object
    Bitmap curBitmap = new Bitmap("roses.jpg");
    // Color matrix elements
    float[][] ptsArray =
    {
        new float[] {1,  0,  0,  0, 0},
```

```
            new float[] {0,  1,  0,  0, 0},
            new float[] {0,  0,  1,  0, 0},
            new float[] {0,  0,  0,  1, 0},
            new float[] {.90f, .0f, .0f, .0f, 1}
        };
        // Create a ColorMatrix object
        ColorMatrix clrMatrix = new ColorMatrix(ptsArray);
        // Create image attributes
        ImageAttributes imgAttribs = new ImageAttributes();
        // Set color matrix
        imgAttribs.SetColorMatrix(clrMatrix,
            ColorMatrixFlag.Default,
            ColorAdjustType.Default);
        // Draw image with no effects
        g.DrawImage(curBitmap, 0, 0, 200, 200);
        // Draw image with image attributes
        g.DrawImage(curBitmap,
            new Rectangle(205, 0, 200, 200),
            0, 0, curBitmap.Width, curBitmap.Height,
            GraphicsUnit.Pixel, imgAttribs) ;
        // Dispose of objects
        curBitmap.Dispose();
        g.Dispose();
    }
```

Figure 10.24 shows the output from Listing 10.18. The original image is on the left; on the right we have the results of our color translation. If you

FIGURE 10.24: Translating colors

change the values of other components (red, blue, and alpha) in the last row of the color matrix, you'll see different results.

10.8.2 Scaling Colors

Scaling color involves multiplying a color component value by a scaling factor. For example, the t1, t2, t3, and t4 values in the following color matrix represent the red, green, blue, and alpha components, respectively. If we change the value of M[2][2] to 0.5, the transformation operation will multiply the green component by 0.5, cutting its intensity by half.

```
Color Matrix = {
{t1,  0,  0,  0, 0},
{0,  t2,  0,  0, 0},
{0,  0,  t3,  0, 0},
{0,  0,  0,  t4, 0},
{0,  0,  0,  0, 1}};
```

Listing 10.19 uses the `ColorMatrix` object to scale image colors.

Listing 10.19: Scaling colors

```
private void ScalingMenu_Click(object sender,
    System.EventArgs e)
{
    // Create a Graphics object
    Graphics g = this.CreateGraphics();
    g.Clear(this.BackColor);
    // Create a Bitmap object
    Bitmap curBitmap = new Bitmap("roses.jpg");
    // Color matrix elements
    float[][] ptsArray =
    {
         new float[] {1,  0,  0,  0, 0},
         new float[] {0,  0.8f,  0,  0, 0},
         new float[] {0,  0,  0.5f,  0, 0},
         new float[] {0,  0,  0,  0.5f, 0},
         new float[] {0, 0, 0, 0, 1}
    };
    // Create a ColorMatrix object
    ColorMatrix clrMatrix = new ColorMatrix(ptsArray);
    // Create image attributes
    ImageAttributes imgAttribs = new ImageAttributes();
    // Set color matrix
    imgAttribs.SetColorMatrix(clrMatrix,
        ColorMatrixFlag.Default,
        ColorAdjustType.Default);
```

```
        // Draw image with no effects
        g.DrawImage(curBitmap, 0, 0, 200, 200);
        // Draw image with image attributes
        g.DrawImage(curBitmap,
            new Rectangle(205, 0, 200, 200),
            0, 0, curBitmap.Width, curBitmap.Height,
            GraphicsUnit.Pixel, imgAttribs) ;
        // Dispose of objects
        curBitmap.Dispose();
        g.Dispose();
    }
```

Figure 10.25 shows the output from Listing 10.19. The original image is on the left; on the right is the image after color scaling. If you change the values of t1, t2, t3, and t4, you will see different results.

10.8.3 Shearing Colors

Earlier in this chapter we discussed image shearing. It can be thought of as anchoring one corner of a rectangular region and stretching the opposite corner horizontally, vertically, or in both directions. Shearing colors is the same process, but here the object is the color instead of the image.

Color shearing increases or decreases a color component by an amount proportional to another color component. For example, consider the

FIGURE 10.25: Scaling colors

transformation in which the red component is increased by one half the value of the blue component. Under such a transformation, the color (0.2, 0.5, 1) would become (0.7, 0.5, 1). The new red component is 0.2 + (0.5)(1) = 0.7. The following color matrix is used to shear image colors.

```
float[][] ptsArray = {
 new float[] {1,  0,  0,  0, 0},
 new float[] {0,  1,  0,  0, 0},
 new float[] {.50f,  0,  1,  0, 0},
 new float[] {0,  0,  0,  1, 0},
 new float[] {0, 0, 0, 0, 1}};
ColorMatrix clrMatrix = new ColorMatrix(ptsArray);
```

If we substitute this color matrix into Listing 10.19, the output will look like Figure 10.26.

FIGURE 10.26: Shearing colors

10.8.4 Rotating Colors

As explained earlier, color in GDI+ has four components: red, green, blue, and alpha. Rotating all four components in a four-dimensional space is hard to visualize. However, such rotation can be visualized in a three-

dimensional space. To do this, we drop the alpha component from the color structure and assume that there are only three colors—red, green, and blue—as shown in Figure 10.27. The three colors—red, green, and blue—are perpendicular to each other, so the angle between any two primary colors is 90 degrees.

Suppose that the red, green, and blue colors are represented by points (1, 0, 0), (0, 1, 0), and (0, 0, 1), respectively. If we rotate a color with a green component of 1, and red and blue components of 0 each, by 90 degrees, the new color will have a red component of 1, and green and blue components of 0 each. If we rotate the color less than 90 degrees, the new color will be located somewhere between green and red.

Figure 10.28 shows how to initialize a color matrix to perform rotations about each of the three components: red, green, and blue.

Listing 10.20 rotates the colors by 45 degrees from the red component.

LISTING 10.20: Rotating colors

```
private void RotationMenu_Click(object sender,
    System.EventArgs e)
{
    float degrees = 45.0f;
    double r = degrees*System.Math.PI/180;
    // Create a Graphics object
    Graphics g = this.CreateGraphics();
    g.Clear(this.BackColor);
    // Create a Bitmap object from a file
    Bitmap curBitmap = new Bitmap("roses.jpg");
```

continues

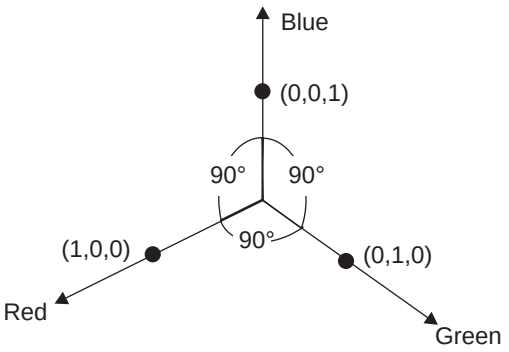

FIGURE 10.27: RGB rotation space

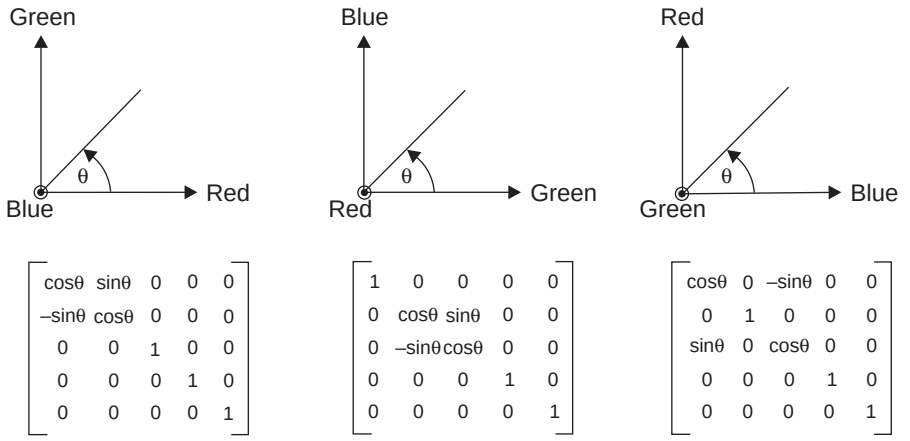

FIGURE 10.28: RGB initialization

```
    // Color matrix elements
    float[][] ptsArray =
    {
         new float[] {(float)System.Math.Cos(r),
                      (float)System.Math.Sin(r),
                      0,  0, 0},
         new float[] {(float)-System.Math.Sin(r),
                       (float)-System.Math.Cos(r),
                           0,  0, 0},
         new float[] {.50f,  0,  1,  0, 0},
         new float[] {0,  0,  0,  1, 0},
         new float[] {0, 0, 0, 0, 1}
    };
    // Create a ColorMatrix object
    ColorMatrix clrMatrix = new ColorMatrix(ptsArray);
    // Create image attributes
    ImageAttributes imgAttribs = new ImageAttributes();
    // Set ColorMatrix to ImageAttributes
    imgAttribs.SetColorMatrix(clrMatrix,
        ColorMatrixFlag.Default,
        ColorAdjustType.Default);
    // Draw image with no effects
    g.DrawImage(curBitmap, 0, 0, 200, 200);
    // Draw image with image attributes
    g.DrawImage(curBitmap,
        new Rectangle(205, 0, 200, 200),
        0, 0, curBitmap.Width, curBitmap.Height,
        GraphicsUnit.Pixel, imgAttribs) ;
    // Dispose of objects
    curBitmap.Dispose();
    g.Dispose();
}
```

FIGURE 10.29: Rotating colors

Figure 10.29 slows the output from Listing 10.20. On the left is the original image; on the right is the image after color rotation.

10.9 Text Transformation

In Chapter 5 we discussed how to use the `ScaleTransform`, `RotateTransform`, and `TranslateTransform` methods to transform text. We can also use a transformation matrix to transform text.

We create a `Matrix` object with the transformation properties and apply it to the surface using the `Transform` property of the `Graphics` object. Listing 10.21 creates a `Matrix` object and sets it as the `Transform` property. We then call `DrawString`, which draws the text on the form. To test this code, add the code to a form's paint event handler.

LISTING 10.21: Text transformation example

```
Graphics g = e.Graphics;
string str =
"Colors, fonts, and text are common" +
" elements of graphics programming." +
"In this chapter, you learned " +
" about the colors, fonts, and text" +
" representations in the "+
".NET Framework class library. "+
"You learned how to create "+
"these elements and use them in GDI+.";
// Create a Matrix object
Matrix M = new Matrix(1, 0, 0.5f, 1, 0, 0);
g.RotateTransform(45.0f,
System.Drawing.Drawing2D.MatrixOrder.Prepend);
g.TranslateTransform(-20, -70);
g.Transform = M;
g.DrawString(str,
new Font("Verdana", 10),
new SolidBrush(Color.Blue),
new Rectangle(50,20,200,300) );
```

Figure 10.30 shows the outcome of Listing 10.21.

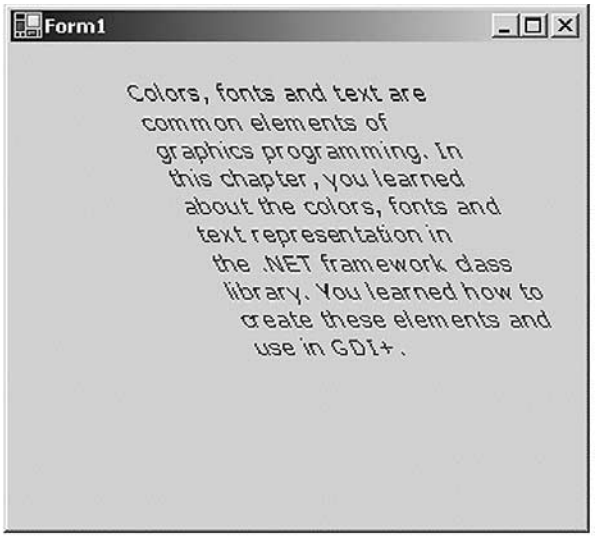

FIGURE 10.30: Using the transformation matrix to transform text

We can apply shearing and other effects by changing the values of `Matrix`. For example, if we change `Matrix` as follows:

```
Matrix M = new Matrix(1, 0.5f, 0, 1, 0, 0);
```

the new code will generate Figure 10.31.

We can reverse the text just by changing the value of the `Matrix` object as follows:

```
Matrix M = new Matrix(1, 1, 1, -1, 0, 0);
```

with the results shown in Figure 10.32.

10.10 The Significance of Transformation Order

The `Matrix` object can store a single transformation or a sequence of transformations. As we learned in Section 10.5, a sequence of transformations is called a *composite transformation*, which is a result of multiplying the matrices of the individual transformations.

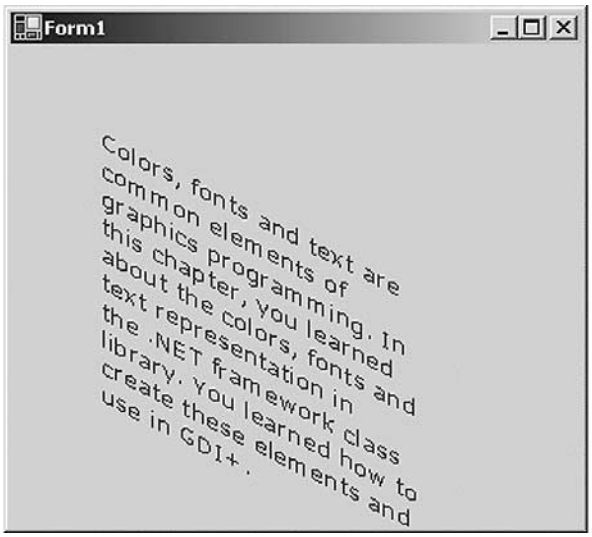

FIGURE 10.31: Using the transformation matrix to shear text

FIGURE 10.32: Using the transformation matrix to reverse text

In a composite transformation, the order of the individual transformations is very important. Matrix operations are not cumulative. For example, the result of a Graphics → Rotate → Translate → Scale → Graphics operation will be different from the result of a Graphics → Scale → Rotate → Translate → Graphics operation. The main reason that order is significant is that transformations like rotation and scaling are done with respect to the origin of the coordinate system. The result of scaling an object that is centered at the origin is different from the result of scaling an object that has been moved away from the origin. Similarly, the result of rotating an object that is centered at the origin is different from the result of rotating an object that has been moved away from the origin.

The `MatrixOrder` enumeration, which is an argument to the transformation methods, represents the transformation order. It has two values: `Append` and `Prepend`.

Let's write an application to see how transformation order works. We create a Windows application and add a `MainMenu` control and three menu items to the form. The `MatrixOrder` class is defined in the `System.Drawing.Drawing2D` namespace, so we also add a reference to this namespace.

Listing 10.22 draws a rectangle before and after applying a Scale → Rotate → Translate transformation sequence.

LISTING 10.22: Scale → Rotate → Translate transformation order

```
private void First_Click(object sender,
    System.EventArgs e)
{
    // Create a Graphics object
    Graphics g = this.CreateGraphics();
    g.Clear(this.BackColor);
    // Create a rectangle
    Rectangle rect =
        new Rectangle(20, 20, 100, 100);
    // Create a solid brush
    SolidBrush brush =
        new SolidBrush(Color.Red);
    // Fill rectangle
    g.FillRectangle(brush, rect);
    // Scale
    g.ScaleTransform(1.75f, 0.5f);
    // Rotate
    g.RotateTransform(45.0f, MatrixOrder.Append);
    // Translate
    g.TranslateTransform(150.0f, 50.0f,
        MatrixOrder.Append);
    // Fill rectangle again
    g.FillRectangle(brush, rect);
    // Dispose of objects
    brush.Dispose();
    g.Dispose();
}
```

Figure 10.33 shows the output from Listing 10.22. The original rectangle is in the upper left; on the lower right is the rectangle after composite transformation.

Now let's change the order of transformation to Translate → Rotate → Scale with `Append`, as shown in Listing 10.23.

LISTING 10.23: Translate → Rotate → Scale transformation order with `Append`

```
private void Second_Click(object sender,
    System.EventArgs e)
{
    // Create a Graphics object
    Graphics g = this.CreateGraphics();
```

continues

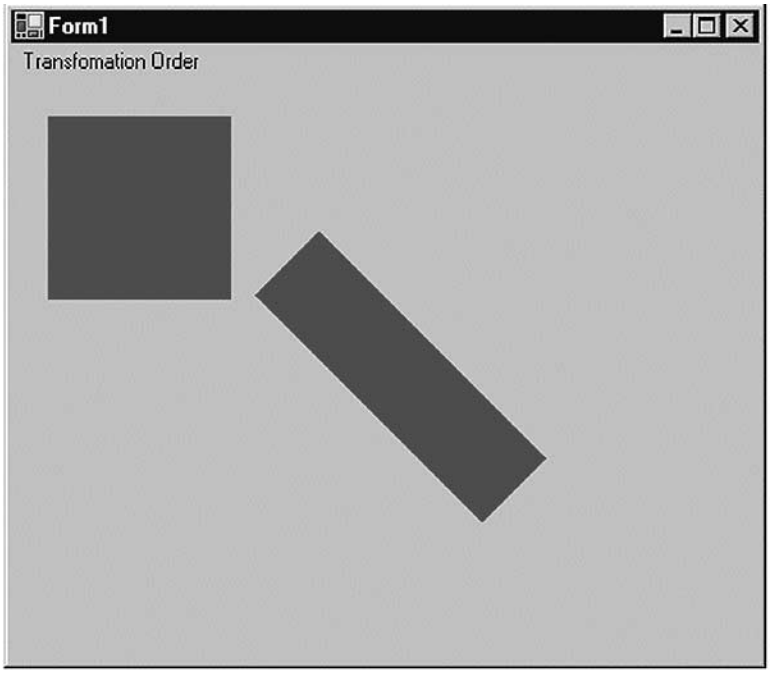

FIGURE 10.33: Scale → Rotate → Translate composite transformation

```
        g.Clear(this.BackColor);
        // Create a rectangle
        Rectangle rect =
            new Rectangle(20, 20, 100, 100);
        // Create a solid brush
        SolidBrush brush =
            new SolidBrush(Color.Red);
        // Fill rectangle
        g.FillRectangle(brush, rect);
        // Translate
        g.TranslateTransform(100.0f, 50.0f,
            MatrixOrder.Append);
        // Scale
        g.ScaleTransform(1.75f, 0.5f);
        // Rotate
        g.RotateTransform(45.0f,
            MatrixOrder.Append);
        // Fill rectangle again
        g.FillRectangle(brush, rect);
        // Dispose of objects
        brush.Dispose();
        g.Dispose();
    }
```

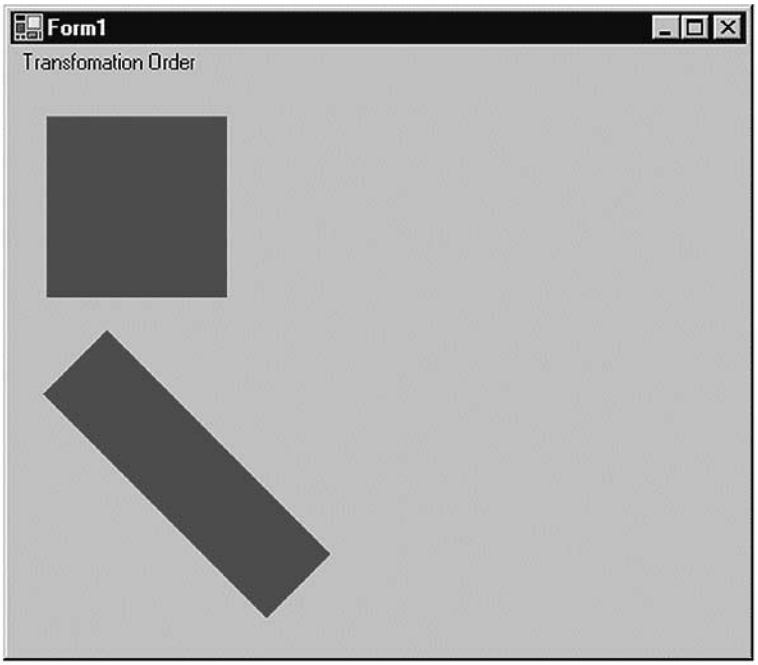

FIGURE 10.34: Translate → Rotate → Scale composite transformation with Append

Figure 10.34 shows the output from Listing 10.23. The original rectangle is in the same place, but the transformed rectangle has moved.

Now let's keep the code from Listing 10.23 and change only the matrix transformation order from `Append` to `Prepend`, as shown in Listing 10.24.

LISTING 10.24: Translate → Rotate → Scale transformation order with `Prepend`

```
private void Third_Click(object sender,
    System.EventArgs e)
{
    // Create a Graphics object
    Graphics g = this.CreateGraphics();
    g.Clear(this.BackColor);
    // Create a rectangle
    Rectangle rect =
        new Rectangle(20, 20, 100, 100);
    // Create a solid brush
    SolidBrush brush =
        new SolidBrush(Color.Red);
```

continues

```
        // Fill rectangle
        g.FillRectangle(brush, rect);
        // Translate
        g.TranslateTransform(100.0f, 50.0f,
            MatrixOrder.Prepend);
        // Rotate
        g.RotateTransform(45.0f,
            MatrixOrder.Prepend);
        // Scale
        g.ScaleTransform(1.75f, 0.5f);
        // Fill rectangle again
        g.FillRectangle(brush, rect);
        // Dispose of objects
        brush.Dispose();
        g.Dispose();
    }
```

The new output is shown in Figure 10.35. The matrix order affects the result.

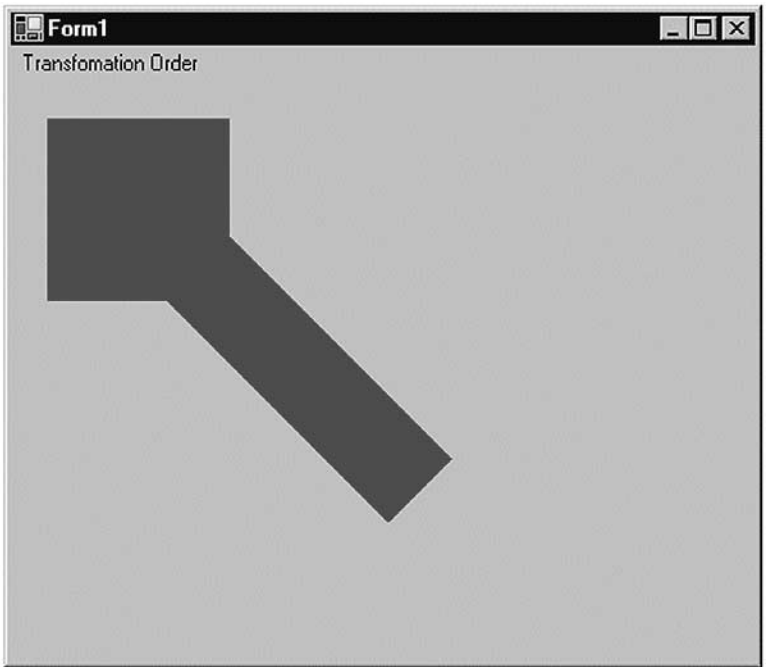

FIGURE 10.35: Translate → Rotate → Scale composite transformation with `Prepend`

SUMMARY

In this chapter we first discussed the basics of transformation, coordinate systems, the role of coordinate systems in the transformation process, and transformation functionality. We learned

- How to distinguish among global, local, and composite transformations
- How to use the `Graphics` class transformations in applications
- How to translate, scale, shear, and rotate graphics objects

Matrices play a vital role in transformation. We can customize the transformation process and its variables by creating and applying a transformation matrix. This chapter showed

- How to use the `Matrix` and `ColorMatrix` classes, and their role in transformation
- How to use the matrix operations for image processing, including translation, scaling, shearing, and rotation
- How to use recoloring and color transformation to manipulate the colors of graphics objects
- How to perform color transformations

Transformations can be applied not only to graphics images and objects, but also to text strings. Drawing vertical or skewed text is one example of text transformation. This chapter explained how to transform text.

Printing also plays an important part in GDI+. In Chapter 11 you will learn various components of the `System.Drawing.Printing` namespace and how to use them.