CMPE2800 – Working with the GDI+

The GDI+ is a newer form of the GDI (Graphics Device Interface) that supports many improvements:

- Supports GDI objects that are easy to modify (and are more advanced)
- Supports ARGB color
- Supports modifiable drawing paths
- Supports composite affine matrix transformations (scaling, translation and rotation)

GDI+ became natively available starting with Windows XP, and as a redistributable on other windows platforms. In C# applications, GDI+ is readily available in the `System.Drawing` and `System.Drawing.Drawing2D` namespaces.

Understanding the basics of GDI+ makes an excellent staging point for understanding many 3D programming concepts.

The Graphics Class

Drawing is principally done through the `Graphics` class. Any object that represents a window can generate a `Graphics` object on demand with a call to `CreateGraphics()`. For example, you can create a `Graphics` object for a form at any time with the following code (executed within a form member function):

```
Graphics gr = CreateGraphics();
```

The `Graphics` object represents the pixels (indirectly) in a surface, and provides the methods that modify them. It is through the `Graphics` object that GDI+ drawing and other operations are performed. In the case of a windows form, the `Graphics` object will represent the *client area* of the window.

Note that the `Paint` handler for a window will provide a `Graphics` object for you. In this particular case, you do not need to create a `Graphics` object. The `Paint` event hander is automatically called when the client area needs repainting, and a `Graphics` object is passed to you. This occurs when any portion of the window becomes invalid. A similar mechanism will be observed when we cover printing.

Pens

The GDI+ requires that you specify a pen for drawing any primitive that uses lines. The `Pen` class encapsulates pen attributes, including color, thickness, and end cap style. The simplest `Pen` constructor accepts a `Color` object, which describes (A)RGB values:

```
Pen p = new Pen(Color.Red);
```

Or

```
Pen p = new Pen(Color.FromArgb (255, 0, 0));
```

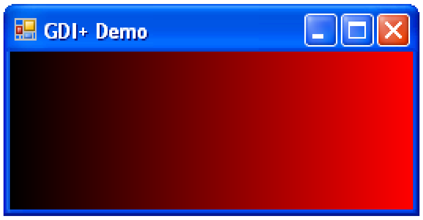
The arguments for `Color` are `A`, `R`, `G`, `B`, where each component is an integer from `0 - 255`. The alpha component specifies the opacity of the color. `0` means completely transparent and `255` means completely opaque. The alpha component determines the weight for blending the source color (the color being drawn) into the destination color (the color that already exists in the surface). GDI+ uses the following formula to determine the destination color when blending:

```
displayColor = sourceColor × alpha / 255 + backgroundColor × (255 - alpha) / 255
```

The following example shows how a `Pen` can be reprogrammed for color:

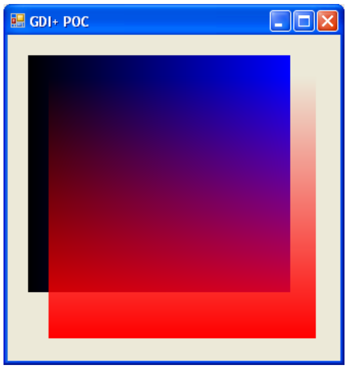
Note: Color components can't be directly modified!

```
Graphics gr = CreateGraphics();

Pen p = new Pen(Color.FromArgb (0, 0, 0));
for (int i = 0; i < 256; ++i)
{
    p.Color = Color.FromArgb(i, 0, 0);
    gr.DrawLine(p, i, 0, i, Height);
}
```



Transparency can be included by specifying a useful alpha component:

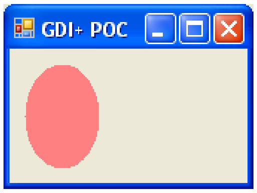
```
// create a graphics object
Graphics gr = CreateGraphics();

// blue pen
Pen BluePen = new Pen(Color.FromArgb(0, 0, 255));

// red pen, play with alpha
Pen RedPen = new Pen(Color.FromArgb(255, 255, 0, 0));

for (int i = 255; i >= 0; --i)
{
    // reprogram the color of the blue pen
    BluePen.Color = Color.FromArgb(0, 0, i);

    // draw a solid color fade
    gr.DrawLine(BluePen, new Point (20 + i, 20), new Point (20 + i, 250));
    //or gr.DrawLine(BluePen, 20 + i, 20, 20 + i, 250);
}

for (int i = 255; i >= 0; --i)
{
    // reprogram alpha in red pen
    RedPen.Color = Color.FromArgb(i, 255, 0, 0);

    // draw an alpha color fade
    gr.DrawLine(RedPen, new Point(40, 40 + i), new Point(300, 40 + i));
}
```



Brushes

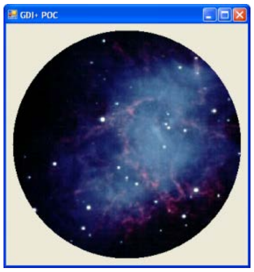
In GDI+ filled shapes are filled with brushes. A brush is similar to a pen in creation:

```
// create a graphics object
Graphics gr = CreateGraphics();

// create a solid brush
SolidBrush br = new SolidBrush(Color.FromArgb(255, 128, 128));

// fill an ellipse with the solid brush
gr.FillEllipse(br, 10, 10, 50, 70);
```



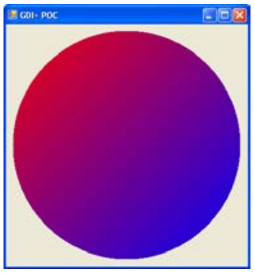
In GDI+ you have a choice of brush types, including solid brushes (as seen above), textured brushes:

```
// create a graphics object
Graphics gr = CreateGraphics();

// create a texture brush
TextureBrush br = new TextureBrush(Image.FromFile ("..\\..\\n1952.jpg"));

// fill an ellipse with the texture brush
gr.FillEllipse(br, 10, 10, 350, 350);
```



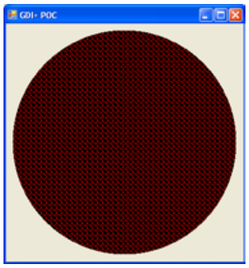
LinearGradientBrush:

```
// create a graphics object
Graphics gr = CreateGraphics();

// create a linear gradient brush
LinearGradientBrush br = new LinearGradientBrush(
    new PointF(0, 0),        // first point position
    new PointF(360, 360),    // second point position
    Color.Red,               // first point color
    Color.Blue);             // second point color

// fill an ellipse with the lg brush
gr.FillEllipse(br, 10, 10, 350, 350);
```



HatchBrush (There are 50+ hatch brush styles):

```
// create a graphics object
Graphics gr = CreateGraphics();

// create a hatch brush
HatchBrush br = new HatchBrush (HatchStyle.Shingle, Color.Red);

// fill an ellipse with the hatch brush
gr.FillEllipse(br, 10, 10, 350, 350);
```



If you wish to explore further, the `System::Drawing::Drawing2D` namespace contains some very exciting things.

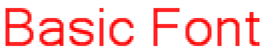
You should also research the `Graphics` object to find out what drawing abilities you now have at your disposal!

Fonts

Drawing text in GDI+ requires the use of fonts. Font objects are created from the `Font` class, and there are many different ways to describe the type of font you are creating. The `Font` constructor has 13 different forms, but principally a face name or `FontFamily` is used to create the font.

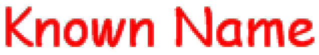
The following examples illustrate the use of some of the forms:

```
// create the most basic sans font, 20 pt
Font foo = new Font(FontFamily.GenericSansSerif, 20);
// draw some text with a red brush
e.Graphics.DrawString("Basic Font", foo, new SolidBrush(Color.Red), 10, 150);
```



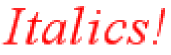
Basic Font

```
// create a font from a known face name, 20 pt
Font foo = new Font("Comic Sans MS", 20);
// draw some text with a red brush
e.Graphics.DrawString("Known Name", foo, new SolidBrush(Color.Red), 10, 150);
```



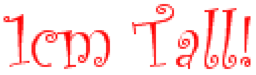
Known Name

```
// create a basic font with a style, 20 pt
Font foo = new Font(FontFamily.GenericSerif, 20, FontStyle.Italic);
// draw some text with a red brush
e.Graphics.DrawString("Italics!", foo, new SolidBrush(Color.Red), 10, 150);
```



Italics!

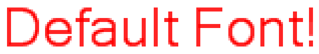
```
// create a font with a specific size (not point)
Font foo = new Font("Curlz MT", 10, GraphicsUnit.Millimeter);
// draw some text with a red brush
e.Graphics.DrawString("1cm Tall!", foo, new SolidBrush(Color.Red), 10, 150);
```



1cm Tall!

It is best not to make assumptions about what fonts are installed on a user's machine. In fact, you can only be guaranteed that there are four fonts installed. Use the generic font types, install your own, or enumerate the fonts available on the system. Fortunately, if you specify a font that does not exist, the system will fall back to a generic:

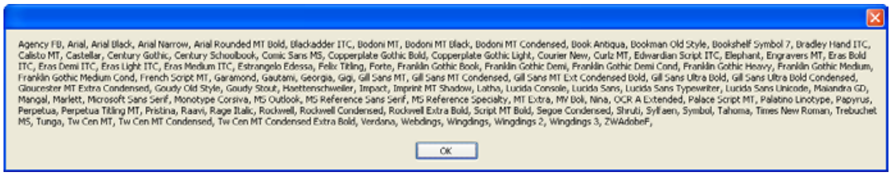
```
// create a font that does not exist
Font foo = new Font("This Font Does Not Exist!", 20);
// draw some text with a red brush
e.Graphics.DrawString("Default Font!", foo, new SolidBrush(Color.Red), 10, 150);
```



Default Font!

You can enumerate the fonts available on the system by calling `FontFamily.GetFamilies()`:

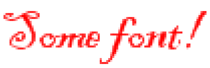
```
// create a graphics object
Graphics gr = CreateGraphics();

// get array of all fonts available
FontFamily[] ff = FontFamily.GetFamilies(gr);

// build string of all fonts available
string temp = "";
foreach (FontFamily f in ff)
    temp += f.GetName(0) + ", ";

MessageBox.Show(temp);
```



Agency FB, Arial, Arial Black, Arial Narrow, Arial Rounded MT Bold, Blackadder ITC, Bodoni MT, Bodoni MT Black, Bodoni MT Condensed, Book Antiqua, Bookman Old Style, Bookshelf Symbol 7, Bradley Hand ITC, Calisto MT, Castellar, Century Gothic, Century Schoolbook, Comic Sans MS, Copperplate Gothic Bold, Copperplate Gothic Light, Courier New, Curlz MT, Edwardian Script ITC, Elephant, Engravers MT, Eras Bold ITC, Eras Demi ITC, Eras Light ITC, Eras Medium ITC, Estrangelo Edessa, Felix Titling, Forte, Franklin Gothic Book, Franklin Gothic Demi, Franklin Gothic Demi Cond, Franklin Gothic Heavy, Franklin Gothic Medium, Franklin Gothic Medium Cond, French Script MT, Garamond, Gautami, Georgia, Gigi, Gill Sans MT, Gill Sans MT Condensed, Gill Sans MT Ext Condensed Bold, Gill Sans Ultra Bold, Gill Sans Ultra Bold Condensed, Gloucester MT Extra Condensed, Goudy Old Style, Goudy Stout, Haettenschweiler, Impact, Imprint MT Shadow, Latha, Lucida Console, Lucida Sans, Lucida Sans Typewriter, Lucida Sans Unicode, Maiandra GD, Mangal, Marlett, Microsoft Sans Serif, Monotype Corsiva, MS Outlook, MS Reference Sans Serif, MS Reference Specialty, MT Extra, MV Boli, Nina, OCR A Extended, Palace Script MT, Palatino Linotype, Papyrus, Perpetua, Perpetua Titling MT, Pristina, Raavi, Rage Italic, Rockwell, Rockwell Condensed, Rockwell Extra Bold, Script MT Bold, Segoe Condensed, Shruti, Sylfaen, Symbol, Tahoma, Times New Roman, Trebuchet MS, Tunga, Tw Cen MT, Tw Cen MT Condensed, Tw Cen MT Condensed Extra Bold, Verdana, Webdings, Wingdings, Wingdings 2, Wingdings 3, ZWAdobeF,

Once you have the array of `FontFamily`, you may use any index to create a font.

```
// build a font from the available list
Font foo = new Font(ff[5], 20);
// draw some text with a red brush
gr.DrawString("Some font!", foo, new SolidBrush(Color.Red), 10, 150);
```



Some font!

Images

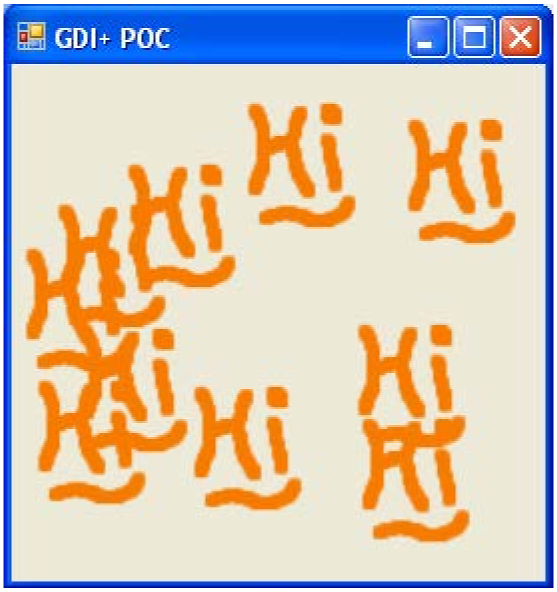
GDI+ also supports fairly advanced image support. Images are managed through the `Image` class.

Loading and displaying an image is quite simple:

```
// create a graphics object
Graphics gr = CreateGraphics();

// load image
Image im = Image.FromFile(@"..\..\hi.png");

// create a random number generator
Random rnd = new Random ();

// draw image 10 times in random locations
for (int i = 0; i < 10; i++)
{
    gr.DrawImage(im, rnd.Next(200), rnd.Next(200));
}
```

If the source image contains transparency information, like the `.png` format does, then `DrawImage` will automatically draw the image with transparency.

You will not normally load images in this way. Normally, images are loaded into an `ImageList` control at design time, or as a resource.



Sew this on your pillow:

DO NOT MAKE GDI RESOURCES MEMBERS!

(Unless you know what you are doing!)

Double-Buffering in C#

In certain situations, drawing GDI+ output directly to the front buffer (the screen) can cause problems. For example, if you are rendering a complex set of layered shapes, significant flicker could occur as each layered object is drawn then partially obscured by the layers on top. The way around this is to draw all output onto an off-screen surface, or back-buffer. A back-buffer is basically a representation of the front buffer that exists purely in memory. Once the complete scene is rendered to the back-buffer, it can be copied in one shot to the front buffer flicker free.

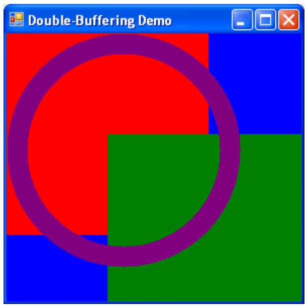
This whole process is known as 'back-buffering' or 'double-buffering'. Double-buffering is usually used when an application does frequent updates to the screen, as in animation. This also includes the nasty issue of clearing the old image so that the new frame can be drawn. So necessary is back-buffering to Direct3D, it is part of the default behavior.

In GDI+, double buffering support is provided primarily through the `BufferedGraphicsContext` and `BufferedGraphics` classes. These classes work together to provide the back-buffer that drives the double-buffering process.

The following code shows the general form of double-buffering code, assuming '`gr`' is the `Graphics` object reference for the target surface:

```
using (BufferedGraphicsContext bgc = new BufferedGraphicsContext())
{
    // bind a back-buffer to the primary surface, spec size to create as client size
    using (BufferedGraphics bg = bgc.Allocate(gr, this.DisplayRectangle))
    {
        // clear the back-buffer
        bg.Graphics.Clear(Color.FromKnownColor(KnownColor.Control));

        // draw whatever you need to draw
        DrawJunk(bg.Graphics);

        // flip the back-buffer to the primary surface
        bg.Render();
    }
}
```

Your instructor will show you an animated demo that contrasts double-buffering vs. no double-buffering. Leverage the above code for all of your double-buffering needs.

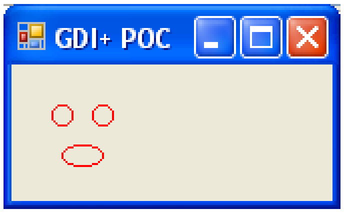


Further investigation of the GDI+ is now your responsibility. The best source of information is probably the web, however, the MSDN site, at Microsoft, has a nice section devoted to GDI+, and it is always updated. This little nugget of encouragement should probably appear at the end of this document, but there was no room, so I'm leaving it here…

Graphics Paths

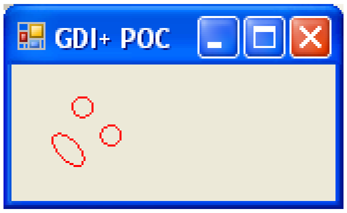
Another nice feature of the GDI+ is the graphics path. A graphics path is a set of drawing instructions in a container. The entire collection of drawing instructions may then be used for any number of operations. This functionality is found in the `System.Drawing.Drawing2D.GraphicsPath` class:

```
// create a graphics object
Graphics gr = CreateGraphics();

// make a pen
Pen p = new Pen(Color.Red);

// build a path, add elements
GraphicsPath gp = new GraphicsPath();
gp.AddEllipse(20, 20, 10, 10);
gp.AddEllipse(40, 20, 10, 10);
gp.AddEllipse(25, 40, 20, 10);

// draw the entire path
gr.DrawPath(p, gp);
```



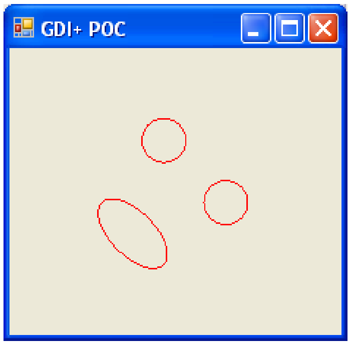
A `GraphicsPath` object may be manipulated as a single unit, to produce rotation, translation and scaling transformations. A `Matrix` object is used to apply the transformation. These elements are critical to 2D and 3D animation and gaming:

```
// create a graphics object
Graphics gr = CreateGraphics();

// make a pen
Pen p = new Pen(Color.Red);

// build a path, add elements
GraphicsPath gp = new GraphicsPath();
gp.AddEllipse(20, 20, 10, 10);
gp.AddEllipse(40, 20, 10, 10);
gp.AddEllipse(25, 40, 20, 10);

// rotate at centre of 'image'
Matrix mat = new Matrix();
mat.RotateAt(45, new PointF(35, 35));

// apply the transform to the path
gp.Transform(mat);

// draw the entire path
gr.DrawPath(p, gp);
```



Multiple transforms may be applied to a path. Matrices and transforms will be discussed later…

```
// rotate at centre of 'image'
Matrix matRot = new Matrix();
matRot.RotateAt(45, new PointF(35, 35));

// scale 3 times bigger
Matrix matScale = new Matrix();
matScale.Scale(3, 3);

// multiply matrices to create additive effect
matScale.Multiply(matRot);

// apply the transform to the path
gp.Transform(matScale);
```



Regions

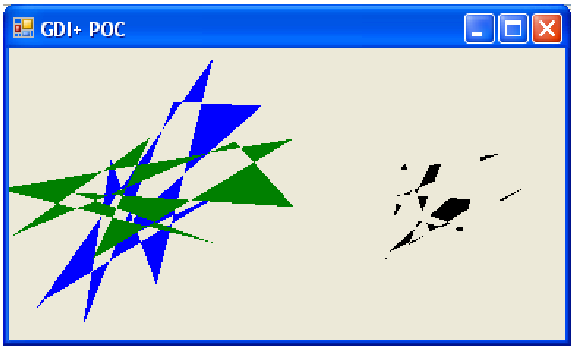
One of the most powerful features of the GDI+ is the `Region`. A region is a description of 2D space that has very useful features:

- Can be constructed as a simple rectangle
- Can be constructed from a path
- Can create intersections, unions, exclusions, or XOR areas of two regions
- Can be transformed

The power of regions will shine when hit-detection for irregular, transformed objects is considered. You will use regions in your final project to perform all hit detection.

The following code shows some of these features in action:

```
// create a graphics object
Graphics gr = CreateGraphics();

// make a pen
Pen p = new Pen(Color.Red);

// build a path, add elements (makes a crazy set of triangles)
GraphicsPath gp = new GraphicsPath();
Random rnd = new Random(3);
PointF[] pts = new PointF[10];
for (int i = 0; i < 10; ++i)
    pts[i] = new PointF(rnd.Next(200), rnd.Next(200));
gp.StartFigure();
gp.AddLines(pts);
gp.CloseFigure();

// create a region from the crazy lines (triangles, now)
Region RegA = new Region(gp);

// fill the region in blue, to show what the region looks like
gr.FillRegion(new SolidBrush(Color.Blue), RegA);

// rotate the path by 45 degrees
Matrix mat = new Matrix();
mat.RotateAt(45, new PointF(100, 100));
gp.Transform(mat);

// build another region out of the rotated path
Region RegB = new Region(gp);

// show the rotated region in green
gr.FillRegion(new SolidBrush(Color.Green), RegB);

// create a 3rd region that is an intersection of the two regions
Region RegC = RegA.Clone(); // start as a copy of first region
RegC.Intersect(RegB);

// translate 200 pixels to the right and draw intersection in black
mat.Reset(); // lazy, recycle old matrix...
mat.Translate(200, 0);
RegC.Transform(mat);
gr.FillRegion(new SolidBrush(Color.Black), RegC);
```



A note on regions

Regions store information about the area they describe internally as an array of bytes. The more complex the region, the more data required to describe it. While understanding the exact mechanics behind this is not critical to using regions, it can be instructive. Consider the size of a simple rectangular region:

```
// this region reports it's size as 36 bytes:
Region A = new Region(new Rectangle(0, 0, 800, 600));
Console.WriteLine(A.GetRegionData().Data.Length.ToString());
```

A region that has been through multiple convoluted operations will be larger:

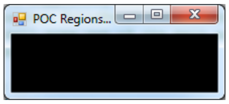
```
Random rnd = new Random ();

// this region reports it's size as 144036 bytes
Region A = new Region(new Rectangle(0, 0, 800, 600));

// punch some holes in the region
for (int i = 0; i < 1000; ++i)
{
    GraphicsPath gp = new GraphicsPath();
    gp.AddEllipse(rnd.Next(-10, 811), rnd.Next(-10, 611), 20, 20);
    A.Exclude(gp);
}

Console.WriteLine(A.GetRegionData().Data.Length.ToString());
```

The internal size of the region is not a huge concern of ours. The problem with complex regions is the time it takes to perform operations on them. The complex region above takes nearly 1 second to render.

Once again, this is not something you will usually need to worry about, as the regions you will be working with will typically be significantly simpler. It would be a good idea to keep this in mind, in the event that you go crazy while doing the hit-detection in your final project.

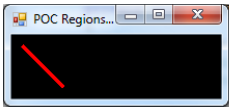
Another thing to note is that regions created from simple lines will appear to not exist:

```
// create a path and add a line to it
GraphicsPath pth = new GraphicsPath();
pth.AddLine(10, 10, 50, 50);

// build a region from the pen, and fill it in red
Region reg = new Region(pth);
bg.Graphics.FillRegion(new SolidBrush(Color.Red), reg);
```



The way around this is to use `Widen` to add area to the path:

```
// create a path and add a line to it
GraphicsPath pth = new GraphicsPath();
pth.AddLine(10, 10, 50, 50);

// add an outline to the path
pth.Widen(new Pen(Color.Green, 3));

// build a region from the pen, and fill it in red
Region reg = new Region(pth);
bg.Graphics.FillRegion(new SolidBrush(Color.Red), reg);
```



GDI+ Coordinate Systems

GDI+ uses three coordinate spaces: world, page, and device. World coordinates are what you specify as arguments to graphics methods. Page coordinates are used by a drawing surface, such as the main form. Device coordinates are used by a physical device, such as your monitor.

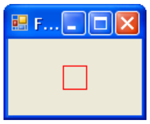
World coordinates go through two transformations before they are used in actual drawing. The first transformation, called the 'world transformation', transforms world coordinates into page coordinates. The world transformation is held in the `Transform` property of the `Graphics` class. The second transformation, called the 'page transformation', transforms page coordinates into device coordinates. The page transformation is modified through the `PageUnit` and `PageScale` properties of the `Graphics` class.

Page coordinate space *always* has its origin in the upper-left corner of the drawing area.

By default, page coordinates are mapped to pixels, so page coordinates and device coordinates are the same. In fact, if you use the default world transform you will find that world, page, and device coordinates are the same. We will experiment with the page transform when we do some printing. For now, we will not concern ourselves with the page transform.

The world transform may be used to provide some nice features in GDI+. For example, the origin of the client area can be centered in the window if the world transform is set to translate all output by half the height and half the width of the client area:

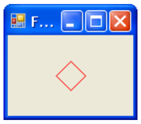
```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    e.Graphics.TranslateTransform(ClientSize.Width / 2, ClientSize.Height / 2);
    e.Graphics.DrawRectangle(new Pen(Color.Red), -10, -10, 20, 20);
}
```



The `TranslateTransform` method of the `Graphics` object will apply the specified translation to the world transformation. The transformation is held in a 3 * 3 affine matrix, represented by the `System.Drawing.Drawing2D.Matrix` class.

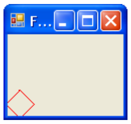
There are other transforms that you may directly apply to the world transform, including rotation and scaling:

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    e.Graphics.TranslateTransform(ClientSize.Width / 2, ClientSize.Height / 2);
    e.Graphics.RotateTransform(45, System.Drawing.Drawing2D.MatrixOrder.Prepend);
    e.Graphics.DrawRectangle(new Pen(Color.Red), -10, -10, 20, 20);
}
```



A matrix is able to hold the result of multiple transforms, by multiplying the matrices that represent each transform. The resultant matrix is said to contain a composite transform. It is imperative that you understand that matrix multiplication is not commutative! If, for example, the above code was reordered, the result would not be the same:

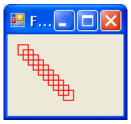
```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    e.Graphics.RotateTransform(45, System.Drawing.Drawing2D.MatrixOrder.Prepend);
    e.Graphics.TranslateTransform(ClientSize.Width / 2, ClientSize.Height / 2);
    e.Graphics.DrawRectangle(new Pen(Color.Red), -10, -10, 20, 20);
}
```



In GDI+, matrices are multiplied left-to-right. By default, these operations prepend the matrix in the multiplication order. This means that a `RotateTransform` call followed by a `TranslateTransform` call actually applies the translation to the composite transform first. You are able to override the multiplication order by using alternate overrides for each function, specifying the order.

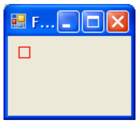
Note that repeated transforms will cause composite transforms:

```
    for (int i = 0; i < 10; i++)
    {
        e.Graphics.TranslateTransform(5, 5);
        e.Graphics.DrawRectangle(new Pen(Color.Red), 5, 5, 10, 10);
    }
```



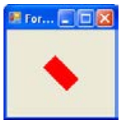
To prevent cumulative effects in your transforms, you may use the `ResetTransform` method to load the identity matrix back into the transform property. The identity matrix is the same as multiplying a number by 1; it has no effect:

```
    for (int i = 0; i < 10; i++)
    {
        e.Graphics.TranslateTransform(5, 5);
        e.Graphics.DrawRectangle(new Pen(Color.Red), 5, 5, 10, 10);
        e.Graphics.ResetTransform();
    }
```



When you apply matrix transforms to a graphics path, the transforms are applied to the world coordinates of the path items separately. The resultant coordinates are then used as world coordinates for the graphics world transform:

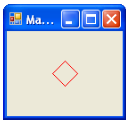
```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    // move origin to middle of client area
    e.Graphics.TranslateTransform(ClientSize.Width / 2, ClientSize.Height / 2);
    e.Graphics.RotateTransform(45);

    // add a path with it's own transform
    GraphicsPath gp = new GraphicsPath();
    gp.AddRectangle(new Rectangle(-10, -10, 20, 20));
    Matrix mPath = new Matrix ();
    mPath.Scale(2, 1);
    gp.Transform(mPath);
    e.Graphics.FillPath(new SolidBrush(Color.Red), gp);
}
```



Only to confuse matters a little, when you apply a transform to a `GraphicsPath` with the `Transform` method, the matrix supplied is appended in the multiplication. This, of course, is the reverse of how the transform methods work in the `Graphics` object:

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    GraphicsPath gp = new GraphicsPath();
    gp.AddRectangle(new Rectangle(-10, -10, 20, 20));

    Matrix matRot = new Matrix();
    matRot.Rotate(45);
    gp.Transform(matRot);

    Matrix matTrans = new Matrix();
    matTrans.Translate(ClientSize.Width / 2, ClientSize.Height / 2);
    gp.Transform(matTrans);

    e.Graphics.DrawPath(new Pen(Color.Red), gp);
}
```

The rotation will happen first!



Some instructors, Herb, would suggest that you always supply the `MatrixOrder.Append` argument in transform calls with `Graphics`. This will provide a consistent behavior for for `Graphics` and `GraphicsPath` transform operations. Either way, an understanding is crucial.

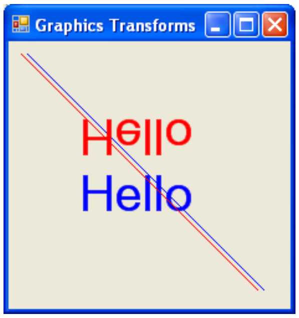
Y-Axis Flipping in GDI+

If you sit and think about it for a while, it's a little disappointing that the Y-axis in the GDI increases downward. This is somewhat contrary to conventional Cartesian geometry and really annoying when you make GDI models for other APIs like Direct3D.

Fortunately, as previously seen, it is possible to apply transforms directly to a `Graphics` object. Applying transforms directly to a `Graphics` object allows for a variety of effects, including axis flipping, scaling, etc...

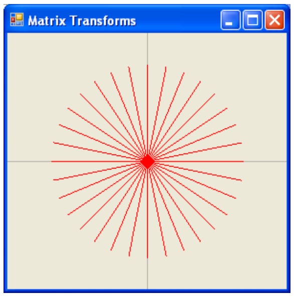
The following code illustrates how to apply a transform to a `Graphics` object that will flip the direction of the Y axis:

```
// show some text and a line with the identity transform:
e.Graphics.DrawString(
    "Hello",
    new Font("Ariel", 30),
    new SolidBrush (Color.Blue), 50, 100);
e.Graphics.DrawLine(new Pen(Color.Blue), 15, 10, 205, 200);

// flip Y axis
Matrix mat = new Matrix(1, 0, 0, -1, 0, 0);
e.Graphics.Transform = mat;

// show some text and a line with the inverted Y axis
e.Graphics.DrawString(
    "Hello",
    new Font("Ariel", 30),
    new SolidBrush(Color.Red), 50, -100);                       // note y coords now neg
e.Graphics.DrawLine(new Pen(Color.Red), 10, -10, 200, -200);    // note y coords now neg
```



Just flipping the Y axis might not be enough. Working with graphics often involves world and model spaces that are centered about the origin. To be truly useful, not only should the Y axis be flipped, but the origin should be moved to the centre of the window. This would make objects drawn around the origin completely visible. The following code shows how to do both tasks:

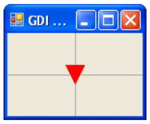
```
// move origin of the window to the centre of the window:
e.Graphics.TranslateTransform(
    (float)ClientSize.Width / 2,
    (float)ClientSize.Height / 2);

// flip Y axis
e.Graphics.Transform.Multiply (new Matrix(1, 0, 0, -1, 0, 0));

// draw cross-hairs in the window
e.Graphics.DrawLine(
    new Pen(Color.FromArgb(128, 128, 128, 128)),
    0 - ClientSize.Width / 2, 0, ClientSize.Width / 2, 0);
e.Graphics.DrawLine(
    new Pen(Color.FromArgb(128, 128, 128, 128)),
    0, 0 - ClientSize.Height / 2, 0, ClientSize.Height / 2);

// draw some lines that radiate out from the origin (use all 4 quads)
for (float rot = 0; rot < Math.PI * 2; rot += (float)(Math.PI / 16))
{
    e.Graphics.DrawLine(
        new Pen(Color.Red),
        0,
        0,
        (float)(Math.Cos(rot) * 100),
        (float)(Math.Sin(rot) * 100));
}
```



Y-Axis Flipping – Another Approach

You will find that working with the GDI is somewhat confusing if you want to produce output that matches the Cartesian coordinate system you are familiar with in mathematics i.e.: X increases to the right, and Y increases upward. This will be confirmed if you attempt to adapt your work to the left-handed 3D world, where Y increases upward, X increases to the right, and Z increases with depth (into the scene). The GDI transforms that you will use in 2D are designed to work with the 'upside down' Y axis. For example, consider the following code (taken from a paint event handler):

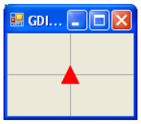
```
// make center of window the origin
e.Graphics.TranslateTransform(ClientRectangle.Width / 2, ClientRectangle.Height / 2);

// model space is triangle with point up (Cartesian coords, anyway)
GraphicsPath gp = new GraphicsPath();
gp.AddLine(0, 10, 10, -10);
gp.AddLine(10, -10, -10, -10);
gp.AddLine(-10, -10, 0, 10);

// presented in world space, centered on origin, point down
e.Graphics.FillPath(new SolidBrush(Color.Red), gp);
```



The first problem encountered is the incongruity between Cartesian model space and the model's untransformed appearance when rendered. This, of course, is caused by generating the model using Cartesian coordinates, and rendering with the GDI. The issue is perspective, not so much a fault, but that Y axis will cause some angst. The triangle, in model space, should have the point upwards. Because up is actually down to the GDI, the triangle appears inverted.

An 'easy' way to correct this would be to apply a 180° rotation to the model to make it appear in the correct orientation:

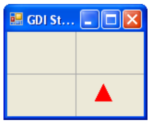
```
Matrix ModelRot = new Matrix();
ModelRot.Rotate(180);
gp.Transform(ModelRot);
```



This allows the model to be generated in coordinates that make sense. Of course, you could just flip the sign of all the model Y axis coordinates instead…

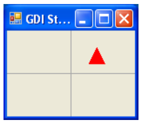
The next issue concerns transforms. Consider a simple transform:

```
Matrix MatTrans = new Matrix();
MatTrans.Translate(30, 20);
gp.Transform(MatTrans);
```



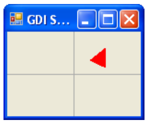
A translation that specifies a positive Y displacement causes the translated object to move downwards. This can be avoided by changing the sign of the Y component in the translation:

```
Matrix MatTrans = new Matrix();
MatTrans.Translate(30, -20);
gp.Transform(MatTrans);
```



Rotation, luckily, works just fine, as long as you remember to rotate first:

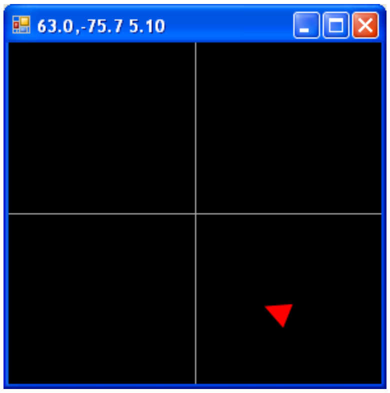
```
// create the rotation matrix
Matrix MatRot = new Matrix();
MatRot.Rotate(25);

// create the translation matrix
Matrix MatTrans = new Matrix();
MatTrans.Translate(30, -20);

// create composite transform (prepend actually redundant)
MatTrans.Multiply(MatRot, MatrixOrder.Prepend);

// apply final transform
gp.Transform(MatTrans);
```



Be advised that the GDI matrix transforms specify angles in degrees, while Direct3D (and just about everything else) uses radians!

The following code shows how the little triangle from above could be 'driven around' in a simple application:

```
public partial class Form1 : Form
{
    // triangle position and heading values
    double m_dTriangleXPos = 0, m_dTriangleYPos = 0, m_dTriangleHeading = 0;

    // key state information
    bool m_bUp = false, m_bLeft = false, m_bRight = false, m_bDown = false;

    public Form1()
    {
        InitializeComponent();
    }

    // this function updates the position of the triangle and renders the result
    private void UI_TIM_Render_Tick(object sender, EventArgs e)
    {
        // rendering timer has gone off, so process turn and redraw
        if (m_bLeft)
            m_dTriangleHeading -= 0.1;
        if (m_bRight)
            m_dTriangleHeading += 0.1;

        if (m_bUp)
        {
            m_dTriangleXPos += Math.Sin(m_dTriangleHeading) * 5;
            m_dTriangleYPos += Math.Cos(m_dTriangleHeading) * 5;
        }
        if (m_bDown)
        {
            m_dTriangleXPos += Math.Sin(m_dTriangleHeading + Math.PI) * 5;
            m_dTriangleYPos += Math.Cos(m_dTriangleHeading + Math.PI) * 5;
        }

        // actually draw it...
        Render();
    }
    private void Render()
    {
        Graphics gr = CreateGraphics();

        BufferedGraphicsContext bgc = new BufferedGraphicsContext();
        BufferedGraphics bg = bgc.Allocate(gr, DisplayRectangle);
        bg.Graphics.Clear(Color.Black);

        // make center of window the origin
        bg.Graphics.TranslateTransform(ClientRectangle.Width / 2, ClientRectangle.Height / 2);
        bg.Graphics.DrawLine(new Pen(Color.DarkGray), 0, -ClientRectangle.Height / 2, 0, ClientRectangle.Height / 2);
        bg.Graphics.DrawLine(new Pen(Color.DarkGray), -ClientRectangle.Width / 2, 0, ClientRectangle.Width / 2, 0);

        // model space is triangle with point up (Cartesian coord, anyway)
        GraphicsPath gp = new GraphicsPath();
        gp.AddLine(0, 10, 10, -10);
        gp.AddLine(10, -10, -10, -10);
        gp.AddLine(-10, -10, 0, 10);
        Matrix ModelRot = new Matrix();
        ModelRot.Rotate(180);
        gp.Transform(ModelRot);

        // create the rotation matrix
        Matrix MatRot = new Matrix();
        MatRot.Rotate((float)(m_dTriangleHeading * 360 / (Math.PI * 2)));
        gp.Transform(MatRot);

        // create the translation matrix
        Matrix MatTrans = new Matrix();
        MatTrans.Translate((float)m_dTriangleXPos, -(float)m_dTriangleYPos);
        gp.Transform(MatTrans);

        // show in screen space
        bg.Graphics.FillPath(new SolidBrush(Color.Red), gp);

        // flip back-buffer to front buffer
        bg.Render();

        // show positional information in main window
        Text = String.Format("{0:f1},{1:f1} {2:f2}", m_dTriangleXPos, m_dTriangleYPos, m_dTriangleHeading);
    }

    private void Form1_KeyUp(object sender, KeyEventArgs e)
    {
        if (e.KeyCode == Keys.Up)
            m_bUp = false;
        if (e.KeyCode == Keys.Down)
            m_bDown = false;
        if (e.KeyCode == Keys.Left)
            m_bLeft = false;
        if (e.KeyCode == Keys.Right)
            m_bRight = false;
    }

    private void Form1_KeyDown(object sender, KeyEventArgs e)
    {
        if (e.KeyCode == Keys.Up)
            m_bUp = true;
        if (e.KeyCode == Keys.Down)
            m_bDown = true;
        if (e.KeyCode == Keys.Left)
            m_bLeft = true;
        if (e.KeyCode == Keys.Right)
            m_bRight = true;
    }
}
```



63.0,-75.7 5.10

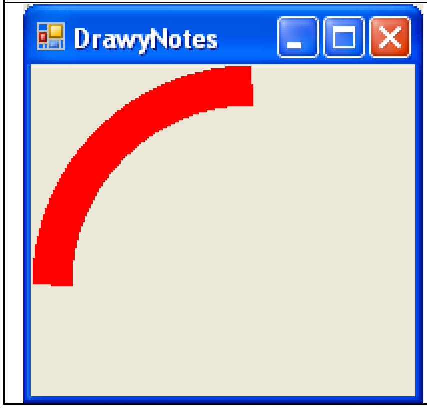
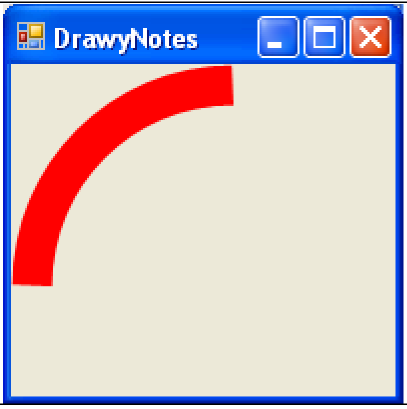
It is imperative that your applications maintain real data. If you need to overcome any oddities in the display process, compensate during the display process. Don't compromise your data.

Miscellaneous GDI+ Topics

Antialiasing

You may add antialiasing to your GDI+ drawings by setting the '`SmoothingMode`' property of a `Graphics` object to '`AntiAlias`'. This will get rid of the 'jaggies' in the rendered output.

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    e.Graphics.SmoothingMode = System.Drawing.Drawing2D.SmoothingMode.AntiAlias;
    e.Graphics.DrawArc(new Pen(Color.Red, 20), new Rectangle(10, 10, 200, 200), 180, 90);
}
```

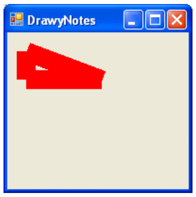
Before	After
	

There are other smoothing options, but this one is the best for a balance between high quality output and performance.

Pen Mitering

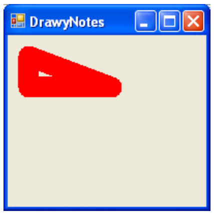
In certain circumstances you may want to change how joined lines are mitered. In the following example, the default behaviour for the mitering can be observed:

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    Point[] pts = new Point[] {
        new Point (20, 20),
        new Point (100, 50),
        new Point (20, 50) };

    // draw, with default square-cap pen
    Pen p = new Pen(Color.Red, 20);
    e.Graphics.DrawLine(p, pts[0], pts[1]);
    e.Graphics.DrawLine(p, pts[1], pts[2]);
    e.Graphics.DrawLine(p, pts[2], pts[0]);
}
```



You have the option to change how pen caps are drawn by calling the '`SetLineCap`' method in your pen instance:

```
    Pen p = new Pen(Color.Red, 20);
    p.SetLineCap(
        System.Drawing.Drawing2D.LineCap.Round,
        System.Drawing.Drawing2D.LineCap.Round,
        System.Drawing.Drawing2D.DashCap.Round);
```


DrawyNotes