Smoother Graphics – Taking Control of Painting the Screen

It is very likely that by now you've tried something that made your game run rather slow. Perhaps you tried to use an image with a transparent background, or had a gazillion objects moving on the window at the same time. And…things slooooowwwwed down…you ended up with a slide show rather than a smooth running game.

So how to make the game run smoother … is smoother actually a word?

One likely reason your game was bogging down is that trying to animate a sprite using a *PictureBox* is really, really, really slow. *PictureBoxes* simply weren't designed for sprite animation, they were designed to…surprise…display pictures.

One of the first steps we can take to have smoother sprite animation is to take control of drawing (actually referred to as painting) the window ourselves … which immediately begs the question "how does the window get painted?"

Hopefully by now you are familiar with the concept of events and event handlers. For example, you should now be familiar with the *KeyDown* event: this event is triggered every time a keyboard key is pushed down. You have successfully created a handler for this event by going to the properties tab for the *Form* and double-clicking on the *KeyDown* event line. You've also played with timers and have created a handler for the timer *Tick* event.

One great feature of event handlers is each provides parameters that contain essential information pertaining to the event. Again looking at the example of the *KeyDown* event, the handler provides two parameters:

```
private void Form1_KeyDown(object sender, KeyEventArgs e)
```

1. *object sender*
2. *KeyEventArgs e*

You've used the *KeyEventArgs e* parameter to determine which key was pressed by referencing the *KeyCode* property like this:

```
if (e.KeyCode == Keys.Down)
```

Well, there just so happens to be an event for painting the window. This event happens anytime the system determines the window needs to be repainted. Examples of when this will happen are:

- If the window is restored after being minimized
- If something (perhaps another window) was covering or partially covering our window and has been moved

Now those are things over which we really have no control over ***in*** our program…they are things that happen ***to*** our program. What we need is ***in*** our program, some way of saying "we've changed the stuff on the window, please repaint it." Well there is a very easy way to do this: we just have our Form call the *Invalidate()* method.

The *Invalidate* method "invalidates the entire surface of the control and causes the control to be redrawn." Boom, exactly what we need. Let's get to it then.

Smoother Graphics – Taking Control of Painting the Screen

Create a new windows form app project (name it *UserPaintExample*). Hit *F7* to get to the code window.

…now before we go any further, I want you to understand that we are now going to start writing by hand some of the code the IDE has been writing for us. This means you have to type things ***EXACTLY*** the way I show you. It isn't much, but you do have to pay attention to detail and get everything perfect.

The first thing we need to do is tell Windows that we are taking control of painting our window, and are providing our own *Paint* event handler. In your *Form1* constructor, enter the following:

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();

        // Tell windows we qre taking responsibility for painting the screen
        SetStyle(ControlStyles.AllPaintingInWmPaint | ControlStyles.UserPaint | ControlStyles.OptimizedDoubleBuffer, true);
        this.Paint += new PaintEventHandler(Form1_Paint);
    }
}
```

You will notice the red squiggly line under *Form1_Paint* in the code above … that is the name of the event handler method and is okay because we haven't written it yet.

There are some things in the above code that may look like hieroglyphics:

- *Control.Styles*.*UserPaint*: if *true* the form is responsible for painting itself rather than the system.
- *Control.Styles*.*AllPaintingInWmPaint*: if this and *UserPaint* are both *true*, the background is not erased in order to reduce flicker.
- *Control.Styles*.*OptimizedDoubleBuffer*: the form is drawn using "double buffering" … the drawing is actually done to a memory buffer (which is quite fast), then the buffer is blasted to the screen as one. Much faster.

At this point you don't really need a deep understanding of the above items. It is sufficient to understand that these settings/styles are what enable us to have faster graphics. Double buffering the graphics and not erasing the background all the time are huge wins for us.

Smoother Graphics – Taking Control of Painting the Screen

Let's get rid of the red squiggly line under *Form1_Paint*: enter the following as a new method after the constructor:

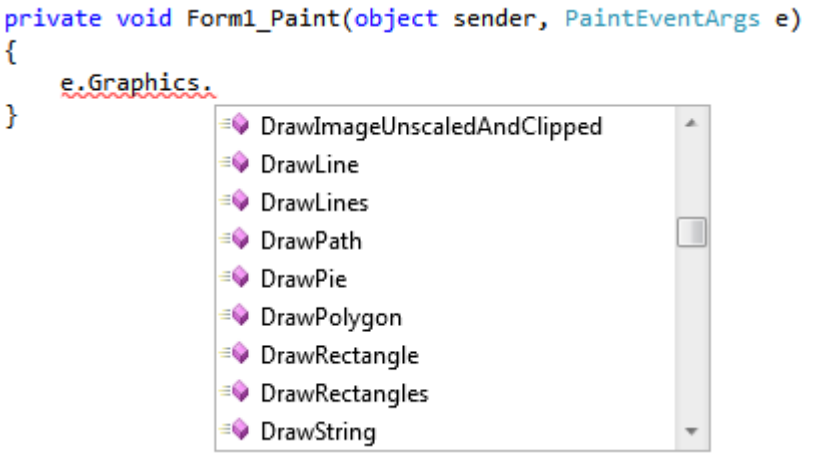
```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();

        // Tell windows we qre taking responsibility for painting the screen
        SetStyle(ControlStyles.AllPaintingInWmPaint | ControlStyles.UserPaint | ControlStyles.OptimizedDoubleBuffer, true);
        this.Paint += new PaintEventHandler(Form1_Paint);
    }

    private void Form1_Paint(object sender, PaintEventArgs e)
    {
    }
}
```

Notice that the red squiggly line under *Form1_Paint* in the constructor is now gone. That is because the event handler now exists. ☺

Also notice the 2nd parameter of the event handler: *PaintEventArgs e*. This gives us access to everything necessary to draw stuff on the screen, because one of the things it provides is the *graphics context* for drawing. Think of the *graphics context* as the canvas we will be drawing and painting on.

Smoother Graphics – Taking Control of Painting the Screen

Just for fun, add a line inside the *Form1_Paint* method and type *e.Graphics.* and the IDE will pop-up the help dialog showing you all the features of the *Graphics* context. Scroll down in it and you will find things like *DrawRectangle*, *DrawPolygon*, *DrawEllipse, among many others:*



These are your key to faster graphics, and are what we will use instead of *PictureBox* to draw sprites. ☺

Go ahead and erase what you just typed (*e.Graphics.*).

Smoother Graphics – Taking Control of Painting the Screen

Now if you recall, on your prior games you likely had a timer running that would move all the *PictureBoxes* on the screen every timer tick. To make things work with our new *Paint* event handler, all we need to do is have our timer tick tell Windows that our screen is "invalidated" and needs to be redrawn. As mentioned before, we do that by calling *Invalidate()*.

Add a timer called *refreshTimer* … only this time instead of dragging and dropping it into the Design window, let's add it by hand in our own code. Don't worry, this is super easy. ☺

We will add a new field to our *Form1* class. Its type will be *Timer* and we'll initialize it in the *Form1* constructor as follows:

- Set its interval to something like 25
- Add a *Tick* event handler called *refreshTimer_Tick*
- And enable the timer.

Here is what it should look like:

```
public partial class Form1 : Form
{
    private Timer refreshTimer;

    public Form1()
    {
        InitializeComponent();

        // Tell windows we qre taking responsibility for painting t
        SetStyle(ControlStyles.AllPaintingInWmPaint | ControlStyles
        this.Paint += new PaintEventHandler(Form1_Paint);

        // Screen refresh timer
        refreshTimer = new Timer();
        refreshTimer.Interval = 25;
        refreshTimer.Tick += new EventHandler(refreshTimer_Tick);
        refreshTimer.Enabled = true;
    }
```

Again you will notice the red squiggly line under *refreshTimer_Tick* which is the event handler method … we haven't written it yet.

Smoother Graphics – Taking Control of Painting the Screen

So … let's write it now. It is super easy and super short; just make sure you get everything exact.

```
public partial class Form1 : Form
{
    private Timer refreshTimer;

    public Form1()
    {
        InitializeComponent();

        // Tell windows we qre taking responsibility for painting the screen
        SetStyle(ControlStyles.AllPaintingInWmPaint | ControlStyles.UserPaint | ControlStyles.OptimizedDoubleBuffer, true);
        this.Paint += new PaintEventHandler(Form1_Paint);

        // Screen refresh timer
        refreshTimer = new Timer();
        refreshTimer.Interval = 25;
        refreshTimer.Tick += new EventHandler(refreshTimer_Tick);
        refreshTimer.Enabled = true;
    }

    private void refreshTimer_Tick(object sender, EventArgs e)
    {
        Invalidate();
    }

    private void Form1_Paint(object sender, PaintEventArgs e)
    {
    }
}
```

I added it right before the *Paint* event handler; doesn't really matter where you put it as long as it's in the *Form1* class. ☺

Right now all it needs to do is tell Windows it is time to repaint the screen by calling *Invalidate()*.

If you build and run your program right now, you should see a blank window appear. That's okay because we're not drawing anything yet!

Next, let's add a ball on the screen. Okay, it won't be a "ball" … we're going to start off very simple and draw a square in the center of the screen.

Smoother Graphics – Taking Control of Painting the Screen

There are a couple things we will need to keep track of for our ball that *PictureBox* did automatically for us:

1. The location of the ball
2. The size of the ball

Let's add some fields to keep track of this info. The location can be a *Point*, the size can be a *Size*. We will initialize each in the constructor. We'll put the location in the center of the screen, and make the size 5 x 5:

```
public partial class Form1 : Form
{
    private Timer refreshTimer;

    private Point loc;
    private Size  size;

    public Form1()
    {
        InitializeComponent();

        // tell windows we are taking responsibility for painting the screen
        SetStyle(ControlStyles.AllPaintingInWmPaint | ControlStyles.UserPaint | C
        this.Paint += new PaintEventHandler(Form1_Paint);

        // initialize the critical ball information
        loc = new Point(ClientRectangle.Width / 2, ClientRectangle.Height / 2);
        size = new Size(5, 5);

        // screen refresh timer
        refreshTimer = new Timer();
```

Excellent! Now, let's draw our square. ☺ Inside our *Paint* event handler, we just need to use the *Graphics* context to draw and fill a rectangle (remember from geometry that a square is just a special rectangle).

Smoother Graphics – Taking Control of Painting the Screen

Add the following code in the *Paint* event handler:

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.DrawRectangle(pen, loc.X, loc.Y, size.Width, size.Height);
    g.FillRectangle(brush, loc.X, loc.Y, size.Width, size.Height);
}
```

First we get a reference to the *Graphics* context with the line `Graphics g = e.Graphics;`

Next we draw the rectangle (creates the outline) and then we fill it (color it in). To do so we have to provide the location and size of the rectangle. Remember, we set the size to (5, 5) which will make this a square.

Also notice the red squiggly lines under the words *pen* and *brush* – this is where things start to get a little different. In the *Graphics* context, we draw with a *pen* and fill with a *brush*. They have the red squiggly lines because we haven't created the *pen* or *brush* yet. Let's do that now.

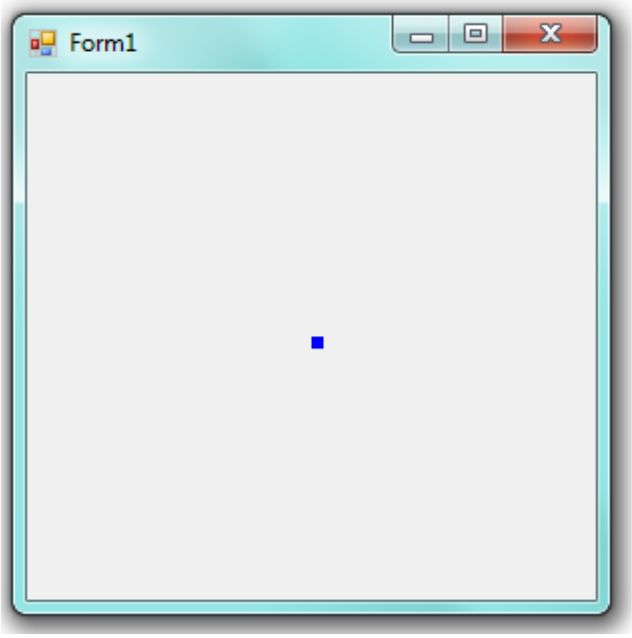
Add them as fields and initialize them in the *Form1* constructor:

```
public partial class Form1 : Form
{
    private Timer refreshTimer;

    private Point loc;
    private Size  size;

    private Pen   pen;
    private Brush brush;

    public Form1()
    {
        InitializeComponent();

        // Tell windows we qre taking responsibility for painting the screen
        SetStyle(ControlStyles.AllPaintingInWmPaint | ControlStyles.UserPaint |
        this.Paint += new PaintEventHandler(Form1_Paint);

        // initializae the critical ball information
        loc = new Point(ClientRectangle.Width / 2, ClientRectangle.Height / 2);
        size = new Size(5, 5);
        brush = new SolidBrush(Color.Blue);
        pen = new Pen(brush);

        // Screen refresh timer
        refreshTimer = new Timer();
        refreshTimer.Interval = 25;
```

We create a blue brush and use that brush to create a blue pen.

Now, compile and run. You should see a little blue box in the center of the window!

Smoother Graphics – Taking Control of Painting the Screen



Cool but boring. Let's get the ball moving.

Smoother Graphics – Taking Control of Painting the Screen

Nothing new here … we'll use the same technique used to move *PictureBoxes* … we'll just repeatedly add a delta to the X coordinate and a delta to the Y coordinate of the location. These deltas (let's call them dX and dY) will control the movement of the ball. The *loc* point controls the location of the ball.

Create *dX* and *dY* fields (*ints*) and initialize them in the constructor. For now set the movement to 4 & 6:

```
public partial class Form1 : Form
{
    private Timer refreshTimer;

    private Point loc;
    private Size  size;

    private Pen   pen;
    private Brush brush;

    private int dX;
    private int dY;

    public Form1()
    {
        InitializeComponent();

        // Tell windows we qre taking responsibility for p
        SetStyle(ControlStyles.AllPaintingInWmPaint | Cont
        this.Paint += new PaintEventHandler(Form1_Paint);

        // initializae the critical ball information
        loc = new Point(ClientRectangle.Width / 2, ClientR
        size = new Size(5, 5);
        brush = new SolidBrush(Color.Blue);
        pen = new Pen(brush);
        dX = 4;
        dY = 6;
```

Smoother Graphics – Taking Control of Painting the Screen

If you compile and run, the ball won't be moving yet. In the *refreshTimer_Tick* event handler (where we tell windows to repaint the screen) we will add the code to actually change the location of the ball. Again, this is exactly what we've done before: simply add *dX* to the *loc.X* and same with *Y.*

```
private void refreshTimer_Tick(object sender, EventArgs e)
{
    // move the ball
    loc.X += dX;
    loc.Y += dY;

    // tell windows to repaint the window
    Invalidate();
}
```

There is one trick we need to pull … while we're painting the screen, we don't want or need the refresh timer to tick again. We will pause the refresh timer and then restart it when we're done drawing. Add the following to the *Form1_Paint* event handler:

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    refreshTimer.Stop();
    Graphics g = e.Graphics;
    g.DrawRectangle(pen, loc.X, loc.Y, size.Width, size.Height);
    g.FillRectangle(brush, loc.X, loc.Y, size.Width, size.Height);
    refreshTimer.Start();
}
```

Compile and run, and you'll see the ball move diagonally downward and off the screen. We're moving!

Smoother Graphics – Taking Control of Painting the Screen

We're basically there; let's just make the ball bounce off the edges of the screen as a final touch. Again, there is nothing new to this. Main trick is to figure out where to do this. It makes the most sense to check if the ball has hit the screen bounds right after moving it. That would mean we'll add this code in the *refreshTimer_Tick* immediately after adding the deltas to the location, and before invalidating the screen:

```
private void refreshTimer_Tick(object sender, EventArgs e)
{
    // move the ball
    loc.X += dX;
    loc.Y += dY;

    // keep the ball on the screen, and bounce off
    if (loc.X < 0)
    {
        loc.X = 0;  // hit left edge
        dX = -dX;
    }
    else if (loc.X + size.Width > ClientRectangle.Width)
    {
        loc.X = ClientRectangle.Width - size.Width;  // hit right edge
        dX = -dX;
    }
    else if (loc.Y < 0)
    {
        loc.Y = 0;  // hit top edge
        dY = -dY;
    }
    else if (loc.Y + size.Height > ClientRectangle.Height)
    {
        loc.Y = ClientRectangle.Height - size.Height;  // hit bottom edge
        dY = -dY;
    }

    // tell windows to repaint the window
    Invalidate();
}
```

Compile and run, the ball should now bounce off the sides of the screen.

There is lots you can quickly play with. For instance see if you can figure out how to make the ball an actual circle. I'll give you a hint: just like a square is a special rectangle, a circle is a special ellipse. ☺

If you succeed in changing the ball to a circle, you may want to make it bigger so you can see its shape better; I'd suggest making the size 15x15.

So there you have it. Yell if you have any questions! Oh yeah … make sure you save this project, we'll be modifying it in future lessons. ☺