

## **Animation**

Animation is implemented as follows:

- Create a memory DC for double buffering
- Every so many milliseconds, update the image in the memory DC to reflect the motion since the last update, and then update the screen during a single vertical retrace interval.

The following points are vital:

- The updates must take place at least 30 times a second for smooth animation, because of the physiology of the human vision system.
- The use of double buffering is essential to avoid flicker.

To do this in .NET, we use a *Bitmap* to create the memory DC, and we get a *Graphics* object to write to that memory DC using the *FromImage* method provided by the FCL for that purpose. This is a step beyond the last lecture, in which we only used the *SetPixel* method of the *Bitmap* class to write to the memory DC.

## **Timers in .NET**

We also need to go beyond the last lecture for timing. In the last lecture, we updated our memory DC in processing the *Idle* event, and one can also do animation that way, but then the speed of the animation is dependent on the speed of the host machine. That can wreak havoc with a video game, rendering it impossible to play on a fast machine and boring on a slow machine. More accurate timing is provided by the *Timer* class in the FCL.

This could not possibly be simpler to use. (But as a Win32 veteran, I assure you that it could be more complex!) Just create a new *Timer* object, *timer1*, set *timer1.Interval* to the desired number of milliseconds, say 20, between ticks. Then *timer1* will generate *Tick* events every 20 milliseconds. Process those events and do whatever it was you wanted to do every 20 milliseconds. Just don't take longer than 20 milliseconds to do it.

You can do all this in Visual Studio by dragging and dropping a *Timer* onto your form in the design editor, and using its property sheet to set the interval and create a handler for the *Tick* event. The code that Visual Studio writes for you boils down to this:

```
this.timer1 = new Timer();
this.timer1.Interval = 20;
this.timer1.Tick += new
    System.EventHandler(this.timer1_Tick);
```

It may be hard to believe that ten years ago, I would give an entire 50-minute lecture about how to use timers in the Win32 API.

## Animating a Rectangle that Grows in Size

As a warmup exercise, let's try to animate *something*.

Create a new application called *Balls* (in anticipation of what we will eventually animate). Then add a *Bitmap* member variable *m\_theBitmap* to serve as the memory DC. As in last week's lecture, write this code:

```
private void resetDoubleBuffer()
{ Graphics g = CreateGraphics();
  if(m_theBitmap != null)
    m_theBitmap.Dispose();
  m_theBitmap = new Bitmap(this.Width, this.Height, g);
}
```

and call it in processing the *Resize* event:

```
private void Form1_Resize(object sender,
                        System.EventArgs e)
{
    resetDoubleBuffer();
}
```

I found it necessary to call *resetDoubleBuffer* in the form constructor as well. The following code snippet show this call, as well as the result of creating a timer. Note the line *ResizeRedraw = true*.

```
private System.Windows.Forms.Timer timer1;
private Bitmap m_theBitmap;
public Form1()
{
    InitializeComponent();
    ResizeRedraw = true;
    resetDoubleBuffer();
    m_Balls = new ArrayList(50);
    timer1.Start();
}
```

Now just to get started, let's try to get something drawn that changes smoothly every 20 milliseconds. Set your timer's interval property to 20, add an integer member variable *ticks*, and put in the following handler (from the timer's property sheet):

```
private void timer1_Tick(object sender, System.EventArgs e)
{
    update();
    ++ticks;
}
```

Before that will even compile, of course, we have to write *update*. Start out with the simplest possible thing:

```
private void update()
{
    if(m_theBitmap == null)
        return;
    Graphics g = Graphics.FromImage(m_theBitmap);
    g.FillRectangle(Brushes.Black, ClientRectangle);
    g.FillRectangle(Brushes.Red, 0, 0, 50+ticks, 50+ticks);
    //just for testing
}
```

```
    Invalidate();  
}
```

The first two lines may be just paranoia: it seems to work OK without them, but I could not be sure that I might not get an update message before everything has been properly initialized. I have been burned too many times in the past.

We are hoping to see a red rectangle start in the upper left corner and smoothly expand until it fills the window. However, that's not what you will see! You will see a lot of flickering. We will fix that problem next.

### **The *EraseBackground* event**

We have to go back to a basic point about Windows graphics. When you call *Invalidate*, you have learned that it causes a WM\_PAINT message to be sent to your window, which in .NET causes a *Paint* event. But the whole truth is slightly more complicated: it causes *two* messages to be sent, in this order:

- WM\_ERASEBKGND
- WM\_PAINT

The default processing of the WM\_ERASEBKGND message is to fill the client rectangle with the background color of the form (you can set that on the form's property sheet). The flicker that you see (and that can be seen in simpler programs, such as one that changes the color of a rectangle on a mouse click) is due to this. The background color of our form is some light color, not black. Now, we could try to fix this problem by changing the background color of the form to black, but that would be cosmetic. What we want to do, especially in connection with double-buffering, is to *kill all processing of WM\_ERASEBKGND*. There is absolutely no need for it: since we

intend to use *BitBlt* to paste an image over the entire client area, it would be a waste of time to first erase the background.

Here is how to do that: add this code (yes, the function has an empty body) to your form:

```
protected override void OnPaintBackground (PaintEventArgs e)
{ // stops processing WM_ERASEBKGD
}
```

You do not see the *PaintBackground* event listed in the form's property sheet under events that can be handled. Nevertheless, you can add this code, and it works to fix the flicker problem in animating the red rectangle.

## The Balls program

Once we can animate a rectangle, we can animate just about anything, and the only further complications will be specific to whatever it is we choose to animate.

As an example of animation, the *Balls* program allows the animation of a number of bouncing colored balls. The balls bounce off the sides of the client area, but they otherwise move in straight lines and they move transparently through each other, since making them bounce off each other or respond to “gravity” are exercises in physics and algebra rather than in .NET programming.

Add an *ArrayList* called *m\_Balls* to hold the balls that we will animate. Since this list holds arbitrary objects, we don't need to define our *Ball* class before we define *m\_Balls*, but defining the *Ball* class is surely the next order of business. The plan is this:

- Define the *Ball* class, giving each ball properties such as color, radius, position, and velocity.

- Implement *update()* so that it moves each ball to the new position it would move to in *timer1.Interval* milliseconds, given its present velocity and position; but if this would take it outside the client rectangle, it must “bounce off the wall” instead, changing its position and velocity as a ball would when bouncing.

Taking these in the opposite order, we implement *update* like this:

```
private void update()
{
    if(m_theBitmap == null)
        return;
    Graphics g = Graphics.FromImage(m_theBitmap);
    g.FillRectangle(Brushes.Black,ClientRectangle);
    // g.FillRectangle(Brushes.Red,0,0,50+ticks,50+ticks);
    if(m_Balls.Count > 0)
    {
        Ball.UpdateBalls(m_Balls,
                        timer1.Interval,
                        ClientRectangle);
        foreach(Ball b in m_Balls)
            b.Draw(g);
    }
    Invalidate();
}
```

This puts the responsibility for knowing how to update *m\_Balls* in the *Ball* class, which seems like the right place for it. Now the next part of the coding will be in the *Balls* class. Here are the fields I used for the *Ball* class:

```
public Color color;
public int x,y;    // position of center
public int p,q;    // x and y velocities;
public int r;     // radius of ball
```

If we wanted to make the balls bounce off of each other as well as the walls, then they should also have a *mass* field, but that is not used in

this version. I wrote a constructor that takes six arguments corresponding to these fields.

*UpdateBalls* is very simple since we're not making the balls collide with each other. Each ball's update only depends on that ball, not on the other balls:

```
public static void UpdateBalls(ArrayList balls, int t,
Rectangle box)
// update positions and velocities of all the balls to
// reflect the passage of time t (in milliseconds)
{ foreach(Ball b in balls)
    b.Bounce(t, box);
}
```

The actual work has to be done in *Bounce*. We first calculate what the new position would be if the walls weren't there, using the formula  $distance = rate * time$ . Then we check if that new position would make the ball stick out of the box. If it would, we reposition it inside the box by as much as it stuck out, and we reverse the direction of its x-velocity (if it stuck out a vertical wall) and/or its y-velocity (if it stuck out a horizontal wall). Here's the code:

```
private void Bounce(int t, Rectangle box)
{ // update position and velocity after t milliseconds
  // including bouncing off the walls of box
  x = (int)(x + p*t/1000.0);
  y = (int)(y + q*t/1000.0);
  int oops = x + r - box.Right;
  if(oops > 0)
    { x -= oops; // bounce off right wall
      p = -p;
    }
  oops = r - x;
  if(oops > 0) // bounce off left wall
    { x += oops;
      p = -p;
    }
  oops = y + r - box.Bottom;
  if(oops > 0) // bounce off bottom wall
    { y -= oops;
      q = -q;
    }
}
```

```

    }
    oops = r-y;
    if(oops > 0) // bounce off top wall
        { y += oops;
          q = -q;
        }
}

```

This code works correctly if a ball goes off-window in a corner: *both* horizontal and vertical velocities get reversed, and it gets both a horizontal and vertical displacement.

Once this code compiles, you still can't test it, because we haven't yet written any code to create balls. That's done back in *Form1* by processing *MouseDown*. We create a new ball where the mouse was clicked. But you have a lot of freedom as to how to define the color, radius, and velocity of the ball. The following code follows these simple rules:

- The colors cycle through red, blue, yellow, and green on subsequent mouse clicks.
- The sizes cycle through small, medium, and large. Thus the same size and color repeat only after twelve clicks.
- The speed of the ball is such that, if it moved straight to the left, it would reach the edge of the window in one second. Thus fast balls are created by clicking near the right of the window, slow balls by clicking near the left.
- The direction of the velocity is random.

Here's the code:

```

private void Form1_MouseDown(object sender,
System.Windows.Forms.MouseEventArgs e)
{ // create a new ball with a random direction and
  // speed sufficient to reach the left wall in one second if
  // its direction were (-1,0).
  double theta = // a random angle

```

```

        m_RandomNumbers.NextDouble() * Math.PI * 2.0;
    int speed = e.X;
    int p = (int) (speed * Math.Cos(theta));
    int q = (int) (speed * Math.Sin(theta));

    Color c;
    int nBalls = m_Balls.Count;
    switch(m_Balls.Count % 4)
    {
        case 0: c = Color.Red; break;
        case 1: c = Color.Yellow; break;
        case 2: c = Color.Blue; break;
        default: c = Color.Green; break;
    }
    int radius;
    switch(m_Balls.Count % 3)
    {
        case 0: radius = 10; break;
        case 1: radius = 20; break;
        default: radius = 40; break;
    }
    int x = e.X;
    int y = e.Y;
    int oops = e.X + radius - ClientRectangle.Width;
    if (oops > 0)
        x -= oops;
    oops = radius - e.X;
    if(oops > 0)
        x += oops;
    oops = e.Y + radius - ClientRectangle.Height;
    if(oops > 0)
        y -= oops;
    oops = radius - e.Y;
    if(oops > 0)
        y += oops;
    Ball b = new Ball(c,x,y,p,q,radius);
    m_Balls.Add(b);
    // don't call Invalidate!
}

```

All this code is *Ball*-specific. The only point of general interest for animation coding is the last comment: *don't call Invalidate!* You are accustomed to calling *Invalidate* when a mouse-click changes the data, in order to cause the changes to show up on the screen. But when you are using double-buffering, usually the update of the screen is controlled another way. In this case, it's controlled by

the timer, so your new ball will show up within 20 milliseconds anyway. There is no need to disrupt the flow of steady updates every 20 milliseconds by causing another one in between.

Here's a screen shot of the program; of course, you have to run the actual program to see the balls move.

