10

Timers, Animation, Keyboard Events

Review and Preview

It's the last class. By now, you should have some confidence in your abilities as a Visual C# programmer. In this class, we'll look at one more control that's a lot of fun - the timer control.

It's a key control for adding animation (motion) to graphics in projects. We study some animation techniques and we'll examine how to recognize user inputs from the keyboard via keyboard events. Then, you'll build one last project (at least, the last project in this class) - your first video game!

Timer Control

The Visual C# **timer** control has an interesting feature. It is the one control that can generate events without any input from the user. Timer controls work in your project's background, generating events at time intervals you specify. This event generation feature comes in handy for graphics animation where screen displays need to be updated at regular intervals. The timer control is selected from the toolbox. It appears as:

In Toolbox:

Below Form (default properties):

There is no user interface (nothing to click or nothing to look at) for the timer control, so it will not appear on the form. Such controls are placed in the "tray area" below the form in the design window.

Properties

The timer control properties are:

Property	Description
Name	Name used to identify timer control. Three letter prefix for timer names is **tim**.
Interval	Number of milliseconds between timer events. There are 1000 milliseconds in one second.
Enabled	Used to turn timer control on and off. When True, timer continues to generate events until set to False.

Events

The timer control has a single event:

Event	Description
Tick	Event method executed every Interval milliseconds when timer control Enabled property is True.

Examples

A few examples should clarify how the timer control works. It's very simple and very powerful. Here's what happens. If a timer control's **Enabled** property is **True** (the timer is on), every **Interval** milliseconds, Visual C# will generate an event and execute the corresponding **Tick** event method. No user interaction is needed. If your timer is named **timExample**, the Timer event method has the form:

```
private void timExample_Tick(object sender, EventArgs e)
{
      [C# code to be executed every Interval milliseconds]
}
```

Whatever C# code you want to execute is put in this method.

The Interval property is the most important timer control property. This property is set to the number of milliseconds between timer events. A millisecond is 1/1000th of a second, or there are 1,000 milliseconds in a second. If you want to generate N events per second, set Interval to 1000 / N. For example, if you want a timer event to occur 4 times per second, set Interval to 250. About the lowest practical value for Interval is 50 and values that differ by 5, 10, or even 20 are likely to produce similar results. It all depends on your particular computer.

The only other property to worry about is the Enabled property. It is used to turn the timer on (true) or off (false). In design mode, the timer control Enabled property is given a default value of False. We will always leave this at False. It is good programming practice to control timers programmatically. This simply means turn your timers on and off in C# code. It's a matter of changing the Enabled property. And, always make sure if you turn a timer on that you turn it off when you need to. Now, the first example.

Start Visual C# and start a new project. Add a timer control (it will appear below the form) and button to the form. In this example, we will use the timer control to make your computer beep every second. The button will turn the timer on and off. Set the timer control (**timer1** default name) Interval property to 1000 (1000 milliseconds equals one second). Put this code in the **timer1_Tick** event method (it will be one of the few events listed for the timer control in the code window):

```
private void timer1_Tick(object sender, EventArgs e)
{
    System.Media.SystemSounds.Beep.Play();
}
```

Beep is the C# function that makes the computer beep, or is that obvious?

Put this code in the button (default name **Button1**) **Button1_Click** event method:

```
private void button1_Click(object sender, EventArgs e)
{
    if (timer1.Enabled)
    {
        timer1.Enabled = false;
    }
    else
    {
        timer1.Enabled = true;
    }
}
```

What does this code do? If the timer is on (timer1.Enabled = true), it turns it off (timer1.Enabled = false), and vice versa. We say this code "toggles" the timer. Run the project. Click the button. Your computer will beep every second (the Tick event is executed every 1000 milliseconds, the Interval value) until you click the button again. Notice it does this no matter what else is going on. It requires no input (once the timer is on) from you, the user. Click the button. The beeping will stop. Remember to always let your C# code turn timer controls on and off. Stop the project when you get tired of the beeping.

Add the two shaded lines of code to the **timer1_Tick** event, so it now reads:

```
private void timer1_Tick(object sender, EventArgs e)
{
    System.Media.SystemSounds.Beep.Play();
    Random myRandom = new Random();
    this.BackColor = Color.FromArgb(myRandom.Next(256),
myRandom.Next(256), myRandom.Next(256));
}
```

This extra code randomly changes the form (name **this**) background color using the **FromArgb** method (using random red, green and blue values). Run the project. Click the button. Now, every second, the computer beeps and the form changes color. Stop the timer. Stop the project.

What if we want the computer to beep every second, but want the form color to change four times every second? If events require different intervals, each event needs its own timer. Add another timer control to the form (default name **timer2**). We'll use this timer to control the form color. Set timer2's Interval to 250 (Tick event executed every 0.25 seconds, or 4 color changes per second). Cut and paste the lines of code in **timer1_Tick** that sets color into the **timer2_Tick** event. The two timer Tick events are now:

```
private void timer1_Tick(object sender, EventArgs e)
{
    System.Media.SystemSounds.Beep.Play();
}

private void timer2_Tick(object sender, EventArgs e)
{
    Random myRandom = new Random();
    this.BackColor = Color.FromArgb(myRandom.Next(256),
myRandom.Next(256), myRandom.Next(256));
}
```

We also need to add code to the **button1_Click** event to toggle (turn it on and off) this new timer. We could copy and paste the five lines of code there for timer1 and change all the timer1 words to timer2. And, this would work. But, let me show you a quick way to toggle Boolean (bool type) variables, like the Enabled property. We'll be able to replace five lines of code with one!

Way back in Class 6, we studied logical operators - operators that work with Boolean variables. Remember and (**&&**)? Remember or (**||**)? Well, there's another logical operator that comes in handy - the **not** operator, represented by the exclamation point (**!**).. This operator works on a single Boolean variable. If we have a Boolean variable (**bool** type) named x, it can have two values, true or false. **!x** has the opposite value of X as shown in this simple logic table:

x	!x
true	false
false	true

Notice the not operator toggles the Boolean variable x. If x is on (true), the not operator turns x off (false). If x is off (false), the not operator turns x on (true). So, we can use the not operator to turn timer controls on and off. Use this code in the **button1_Click** event method in our example:

```
private void button1_Click(object sender, EventArgs e)
{
    timer1.Enabled = !(timer1.Enabled);
    timer2.Enabled = !(timer2.Enabled);
}
```

Notice how the not operator simplifies using timer controls. Do you see that one line of C# code using not has exactly the same effect as the five lines of code we used earlier to toggle the timer? Run the project. Click the button. Do you see how the two timer events are interacting? You should hear a beep every four times the screen changes color. Stop the project when you're done playing with it.

Let's use the timer to do some flashier stuff. Start a new project. Add a panel control (default name **panel1**). Make the panel fairly big – make it wider than it is tall. Add a button (default name **button1**), and a timer control (default name **timer1**). Set the timer control Interval property to 50. Declare and initialize an int type variable **delta** in the general declarations area:

```
int delta = 0;
```

Toggle the timer in the **Button1_Click** event:

```
private void button1_Click(object sender, EventArgs e)
{
    timer1.Enabled = !(timer1.Enabled);
}
```

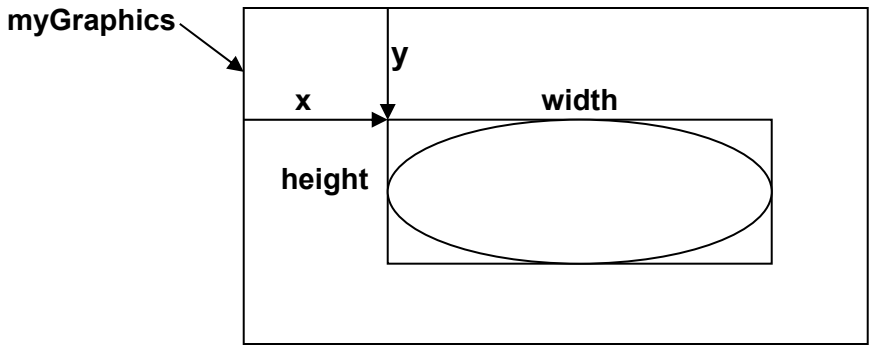
Put this code in the **Timer1_Tick** event:

```
private void timer1_Tick(object sender, EventArgs e)
{
    Graphics myGraphics;
    Pen myPen;
    Random myRandom = new Random();
    myGraphics = panel1.CreateGraphics();
    myPen = new Pen(Color.FromArgb(myRandom.Next(256),
myRandom.Next(256), myRandom.Next(256)), 2);
    myGraphics.DrawEllipse(myPen, delta, delta, panel1.Width
- 2 * delta, panel1.Height - 2 * delta);
    delta = delta + (int) myPen.Width;
    if (delta > panel1.Height / 2)
    {
        delta = 0;
        myGraphics.Clear(panel1.BackColor);
    }
    myPen.Dispose();
    myGraphics.Dispose();
}
```

You should recognize most of what's here. We've created a graphics object and pen object (with a random color) to do some drawing. Notice, though, we use a graphics method (**DrawEllipse**) we haven't seen before. You should be able to understand it and you'll see it gives a really neat effect in this example. The **DrawEllipse** method has the form:

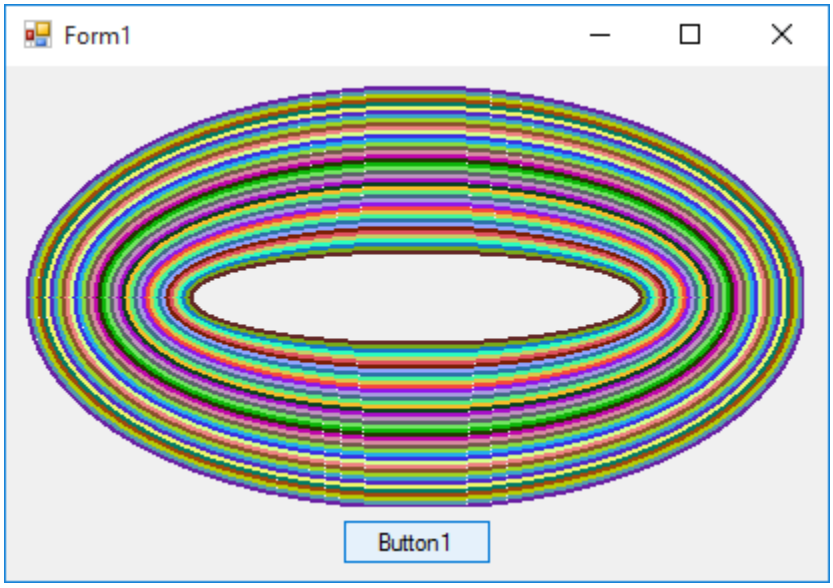
```
myGraphics.DrawEllipse(myPen, x, y, width, height);
```

Here, **myGraphics** is the graphics object. This command draws an ellipse, with a width **width** and height **height**, in the graphics object starting at the point (**x**, **y**). A picture shows the result:



In your work with Visual C#, you will often see code you don't recognize. Learn to use the on-line help facilities (try it with **DrawEllipse**) in these cases.

Back to the code, you should see the **DrawEllipse** method draws the first ellipse around the border of the panel control (**x = 0** initially). The surrounding rectangle moves "in" an amount **delta** (in each direction) with each Tick event, resulting in a smaller rectangle (the width and height are decreased by both **2*delta**). Once delta (incremented by the pen width in each step) exceeds half of the panel height, it is reset to 0, the panel is cleared and the process starts all over. Run the project. Click the button. Are you hypnotized? Here's a sample of a run I made:



Can you think of other things you could draw using other graphics methods? Look at **DrawRectangle** for example. Try your ideas.

In this last example, the periodic (every 0.050 seconds) changing of the display in the graphics object, imparted by the timer control, gives the appearance of motion – the ellipses seem to be moving inward. This is the basic concept behind a very powerful graphics technique - **animation**. In animation, we have a sequence of pictures, each a little different from the previous one. With the ellipse example, in each picture, we add a new ellipse. By displaying this sequence over time, we can trick the viewer into thinking things are moving. It all has to do with how fast the human eye and brain can process information. That's how cartoons work - 24 different pictures are displayed every second - it makes things look like they are moving, or animated. Obviously, the timer control is a key element to animation, as well as for other Visual C# timing tasks. In the C# lesson for this class, we will look at how to do simple animations and some other things.

Typical Use of Timer Control

The usual design steps to use a timer control are:

- Set the **Name** property and **Interval** property.
- Write code in **Tick** event.
- At some point in your application, set **Enabled** to **True** to start timer. Also, have capability to reset **Enabled** to **False**, when desired.

C# - The Final Lesson

In this last C# lesson, we study some simple animation techniques, look at math needed with animations, and learn how to detect keyboard events.

Animation - DrawImage Graphics Method

In the last example, we saw that by using a timer to periodically change the display in a panel control, a sense of motion, or animation, is obtained. We will use that idea here to do a specific kind of animation - moving objects around. This is the basis for nearly every video game ever made. The objects we move will be images contained in Visual C# picture box controls.

Moving images in a panel is easy to do. First, establish an image (set the **Image** property) in the picture box. This image is then placed in the panel control using the **DrawImage** graphics method. Like the other graphics methods we've seen (DrawLine and DrawEllipse), before using DrawImage, you need to establish a graphics object to draw to. The graphics object is declared in the usual manner (usually in the **general declarations** area):

```
Graphics myGraphics;
```

We then create the graphics object (assume **myControl** is the host control; we'll use a panel):

```
myGraphics = myControl.CreateGraphics();
```

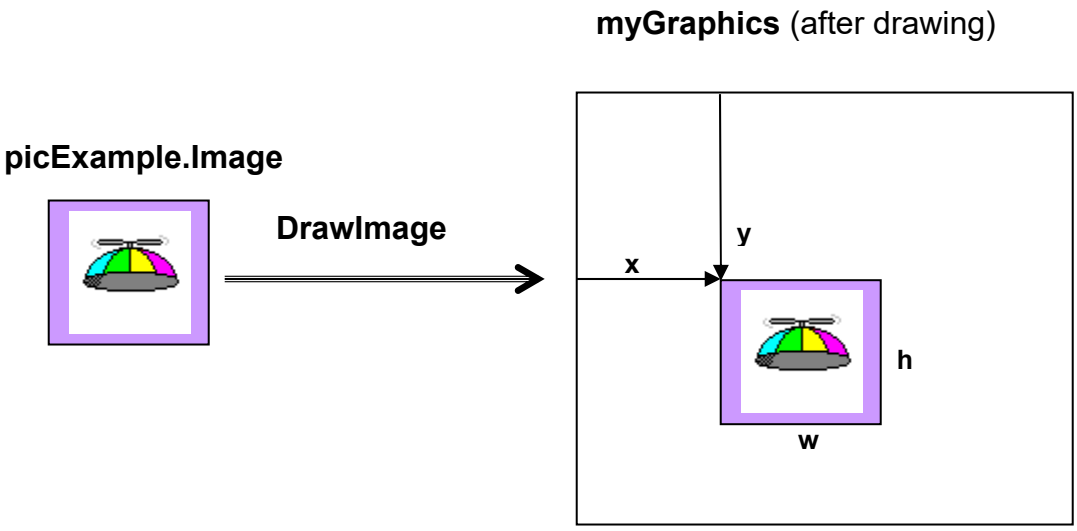
This creation usually occurs in the form **Load** method. You dispose of the object in the form **FormClosing** method.

Now, assume we have an image in a picture box control named **picExample**. At this point, we can draw **picExample.Image** in **myGraphics**, using **DrawImage**. The **DrawImage** method that does this is:

```
myGraphics.DrawImage(picExample.Image, x, y, w, h);
```

where **x** is the horizontal position of the image within **myGraphics** and **y** is the vertical position. The image will have a width value **w** and a height **h**. The width and height can be the original image size or scaled up or down. It's your choice.

A picture illustrates what's going on with **DrawImage**:

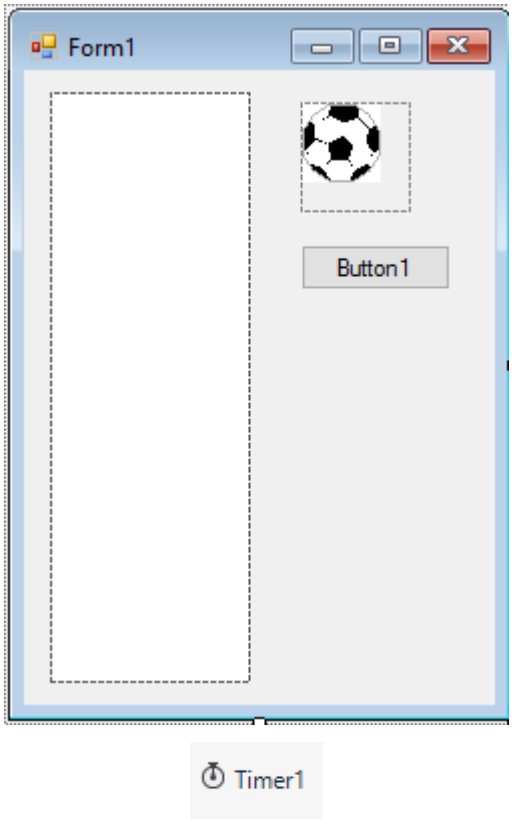


Note how the transfer of the rectangular image occurs. Successive transfers (always erasing the previous position of the image) gives the impression of motion, or animation. Where do we put the DrawImage statement?

Each picture box image to be moved must have an associated timer control. If desired, several images can use the same timer. The DrawImage statement is

placed in the corresponding timer control Tick event. Whenever a Tick event is triggered, the image is erased at its old position (by putting a “blank” image in that position), a new image position is computed and the DrawImage method executed. This periodic movement is **animation**. Let’s look at an example to see how simple it really is.

Start Visual C# and start a new project. Put a panel (default name **panel1**) on the form - make it fairly tall with a white background color. Put a small picture box (name **pictureBox1**) on the form. Set its **Image** property (I used the soccer ball bitmap graphic in the **\BeginVCS\BVCS Projects\Graphics** folder). Place a timer control (default name **timer1**) on the form. Use an Interval property of 100. Place a button (default name **button1**) on the form for starting and stopping the timer. We will use this example a lot. Try to make it look something like this:



Define the needed graphics object in the **general declarations** area. Also include a variable (**imageY**) to keep track of the vertical position of the image:

```
Graphics myGraphics;
int imageY;
```

And create the object in the **Form1_Load** event method:

```
private void Form1_Load(object sender, EventArgs e)
{
    myGraphics = panel1.CreateGraphics();
}
```

Dispose of the graphics object in the **Form1_FormClosing** event method:

```
private void Form1_FormClosing(object sender,
FormClosingEventArgs e)
{
    myGraphics.Dispose();
}
```

Use **button1_Click** to toggle the timer and initialize the position (**imageY**) of **pictureBox1.Image** at the top of the panel control:

```
private void button1_Click(object sender, EventArgs e)
{
    timer1.Enabled = !(timer1.Enabled);
    imageY = 0;
}
```

Now, move the image in the **timer1_Tick** event:

```
private void timer1_Tick(object sender, EventArgs e)
{
    int imageX = 10;
    int imageW = 30;
    int imageH = 25;
    myGraphics.Clear(panel1.BackColor);
    imageY = imageY + panel1.Height / 40;
    myGraphics.DrawImage(pictureBox1.Image, imageX, imageY,
imageW, imageH);
}
```

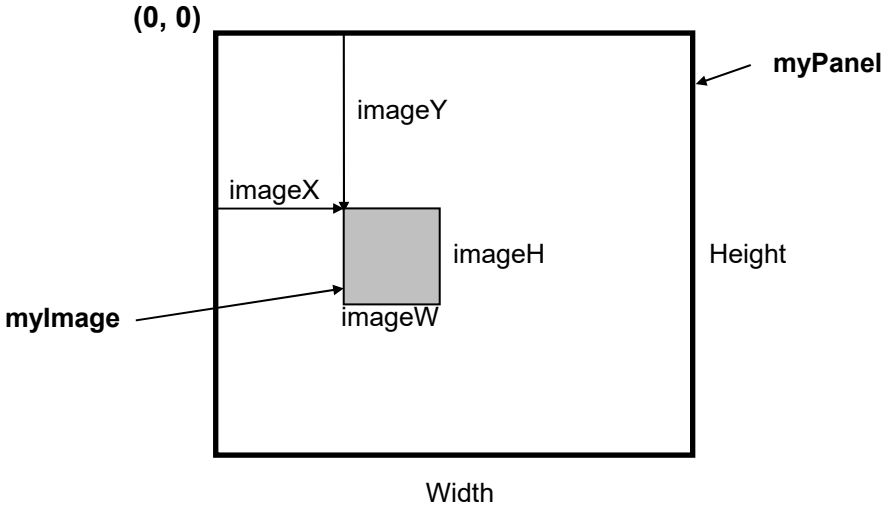
In this event, the image width (**imageW**) and height (**imageH**) are given values, as is the horizontal location (**imageX**). Then, the graphics object is cleared to erase the previous image. The vertical position of the image (**ImageY**) is increased by 1/40th of the panel height each time the event is executed (every 0.1 seconds). The picture box image is moving down. It should take 40 executions of this routine, or about 4 seconds, for the image to reach the bottom. Let's try it.

Run the example project. Click the button to start the timer. Watch the image drop. Notice the image is scaled to fit the area defined by the DrawImage method. Pretty easy, wasn't it? How long does it take the image to reach the bottom? What happens when it reaches the bottom? It just keeps on going down through the panel, through the form and out through the bottom of your computer monitor to who knows where! We need to be able to detect this disappearance and do something about it. We'll look at two ways to handle this. First, we'll make the image reappear at the top of the panel, or scroll. Then, we'll make it bounce. Stop the project. Save it too. We'll be using it again.

Image Disappearance

When images are moving in a panel, we need to know when they move out of the panel across a border. Such information is often needed in video type games. We just saw this need with the falling ball example. When an **image disappearance** happens, we can either ignore that image or perhaps make it "scroll" around to other side of the panel control. How do we decide if an image has disappeared? It's basically a case of comparing various positions and dimensions.

We need to detect whether a image has completely moved across one of four panel borders (top, bottom, left, right). Each of these detections can be developed using this diagram of a picture box image (**myImage**) within a panel (**myPanel**):



Notice the image is located at (**imageX**, **imageY**), is **imageW** pixels wide and **imageH** pixels high.

If the image is moving down, it completely crosses the panel bottom border when its top (**imageY**) is lower than the bottom border. The bottom of the panel is **myPanel.Height**. C# code for a bottom border disappearance is:

```
if (imageY > myPanel.Height)
{
      [C# code for bottom border disappearance]
}
```

If the image is moving up, the panel top border is completely crossed when the bottom of the image (**imageY + imageH**) becomes less than 0. In C#, this is detected with:

```
if ((imageY + imageH) < 0)
{
      [C# code for top border disappearance]
}
```

If the control is moving to the left, the panel left border is completely crossed when image right side (**imageX + imageW)** becomes less than 0. In C#, this is detected with:

```
if ((imageX + imageW) < 0)
{
      [C# code for left border disappearance]
}
```

If the image is moving to the right, it completely crosses the panel right border when its left side (**imageX**) passes the border. The right side of the panel is **myPanel.Width**. C# code for a right border disappearance is:

```
if (imageX > myPanel.Width)
{
      [C# code for right border disappearance]
}
```

Let's add disappearance detection to our "falling soccer ball" example. Return to that project. Say, instead of having the image disappear when it reaches the bottom, we have it magically reappear at the top of the panel. We say the image is scrolling. Modify the **timer1_Tick** event to this (new lines are shaded):

```
private void timer1_Tick(object sender, EventArgs e)
{
    int imageX = 10;
    int imageW = 30;
    int imageH = 25;
    myGraphics.Clear(panel1.BackColor);
    imageY = imageY + panel1.Height / 40;
    myGraphics.DrawImage(pictureBox1.Image, imageX, imageY,
imageW, imageH);
    if (imageY > panel1.Height)
    {
        imageY = -imageH;
    }
}
```

We added the bottom border disappearance logic. Notice when the image disappears, we reset its imageY value so it is repositioned just off the top of the panel. Run the project. Watch the image scroll. Pretty easy, wasn't it? Stop and save the project.

Border Crossing

What if, in the falling image example, instead of scrolling, we want the image to bounce back up when it reaches the bottom border? This is another common animation task - detecting the initiation of **border crossings**. Such crossings are used to change the direction of moving images, that is, make them bounce. How do we detect border crossings?

The same diagram used for image disappearances can be used here. Checking to see if an image has crossed a panel border is like checking for image disappearance, except the image has not moved quite as far. For top and bottom checks, the image movement is less by an amount equal to its height value (imageH). For left and right checks, the control movement is less by an amount equal to its width value (imageW). Look back at that diagram and you should see these code segments accomplish the respective border crossing directions:

```
if (imageY < 0)
{
      [C# code for top border crossing]
}
```

```
if ((imageY + imageH) > myPanel.Height)
{
      [C# code for bottom border crossing]
}
```

```
if (imageX < 0)
{
      [C# code for left border crossing]
}
```

```
if ((imageX + imageW) > myPanel.Width)
{
      [C# code for right border crossing]
}
```

Let's modify the falling image example to have it bounce when it reaches the bottom of the panel. Declare an integer variable **imageDir** in the **general declarations** area:

```
int imageDir;
```

imageDir is used to indicate which way the image is moving. When imageDir is 1, the image is moving down (imageY is increasing). When imageDir is -1, the image is moving up (imageY is decreasing). Change the **button1_Click** event to (new line is shaded):

```
private void button1_Click(object sender, EventArgs e)
{
    timer1.Enabled = !(timer1.Enabled);
    imageY = 0;
    imageDir = 1;
}
```

We added a single line to initialize imageDir to 1 (moving down).

Change the **timer1_Tick** event to this (again, changed and/or new lines are shaded):

```
private void timer1_Tick(object sender, EventArgs e)
{
    int imageX = 10;
    int imageW = 30;
    int imageH = 25;
    myGraphics.Clear(panel1.BackColor);
    imageY = imageY + imageDir * panel1.Height / 40;
    myGraphics.DrawImage(pictureBox1.Image, imageX, imageY,
imageW, imageH);
    if (imageY + imageH > panel1.Height)
    {
        imageY = panel1.Height - imageH;
        imageDir = -1;
    }
}
```

We modified the calculation of imageY to account for the imageDir variable. Notice how it is used to impart the proper direction to the image motion (down when imageDir is 1, up when imageDir is –1). We have also replaced the code in the existing if structure for a bottom border crossing. Notice when a crossing is detected, the image is repositioned (by resetting imageY) at the bottom of the panel (**panel1.Height - imageH**) and imageDir is set to -1 (direction is changed so the image will start moving up). Run the project. Now when the image reaches the bottom of the panel, it reverses direction and heads back up. We've made the image bounce! But, once it reaches the top, it's gone again!

Add top border crossing detection, so the **timer1_Tick** event is now (changes are shaded):

```
private void timer1_Tick(object sender, EventArgs e)
{
    int imageX = 10;
    int imageW = 30;
    int imageH = 25;
    myGraphics.Clear(panel1.BackColor);
    imageY = imageY + imageDir * panel1.Height / 40;
    myGraphics.DrawImage(pictureBox1.Image, imageX, imageY,
imageW, imageH);
    if (imageY + imageH > panel1.Height)
    {
        imageY = panel1.Height - imageH;
        imageDir = -1;
        System.Media.SystemSounds.Beep.Play();
    }
    else if (imageY < 0)
    {
        imageY = 0;
        imageDir = 1;
        System.Media.SystemSounds.Beep.Play();
    }
}
```

In the top crossing code (the else if portion), we reset imageY to 0 (the top of the panel) and change imageDir to 1. We've also added a couple of Beep statements so there is some audible feedback when either bounce occurs. Run the project again. Your image will now bounce up and down, beeping with each bounce, until you stop it. Stop and save the project.

The code we've developed here for checking and resetting image positions is a common task in Visual C#. As you develop your programming skills, you should make sure you are comfortable with what all these properties and dimensions mean and how they interact. As an example, do you see how we could compute imageX so the image is centered in the panel? Try this in the **timer1_Tick** method:

```
imageX = (int) (0.5 * (panel1.Width - imageW));
```

Make sure you put this line after the line declaring imageW. Note the use of the cast (conversion) of the computation to an **int** type. Save the project one more time.

You've now seen how to do lots of things with animations. You can make images move, make them disappear and reappear, and make them bounce. Do you have some ideas of simple video games you would like to build? You still need two more skills – image erasure and collision detection - which are discussed next.

Image Erasure

In the little example we just did, we had to clear the panel control (using the **Clear** graphics method) prior to each DrawImage method. This was done to erase the image at its previous location before drawing a new image. This “erase, then redraw” process is the secret behind animation. But, what if we are animating many images? The Clear method would clear all images from the panel and require repositioning every image, even ones that haven’t moved. This would be a slow, tedious and unnecessary process. It would also result in an animation with lots of flicker.

We will take a more precise approach to erasure. Instead of erasing the entire panel before moving an image, we will only erase the rectangular region previously occupied by the image. To do this, we will use the **FillRectangle** graphics method, a new concept. This method is straightforward and, with your Visual C# knowledge, you should easily understand how it is used. If applied to a graphics object named **myGraphics**, the form is:

```
myGraphics.FillRectangle(myBrush, x, y, width, height);
```

This line of code will “paint” a rectangular region located at (**x, y**), **width** wide, and **height** high with a brush object (**myBrush**).

And, yes, there’s another new concept – a **brush** object. A brush is like a “wide” pen. It is used to fill areas with a color. A brush object is declared (assume an object named **myBrush**) using:

```
Brush myBrush;
```

Then, a solid brush (one that paints with a single color) is created using:

```
myBrush = new SolidBrush(Color);
```

where you select the **Color** of the brush. Once done with the brush, dispose of the object using the **Dispose** method.

So, how does this work with the problem at hand? We will create a “blank” brush (we’ll even name it **blankBrush**) with the same color as the BackColor property of the panel (**myPanel**) control. The code to do this (after declaring the brush object) is:

```
blankBrush = new SolidBrush(myPanel.BackColor);
```

Then, to erase an image located in **myGraphics** at (imageX, imageY), imageW pixels wide and imageH pixels high, we use:

```
myGraphics.FillRectangle(blankBrush, imageX, imageY, imageW,
imageH);
```

This will just paint the specified rectangular region with the panel background color, effectively erasing the image that was there.

Open up the “bouncing soccer ball” example one more time. Add this line of code in the **general declarations** area:

```
Brush blankBrush;
```

Add this line in the **Form1_Load** method:

```
blankBrush = new SolidBrush(panel1.BackColor);
```

And, add this line in the **Form1_FormClosing** method:

```
blankBrush.Dispose();
```

These three lines declare, create and dispose of the brush object at the proper times. Finally, in the **Timer1_Tick** method, replace the line using the Clear method with this new line of code (selective erasing):

```
myGraphics.FillRectangle(blankBrush, imageX, imageY, imageW,
imageH);
```

Rerun the project. You probably won't notice much difference since we only have one object moving. But, in more detailed animations, this image erasing approach is superior.

Collision Detection

Another requirement in animation is to determine if two images have collided. This is needed in games to see if a ball hits a paddle, if an alien rocket hits its target, or if a cute little character grabs some reward. Each image is described by a rectangular area, so the **collision detection** problem is to see if two rectangles collide, or overlap. This check is done using each image's position and dimensions.

Here are two images (**image1** and **image2**) in a panel control:

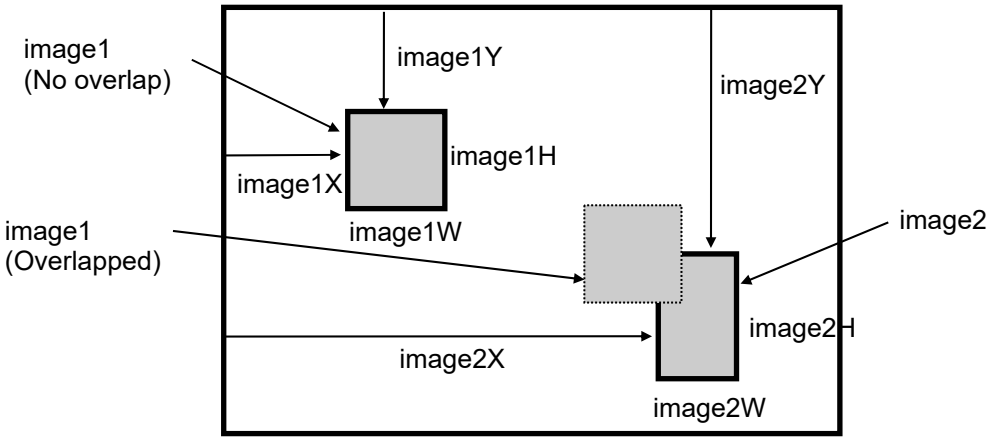


image1 is positioned at (**image1X**, **image1Y**), is **image1W** wide and **image1H** high. Similarly, **image2** is positioned at (**image2X**, **image2Y**), is **image2W** wide and **image2H** high.

Looking at this diagram, you should see there are four requirements for the two rectangles to overlap:

1. The right side of image1 (**image1X + image1H**) must be “farther right” than the left side of image2 (**image2X**)
2. The left side of image1 (**image1X**) must be “farther left” than the right side of image2 (**image2X + image2W**)
3. The bottom of image1 (**image1Y + image1H**) must be “farther down” than the top of image2 (**image2Y**)
4. The top of image1 (**image1Y**) must be “farther up” than the bottom of image2 (**image2Y + image2H**)

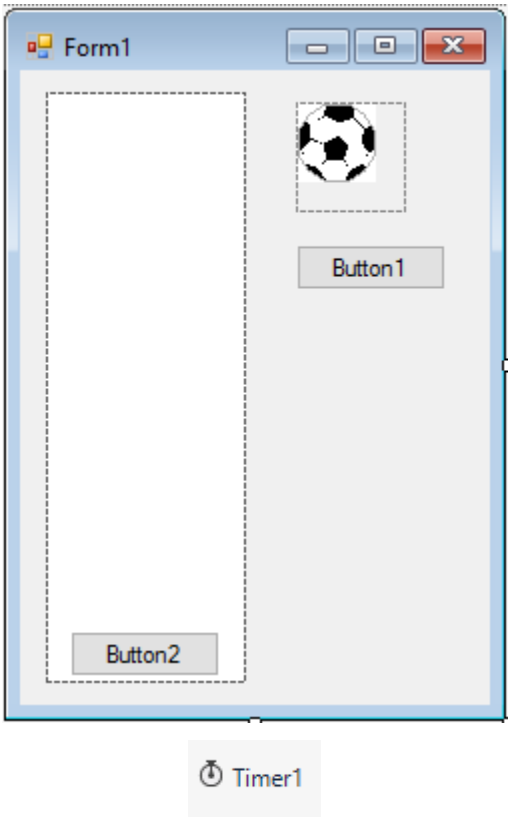
All four of these requirements must be met for a collision.

The C# code to check if these rectangles overlap is:

```
if ((image1X + image1W) > image2X)
{
    if (image1X < (image2X + image2W))
    {
        if ((image1Y + image1H) > image2Y)
        {
            if (image1Y < (image2Y + image2H))
            {
                [C# code for overlap, or collision]
            }
        }
    }
}
```

This code checks the four conditions for overlap using four “nested” if structures. The C# code for a collision is executed only if all four conditions are found to be true.

Let's try some collision detection with the bouncing soccer ball example. Add a button (default name **button2**) control near the bottom of the panel – narrow the width a bit. Blank out the Text property of the button so no text is on it. Make sure the button is "attached" to the panel. Yes, we know a button is not an image, but it is a rectangle and the same overlap rules apply. We want to see if the image will collide with the button control and bounce up. Your form should look something like this:



Change the **timer1_Tick** event code to (added code is shaded):

```
private void timer1_Tick(object sender, EventArgs e)
{
    int imageX = 10;
    int imageW = 30;
    int imageH = 25;
    bool collision;
    myGraphics.FillRectangle(blankBrush, imageX, imageY,
imageW, imageH);
    imageY = imageY + imageDir * panel1.Height / 40;
    myGraphics.DrawImage(pictureBox1.Image, imageX, imageY,
imageW, imageH);
    collision = false;
    if ((imageX + imageW) > button2.Left)
    {
        if (imageX < (button2.Left + button2.Width))
        {
            if ((imageY + imageH) > button2.Top)
            {
                if (imageY < (button2.Top + button2.Height))
                {
                    collision = true;
                }
            }
        }
    }
    if (collision)
    {
        imageY = button2.Top - imageH;
        imageDir = -1;
        System.Media.SystemSounds.Beep.Play();
    }
    else if (imageY < 0)
    {
        imageY = 0;
        imageDir = 1;
        System.Media.SystemSounds.Beep.Play();
    }
}
```

We declare a method level bool variable **collision** to indicate an overlap (true for overlap, false for no overlap). The overlap code [using the button control properties for location (Left, Top) and size (Width, Height)] follows the DrawImage method. If a collision is detected, the image is repositioned so it just touches the top of button2, its direction is reversed and a beep is played. The code for bouncing off the top of the panel is unchanged. Run the project. Notice the image now bounces off the button. Stop the project. Move button2 out of the panel control so the image won't collide with it. The image should just drop off the screen. See how close the image can pass by button2 without colliding to make sure the overlap routine works properly. Stop and save the project.

Now that you know how to detect collisions, you're well on your way to knowing how to build a simple video game. Next, we'll learn how to detect keyboard events from the user. One possible use for these events, among many, is to allow a user to move a little paddle to "hit" a dropping ball. The collision technique we just learned will come in handy for such a task.

Keyboard Events

In Class 8, we looked at ways for a user to interact with a Visual C# project using the mouse for input. We studied three mouse events: MouseDown, MouseMove, and MouseUp. Another input device available for use is the computer keyboard. Here we look at **keyboard events** which give our projects the ability to detect user input from the keyboard. Two keyboard events are studied: the **KeyDown** event and the **KeyPress** event.

Several Visual C# controls can recognize keyboard events, notably the form and the text box. Yet, only the control that has **focus** can receive a keyboard event. (Recall the control with focus is the active control.) When trying to detect a keyboard event for a certain control, we need to make sure that control has focus. We can give a control focus by clicking on it with the mouse. But, another way to assign focus to a control is with the **Focus** method. The format for such a statement is:

```
controlName.Focus();
```

This command in C# will give **controlName** focus and make it the active control. It has the same effect as clicking on the control. The control can then recognize any associated keyboard events. We use the Focus method with keyboard events to insure proper execution of each event.

To detect keyboard events on the form, you need to set the form **KeyPreview** property to **True**. This bypasses any keystrokes used by the controls to generate events.

KeyDown Event

The **KeyDown** event has the ability to detect the pressing of any key on the computer keyboard. It can detect:

- Special combinations of the Shift, Ctrl, and Alt keys
- Insert, Del, Home, End, PgUp, PgDn keys
- Cursor control keys
- Numeric keypad keys (it can distinguish these numbers from those on the top row of the keyboard)
- Function keys
- Letter, number and character keys

The KeyDown event for a control **controlName** is executed whenever that control has focus and a key is pressed. The form of this event method is:

```
private void controlName_KeyDown(object sender, KeyEventArgs
e)
{
      [C# code for KeyDown Event]
}
```

The KeyDown event has two arguments: **sender** and **e**. We won't be concerned with the sender argument in this class. And, we won't be concerned with the status of any of the control keys (such as Shift, Ctrl, Alt). We only want to know what key was pressed down to invoke this method.

The property **e.KeyCode** can be used to determine which key was pressed down. There is a **KeyCode** value for each key on the keyboard. By evaluating the **e.KeyCode** argument, we can determine which key was pressed. There are nearly 100 KeyCode values, some of which are:

e.KeyCode	Description
Keys.Back	The BACKSPACE key.
Keys.Cancel	The CANCEL key.
Keys.Delete	The DEL key.
Keys.Down	The DOWN ARROW key.
Keys.Enter	The ENTER key.
Keys.Escape	The ESC key.
Keys.F1	The F1 key.
Keys.Home	The HOME key.
Keys.Left	The LEFT ARROW key.
Keys.NumPad0	The 0 key on the numeric keypad.
Keys.PageDown	The PAGE DOWN key.
Keys.PageUp	The PAGE UP key.
Keys.Right	The RIGHT ARROW key.
Keys.Space	The SPACEBAR key.
Keys.Tab	The TAB key.
Keys.Up	The UP ARROW key.

Using the KeyDown event is not easy. There is a lot of work involved in interpreting the information provided in the KeyDown event. For example, the KeyDown event cannot distinguish between an upper and lower case letter. You need to make that distinction in your C# code. You usually use a switch or if structure (based on e.KeyCode) to determine which key was pressed. Let's see how to use KeyDown to recognize some keys.

Start Visual C# and start a new project. Put a text box control (**textBox1**) on the form. Use this **textBox1_KeyDown** event (make sure you pick the correct event):

```
private void textBox1_KeyDown(object sender, KeyEventArgs e)
{
    textBox1.Text =
Convert.ToString(Convert.ToInt32(e.KeyCode));
}
```

Run the project. Type a letter. The letter and its corresponding e.KeyCode (a numeric value) are shown (there is no space between the two values). Press the same letter while holding down the <Shift> key. The same code will appear – there is no distinction between upper and lower case. Press each of the four arrow keys to see their different values. Notice for such 'non-printable' keys, only a number displays in the text box. Type numbers using the top row of the keyboard and the numeric keypad (make sure your **NumLock** key is selected). Notice the keypad numbers don't display and have different KeyCode values than the "keyboard numbers." This lets us distinguish the keypad from the keyboard. Try various keys on the keyboard to see which keys have a KeyCode (all of them). Notice it works with function keys, cursor control keys, letters, number, everything! Stop the project.

Add a second text box (**textBox2**) to the form. Run the project. Click on this new text box and type some text. Notice the **textBox1_KeyDown** event does not detect any key press. Why not? textBox2 has focus - textBox1 does not. The **textBox2_KeyDown** event is being executed instead (but, there's no code there). Click on textBox1 with the mouse - this gives it focus. Now, press a key. The key detection works again. Remember, for a keyboard event to be detected, the corresponding control must have focus.

KeyPress Event

The **KeyPress** event is similar to the KeyDown event, with one distinction. Many characters in the world of computers have what are called Unicode values. Unicode values are simply numbers (ranging from 0 to 255) that represent all letters (upper and lower case), all numbers, all punctuation, and many special keys like Esc, Space, and Enter. The KeyPress event can detect the pressing of any key that has a corresponding Unicode code. A nice thing about the KeyPress event is that you immediately know what the user input is - no interpretation of any other key(s) is required (like with the KeyDown event). For example, there are different Unicode values for upper and lower case letters. Unicode values are related to ASCII (pronounced “askey”) codes you may have seen in other languages.

The KeyPress event method for a control named **controlName** has the form:

```
private void controlName_KeyPress(object sender, KeyEventArgs
e)
{

      [C# code for KeyPress Event]

}
```

Again, there are two arguments, **sender** and **e**. We are interested in what key was pressed. That information is in the value of **e.KeyChar**. e.KeyChar is a **char** type variable (a type we haven’t seen before), returning a single character, corresponding to the pressed key. The pressed key can be a readable character (letter, number, punctuation) or a non-readable character (Esc, Enter). It’s easy to look at a readable character and know what it is. How can we distinguish one non-readable character from another?

Recall each possible key recognized by the KeyPress event has a Unicode value. If you want to know a Unicode value of a particular character, you simply cast the character to an int type. So, to determine the Unicode (**myCode**) value for a **char** type variable (named **myChar**), use:

```
myCode = (int) myChar;
```

For example:

```
myCode = (int) 'A';
```

returns the Unicode value (myCode) for the upper case A (65, by the way). Notice a character (**char**) type variable is enclosed in a pair of single quotes. To convert a Unicode value (myValue) to the corresponding character, cast the value to a **char** type::

```
myChar = (char) myCode;
```

For example:

```
myChar = (char) 49;
```

returns the character (myChar) represented by a Unicode value of 49 (a "1").

So, to recognize key presses of non-readable characters, known as control keys, we can examine the corresponding Unicode values. Two values we will use are:

Definition	Unicode Value
Backspace	8
<Enter>	13

Let's try an example with the KeyPress event. Start a new project. Add a label control (**label1**) and a text box (**textBox1**) control. Add this code to the **textBox1_KeyPress** event method:

```
private void textBox1_KeyPress(object sender,
KeyPressEventArgs e)
{
    label1.Text = Convert.ToString(e.KeyChar) + " " +
Convert.ToString((int) e.KeyChar);
}
```

Run the project. Press a key. The character typed (if it's printable) and its corresponding Unicode value (a space separates the two values) will appear in the label control. Press as many keys as you like. Notice different values are displayed for upper and lower case letters. Notice not every key has a Unicode value. In particular, press a function key or one of the arrow keys. What happens? Nothing. You can't detect function key or arrow key presses with a KeyPress event. That's why we needed to talk about the KeyDown event. Stop and save the project.

Let's look at a very powerful use of the KeyPress event. Say we have an application where we only want the user to be able to type numbers in a text box. In that text box's KeyPress event, we would like to examine e.KeyChar and determine if it's a number. If it is a number, great! If not, we want to ignore that key! This process of detecting and ignoring unwanted key strokes is called **key trapping**. By comparing the input e.KeyChar with acceptable values, we can

decide (in C# code) if we want to accept that value as input. Key trapping is a part of every sophisticated Visual C# application.

The only question remaining is: if we decide a pressed key is not acceptable, how do we ignore it? We do that using the **e.Handled** property. If an unacceptable key is detected, we set **e.Handled** to **true**. This 'tricks' Visual C# into thinking the KeyPress event has already been handled and the pressed key is ignored. If a pressed key is acceptable, we set the **e.Handled** property to **false**. This tells Visual C# that this method has not been handled and the KeyPress should be allowed (by default, e.Handled is false, allowing all keystrokes).

Go back to the Visual C# example we've been using. Change the code in the example **textBox1_KeyPress** event to this:

```
private void textBox1_KeyPress(object sender,
KeyPressEventArgs e)
{
    if (e.KeyChar < '0' || e.KeyChar > '9')
    {
        e.Handled = true;
        label1.Text = "Not a number";
    }
    else
    {
        e.Handled = false;
        label1.Text = Convert.ToString(e.KeyChar) + " " +
Convert.ToString((int)e.KeyChar);
    }
}
```

Look at what's happening here. If e.KeyChar is outside the range of values from '0' to '9' (again, note **char** types are enclosed in single quotes), **e.Handled** is set to **true**, ignoring this key press. This method will only accept a typed value from 0 to 9. We are restricting our user to just those keys. This comes in handy in applications where only numerical input is allowed. Run the project. Try typing numbers. Try typing non-numerical values - nothing will appear in the text control, indicating the key press was ignored.

Project – Beach Balls

In our final class project, we will build a little video game. Colorful beach balls are dropping from the sky. You maneuver your popping device under them to make them pop and get a point. You try to pop as many balls as you can in one minute. This project is saved as **BeachBalls** in the projects folder (**\BeginVCS\BVCS Projects**).

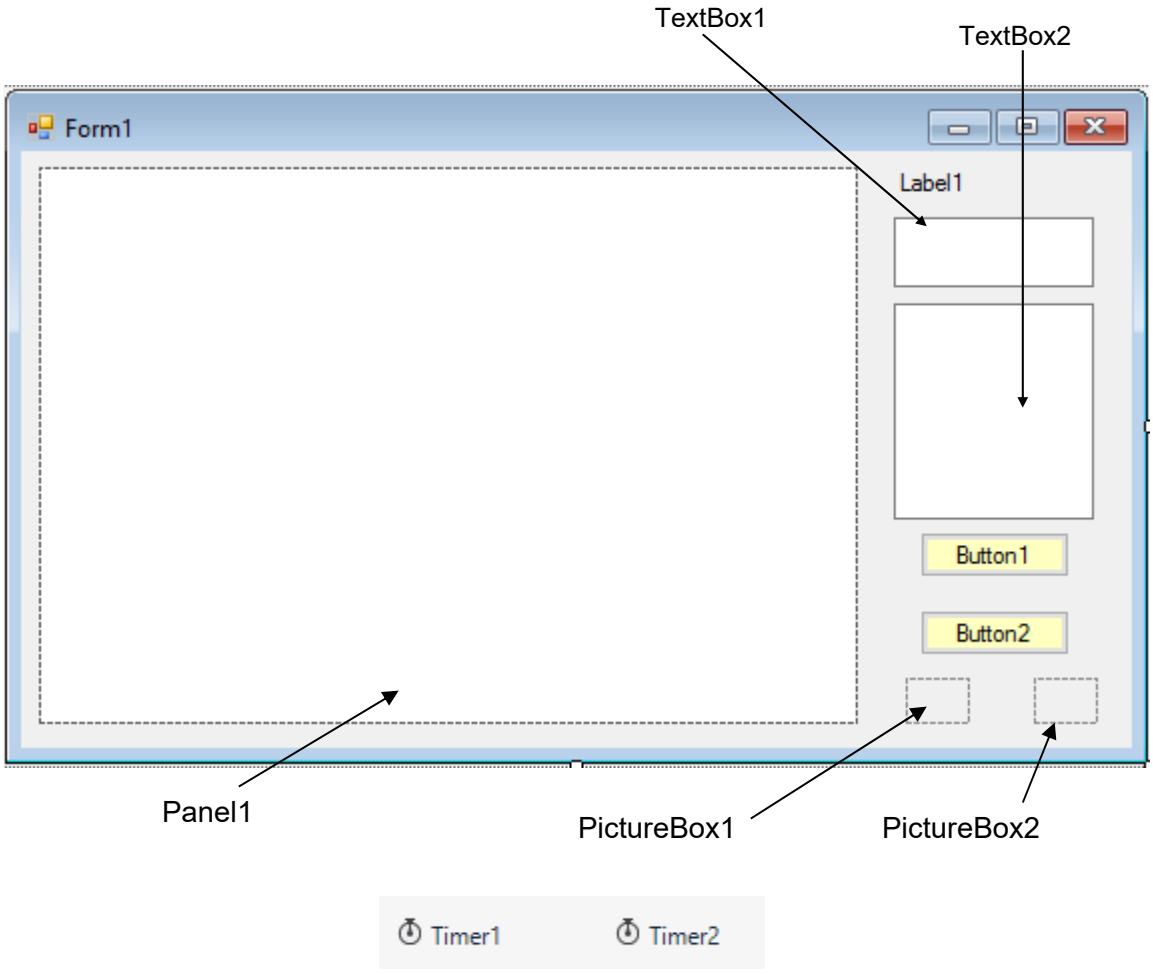
Project Design

All of the game action will go on in a panel control. There will be five possible balls, the image used is contained in a picture box. An picture box control will also hold the “popping arrow” image. This image will be moved using keys on the keyboard. A button will control starting and stopping the game. Another button will stop the program. The current score (number of balls popped) will be displayed in a titled text box.

Place Controls on Form

Start a new project in Visual C#. Place a panel control on the form - make it fairly wide and tall. This is where the game will be played. Place two picture box controls on the form. Add a label control. Add a text box under the label for keeping score. Add larger text box to tell us when the game is over. Add two buttons. And, add two timer controls to use for animation and for timing the overall game.

Try to make your form look something like this when done:



Set Control Properties

Set the control properties using the properties window:

Form1 Form:

Property Name	Property Value
BackColor	Light Red
Text	Beach Balls
FormBorderStyle	FixedSingle
StartPosition	CenterForm
KeyPreview	True (this allows us to detect key presses)

panel1 Panel:

Property Name	Property Value
Name	pnlBeachBalls
BackColor	Light Blue
BorderStyle	FixedSingle

pictureBox1 Picture Box:

Property Name	Property Value
Name	picBall
Image	ball.gif (in **\BeginVCS\BVCS Projects\BeachBalls** folder)
SizeMode	StretchImage
Visible	False

pictureBox2 Picture Box:

Property Name	Property Value
Name	picArrow
Image	arrow.gif (in **\BeginVCS\BVCS Projects\BeachBalls** folder)
SizeMode	StretchImage
Visible	False

label1 Label:

Property Name	Property Value
Name	lblHead
Text	Balls Popped
Font Size	10
Font Style	Bold

textBox1 Text Box:

Property Name	Property Value
Name	txtScore
Text	0
Font Size	18
ReadOnly	True
TabStop	False
TextAlign	Center
BackColor	White
ForeColor	Blue

textBox2 Text Box:

Property Name	Property Value
Name	txtOver
Text	Game Over
Font Size	18
ReadOnly	True
TabStop	False
TextAlign	Center
BackColor	White
ForeColor	Red

button1 Button:

Property Name	Property Value
Name	btnStart
BackColor	Light Yellow
Text	Start

button2 Button:

Property Name	Property Value
Name	btnExit
BackColor	Light Yellow
Text	Exit

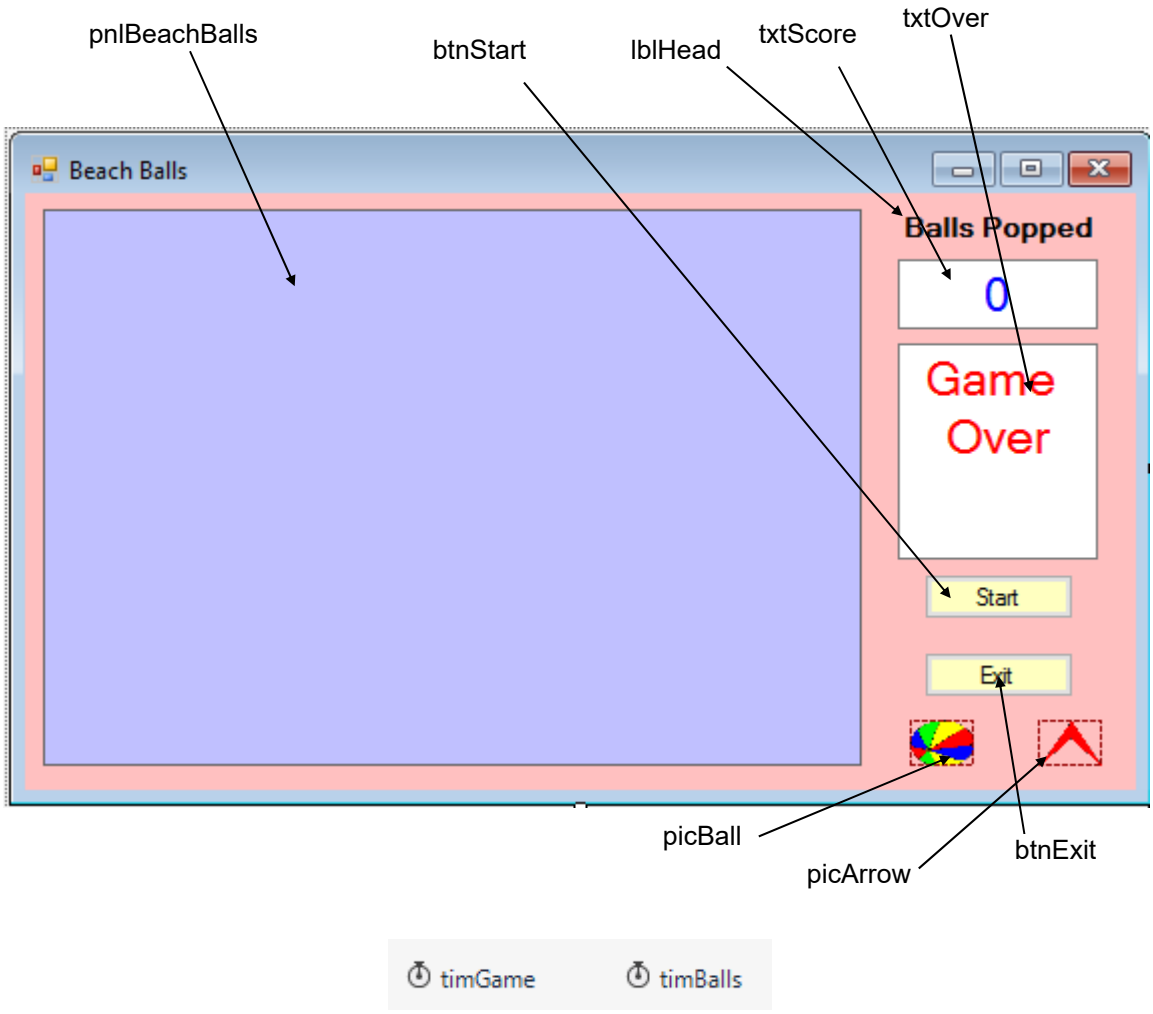
timer1 Timer:

Property Name	Property Value
Name	timBalls
Interval	100

timer2 Timer:

Property Name	Property Value
Name	timGame
Interval	60000

When done setting properties, my form looks like this:



We have used **gif** files for our graphics (the ball and the arrow). With such graphics types, you can select one color to be transparent, allowing the background color to come through. How this is done is beyond the scope of this course. Do a little study on your own using paintbrush programs – PaintShop Pro by JASC (look on the Internet) is a great program for graphics.

Write Event methods

The Beach Balls game is simple, in concept. To play, click the **Start** button. Five balls will drop down the panel, each at a different speed. Use the keyboard to move the arrow. If the arrow is under a ball when a collision occurs, the ball pops and you get a point. Balls reappear at the top after popping or after reaching the bottom of the screen without being popped. You pop as many balls as you can in 60 seconds. At that point, a 'Game Over' message appears. You can click **Start** to play again or click **Exit** to stop the program.

It looks like there are only three events to code, clicking the **Start** button, clicking the **Exit** button, or using **picBalls_KeyDown** to check for arrow key presses. But, recall there are two timer controls on the form. The control named **timBalls** controls the ball animation, updating the panel 10 times a second (Interval is 100). The timer control named **timGame** controls the overall time of the game. It generates a Tick event only once - when the game is over (Interval is 60000 - that's 60 seconds). So, in addition to button clicks and key down events, we need code for two Timer events. There is a substantial amount of C# code to write here, even though you will see there is a lot of repetition. We suggest writing the event methods in stages. Write one method or a part of a method. Run the project. Make sure the code you wrote works. Add more code. Run the project again. Make sure the added code works. Continue adding code until complete. Building a project this way minimizes the potential for error and makes the debugging process much easier. Let's go.

Each ball will occupy a square region. We will compute the size (ballSize) of the ball to fit nicely on the panel. We need array variables to keep track of each ball's location (ballX, ballY) and dropping speed (ballSpeed). We need to know the arrow's size (arrowSize) and position (arrowX). We also need a graphics object to draw the balls (myGraphics) and a blank brush object (blankBrush, for erasing balls). Lastly, we need a random number object (myRandom). Add this code to the **general declarations** area:

```
int ballSize;
int[] ballX = new int[5];
int[] ballY = new int[5];
int[] ballSpeed = new int[5];
int arrowSize;
int arrowX;
Graphics myGraphics;
Brush blankBrush;
Random myRandom = new Random();
```

The array **ballSpeed** holds the five speeds, representing the number of pixels a ball will drop with each update of the viewing panel. We want each ball to drop at a different rate. In code, each speed will be computed using:

```
myRandom.Next(4) + 3
```

Or, it will be a random value between 3 and 6. A new speed will be computed each time a ball starts its trip down the panel. How do we know this will be a good speed, providing reasonable dropping rates? We didn't before the project began. This expression was arrived at by 'trial and error.' We built the game and tried different speeds until we found values that worked. You do this a lot in developing games. You may not know values for some numbers before you start. So, you go ahead and build the game and try all kinds of values until you find ones that work. Then, you build these numbers into your code.

Use this **Form1_Load** method:

```
private void Form1_Load(object sender, EventArgs e)
{
    int x;
    // Have the balls spread across the panel with 20 pixels
borders
    ballSize = (int) ((pnlBeachBalls.Width - 6 * 20) / 5);
    x = 10;
    for (int i = 0; i < 5; i++)
    {
        ballX[i] = x;
        x = x + ballSize + 20;
    }
    // Make arrow one-half the ball size
    arrowSize = (int) (ballSize / 2);
    myGraphics = pnlBeachBalls.CreateGraphics();
    blankBrush = new SolidBrush(pnlBeachBalls.BackColor);
    // Give form focus
    this.Focus();
}
```

In this code, initial horizontal positions for each of the balls are computed (ballX array). The balls are spread evenly across the panel (see if you can understand the code). The arrow is made to be one-half the ball size (arrowSize). Lastly, the graphics object and brush object are created and the form is given focus so KeyDown events can occur.

Add this code to the **Form1_FormClosing** event to dispose of our objects:

```
private void Form1_FormClosing(object sender,
FormClosingEventArgs e)
{
    myGraphics.Dispose();
    blankBrush.Dispose();
}
```

To move the arrow (using DrawImage), we need a **Form1_KeyDown** event method (the panel control does not have a KeyDown event). Make sure you have set the form's **KeyPreview** property to **True**, so the KeyDown event will be "seen." Pick a key that will move the arrow to the left and a key that will move it to the right. I chose **F** for **left** movement and **J** for **right** movement. Why? The keys are in the middle of the keyboard, with F to the left of J, and are easy to reach with a natural typing position. You could pick others. The arrow keys are one possibility. I hardly ever use these because they are always at some odd location on a keyboard and just not "naturally" reached. Also, the arrow keys are often used to move among controls on the form and this can get confusing. The code I use is (change the key code values if you pick different keys for arrow motion):

```
private void Form1_KeyDown(object sender, KeyEventArgs e)
{
    // Erase arrow at old location
    myGraphics.FillRectangle(blankBrush, arrowX,
pnlBeachBalls.Height - arrowSize, arrowSize, arrowSize);
    // Check for F key (left) and J key (right) and compute
arrow position
    if (e.KeyCode == Keys.F)
    {
        arrowX = arrowX - 5;
    }
    else if (e.KeyCode == Keys.J)
    {
        arrowX = arrowX + 5;
    }
    // Position arrow
    myGraphics.DrawImage(picArrow.Image, arrowX,
pnlBeachBalls.Height - arrowSize, arrowSize, arrowSize);
}
```

Notice if the F key is pressed, the arrow (**imgArrow**) is moved to the left by 5 pixels. The arrow is moved right by 5 pixels if the J key is pressed. Again, the 5 pixels value was found by 'trial and error' - it seems to provide smooth motion. After typing in this method, save the project, then run it. Make sure the arrow moves as expected. Press the J key to see it. It should start at the left side of the form (arrowX = 0) since we have not given it an initial position. This is what we

meant when we suggested building the project in stages. Notice there is no code that keeps the arrow from moving out of the panel - you could add it if you like. You would need to detect a left or right border crossing. Stop the project. Now, let's do the button events.

The **btnExit_Click** method is simple, so let's get it out of the way first. It's the usual one line (well, two with the comment) that stops the project:

```
private void btnExit_Click(object sender, EventArgs e)
{
    this.Close();
}
```

Let's outline the steps involved in the **btnStart_Click** event. We use this button for two purposes. It either starts the game (**Text** is **Start**) or stops the game (but, not the program - **Text** is **Stop**). So, the Click event has two segments. If Text is Start, the steps are:

- Hide 'Game Over' message
- Set btnStart Text to "Stop"
- Disable btnExit button
- Clear balls off screen
- Set score to 0
- Initialize each ball's position and speed
- Initialize arrow position
- Give form focus (so KeyDown can be recognized)
- Start the timers

If the Text is Stop when the button is clicked, the program steps are:

- Display ‘Game Over’ message
- Set btnStart Text to “Start”
- Enable btnExit button
- Stop the timers

Look at the **btnStart_Click** event method and see if you can identify all of the outlined steps. Notice the balls are positioned just above the panel and the speeds are set using the formula given earlier:

```
private void btnStart_Click(object sender, EventArgs e)
{
    if (btnStart.Text == "Start")
    {
        // New Game
        myGraphics.Clear(pnlBeachBalls.BackColor);
        txtOver.Visible = false;
        btnStart.Text = "Stop";
        btnExit.Enabled = false;
        txtScore.Text = "0";
        // set each ball off top of panel and give new speed
        for (int i = 0; i < 5; i++)
        {
            ballY[i] = -ballSize;
            ballSpeed[i] = myRandom.Next(4) + 3;
        }
        // Set arrow near center
        arrowX = (int)(pnlBeachBalls.Width / 2);
        myGraphics.DrawImage(picArrow.Image, arrowX,
pnlBeachBalls.Height - arrowSize, arrowSize, arrowSize);
        // Give form focus so it can accept KeyDown events
        this.Focus();
    }
    else
    {
        // Game stopped
        txtOver.Visible = true;
        btnStart.Text = "Start";
        btnExit.Enabled = true;
```

```
    }
    // Toggle timers
    timBalls.Enabled = !(timBalls.Enabled);
    timGame.Enabled = !(timGame.Enabled);
}
```

Save and run the project. There should be no balls displayed. Make sure you get no run-time errors. The arrow should be centered on the panel. Make sure the arrow motion keys (F and J) still work OK. Stop the project.

The **btnStart_Click** event method toggles the two timer controls. What goes on in the two Tick events? We'll do the easy one first. Each game lasts 60 seconds. This timing is handled by the timGame timer. It has an Interval of 60000, which means it's Tick event is executed every 60 seconds. We'll only execute that event once - when it is executed, we stop the game. The code to do this is identical to the code executed if the btnStart button is clicked when its Text is **Stop**. The **timGame_Tick** event method should be:

```
private void timGame_Tick(object sender, EventArgs e)
{
    // 60 seconds have elapsed - stop game
    timBalls.Enabled = false;
    timGame.Enabled = false;
    txtOver.Visible = true;
    btnStart.Text = "Start";
    btnExit.Enabled = true;
}
```

Save the project. Run it. Click **Start**. Play with the arrow motion keys or just sit there. After 60 seconds, you should see the 'Game Over' notice pop up and see the buttons change appearance. If this happens, the timGame timer control is working properly. If it doesn't happen, you need to fix something. Stop the project.

Now, to the heart of the Beach Balls game - the **timBalls_Tick** event. We haven't seen any dropping balls yet. Here's where we do that, and more. The timBalls timer control handles the animation sequence. It drops the balls down the screen, checks for popping, and checks for balls reaching the bottom of the panel. It gets new balls started. There's a lot going on. The method steps are identical for each ball. They are:

- Move the ball.
- Check to see if ball has popped. If so, sound a beep, make the ball disappear, increment score and make ball reappear at the top with a new speed.
- Check to see if ball has reached the bottom without being popped. If so, start a new ball with a new speed.

The steps are easy to write, just a little harder to code. Moving a ball simply involves erasing it at its old location and redrawing it at its new location (determined by the ballY value). To check if the ball has reached the bottom, we use the border crossing logic discussed earlier. The trickiest step is checking if a ball has popped. One way to check for a ball pop is to check to see if the ball image rectangle overlaps the arrow rectangle using the collision detection logic developed earlier. This would work, but a ball would pop if the arrow barely touched the ball. In our code, we modify the collision logic such that we will not consider a ball to be popped unless the entire width of the arrow is within the width of the ball.

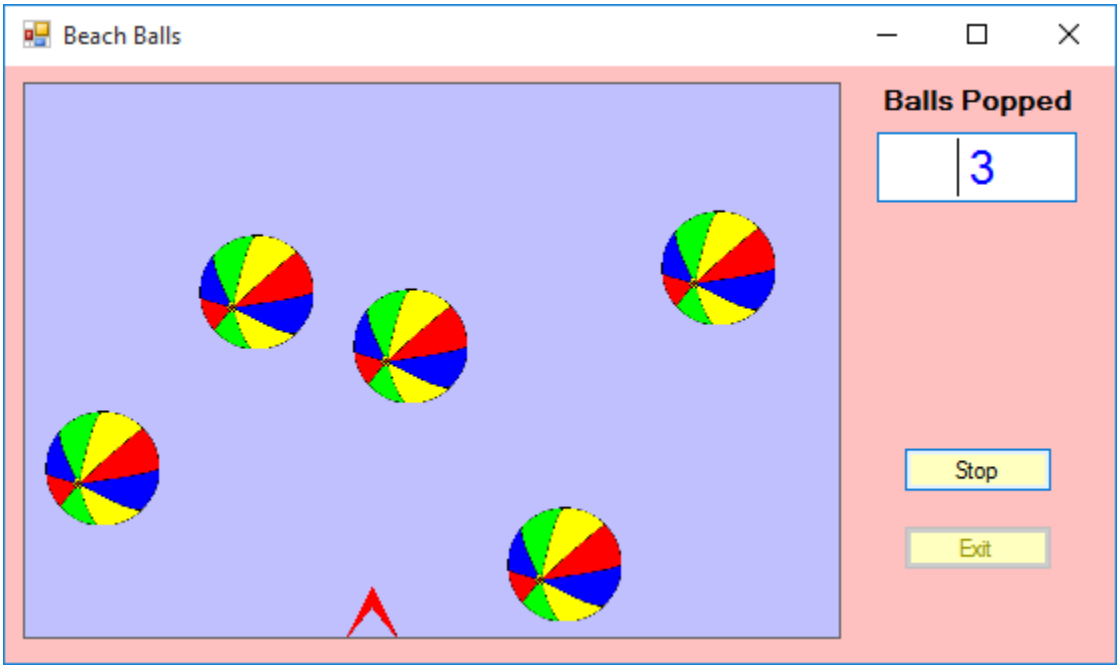
Here's the complete **timBall_Tick** event implementing these steps. The balls are handled individually within the structure of a for loop:

```
private void timBalls_Tick(object sender, EventArgs e)
{
    for (int i = 0; i < 5; i++)
    {
        // erase ball
        myGraphics.FillRectangle(blankBrush, ballX[i],
ballY[i], ballSize, ballSize);
        // move ball
        ballY[i] = ballY[i] + ballSpeed[i];
        // check if ball has popped
        if ((ballY[i] + ballSize) > (pnlBeachBalls.Height -
arrowSize))
        {
            if (ballX[i] < arrowX)
            {
                if ((ballX[i] + ballSize) > (arrowX +
arrowSize))
                {
                    // Ball has popped
                    // Increase score - move back to top
                    System.Media.SystemSounds.Beep.Play();
                    txtScore.Text =
Convert.ToString(Convert.ToInt32(txtScore.Text) + 1);
                    ballY[i] = -ballSize;
                    ballSpeed[i] = myRandom.Next(4) + 3;
                }
            }
        }
        // check for moving off bottom
        if ((ballY[i] + ballSize) > pnlBeachBalls.Height)
        {
            // Ball reaches bottom without popping
            // Move back to top with new speed
            ballY[i] = -ballSize;
            ballSpeed[i] = myRandom.Next(4) + 3;
        }
        // redraw ball at new location
        myGraphics.DrawImage(picBall.Image, ballX[i],
ballY[i], ballSize, ballSize);
    }
}
```

Do you see how all the steps are implemented? We added a Beep statement for some audio feedback when a ball pops.

Run the Project

Run the project. Make sure it works. Make sure each ball falls. Make sure when a ball reaches the bottom, a new one is initialized. Make sure you can pop each ball. And, following a pop, make sure a new ball appears. Make sure the score changes by one with each pop. Here's what my screen looks like in the middle of a game:



By building and testing the program in stages, you should now have a thoroughly tested, running version of Beach Balls. So relax and have fun playing it. Show your friends and family your great creation. If you do find any bugs and need to make any changes, make sure you resave your project.

Other Things to Try

I'm sure as you played the Beach Balls game, you thought of some changes you could make. Go ahead - give it a try! Here are some ideas we have.

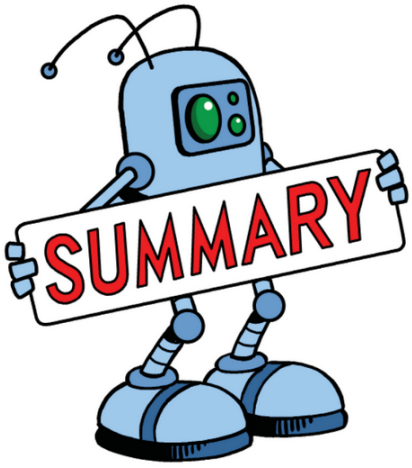
When a ball pops, it just disappears from the screen. Can you think of a more dramatic way to show popping? Maybe change the Image property of the picture box control. Or flash the panel background color.

Add selectable difficulty levels to the game. This could be used to make the game easy for little kids and very hard for experts. What can you do to adjust the game difficulty? One thing you could do is adjust the size of the popping arrow. To pop a ball, the entire arrow width must fit within the width of a ball. Hence, a smaller (narrower) arrow would make it easier to pop balls before they reach the bottom of the picture box. A larger (wider) arrow makes popping harder. The ball dropping speed also affects game difficulty. Slowly dropping balls are easy to pop - fast ones are not. Play with the game to see what speeds would work for different difficulty levels.

Make it possible to play longer games and, as the game goes on, make the game more difficult using some of the ideas above (smaller arrow, faster balls). You've seen this in other games you may have played - games usually get harder as time goes on.

Players like to know how much time they have left in a game. Add this capability to your game. Use a text box control to display the number of seconds remaining. You'll need another timer control with an Interval of 1000 (one second). Whenever this timer's Tick event is executed, another second has gone by. In this event, subtract 1 from the value displayed in the label. You should be comfortable making such a change to your project.

Another thing players like to know is the highest score on a game. Add this capability. Declare a new variable to keep track of the highest score. After each game is played, compare the current score with the highest score to see if a new high has been reached. Add a text box control to display the highest score. One problem, though. When you stop the program, the highest score value will be lost. A new high needs to be established each time you run the project. As you become a more advanced Visual C# programmer, you'll learn ways to save the highest score.



In this final class, we found that the timer control is a key element in computer animation. By periodically changing the display in a panel control, the sensation of motion was obtained. We studied “animation math” - how to detect if an image disappeared from a panel, how to detect if an image crosses the border of a panel, and how to detect if two images (rectangles) collide. We learned how to detect keyboard events. And, you built your first video game.

The **Beginning Visual C#** class is over. You’ve come a long way. Remember back in the first class when you first learned about events? You’re an event expert by now. But, that doesn’t mean you know everything there is to know about programming. Computer programming is a never-ending educational process. There are always new things to learn - ways to improve your skills. Believe it or not, you’ve just begun learning about Visual C#.

Our company, KIDware Software, offers additional Visual C# courses that cover some advanced topics and lets you build more projects. Fun projects are built with step-by-step details. What would you gain from these courses? Here are a few new things you would learn:

- More C# and more controls
- How to do many programming tasks using Visual C#
- Object-oriented programming concepts
- How to distribute your projects (develop SETUP programs)
- How to use the Visual C# debugger
- How to read files from disk and write files to disk (this could be used to save high scores in games)
- How to do more detailed animations
- How to play elaborate sounds (the Beep is pretty boring)
- How to add menus and toolbars to your projects
- How to use your printer

Contact us if you want more information. Or, visit our website - the address is on the title page for this course. Before you leave, try the additional projects. They give you some idea of what you can learn in the next Visual C# class.