8 XNA on Windows Phone

After all the hard work of the previous sections, now would seem a good place to have some fun and play some games. You can write games in Silverlight, but that is not really what it was designed for. XNA on the other hand was built from the ground up to be an efficient and powerful tool for game creation. In this section we are going to take a look at XNA and how to use it to create games for the Windows Phone device.

8.1 XNA in context

XNA is for creating games. It provides a complete ecosystem for game creation, including Content Management (how you get your sound effects, maps and textures into a game) and the game build process (how you combine all the elements into the single distributable game element). We are not going to delve too deeply into all these aspects of the system, instead we are going to focus on the XNA Framework, which is a library of C# objects that is used to create the games programs themselves. XNA allows game developers to use C# to create high performance games on a variety of platforms, including Windows PC, Xbox 360 and Windows Phone. By careful design of your program structure it is possible to reuse the base majority of your game code across these three platforms.

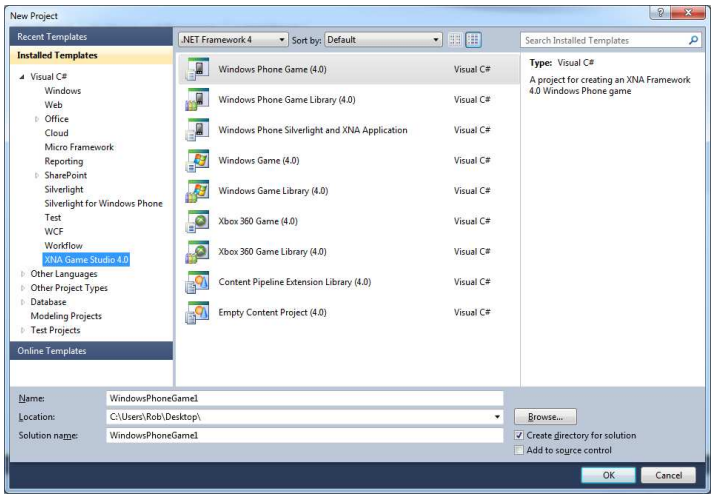
2D and 3D Games

Games can be “2D” (flat images that are drawn in a single plane) or “3D” (a visual simulation of a 3D environment). XNA provides support for both kinds of games and the Windows Phone hardware acceleration makes it capable of displaying realistic 3D worlds. For the purpose of this section we are going to focus on 2D, sprite based, games however. We can create a good gameplay experience this way, particularly on a mobile device.

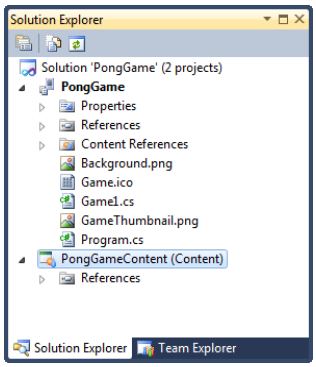
XNA and Silverlight

When we create a Windows Phone project we can create either a Silverlight application or an XNA one. Visual Studio uses the appropriate template and builds the solution with the required elements to make that kind of program. It is not possible to combine these. It would be lovely to think of a system which allowed us to use Silverlight to make the game menus and XNA to provide the gameplay but at the present this is not possible.

8.2 Making an XNA program



The New Project dialog in Visual Studio lets us select Windows Phone Game as the project type. If we want to make an XNA game for Windows PC or Xbox 360 we can do that here as well. One thing worth bearing in mind is that it is possible to use an existing XNA project as the basis of one for another platform, inside the same solution. Visual Studio will copy a project and make a new version which targets the new device. This means that we can take a Windows Phone XNA game and make a version for the Xbox 360 if we wish.



The Solution for an XNA project looks rather like a Silverlight one, but there are some differences. There is a second project in the solution which is specifically for game content. This is managed by the Content Manager which provides a set of input filters for different types of resources, for example PNG images, WAV sound files etc. These resources are stored within the project and deployed as part of the game onto the target platform. When the game runs the Content Manager loads these resources for use in the game. The result of this is that whatever the platform you are targeting the way that content assets are managed is the same as far as you are concerned.

One thing you need to be aware of though is that Windows Phone has a reduced display size compared to the Windows PC and Xbox 360 platforms. The maximum resolution of a Windows Phone game is 800x480 pixels. To make your XNA games load more quickly and take up less space it is worth resizing your game assets for the Windows Phone platform.

If you run the new game project as created above you will be rewarded by a blue screen. This is not the harbinger of doom that it used to be, but simply means that the initial behaviour of an XNA game is to draw the screen blue.

How an XNA Game Runs

If you look at a Silverlight application you will find that a lot of the time the program never seems to be doing anything. Actions are only carried out in response to events, for example when a user presses a button, or a network transaction completes, or a new display page is loaded.

XNA programs are quite different. An XNA program is continuously active, updating the game world and drawing the display. Rather than waiting for an input from a device, an XNA game will check devices that may have input and use these to update the game model and then redraw the screen as quickly as possible. In a game context this makes a lot of sense. In a driving game your car will continue moving whether you steer it or not (although this might not end well). The design of XNA recognises that the game world will update all the time around the player, rather than the player initiating actions.

When an XNA game runs it actually does three things:

1. Load all the content required by the game. This includes all the sounds, textures and models that are needed to create the game environment.
2. Repeatedly run the Game Engine
    - Update the game world. Read controller inputs, update the state and position of game elements.
    - Draw the game world. Take the information in the game world and use this to render a display of some kind, which may be 2D or 3D depending on the game type.

These three behaviours are mapped onto three methods in the Game class that is created by Visual Studio as part of a new game project.

```
partial class Game1 : Microsoft.Xna.Framework.Game
{

    protected override void LoadContent
                                (bool loadAllContent)
    {
    }
    protected override void Update(GameTime gameTime)
    {
    }
    protected override void Draw(GameTime gameTime)
    {
    }
}
```

When we write a game we just have to fill in the content of these methods. Note that we never actually call these methods ourselves, they are called by the XNA framework when the game runs. **`LoadContent`** is called when the game starts. The **`Draw`** method is called as frequently as possible and **`Update`** is called thirty times a second. Note that this is not the same as XNA games on the Windows PC or Xbox 360. In order to reduce power consumption a Windows Phone game only updates 30 times a second, rather than the 60 times a second of the desktop or console version of the game. If you

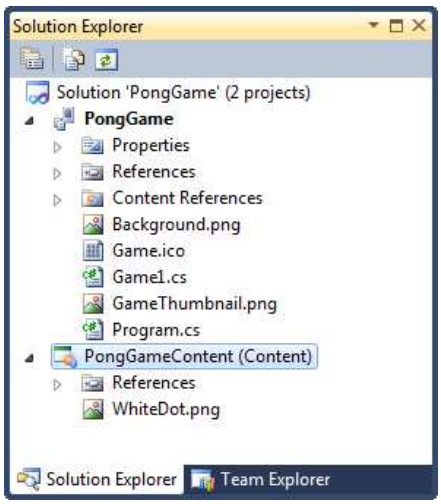
want to write a game which will work on all three platforms you will have to bear this in mind.

Before we can start filling in the **`Update`** and **`Draw`** methods in our game we need to have something to display on the screen. We are going to start by drawing a white ball. Later we will start to make it bounce around the screen and investigate how we can use this to create a simple bat and ball game.

Game Content

We use the word *content* to refer to all the assets that make a game interesting. This includes all the images on the textures in a game, the sound effects and 3D models. The XNA framework Content Management system can take an item of content from its original source file (perhaps a PNG image) all the way into the memory of the game running on the target device. This is often called a *content pipeline* in that raw resources go into one end and they are then processed appropriately and finally end up in the game itself.

The content management is integrated into Visual Studio. Items of content are managed in the same way as files of program code. We can add them to a project and then browse them and manage their properties.



Above you can see an item of content which has been added to a Content project. The item is a PNG (Portable Network Graphics) image file that contains a White Dot which we could use as a ball in our game. This will have the asset name “WhiteDot” within our XNA game. Once we have our asset as part of the content of the game we can use it in games.

We have seen something that looks a bit like this in the past when we added resources to Silverlight games. Those resources have actually been part of Visual Studio solutions as well. However, it is important that you remember that the Content Manager is specifically for XNA games and is how you should manage XNA game content.

Loading Content

The **`LoadContent`** method is called to load the content into the game. (the clue is in the name). It is called once when the game starts running.

```
protected override void LoadContent()
{
    // Create a new SpriteBatch, which can be used to
    // draw textures.
    spriteBatch = new SpriteBatch(GraphicsDevice);

}
```

The first thing that **LoadContent** does is actually nothing to do with content at all. It creates a new **SpriteBatch** instance which will be used to draw items on the screen. We will see what the **SpriteBatch** is used for later.

We can add a line to this method that loads image into our game:

```
ballTexture  = Content.Load<Texture2D>("WhiteDot");
```

The **Load** method provided by the content manager is given the type of resource to be loaded (in this case **Texture2D**). The appropriate loader method is called which loads the texture into the game.

```
Texture2D ballTexture;
```

XNA provides a type called **Texture2D** which can hold a single two-dimensional texture in a game. Later in the program we will draw this on the screen. A game may contain many textures. At the moment we are just going to use one.

Creating XNA Sprites

In computer gaming terms a sprite is an image that can be positioned and drawn on the display. In XNA terms we can make a sprite from a texture (which gives the picture to be drawn) and a rectangle (which determines the position on the screen). We already have the texture, now we need to create the position information.

XNA uses a coordinate system that puts the origin (0,0) of any drawing operations in the top left hand corner of the screen. This is not same as a conventional graph, where we would expect the origin to be the bottom left hand corner of the graph. In other words, in an XNA game if you increase the value of Y for an object this causes the object to move down the screen.

The units used equate to pixels on the screen itself. Most Windows Phones have a screen resolution of 800 x 480 pixels. If we try to draw objects outside this area XNA will not complain, but it won't draw anything either.

XNA provides a **Rectangle** structure to define the position and size of an area on the screen.

```
Rectangle ballRectangle = new Rectangle(
        0, 0,
        ballTexture.Width, ballTexture.Height);
```

The above code creates a rectangle which is positioned at the top left hand corner of the screen. It is the same width and height as the ball texture.

Drawing Objects

Now that we have a sprite the next thing we can do is make the game draw this on the screen. To do this we have to fill in the **Draw** method.

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    base.Draw(gameTime);
}
```

This is the "empty" **Draw** method that is provided by Visual Studio when it creates a new game project. It just clears the screen to blue. We are going to add some code to the method that will draw our ball on the screen. Before we can do this we have to understand a little about how graphics hardware works.

Modern devices have specialised graphics hardware that does all the drawing of the screen. A Windows Phone has a Graphics Processor Unit (GPU) which is given the textures to be drawn, along with their positions, and then renders these for the game player to see. When a program wants to draw something it must send a batch of

drawing instructions to the GPU. For maximum efficiency it is best if the draw requests are be batched together so that they can be sent to the GPU in a single transaction. We do this by using the **SpriteBatch** class to do the batching for us.

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    spriteBatch.Begin();

    // Draw operations go here

    spriteBatch.End();

    base.Draw(gameTime);
}
```

When we come to write our game the only thing we have to remember is that our code must start and end a batch of drawing operations, otherwise the drawing will not work.

We now have our texture to draw and we know how the draw process will work, the next thing to do is position the draw operation on the screen in the correct place. Once we have done this we will have created a *sprite*. The **SpriteBatch** command that we use to draw on the screen is the **Draw** method:

```
spriteBatch.Draw(ballTexture, ballRectangle, Color.White);
```

The **Draw** method is given three parameters; the texture to draw, a **Rectangle** value that gives the draw position and the color of the "light" to shine on the texture when it is drawn. The complete **Draw** method looks like this

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    spriteBatch.Begin();

    spriteBatch.Draw(ballTexture, ballRectangle, Color.White);


    spriteBatch.End();

    base.Draw(gameTime);
}
```

When we run this program we find that the ball is drawn in the top left hand corner of the screen on the phone:

Note that, by default, XNA games expect the player to hold the phone horizontally, in "landscape" mode, with the screen towards the left. We will see later that your game can change this arrangement if it needs to.

```
new Rectangle(
  0, 0,
  ballTexture.Width,
  ballTexture.Height)
```

```
new Rectangle(
  0, 0,       // position
  200,100)    // size
```

```
new Rectangle(
  50, 50,     // position
  60, 60)     // size
```

The first version of our program drew the ball with the same size as the original texture file. However, we can use whatever size and position we like for the drawing process and XNA will scale and move the drawing appropriately, as you can see above.

The solution in *Demo 01White Dot* contains a Windows Phone XNA game that just draws the white dot in the top left hand corner of the display.

Screen Sizes and Scaling

When you make a game it is important that it looks the same whenever it is played, irrespective of the size of the screen on the device used to play it. Windows Phone devices have a particular set of resolutions available to them, as do Windows PC and Xbox 360s. If we want to make a game that looks the same on each device the game must be able to determine the dimensions of the screen in use and then scale the display items appropriately.

An XNA game can obtain the size of the screen from the properties of the viewport used by the graphics adapter.

```
ballRectangle = new Rectangle(
    0, 0,
    GraphicsDevice.Viewport.Width / 20,
    GraphicsDevice.Viewport.Width / 20);
```

This code creates a ball rectangle that will be a 20th of the width of the display, irrespective of the size of the screen that is available.

Updating Gameplay

At the moment the game draws the ball in the same place every time. A game gives movement to the objects in it by changing their position each time they are drawn. The position of objects can be updated in the aptly named **`Update`** method. This is called thirty times a second when the game is running. We could add some code to update the position of the ball:

```
protected override void Update(GameTime gameTime)
{

    ballRectangle.X++;
    ballRectangle.Y++;

    base.Update(gameTime);
}
```

This version of **Update** just makes the X and Y positions of the ball rectangle one pixel bigger each time it is called. This causes the ball to move down (remember that the Y origin is the top of the screen) and across the screen at a rate of one pixel every thirtieth of a second. After fifteen seconds or so the ball has left the screen and will continue moving (although not visible) until we get bored and stop the game from running.

The solution in *Demo 02Moving Dot* contains a Windows Phone XNA game that draws a white dot that slowly moves off the display.

Using floating point positions

The code above moves the ball one pixel each time **Update** is called. This doesn't give a game very precise control over the movement of objects. We might want a way to move the ball very slowly. We do this by creating floating point variables that will hold the ball position:

```
float ballX;
float ballY;
```

We can update these very precisely and then convert them into pixel coordinates when we position the draw rectangle:

```
ballRectangle.X = (int)(ballX + 0.5f);
ballRectangle.Y = (int)(ballY + 0.5f);
```

The above statements take the floating point values and convert them into the nearest integers, rounding up if required. We can also use floating point values for the speed of the ball:

```
float ballXSpeed = 3;
float ballYSpeed = 3;
```

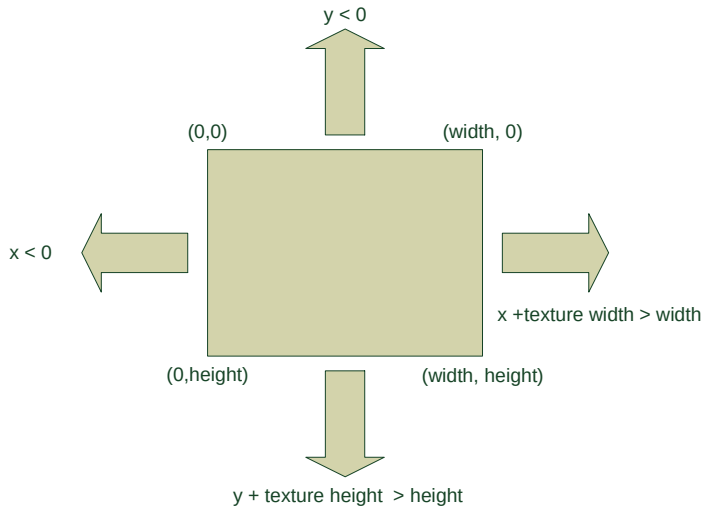
Each time the ball position is updated we now apply the speed values to the ball position:

```
ballX = ballX + ballXSpeed;
ballY = ballY + ballYSpeed;
```

Now that we can position the ball and control the speed very precisely we can think about making it bounce off the edge of the screen.

Making the ball bounce

Our first game just made the ball fly off the screen. If we are creating some kind of bat and ball game we need to make the ball "bounce" when it reaches the screen edges. To do this the game must detect when the ball reaches the edge of the display and update the direction of movement appropriately. If the ball is going off the screen in a horizontal direction the game must reverse the X component of the ball speed. If the ball is going of the top or the bottom of the screen the game must reverse the Y component of the ball speed.



The figure above shows the directions the ball can move off the screen and the conditions that become true when the ball moves in that direction.

```
if (ballX < 0 ||
    ballX + ballRectangle.Width > GraphicsDevice.Viewport.Width)
{
    ballXSpeed = -ballXSpeed;
}
```

The code above reverses the direction of the ball in the X axis when the ball moves off either the left or right hand edge of the display. We can use a similar arrangement to deal with movement in the Y direction.

The solution in *Demo 03 Bouncing Ball* contains a Windows Phone XNA game that draws a ball that bounces around the display.

Adding Paddles to Make a Bat and Ball Game

We now have a ball that will happily bounce around the screen. The next thing that we need is something to hit the ball with.

The paddle is a rectangular texture which is loaded and drawn in exactly the same way as the ball. The finished game uses two paddles, one for the left hand player and one for the right. In the first version of the game we are going to control the left paddle and the computer will control the right hand one. This means that we have three sprites on the screen. Each will need a texture variable to hold the image on the sprite and a rectangle value to hold the position of the sprite on the screen.

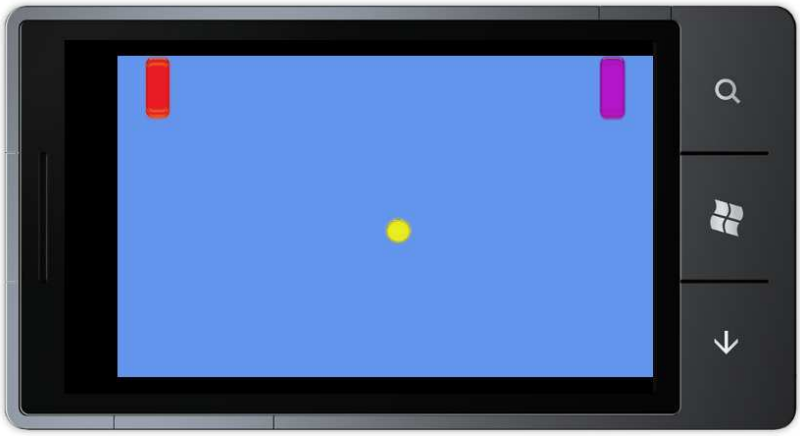
```
// Game World
Texture2D ballTexture;
Rectangle ballRectangle;
float ballX;
float ballY;
float ballXSpeed = 3;
float ballYSpeed = 3;


Texture2D lPaddleTexture;
Rectangle lPaddleRectangle;
float lPaddleSpeed = 4;
float lPaddleY;

Texture2D rPaddleTexture;
Rectangle rPaddleRectangle;
float rPaddleY;

// Distance of paddles from screen edge
int margin;
```

These variables represent our “Game World”. The **`Update`** method will update the variables and the **`Draw`** method will use their values to draw the paddles and ball on the screen in the correct position.

When the game starts running the **`LoadContent`** will fetch the images using the Content Manager and then set up the draw rectangles for the items on the screen.

```
protected override void LoadContent()
{
    // Create a new SpriteBatch, which is used to draw textures.
    spriteBatch = new SpriteBatch(GraphicsDevice);

    ballTexture = Content.Load<Texture2D>("ball");
    lPaddleTexture = Content.Load<Texture2D>("lpaddle");
    rPaddleTexture = Content.Load<Texture2D>("rpaddle");

    ballRectangle = new Rectangle(
        GraphicsDevice.Viewport.Width/2,
        GraphicsDevice.Viewport.Height/2,
        GraphicsDevice.Viewport.Width / 20,
        GraphicsDevice.Viewport.Width / 20);

    margin = GraphicsDevice.Viewport.Width / 20;

    lPaddleRectangle = new Rectangle(
        margin, 0,
        GraphicsDevice.Viewport.Width / 20,
        GraphicsDevice.Viewport.Height / 5);

    rPaddleRectangle = new Rectangle(
        GraphicsDevice.Viewport.Width -
            lPaddleRectangle.Width - margin, 0,
        GraphicsDevice.Viewport.Width / 20,
        GraphicsDevice.Viewport.Height / 5);

    lPaddleY = lPaddleRectangle.Y;
    rPaddleY = rPaddleRectangle.Y;
}
```

This **`LoadContent`** method repays careful study. It loads all the textures and then positions the paddle draw positions each side of the screen as shown below.



When the game starts the ball will move and the paddles will be controlled by the player.

Controlling Paddles Using the Touch Screen

We can use the Windows Phone touch screen to control the movement of a paddle. We can actually get very precise control of items on the screen, but we are going to start very simple. If the player touches the top half of the screen their paddle will move up. If they touch the bottom half of the screen their paddle will move down.

The Touch Panel returns a collection of touch locations for our program to use:

```
TouchCollection touches = TouchPanel.GetState();
```

Each **`TouchLocation`** value contains a number of properties that tell the game information about the location, including the location on the screen where the touch took place. We can use the Y component of this location to control the movement of a paddle.

```
if (touches.Count > 0)
{
    if (touches[0].Position.Y > GraphicsDevice.Viewport.Height / 2)
    {
        lPaddleY = lPaddleY + lPaddleSpeed;
    }
    else
    {
        lPaddleY = lPaddleY - lPaddleSpeed;
    }
}
```

The first statement in this block of code checks to see if there are any touch location values available. If there are it gets the location of the first touch location and checks to see if it is above or below the mid-point of the screen. It then updates the position of the paddle accordingly. If we run this program we find that we can control the position of the left hand paddle.

The solution in *Demo 04 Paddle Control* contains a Windows Phone XNA game that draws a ball that bounces around the display. You can also control the movement of the left hand paddle by touching the top or bottom of the screen area.

You can use the touch panel for much more advanced purposes than the example above. You can track the location and movement of a particular touch event very easily. You can also ask the touch panel to detect gestures that you are interested in.

Detecting Collisions

At the moment the bat and the ball are not “aware” of each other. The ball will pass straight through the bat rather than bounce off it. What we need is a way to determine when two rectangles intersect.

The **`Rectangle`** structure provides a method called **`Intersects`** that can do this for us:

```
if (ballRectangle.Intersects(lPaddleRectangle))
{
    ballXSpeed = -ballXSpeed;
}
```

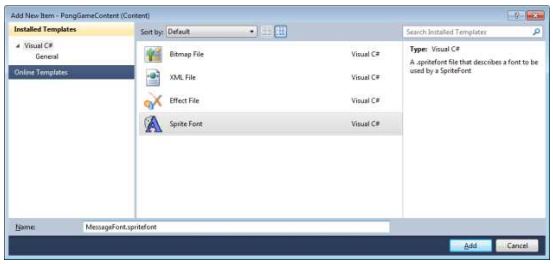
The above code tests to see if the ball has hit the left hand paddle. If it has the direction of movement of the ball is reversed to make the ball bounce off the paddle.

Keeping score

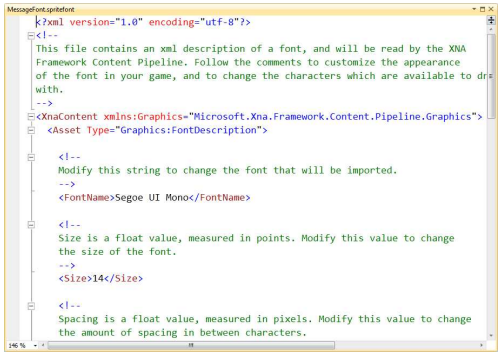
For the game to be worth playing it must also keep a score and display this for the players. Points are scored by hitting the back wall of the opponent player. We already have the code that does this. At the moment it just reverses the direction of movement of the ball, it is a simple matter to add a score variable for each player and increase this each time a “goal” is scored. The final thing we need to be able to do at this point is display the score for the players to see. To do this we need to find out how an XNA game draws text for a player.

Displaying Text

The XNA environment does not provide direct access to text drawing. To write text on the screen a game must load a character font and then use this to draw the characters. The font to be used is an item of game content which we need to add to the game. When the font is added the size and the font design is selected. When the game is built the Content Manager will build a set of character images and add these to the game. The game can then draw the text on the game screen along with the other game items.



The font, called a “SpriteFont”, is created as a new item and given a name. The actual font information is held in an XML file which you can edit with Visual Studio:



This file is where the font size and the name of the font are selected. Above you can see that the font “Segoe UI Mono” is being used with a font size of 14. Further down the XML file you will find settings for bold and italic versions of the font.

Once the **SpriteFont** has been added to the content in a program it can then be loaded when the program runs:

```
SpriteFont font;


font = Content.Load<SpriteFont>("MessageFont");
```

Note that this is exactly the same format as the Load we used to fetch the textures in our game only this time it is being used to fetch a **SpriteFont** rather than a **Texture2D**.

Once we have the font we can use it to draw text:

```
spriteBatch.DrawString( font, "Hello", new Vector2(50,100),
    Color.White);
```

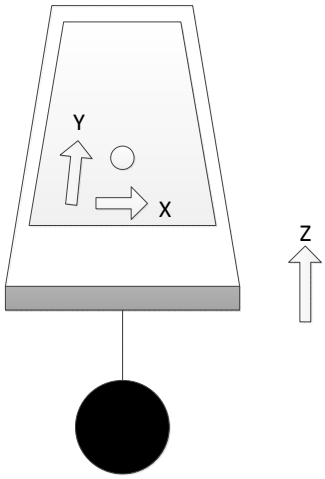
The **DrawString** method is given four parameters. These are the font to use, the string to display, a vector to position the text on the screen and the colour of text to draw. A vector is a way that we can specify a position on the screen, in the example code above the text would be drawn 50 pixels across the screen and 100 pixels down (remember that the origin for Y is at the top of the screen).

The solution in *Demo 05 Complete Pong Game* contains a Windows Phone XNA game that implements a working pong game. The left hand paddle is controlled by the touch panel. The right hand paddle is controlled by an ultra-intelligent AI system that makes it completely unbeatable. Take a look at the code to discover how this top secret technology works. This game is not perfect. Because of the way the ball movement is managed it can sometimes get "stuck" on a paddle or off the edge of the screen. It is up to you to make a completed version of this.

8.3 Player interaction in games

The touch panel is not the only novel input device provided by the Windows Phone. Games can also make use of the accelerometer so that we can create games that are controlled by the tipping of the phone. The accelerometer can measure acceleration in three axes. While it can be used to measure acceleration (you could use your Windows Phone to measure the acceleration of your sports car if you like) it is most often used to measure the orientation of the phone. This is because the accelerometer is acted on by gravity, which gives a constant acceleration of 1 towards the centre of the earth.

You can visualize the accelerometer as a weight on the end of a spring, attached to the back of the phone. The picture below shows us how this might look.



If we hold the phone flat as shown, the weight will hang straight down underneath the phone. If we were to measure the distance in the X, Y and Z directions of the weight relative to the point where it is attached to the phone it would read 0, 0, -1, assuming

that the spring is length 1. The value of Z is -1 because at the moment the position of the weight is below the phone and the coordinate system being used has Z, the third dimension, increasing as we move up from the display.

If you tip the bottom of the phone up so that the far edge of the phone is now pointing towards your shoes the weight would swing away from you, increasing the value of Y that it has relative to the point where the string is attached. If you tip the bottom of the phone down, so that the phone tilts towards you and you can see the screen properly, the weight moves the other way, and the value of Y becomes less than 0. If the phone is vertical (looking a bit like a tombstone) the weight is directly below and in line with it. In this situation the value of Z will be 0, and the value of Y will be -1. Twisting the phone will make the weight move left or right, and will cause the value of X to change.

These are the values that we actually get from the accelerometer in the phone itself. So, at the moment the accelerometer seems to be measuring orientation (i.e. the way the phone is being held) not acceleration. If the phone starts to accelerate in other directions you can expect to see the values in the appropriate direction change as well. Our diagram above actually helps us visualise this too. If we push the phone away from us the weight will "lag" behind the phone for a second until it caught up resulting in a brief change in the Y value. We would see exactly the same effect in the readings from the accelerometer if we did move the phone in this way.

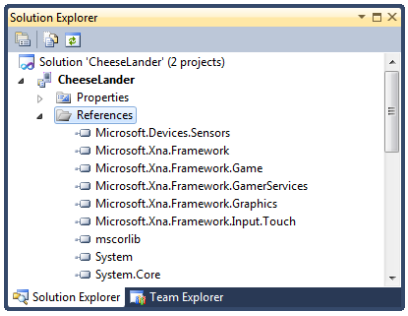
Getting Readings from the Accelerometer Class

When we used the touch panel from XNA we found that we just had to ask the panel for a list of active touch locations. It would be nice if we could do the same with the accelerometer but this is not how it works. The accelerometer driver has been written in a way that makes it useable in Silverlight programs as well. In Silverlight we were used to receiving messages from things when we wanted them to tell us something. The elements on a Silverlight page generate events when they are used and web request cause an event when they have data for us to use.

The accelerometer in Windows Phone works the same way. A program must make an instance of the **`Accelerometer`** class and connect a method to the **`ReadingChanged`** event that the class provides. Whenever the hardware has a new acceleration reading it will fire the event and deliver some new values for our program to use. If the program is written using Silverlight it can use those values directly and perhaps bind them to properties of elements on the screen. If the program is written using XNA it must make a local copy of the new values so that they used by code in the **`Update`** method next time it is called.

Using the Accelerometer Values

Before we can use the **`Accelerometer`** class we need to add the system library that contains it.



We can also add the appropriate namespace:

```
using Microsoft.Devices.Sensors;
```

Now the game can create an instance of the accelerometer, connect an event handler to it and then start the accelerometer running.

```
protected override void Initialize()
{
    Accelerometer acc = new Accelerometer();
    acc.ReadingChanged +=
            new EventHandler<AccelerometerReadingEventArgs>
                                            (acc_ReadingChanged);
    acc.Start();
    base.Initialize();
}
```

We can do this work in the **`Initialise`** method, as shown above. This is another method provided by XNA. It is called when a game starts running. It is a good place to put code to set up elements in our program. Next we have to create our event handler. This will run each time the accelerometer has a new value for the game.

```
Vector3 accelState = Vector3.Zero;

void acc_ReadingChanged
                (object sender, AccelerometerReadingEventArgs e)
{
    accelState.X = (float)e.X;
    accelState.Y = (float)e.Y;
    accelState.Z = (float)e.Z;
}
```

This method just copies the readings from the arguments to the method call into a **`Vector3`** value. The **`Update`** method in our game can then use these values to control the movement of the game paddle:

```
lPaddleY = lPaddleY - (accelState.X * lPaddleSpeed);
```

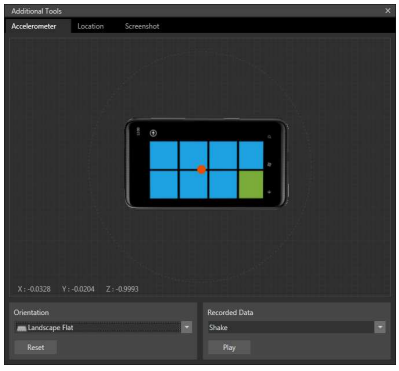
This code is very simple. It uses the X value of the accelerometer state to control the movement up and down of the left hand side paddle. You might thing we should use the Y value from the accelerometer, but remember that our game is being played in *landscape* mode (with the phone held on its side) and the accelerometer values are always given as if the phone is held in *portrait* mode.

The accelerometer, in association with a very simple physics model, can be used to create “tipping” games, where the player has to guide a ball around a maze or past obstacles.

The solution in *Demo 06 Tipping Pong* contains a Windows Phone XNA game that implements a working pong game. The left hand paddle is controlled by tipping the phone to make the paddle move. You will notice that the further you tip the phone the faster the paddle moves, which is just how it should be.

Using the Accelerometer Emulation

If you start the program in the emulator you can use the Additional Tools menu to allow you to test the behaviour of the game by simulating the tipping of the emulated phone.



If you rest the mouse in the top left hand corner of the emulator a menu will appear and by clicking the >> on that menu you can open the Advanced Tools dialog. By selecting the Accelerometer tab and the Landscape Flat orientation you can get a 3D display of the phone which you can tip by dragging the red dot around. This allows you to get a feel of how the accelerometer input works. This is not a true guide to gameplay, but it does allow you to check that the game is using the correct axes.

Threads and Contention

The version of the accelerometer code above will work OK, but code like this can be vulnerable to a problem because of the way the program is written. In the case of this version of the game it would not drastically affect gameplay, but this issue is worth exploring because you may fall foul of similar problems when you start writing programs like these. Mistakes like these can give rise to the worst kind of programming bug imaginable, which is where the program works fine 99.999 per cent of the time but fails every now and then.

I hate bugs like these. I'm much happier when a program fails completely every time I run it because I can easily dive in and start looking for problems. If the program only fails once in a blue moon I have to wait around until I see the fault occur.

The problem has to do with the way that accelerometer readings are created and stored. We have two processes running working at the same time.

- The accelerometer is generating readings and storing them
- The Update method is using these readings to move the paddle

However, the Windows Phone only has one computer in it, which means that in reality only one of these processes can ever be active at any given time. The operating system in Windows Phone gives the illusion of multiple computer processors by switching rapidly between each active process. This is the cause of our problem. Consider the following sequence of events:

1. Update runs and reads the X value of the acceleration
2. The Accelerometer event fires and starts running. It generates and stores new values of X, Y and Z
3. Update reads the Y and Z values from the updated values
4. Update now has "scrambled" data made up of a mix of old and new readings.

In the case of our game we don't have much of a problem here, in that it only uses the X value of the accelerometer anyway. But if we had a more advanced game which used all three values the result would be that every now and then the `Update` method would be given information that wasn't correct. In an even larger and more complex program that used multiple processes this could cause huge problems.

Of course the problem will only happen every now and then, when the timing of the two events was very close together, but the longer the program runs the more chance there is of the problem arising.

The way to solve the problem is to recognise that there are some operations in a program that should not be interruptible. We need a way to stop the accelerometer process from being able to interrupt the **`Update`** process, and vice versa. The C# language provides a way of doing this. It is called a **`lock`**.

A given process can grab a particular lock object and start doing things. While that process has the lock object it is not possible for another process to grab that object. In the program we create a lock object which can be claimed by either the event handler or the code in **`Update`** that uses the accelerometer value:

```
object accelLock = new object();

void acc_ReadingChanged
                (object sender, AccelerometerReadingEventArgs e)
{
    lock (accelLock)
    {
        accelState.X = (float)e.X;
        accelState.Y = (float)e.Y;
        accelState.Z = (float)e.Z;
    }
}
```

The variable **accelLock** is of the simplest possible type, that of object. We are not actually storing any data in this object at all. Instead we are just using it as a token which can be held by one process or another. This is the new code in **Update** that uses the accelerometer reading and is also controlled by the same lock object

```
lock (accelLock)
{
    lPaddleY = lPaddleY - (accelState.X * lPaddleSpeed);
}
```

When a process tries to enter a block of code protected by a **lock** it will try to grab hold of the lock object. If the object is not available the process will be made to wait for the lock to be released. This means that it is now not possible for one method to interrupt another. What will happen instead is that the first process to arrive at the block will get the lock object and the second process will have to wait until the lock is released before it can continue.

Locks provide a way of making sure that processes do not end up fighting over data. It is important that any code protected by a lock will complete quickly, so that other processes do not have to spend a lot of time waiting for access to the lock.

It is also important that we avoid what is called the “Deadly Embrace” problem where process A has obtained lock X and is waiting for lock Y, and process B has obtained lock Y and is waiting for lock X. We can help prevent this by making a process either a “producer” which generates data or a “consumer” which uses data. If no processes are both consumers and producers it is unlikely that they will be stuck waiting for each other.

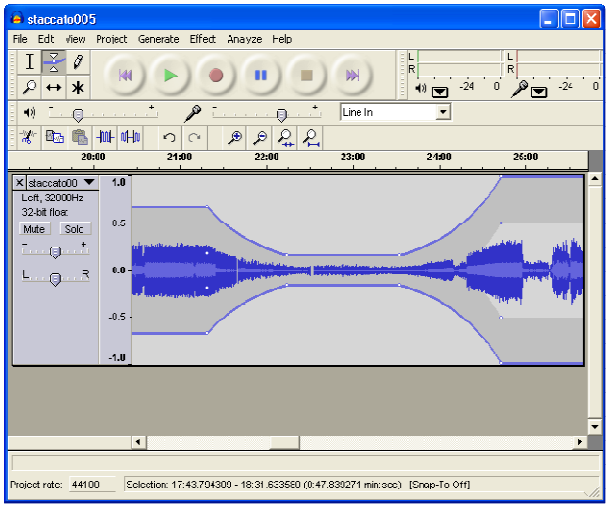
8.4 Adding sound to a game

We can make games much more interesting by adding sound effects to them. As far as XNA is concerned a sound in a game is just another form of game resource. There are two kinds of sounds in a game. There are sound effects which will accompany particular game events, for example the sound of a spaceship exploding when it is hit with a missile, and there is background music which plays underneath the gameplay.

Sound effects are loaded into the game as content. They start as WAV files and are stored in memory when the game runs. They are played instantly on request. Music files can also be supplied as game content but are played by the media player.

Creating Sounds

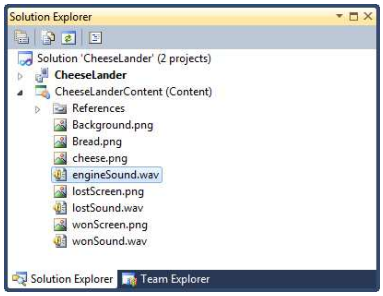
There are many programs that can be used to prepare sound files for inclusion in games. A good one is the program called Audacity. This provides sound capture and editing facilities. It can also convert sound files between different formats and re-sample sounds to reduce the size of sound files. Audacity is a free download from *http://audacity.sourceforge.net*



This shows a sound sample being edited in Audacity.

Sound Samples and Content

As far as an XNA game is concerned, a sound is just another item of content. It can be added alongside other content items and stored and managed by the content manager.



Within the game the **`SoundEffect`** class is used to hold the sound items:

```
// Sound effects
SoundEffect dingSound;
SoundEffect explodeSound;
```

These are loaded in the **`LoadContent`** method, as usual:

```
dingSound = Content.Load<SoundEffect>("ding");
explodeSound = Content.Load<SoundEffect>("explode");
```

A **`SoundEffect`** instance provides a **`Play`** method that triggers the playback of the sound:

```
if (ballRectangle.Intersects(rPaddleRectangle))
{
    ballXSpeed = -ballXSpeed;
    dingSound.Play();
}
```

This is the code that detects a collision of the bat with the ball. The ding sound effect is played when this happens. This is the simplest form of sound effect playback. The sound is made the instant the method is called. The program will continue running as the sound is played. Windows Phone can play multiple sound effects at the same time; the hardware supports up to 64 simultaneous sound channels.

Using the SoundEffectInstance Class

The **`SoundEffect`** class is very easy to use. It is a great way of playing sounds quickly. A game does not have to worry about the sound once it has started playing. However,

sometimes a game needs to do more than just play the sound to completion. Some sound effects have to be made to play repeatedly, change in pitch or move to the left or the right. To get this extra control over a sound a game can create a **SoundEffectInstance** value from a particular sound effect. We can think of this as a handle onto a playing sound. For example we might want to create the sound of a continuously running engine. To do this we start with a **SoundEffect** which contains the sound sample.

```
SoundEffect engineSound;
```

This would be loaded from an item of content which contains the engine sound. Next we need to declare a **SoundEffectInstance** variable:

```
SoundEffectInstance engineInstance;
```

We can ask a **SoundEffect** to give us a playable instance of the sound:

```
engineInstance = engineSound.CreateInstance();
```

Now we can all methods on this to control the sound playback of the sound effect instance:

```
engineInstance.Play();
...
engineInstance.Pause();
...
engineInstance.Stop();
```

Note that the program can stop and start the playback of the sound effect instance, which is not something that is possible with a simple sound effect. A game can also control the volume, pitch and pan of a sound effect instance:

```
engineInstance.Volume = 1; // volume ranges from 0 to 1

engineInstance.Pitch = 0;  // pitch ranges from -1 to +1
                           // -1 – octave lower
                           // +1 – octave higher

engineInstance.Pan = 0;    // pan ranges from -1 to +1
                           // -1 – hard left
                           // +1 – hard right
```

The program can also make a sound loop, i.e. play repeatedly:

```
engineInstance.IsLooped = true;
```

Finally a program can test the state of the sound playback:

```
if (engineInstance.State == SoundState.Stopped)
{
    engineInstance.Play();
}
```

This would start the sound playing again if it was stopped. Using a **SoundEffectInstance** a game can make the pitch of the engine increase in frequency as the engine speeds up and even track the position of the car on the screen.

A **SoundEffectInstance** is implemented as a handle to a particular sound channel on the Windows Phone device itself. We have to be careful that a game doesn't create a very large number of these because this might cause the phone to run out of sound channels.

The solution in *Demo 07 Game with Sounds* contains a Windows Phone XNA game that implements a working pong game with sound effects.

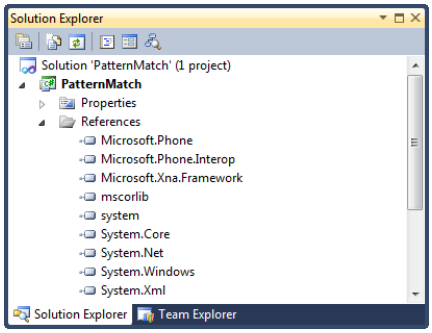
8.5 Playing Sound in a Silverlight Program

A Silverlight program can use the XNA sound playback to provide sound effects. To do this it must include the XNA framework in the Silverlight program, load the sound

from the project resources and then the sound can be played. Then, once the sound has started playing it must repeatedly update the XNA framework as the program runs so that sound playback is managed correctly.

Loading the XNA Namespace

The XNA namespace is not usually included in a Silverlight program but we can add it as a resource to a Silverlight project.



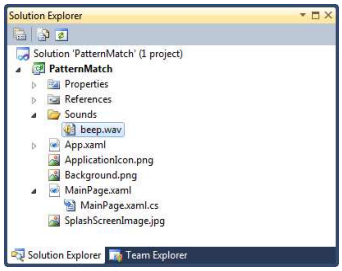
Once we have added the framework we can add using statements to make it easy to use the XNA classes in the Silverlight program:

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
```

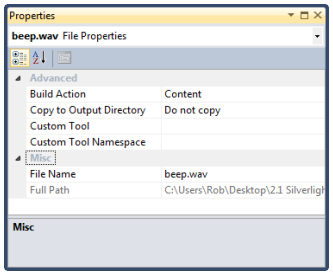
Note that we can't use many of the XNA elements, for example it would be impossible to use a **`SpriteBatch`** or a **`Texture2D`**, but we can use the audio classes.

Adding the Sound Resources

The sound resources are added to our Silverlight project as we would load any other.



Here you can see that I have created a folder called **`Sounds`** which is going to hold all the sound samples in the program. The resources themselves must be added as **`Content`**:



Now that the sound sample is part of the project, the next thing that the Silverlight program must do is load the sound data when the program starts running.

Loading a SoundEffect in a Silverlight program

Because there is no content management in a Silverlight program it must use a different way to load the **SoundEffect** value. A **SoundEffect** can be constructed using a method which loads the sound data from a stream. The following statement will do this for us:

```
SoundEffect beep;
...
beep = SoundEffect.FromStream(
                  TitleContainer.OpenStream("Sounds/beep.wav"));
```

The **TitleContainer** class is part of the Silverlight solution, and can open any of the content items in the project as a stream. The result of this statement is that we now have a **SoundEffect** that we can play in a Silverlight program.

Playing the sound

A Silverlight program plays the sound in exactly the same way as an XNA program.

```
beep.Play();
```

A Silverlight program can also use **SoundEffectInstance** values to play more complicated sounds.

However, there is one important thing that a Silverlight system must do to make sure that sound playback works correctly. You will remember that in an XNA game the **Update** method is called 30 times a second. When an XNA game updates it also updates some of the system elements, including sound playback. A Silverlight program does not have this regular update behaviour, and so we need to add an update call to keep the framework running:

```
FrameworkDispatcher.Update();
```

This must be called around 30 times a second, otherwise sound playback may fail and the program itself may be stopped.

The best way to get this method called regularly is to create a timer to make the call for us. A timer is an object that will generate events at regular intervals. We can bind a method to the timer event and make the method call our despatcher.

Our program can use a **DispatcherTimer**. This is part of the **System.Threading** namespace:

```
using System.Windows.Threading;
```

The code to create the timer and start it ticking is as follows:

```
DispatcherTimer timer = new DispatcherTimer();
timer.Tick += new EventHandler(timer_Tick);
timer.Interval = TimeSpan.FromTicks(333333);
timer.Start();
```

This code creates the timer, connects a method to the **Tick** event, sets the update rate to 30 Hz and then starts the timer running. The **timer_tick** method just has to update the XNA framework.

```
void timer_Tick(object sender, EventArgs e)
{
    FrameworkDispatcher.Update();
}
```

Timers are very useful things. A Silverlight program does have the regular update/draw method calls that an XNA game does, and so you can use a timer to allow you to create moving backgrounds or ticking clocks and the like.

> The solution in *Demo 08 Silverlight with Sound* contains a Windows Phone Silverlight adding machine that plays a sound effect each time a new result is displayed.

8.6 Managing screen dimensions and orientation

We have already seen that a Windows Phone program can be used in more than one orientation. By default (i.e. unless we say otherwise) an XNA game is created that is played in landscape mode with the display towards the left. However, sometimes we want to play games in portrait mode. Our games can tell XNA the forms of orientation they can support and then receive events when the player tips the phone into a new orientation.

The graphics class in a game provides a number of properties that games can use to manage orientation and screen size:

```
graphics.SupportedOrientations = DisplayOrientation.Portrait |
    DisplayOrientation.LandscapeLeft |
    DisplayOrientation.LandscapeRight;
```

To allow a particular orientation we just have to add it to the values that are combined using the arithmetic OR operator, as shown above.

If we want our game to get control when the orientation of the screen changes we can bind a method to the event handler as shown below:

```
Window.OrientationChanged +=
       new EventHandler<EventArgs>(Window_OrientationChanged);
```

The **`Window_OrientationChanged`** method can then resize all the objects on the screen to reflect the new orientation of the game. We have already seen how a program can get the width and height of the display screen. The orientation handler method would use these values to set the new dimensions of the elements in the game.

```
void Window_OrientationChanged(object sender, EventArgs e)
{
    // resize the objects here
}
```

The required orientation is best set in the constructor for the game class.

Selecting a Screen Size

Games can also select a specific screen size:

```
graphics.PreferredBackBufferWidth = 480;
graphics.PreferredBackBufferHeight = 800;
```

If our game does this it forces the display to run at the requested resolution. It also forces the orientation too. The above width and height values would make the game work on portrait mode because the height of the game is greater than the width.

The clever thing about this is that when we set a particular display resolution the Windows Phone display hardware will make sure that we get that resolution, irrespective of the dimensions of the actual screen. In other words our game can operate as if the screen is that size and the hardware will make sure that it looks correct. This is a way you can make sure that your games will always look correct on the current version, and any new versions, of the Windows Phone hardware. No matter what new devices come along and whatever screen sizes they have the game will always look correct because the display will be scaled to fit the requested size.

We can use this to our advantage to improve the performance of our games:

```
graphics.PreferredBackBufferWidth = 240;
graphics.PreferredBackBufferHeight = 400;
```

The above statements ask for a screen which is smaller than the one fitted to most Windows Phone devices. In fact this screen is a quarter the size of the previous one. However, the hardware scaling in the Windows Phone will ensure that the screen looks correct on a larger display by performing scaling of the image. In a fast moving game

this is not usually noticeable and the fact that the display is only a quarter the size of the previous one will make a huge difference to the graphical performance.

Using the Full Screen

At the moment every XNA game that we have produced has been scaled slightly to fit a smaller screen. This is because by default an XNA game does not use the top bar of the phone display. This display area is set aside for status messages. If we want our game to use the entire screen we must explicitly request this:

```
graphics.IsFullScreen = true;
```

A game can set this property to true or false. If it is set to true this means that the game can use the whole of the Windows Phone screen. I always set this to true to give my games the maximum possible amount of display area.

Disabling the Screen Timeout

The owner of a Windows Phone can set a screen timeout for the device. The idea is that if the phone is left doing nothing for a while it will shut down the screen and lock itself to save battery life. The phone monitors the touch screen to detect user input and if there is no input for the prescribed time it will shut down. The phone does not check the accelerometer however, so if we make a game that is entirely controlled by tipping the phone our customers would become very upset when they played the game. They would just be getting to an interesting part and then find that their screen went blank.

A game can disable the screen timeout function by turning off the screen saver in the XNA Guide:

```
Guide.IsScreenSaverEnabled = false;
```

The XNA Guide is the part of XNA that has conversations with the player about their Xbox Live membership and achievements etc. If we tell it to stop the screen saver from appearing this means that the game can continue until the phone battery goes flat. You should use this option with care. If a user gets bored by your game and puts the phone down it will not shut down if the screensaver has been turned off. This may mean that the phone battery will go flat as the game has been left running.

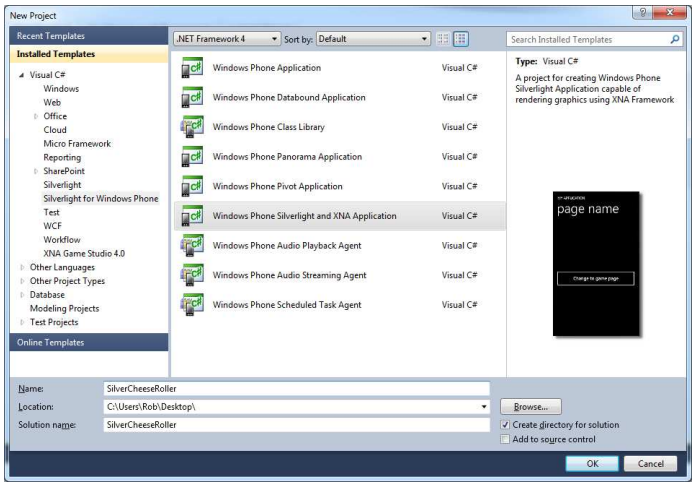
If you do have a game that is controlled by the accelerometer I would also add some gameplay element where a player has to touch the screen every now and then to keep the phone awake. I would much rather have a game that worked in this way rather than one that ran the risk of flattening the phone battery.

The solution in *Demo 09 Full Screen Pong* plays pong on a lower resolution screen that covers the entire phone surface. It is noticeable that from a player's point of view it is hard to tell from the higher resolution version.

8.7 Combining XNA and Silverlight

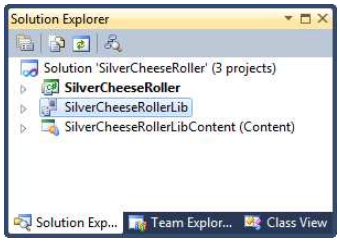
It is possible to create a single Visual Studio solution that contains a Silverlight application and also an XNA game component. This is very useful because Silverlight is very good for creating user interfaces but not very good at games, whereas XNA is very good for game creation but it is hard to create menus. We can use this to create high quality in game menus for games and we could also use this technique to add some 3D graphics to a Silverlight application.

The starting point for a combined application is the New Project dialog in Visual Studio. We can select the template that makes a combined project:

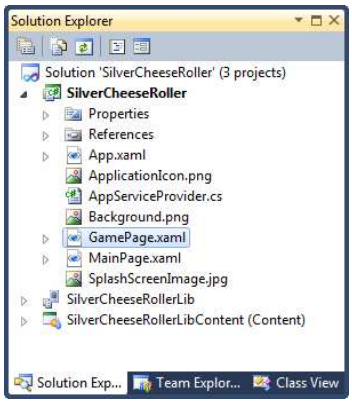


We can create combined applications from both the XNA and the Silverlight templates, in each case the application that is created is the same thing.

When we create the combined application we find that the solution now contains three projects.



The top project is the Silverlight application that provides the starting point. The second project contains a set of library routines that and the third is where the XNA content for the game part of the solution is stored.



When we create a combined application we also get a new Silverlight page that will contain the XNA game. The XNA gameplay is introduced as content on the GamePage. Later we will see how we can combine XNA and Silverlight controls on the same page, for now we will notice that if we look inside the GamePage.xaml page design there is a placeholder that indicates that there are no Silverlight components on this page.

Creating an XNA Game Environment in Silverlight

We have seen that when an XNA game runs a set of methods are called that are where the game initialises, loads its content and then repeatedly performs update (to move on the game world) and draw (to render the game world on the screen). When a game runs inside a Silverlight page it will need to have these behaviours provided for it.

```
protected override void OnNavigatedTo(NavigationEventArgs e)
protected override void OnNavigatedFrom(NavigationEventArgs e)
private void OnUpdate(object sender, GameTimerEventArgs e)
private void OnDraw(object sender, GameTimerEventArgs e)
```

These are the methods in the game page that we need to fill in. The **OnNavigatedTo** method takes the role of **Initialise** and **LoadContent** and the **OnUpdate** and **OnDraw** methods provide updating and drawing. The **OnNavigatedTo** method is always called when the user navigates to a page, and this is where a game should load content and get ready to run. This method also creates a timer that performs the calls of **OnUpdate** and **OnDraw** when the game is active.

Note that this is an XNA program running inside a Silverlight page, which means that it is possible the user may navigate away from the game and move to another page in the application. If this happens the **OnNavigatedFrom** method will be called. This could be used to make the game automatically pause when the user moves away from the screen.

In the template application the game will start when the user presses the

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    NavigationService.Navigate(new Uri("/GamePage.xaml",
                               UriKind.Relative));
}
```

When the button is pressed the GamePage is navigated to and the game begins.

The solution in *Demo 10 Silverlight Pong* contains a Silverlight application that contains a version of the Pong game.

If you press the button the game will start running. If you press the Back button on the phone during the game the Silverlight application will navigate back to the MainPage and the game will stop. Note that the game is restarted each time the page is navigated to. The present version of the game reloads all the textures and game assets each time the page is recreated. It might be more sensible to store the textures and game state in a separate class and cache the data so that the user can return to where they left off, rather than getting a new copy each time.

XNA and Silverlight together

It is possible to actually overlay Silverlight display elements on top of an XNA game. This makes it very easy to add buttons and text to the game by adding Button and TextBlock elements. To put some Silverlight elements on the game screen we first have to create the elements themselves and add them to **GamePage.xaml**:

```
<Grid x:Name="LayoutRoot">
    <StackPanel>
        <TextBlock x:Name="ScoreTextBlock" Text="0:0"
           TextAlignment="Center"
           Style="{StaticResource PhoneTextTitle1Style}" />
        <Button Content="Quit" Name="quitButton"
           Width="480" Click="quitButton_Click" />
    </StackPanel>
</Grid>
```

This creates a **`StackPanel`** containing a button and a textblock. The button will provide a quit behaviour and the textblock will display the game score.

Now that the page has some display elements, we need to get the page rendered on the display. The Silverlight element rendering is performed by a class with the fairly obvious name of `UIElementRenderer`.

```
UIElementRenderer elementRenderer;
```

We create an instance of the renderer when the page is loaded:

```
if (null == elementRenderer)
{
    elementRenderer = new UIElementRenderer(this,
                          (int)ActualWidth, (int)ActualHeight);
}
```

This code checks to see if an element renderer exists and creates one if it is not present. Note that the constructor for this element is given a reference to the page it is rendering and the width and height of the draw area.

Once we have our renderer we can use it in the OnDraw method to render the Silverlight controls:

```
private void OnDraw(object sender, GameTimerEventArgs e)
{

    // Render the Silverlight controls using the UIElementRenderer.
    elementRenderer.Render();

    SharedGraphicsDeviceManager.Current.GraphicsDevice.Clear(
                                        Color.CornflowerBlue);

    spriteBatch.Begin();

    spriteBatch.Draw(ballTexture, ballRectangle, Color.White);

    ...

    spriteBatch.Draw(elementRenderer.Texture, Vector2.Zero,
                                              Color.White);


    spriteBatch.End();
}
```

The final drawing call in the above OnDraw method is where the Silverlight controls are rendered. Note that this means the controls will be drawn on top of the game elements. If we wanted to use the controls as the backdrop for the game we would draw the elements first.

When the XNA game runs the Silverlight elements respond to their inputs and we can connect event handlers to them. Note that this would be the case even if the elements were not drawn.

The solution in *Demo 11 Silverlight Elements* contains a Silverlight application that contains a version of the Pong game with Silverlight buttons and text on it.

If you press the Quit button on the game page it will return you to the previous page.

What We Have Learned

1. The XNA framework provides an environment for creating 2D and 3D games. The games can be created for Windows PC, Xbox 360 or Windows Phone.
2. XNA games operate in a completely different way from Silverlight applications. An XNA game contains methods to load content, update the game world and draw the game world which are called by the XNA framework when a game runs.
3. When Visual Studio creates a new XNA game it makes a class which contains empty LoadContent, Update and Draw methods that are filled in as a game is written. The LoadContent method should load all the required game content. The Update method is called by XNA 30 times a second to update the game content and the Draw method is called to render the game on the screen.
4. Any content (for example images or sounds) added to an XNA game is managed by the Content Management process that provides input filters for different file types and also output processors that prepare the content for deployment to the target device. Within an XNA game the way that content is loaded and used the same irrespective of the target hardware.
5. A sprite is made up of a texture which gives the image to be drawn and a position which determines where to draw the item and its size. The XNA Framework provides objects that can hold this information.
6. When drawing in 2D within the XNA system a given position on the screen is represented by pixel coordinates with the origin at the top left hand corner of the display.
7. The Windows Phone touch panel can provide a low level array of touch location information which can contain details of at least 4 touch events.
8. XNA games can display text by using spritefonts made from fonts on the host Windows PC
9. XNA and Silverlight programs can play sound effects. For greater control of sound effect playback a program can create a SoundEffectInstance value which is a handle to a playing sound.
10. XNA programs on Windows Phone can select a particular display resolution and orientation. If they select a resolution lower than that supported by the phone display the GPU will automatically scale the selected size to fit the screen of the device.

11. XNA programs on Windows Phone can disable the status bar at the top of the screen so that they can use the entire screen area. They can also disable the screensaver so that they are not timed out by the phone.
12. XNA and Silverlight can be combined in a single solution. The XNA game runs within a Silverlight page which can also contain Silverlight components.