Chapter 0: Setting Up

Before you can start programming you need to set your computer up with the proper software. This involves downloading and installing Visual Studio and installing the XNA framework on top of that. After that you will need to familiarize yourself to the Visual Studio environment, learn how to set up projects, and build and run your first XNA program.

Installing Visual Studio

There are multiple versions of Visual Studio that Microsoft makes. The features we will be using are available in all versions and luckily Microsoft makes a free version of Visual Studio called Visual Studio Express. Visual Studio Professional is also available as a free download to students. Go to the Bonus section to learn how to install that version instead of Visual Studio Express, it is highly recommended! If not continue on, however you cannot install both versions of Visual Studio, you can only install one so choose wisely.

Downloading Visual Studio Express

To download Visual Studio Express you will need to go to http://www.microsoft.com/express. From there you can choose from multiple different versions of express for different programming languages. The one we will be using however will be Visual C# 2010 Express. C# is the language that allows us to program XNA games. Here is a direct link for the download if needed: http://go.microsoft.com/?linkid=9709939. This will download the download manager for Visual C# 2010 Express. Silly, isn't it? Go to where you downloaded the file and start the setup. The file should be called vcs_web. Read the dialogues and the setup will download and install Visual Studio.

Starting Visual Studio

After installing Visual Studio you should go to All Programs and launch the program to make sure everything is working correctly. When you first launch Visual Studio it will set up for the first time. It may also ask you to register. If not you need to register anyway as the program will stop working after a certain number of days if you do not. Go to Help > Register Product and follow the steps to register Visual Studio. After that you will be all set to program in C#!

Installing XNA Game Studio

After Visual Studio is set up you still need to install XNA so that you can program games in XNA. To do this you need to go to http://create.msdn.com. This is the main site for XNA development

for Xbox 360, Windows Phone, and Windows. Go to Resources > Downloads and then click Download XNA Game Studio 4.0. Click the orange red download button and the download should start. Here is a direct link: http://download.microsoft.com/download/0/1/4/01483A18-289E-4779-BB5A-0A28DFE18BC5/XNAGS40_setup.exe. Once that is done downloading install XNA Game Studio, this will install on top of Visual Studio. Make sure Visual Studio is closed before doing this however!

Your "First" XNA Program

Now that XNA is installed you can make your first XNA program. Launch Visual Studio if it is not already open and then go to File > New > Project. This is how we will start every new project. A window will open up allowing you to start a new project. Click the XNA Game Studio 4.0 tab and then select Windows Game (4.0). On the bottom of the window you will see Name, Location, and Solution Name textboxes. Keep these default as we will not be using this project anymore after this chapter. Click OK to set up the project. You should now see the project open up with a tab called Game1.cs open near the top. This is the main game class. In this class are all the methods that you can use to make your games. We will not be editing anything right now. All you need to do is press F5. This will start the program. You should see a window pop up called WindowsGame1 with a light blue background. This is the game window where all game events will happen when we build our games. When creating a new XNA project, XNA will set up all the basic things needed before you can actually start making a game. This allows you to focus on programming the game instead of setting up to make a game. Also XNA simplifies the game making process. You will see how easy this is in the next chapter.

Bonus 1: Visual Studio Professional

If you wanted to install Visual Studio Professional instead of just the Express version you are in the right section. This section will teach you how to download a free copy of Visual Studio Professional. Visual Studio Professional has more features and also includes all versions of Visual Studio Express allowing you to program in multiple different languages instead of downloading a separate version of Visual Studio Express each time. To start go to https://www.dreamspark.com. You need to set up an account first. To do that click the create an account link on the right of the screen and fill in the information. The school information is very important as it verifies if you are eligible to be a member of Dreamspark. (Moses Brown students are eligible). After it will ask you to enter your school email address to verify you are eligible. It will send the activation email to your school student account. Click on the link in the email to verify yourself and now you should be signed up for Dreamspark. Back on the homepage there should be a link to download Visual Studio Professional. Follow the steps and install Visual Studio Professional. It will ask you to register. These steps are explained above.

You should now go back and follow the rest of the steps to install XNA and run your first XNA program.

Bonus 2: Hello World!

This will be your first real program! In this program we will make a console window display Hello World. Go to File > New > Project. Then go to Visual C# or Visual C# and then Windows if using Professional and click Console Application. In the Name textbox enter HelloWorld and press ok. This will launch the project. You should see this on the screen.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

This is what Visual Studio generates for you when you make a new console application. You don't need to worry about what any of this means right now. Just type in Console.WriteLine("Hello World!"); into the last set of curly braces. It should look like this.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Basically what this line of code is doing is writing a string to the console. That string contained in the quotation marks is Hello World!. The compiler (the compiler turns this code that we can read into machine code that the computer can run) knows to run this line by the semicolon at the end. This semicolon is very important and if you forget it on certain lines your program will not run! Now we can run this program but this time instead of pressing F5 press Control and F5.

If you just press F5 the console window will pop up, the program will run, and then the console window will close because the program is over. By pressing Control + F5 the window will stay open waiting for a key press to know to close the window. That's it! Your first real program!

Chapter 1: Getting Started

This chapter will show you how simple it can be to start programming in XNA and C#. We will be drawing an image onto the game screen from chapter 0. XNA is so easy to use that this can be done in only five lines of code. You will also learn how import content into your projects so that you can use them in your games.

Setting Up a New Project

Start a new XNA Windows Game like usual. Go to File > New > Project. Select XNA Game Studio. Click Windows Game (4.0) and in the Name box type in GettingStarted and press OK. You will see XNA set up all the things you need to start working. Now run the program by pressing F5 to make sure everything is working correctly and to see where you are starting from.

Importing an Image

To draw an image on our game screen we first need to import an image into Visual Studio. There are multiple ways to bring an image into Visual Studio. The first way is directly importing inside Visual Studio. The second way is to save it or move it into our project files and then include it in the project.

Importing an Image inside Visual Studio

First we need an image to import. Open up your favorite image manipulation software like Paint. Make the canvas size 50 by 50 and make sure it is white. Save this image as a .png somewhere you can access it later and call it square. Now we need to import it. In Visual Studio on the right side of the screen there should be a tab or window called Solution Explorer. If it is a tab mouse over it and click the pin icon so that it stays open. The Solution Explorer lets us see all the files and folders that are in our Solution. A solution holds projects. Projects hold files and folders that we need. You can see that there are two projects in our solution. One called GettingStarted and another one called GettingStartedContent (Content). The second one will hold all the content that we use in our games. Content will mainly be images for us but can also include models, sounds, and spritefonts. We will go over spritefonts later. For now right click on GettingStartedContent (Content) and go to Add > Existing Item. This will open a window so that we can choose the content we want to import into our project. We want to include the square

we made so browse to where you saved the image, select it, then click Add. This will add the content to our project so that we can use it.

Including an Image in a Project

The other way to bring content into our projects is to copy the file to our project files outside Visual Studio and include it in the project. You don't need to follow these steps because our image is already in our project but this is good to know anyway. To do this you can either directly save our content to our project folder or copy paste it into our project folder. You should know where your project folder is but the default location for all Visual Studio projects is: C:\Users\[USERNAME]\Documents\Visual Studio 2010\Projects. This folder holds all projects you make. The location of the content folder would be: GettingStarted\GettingStarted\GettingStartedContent inside the folder where your project is saved. You would save your content into this folder or copy paste it into this folder. When that is done you would click the second button in the Solution Explorer window called show all files. This shows all the files and folders inside our solution folder, not just the files Visual Studio shows to us. Inside the GettingStartedContent (Content) project you would see your content file with 2 other folders. To include the file right click the content you want to include and click Include In Project. This adds the content into Visual Studio so that we can use it.

Constructing the Square

Our square image file should be ready for us to use. Now we need to add code to Visual Studio so that we can use the square in code. On the top of the Game1.cs class file there should be a lot of code.

```
public class Game1 : Microsoft.Xna.Framework.Game
{
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

        public Game1()
        {
            graphics = new GraphicsDeviceManager(this);
            Content.RootDirectory = "Content";
        }
```

Click right after the spriteBatch semicolon and press enter twice. Now type Texture2D square; and do not forget the semicolon! It should look like this:

```
public class Game1 : Microsoft.Xna.Framework.Game
{
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

        Texture2D square;

        public Game1()
        {
            graphics = new GraphicsDeviceManager(this);
            Content.RootDirectory = "Content";
        }
```

If you hover over Texture2D, Visual Studio will tell you what a Texture2D is. Basically a Texture2D can hold data for a 2D image. We are telling Visual Studio now that we want to make room in the computer's memory to hold an image called square. The square we added after Texture2D is the variable name. The variable name square is what we call our texture in code so that we know how to reference it in other parts of the program. Make sure your variable names are descriptive and accurate so that you know what they are later!

Loading the Square

Now we need to load the square image so that we can use it during our game. There is a method (a method contains a block of code that does something) called LoadContent in our file. You may need to scroll down to see it. It will look like this:

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

        // TODO: use this.Content to load your game content here
}
```

This method when called loads the content that you tell it to load. Right now it is loading nothing. We want it to load our square so that we can then draw it on the screen. To do that delete the TODO line and add this in its place: square = Content.Load<Texture2D>("square"); It should look like this.

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

        square = Content.Load<Texture2D>("square");
}
```

So what does this line do? The square at the beginning of the line is the Texture2D variable square that we set up earlier in the program. We are then telling the Content Manger to load the file square as a Texture2D. The Content Manager is built into XNA to help make loading content easier. Notice that when we type square a second time in the line it is surrounded by quotation marks. The quotation marks tell Visual Studio this is a string and what is inside the quotation marks Is what is contained inside the string. This string has the word square in it. Strings can only hold words, letters, characters, or whole sentences. This string it telling Visual Studio to load a texture called square. That texture is the file square.png. Notice we did not need to add the file extension to the string. We just needed the name of the file.

Drawing the Square

Now we have the square loaded into our Texture2D variable. Now we need to draw the square on the screen. Scroll down to the method called Draw. This method will draw all our stuff we want to draw on the screen. It will look like this.

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

        // TODO: Add your drawing code here

        base.Draw(gameTime);
}
```

Now delete the TODO line and type spriteBatch.Begin(); This tells our program to set up the spriteBatch to start drawing textures to the screen. Press enter and type spriteBatch.Draw(square, Vector2.Zero, Color.White); This line tells the spriteBatch to draw what is contained in the Texture 2D variable square. Our square image is contained inside the variable so it will draw our square. Vector2.Zero tells the program where to draw our image. A Vector2 variable holds location data and Vector2.Zero is a location of 0, 0 inside our game screen. 0, 0 in our game screen is the top left hand corner of the window. Color.White tells the spriteBatch to tint our sprite the color white which makes it the original color. Since our square is white if we told the spriteBatch to tint our square another color like red our square would be colored red when we ran our program. The last line we need to add is spriteBatch.End(); and this will tell the spriteBatch we are done drawing items to the screen. This is what our final draw method should look like.

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

        spriteBatch.Begin();
        spriteBatch.Draw(square, Vector2.Zero, Color.White);
        spriteBatch.End();

        base.Draw(gameTime);
}
```

Now we can run our program. Press F5 and you should see a white square in the top left hand corner of our game window. That's it! You have made your first program in XNA! If the program runs into an error check over your code and make sure you typed everything correctly. Everything in C# is case sensitive so test is different from Test and is different from TEST and is different from tEsT. Also make sure to add semicolon's to the lines we typed out!

Bonus 1: Naming Things

As stated earlier you want to name your variables something informative that you can use later on in programs. Also remember everything is case sensitive. You may also not have any spaces in a variable name. If you want to add a space you can use the underscore to replace it, for example Hello_World could be a variable name but Hello World could not be. You may have also noticed that some variables alternate casing, for example spriteBatch in the draw method has its first word start lower case and its second word start uppercase. This is called camel casing. Camel casing is a style that programmers use to make their code easier to read. You can camel case your variables however you want but this book will be lower upper casing variables and uppercasing methods. So this would be a variable name: codingCodingCoding and this would be a method name: ThisRunsLotsOfCode.

Bonus 2: IntelliSense

You may have noticed when you were typing in some code that a box showed up and contained many different little lines of code. This box is IntelliSense. IntelliSense helps you auto complete things when coding. When starting to type Texture2D IntelliSense would highlight Texture. You could then press the down arrow to highlight Texture2D then you can either press Enter or Space to have it auto complete. IntelliSense can work in other ways as well. Say you want to change the color of the square you drew. Click after Color.White in the spriteBatch.Draw line and delete White and the period. Then type the period again and IntelliSense will come up. IntelliSense will then list all the colors that you can color your square. Start typing a color and IntelliSense will narrow down the list with the color you are starting to type. If you type red you will see all the colors that have red in their name. You should see six colors with red in them. Highlight the color OrangeRed and press enter. You will see that Visual Studio has filled where

we typed red with OrangeRed. Now the color of our square will be orange red when we run our program.

Bonus 3: Line Numbers and Errors

We are going to purposely cause an error now. Delete the semicolon after the spriteBatch.Draw line. You should see a red squiggle after the line. This is telling you there is an error. Try running the program now, a window will pop up saying there were errors and if you want to run the last successful version of the program. Click no and on the bottom of the screen you should see a box called Error List. Click the pin tab so that is stays open. We are going to figure out what this box is telling us. The error list will tell us that there is one error. The error description is: ; expected. Visual Studio expected a semicolon somewhere but it did not find it. On the right it will tell you the file where the error happened, the line number, the column number, and the project. All this information is very helpful to find and fix errors. If you double click the description, the error will be highlighted. To fix it add a semicolon at the end of the line. The error will now disappear. If you want to find an error manually with line numbers you first need to make Visual Studio show line numbers. Go to Debug > Options and Settings. Click the arrow next to Text Editor, click All Languages, and then check the box Line Numbers under Display if it is not already checked. Click ok and now you should see line numbers next to every line.

Chapter 2: How It Works

We have drawn an image to the screen. We have seen the code that XNA generates when first starting up a project. But what does it all mean? This chapter will explain what XNA is doing in the background, what each pre generated method does and how you would use it in a game. Before we can do that you need to load up the WindowsGame1 project we made back in chapter 0. There is no point in making another empty project if we already have one. Go to File > Open > Project/Solution and browse to where you saved the WindowsGame1 project. Click and open the WindowsGame1.sln file.

Using

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Media;
```

At the top of the Game1.cs class you should see many lines that start with using. These allow us to use code that somebody else wrote in our projects. If you hover over any of the words you will see namespace and then what that namespace is. The important ones for XNA are the Microsoft.Xna.Framework namespaces. This allows us to use all the code XNA has to build games.

Constructor and Friends

```
namespace WindowsGame1
{
    /// <summary>
    /// This is the main type for your game
    /// </summary>
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

        public Game1()
        {
            graphics = new GraphicsDeviceManager(this);
            Content.RootDirectory = "Content";
        }
```

This is the next section of code you see. You can see that our game has a namespace called WindowsGame1. If we wanted to use any of our code in this project in another project we would import it into the solution and put a using line at the top with our namespace name. We will learn more about this later on. The next three lines are documentation comments. Comments in general are lines that are ignored by the compiler when the project is being built. These lines are very useful though because we can use them to leave notes for ourselves or others reading our code. We will learn more about comments later. Documentation comments are a special type of comment C#. These comments explain what this piece of code does, this comment can also explain what this code does in other projects. When we were writing code to draw our square last chapter you might have seen a box come up. That box explained all the different versions of that method and how it would be used. That is a documentation comment in action. We will learn more about these later. The next line tells us what this file is. Public means that other classes can use variables and methods in this class directly. The opposite of this is private meaning another class cannot use variables and methods in this class directly. In C# variables and methods default to private, be wary of this if you want to use other variables and methods somewhere else. The next two lines are the graphics manager and the sprite batch. You do not need to worry too much about these. What the graphics manager does is manage the graphics for our game. Things like window size and color can be changed with this. The sprite batch lets us draw sprites to the screen. A sprite is another name for a 2D image on the screen. The last thing we see is the constructor. The constructor sets up this class for use.

We see that the graphics manager is loaded to use and that the content manager is pointed to our Content project. We will learn more about constructors when we make our own classes.

Initialize

```
        /// <summary>
        /// Allows the game to perform any initialization it needs to before starting to
run.
        /// This is where it can query for any required services and load any non-graphic
        /// related content.  Calling base.Initialize will enumerate through any
components
        /// and initialize them as well.
        /// </summary>
        protected override void Initialize()
        {
            // TODO: Add your initialization logic here

            base.Initialize();
        }
```

The Initialize method allows us to set up variables before we start loading content. For example if we wanted to set a score to 0 before the game started we would set score equal to zero here. This method also only runs once and this is when the game starts. If we want to run it again we can do that by calling it again. We usually do not use this method though.

Loading Content

```
        /// <summary>
        /// LoadContent will be called once per game and is the place to load
        /// all of your content.
        /// </summary>
        protected override void LoadContent()
        {
            // Create a new SpriteBatch, which can be used to draw textures.
            spriteBatch = new SpriteBatch(GraphicsDevice);

            // TODO: use this.Content to load your game content here
        }
```

The load content method allows us to load all our content before we start the game. This is the only place where we can load content. Loading content like textures, models, and sounds before a game starts is usually the best thing to do because we do not want to slow our game down when we are playing it. This is also called once per game. If we had a lot of content and wanted to load a portion of it at a different time we could call this method again and make it load that content.

Unloading Content

```
/// <summary>
/// UnloadContent will be called once per game and is the place to unload
/// all content.
/// </summary>
protected override void UnloadContent()
{
    // TODO: Unload any non ContentManager content here
}
```

This is the unload content method where we can unload the content we loaded at the beginning of our game. This is called when we close our game. We will never use this method as C# handles the garbage collection of our assets for us.

Update

```
/// <summary>
/// Allows the game to run logic such as updating the world,
/// checking for collisions, gathering input, and playing audio.
/// </summary>
/// <param name="gameTime">Provides a snapshot of timing values.</param>
protected override void Update(GameTime gameTime)
{
    // Allows the game to exit
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    // TODO: Add your update logic here

    base.Update(gameTime);
}
```

This is the update method. Unlike the other methods this method is called once every frame. It will always be called every frame. The default frame time is 1/60 of a second so this method is called 60 times a second. Another name for this is frames per second or FPS. So when you hear this game is running at 60 FPS that means it is updating all its game logic 60 times a second. In here we will put all our game logic. In the next chapter we will use this method to make the square from the last chapter move and bounce around the screen.

Draw

```
/// <summary>
/// This is called when the game should draw itself.
/// </summary>
/// <param name="gameTime">Provides a snapshot of timing values.</param>
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    // TODO: Add your drawing code here

    base.Draw(gameTime);
}
```

This method is the draw method. Almost like the draw method this method will try to run every frame. Try is the key word here; if the game cannot keep up the draw method may not get called every frame. When this happens the game objects will be updated where they appear on the screen will differ because the draw method is not being called every frame. There are ways to make sure it is called every frame but we do not need to worry about them right now. The graphicsdevice.clear line you see is the color the background of our game window is drawn. Right now it is colored cornflower blue. You can change this to any color you want but we will not really be using it.

Chapter 3: Bouncing the Square

We have drawn a square on the screen but just drawing images that do nothing is not really much of a game. A game has movement and to make that movement we need to use the update method. In this chapter we will make our square update and bounce around the screen. First you need to open the project GettingStarted so we can use it.

Tracking Position

Before we can make the square bounce around our game window we need a way to know where the square is on the screen. When we first drew our square we set our position to zero. We however set the position of our square right in the line of code we needed. This is called hard coding and we do not want to do that. We want to be able to keep track of where the square is anywhere in our code. To do that we need to set up a new variable called a Vector2. A Vector2 holds two sets of numbers and is usually used for X, Y coordinates. We want to make a new Vector2 right under the Texture2D line. Type Vector2 squarePosition = new Vector2(0.0f, 0.0f); It should look like this:

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;

    Texture2D square;
    Vector2 squarePosition = new Vector2(0.0f, 0.0f);

    public Game1()
    {
        graphics = new GraphicsDeviceManager(this);
        Content.RootDirectory = "Content";
    }
```

The new keyword means that we are creating a new vector2 variable with the position arguments 0.0f and 0.0f for X and Y respectively. An argument is data that a function asks for. It asks for data though parameters. 0.0f means we are passing a float variable that is set to 0.0. Floats can hold variables that are decimals. When using a float we put an f after the number to denote that it is a float. Since we want the square to move we will also need a way to track the velocity of the square. Add another Vector2 line after our first Vector2. Vector2 squareVelocity = new Vector2(50.0f, 50.0f);

Moving the Square

Now that we know where the square is and how fast the square is moving we need to update the position of the square using the velocity of the square. In the update method we want to add a line that adds the velocity of the square to the position of the square. In the update method delete the TODO comment and add this. squarePosition += squareVelocity; It should look like this.

```
    protected override void Update(GameTime gameTime)
    {
        // Allows the game to exit
        if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
            this.Exit();

        squarePosition += squareVelocity;

        base.Update(gameTime);
    }
```

With this line we are adding the square's velocity to the position of the square. The += sign shows this. This sign would be the same as writing squarePosition = squarePosition + squareVelocity but += shorten's this. Now try running the program. The square does nothing. But we did program it to do something. What's wrong? Let's think this through. We are telling the square to move in the update method but the square is not moving on screen. Let's look at our draw method. When we are drawing the square we have set the position to Vector2.Zero. It

will always be drawing our square at the position 0, 0. We want it to draw at the position squarePosition has stored. Change Vector2.Zero to squarePosition. Now the square flies off the screen. It is also moving very fast. We need to fix a couple things here.

Slowing the Square

There are a couple ways to slow the square down. We can just change the velocity values of the square to a slower speed. We can also multiply the value of the velocity by another number to slow it down. We are going to multiply the square velocity by how fast the game is actually running to slow the square down. We can get the value by using the variable gameTime. The gameTime variable can tell us how fast the update and draw methods are looping. By multiplying the square's velocity by gameTime the square should move at the same speed on different computers. This is also good because we want the experience to be the same for everybody. Add this to the squarePosition += squareVelocity line: * (float)gameTime.ElapsedGameTime.TotalSeconds; What this line does is gets how long it took the game to update between this frame and the last frame and gives us that value in seconds, it then turns that value into a float, denoted by the (float) at the beginning. This is casting, we need to cast the value TotalSeconds into a float because it is outputted as a double originally. By turning it into a float we can now use it. This change will slow the square down because the time between each frame should be around 1/60th of a second so we are multiplying the velocity of the square by a very small number. This is what the update method should look like now:

```
        protected override void Update(GameTime gameTime)
        {
            // Allows the game to exit
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
                this.Exit();

            squarePosition += squareVelocity *
(float)gameTime.ElapsedGameTime.TotalSeconds;

            base.Update(gameTime);
        }
```

Bouncing the Square

Now we have slowed down the square but we still need to make the square bounce. To do this we want the square to change its velocity so that it is moving in the opposite direction when it hits one of the edges of the game window. We first need to figure out where the game window borders are. To do this we are going to use the graphics device manager. We are going to then set those values to an integer variable so that we can use them. Type these four lines right after the squarePosition line in the update method. int minimumX = 0;, int minimumY = 0; int

maximumX = graphics.GraphicsDevice.Viewport.Width; int maximumY = graphics.GraphicsDevice.Viewport.Height; It should look like this:

```
        protected override void Update(GameTime gameTime)
        {
            // Allows the game to exit
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
                this.Exit();

            squarePosition += squareVelocity *
(float)gameTime.ElapsedGameTime.TotalSeconds;

            int minimumX = 0;
            int minimumY = 0;
            int maximumX = graphics.GraphicsDevice.Viewport.Width;
            int maximumY = graphics.GraphicsDevice.Viewport.Height;

            base.Update(gameTime);
        }
```

An int stores only whole numbers. The first two new lines are the left and top borders of the screen. The last two new lines get the right and bottom borders of the screen. We are using the graphics device variable graphics to get access to the variables in the graphics device, we are then accessing the viewport which is the game window and then we are getting the values of the width and the height of the game window. Now we need to check where the square is and make it change directions when it hits the edge of our window. We are going to do this with an if statement. An if statement only runs a section of code if a condition has been met. Our condition will be if the square touches the edge of the screen. The code we will run if that condition is met will be to change the direction of the square. This is what the new update method should look like after we add our code.

```
        protected override void Update(GameTime gameTime)
        {
            // Allows the game to exit
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
                this.Exit();

            squarePosition += squareVelocity *
(float)gameTime.ElapsedGameTime.TotalSeconds;

            int minimumX = 0;
            int minimumY = 0;
            int maximumX = graphics.GraphicsDevice.Viewport.Width;
            int maximumY = graphics.GraphicsDevice.Viewport.Height;

            if (squarePosition.X > maximumX)
            {
                squareVelocity.X *= -1;
                squarePosition.X = maximumX;
            }

            if (squarePosition.X < minimumX)
            {
                squareVelocity.X *= -1;
                squarePosition.X = minimumX;
            }

            if (squarePosition.Y > maximumY)
            {
                squareVelocity.Y *= -1;
                squarePosition.Y = maximumY;
            }

            if (squarePosition.Y < minimumY)
            {
                squareVelocity.Y *= -1;
                squarePosition.Y = minimumY;
            }

            base.Update(gameTime);
        }
```

When writing an if statement you start with if then put the condition into parentheses. The if statement asks if that condition is true so it must be able to be either true or false. Then the code it will run if that condition is true is placed in brackets. For example the first if statement asks if the X position of the square is greater than the maximum X position of our window then we will multiply the X component of the square's velocity by -1 which will reverse its direction. We are then setting the X position of the square to the maximum coordinate of the game window so that the square always stays inside the window. Now try running the project. You will see that the square starts bouncing but does not stay completely inside the window. To fix this we need to look back at the maximum X and Y lines we set earlier. We set those to the width and height of the game window but we did not take into account the size of the square as well. To fix this add change the maximum X and Y lines to this.

```
            int maximumX = graphics.GraphicsDevice.Viewport.Width - square.Width;
            int maximumY = graphics.GraphicsDevice.Viewport.Height - square.Height;
```

The variable square is the texture variable. We are getting the width and height of the square texture from that variable and subtracting that from the width and height of the game window to get the bounce we want. Run the program now and you will see the square bounces correctly. That is all it takes to add movement our games.

Bonus 1: Keywords

There are numerous keywords in C#. Visual Studio notes keywords by coloring them darker blue in our code. We cannot use these keywords as variable names so you could not name an int, int. Some other keywords we have seen are float, private, public, class, and namespace. Others we will use later will include string, else, for, foreach, and bool. There are more however. If you want to see a full list here is a link: http://msdn.microsoft.com/en-us/library/x53a06bb.aspx.

Chapter 4: Controlling the Square

We have gotten the square to move on its own but games also require user input. We need to be able to control the square movement on our own. To do this we are going to read the input from the keyboard to control the square.

Keyboard Control

We first need to tell our program we are going to start keeping track of the keyboard. To do this we use a class in XNA called KeyboardState. KeyboardState records all the key presses that happen in our game. There is also a MouseState and even a GamePadState for the Xbox 360 Controller but for now we will just focus on KeyboardState. Open up the GettingStarted project and add this line in the update method before the line where we add our square's velocity to its position. KeyboardState keyboardState = Keyboard.GetState(); it should look like this.

```
        protected override void Update(GameTime gameTime)
        {
            // Allows the game to exit
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
                this.Exit();

            KeyboardState keyboardState = Keyboard.GetState();

            squarePosition += squareVelocity *
(float)gameTime.ElapsedGameTime.TotalSeconds;
```

This line sets up a new variable called keyboardState and then equates it to the state our program gets from the keyboard. A thing to note is the scope of this new variable. Scope is where the variable can be accessed within our project. Right now the keyboardState variable can only be accessed within the update method. This means that if we wanted to use the variable somewhere else outside this method we would not be able to. There are ways around this however which we will use later on in this chapter.

Key Presses

To check whether or not a key was pressed we will use the if statement again. Our condition will be if a certain key is pressed. The code that will be run is to move the square in the direction we want. The code that goes inside the if statement condition is keyboardState.IsKeyDown(Keys.Up) and the code that is run is squarePosition += 5.0f; Put this if statement right after the line where we set up the keyboardState. If you cannot figure out what the line looks like here it is:

```
        protected override void Update(GameTime gameTime)
        {
            // Allows the game to exit
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
                this.Exit();

            KeyboardState keyboardState = Keyboard.GetState();

            if (keyboardState.IsKeyDown(Keys.Up))
            {
                squarePosition.Y -= 5.0f;
            }

            squarePosition += squareVelocity *
(float)gameTime.ElapsedGameTime.TotalSeconds;
```

This if statement checks if the up key is pressed on the keyboard, if it is pressed then move the square 5 “floats” in the negative Y direction. Remember moving up is negative Y, moving down is positive Y, moving left is negative X, and moving right is positive X. Try writing the other three if statements. They keys you need to use are Down, Left, and Right. Pair those directions with the appropriate X or Y direction and negative or positive movement. This is what it should look like when you are done:

```
        protected override void Update(GameTime gameTime)
        {
            // Allows the game to exit
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
                this.Exit();

            KeyboardState keyboardState = Keyboard.GetState();

            if (keyboardState.IsKeyDown(Keys.Up))
            {
                squarePosition.Y -= 5.0f;
            }

            if (keyboardState.IsKeyDown(Keys.Down))
            {
                squarePosition.Y += 5.0f;
            }

            if (keyboardState.IsKeyDown(Keys.Left))
            {
                squarePosition.X -= 5.0f;
            }

            if (keyboardState.IsKeyDown(Keys.Right))
            {
                squarePosition.X += 5.0f;
            }

            squarePosition += squareVelocity *
(float)gameTime.ElapsedGameTime.TotalSeconds;
```

This should let us control our square. Let's try running our program now. We can control the square but it still wants to move by itself around the screen. We first need to disable that code. We do not want to get rid of it however, we will comment out the code so that the computer will not run it anymore but we can still see it and uncomment it if we want to use it again. To comment it out highlight the square position plus square velocity code and click the comment out button in the toolbar, or hit Control + E, then hit C. Also comment out the reverse velocity lines in each if statement. The code should look like this:

```
        protected override void Update(GameTime gameTime)
        {
            // Allows the game to exit
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
                this.Exit();

            KeyboardState keyboardState = Keyboard.GetState();

            if (keyboardState.IsKeyDown(Keys.Up))
            {
                squarePosition.Y -= 5.0f;
            }

            if (keyboardState.IsKeyDown(Keys.Down))
            {
```

```
                squarePosition.Y += 5.0f;
            }

            if (keyboardState.IsKeyDown(Keys.Left))
            {
                squarePosition.X -= 5.0f;
            }

            if (keyboardState.IsKeyDown(Keys.Right))
            {
                squarePosition.X += 5.0f;
            }

            //squarePosition += squareVelocity *
(float)gameTime.ElapsedGameTime.TotalSeconds;

            int minimumX = 0;
            int minimumY = 0;
            int maximumX = graphics.GraphicsDevice.Viewport.Width - square.Width;
            int maximumY = graphics.GraphicsDevice.Viewport.Height - square.Height;

            if (squarePosition.X > maximumX)
            {
                //squareVelocity.X *= -1;
                squarePosition.X = maximumX;
            }

            if (squarePosition.X < minimumX)
            {
                //squareVelocity.X *= -1;
                squarePosition.X = minimumX;
            }

            if (squarePosition.Y > maximumY)
            {
                //squareVelocity.Y *= -1;
                squarePosition.Y = maximumY;
            }

            if (squarePosition.Y < minimumY)
            {
                //squareVelocity.Y *= -1;
                squarePosition.Y = minimumY;
            }

            base.Update(gameTime);
        }
```

Now run the code and you should be able to control the square without the square trying to move on its own. We should also try to change the color of the square when we hit the spacebar. We are going to change it to a random color when we hit the spacebar. We need to add a couple things to our code before we can start handling the keyboard output though. At the top of our class file we need to add this line right under the squareVelocity line. Random random = new Random(); We also need a line to store the color of the square. Color squareColor = new ColorThis is what it will look like.

```
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

        Texture2D square;
        Vector2 squarePosition = new Vector2(0.0f, 0.0f);
        Vector2 squareVelocity = new Vector2(50.0f, 50.0f);
        Random random = new Random();
        Color squareColor = Color.White;
```

This sets up a new variable called random which can generate pseudo random numbers for us. We will use this to make the square turn a random color. The next line sets up a variable which will hold the color of the square. Right now we set it so that it is its default color. Now in the update method we need to set up another if statement that will check if the space bar is pressed and that will run code that will change the color of the square. Place the if statement after the four if statements that control the square. We want to check if the space bar is pressed. This is the code we want inside the if statement. squareColor = new Color(random.Next(256), random.Next(256), random.next(256)); This is what it should look like.

```
            if (keyboardState.IsKeyDown(Keys.Right))
            {
                squarePosition.X += 5.0f;
            }

            if (keyboardState.IsKeyDown(Keys.Space))
            {
                squareColor = new Color(random.Next(256), random.Next(256),
random.Next(256));
            }

            //squarePosition += squareVelocity *
(float)gameTime.ElapsedGameTime.TotalSeconds;
```

When the space bar is pressed we are setting the squareColor variable to a new color. The color constructor we are using is looking for an integer between 0 and 255 for the three color channels, red, green, and blue. We are using the random variable to access one of the random class's functions called Next which gets an integer. The version of the Next function we are using gets a number between 0 and the number we put as an argument. It is exclusive however so it will get a number between 0 and 255 not 0 and 256. We do this for all three color channels. We also need to make sure we are using the new squareColor variable for the square color when we are drawing the square. The draw method should look like this:

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

            spriteBatch.Begin();
            spriteBatch.Draw(square, squarePosition, squareColor);
            spriteBatch.End();

            base.Draw(gameTime);
        }
```

Now try running the game. We can move the square but when we try to change its color it flashes a lot of colors and only stops when we release space. We only want it to change once color at a time. To do this we need to check if the space bar was released before it can run the line again. To do this we need to add this line at the beginning of our class file right after the squareColor line. KeyboardState previousKeyboardState; It should look like this:

```
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

        Texture2D square;
        Vector2 squarePosition = new Vector2(0.0f, 0.0f);
        Vector2 squareVelocity = new Vector2(50.0f, 50.0f);
        Random random = new Random();
        Color squareColor = Color.White;
        KeyboardState previousKeyboardState;
```

Now we need to adjust our if statement that controls the square's color. The this is what our new condition will look like:

```
            if (keyboardState.IsKeyDown(Keys.Space) &&
previousKeyboardState.IsKeyUp(Keys.Space))
            {
                squareColor = new Color(random.Next(256), random.Next(256),
random.Next(256));
            }
```

Notice we added && and an IsKeyUp check. The && in an in statement mean and. It is now checking if the previousKeyboardState for space is released. There are other operators we will use, they are || which is or, == which is equal (this one checks if something is equal to something else, = sets something equal to something), ! which means not (!= means not equal to), <= and >=. The last thing we need to do is set the previousKeyboardState equal to the current keyboardState so that is has the last updated keyboard state from the frame. We want to do this at the end of the update method so that it does not get affected by any other code. It should look like this:

```
            if (squarePosition.Y < minimumY)
            {
                //squareVelocity.Y *= -1;
                squarePosition.Y = minimumY;
            }

            previousKeyboardState = keyboardState;

            base.Update(gameTime);
        }
```

Now if we run our game when we press space it will only change the color of the square once even if we hold it down. That's it! That is how we add keyboard input to our games.

Bonus 1: Regions

When we were writing our movement code we added a lot of if statements that can get annoying to look at. We can however hide these lines of code easily using a region. A region is set up by adding #region and #endregion before and after code we want to hide. When using #region we also want to give the region a name. Let's region the four if statements that control the square with the name Square Keyboard Movement, it should look like this before we collapse it:

```
        protected override void Update(GameTime gameTime)
        {
            // Allows the game to exit
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
                this.Exit();

            KeyboardState keyboardState = Keyboard.GetState();

            #region Square Keyboard Movement
            if (keyboardState.IsKeyDown(Keys.Up))
            {
                squarePosition.Y -= 5.0f;
            }

            if (keyboardState.IsKeyDown(Keys.Down))
            {
                squarePosition.Y += 5.0f;
            }

            if (keyboardState.IsKeyDown(Keys.Left))
            {
                squarePosition.X -= 5.0f;
            }

            if (keyboardState.IsKeyDown(Keys.Right))
            {
                squarePosition.X += 5.0f;
            }
            #endregion

            if (keyboardState.IsKeyDown(Keys.Space) &&
previousKeyboardState.IsKeyUp(Keys.Space))
            {
                squareColor = new Color(random.Next(256), random.Next(256),
random.Next(256));
            }
```

To collapse the code press the minus button on the left side of the screen on the #region line. The code should shrink and you should only see the name we gave the region. If you want to see the code again just press the plus button on the line. The code will still run in the region if it is collapsed or not, we will just not see all the lines of code we have written.

Chapter 5: Pong

Before this chapter you probably could not have made a game. Right now you probably still think you cannot make a game. Actually with all the things you have learned so far you can make a game. We are going to remake one of the first ever games called Pong. You should recognize the game. There are two paddles and a ball and the ball bounces around the screen and you are trying to score to get points. It is a simple enough first project. Let's get started.

Planning

First we need to plan what we want in our game. We want two rectangle sprites on either sides of the screen that can be controlled using the keyboard. They should not be able to move outside the bounds of our game screen though. We also want a square sprite that will bounce around on the screen. It will also bounce when it hits a paddle. When the ball gets to the left or right edge of the screen we need to show that one player scored The last thing we need is a way to keep track of the score and show a number value somewhere on the screen for both players.

Artwork

For most games you make you will be making your own game art. For this game it is pretty simple. We want one paddle texture and one square texture. We only need one paddle texture because we can just copy the texture to the other player paddle in our code. The paddle texture will be 10 pixels wide and 100 pixels high and the ball texture will be 10 by 10. They should also be colored white, name them paddle and ball respectively, and you should save them somewhere where you can access them later to add them to our content project.

Setting up Our Project

Next we want to create a new solution. File > New > Project. Make sure XNA Windows Game is selected and name it Pong. Now it will open up. First we want to add our content to our content folder. Make sure you remembered where you saved the art files.

Variables

First we need to set up our variables for both paddles and the ball. The variables for both paddles will be very similar but we will distinguish them with player1 and player2 affixed to them. player1 will be the left paddle and player2 will be the right paddle. Set up a Texture2D, a Vector2, and an int variable which will keep score at the beginning of the Game1 class. We also need a Texture2D and Vector2 for the ball. It should look like this:

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
    Texture2D player1Texture;
    Texture2D player2Texture;
    Vector2 player1Position;
    Vector2 player2Position;
    int player1Score = 0;
    int player2Score = 0;

    Texture2D ballTexture;
    Vector2 ballPosition;
```

Notice we did not set up the positions of our paddles yet. We will do this in the load content method to make sure they are in the right place.

Loading Our Textures

Now we need to load the textures for each player paddle and the ball. Double check to make sure the textures are in our Content project; now load the textures. Now we are going to set up the default position of the paddles and the ball. We want to be able to reset this though when a player scores so we are going to write a method to store this code. Right under the LoadContent method write this line: void ResetObjects(). Press enter and add an open curly brace then press enter again and add a closed curly brace. This will be the basis for our method. Now we need to set up the positions of the paddles and the ball. We will use the width and height of the screen to place our objects. It is a lot of slightly complicated code so here it is:

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

            player1Texture = Content.Load<Texture2D>("paddle");
            player2Texture = Content.Load<Texture2D>("paddle");
            ballTexture = Content.Load<Texture2D>("ball");

            ResetObjects();
        }

        void ResetObjects()
        {
            player1Position = new Vector2(50, graphics.GraphicsDevice.Viewport.Height / 2
- player1Texture.Height / 2);
            player2Position = new Vector2(graphics.GraphicsDevice.Viewport.Width -
player2Texture.Width - 50, graphics.GraphicsDevice.Viewport.Height / 2 -
player2Texture.Height / 2);
            ballPosition = new Vector2(graphics.GraphicsDevice.Viewport.Width / 2 -
ballTexture.Width / 2, graphics.GraphicsDevice.Viewport.Height / 2 - ballTexture.Height /
2);
        }
```

In the LoadContent method we are loading the textures we want to use. We are allowed to load the same texture twice. Next we completed our ResetObjects method. First we are setting the player1Position 50 away from the left side of the screen. We are then placing it in the middle of the screen by its height. We are dividing the height of the screen by two and then subtracting it from the texture height divided by two. C# does follow order of operations so you do not need parentheses here to break up operations. We are doing the same thing for the player2 paddle but for the X coordinate we are subtracting the width of the window by the width of the paddle texture then subtracting 50 to make sure it is in the same place as the first paddle. For the ball we are just making sure it is in the middle of the screen. Divide the screen in two and the ball texture in two and subtract. Then in the LoadContent method we are calling the ResetObjects method to set the position of the objects.

Drawing Our Textures

Now we should draw our textures to make sure we placed them in the right places. With textures a good way to know if something is working is to look at it physically on the screen. By now you should know how to draw textures. We need a separate spriteBatch.Draw call for each texture we want to draw. Plug in the right variables and color each texture white. Here is what it should look like:

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

    spriteBatch.Begin();
    spriteBatch.Draw(ballTexture, ballPosition, Color.White);
    spriteBatch.Draw(player1Texture, player1Position, Color.White);
    spriteBatch.Draw(player2Texture, player2Position, Color.White);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

We are drawing the ball texture first because it will be under all the other textures. Remember that for the most part programs are run from top to bottom so the ball texture will be drawn first and then the player1 paddle will be drawn on top of that and then the player2 texture will be drawn last on top of everything else. Run the program and you should see the two paddles and the ball in the correct places.

Controlling Our Paddles

We want to be able to control each paddle with the keyboard. For the player1 paddle we will use the A and Z keys to move it up and down respectively and for the player2 paddle we will use the Up and Down keys. In the Update method set up a keyboardState variable and write

the if statements to control the paddles. When choosing a value for the paddles to move play around with it to see how fast or slow you want to paddle to move, make sure the paddles move at the same speed though. Also set up a region to hide the if statements. In total everything should look something like this:

```
protected override void Update(GameTime gameTime)
{
    // Allows the game to exit
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    KeyboardState keyboardState = Keyboard.GetState();

    #region Paddle Movement
    if (keyboardState.IsKeyDown(Keys.A))
    {
        player1Position.Y -= 2.5f;
    }

    if (keyboardState.IsKeyDown(Keys.Z))
    {
        player1Position.Y += 2.5f;
    }

    if (keyboardState.IsKeyDown(Keys.Up))
    {
        player2Position.Y -= 2.5f;
    }

    if (keyboardState.IsKeyDown(Keys.Down))
    {
        player2Position.Y += 2.5f;
    }
    #endregion

    base.Update(gameTime);
}
```

We also need to make sure the paddles do not move outside the boundaries of the window. We are not going to set up maxX and Y variables here because the ball will have different values. Here is what the if statements should look like:

```
            #region Paddle Boundries
            if (player1Position.Y < 0)
            {
                player1Position.Y = 0;
            }

            if (player1Position.Y > graphics.GraphicsDevice.Viewport.Height -
player1Texture.Height)
            {
                player1Position.Y = graphics.GraphicsDevice.Viewport.Height -
player1Texture.Height;
            }

            if (player2Position.Y < 0)
            {
                player2Position.Y = 0;
            }

            if (player2Position.Y > graphics.GraphicsDevice.Viewport.Height -
player2Texture.Height)
            {
                player2Position.Y = graphics.GraphicsDevice.Viewport.Height -
player2Texture.Height;
            }
            #endregion
```

Bouncing the Ball

Now we need to make the ball bounce around the screen. We want it to start at a random velocity though so the players do not know where it will go. We need to set up a ballVelocity and random variable at the beginning of our class. Our variables should look like this now:

```
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
        Texture2D player1Texture;
        Texture2D player2Texture;
        Vector2 player1Position;
        Vector2 player2Position;
        int player1Score = 0;
        int player2Score = 0;

        Texture2D ballTexture;
        Vector2 ballPosition;
        Vector2 ballVelocity;
        Random random = new Random();
```

Notice we do not set up the ballVelocity variable yet. We want to give the ball a random velocity, we will do this in the reset objects method. This is what the ResetObjects method should look like now:

```
        void ResetObjects()
        {
            player1Position = new Vector2(50, graphics.GraphicsDevice.Viewport.Height / 2
- player1Texture.Height / 2);
            player2Position = new Vector2(graphics.GraphicsDevice.Viewport.Width -
player2Texture.Width - 50, graphics.GraphicsDevice.Viewport.Height / 2 -
player2Texture.Height / 2);
            ballPosition = new Vector2(graphics.GraphicsDevice.Viewport.Width / 2 -
ballTexture.Width / 2, graphics.GraphicsDevice.Viewport.Height / 2 - ballTexture.Height /
2);

            ballVelocity = new Vector2(random.Next(2, 5), random.Next(2, 5));
        }
```

We set our velocity between 2 and 4 for both the X and the Y. This makes sure the velocity is not 0 in one or both directions. Now in the Update method add this line right after our keyboardState line. ballPosition += ballVelocity. We do not need to slow down the ball because we have already done that. We now need to write if statements to bounce the ball off the top and bottom of the screen. We also need to write if statements to reset the objects if the ball touches the left or right of the screen and to add a point to the correct player. This is what everything should look like when it is done:

```
            #region Ball Boundries and Scoring
            if (ballPosition.Y < 0)
            {
                ballVelocity.Y *= -1;
                ballPosition.Y = 0;
            }

            if (ballPosition.Y > graphics.GraphicsDevice.Viewport.Height -
ballTexture.Height)
            {
                ballVelocity.Y *= -1;
                ballPosition.Y = graphics.GraphicsDevice.Viewport.Height -
ballTexture.Height;
            }

            if (ballPosition.X < 0)
            {
                ResetObjects();
                player2Score++;
            }

            if (ballPosition.X > graphics.GraphicsDevice.Viewport.Width -
ballTexture.Width)
            {
                ResetObjects();
                player1Score++;
            }
            #endregion
```

When the ball reaches the left or right side of the screen we are resetting the game objects by calling our method. Notice we are also adding the correct score. ++ means to increment that value by one. If player1's value was 1 now it becomes 2.

Collisions

Our game is almost finished but the ball cannot collide with either of the paddles. To add collisions to the game we need to add rectangles to the game. Rectangles can tell other objects where they are and if they intersect with them. At the beginning of our class we need to add two rectangles for the paddles and one for the ball.

```
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
        Texture2D player1Texture;
        Texture2D player2Texture;
        Vector2 player1Position;
        Vector2 player2Position;
        int player1Score = 0;
        int player2Score = 0;

        Texture2D ballTexture;
        Vector2 ballPosition;
        Vector2 ballVelocity;
        Random random = new Random();

        Rectangle player1Rectangle;
        Rectangle player2Rectangle;
        Rectangle ballRectangle;
```

Now we will set up these rectangles in the update method. After the keyboardState line but before the ballPosition line add this:

```
            player1Rectangle = new Rectangle((int)player1Position.X,
(int)player1Position.Y, player1Texture.Width, player1Texture.Height);
            player2Rectangle = new Rectangle((int)player2Position.X,
(int)player2Position.Y, player2Texture.Width, player2Texture.Height);
            ballRectangle = new Rectangle((int)ballPosition.X, (int)ballPosition.Y,
ballTexture.Height, ballTexture.Width);
```

The Rectangle constructor looks for the X and Y coordinates of the object and the width and height of the object. We are casting the X and Y coordinates to integers because the constructor wants an integer and the Vector2 for the positions is a float. Copy these lines to the end of the method. Now we need to set up the collisions between each paddle and the ball. We will use if statements here. Here is what it should look like:

```
if (ballRectangle.Intersects(player1Rectangle))
{
    ballVelocity.X *= -1;
    ballPosition.X = player1Position.X + player1Texture.Width;
}

if (ballRectangle.Intersects(player2Rectangle))
{
    ballVelocity.X *= -1;
    ballPosition.X = player2Position.X - ballTexture.Width;
}
```

There is a method Rectangles have called Intersects and its argument is another Rectangle. We check to see if one rectangle intersects another rectangle. If it does we reverse the velocity of the ball in that direction. Then we set the position of the ball outside the paddle so that it will bounce correctly. Notice how we add and subtract values to get the right position. For the left paddle the ball is hitting the right edge so we take the X coordinate of the paddle and add the width of the paddle to get the right edge. For the right paddle we take the X coordinate of the paddle and subtract the ballTexture width to get the right edge of the ball. Now we should be able to play our game. We are missing one thing however which is the score.

Displaying the Score

The last thing we need to do is display the score. We are going to use a new type of content called a sprite font. To use a sprite font right click the content project and go to Add > New Item, select Sprite Font and name it PongFont. Make sure the extension stays when you are naming it. When you click add a new file will open up. This file holds all the properties for our font in XML. It should be pretty easy to figure out what is happening. FontName is the name of the font we want to use, leave it default. Size it the size of the font, put 48 as the size. Spacing is how far apart the want the characters to be, leave it at default spacing. Kerning is not important. Style is whether we want it to be regular, bold, and or italicized, leave it default. Now go back to our Game1.cs file and add these lines to the top of the class.

```
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
        Texture2D player1Texture;
        Texture2D player2Texture;
        Vector2 player1Position;
        Vector2 player2Position;
        int player1Score = 0;
        int player2Score = 0;

        Texture2D ballTexture;
        Vector2 ballPosition;
        Vector2 ballVelocity;
        Random random = new Random();

        Rectangle player1Rectangle;
        Rectangle player2Rectangle;
        Rectangle ballRectangle;

        SpriteFont pongFont;
        Vector2 player1TextPosition;
        Vector2 player2TextPosition;
```

The SpriteFont variable holds the information for our font. The two Vector2's will hold the position information for the scores. Now in the LoadContent method add this:

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

            player1Texture = Content.Load<Texture2D>("paddle");
            player2Texture = Content.Load<Texture2D>("paddle");
            ballTexture = Content.Load<Texture2D>("ball");

            pongFont = Content.Load<SpriteFont>("PongFont");

            player1TextPosition = new Vector2(100, 25);
            player2TextPosition = new Vector2(graphics.GraphicsDevice.Viewport.Width -
100, 25);

            ResetObjects();
        }
```

We are loading the SpriteFont like we do for textures. We set pongFont equal to the correct file in our content project. We are using PongFont and instead of Texture2D we are loading it as a SpriteFont. We also set up the positions for the scores. We already have our scores incrementing so we just need to draw the scores on screen. Instead of using spriteBatch.Draw we will use spriteBatch.DrawString. This method will draw a string on the screen. Try figuring out what variables go where. Here it is if you need it:

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

            spriteBatch.Begin();
            spriteBatch.Draw(ballTexture, ballPosition, Color.White);
            spriteBatch.Draw(player1Texture, player1Position, Color.White);
            spriteBatch.Draw(player2Texture, player2Position, Color.White);
            spriteBatch.DrawString(pongFont, player1Score.ToString(),
player1TextPosition, Color.White);
            spriteBatch.DrawString(pongFont, player2Score.ToString(),
player2TextPosition, Color.White);
            spriteBatch.End();

            base.Draw(gameTime);
        }
```

We first tell the method we are using the pongFont file. We are then telling it to draw a string of the score. We are using a method in int called ToString which turns an integer value into a string. We then are drawing it in the correct position using our TextPosition variables we set up earlier. We then are coloring the text white. If you run your game now you should see everything working. That's it! Your first real game in XNA!

Chapter 6: Rewriting Pong

That's right. We are going to rewrite our code for Pong. There are many ways to improve the code using classes. Classes make it much easier to make multiple copies of an object. It also distributes the code and helps eliminate writing code that is almost the same. We are going to create a class for sprites, a class for our paddles, and a class for the ball. You will see we will clean up our code by using classes.

Adding a New Class

The first thing we are going to do is add a new class called Sprite. Our sprite class will let us create multiple sprite objects. Sprites are just textures that we draw on the screen that have properties and attributes. Open the Pong project and go to the Solution Explorer. Right click the Pong project and go to Add > Class, then name it Sprite.cs and click add. It will open up the new class. The first thing we need to do is copy over our using statements so we can use XNA stuff in this class. Go back to the Game1.cs class copy all the using statements, go to the Sprite.cs class and select the usings there and paste the usings we want over.

Class Variables

The next thing we need to do is make our class public. Add public before the line class Sprite so that we can use the class in our Game1.cs class. Now we want to add our class variables. We actually only need to add three, type these lines at the beginning of our class file. public Texture2D tex, public Vector2 pos, public Rectangle rect. This is what the class should look like at this point:

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Media;

namespace Pong
{
    public class Sprite
    {
        public Texture2D tex;
        public Vector2 pos;
        public Rectangle rect;
    }
}
```

The variable tex will replace the texture lines for our paddles and ball, the pos variable will replace the position variable, and the rect variable will replace the rectangle variable.

Class Constructor

Now we will add our constructor. We will use this constructor to set up our three variables when we make a new variable with this class. Add public Sprite(Texture2D loadedTex) as a method to the new class. In that method add tex = loadedTex, pos = Vector2.Zero, rect = new Rectangle((int)pos.X, (int)pos.Y, tex.Width, tex.Height). It should look like this:

```
namespace Pong
{
    public class Sprite
    {
        public Texture2D tex;
        public Vector2 pos;
        public Rectangle rect;

        public Sprite(Texture2D loadedTex)
        {
            tex = loadedTex;
            pos = Vector2.Zero;
            rect = new Rectangle((int)pos.X, (int)pos.Y, tex.Width, tex.Height);
        }
    }
}
```

When we create a new version of our class our constructor will ask for a Texture2D as a parameter. It will also set that parameter as our actual tex variable, the position of the sprite will be 0, 0, and we will set up the rectangle for the sprite.

Inheritance

We are done with our Sprite class for now but we need to make two more classes for the Player and Ball. Create two more classes with the name Player and Ball and copy over the using statements. We want these classes to have access to the variables and functions we wrote in the Sprite class without rewriting them in the other two classes. To get around this we are going to use inheritance. This is a very important part of object oriented programming (OOP), classes are also part of this. To inherit the variables and functions of the Sprite class first change Player class to public (it does not have to be public but we need it to be anyway), add a colon after Player and then type Sprite. It should look like this:

```
namespace Pong
{
    public class Player : Sprite
    {
    }
}
```

Now we want to write our Player class specific variables. We can use these variables in the Player class but not the Sprite class. Inheritance only works one way. The only variable we need to add for now is score. Add public int score; We do however need to add a constructor to our Player class. Add public Player(Texture2D texture) : base(texture) and close of the constructor with curly braces. This constructor method Is different than our first one. This is because we need to pass a texture to the inherited class. The other thing we need to add in the Player class is to set score = 0; Our class should look like this now:

```
namespace Pong
{
    public class Player : Sprite
    {
        public int score;

        public Player(Texture2D texture) : base(texture)
        {
            score = 0;
        }
    }
}
```

We are going to do about the same thing for our Ball class. Set it to public, add Sprite as the inherited class, add a Vector2 vel variable. Add a constructor that is almost identical to our Player class constructor except it is called Ball. Lastly set vel = Vector2.Zero; The ball class should look like this when you are done:

```
namespace Pong
{
    public class Ball : Sprite
    {
        public Vector2 vel;

        public Ball(Texture2D texture) : base(texture)
        {
            vel = Vector2.Zero;
        }
    }
}
```

Using Our Classes

Now we need to edit our Game1.cs class to use the new classes. Delete the Texture2D, Vector2, int, and Rectangle classes for the player and ball. The beginning of our class should look like this now:

```
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

        Random random = new Random();

        SpriteFont pongFont;
        Vector2 player1TextPosition;
        Vector2 player2TextPosition;

        public Game1()
        {
            graphics = new GraphicsDeviceManager(this);
            Content.RootDirectory = "Content";
        }
```

Now add these lines after the spriteBatch line: Player player1, Player player2, Ball ball. The beginning of the class should look like this now:

```
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

        Player player1;
        Player player2;
        Ball ball;

        Random random = new Random();

        SpriteFont pongFont;
        Vector2 player1TextPosition;
        Vector2 player2TextPosition;
```

We have condensed 12 lines into three lines in this class. If you look in the rest of the Game1.cs class you will see lots of red squiggly lines, this is because Visual Studio does not know what the old variables are anymore because we deleted them. We will fix all those now.

Loading Content

To load the textures into the Player and Ball classes we will still use the LoadContent method but it will look slightly different. Here is what player1 will look like: player1 = new Player(Content.Load<Texture2D>(“paddle”)); The other two lines for the other player and the ball will look very similar. Here is what they the new LoadContent should look like now:

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

            player1 = new Player(Content.Load<Texture2D>("paddle"));
            player2 = new Player(Content.Load<Texture2D>("paddle"));
            ball = new Ball(Content.Load<Texture2D>("ball"));

            pongFont = Content.Load<SpriteFont>("PongFont");

            player1TextPosition = new Vector2(100, 25);
            player2TextPosition = new Vector2(graphics.GraphicsDevice.Viewport.Width -
100, 25);

            ResetObjects();
        }
```

Resetting Objects

We need to change the ResetObject method as well to use the new classes and variables we set up. We only need to change the variables underlined in red however. The first old variable is player1Position. This is where giving variables good names comes into play. We know player1Position is looking for the position of player1. To give it the new variable change player1Position to player1.pos. This will use the new variable player1 and will use pos variable from the Sprite class. The second old variable is player1Texture and this will be replaced with player1.tex. Try figuring out what the rest of the variables should be changed to in this method. Here is what it should look like when it is done:

```
        void ResetObjects()
        {
            player1.pos = new Vector2(50, graphics.GraphicsDevice.Viewport.Height / 2 -
player1.tex.Height / 2);
            player2.pos = new Vector2(graphics.GraphicsDevice.Viewport.Width -
player2.tex.Width - 50, graphics.GraphicsDevice.Viewport.Height / 2 - player2.tex.Height
/ 2);
            ball.pos = new Vector2(graphics.GraphicsDevice.Viewport.Width / 2 -
ball.tex.Width / 2, graphics.GraphicsDevice.Viewport.Height / 2 - ball.tex.Height / 2);

            ball.vel = new Vector2(random.Next(2, 5), random.Next(2, 5));
        }
```

Updating the Update Method

We now need to update the Update method but let's add a couple things to the Sprite class. We are going to add an UpdateRectangle method. After the constructor add public void UpdateRectangle(). Copy the rect line from the constructor and paste it in the UpdateRectangle method. We will add a Draw method to the class as well. After the UpdateRectangle method add public void Draw(SpriteBatch spriteBatch) and in the method add spriteBatch.Draw(tex, pos, Color.White). The two methods should look like this:

```
        public void UpdateRectangle()
        {
            rect = new Rectangle((int)pos.X, (int)pos.Y, tex.Width, tex.Height);
        }

        public void Draw(SpriteBatch spriteBatch)
        {
            spriteBatch.Draw(tex, pos, Color.White);
        }
```

Now we will change our Game1.cs update method to use the UpdateRectangle method and our new variables. Replace the player1, 2 and ball rectangle lines with player1.UpdateRectange(), player2.UpdateRectangle(), and ball.UpdateRectangle(). Do this at the beginning and end of the update method. Now we could go through the entire update method and change all the variables around or we could use another tool. Go to Edit > Find and Replace > Quick Replace or

you can press Control + H. To use this window type in what you want to find in the Find what box and put what you want to replace it with in the Replace with box. First un-minimize all the regions so we can see all our code. Then select the whole update method and change Look in to selection. Now type in ballPosition to Find what and ball.pos into Replace with then click the Replace All button. You should see a window pop up saying 9 things were changed. We need to now change the rest of the variables. Look for the next underlined variable and replace it with the appropriate variable. The updated update method should look like this:

```
        protected override void Update(GameTime gameTime)
        {
            // Allows the game to exit
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
                this.Exit();

            KeyboardState keyboardState = Keyboard.GetState();

            player1.UpdateRectangle();
            player2.UpdateRectangle();
            ball.UpdateRectangle();

            ball.pos += ball.vel;

            #region Paddle Movement
            if (keyboardState.IsKeyDown(Keys.A))
            {
                player1.pos.Y -= 2.5f;
            }

            if (keyboardState.IsKeyDown(Keys.Z))
            {
                player1.pos.Y += 2.5f;
            }

            if (keyboardState.IsKeyDown(Keys.Up))
            {
                player2.pos.Y -= 2.5f;
            }

            if (keyboardState.IsKeyDown(Keys.Down))
            {
                player2.pos.Y += 2.5f;
            }
            #endregion

            #region Paddle Boundries
            if (player1.pos.Y < 0)
            {
                player1.pos.Y = 0;
            }

            if (player1.pos.Y > graphics.GraphicsDevice.Viewport.Height -
player1.tex.Height)
            {
```

```
                player1.pos.Y = graphics.GraphicsDevice.Viewport.Height -
player1.tex.Height;
            }

            if (player2.pos.Y < 0)
            {
                player2.pos.Y = 0;
            }

            if (player2.pos.Y > graphics.GraphicsDevice.Viewport.Height -
player2.tex.Height)
            {
                player2.pos.Y = graphics.GraphicsDevice.Viewport.Height -
player2.tex.Height;
            }
            #endregion

            #region Ball Boundries and Scoring
            if (ball.pos.Y < 0)
            {
                ball.vel.Y *= -1;
                ball.pos.Y = 0;
            }

            if (ball.pos.Y > graphics.GraphicsDevice.Viewport.Height - ball.tex.Height)
            {
                ball.vel.Y *= -1;
                ball.pos.Y = graphics.GraphicsDevice.Viewport.Height - ball.tex.Height;
            }

            if (ball.pos.X < 0)
            {
                ResetObjects();
                player2.score++;
            }

            if (ball.pos.X > graphics.GraphicsDevice.Viewport.Width - ball.tex.Width)
            {
                ResetObjects();
                player1.score++;
            }
            #endregion

            if (ball.rect.Intersects(player1.rect))
            {
                ball.vel.X *= -1;
                ball.pos.X = player1.pos.X + player1.tex.Width;
            }

            if (ball.rect.Intersects(player2.rect))
            {
                ball.vel.X *= -1;
                ball.pos.X = player2.pos.X - ball.tex.Width;
            }

            player1.UpdateRectangle();
            player2.UpdateRectangle();
            ball.UpdateRectangle();
```

```
            base.Update(gameTime);
        }
```

Finishing Up

The last thing we need to do is edit the Draw method to use our new draw function. Delete the three draw lines and replace them with ball.Draw(spriteBatch), player1.Draw(spriteBatch), and player2.Draw(spriteBatch). We also need to change player1score and player2score to player1.score and player2.score in the DrawString lines. The new draw method should look like this:

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

            spriteBatch.Begin();
            ball.Draw(spriteBatch);
            player1.Draw(spriteBatch);
            player2.Draw(spriteBatch);
            spriteBatch.DrawString(pongFont, player1.score.ToString(),
player1TextPosition, Color.White);
            spriteBatch.DrawString(pongFont, player2.score.ToString(),
player2TextPosition, Color.White);
            spriteBatch.End();

            base.Draw(gameTime);
        }
```

If you run the game now everything should look the same but we have cleaned up our code significantly. The last thing you should do is comment the code to explain it.

Chapter 7: Libraries and Navigation

If we wanted to reuse the Sprite class code in another solution we would have to rewrite it in that project or copy the file over. If we wrote more classes we wanted to use in other games it would become a large hassle. Instead what we can do is create a library. A library holds all our code we want to reuse. To use a library in a project we import it and add a reference and a using to the other project. The other thing we will do in this chapter will be to create menus for our games.

Creating a Library

To create a new library go to File > New > Project then select XNA Game Studio and instead of Windows Game click Windows Game Library and name it XNALibrary. Now our project will open up but it will look different than other XNA projects we have made. It has all the XNA

using lines we are used to but there is no pre-generated XNA game code. This is because this game library will be an addition to your game projects and will just hold class files you want to reuse in other games.

Adding the Sprite Class

We want to use the Sprite class in other games so first we have to import it. Right click on XNALibrary and go to Add > Existing Item. Then browse to where you saved the Pong project and go add the Sprite class. The folder path should be something like \Pong\Pong\Pong. After we added the class it should be ready to use except for the fact that the namespace is wrong. If you look at pre-generated class file named Class1 you should see the namespace for the class is XNALibrary. We need to change the namespace for the Sprite class to XNALibrary. We also want to add a couple more things to this class. We want a vel variable of type Vector2 which will hold the velocity of the sprite and a color variable of type Color which will hold the color of the sprite. Set up these variables before the constructor and set them to these variables in the constructor. vel = Vector2.Zero and color = Color.White. We also need to change the Sprite class draw method to use the new color variable. Here is what the new Sprite class should look like:

```
namespace XNALibrary
{
    public class Sprite
    {
        public Texture2D tex;
        public Vector2 pos;
        public Rectangle rect;
        public Vector2 vel;
        public Color color;

        public Sprite(Texture2D loadedTex)
        {
            tex = loadedTex;
            pos = Vector2.Zero;
            rect = new Rectangle((int)pos.X, (int)pos.Y, tex.Width, tex.Height);
            vel = Vector2.Zero;
            color = Color.White;
        }

        public void UpdateRectangle()
        {
            rect = new Rectangle((int)pos.X, (int)pos.Y, tex.Width, tex.Height);
        }

        public void Draw(SpriteBatch spriteBatch)
        {
            spriteBatch.Draw(tex, pos, color);
        }
    }
}
```

TextObject Class

We are going to create a new class before we start working with navigation. This class will be called TextObject and will allow for easier manipulation of sprite fonts we use. First we must rename our Class1 file. Right click it in the Solution Explorer and click Rename. Change the name to TextObject.cs and click yes when the message box comes up. Next we want to inherit the Sprite class variables and methods for this class but there is a problem. Our Sprite class is looking for a Texture2D when a new Sprite object is made, however SpriteFonts are not textures but are SpriteFonts so we need to create a new constructor in the Sprite class that handles SpriteFonts. In the sprite class add this right after our original constructor. public Sprite(SpriteFont loadedFont). We will not put anything in this constructor because we will do that in our TextObject class. This is what the new constructor should look like:

```
        public Sprite(SpriteFont loadedFont)
        {
        }
```

Now in our TextObject class we will add two new variables. public SpriteFont font and public string text. These two variables will hold the actual SpriteFont we are using and the text that our text object will display. Next we need the constructor for the class. public TextObject(SpriteFont loadedFont, string spriteText) : base(loadedFont). Inside this constructor we want to set these variables. font = loadedFont, text = spriteText, pos = Vector2.Zero, vel = Vector2.Zero, color = Color.White, and rect = new Rectangle((int)pos.X, (int)pos.Y, (int)MeasureString().X, (int)MeasureString().Y). The class should look like this right now:

```
namespace XNALibrary
{
    public class TextObject : Sprite
    {
        public SpriteFont font;
        public string text;

        public TextObject(SpriteFont loadedFont, string spriteFont) : base(loadedFont)
        {
            font = loadedFont;
            text = spriteFont;
            pos = Vector2.Zero;
            vel = Vector2.Zero;
            color = Color.White;
            rect = new Rectangle((int)pos.X, (int)pos.Y, (int)MeasureString().X,
(int)MeasureString().Y);
        }
    }
}
```

Don’t worry about the MeasureString being underlined. We are going to write that method right now.

Measuring a String

To find the width and height of a string we will use a method contained in the SpriteFont class. After the constructor add public Vector2 MeasureString() and in that function add return font.MeasureString(text). It should look like this:

```
public Vector2 MeasureString()
{
    return font.MeasureString(text);
}
```

This is a different method than the ones we have written before. Instead of being void it now is a Vector2. This means when the method is called it will run the code inside it and will then return a Vector2. Inside the method we must have a return statement and we are returning the measurement of our text.

Drawing a String

We also need to be able to draw our string. We will write a draw method that will be very similar to our Draw method in our Sprite class but instead it will draw a string so name it DrawString. It should look like this when it's done:

```
public void DrawString(SpriteBatch spriteBatch)
{
    spriteBatch.DrawString(font, text, pos, color);
}
```

Updating the String

We also need a method to update the string's bounding rectangle. It will be almost the same as our Sprite class one except it will use the rectangle line from our TextObject constructor. Name it UpdateRectangle. It should look like this when it is done:

```
public void UpdateRectangle()
{
    rect = new Rectangle((int)pos.X, (int)pos.Y, (int)MeasureString().X,
(int)MeasureString().Y);
}
```

Overloading

Now we have almost everything for our TextObject class. For our constructor though what if we wanted to set the position of our text when we set up the object as well. We would need to ask for the desired position as well. To do this while keeping our old position we will need to overload our constructor. We can overload methods as well. To overload something we just write a copy of our method or constructor but the new copy has different parameters. For this

new constructor we will add a position parameter. Copy the constructor and paste the new version below our old constructor then add a Vector2 parameter called position then set pos = position inside the new constructor. It should look like this:

```
        public TextObject(SpriteFont loadedFont, string spriteFont, Vector2 position)
            : base(loadedFont)
        {
            font = loadedFont;
            text = spriteFont;
            pos = position;
            vel = Vector2.Zero;
            color = Color.White;
            rect = new Rectangle((int)pos.X, (int)pos.Y, (int)MeasureString().X,
(int)MeasureString().Y);
        }
```

Menus

Now save the library and make a new Windows Game project called Navigation. Now to use our library in this project we must first add it to our references. In Solution Explorer right click the Solution 'Navigation' bar and go to Add > Existing Project. Then browse to where the XNALibrary is saved and add the CSPROJ file. Then right click references in our actual Navigation project and go to Add Reference. Then click the Projects tab, make sure XNALibrary is selected and click ok. The last thing we need to do is add a using in the Game1 class. Right after the last using type using XNALibrary. Now we want to start on menus. Let's have only two menu selections to start out with. First we need text to select on the menu so let's define two TextObjects called menuObject1 and menuObject2. Notice how we can use the TextObject class even though there are no other classes in our Navigation project. This is the power of libraries. We also need an int to store the selection number on the menu so define an int called menuSelection and set it equal to 0. We also need a previousKeyboardState so we can select menu objects accurately so add that as well. Our variable list should look like this:

```
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

        TextObject menuObject1;
        TextObject menuObject2;
        int menuSelection = 0;
        KeyboardState previousKeyboardState = Keyboard.GetState();

        public Game1()
        {
            graphics = new GraphicsDeviceManager(this);
            Content.RootDirectory = "Content";
        }
```

Now we need to set up those TextObjects but we also need a SpriteFont. Add a SpriteFont file and name it TestFont and keep all the settings default. Now in LoadContent load the two TextObjects we set up. When you are loading it you will see that you can press the up or down arrow when a tooltip pops up and you can see the different versions of the constructor. For menuObject1 the text should be Play and for menuObject2 the text should be Quit. This is what it should look like:

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

    menuObject1 = new TextObject(Content.Load<SpriteFont>("TestFont"), "Play");
    menuObject2 = new TextObject(Content.Load<SpriteFont>("TestFont"), "Quit");
}
```

Notice we did not use the second version of our constructor. This is because we cannot set up the position of the text correctly without having the font loaded so we can use MeasureString. Since we have loaded the font though we can set our text positions now, in LoadContent after the TextObject lines add these two lines:

```
            menuObject1.pos = new Vector2(graphics.GraphicsDevice.Viewport.Width / 2 -
menuObject1.MeasureString().X / 2, graphics.GraphicsDevice.Viewport.Height / 2 -
menuObject1.MeasureString().Y / 2);
            menuObject2.pos = new Vector2(graphics.GraphicsDevice.Viewport.Width / 2 -
menuObject2.MeasureString().X / 2, menuObject1.pos.Y + menuObject1.MeasureString().Y +
10.0f);
```

There is a lot of code here but if you look carefully you will see that it is just the centering code we have used before. We divide the screen and object both by two to center it. If you look at the Y coordinate for the second menu object however you will see it looks different. What we are doing here is using the first menu object to position the second menu object. We get the position, add the height of the object, and also add 10 pixels for more spacing.

Navigating

Now we want these menu objects to be selectable. We will use the menuSelection integer we set up earlier to do that. First in the update method add a KeyboardState and add this line at the end of the update method: previousKeyboardState = keyboardState so we can select menu objects. Now we want to be able to increase and decrease the menuSelection number with our up and down arrow keys. Do not forget to add a check for a key up from previousKeyboardState. Also when the up key is pressed the menu selection subtracts a number and when the down key is presses it adds a number. By now you should be able to figure this out but here it is:

```
            if (keyboardState.IsKeyDown(Keys.Up) &&
previousKeyboardState.IsKeyUp(Keys.Up))
            {
                menuSelection--;
            }

            if (keyboardState.IsKeyDown(Keys.Down) &&
previousKeyboardState.IsKeyUp(Keys.Down))
            {
                menuSelection++;
            }
```

We only have two menu objects though. We will assign 0 with the Play object and 1 with the Quit object. If it is on the Play object and up is pressed it will go to -1 and nothing will be selected. We want an if statement to catch that. This will also be true for when the selection number is 2. To fix this we will use a nested if statement. This just means we use an if statement inside an if statement. After the menuSelection—line add an if condition of the menuSelection == -1 and a code run of menuSelection = 1. We will do the same thing for the down key except we check if menuSelection equals 2 and set it equal to 0. This is what it should look like:

```
            if (keyboardState.IsKeyDown(Keys.Up) &&
previousKeyboardState.IsKeyUp(Keys.Up))
            {
                menuSelection--;

                if (menuSelection == -1)
                {
                    menuSelection = 1;
                }
            }

            if (keyboardState.IsKeyDown(Keys.Down) &&
previousKeyboardState.IsKeyUp(Keys.Down))
            {
                menuSelection++;

                if (menuSelection == 2)
                {
                    menuSelection = 0;
                }
            }
```

We also need a way to know which TextObject is selected. We can change the color of an object if it is selected. This is an simple if statement. If menuSelection == 0 set menuObject1 = Color.Yellow. Also make sure to set the other menu object to Color.White. Do the same thing for menuSelection == 1 and set menuObject2 = Color.Yellow and set the other menu object to white. It should look like this:

```
if (menuSelection == 0)
{
    menuObject1.color = Color.Yellow;
    menuObject2.color = Color.White;
}

if (menuSelection == 1)
{
    menuObject2.color = Color.Yellow;
    menuObject1.color = Color.White;
}
```

The last thing we can do in the update method is to make one of the menu selections actually work. If the Quit selection is selected and Enter is pressed then quit the game. Remember the Quit selection is menuSelection = 1 and the way to quit a game is this.Quit(); This is what the if statement should look like:

```
if (menuSelection == 1 && keyboardState.IsKeyDown(Keys.Enter))
{
    this.Exit();
}
```

Drawing Everything

We are almost done. The last thing we need to do is draw everything. This would have been more of a hassle before but now this is easy with our library and classes. In the draw method all we need is the spriteBatch.Begin, the menuObject1 and 2 .DrawString methods with spriteBatch given to them and the spriteBatch.End line. This is what the draw method looks like:

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

    spriteBatch.Begin();
    menuObject1.DrawString(spriteBatch);
    menuObject2.DrawString(spriteBatch);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

Now if you run the game you can scroll though the two menu objects and if the Quit selection is chosen you can press enter to close the window. That's it! Menus are really this easy and this method works with menus even larger than 2 choices.

Chapter 8: Wrapping Up

You have come a pretty far way but you did it. You learned how to program, how to program in C#, and how to program in C# with XNA. You made your first game while learning programming concepts along the way. It doesn't have to end here though. Everything you learned here can be expanded. You can play around with things, you can make things better. There are numerous resources available online. Programming teaches you how to solve problems in different ways. For example everything you learned in this book can be coded in a completely different way. It is all up to how the coder which is you now. The square is on your side of the screen now. Here are some resources you can use to continue on your coding journey. Good luck, code well, and always remember, have fun!

Creators Club Website: http://create.msdn.com – This site is the main site for XNA. It has tons of examples and code samples you can look at and use.

MSDN Library: http://msdn.microsoft.com/library - This site is the main site where you can look at all the documentation for XNA and C#. Want to know what enum means? You can look it up there. Trying to figure out how to use the other overloads of SpriteBatch? You can look it up.