PHStudios.com Tutorials

XNA Series: Paddles

Text Edition

Contents

Section I

Section Overview

This section will introduce you to XNA Game Studio, what it can do, what you need, and what this tutorial plans to teach you. For those of you who know what XNA is and have all the requirements, you can skip to *Part III.*

Part I Introduction to XNA

XNA, created by Microsoft®, is a video game tool and framework to make the game creator's job much easier than before. It is recommended for new game programmers or games that do not require as much power as possible to use XNA. There are many great 3D games made by XNA with no performance loss. Microsoft® XNA can make video game design both fun, and simple.

Version History

Microsoft® XNA was first released in December 2006, followed by a version 1 refresh in April 2007. We are currently on XNA 2.0. This version is compatible with any Visual Studio 2005, not just C# 2005 express. Version 3.0 is scheduled for holidays 2008.

Licenses and Distribution

XNA is currently a Microsoft® Windows and Xbox360 only development tool. Plans for XNA 3.0 include Zune support. Version 2.0 included networking capabilities, and if you use these for commercial games, you will need to contact Microsoft and get a signed agreement. You can, however, create your own networking code and release it without trouble. You can release commercial games on Windows, but not for the Xbox360. The EULA for XNA can be found at http://creators.xna.com

Part II Requirements

Microsoft® XNA might not require a whole lot depending on what you want to do. There have been amazing 3D games developed that require a decent video card, and many 2D games that do not require such a powerful card (Shader Models). Any decent video card will work great for 2D games, and most 3D games. Here is a list of what Microsoft® recommends:

- Operating System: XP (with Service Pack 2 or later) or Vista™
- Video Card: Any with Shader Model 1.1 or higher (2.0 is highly recommended) and DirectX 9.0c or higher.
- Visual Studio 2005 or Visual C# 2005 Express
- .NET Framework 2.0
- XNA Creators Club Membership, and LIVE Membership (Xbox 360 development, and using LIVE in Windows games)
- Hard Drive on Xbox 360 (Xbox 360 development)
- DirectX Runtime (included in XNA installer)

Part III Accomplishments

This game might lack graphics and game play, but it is highly recommended as a very first for new developers. There are elements in this game that will stick with you for every game you will make. It is also more object oriented than most simple games are, and this is to show you right away a good way to do game development. This game and tutorial will teach you a lot of useful information that you can expand onto later. You will be introduced to a screen system that will greatly benefit in larger scale games. With this screen system, you can set up various parts of the game to appear at various times. You can also add actions to these screens (a menu screen for example). This game will also include an input system, but will focus on keyboard only for this tutorial. You can then build off this input system for a highly detailed 2D or even a 3D game. Despite the high object oriented approach of this game, it will still be very simple.

Section II

Section Overview

This section will show you the most important part of game development, the design process. You should never jump into the coding without major thought and design behind the game. You need to sit down and make a list of what you want included in the game, graphics, how long you want it to be, how complex, and many more aspects need to be covered before you even begin coding.

Part I How the Game Works

The goal of this game is mainly a teaching lesson, than a complex game. This is indeed a pong clone, which has been done too many times to count, but it should be the first game any new developer should create. The bottom line is, do not jump to an amazing game that will have hours of game play. I made this mistake myself, and have never finished the first game project I started many years ago. Start small, and expand from there. This game does just that, so let's start by designing it!

First Things First: Story

A pong game with a story? No, but this should always be the first step in a game. A story should morph into a game, not the other way around since the story can change the style of the game. You may want an RPG (role playing game) but after writing the story, you decide a FPS (first person shooter) is the best way to go. For games like this, space shooters, and others, you do not need a story so you can skip this part of the design process.

Second, Gameplay Ideas

Now this is not a complete game play thought, this should be a very basic idea of gameplay. This is more of a thought of the games unique features than an in-depth look on gameplay. For example, if you were creating a detailed game, you should answer these questions after you work a decent story:

- What camera view? Will it be top-down or side-scroller 2D? Full 3D?
- What should be visible?
- Will there be buildings, items, and/or weapons?
- How about exploration?

This game is too basic to break the gameplay design into multiple parts, so in the first major game tutorial (Space RPG), we will discuss this more. This section of the gameplay design will let you know what kind of art you need for the game, so you can either assign somebody to do the artwork, get somebody to draw some basic sketches, or just keep in your head until you get around to it.

Third, Finalize Gameplay Ideas

You should finish by covering every aspect of the game. You have a basic understanding of the story and can morph your game to fit that. We will discuss this more in the later games, but for now you already have the gameplay finished for this game, since it is very very simple and has no story.

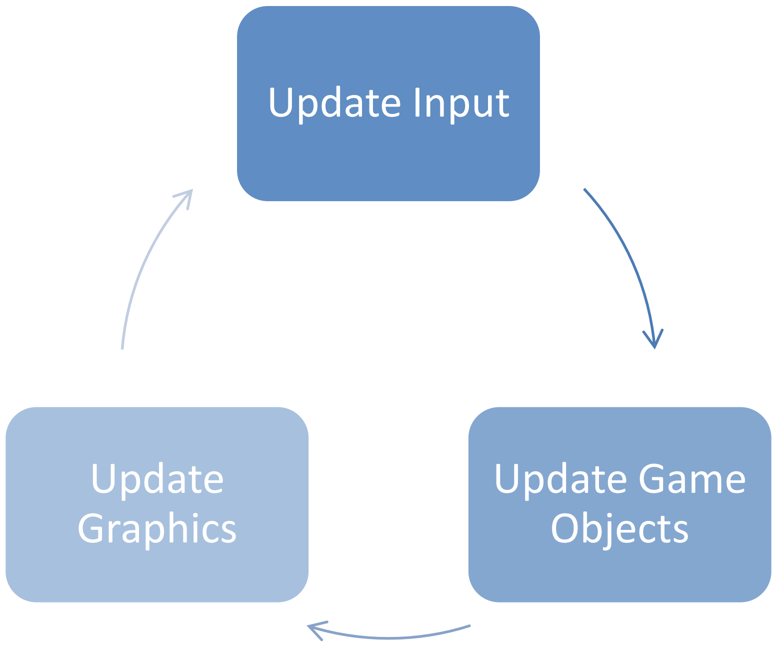
- 2 paddles, 1 will be user controlled and the other will be computer controlled.
- Poor AI (first game, no need for amazing AI yet)
- Ball starts in the middle of the screen, picks a random angle, and moves right away.
- Ball increases speed after every hit until it reaches its max speed.
- Rectangular collision detection (VERY poor, but easiest collision for beginners)
- Computer paddle will not move while the ball is headed in the player's direction.
- If the ball passes a paddle, the paddle's controller gets a point.
- When the computer reaches a certain score, the player loses. When the player reaches that same score, he/she wins.

For simple games like this, the above style of gameplay ideas is fine. However, when you get into larger projects, you need a design document which we will get into on the space RPG.

Finally, the Artwork

I have left out some stuff like audio, but this game will have no audio. We will add the other aspects of the design process as we progress to larger and more complex games. For now, we should keep things short and sweet. Artwork for larger games will take a very long time to finish, so it would be best to find a team after a few basic games if you wish to continue. For this game, the only artwork we need to create is a single white pixel. The single pixel will be used to create the paddles.

Game Loop:



- Update Input
  - Every game loop we will check the keyboard to see which buttons are pressed. We will add gamepad support in a later tutorial.
- Update Game Objects
  - We will need to do quite a bit of stuff in the area. We need to update our objects based on input or time if it is computer controlled. We should apply physics if available as well. There will be no physics in this game. After you update the object's position, you should check collision.
- Update Graphics
  - The last thing you should do is update the graphics based on the object's properties. After the graphics are updated, you need to draw them.

Part III Game Art

As stated on page 6, the artwork for this game will be a simple single white pixel. This is all you should need for a pong clone. As always you can change the graphics to fit your needs. If you wish to make better looking graphics, you are welcome to. I will show you how to morph the single pixel into a ball and paddles in the coding. You can use Microsoft® Paint to create the art or anything else you like to use.

Section III

Section Overview

Now that we have a good understanding of our game and the gameplay we want, we can begin coding. Since this might be the first time seeing the basic XNA game engine for most of you, we need to look at that first. We will then modify the basic engine to fit our games needs.

Part I Introduction

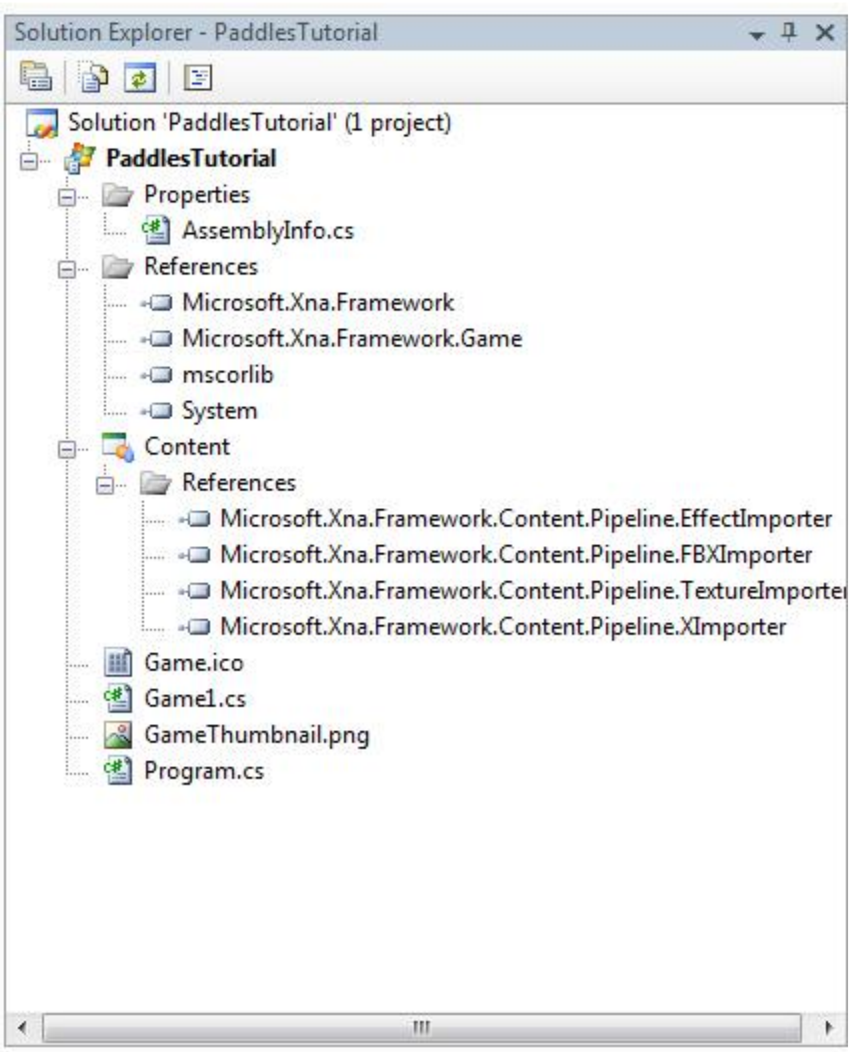
When you start a new game project (Windows or Xbox 360), a basic game engine will be created for you. You can build on top of this engine to create one that fits your game. For this game, we will be taking a more advanced approach than what is needed for it. You can create the same game with a large number of variables, but a more object-oriented way will give you more control. Let's take a look at the basic game engine. Start Visual Studio 2005, or Visual C# 2005 Express and create a new XNA 2.0 Windows Game called PaddlesTutorial.

Code Automatically Generated for New Projects.

```
using System;
using System.Collections.Generic;
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.Net;
using Microsoft.Xna.Framework.Storage;

namespace PaddlesTutorial
{
    /// <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";
        }

        /// <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();
        }

        /// <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
        }

        /// <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
        }

        /// <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);
        }

        /// <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)
        {
            graphics.GraphicsDevice.Clear(Color.CornflowerBlue);

            // TODO: Add your drawing code here

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

This code is very well commented. Basically you put initialization code inside the initialize method, graphics and content loading in the load content method, update code in the update method, and drawing code in the draw method. Now let's take a look at the solution explorer.



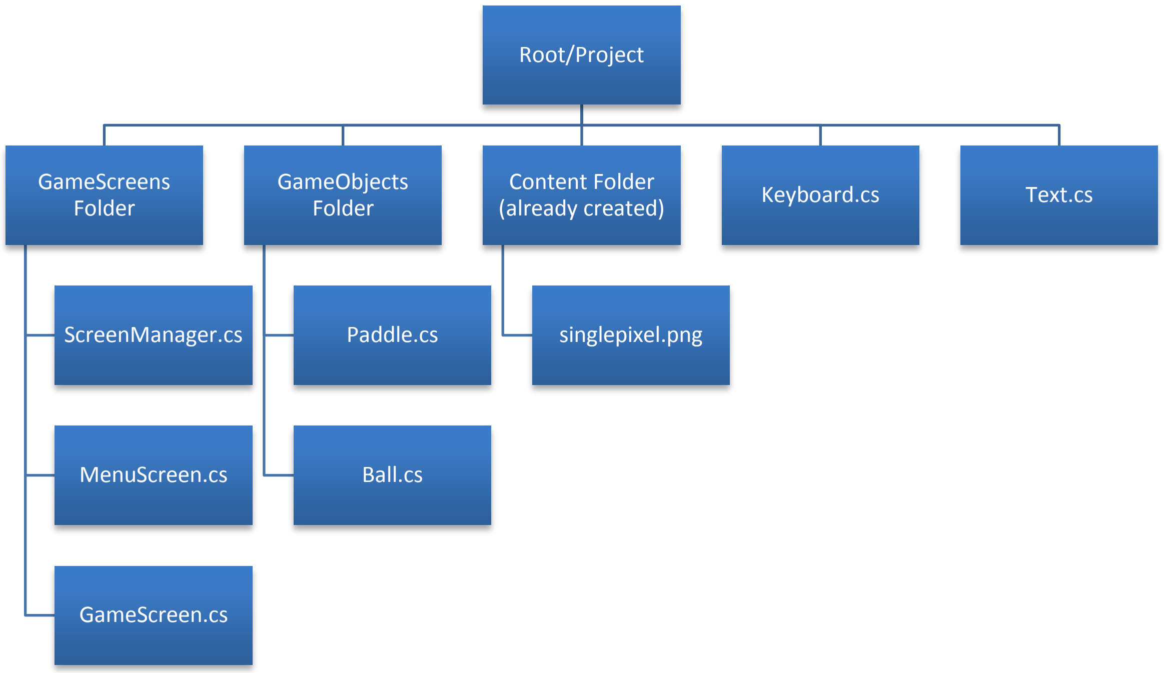
```
Solution 'PaddlesTutorial' (1 project)
  PaddlesTutorial
    Properties
      AssemblyInfo.cs
    References
      Microsoft.Xna.Framework
      Microsoft.Xna.Framework.Game
      mscorlib
      System
    Content
      References
        Microsoft.Xna.Framework.Content.Pipeline.EffectImporter
        Microsoft.Xna.Framework.Content.Pipeline.FBXImporter
        Microsoft.Xna.Framework.Content.Pipeline.TextureImporte
        Microsoft.Xna.Framework.Content.Pipeline.XImporter
    Game.ico
    Game1.cs
    GameThumbnail.png
    Program.cs
```

File/Folder	Description
AssemblyInfo.cs	**This file holds all the information for the exe. The title, product, description, company, copyright, trademark, and file version.**
Content	**This area should hold all your content. Music, sounds, models, sprites, text, and anything else should go here.**
Game.ico	**This holds the icon for the game that will appear on the desktop and the upper left of the window.**
Game1.cs	**This holds the very basic game engine for your game. All it does right now is display a blue screen.**
GameThumbnail.png	**This image will appear in the creators club package (.ccgame)**
Program.cs	**This is the entry point for our game. The main method is located here.**

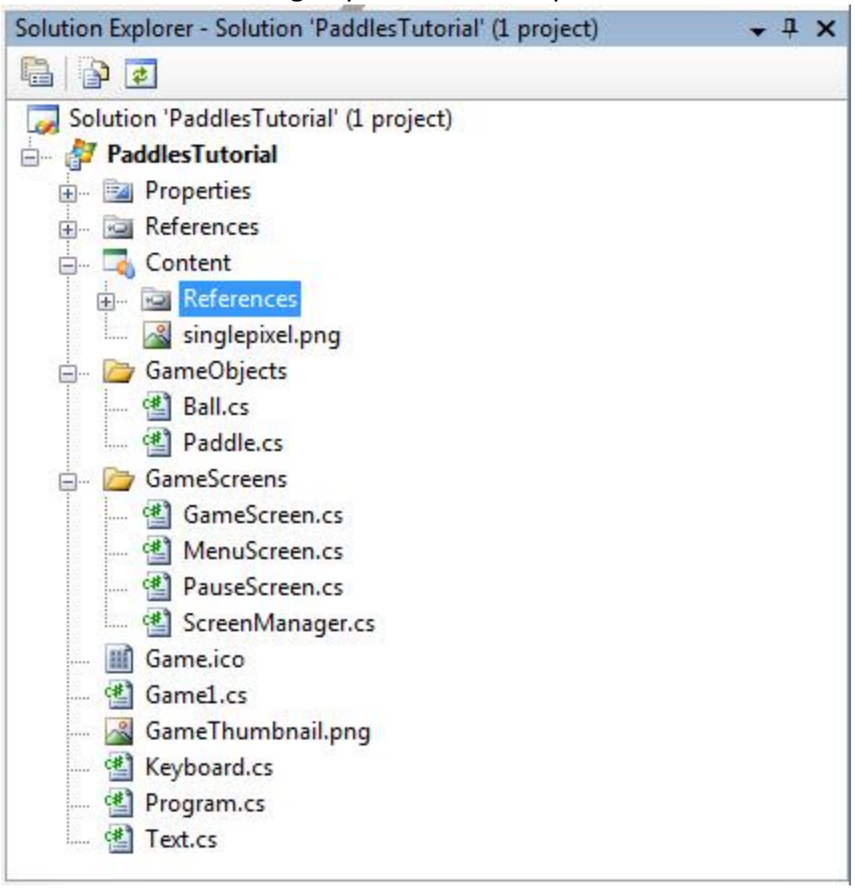
We have everything we need to get started!

Part II Modifications

We need to make some modifications before we start coding. Let's create the files and folders needed for our project. The only folders we need to create are the screens folder called GameScreens, and objects folder called GameObjects. Use the following chart to create your files.



You should have the following in your solution explorer.



Inside Game1.cs, create a public static ContentManager object (I named it content). Inside the Game1.cs constructor, rename Content to content in line `Content.RootDirectory = "Content";`. Above that line initialize the lowercase content by adding this `content = new ContentManager(Services);` Now we can start building our game!

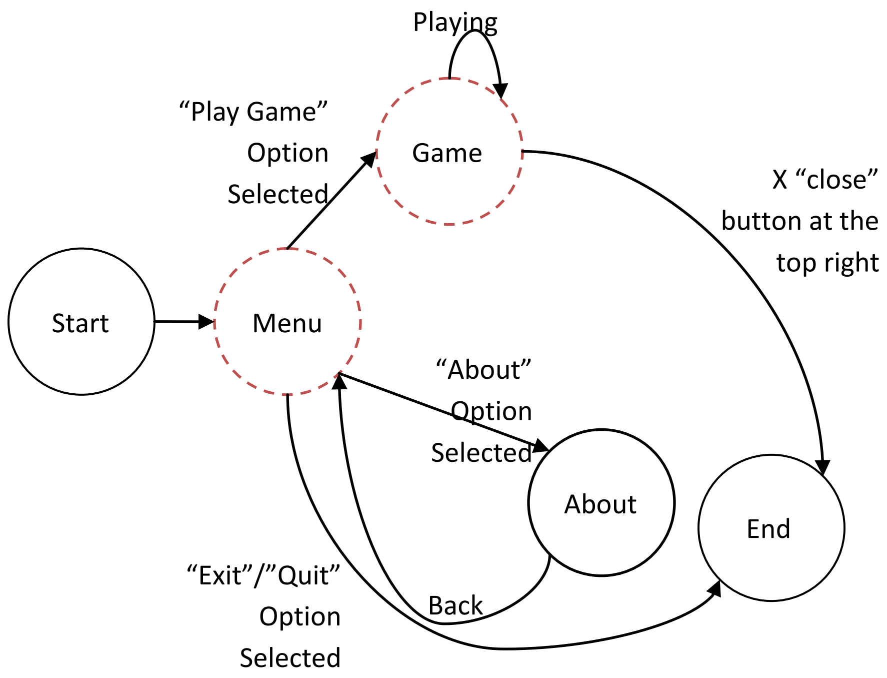
Section IV

Section Overview

This will be the first thing to do for our game. We do not want to start in the gameplay first, and then have to deal with fixing everything to work to add the menu and game screens. The best thing to do is work in progression. The menu will be the first thing the players will see, so we need to build the screen manager and menu screen first.

Part I Why Screens?

Before the coding, we need to discuss why screens are important. For this game, screens are not really necessary. For more complex games, you really should split the game into as many screens as possible. My way of thinking is, we should not get comfortable with cramming all our variables and logic into the Game1 class. I have seen many basic game source codes, and most of them put everything in the Game1 class. I feel it should be important to cover how to split up the coding. This type of approach might sound confusing, but this tutorial will guide you through the process. Screens are exactly like states, in that we can split the game into multiple states to make the code more organized. The following graph is known as a state diagram, and we will be following it for the states. The red-dotted circles are the screens.


Playing
"Play Game" Option Selected
Game
X "close" button at the top right
Start
Menu
"About" Option Selected
About
End
"Exit"/"Quit" Option Selected
Back

The arrows pointing from one state to another indicate the condition needed to change states. You will notice in the game state that there is an arrow pointing to itself, this is where the game loop will take place. All states will be part of the main loop. Now we need to create the screen manager to handle the three red-dotted circles.

The screen manager will handle all screens we create. We will take a very basic approach to the manager for now. Later in the series we will create a very complex manager to make it more functional. Screen Manager will be a derivative of DrawableGameComponent (class ScreenManager:DrawableGameComponent). Here is a list of what the screen manager should contain.

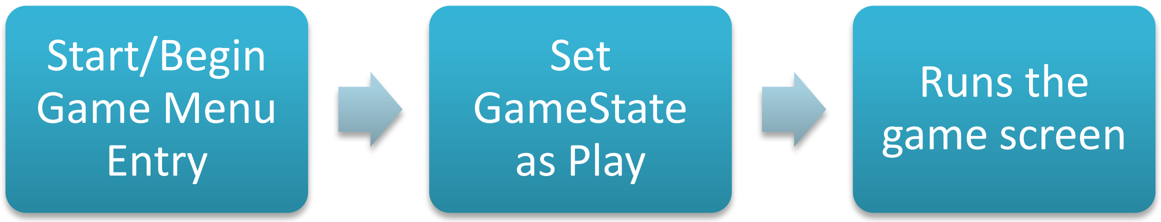
- Enumeration – GameState
  - Menu and Play values.
- Attributes
  - GameState object, set to public static, and initialized to GameState.Menu
  - Keyboard object which should also be set to public static.
  - SpriteFont object which should also be set to public static.
  - bool variable called isExiting which is public static, and set to false.
  - MenuScreen object
  - GameScreen object
  - SpriteBatch object
- Constructor, which accepts Game object as a parameter.
  - Initialize the two screen objects and the keyboard objects.
- Override the LoadContent method and initialize the SpriteBatch object.
- Update method, which is public override void, to update the keyboard and perform individual screen updates based on the value of the GameState object.
- Draw method, which is public override void, to do individual draw calls based on the value of the GameState object.

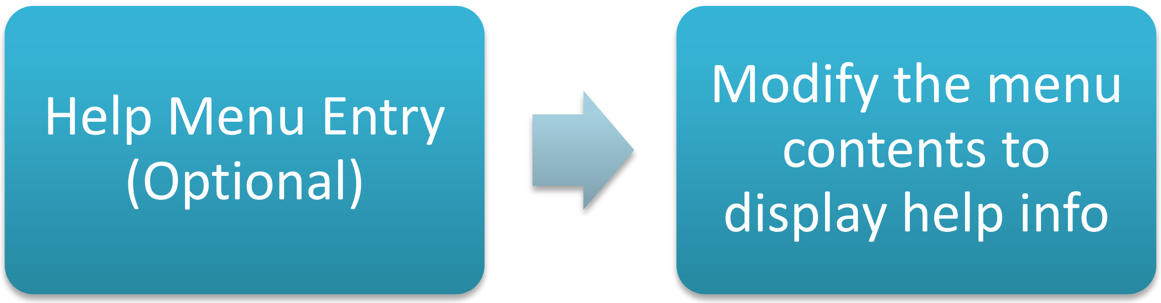
ScreenManager.cs

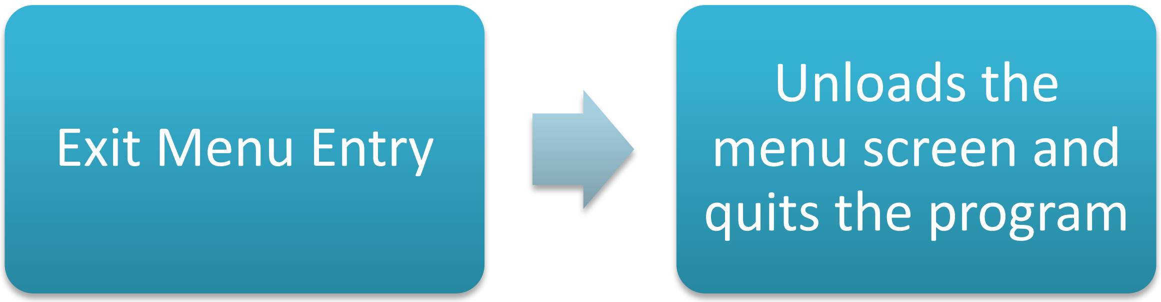
```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;

namespace PaddlesTutorial.GameScreens
{
    public enum GameState
    {
        Menu,
        Play
    }
    class ScreenManager:DrawableGameComponent
    {
        public static GameState gameState = GameState.Menu;
        public static Keyboard keyboard;
        public static SpriteFont spriteFont;
        public static bool isExiting = false;
        MenuScreen screen_Menu;
        GameScreen screen_Game;
        SpriteBatch spriteBatch;
        public ScreenManager(Game game)
            : base(game)
        {
            screen_Game = new GameScreen();
            screen_Menu = new MenuScreen();
            keyboard = new Keyboard();
        }
        protected override void LoadContent()
        {
            spriteBatch = new SpriteBatch(GraphicsDevice);
        }
        public override void Update(GameTime gameTime)
        {
            //TODO:  Update screens and keyboard

            base.Update(gameTime);
        }
        public override void Draw(GameTime gameTime)
        {
            //TODO:  Draw screens

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

This is a very basic way of doing screens. We will modify this later in the tutorial and enhance this later on in the series.

The menu on this game will be very simple. We will have menu entries that will have actions linked with them. Highlighted entries will be a different color than the others, but the same size and style font. Later in the series we will look at a more advanced menu system. Here are some graphics on what menu entry does what.




Help Menu Entry (Optional)
Modify the menu contents to display help info



Let's take this one step at a time. We will first add text to our menu. In order to add text we need to add a spritefont file to our project.

Adding SpriteFont

To add a Sprite Font file, go to the Solution Explorer and right-click the Content sub-project, go to Add -> New Item and select Sprite Font with a name of your choosing. I named it *basic.spritefont*. A Sprite Font file is an XML document with information on the text. The following is from the initially created file:

basic.spritefont **before edit**

```
<?xml version="1.0" encoding="utf-8"?>
<!--
This file contains an xml description of a font, and will be read by
the XNA
Framework Content Pipeline. Follow the comments to customize the
appearance
of the font in your game, and to change the characters which are
available to draw
with.
-->
<XnaContent
xmlns:Graphics="Microsoft.Xna.Framework.Content.Pipeline.Graphics">
  <Asset Type="Graphics:FontDescription">

    <!--
    Modify this string to change the font that will be imported.
Redistributable sample
    fonts are available at
http://go.microsoft.com/fwlink/?LinkId=104778&clcid=0x409.
```

```
    -->
    <FontName>basic</FontName>

    <!--
    Size is a float value, measured in points. Modify this value to
change
    the size of the font.
    -->
    <Size>14</Size>

    <!--
    Spacing is a float value, measured in pixels. Modify this value to
change
    the amount of spacing in between characters.
    -->
    <Spacing>0</Spacing>

    <!--
    UseKerning controls the layout of the font. If this value is true,
kerning information
    will be used when placing characters.
    -->
    <UseKerning>true</UseKerning>

    <!--
    Style controls the style of the font. Valid entries are "Regular",
"Bold", "Italic",
    and "Bold, Italic", and are case sensitive.
    -->
    <Style>Regular</Style>

    <!--
    CharacterRegions control what letters are available in the font.
Every
    character from Start to End will be built and made available for
drawing. The
    default range is from 32, (ASCII space), to 126, ('~'), covering
the basic Latin
    character set. The characters are ordered according to the Unicode
standard.
    See the documentation for more information.
    -->
    <CharacterRegions>
      <CharacterRegion>
        <Start>&#32;</Start>
        <End>&#126;</End>
      </CharacterRegion>
    </CharacterRegions>
  </Asset>
</XnaContent>
```

With all the comments, I will not go into detail on what every attribute means. Just replace the font you want to use in the FontName attribute (Times New Roman for example). I used Trebuchet MS which is included in Windows 2000 or greater. Change the font size with the Size attribute, and the spacing of the letters in the Spacing attribute

(I changed the spacing to 2). Here is my final Sprite Font.

***basic.spritefont* after edit**

```
<?xml version="1.0" encoding="utf-8"?>
<!--
This file contains an xml description of a font, and will be read by
the XNA
Framework Content Pipeline. Follow the comments to customize the
appearance
of the font in your game, and to change the characters which are
available to draw
with.
-->
<XnaContent
xmlns:Graphics="Microsoft.Xna.Framework.Content.Pipeline.Graphics">
  <Asset Type="Graphics:FontDescription">

    <!--
    Modify this string to change the font that will be imported.
Redistributable sample
    fonts are available at
http://go.microsoft.com/fwlink/?LinkId=104778&clcid=0x409.
    -->
    <FontName>Trebuchet MS</FontName>

    <!--
    Size is a float value, measured in points. Modify this value to
change
    the size of the font.
    -->
    <Size>14</Size>

    <!--
    Spacing is a float value, measured in pixels. Modify this value to
change
    the amount of spacing in between characters.
    -->
    <Spacing>2</Spacing>

    <!--
    UseKerning controls the layout of the font. If this value is true,
kerning information
    will be used when placing characters.
    -->
    <UseKerning>true</UseKerning>

    <!--
    Style controls the style of the font. Valid entries are "Regular",
"Bold", "Italic",
    and "Bold, Italic", and are case sensitive.
    -->
    <Style>Regular</Style>

    <!--
    CharacterRegions control what letters are available in the font.
```

```
Every
    character from Start to End will be built and made available for
drawing. The
    default range is from 32, (ASCII space), to 126, ('~'), covering
the basic Latin
    character set. The characters are ordered according to the Unicode
standard.
    See the documentation for more information.
    -->
    <CharacterRegions>
      <CharacterRegion>
        <Start>&#32;</Start>
        <End>&#126;</End>
      </CharacterRegion>
    </CharacterRegions>
  </Asset>
</XnaContent>
```

Modifying the Menu Screen

We will not be completely finished with this screen until part 4, since we will be making a text class to make it easier to handle text. Here is what we need to have in our class.

- Enumerated variable called MenuState that will have values of Main and Help, with access as Public.
- MenuState object called mState and is initialized to MenuState.Main
- int variable called selection to hold the current selected menu entry.
- Update method with a GameTime parameter and 2 regions, Main and Help. Access is public and return type is void.
- Draw method, no parameter, and 2 regions Main and Help. Access is public and return type is void.

MenuScreen.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Xna.Framework;

namespace PaddlesTutorial.GameScreens
{
    public enum MenuState
    {
        Main,
        Help
    }
    class MenuScreen
    {
        MenuState mState = MenuState.Main;
        int selection;

        public void Update(GameTime gameTime)
        {
            #region Main
            //Code to update when we are in the Main area
            #endregion
            #region Help
            //Code to update when we are in the Help area
            #endregion
        }
        public void Draw()
        {
            #region Main
            //Code to draw when we are in the Main area
            #endregion
            #region Help
            //Code to draw when we are in the Help area
            #endregion
        }
    }
}
```

We will be editing the regions after we create the Text class.

Part IV Text Class

Instead of making a mess with drawstring calls and a list of strings, positions, color, and other statistics of text, we are going to create a Text class that will handle all of this for us. This way all we need is a list of Text objects. Use the following bullets to create a text class.

- Variables
    - text of type string
    - position of type Vector2
    - aColor, bColor, and sColor of type Color. aColor holds the current color of the text (active color), bColor holds the basic color of the text (non-selected item), and sColor holds the selected color of the item.
    - Position (capital P) property that gets and sets the position vector with access as public.
    - active of type bool, and initially set as false.
    - Active (capital A) property that will get and set the text's activation
- Constructor with a string and a vector2 parameter. Inside, initialize the object to have a text of the passed string, a position of the passed Vector2, bColor of black, and sColor of yellow.
- Update method with an access of public and a return type of void. We will implement this later.
- Draw method with an access of public and a return type of void. The Draw method will have a SpriteBatch parameter.

Text class

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using PaddlesTutorial.GameScreens;

namespace PaddlesTutorial
{
    class Text
    {
        string text;
        Vector2 position;
        public Vector2 Position
        {
            get { return position; }
            set { position = value; }
        }
        Color aColor, bColor, sColor;
        bool active = false;
        public bool Active
        {
            get { return active; }
```

```
                set { active = value; }
            }
            public Text(string text, Vector2 position)
            {
                this.text = text;
                this.position = position;
                this.aColor = this.bColor = Color.Black;
                this.sColor = Color.Yellow;
            }
            public void Update()
            {
                //Update the text's activation + color
            }
            public void Draw(SpriteBatch spriteBatch)
            {
                //Draw the string
            }
        }
    }
```

Editing Menu Screen

We now need two lists of type Text to hold the menu entries, and help contents. Every text position's Y coordinate will be the last text position's Y coordinate + 5 (or any value you like). Create a constructor that will add the following text to the appropriate list.

- Menu Entries
    - Play Game
    - Help
    - Quit
- Help Contents (whenever you see *KEY*, you decide which key you want to control. I will show which keys I used so you can use the same)
    - Help (this is a title)
    - *KEY* – Move Up
    - *KEY* – Move Down
    - *KEY* (in main menu) – Quits the game
    - < Press *KEY* to go back to the main menu

You are welcome to change the wording of these if you wish. Also in the constructor, initialize selection to 0.

Menu Screen

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Xna.Framework;

namespace PaddlesTutorial.GameScreens
{
    public enum MenuState
    {
        Main,
        Help
    }
    class MenuScreen
    {
        List<Text> menuEntries, helpText;
        MenuState mState = MenuState.Main;
        int selection;
        public MenuScreen()
        {
            selection = 0;

            #region Menu Items
            menuEntries = new List<Text>();
            menuEntries.Add(new Text("Play Game", new Vector2(30,
80)));
            menuEntries.Add(new Text("Help", new Vector2(30,
menuEntries[0].Position.Y + 5.0f)));
            menuEntries.Add(new Text("Quit", new Vector2(30,
menuEntries[1].Position.Y + 5.0f)));
            #endregion
            #region Help Items
            helpText = new List<Text>();
            helpText.Add(new Text("Help", new Vector2(30, 80)));
            helpText.Add(new Text("Up Arrow - Move Up", new
Vector2(30, helpText[0].Position.Y + 5.0f)));
            helpText.Add(new Text("Down Arrow - Move Down", new
Vector2(30, helpText[1].Position.Y + 5.0f)));
            helpText.Add(new Text("Escape (In Main Menu) - Quits the
game", new Vector2(30, helpText[2].Position.Y + 5.0f)));
            helpText.Add(new Text("<  Press Backspace to go back to
the Main Menu", new Vector2(30, helpText[3].Position.Y + 5.0f)));
            #endregion
        }
        public void Update(GameTime gameTime)
        {
            #region Main
            //Code to update when we are in the Main area
            #endregion
            #region Help
            //Code to update when we are in the Help area
            #endregion
        }
        public void Draw()
        {
            #region Main
            //Code to draw when we are in the Main area
            #endregion
            #region Help
            //Code to draw when we are in the Help area
            #endregion
        }
    }
}
```

Now that we have some text, let's make more changes to make the menu screen appear (follow the following guidelines)

Initialize SpriteFont object

In order to draw strings, we need a SpriteFont object and we also need it initialized with a certain SpriteFont file (the XML file a few pages above). In order to initialize the object, we need a ContentManager object. Luckily we have one thanks to making on in the Game1.cs. Later in the series, you will see how to create powerful games that will not require static variables.

ScreenManager.cs *LoadContent* method, <u>add</u> the following

```
spriteFont = Game1.content.Load<SpriteFont>("basic");
```

Adding Drawing Capability in the Text Class

Now that we have a valid SpriteFont object, we can call the DrawString method.

Text.cs *Draw* method, <u>change</u> to

```
public void Draw(SpriteBatch spriteBatch)
{
    spriteBatch.DrawString(ScreenManager.spriteFont, this.text,
this.position, this.color);
}
```

Adding SpriteBatch Parameter in MenuScreen

Currently, SpriteBatch is not provided in the MenuScreen's Draw method, so let's add it!

MenuScreen.cs add a Using

```
using Microsoft.Xna.Framework.Graphics;
```

MenuScreen.cs change Draw header to

```
public void Draw(SpriteBatch spriteBatch)
```

Adding ScreenManager object

The screen manager handles all the screens. We need an object of this class in the Game1 class and we need to add it to our game using Components.Add.

Game1.cs Add the following attribute

```
ScreenManager screenManager;
```

Game1.cs Initialize the ScreenManager object in the constructor

```
screenManager = new ScreenManager(this);
```

Game1.cs Add the ScreenManager to the game components in the constructor AFTER the initialize line

```
Components.Add(screenManager);
```

Adding Draw Calls to the Menu Screen

MenuScreen.cs, add draw calls using if to determine what to draw.

```
public void Draw(SpriteBatch spriteBatch)
{
    #region Main
    if (mState == MenuState.Main)
    {
        foreach (Text t in menuEntries)
            t.Draw(spriteBatch);
    }
    #endregion
    #region Help
    else
    {
        foreach (Text t in helpText)
            t.Draw(spriteBatch);
    }
    #endregion
}
```

Add SpriteBatch argument for screen calls in ScreenManager.

ScreenManager.cs, add spritebatch to the screen's draw calls.

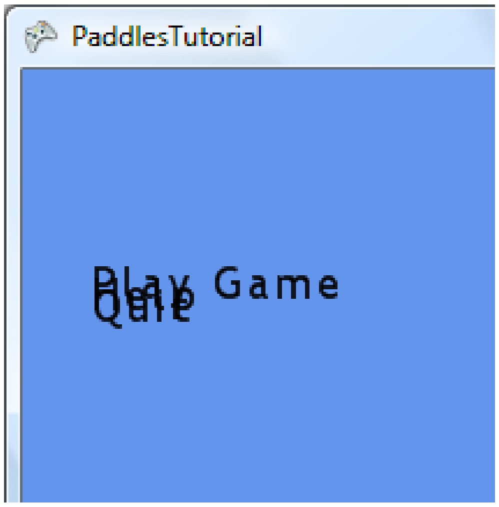
```
screen_Menu.Draw(spriteBatch);
```

Adding spriteBatch.Begin(); and spriteBatch.End();

ScreenManager.cs, change Draw method to

```
public override void Draw(GameTime gameTime)
        {
            spriteBatch.Begin();
            screen_Menu.Draw(spriteBatch);
            base.Draw(gameTime);
            spriteBatch.End();
        }
```

Output So Far



The Text is Messy!

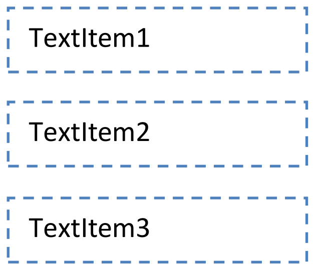
At least we know our game is working like it should at this point. We can fix the text now. The reason the text is messed up is because of the way we set the positioning. We need to find the measurement of the text in order to display it properly. To do so, we use the SpriteFont's object's MeasureString method. Let's add a Vector2 property (get only) in the Text class so retrieve the measurement.

Text.cs, add a Vector2 property named Size that will get the measure string

```
public Vector2 Size
{
    get { return ScreenManager.spriteFont.MeasureString(this.text); }
}
```

Changing the Positions

Now that we can get the size quickly, we need to fix the positions. Here is a graphic that shows what we need to accomplish.



We set an initial text position for item 1. The other items are relative to each other. You notice we do not change the X coordinate, just the Y coordinate. To set the Y coordinate, it needs to be the last text item's Y coordinate + last text item's Y measurement + 5.

Now, we have a problem. If we provide this in the MenuScreen constructor, our program will crash before beginning due to a null exception. So change the constructor to have 5.0f as Y values (except the first item in each list), and create a new method called UpdateTextPositioning() that will add the text size to the text position vector. This method will have a return type of void and will set every position relative to the last (besides the first text item in both lists).

MenuScreen.cs, Change constructor to

```
public MenuScreen()
{
    selection = 0;

    #region Menu Items
    menuEntries = new List<Text>();
    menuEntries.Add(new Text("Play Game", new Vector2(30, 80)));
    menuEntries.Add(new Text("Help", new Vector2(30, 5.0f)));
    menuEntries.Add(new Text("Quit", new Vector2(30, 5.0f)));
    #endregion
    #region Help Items
    helpText = new List<Text>();
    helpText.Add(new Text("Help", new Vector2(30, 80)));
    helpText.Add(new Text("Up Arrow - Move Up", new Vector2(30,
5.0f)));
    helpText.Add(new Text("Down Arrow - Move Down", new Vector2(30,
5.0f)));
    helpText.Add(new Text("Escape (In Main Menu) - Quits the game",
new Vector2(30, 5.0f)));
    helpText.Add(new Text("Escape (While Playing) - Pause Menu", new
Vector2(30, 5.0f)));
    helpText.Add(new Text("<  Press Enter or Escape to go back to
the Main Menu", new Vector2(30, 50.0f)));
    #endregion
}
```

MenuScreen.cs, Add the following method

```
public void UpdateTextPositioning()
{
    #region Menu Items
    for (int i = 1; i < menuEntries.Count; i++)
        menuEntries[i].Position += new Vector2(0, menuEntries[i -
1].Position.Y + menuEntries[i - 1].Size.Y);
    #endregion
    #region Help Items
    for (int i = 1; i < helpText.Count; i++)
        helpText[i].Position += new Vector2(0, helpText[i -
1].Position.Y + helpText[i - 1].Size.Y);
    #endregion
}
```

Calling the *UpdateTextPositioning()* Method

ScreenManager.cs, Add the following line after spriteFont has been loaded

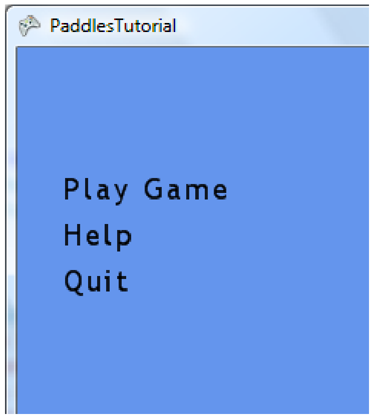
```
screen_Menu.UpdateTextPositioning();
```

Entire LoadContent() Method in ScreenManager

ScreenManager.cs, LoadContent Method

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    spriteFont = Game1.content.Load<SpriteFont>("basic");
    screen_Menu.UpdateTextPositioning();
}
```

Output So Far



Section V

Section Overview

This section is devoted for input. For this game, we need to keep input as simple as possible. Therefore, we will only implement the use of the keyboard. Let's discuss input a little first.

Part I Keyboard Discussion

Using keyboard for input should always be a first priority for new game developers. There is no need to handle various inputs for your first game, keyboard only is good enough. What we do to gather any input, is take a snapshot of their state (keyboard, gamepad, joystick, and so on) and that state contains button presses, key presses, or joystick vectors. Once we have those, we can have various if statements to handle our game's response to those keys. In this game, we will be using properties that will call a method to see if a key was pressed. As we get into more advanced games, we will look at input mapping and gamepads.

Part II Keyboard Class

As stated earlier, our input handling will be very basic for this tutorial. I will also only cover keyboard for now; however, you can add gamepad support on your own if you wish. Next tutorial we will cover gamepad. Now, let's start building the keyboard class.

- 3 using statements (`using Microsoft.Xna.Framework.Input;, using Microsoft.Xna.Framework;, and using PaddlesTutorial.GameScreens;`)
- Make class public class
- Fields/Attributes
  - A KeyboardState object called `CurrentKeyboardState`, and another called `LastKeyboardState.`
- Properties
  - Create a property for each action (MenuUp, MenuDown, MoveUp, MoveDown, Pause, and so on). These properties will only have get inside them. Inside the get paragraph, we call a method to determine if that specific key to perform that action has been pressed. If it has, that method returns true and we will know to perform an appropriate action.
- Methods
  - Update method to set the last keyboard state as the CurrentKeyboardState object, get the current loop's keyboard state.

- IsKeyPress to determine if a key has been pressed. Parameter is a type Keys. Method to determine if a key is pressed `CurrentKeyboardState.IsKeyDown(KEY);`
- IsNewKeyPress to determine if a pressed key is newly pressed. This is required for menu operations, since the selection would move too fast when you pressed the key with the other method (if you hold a key for longer than 16.66 milliseconds, it will cause some problems).

Keyboard Class

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Input;
using PaddlesTutorial.GameScreens;

namespace PaddlesTutorial
{
    public class Keyboard
    {
        public KeyboardState CurrentKeyboardState,
LastKeyboardState;

        public bool PauseOrQuit
        {
            get { return IsNewKeyPress(Keys.Escape); }
        }
        public bool MenuSelection
        {
            get { return IsNewKeyPress(Keys.Enter); }
        }
        public bool Up
        {
            get
            {
                if (ScreenManager.gameState == GameState.Menu)
                    return IsNewKeyPress(Keys.Up);
                else
                    return IsKeyPress(Keys.Up);
            }
        }
        public bool Down
        {
            get
            {
                if (ScreenManager.gameState == GameState.Menu)
                    return IsNewKeyPress(Keys.Down);
                else
                    return IsKeyPress(Keys.Down);
            }
        }

        public void Update()
        {
            //Since we need to update the currentkeyboardstate, it
actually holds last loops
            //state.  So set the LastKeyboardState as
CurrentKeyboardState before we update it
            LastKeyboardState = CurrentKeyboardState;

            CurrentKeyboardState =
Microsoft.Xna.Framework.Input.Keyboard.GetState();
        }
        public bool IsKeyPress(Keys key)
        {
            return CurrentKeyboardState.IsKeyDown(key);
        }
        public bool IsNewKeyPress(Keys key)
        {
            return (LastKeyboardState.IsKeyUp(key) &&
CurrentKeyboardState.IsKeyDown(key));
        }
    }
}
```

The Microsoft.Xna.Framework.Input.Keyboard.GetState(); Line

The reason we need the long call line is because our class is called Keyboard as well. If you want to shorten that line, you can change the class name to Input instead of Keyboard.

Important Note

I combined the MoveUp, MoveDown, MenuUp, and MenuDown into two properties Up and Down. As a result, I needed to add a if inside each get in order to determine which method to call. For menu actions, we want NEW key presses and for gameplay, we want regular key presses. This way our code is shorter.

Section VI

Section Overview

Now that we have a way to retrieve input, we can finish the menu screen. All we need to do is add a line in the Screen Manager to update the keyboard object every loop, then have the menu screen perform actions depending on what key has been pressed.

Part I Menu Screen Modifications + Screen Manager Changes

Let's Begin by Editing the ScreenManager.

As stated in the section overview, we need to update the keyboard object every loop. We also need to add a if statement to update the appropriate screen.

ScreenManager.cs, Inside Update method, add the following before base.Update

```
keyboard.Update();

if (gameState == GameState.Menu)
    screen_Menu.Update(gameTime);
```

ScreenManager.cs, Inside Draw method, add the following before ScreenMenu.Draw

```
if (gameState == GameState.Menu)
```

Adding Text Highlighting

Our menu comes up as all black right now, how are players supposed to know which menu item has focus for actions? As you saw earlier when we built the Text class, we made 3 color objects. We named one color to hold the current color of the text, another one as aColor to hold what color the text will be if active, and finally bColor to hold what color the text will be in its basic form (normal text). Now we need a way to change the colors around. In order to do so, we need to implement the Update method inside the Text class. Use the following bullets to help.

- If the player presses Up or Down, AND the selection is out of range (0 to MenuEntries.Count – 1 is the range), then change the selection to the appropriate edge of the range.
- If the text is active AND the color is still the basic color, change the color to the active color.
- If the text is NOT active AND the color is still the active color, change the color to the basic color.

Text.cs, Update method implementation

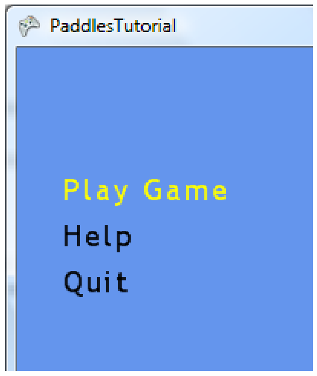
```
public void Update()
{
    if (this.active && this.color == this.bColor)
        this.color = this.aColor;
    else if (!this.active && this.color == this.aColor)
        this.color = this.bColor;
}
```

Now we need to do something inside the MenuScreen, which is to implement its Update method. You notice we have a variable called selection of type int. This is to determine which menu item is activated. We need to add keyboard support in this update method as well as updating all menu items and help items. Let's build the Menu region first.

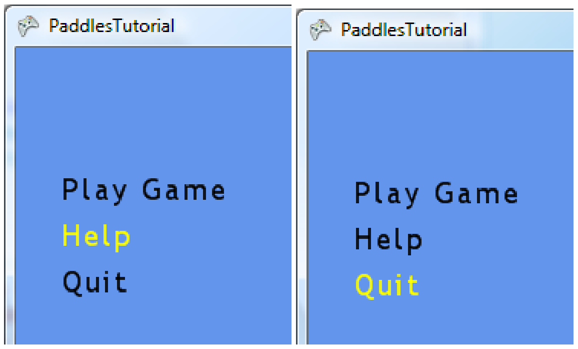
MenuScreen.cs, Inside Update method, add the following inside the Main region

```
#region Main
if (mState == MenuState.Main)
{
    if (ScreenManager.keyboard.Down)
    {
        if (selection < menuEntries.Count - 1)
            selection++;
        else
            selection = 0;
    }
    else if (ScreenManager.keyboard.Up)
    {
        if (selection > 0)
            selection--;
        else
            selection = menuEntries.Count - 1;
    }
    else if (ScreenManager.keyboard.PauseOrQuit)
        ScreenManager.isExiting = true;
    for (int i = 0; i < menuEntries.Count; i++)
    {
        if (i == selection)
        {
            if (!menuEntries[i].Active)
                menuEntries[i].Active = true;
        }
        else
        {
            if (menuEntries[i].Active)
                menuEntries[i].Active = false;
        }
        menuEntries[i].Update();
    }
}
#endregion
```

Output So Far



Pressing Up and Down



Adding MenuSelect Action

Now that we have moving support for our menu, let's add action support (player presses enter). To do so, I will be using a switch statement. You can use whatever you want to determine what to do. I will be switching the selection variable and perform actions based on its value. If we press enter while we are on Play Game (selection 0), we need to change the game state to play. If we press enter while Help is selected, we need to change the menu state to Help. Finally, if we press enter while Exit is selected, we need to set ScreenManager.isExiting to true.

MenuScreen.cs, Add the following below `ScreenManager.isExiting = true;`

```
else if (ScreenManager.keyboard.MenuSelection)
{
    switch (selection)
    {
        case 0: ScreenManager.gameState = GameState.Play; break;
        case 1: mState = MenuState.Help; break;
```

```
         case 2: ScreenManager.isExiting = true; break;
    }
}
```

ScreenManager.cs, Adding game quit line in the update method.

```
if (isExiting)
    this.Game.Exit();
```

Filling the Help Regions

There is one last thing we need to do before we are done with the menu system. We need to make the help screen work. In order to do so, we need to fill in the Help region in the Update method of MenuScreen.cs.

MenuScreen.cs, Adding code to the Help region in the Update method.

```
#region Help
else
{
    if (ScreenManager.keyboard.MenuSelection ||
ScreenManager.keyboard.PauseOrQuit)
        mState = MenuState.Main;

    if (!helpText[5].Active)
    {
            helpText[5].Active = true;
            helpText[5].Update();
    }

}
#endregion
```

Now our menu does what we want. Believe it or not, the game is nearly finished. The coding may be way too hard coded, but that's the way it should be for your first game. Next game, we will do more OOP than hard coded. I will also show you how to build a game a lot faster than we are with this (since this is your first game).

Section VII

Section Overview

This section is where we build the actual gameplay. We will implement the paddle class and ball class. Once those are implemented, we need to modify the gameplay screen to include objects of these classes.

Part I Paddle Class

Finally, the actual gameplay! This part is very simple since the gameplay is not very advanced. The first thing we need to do is build the Ball and Paddle classes.

Paddle Class

This is the class for the player controlled and computer controlled paddles. In order to determine which is which, we need to create another enum list called Controlled or a name of your choice. We need an attribute to hold a Controlled object. We also need two Vector2 objects to hold position and velocity. Create a variable to hold the speed of the paddle, this variable will be of type float. Finally, we need an object of Rectangle to hold the boundries.

Underneath the boundary object, create a public property that will get the boundary (this is for the ball class to have access to the object). As for the methods, we need the usual constructor, update, and draw (with SpriteBatch parameter) methods. The constructor will take three arguments: position of type Vector2, speed of type float, and playerControlled of type bool. Update will take one argument, a ball object, and inside will have two regions. One region for player controlled, and another region for computer controlled. Add if statements inside those regions. Make the class public.

Paddle.cs Class listing

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Xna.Framework;

namespace PaddlesTutorial.GameObjects
{
    public enum Controlled
    {
        Player,
        Computer
    }
    public class Paddle
    {
        Vector2 position, velocity;
        Rectangle boundary;
        public Rectangle Boundary
        {
            get { return boundary; }
        }
        Controlled control;
        float speed;

        public Paddle(Vector2 pos, float speed, bool
playerControlled)
        {
            position = pos;
            velocity = Vector2.Zero;
            this.speed = speed;
            if (playerControlled)
                control = Controlled.Player;
            else
                control = Controlled.Computer;
        }
        public void Update(Ball ball)
        {
            #region Player Controlled
            if (control == Controlled.Player)
            {
                //Update player paddle
            }
            #endregion
            #region Computer Controlled
            else
            {
                //Update computer paddle
            }
            #endregion
        }
        public void Draw(SpriteBatch spriteBatch)
        {
            //Draw paddle
        }
    }
}
```

Adding Code to the Update Method

We need to start this method by setting the velocity vector to the Zero vector.

Paddle.cs - Update Method, add the following above both regions

```
velocity = Vector2.Zero;
```

Now we need to fill the Player Controlled region. All the updating required for this region is accepting movement based on the keyboard object. If we press Up, create a velocity with a negative Y value. If we press Down, create a velocity with a positive Y value. After we do that, we need to update the position vector by adding the velocity vector to it. And finally, we need to update the boundary object. You will need the size of the paddles to properly update the boundary object, I choose 10 pixels wide and 50 pixels tall as the paddle size. Since we need the boundary object updated for both paddles, add the line after both regions. In order to call the ScreenManager class, add another using: `using PaddlesTutorial.GameScreens;`

Paddle.cs, Edited Update Method

```
public void Update(Ball ball)
{
    velocity = Vector2.Zero;
    #region Player Controlled
    if (control == Controlled.Player)
    {
        if (ScreenManager.keyboard.Up)
            velocity.Y = speed * -1;
        else if (ScreenManager.keyboard.Down)
            velocity.Y = speed;

        position += velocity;
    }
    #endregion
    #region Computer Controlled
    else
    {
        //Update computer paddle
    }
    #endregion
    boundary = new Rectangle((int)position.X, (int)position.Y, 10,
50);
}
```

Making Sure Paddles do not Go Off-screen

We do not want our paddles, and ball, to fly off the screen. The way to prevent this is to set a max and min value the paddle can reach. If the paddle reaches the min value and still wants to travel up, it will not be able to by setting the position in the code to the min value. The same with the max value. Now, how do we get the max size? By default, any Windows Game you create with XNA will have the dimensions 800 by 600. So for this game, hard-code the max value. For later games, we will not hard-code this. The min value I chose is 10, and the max value is 600 – 60 or 540 (10 pixel gap on the top and bottom sides of the window).

Paddle.cs, Check Position region above boundary update.

```
#region Check Position
if (position.Y < 10)
    position.Y = 10;
else if (position.Y > 540)
    position.Y = 540;
#endregion
```

Computer Controlled Update

Our player update is finished. Now we need to handle the computer controlled paddle. This part is a little hard to make the computer hard, but not impossible. You noticed we have a ball parameter in the Update method. When the ball is traveling toward the player (the ball's velocity will have a negative value for X) the computer paddle will not move. Now, we do not have the ball class implemented yet, but call ball.Velocity anyway. To make things even harder for the computer, we should calculate the Y distance between the paddle and ball. If the distance is smaller than a certain amount (I chose 15) from the center, we will not move. Keep in mind we are not looking for amazing AI, this is your first game. We can come back and visit this AI once you learned more about game design.

Paddle.cs, Computer Controlled Update

```
#region Computer Controlled
else
{
    if (ball.Velocity.X > 0)
    {
        float distance = position.Y + 25 - ball.Position.Y;
        if (distance > 15)
            velocity.Y = speed * -1;
        else if (distance < -15)
            velocity.Y = speed;

        position += velocity;
    }
}
#endregion
```

Next is the ball class, so let's begin!

Ball Class

This class will be very similar to the paddle class. The ball size will be 10 by 10. Use the following bullets to help you build the class.

- 2 Vector2 objects called position and velocity.
- 2 public Vector2 properties to get the position and get the velocity.
- Rectangle object called boundary.
- 2 float variables called speed and angle.
- Constructor with a parameter called speed of type float. Inside, set the ball's speed to the passed float value and call a method named Reset().
- Reset method has access as private and return type of void. Inside this method we need to set the ball's position to the center (X: (800 / 2) – 5 Y: (600 / 2) – 5). After that is set, we need to find a random angle by calling another method (findRandomAngle()) which will be implemented later. Next, we need to apply some trigonometry to get the exact velocity. The X portion of the vector will be speed * cosine of the angle in radians. The Y portion of the vector will be speed * sine of the angle in radians. For the last line in this method, we need to update the boundary.
- findRandomAngle method has access as private and return type of void. Inside, we need a Random object called rAngle. We also need an integer called r to hold the int values we get from rAngle. Since we do not want the ball to fly in a straight line up, down, left, or right, we need to create a while loop so we do not get an even value.
  ```
  while (r % 2 == 0)
      r = rAngle.Next(1, 7);
  ```
  Once the while loop breaks, we need to set the ball's angle by taking the result * 45 and converting it to radians.
- Update method (public access with void return type) and Draw method (same as update, but have a SpriteBatch parameter). Another method needed is a collision method, and I called it Collision with an access as public and return type as void.

Ball.cs Class Listing

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

namespace PaddlesTutorial.GameObjects
{
    public class Ball
    {
        Vector2 position, velocity;
        public Vector2 Position
        {
            get { return position; }
        }
        public Vector2 Velocity
        {
            get { return velocity; }
        }
        Rectangle boundary;
        public Rectangle Boundary
        {
            get { return boundary; }
        }
        float speed, angle;

        public Ball(float speed)
        {
            this.speed = speed;
            Reset();
        }

        private void Reset()
        {
            speed = 6;
            position = new Vector2((800 / 2) - 5, (600 / 2) - 5);
            findRandomAngle();
            velocity = new Vector2((float)(speed * Math.Cos(angle)),
(float)(speed * Math.Sin(angle)));
            boundary = new Rectangle((int)position.X,
(int)position.Y, 10, 10);
        }
        private void findRandomAngle()
        {
            Random rAngle = new Random();
            int r = 0;
            while (r % 2 == 0)
                r = rAngle.Next(1, 7);

            angle = MathHelper.ToRadians(r * 45);
        }
        public void Update()
        {

        }
        public void Draw(SpriteBatch spriteBatch)
        {

        }
        public void Collision()
        {

        }
    }
}
```

Ball's Update Method

The ball object will be updated differently. Before anything is changed, we need to check the positioning of the ball. If the ball's Y position is right at the edge of the game window, we need to modify the Y velocity. If the ball's X position is at the edge of the game window, the paddle farthest from the ball will gain a point. Simply speaking, if the ball is at the left side of the game window, the computer gains a point. If the ball is at the right side of the window, the player gains a point. We will worry about how to keep score in a little bit. If the ball passes the left or the right side of the window, we need to reset the ball again by calling the Reset() method. Finally we need to add the position and velocity vectors together, and update the boundary object.

Ball.cs, Update Method

```
public void Update()
{
    if ((position.Y <= 0) || (position.Y >= 800))
        velocity.Y *= -1;
        if (position.X <= 0)
    {
        //Computer gets a point
        Reset();
    }
    else if (position.X >= 600)
    {
        //Player gets a point
        Reset();
    }
    position += velocity;
    boundary = new Rectangle((int)position.X, (int)position.Y, 10,
10);
}
```

Ball's Collision

When the ball collides with a paddle, it will begin traveling in the other X direction. Also, when a paddle collides with the ball, we need to increase the ball's velocity a bit in order to make it more challenging. We must finally update the speed of the ball by calling the Lengh() method of a vector. We need to set a max value the ball can go or else nothing can hit it.

Ball.cs, Collision Method

```
public void Collision()
{
    if (speed < 11)
    {
        velocity.Y *= 1.1f;
        velocity.X *= -1.1f;

        speed = (float)velocity.Length();
    }
    else
        velocity.X *= -1;

}
```

Section VIII

Section Overview

In this section we will be finalizing our game. All we have left to do is implement the game screen, slight modifications in the game object classes, and do some testing. Once we are finished testing, we will distribute our game so others can play it!

Part I Finalizing the Game

The game screen handles all objects, collisions, and textures for this game. Remember, the only texture we need is the singlePixel texture. Use the following bullets to help you build the class.

- 2 Paddle objects, player and computer
- 1 Ball object, ball
- 2 public static int variables, score_player and score_computer.
- 1 public static Texture2D object, pixel.
- Constructor
    - Initialize the player object with a position of (20, 275). Why 275? That Y value will center our paddle vertically.
    - Initialize the computer object with a position of (780, 275).
    - Both paddles should have a decent speed, I chose 8.
    - Initialize the ball object with a slightly slower speed than the paddles, I chose 6.
- LoadContent method (public access and void return type)
    - Initialize the pixel object using Game1.content to load the singlePixel image.
- Update method (public access, void return type, and GameTime as a parameter)
    - Call a CheckCollisions method (we have not made this method yet)
    - Update each game object
- CheckCollisions method (private access and void return type)
- Draw method (public access, void return type, and SpriteBatch as a parameter)
    - Draw all game objects.

GameScreen.cs, Class Listing

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using PaddlesTutorial.GameObjects;

namespace PaddlesTutorial.GameScreens
{
    public class GameScreen
    {
        Paddle player, computer;
        Ball ball;

        public static int score_player, score_computer;
        public static Texture2D pixel;

        public GameScreen()
        {
            player = new Paddle(new Vector2(20, 275), 8, true);
            computer = new Paddle(new Vector2(780, 275), 8, false);
            ball = new Ball(6);
        }
        public void LoadContent()
        {
            pixel = Game1.content.Load<Texture2D>("singlePixel");
        }
        public void Update(GameTime gameTime)
        {
            CheckCollisions();

            player.Update(ball);
            computer.Update(ball);
            ball.Update();
        }

        private void CheckCollisions()
        {
            //Check collisions
        }
        public void Draw(SpriteBatch spriteBatch)
        {
            player.Draw(spriteBatch);
            computer.Draw(spriteBatch);
            ball.Draw(spriteBatch);
        }
    }
}
```

Check Collisions Method

We need to check to see if objects come in contact with each other. The only object that can come in contact with other objects is the ball. The paddles can only move up and down so they cannot come in contact. There are some occasions where the ball will be in contact with a paddle longer than expected, even when we change its direction. As a result, the ball would appear jumpy and will stick to the paddle until you either move it, or the ball reaches the top. To fix this, we need to test the ball's velocity's X value.

CheckCollision Method

```
private void CheckCollisions()
{
    if (ball.Boundary.Intersects(player.Boundary))
    {
        if (ball.Velocity.X < 0)
            ball.Collision();
    }
    else if (ball.Boundary.Intersects(computer.Boundary))
    {
        if(ball.Velocity.X > 0)
            ball.Collision();
    }
}
```

Adding Draw Calls to Game Objects

With our current code, we cannot draw our game objects. We need to add draw calls to the Paddle class and the Ball class.

Paddle.cs, Draw method

```
public void Draw(SpriteBatch spriteBatch)
{
    spriteBatch.Draw(GameScreen.pixel, boundary, Color.White);
}
```

Ball.cs, Draw method

```
public void Draw(SpriteBatch spriteBatch)
{
    spriteBatch.Draw(GameScreen.pixel, boundary, Color.White);
}
```

Adding Game Screen Support in ScreenManager

There are three things we need to do in ScreenManager.cs. The first thing we need to do is call the LoadContent method. The second thing is adding the Update call, and the third is to add the Draw call.

ScreenManager.cs, At the end of LoadContent, add the following

```
screen_Game.LoadContent();
```

ScreenManager.cs, In the Update method, under `screen_Menu.Update(gameTime);` **add**

```
else if (gameState == GameState.Play)
    screen_Game.Update(gameTime);
```

ScreenManager.cs, In the Draw method, under `screen_Menu.Draw(spriteBatch);` **add**

```
else if (gameState == GameState.Play)
    screen_Game.Draw(spriteBatch);
```

Updating Scores and Displaying Them

The last thing we need to do for our game is to add the ability for our scores to be updated and drawn. If you look in the Ball class, we need to change the comments that say X gets a point to actual code to update the points.

Ball.cs, Update Method

```
public void Update()
{
    if ((position.Y <= 0) || (position.Y >= 600))
        velocity.Y *= -1;
    if (position.X <= 0)
    {
        GameScreen.score_computer++;
        Reset();
    }
    else if (position.X >= 800)
    {
        GameScreen.score_player++;
        Reset();
    }
    position += velocity;
    boundary = new Rectangle((int)position.X, (int)position.Y, 10,
10);
}
```

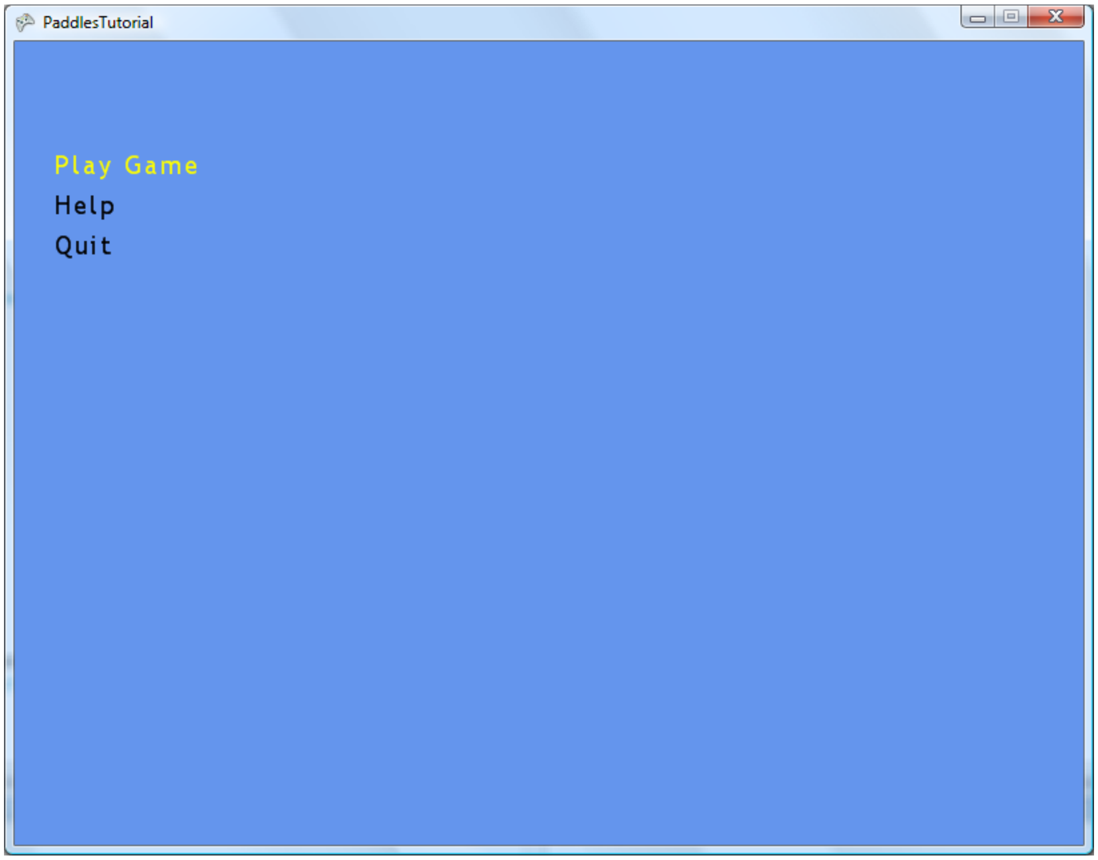
To display the scores, add DrawString calls to the GameScreen. Use any positioning you like. I chose (30, 10) for the player score, and (60, 10) for the computer score. I also chose to have the color Yellow.

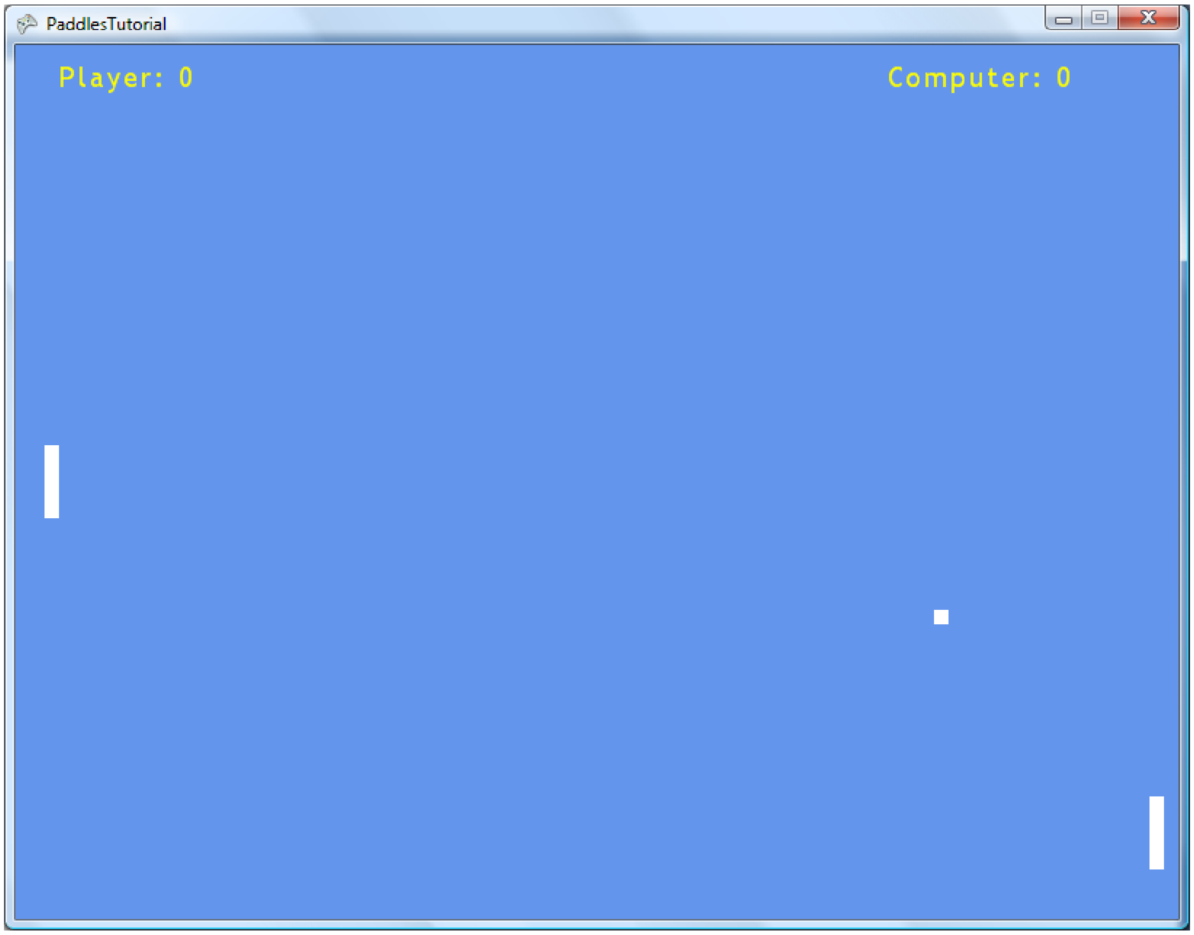
GameScreen.cs, Draw method

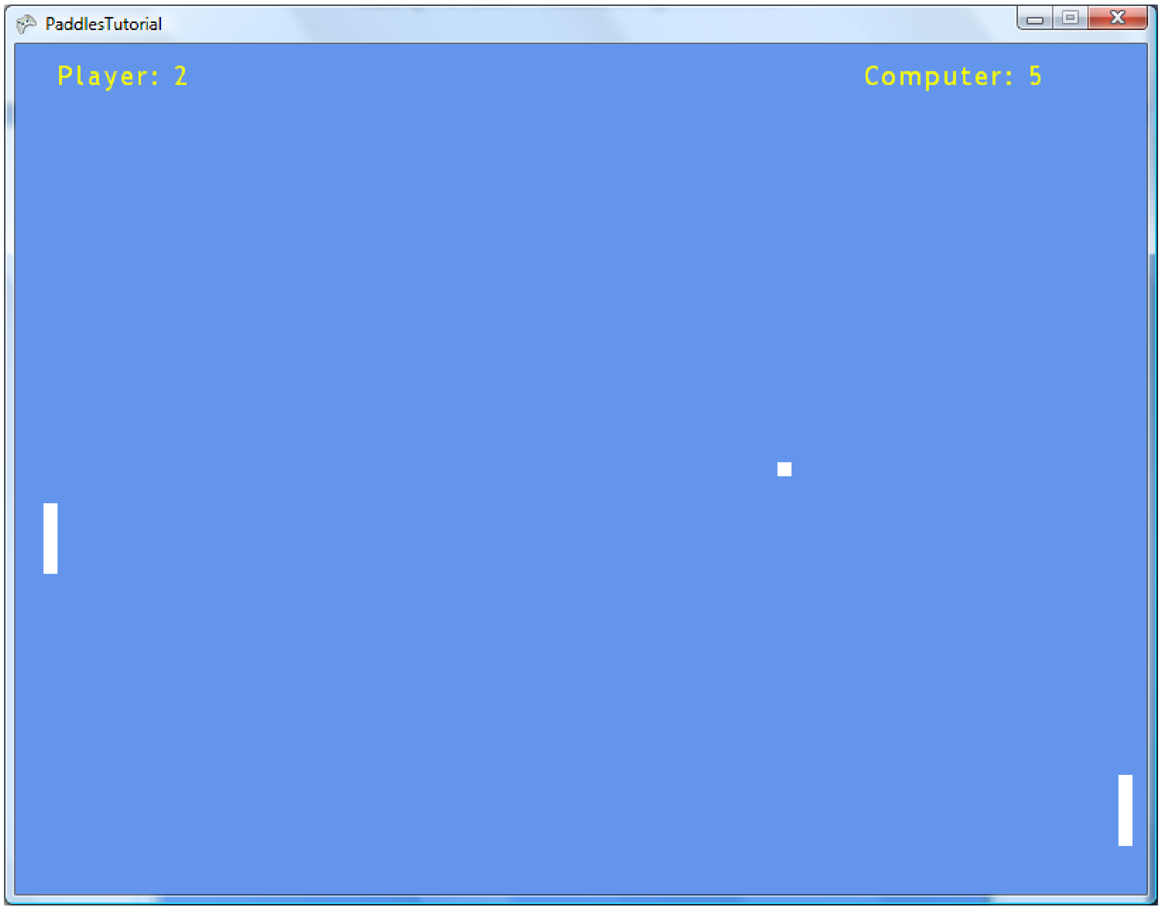
```
public void Draw(SpriteBatch spriteBatch)
{
    player.Draw(spriteBatch);
    computer.Draw(spriteBatch);
    ball.Draw(spriteBatch);

    spriteBatch.DrawString(ScreenManager.spriteFont, "Player: " +
score_player, new Vector2(30, 10), Color.Yellow);
    spriteBatch.DrawString(ScreenManager.spriteFont, "Computer: " +
score_computer, new Vector2(600, 10), Color.Yellow);
}
```

Part II Testing

Now that our game is finished, give it a test run. Remember, we decided to do very basic AI for this game, so some weird things can occur. If you want to change it, you can read up on AI at various places. But for a first game, it is good and not too complicated.


PaddlesTutorial
Play Game
Help
Quit


PaddlesTutorial
Player: 0
Computer: 0


PaddlesTutorial
Player: 2
Computer: 5

Section IX

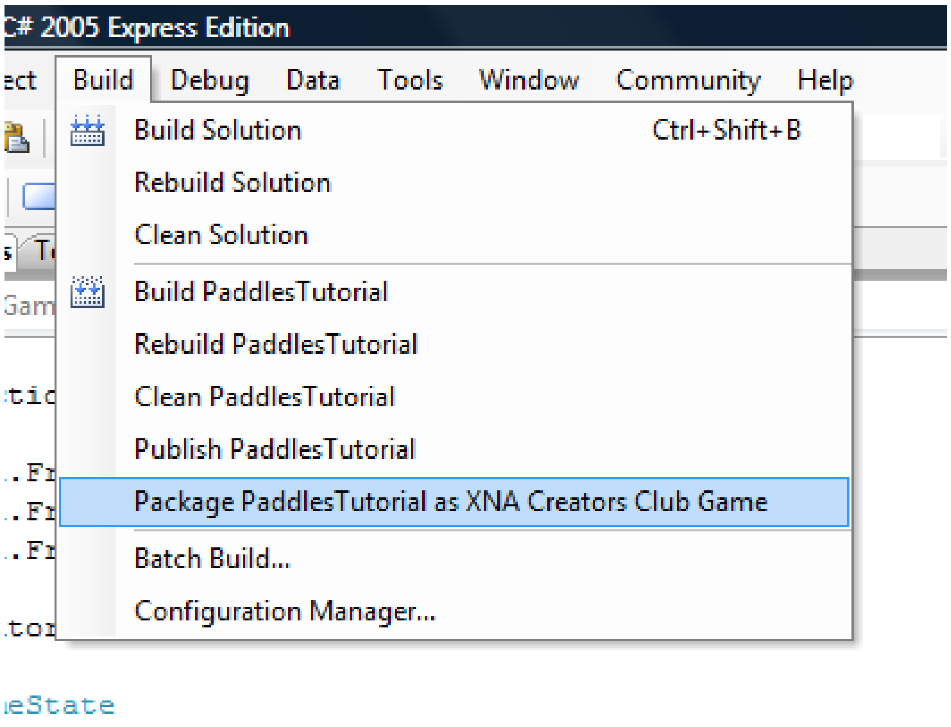
Section Overview

This section is devoted to distribution. We will only cover Windows distribution in this tutorial, but two ways to do so.

Part I Windows

First Way, XNA CC Users Only

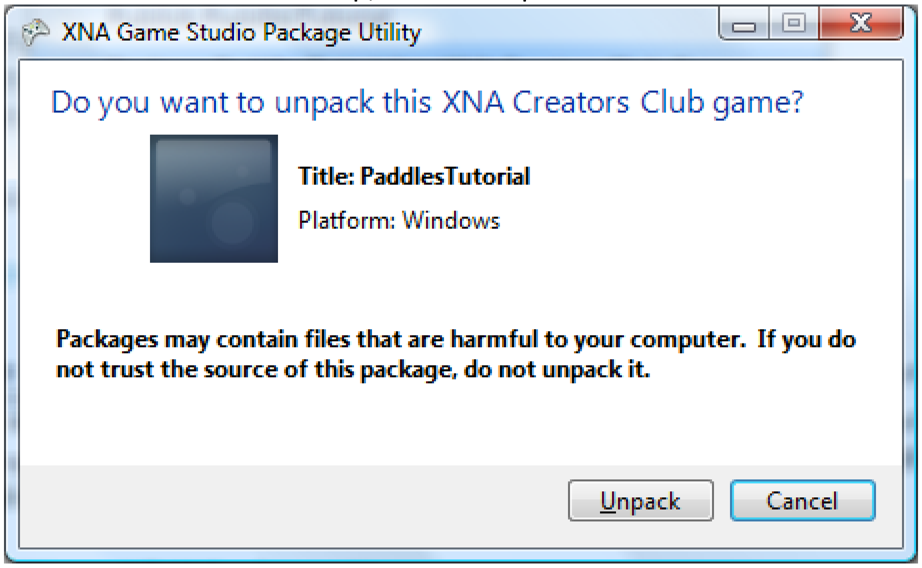
This way is simple. Just open up the game source, go to Build, and select Package PaddlesTutorial as XNA Creators Club Game.



Once finished, you will see a confirmation in the Output window: `Successfully built XNA Creators Club Game package: "C:\Users\Whiplash\Desktop\PaddlesTutorial\PaddlesTutorial\bin\x86\Debug\PaddlesTutorial-Windows.ccgame".` Distribute this .ccgame file to those who have XNA Game Studio so they can play it.

Second Way, All Windows Users

In order for the average user to run XNA games, they need both XNA and .NET frameworks, and the DirectX runtime. One they user has those, they should be able to run the game. Complete the first way, but instead of distributing the .ccgame file, open it. A new window will come up, choose Unpack.



Do you want to unpack this XNA Creators Club game?

Title: PaddlesTutorial

Platform: Windows

Packages may contain files that are harmful to your computer. If you do not trust the source of this package, do not unpack it.

After the game has been unpacked, a new window will show up showing you the files. Simply add those files to a compressed archive (.zip or .rar for example) and distribute that.

Congratulations!

Your First Game is Complete!