XNA 4.0 RPG Tutorials
Part 1
Getting Started

I'm writing these tutorials for the MonoGame 3.8 framework using Visual Studio 2019. The tutorials will make more sense if they are read in order. You can find the list of tutorials on the Eyes of the Dragon page of my web blog. I will be making each version of the project available on GitHub here. It will be included on the page that links to the tutorials.

This is part one in a series of tutorials I plan to write on creating a role playing game with MonoGame. I've worked on similar tutorials in the past using XNA. I have discovered better ways of doing some things in the process of writing more tutorials and had to go back to fix things. I'm hoping in this series to not make those same mistakes.

To get started we want to create a new MonoGame project. Open Visual Studio 2019. From the **Start Page** menu select the **Create a new project tile**. In the dialog box that shows enter **MonoGame Cross-Project** in the search box. Select the **MonoGame Cross-Platform** tile and click Next. Name this new game **EyesOfTheDragon**. Visual Studio will create the basic solution for you.

In MonoGame there are two graphics profiles your game can use. There is the **HiDef** profile that uses Shader Model 3.0 or greater. The **Reach** profile uses Shader Model 1.1 or greater. By default MonoGame uses the **Reach** profile.

I want to add in two class libraries to this project. The first will be a standard class library that holds generic classes that can be reused in other projects. The other is a MonoGame standard library. This library will hold classes that can be shared between MonoGame projects. Right click your solution this time in the solution explorer, not the project. Select the **Add** item and then the **New Project** entry.

In the dialog that shows up search for **MonoGame NetStandard**. Choose that tile on the left and click the Next button. Name this new project **MGRpgLibrary**. Right click your solution in the solution explorer again, select **Add** and then **New Project** again. This time enter **.NET Standard**. Choose the Class Library (.NET Standard) tile and click Next. Name this new project **RpgLibrary**. In the **MGRpgLibrary** right click the **Game1.cs** entry and select **Delete** to remove it from the projects as we won't be it. Similarly, right click the **Class1.cs** entry in the **RpgLibrary** project and select **Delete** because we won't be using it.

There is one last thing you must do in order to use the libraries. You have to tell your game about them so Visual Studio knows to include them when it builds the game and you can use the members. You do that by adding references of the libraries to your game. Right click your game project, **EyesOfTheDragon**, in the solution explorer and select the **Add Project Reference** option. From the dialog box that pops up select the **Projects** tab. In the projects tab there will be entries for the two libraries. Hold down the control key and click both of them to select them both and then click **OK**.

With that done it is time to actually write some code! In a large game, and role playing games are large games, it is a good idea to handle some things on a global level. One such item is input in the game. A MonoGame game component would be perfect for controlling input in your game. When you create a game component and add it to the list of components for your game XNA will automatically

call its **Update** method, and **Draw** method if it is a drawable game component, for you. Game components have **Enabled** properties that you can set to true or false to determine if they should be updated. The drawable ones also have a **Visible** property that can be used the same way to determine if they should be drawn or not.

I'm going to add a MonoGame game component to the **XRpgLibrary** to manage all of the input in the game in one location. For this tutorial I'm only going to handle keyboard input. In future tutorials I will add in support for a mouse and Xbox 360 game pads and mice. Right click the **XRpgLibrary** project in the solution explorer, select **Add** and then **Class**. Name this new class **InputHandler**. I'm a firm believer in showing new code first so you can read it over before explaining things. I'm also breaking the code up into regions using preprocessor directives. They don't affect the code at all but they help to keep your code organized better. The code for the **InputHandler** class follows next.

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Input;

namespace MGRpgLibrary
{
    public class InputHandler : Microsoft.Xna.Framework.GameComponent
    {
        #region Field Region
        static KeyboardState keyboardState;
        static KeyboardState lastKeyboardState;
        #endregion

        #region Property Region
        public static KeyboardState KeyboardState
        {
            get { return keyboardState; }
        }

        public static KeyboardState LastKeyboardState
        {
            get { return lastKeyboardState; }
        }
        #endregion

        #region Constructor Region

        public InputHandler(Game game)
        : base(game)
        {
            keyboardState = Keyboard.GetState();
        }

        #endregion

        #region MG methods

        public override void Initialize()
        {
            base.Initialize();
        }

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

```
            lastKeyboardState = keyboardState;
            keyboardState = Keyboard.GetState();
            base.Update(gameTime);
        }

        #endregion

        #region General Method Region

        public static void Flush()
        {
            lastKeyboardState = keyboardState;
        }

        #endregion

        #region Keyboard Region

        public static bool KeyReleased(Keys key)
        {
            return keyboardState.IsKeyUp(key) &&
            lastKeyboardState.IsKeyDown(key);
        }

        public static bool KeyPressed(Keys key)
        {
            return keyboardState.IsKeyDown(key) &&
            lastKeyboardState.IsKeyUp(key);
        }

        public static bool KeyDown(Keys key)
        {
            return keyboardState.IsKeyDown(key);
        }

        #endregion
    }
}
```

Inside the class there is a region that will hold the fields of the class called **Field Region**. There are two fields in the region. One holds the current state of the keyboard, **keyboardState**, and the other holds the state of the keyboard in the previous frame of the game, **lastKeyboardState**. These fields are static, meaning that they are shared between all instances of the **InputHandler** class and can be referenced using the **InputHandler** class name rather than an instance variable.

Each time the game runs through the **Update** and **Draw** methods is considered to be a frame of the game. When you are looking for a single press of a key you need to compare the state of the keyboard in the last frame of the game and the current frame of the game. I will get to detecting single presses shortly.

The next region it the **Property Region**. This region holds properties that expose the fields of the class. What I mean by expose is that C# is an object oriented programming language and I'm following the basic principles of object oriented programming. The principle that I'm using here is encapsulation.

The idea of encapsulation means that other objects are denied access to the inner works of a class

and access the members of a class through a public interface. This public interface "exposes" what you want to share of the objects of a class. The **LastKeyboardState** and **KeyboardState** properties return the **lastKeyboardState** and **keyboardState** fields respectively.

In the **Constructor Region** is the constructor. The constructor requires a **Game** argument because the base class **GameComponent** requires a **Game** parameter. I set the **keyboardState** field using the **GetState** method of the **Keyboard** class.

In the **Game Component Method Region** are two methods that we are overriding from the base class. They are the **Initialize** and **Update** methods. There is no new code in the **Initialize** method. In the **Update** method I set the **lastKeyboardState** field to be the **keyboardState** field and get the new state of the keyboard. What this does is make sure that **lastKeyboardState** will hold the state of the keyboard in the last frame of the game and **keyboardState** will hold the current state of the keyboard.

The last region in this class is the **Keyboard Region**. In this region I have all of the methods that are related to the keyboard. There are three methods in this region. Two of them are used for detecting single presses, **KeyPressed**, and releases, **KeyReleased**. The other is for determining if a key is down, **KeyDown**.

There is a good reason for having a method for checking for both single presses and releases and not one or the other. Checking for a release is good for things like menus. You don't want the action to occur precisely the moment the key is down. If you check for a press instead of a release in this instance the press may spill over if you change states. There are times when you want instant action, like firing a bullet. You don't want to wait for the button to be release before firing.

To detect a release you check if a key was down in the last frame and up in the current frame. For a press it is reversed. You check if a key that was up in the last frame is now down.

I skipped over one region, the **General Method** region. In this region there is just one method, **Flush**. I couldn't think of a better name for the method. The **Flush** method sets the **lastKeyboardState** field to the **keyboardState** field. What this does is make the **KeyPressed** and **KeyReleased** methods return false. It flushes what I like to call the input buffer. You can use it to make sure, if you are changing state for example, that there is no spill over into the next state.

Now we can add this game component to the list of components for the game. That will take place in the **Game1** class. This is the main class for you game. The first step is to add a using statement for the **MGRpgLibrary** name space to bring it into scope. What is meant by bringing a name space into scope is that you can refer objects with out using their fully qualified name. For example, in the **Game1** class you can reference the **InputHandler** class using **InputHandler** instead of its fully qualified name **MGRpgLibrary.InputHandler**. You will then add an instance of the **InputHandler** class to the list of components of the game. Add the following using statement with the other using statements of the **Game1** class and change the constructor to the following.

```
using MGRpgLibrary;

public Game1()
```

```
{
   _graphics = new GraphicsDeviceManager(this);
   Content.RootDirectory = "Content";
   Components.Add(new InputHandler(this));
}
```

A common mistake made when creating a large game is not handling game state properly from the start. If you try and go back and add it in later it is a major pain and one of the hardest things to fix if not handled properly from the start. I will be managing game states, or game screens, with a game component. I will need a basic state, or screen, to start from. Right click the **MGRpgLibrary** project, select **Add** and then **Class**. Name this game component **GameState**. This is the code for the **GameState** class.

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

namespace MGRpgLibrary
{
    public abstract partial class GameState : DrawableGameComponent
    {
        #region Fields and Properties

        List<GameComponent> childComponents;

        public List<GameComponent> Components
        {
            get { return childComponents; }
        }

        GameState tag;

        public GameState Tag
        {
            get { return tag; }
        }

        protected GameStateManager StateManager;

        #endregion

        #region Constructor Region

        public GameState(Game game, GameStateManager manager)
        : base(game)
        {
            StateManager = manager;
            childComponents = new List<GameComponent>();
            tag = this;
```

```
}

#endregion

#region MG Drawable Game Component Methods
public override void Initialize()
{
    base.Initialize();
}
public override void Update(GameTime gameTime)
{
    foreach (GameComponent component in childComponents)
    {
        if (component.Enabled)
            component.Update(gameTime);
    }

    base.Update(gameTime);
}

public override void Draw(GameTime gameTime)
{
    DrawableGameComponent drawComponent;

    foreach (GameComponent component in childComponents)
    {
        if (component is DrawableGameComponent)
        {
            drawComponent = component as DrawableGameComponent;
            if (drawComponent.Visible)
                drawComponent.Draw(gameTime);
        }
    }

    base.Draw(gameTime);
}

#endregion

#region GameState Method Region
internal protected virtual void StateChange(object sender, EventArgs e)
{
    if (StateManager.CurrentState == Tag)
        Show();
    else
        Hide();
}

protected virtual void Show()
{
    Visible = true;
    Enabled = true;
    foreach (GameComponent component in childComponents)
    {
        component.Enabled = true;
        if (component is DrawableGameComponent)
            ((DrawableGameComponent)component).Visible = true;
    }
}
```

```
        protected virtual void Hide()
        {
            Visible = false;
            Enabled = false;
            foreach (GameComponent component in childComponents)
            {
                component.Enabled = false;
                if (component is DrawableGameComponent)
                    ((DrawableGameComponent)component).Visible = false;
            }
        }

        #endregion
    }
}
```

When you create **Game Components you** inherit from **GameComponent**. Game states will do drawing so they need to derive from **DrawableGameComponent**. The three fields in the class are **childComponents**, a list of any game components that belong to the screen, **tag**, a reference to the game state, and **StateManager**, the game state manager that I haven't added yet. **StateManager** is a protected field and is available to any class that inherits from **GameState**, either directly or indirectly. There is a read only property, **Components**, that returns the collection child components. The property **Tag** exposes the **tag** field and is a read only property as well.

The constructor of the **GameState** class takes two parameters. The first is a **Game** parameter that is a reference to your game object required by game components. The second is a reference to the **GameStateManager** so it can be assigned to the **StateManager** field. The constructor then sets the **StateManager** field to the value passed in, initializes the **childComponents** field, and finally sets the **tag** field to be **this**, the current instance of the class.

The **Update** method just loops through all of the **GameComponents** in **childComponents** and if their **Enabled** property is true calls their **Update** methods. Similarly, the **Draw** method loops through all of the **GameComponents** in **childComponents**. It then checks if the game component is a **DrawableGameComponent**. If it is, it checks to see if it is visible. If it is visible, it then calls the **Draw** method of the components.

The other methods in this class are methods that aren't inherited from **DrawableGameComponent**. The first method, **StateChange**, is the event handler code changing game states. All active screens will subscribe to an event in the game state manager class. All states that are subscribed to the event will receive a message that they active screen was changed. In the **ScreenChange** method I call the **Show** method if the screen that triggered the event, from the **sender** parameter, is the **Tag** property of the current screen. Otherwise I call the **Hide** method. The **Show** method sets a screen enabled and visible. The **Hide** method sets a screen disabled and not visible.

In the **Show** method I set the **Visible** and **Enabled** properties of the **GameScreen** to true. I also loop through all of the **GameComponents** in **childComponents** and set them to enabled. I also check to see if the component is a **DrawableGameComponent** and set it to visible. The **Hide** method works in reverse.

The next class to add is the **GameStateManager** class. Right click the **XRpgLibrary** project, select **Add** and then **Class**. Name this new component **GameStateManager**. The code for that class follows next.

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

namespace MGRpgLibrary
{
    public class GameStateManager : GameComponent
    {
        #region Event Region

        public event EventHandler OnStateChange;

        #endregion

        #region Fields and Properties Region

        Stack<GameState> gameStates = new Stack<GameState>();
        const int startDrawOrder = 5000;
        const int drawOrderInc = 100;
        int drawOrder;
        public GameState CurrentState
        {
            get { return gameStates.Peek(); }
        }

        #endregion

        #region Constructor Region

        public GameStateManager(Game game)
        : base(game)
        {
            drawOrder = startDrawOrder;
        }

        #endregion

        #region MG Method Region

        public override void Initialize()
        {
            base.Initialize();
        }

        public override void Update(GameTime gameTime)
        {
            base.Update(gameTime);
        }

        #endregion

        #region Methods Region
        public void PopState()
        {
```

```
            if (gameStates.Count > 0)
            {
                RemoveState();
                drawOrder -= drawOrderInc;
                if (OnStateChange != null)
                    OnStateChange(this, null);
            }
        }

        private void RemoveState()
        {
            GameState State = gameStates.Peek();
            OnStateChange -= State.StateChange;
            Game.Components.Remove(State);
            gameStates.Pop();
        }

        public void PushState(GameState newState)
        {
            drawOrder += drawOrderInc;
            newState.DrawOrder = drawOrder;
            AddState(newState);

            if (OnStateChange != null)
                OnStateChange(this, null);
        }

        private void AddState(GameState newState)
        {
            gameStates.Push(newState);
            Game.Components.Add(newState);
            OnStateChange += newState.StateChange;
        }

        public void ChangeState(GameState newState)
        {
            while (gameStates.Count > 0)
                RemoveState();

            newState.DrawOrder = startDrawOrder;
            drawOrder = startDrawOrder;
            AddState(newState);

            if (OnStateChange != null)
                OnStateChange(this, null);
        }

        #endregion
    }
}
```

In the **Event** region is the code for the event that I created that will be triggered when there is a change in state, or screens. You saw the handler for the event in the **GameState** class, **StateChange**. To manage the states I used a generic **Stack<GameState>**. A stack, in computer science, is a last-in-first-out data structure. The usual analogy when describing a stack is to think of a stack of plates. To add a plate you place, or push, it on top of the stack. The plate on top of the stack is removed, or popped, off the top.

There are three integer fields: **startDrawOrder**, **drawOrderInc**, and **drawOrder**. These fields are used in determining the order that game screens are drawn. **DrawableGameComponent** have a property called **DrawOrder**. Components will be drawn in ascending order, according to their **DrawOrder** property. The component with the lowest draw order will be drawn first. The component with the highest **DrawOrder** property will be drawn last. I chose 5000 as a good starting point. When a screen is added onto the stack, I will set its **DrawOrder** property higher than the last screen. The **drawOrderInc** field is how much to increase or decrease when a screen is added or removed. **drawOrder** holds the current value of the screen on top. There is a public property, **CurrentScreen**, that returns which screen is on top of the stack.

The constructor of this class just sets the **drawOrder** field to the **startDrawOrder** field. By choosing the magic number, 5000, and 100 for the amount to increase or decrease, there is room for a lot screens. Far more than you will probably have on the stack at once.

In the **Methods** region there are three public and two private methods. The public methods are **PopState**, **PushState**, and **ChangeState**. The private methods are **RemoveState** and **AddState**. **PopState** is the method to call if you have a screen on top of the stack that you want to remove and go back to the previous screen. A good example of this is you open an options menu from the main menu.

You want to go back to the main menu from the options menu so you just pop the options menu off the stack. **PushState** is the method to call if you want to move to another state and keep the previous state.

From above, you would push the options screen on the stack so where you are done you can return to the previous state. The last public method, **ChangeState**, is called when you wish to remove all other states from the stack.

The **PopState** method checks to make sure there is a game state to pop off by checking the **Count** property of the **gameStates** stack. It calls the **RemoveState** method that removes the state that is on the top of the stack. It then decrements the **drawOrder** field so that when the next screen is added to the stack of screens it will be drawn appropriately. It then checks if **OnStateChange** is not **null**. What this means is it checks to see if the **OnStateChange** event is subscribed to. If the event is not subscribed to the event handler code should not be executed. It then calls the event handler code passing itself as the sender and null for the event arguments.

The **RemoveState** method first gets which state is on top of the stack. It then unsubscribes the state from the subscribers to the **OnStateChange** event. It then removes the screen from the components in the game. It then pops the state off the stack.

The **PushState** method takes as a parameter the state to be placed on the top of the stack. It first increases the **drawOrder** field and set it to the **DrawOrder** property of the component so it will have the highest **DrawOrder** property. It then calls the **AddState** method to add the screen to the stack. It then checks to make sure the **OnStateChange** event is subscribed to before calling the event handler code.

The **AddState** method pushes the new screen on the top of the stack. It adds it to the list of components of the game. Finally it subscribes the new state to the **OnStateChange** event.

The first thing the **ChangeState** method does is remove all of the screens from the stack. It then sets the **DrawOrder** property of the new screen to the **startDrawOrder** value and **drawOrder** to the same value. It then calls the **AddScreen** method passing in the screen to be changed to. It then will call the **OnStateChange** event handler if it is subscribed to.

Now you can add a **GameStateManager** to the game. Go to the code for the **Game1** class in your game. You will want to add a field, below the **SpriteBatch** field, for the **GameStateManager**. You will also want to initialize it in the constructor of the game and add it to the list of components. Add this field and change the constructor to the following.

```
GameStateManager _gameStateManager;

public Game1()
{
    _graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";
    IsMouseVisible = true;

    Components.Add(new InputHandler(this));

    _gameStateManager = new GameStateManager(this);
    Components.Add(_gameStateManager);
}
```

That is a lot of code but it doesn't really do anything. You can build and run but all you will get is the blue screen as if you hadn't added any code. Right click the **EyesOfTheDragon** project in the solution explorer, select **Add** and then **New Folder**. Name this new folder **GameScreens**. To this folder you are going to add in two game states. The first state is a base state that will have fields common to all states.

The second is a title screen that will display an image. Special thanks to, **Tuckbone**, a reader of my tutorials for creating the graphic. You can download the title image from here.

To add assets, or content, to your game you use the MonoGame Pipeline Tool. You do that by double clicking the Content.mgcb file under the Content folder. If it does not open for you right click it and select Open With. Scroll down to the MonoGame Pipeline entry and select it. If it is not set as default click the Set As Default button. Click the Open button.

It is a good idea to organize your content. Right click the **Content** node in the MonoGame Pipeline Tool, select **Add** and then **New Folder**. Name this new folder **Backgrounds**. After you have downloaded and extracted the image you will want to add it to the **Backgrounds** folder. Right click the **Backgrounds** folder, select **Add** and then **Existing Item**. You will want to navigate to where you downloaded and extracted the image and add the **titlescreen.png** entry.

When you are making game graphics it is a good idea to use an image format that have lossless compression, unlike JPEG for example. PNG, BMP, TGA, and GIF are all good formats for that. PNG, GIF

and TGA are great because they support transparency. You can emulate transparency in BMP by using magenta #FF00FF or specifying the transparency colour in the MonoGame Pipleline Tool.
Now, going back to the states. Right click the **GameScreens** folder in the **EyesOfTheDragon** project, select **Add** and then **Class**. Name this new class **BaseGameState**. The code for that class follows next.

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

namespace EyesOfTheDragon.GameScreens
{
    public abstract partial class BaseGameState : GameState
    {
        #region Fields region
        protected Game1 GameRef;
        #endregion

        #region Properties region
        #endregion

        #region Constructor Region

        public BaseGameState(Game game, GameStateManager manager)
            : base(game, manager)
        {
            GameRef = (Game1)game;
        }

        #endregion
    }
}
```

The **BaseGameState** class is a very basic class. It inherits from the **GameState** class so it can be used in the **GameStateManager**. The lone field in the class is **GameRef** that is a reference to our game object. The constructor takes two parameter. The first is the current game object required by game components and the second is a **GameStateManager** object required by the **GameState** class. The constructor just sets the **GameRef** field by casting the **Game** parameter to the **Game1** class.

The next screen is going to be a title screen. You are going to get a few errors for the screen because I made quite a few changes to the **Game1** class to get this screen to work. After I've shown you the code for the screen and explained it I will give you the complete code for the **Game1** class. Right click the **GameScreens** folder in the **EyesOfTheDragon** project, select **Add** and then **Class**. Name this new class **TitleScreen**.

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

```
using MGRpgLibrary;

namespace EyesOfTheDragon.GameScreens
{
    public class TitleScreen : BaseGameState
    {
        #region Field region
        Texture2D backgroundImage;
        #endregion

        #region Constructor region
        public TitleScreen(Game game, GameStateManager manager)
        : base(game, manager)
        {
        }

        #endregion

        #region MG Method region
        protected override void LoadContent()
        {
            ContentManager Content = GameRef.Content;
            backgroundImage = Content.Load<Texture2D>(@"Backgrounds\titlescreen");
            base.LoadContent();
        }

        public override void Update(GameTime gameTime)
        {
            base.Update(gameTime);
        }

        public override void Draw(GameTime gameTime)
        {
            GameRef.SpriteBatch.Begin();
            base.Draw(gameTime);

            GameRef.SpriteBatch.Draw(
                backgroundImage,
                GameRef.ScreenRectangle,
                Color.White);

            GameRef.SpriteBatch.End();
        }

        #endregion
    }
}
```

There are using statements to bring classes from the MonGame Framework, MonoGame Framework Graphics, and MonoGame Framework Content name spaces into scope. The Graphics ones are for drawing and the Content ones are for loading in content. There is also a using statement to bring our **XRpgLibrary** name space into scope.

The class inherits from the **BaseGameState** class. There is just one field in the class at this time, a **Texture2D**, for our background image. The constructor takes two parameters. They are the same as any other game states, the current game object and a reference to the **GameStateManager**.

In the **LoadContent** method I get the **ContentManager** that is associated with our game using the **Content** property of **GameRef**. I then use the **Load** method passing in the location of the background of our image.

In the **Draw** method I draw the image. There is first a call to **Begin** on **SpriteBatch**. 2D images, often called sprites, are drawn between calls to **Begin** and **End** of a **SpriteBatch** object. You will have errors related to the **SpriteBatch** because I haven't made that change to the **Game1** class yet. After the call to **Begin** is the call to **base.Draw** to draw any child components. I then call the **Draw** method to draw the background image passing in the image to be drawn, the destination of the image as a rectangle and the tint colour. You can tint objects using this tint colour. Using white means there is no tinting at all. Finally, there is the call to **End**.

There have been a number of changes to the **Game1** class. I will give you the new code for the entire class and then go over the changes that I made. Here's the code.

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using MGRpgLibrary;
using EyesOfTheDragon.GameScreens;

namespace EyesOfTheDragon
{
    public class Game1 : Game
    {
        private GraphicsDeviceManager _graphics;
        private SpriteBatch _spriteBatch;
        private GameStateManager _gameStateManager;

        public const int ScreenWidth = 1280;
        public const int ScreenHeight = 720;

        public readonly Rectangle ScreenRectangle = new Rectangle(
            0,
            0,
            ScreenWidth,
            ScreenHeight);

        public SpriteBatch SpriteBatch { get { return _spriteBatch; } }
        public TitleScreen TitleScreen { get; private set; }

        public Game1()
        {
            _graphics = new GraphicsDeviceManager(this);
            Content.RootDirectory = "Content";
            IsMouseVisible = true;

            Components.Add(new InputHandler(this));

            _gameStateManager = new GameStateManager(this);
            Components.Add(_gameStateManager);

            TitleScreen = new TitleScreen(this, _gameStateManager);

            _gameStateManager.ChangeState(TitleScreen);
        }
```

```
        protected override void Initialize()
        {
            // TODO: Add your initialization logic here
            _graphics.PreferredBackBufferWidth = ScreenWidth;
            _graphics.PreferredBackBufferHeight = ScreenHeight;
            _graphics.ApplyChanges();

            base.Initialize();
        }

        protected override void LoadContent()
        {
            _spriteBatch = new SpriteBatch(GraphicsDevice);

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

        protected override void Update(GameTime gameTime)
        {
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed ||
Keyboard.GetState().IsKeyDown(Keys.Escape))
                Exit();

            // TODO: Add your update logic here

            base.Update(gameTime);
        }

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

            // TODO: Add your drawing code here

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

There are two constants **ScreenWidth** and **ScreenHeight** that are the width and height of our screen, or window. There is a readonly **Rectangle** that has the dimensions of our window. I added a get only property to return our **SpriteBatch** object. There is another property that has a public getter and a private setter for our **TitleScreen**. The constructor now creates the **TitleScreen** instance and calls the **ChangeState** method passing in the **TitleScreen** property. The Initialize method sets the **PreferredBackbufferWidth** to the width of our screen and the **PreferredBackbufferHeight** to the height of our screen. It then calls **ApplyChanges** to have the changes take effect.

I think this is enough for this tutorial. I'd like to try and keep them to a reasonable length so that you don't have too much to digest at once. I encourage you to visit the news page of my site, https://cynthiamcmahon.ca/blog/, for the latest news on my tutorials.

Good luck in your game programming adventures!
Cynthia