Eyes of the Dragon Tutorials 4.0

Part 1

Getting Started

I'm writing these tutorials for the MonoGame 3.8.1 framework using Visual Studio 2022. 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 4.0 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 and the past version of MonoGame 3.8. 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 not to make those same mistakes. Also, I am going to make the game cross-platform. In my previous tutorials, they were focused on Windows only. I want to open things up for macOS, Linux, Andriod, iOS, and potentially any platform the MonoGame supports.

To get started, we want to create a new MonoGame project. Open Visual Studio 2022. 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 fundamental 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. **HiDef** also supports textures up to 8192 pixels by 8192 pixels.

I want to add two class libraries to this project. The first will be a shared 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 will be used for custom content that is not part of the Content Pipeline. 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 Game Library**. Choose that tile on the left and click the Next button. Name this new project **RpgLibrary**. Right-click your solution in the solution explorer again, select **Add** and then **New Project** again. This time enter **MonoGame Shared**. Choose the **MonoGame Shared Library Project** tile and click **Next**. Name this new project **SharedProject**. In the **RpgLibrary** 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 **SharedProject** 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 them about each other, so Visual Studio knows to include them when it builds the game. and you can use their members. You do that by adding references to the libraries of your game and libraries. 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 **RpgLibrary** library. Select the check box beside it. Now, expand the Shared Projects tab. Select the check box beside the **SharedProject** folder. There is one last thing to do.

With that done, it is time to actually write some code! 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 **SharedProject** to manage all of the input in the game in one location. For this tutorial, I'm only going to handle keyboard and mouse input. In future tutorials, I will add support for gamepads and touch screens. Right-click the **SharedProject** project in the solution explorer, select **Add** and then **Class**. Name this new class **Xin**. The name comes from my old, old, old XNA game programming days. It was (X)NA (IN)PUT. I'm a firm believer in showing new code first so you can read it over before explaining things. The code for the **Xin** class follows next.

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

namespace SharedProject
{
    public enum MouseButtons { Left, Right };

    public class Xin : GameComponent
    {
        private static KeyboardState keyboardState;
        private static KeyboardState lastKeyboardState;
        private static MouseState mouseState;
        private static MouseState lastMouseState;

        public static KeyboardState KeyboardState { get { return keyboardState; } }
        public static MouseState MouseState { get { return mouseState; } }

        public static KeyboardState LastKeyboardState { get { return lastKeyboard-
State; } }
        public static MouseState LastMouseState { get { return lastMouseState; } }

        public static Point MouseAsPoint
        {
            get { return new Point(MouseState.X, MouseState.Y); }
        }

        public static Point LastMouseAsPoint
```

```
{
    get { return new Point(LastMouseState.X, LastMouseState.Y); }
}

public Xin(Game game) : base(game)
{
    keyboardState = Keyboard.GetState();
    mouseState = Mouse.GetState();
}

public override void Update(GameTime gameTime)
{
    lastKeyboardState = keyboardState;
    lastMouseState = mouseState;

    keyboardState = Keyboard.GetState();
    mouseState = Mouse.GetState();

    base.Update(gameTime);
}

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

public static bool WasKeyDown(Keys key)
{
    return lastKeyboardState.IsKeyDown(key);
}

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

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

public static bool IsMouseDown(MouseButtons button)
{
    return button switch
    {
        MouseButtons.Left => mouseState.LeftButton == ButtonState.Pressed,
        MouseButtons.Right => mouseState.RightButton == ButtonState.Pressed,
        _ => false,
    };
}

public static bool WasMouseDown(MouseButtons button)
{
    return button switch
    {
        MouseButtons.Left => lastMouseState.LeftButton == ButtonState.Pressed,
        MouseButtons.Right => lastMouseState.RightButton == ButtonState.Pressed,
        _ => false,
    };
}
```

```
        public static bool WasMousePressed(MouseButtons button)
        {
            return button switch
            {
                MouseButtons.Left => mouseState.LeftButton == ButtonState.Pressed &&
lastMouseState.LeftButton == ButtonState.Released,
                MouseButtons.Right => mouseState.RightButton == ButtonState.Pressed &&
lastMouseState.RightButton == ButtonState.Released,
                _ => false,
            };
        }

        public static bool WasMouseReleased(MouseButtons button)
        {
            return button switch
            {
                MouseButtons.Left => mouseState.LeftButton == ButtonState.Released &&
lastMouseState.LeftButton == ButtonState.Pressed,
                MouseButtons.Right => mouseState.RightButton == ButtonState.Released &&
lastMouseState.RightButton == ButtonState.Pressed,
                _ => false,
            };
        }

        public static List<Keys> KeysPressed()
        {
            List<Keys> keys = new();

            Keys[] current = keyboardState.GetPressedKeys();
            Keys[] last = lastKeyboardState.GetPressedKeys();

            foreach (Keys key in current)
            {
                if (!last.Contains(key))
                {
                    keys.Add(key);
                }
            }

            return keys;
        }

        public static List<Keys> KeysReleased()
        {
            List<Keys> keys = new();

            Keys[] current = keyboardState.GetPressedKeys();
            Keys[] last = lastKeyboardState.GetPressedKeys();

            foreach (Keys key in current)
            {
                if (last.Contains(key))
                {
                    keys.Add(key);
                }
            }

            return keys;
        }
```

```
    }
}
```

So, mouse input in MonoGame is done differently than other input devices. To try and make it more consistent, I included an enumeration of the buttons we're going to check for. For now, I'm only including the left and right buttons.

The Xin class inherits from the GameComponent class, so it can be added to the list of components in the game and have its Update method called automatically. The fields, properties, and methods are all static. This is so they can be referenced using the class name. Part of why I named the class Xin is that it was short.

There are two KeyboadState fields in the class. The first, keyboardState, holds the current state of the keyboard. The second, lastKeyboardState, holds the state of the keyboard in the last frame of the game. There are similar fields for the mouse. Next, there are four properties that are read-only to expose the values of the fields.

The constructor initializes the current keyboard and mouse fields. It takes as a parameter a Game object. That is required by the GameComponent class. The Update method sets the value of the lastKeyboardState field to the value of the keyboardState field and the lastMouseState value to the mouseState value. Next, it sets the keyboardState and mouseState values to the current state of the keyboard and mouse, respectively. What this does is allow us to compare the previous frame of the game to the current frame of the key. You will see why that is important shortly.

Next up are some methods to test various keyboard events. The first check is to see if a key on the keyboard is currently down. There is also a method that returns if a key was down in the previous frame. The next test is if a key was pressed. That means it is down in this frame and up in the previous frame. This test is useful when you want to do something the exact moment a key is pressed, such as firing a weapon. The next test is if a key has been released. That happens if a button is up in this frame and down in the previous frame. This is useful in things such as clicking buttons.

As I mentioned earlier, mouse clicks do not happen the same way as keyboard events. There are no down methods. Instead, there are properties that return if the button is pressed or released. In order to check if a button is pressed, the method receives a MouseButton argument. Inside the method, there is a switch expression that returns if the button passed in is down. Similarly, there is a method that checks if the mouse was down in the previous frame. Similar to the keyboard methods, there are methods that check to see if the mouse has been released or pressed.

I included two other useful methods. The first returns a list of keys that were released since the last frame and a list of keys that were pressed since the last frame. This is done with a bit of aid from **LINQ**. It stands for Language Integrated Query. Among other things, it has methods for iterating and querying lists of data. In this case, we will be using **Contains**, which checks if something exists in a collection. There is a collection of the **KeyboadState** class called **GetPressedKeys** that returns an array of keys pressed. The only difference in the methods is one checks for false, and the other checks for true. So, the first step is to get the pressed keys in the current frame and the previous frame. Loop over the keys in the current frame. Using **LINQ**, check to see if the keys or is not contained in the last frame. That key is added to a list of pressed or released keys. The list is then returned.

A common mistake made when creating a large game is not handling the game state properly from the start. If you try and go back and add it in later, it is a significant 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 **SharedProject** 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 Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

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

        private readonly List<GameComponent> childComponents;

        protected SpriteBatch SpriteBatch { get; private set; }

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

        readonly GameState tag;

        public GameState Tag
        {
            get { return tag; }
        }

        protected GameStateManager StateManager;

        #endregion

        #region Constructor Region

        public GameState(Game game)
            : base(game)
        {
            StateManager = Game.Services.GetService<GameStateManager>();
            childComponents = new List<GameComponent>();
            SpriteBatch = Game.Services.GetService<SpriteBatch>();
            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 child)
            {
                child.Visible = true;
            }
        }
    }

    protected virtual void Hide()
    {
        Visible = false;
        Enabled = false;

        foreach (GameComponent component in childComponents)
        {
            component.Enabled = false;

            if (component is DrawableGameComponent child)
            {
```

```
                    child.Visible = false;
                }
            }
        }

        #endregion
    }
}
```

When you create **Game Components,** you inherit either from the **GameComponent** class or the **DrawableGameComponent** class. Game states will do rendering, 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. Finally, there is a **SpriteBatch** that we will use in rendering.

The constructor of the **GameState** class takes one parameter. A **Game** parameter that is a reference to your game object required by game components. The constructor then sets the **StateManager** by retrieving it as a service, more on services in a bit. It initializes the **childComponents** field, and finally sets the **tag** field to be **this**, the current instance of the class. Also, it retrieves the **SpriteBatch** for the game from the list of services.

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 are 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 state 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.

About services. Services can be registered using the AddService method of the Services property of the game. They can then be retrieved using the GetService method in any class that needs to use it. It is just a powerful way to have access to classes with out having to pass them as parameters or using protected or static items. In these tutorials, I will be using mostly services instead of passing

references. I find it cleaner.

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

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

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

        private readonly List<GameComponent> childComponents;

        protected SpriteBatch SpriteBatch { get; set; }

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

        readonly GameState tag;

        public GameState Tag
        {
            get { return tag; }
        }

        protected GameStateManager StateManager;

        #endregion

        #region Constructor Region

        public GameState(Game game)
        : base(game)
        {
            childComponents = new List<GameComponent>();
            tag = this;
        }

        #endregion

        #region MG Drawable Game Component Methods
        public override void Initialize()
        {
            base.Initialize();
        }

        protected override void LoadContent()
        {
            base.LoadContent();
```

```
        SpriteBatch = Game.Services.GetService<SpriteBatch>();
    }

    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)
    {
        StateManager ??= Game.Services.GetService<GameStateManager>();

        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 child)
            {
                child.Visible = true;
            }
        }
    }
```

```
        protected virtual void Hide()
        {
            Visible = false;
            Enabled = false;

            foreach (GameComponent component in childComponents)
            {
                component.Enabled = false;

                if (component is DrawableGameComponent child)
                {
                    child.Visible = false;
                }
            }
        }

        #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 to 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 numbers 5000 and 100 for the amount to increase or decrease, there is room for a lot of 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. An excellent example of this are 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 that when 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 following 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 that 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 current 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 sets 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.

I'm going to add one more class to the **SharedProject** in this tutorial. That is a class that holds settings for the game. It will hold two constants and three properties. The constants will be the base resolution that we will use in rendering. The one property returns a rectangle that defines the base resolution that we will be rendering. The other two properties will be the target resolution when rendering. To begin with, they will be the same. In a future tutorial, I will describe how the scaling between the base resolution and the target resolution will happen. Right-click the **SharedProject** project, select **Add** and then **Class**. Name this new class **Settings**. Here is the code for that trivial class.

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

namespace SharedProject
{
```

```
    public class Settings
    {
        public const int BaseWidth = 1280;
        public const int BaseHeight = 720;

        public static Rectangle BaseRectangle { get { return new(0, 0, BaseWidth, Base-
Height); } }
        public static int TargetWidth { get; set; } = BaseWidth;
        public static int TargetHeight { get; set; } = BaseHeight;
    }
}
```

Actually, one last thing before I leave the **SharedProject**. I'm going to add a game state. In the past, I used to add another class that would inherit from **GameState** inside of all of the different games. However, I've found that having the game states inside of the shared project allowed for easier porting between platforms. You write as much as you can in the libraries, only adding platform-specific code in those projects. Right-click the **SharedProject** project, select **Add** and then **Folder**. Name this new folder **GameScreens.** Now, right-click the **GameScreens** folder, select **Add** and then **Class**. Name this new class **TitleState.** Here is the code for that class.

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

namespace SharedProject.GameScreens
{
    public interface ITitleState
    {
        GameState Tag { get; }
    }

    public class TitleState : GameState, ITitleState
    {
        #region Field region

        Texture2D backgroundImage;

        readonly GameState tag;

        public GameState Tag
        {
            get { return tag; }
        }

        #endregion

        #region Constructor region

        public TitleState(Game game)
            : base(game)
        {
            Game.Services.AddService<ITitleState>(this);
            SpriteBatch = Game.Services.GetService<SpriteBatch>();
        }

        #endregion
```

```
        #region XNA Method region

        protected override void LoadContent()
        {
            base.LoadContent();

            ContentManager Content = Game.Content;
            backgroundImage = Content.Load<Texture2D>(@"Backgrounds\titlescreen");
        }

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

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

            base.Draw(gameTime);

            SpriteBatch.Draw(
                backgroundImage,
                Settings.BaseRectangle,
                Color.White);

            SpriteBatch.End();
        }

        protected override void Show()
        {
            base.Show();

            LoadContent();
        }
        #endregion
    }
}
```

First, there is an interface that will be used to register the state as a service and retrieve it. It allows for a form of multiple inheritance. Because a class can only inherit from one class but can implement multiple inheritance. This is a kind of composition. So, say you had an enemy that was your base class, and you wanted to have a fast enemy, so you inherit the fast enemy from the enemy class. Now, you want a poison enemy. So, you just inherit the class from the enemy base class. Now, you want a fast and a poison enemy. How are you going to accomplish that? You can't inherit from fast and poison. Well, if enemy, fast, and poison were interfaces, you could implement all three interfaces. The other thing about interfaces is that you can use them as members like you would a class.

For the screen, I added an image that was created by one of my readers, **TuckBone**. You can download the title image from here. To add assets, or content, to your game, you use the MGCB Editor. 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 MGCB Editor 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 has lossless compression, unlike JPEG, for example. PNG, BMP, TGA, and GIF are all suitable 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 MGCB Editor.

There are using statements to bring classes from the MonoGame Framework, MonoGame Framework Graphics, and MonoGame Framework Content namespaces into scope. The Graphics are for drawing, and the Content ones are for loading in content.

The class inherits from the **GameState** class. There is just one field in the class at this time, a **Texture2D** for our background image. The constructor takes one parameter. It is the same as any other game state.

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 are 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 that there is no tinting at all. Finally, there is the call to **End**.

The other thing that I did was override the Show method. I have found that the LoadContent method of the first state is often ignored. So, I call it explicitly.

That brings us to the **Game1** class. I will give you the new code for the entire class and then go over the changes that I made to the boilerplate code. Here's the code.

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using SharedProject;
using SharedProject.GameScreens;

namespace EyesOfTheDragon
{
    public class Game1 : Game
    {
        private readonly TitleState _titleState;
        private readonly GraphicsDeviceManager _graphics;
```

```
        private SpriteBatch _spriteBatch;
        public GameStateManager GameStateManager { get; private set; }

        public Game1()
        {
            _graphics = new GraphicsDeviceManager(this);

            Content.RootDirectory = "Content";
            IsMouseVisible = true;

            _graphics.PreferredBackBufferWidth = Settings.BaseWidth;
            _graphics.PreferredBackBufferHeight = Settings.BaseHeight;
            _graphics.ApplyChanges();

            Components.Add(new Xin(this));

            GameStateManager = new GameStateManager(this);
            Components.Add(GameStateManager);
            Services.AddService(typeof(GameStateManager), GameStateManager);

            _titleState = new(this);
        }

        protected override void Initialize()
        {
            // TODO: Add your initialization logic here

            base.Initialize();
        }

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

            Services.AddService(typeof(SpriteBatch), _spriteBatch);
            GameStateManager.PushState(_titleState);
        }

        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 is readonly field for our title state. Reassigning the **GraphicsDeviceManager** is a bad idea, so I made it readonly as well. There is also a property for the **GameStateManager**.

After creating the graphics device manager, I set the back buffer width to our base width and the back buffer height to the base height. What is a back buffer, you may ask? And why should I care? Drawing to the screen takes time. The more items you draw to the screen, the greater the chance that there will be tearing because the refresh rate of the monitor is different from the refresh rate of your game. All it takes is the monitor refreshing while you're in the middle of drawing an object. How do we solve this? Why with a back buffer, of course. You do all of your rending off-screen. When you have finished rendering, you flip the back buffer to the screen. After setting the back buffer parameters, I call the ApplyChanges method to, well, apply the changes. Now you will see the beauty of game components in action. I just create a new instance of Xin and forget about it. MonoGame takes care of it from there. We don't have to think about it. After Xin, I create an instance of the game state manager. I then add it the components as I did with Xin. I then register it as a service. I then create the instance of the title state. In the LoadContent method, after creating the SpriteBatch, I register it as a service. Finally, I push the title states onto the stack.

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 with your game programming adventures!
Cynthia