Eyes of the Dragon Tutorials
Part 2
More Game Components

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 two in my series of Eyes of the Dragon tutorials. I'm going to be adding in a fewvmore core components in this tutorial that you will be using through out the series. Go a head and loadvup your solution from last time.

The first thing I'm going to do is to update the **InputHandler** class to include game pad input and mouse input. I renamed the field and property regions from the last tutorial to include Keyboard. I will give you the entire code for the class and explain the game pad and mouse specific code.

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

namespace MGRpgLibrary
{
    public enum MouseButton { Left, Middle, Right }

    public class InputHandler : Microsoft.Xna.Framework.GameComponent
    {
        #region Keyboard Field Region

        static KeyboardState keyboardState;
        static KeyboardState lastKeyboardState;

        #endregion

        #region Mouse Region

        static MouseState mouseState;
        static MouseState lastMouseState;

        #endregion

        #region Game Pad Field Region

        static GamePadState[] gamePadStates;
        static GamePadState[] lastGamePadStates;

        #endregion

        #region Keyboard Property Region

        public static KeyboardState KeyboardState
        {
            get { return keyboardState; }
        }
```

```
public static KeyboardState LastKeyboardState
{
    get { return lastKeyboardState; }
}

#endregion

#region Mouse Property Region

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

public static MouseState LastMouseState
{
    get { return lastMouseState; }
}

#endregion

#region Game Pad Property Region

public static GamePadState[] GamePadStates
{
    get { return gamePadStates; }
}
public static GamePadState[] LastGamePadStates
{
    get { return lastGamePadStates; }
}

#endregion

#region Constructor Region

public InputHandler(Game game)
    : base(game)
{
    keyboardState = Keyboard.GetState();
    gamePadStates = new GamePadState[Enum.GetValues(typeof(PlayerIndex)).Length];
    mouseState = Mouse.GetState();

    foreach (PlayerIndex index in Enum.GetValues(typeof(PlayerIndex)))
        gamePadStates[(int)index] = GamePad.GetState(index);
}

#endregion

#region MonoGame methods

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

public override void Update(GameTime gameTime)
{
    lastKeyboardState = keyboardState;
    keyboardState = Keyboard.GetState();
```

```
    lastMouseState = mouseState;
    mouseState = Mouse.GetState();

    lastGamePadStates = (GamePadState[])gamePadStates.Clone();

    foreach (PlayerIndex index in Enum.GetValues(typeof(PlayerIndex)))
        gamePadStates[(int)index] = GamePad.GetState(index);

    base.Update(gameTime);
}

#endregion

#region General Method Region

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

#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

#region Mouse Region
public static Point MouseAsPoint
{
    get { return new Point(mouseState.X, mouseState.Y); }
}

public static Vector2 MouseAsVector2
{
    get { return new Vector2(mouseState.X, mouseState.Y); }
}

public static Point LastMouseAsPoint
{
    get { return new Point(lastMouseState.X, lastMouseState.Y); }
}
```

```
public static Vector2 LastMouseAsVector2
{
    get { return new Vector2(lastMouseState.X, lastMouseState.Y); }
}

public static bool CheckMousePress(MouseButton button)
{
    bool result = false;

    switch (button)
    {
        case MouseButton.Left:
            result = mouseState.LeftButton == ButtonState.Pressed &&
                lastMouseState.LeftButton == ButtonState.Released;
            break;
        case MouseButton.Right:
            result = mouseState.RightButton == ButtonState.Pressed &&
                lastMouseState.RightButton == ButtonState.Released;
            break;
        case MouseButton.Middle:
            result = mouseState.MiddleButton == ButtonState.Pressed &&
                lastMouseState.MiddleButton == ButtonState.Released;
            break;
    }
    return result;
}

public static bool CheckMouseReleased(MouseButton button)
{
    bool result = false;

    switch (button)
    {
        case MouseButton.Left:
            result = mouseState.LeftButton == ButtonState.Released &&
                lastMouseState.LeftButton == ButtonState.Pressed;
            break;
        case MouseButton.Right:
            result = mouseState.RightButton == ButtonState.Released &&
                lastMouseState.RightButton == ButtonState.Pressed;
            break;
        case MouseButton.Middle:
            result = mouseState.MiddleButton == ButtonState.Released &&
                lastMouseState.MiddleButton == ButtonState.Pressed;
            break;
    }

    return result;
}

public static bool IsMouseDown(MouseButton button)
{
    bool result = false;

    switch (button)
    {
        case MouseButton.Left:
            result = mouseState.LeftButton == ButtonState.Pressed;
            break;
```

```
                case MouseButton.Right:
                    result = mouseState.RightButton == ButtonState.Pressed;
                    break;
                case MouseButton.Middle:
                    result = mouseState.MiddleButton == ButtonState.Pressed;
                    break;
            }

            return result;
        }

        public static bool IsMouseUp(MouseButton button)
        {
            bool result = false;

            switch (button)
            {
                case MouseButton.Left:
                    result = mouseState.LeftButton == ButtonState.Released;
                    break;
                case MouseButton.Right:
                    result = mouseState.RightButton == ButtonState.Released;
                    break;
                case MouseButton.Middle:
                    result = mouseState.MiddleButton == ButtonState.Released;
                    break;
            }

            return result;
        }

        #endregion

        #region Game Pad Region

        public static bool ButtonReleased(Buttons button, PlayerIndex index)
        {
            return gamePadStates[(int)index].IsButtonUp(button) &&
            lastGamePadStates[(int)index].IsButtonDown(button);
        }

        public static bool ButtonPressed(Buttons button, PlayerIndex index)
        {
            return gamePadStates[(int)index].IsButtonDown(button) &&
            lastGamePadStates[(int)index].IsButtonUp(button);
        }

        public static bool ButtonDown(Buttons button, PlayerIndex index)
        {
            return gamePadStates[(int)index].IsButtonDown(button);
        }
        #endregion
    }
}
```

There could be four game pads connected to the Xbox 360 so I have arrays for the current state of eac game pad and the state of each game pad in the last frame of the game. The **gamePadStates** array holds the states of the game pads in the current frame and the **lastGamePadStates** array holds the states of the game pads in the last frame of the game. There are read only properties to expose these

fields. The nice thing about using array properties is you can get the entire array or the index you are interested in. So, if you wanted all of the game pad states you could get them or if you want a specific game pad you can get that as well.

The **mouseState** field holds the current state of the mouse and the **lastMouseState** field holds the state of the mouse in the last frame of the game. There are public properties to expose their values.

The constructor creates the **gamePadStates** array using a little trickery. The **Enum** class has a method **GetValues** that can return all of the values of an enumeration. The **PlayerIndex** enum has entries for each game pad, from **One** to **Four**. The **Length** property returns the length of an array. After creating the array I set the values of the array using a foreach loop and the return of the **GetValues** array to loop through each of the values in **PlayerIndex**. Inside the loop I call the **GetState** method of **GamePad** passing in **index**, the current **PlayerIndex**, casting it to an integer. The constructor also grabs the current state of the mouse.

In the **Update** method I first assign the **gamePadStates** to **lastGamePadStates** using the **Clone** method of the array class. The **Clone** method returns an exact copy of the array, you have to cast it to be the right type though. After assigning the last states I get the current states using the same process as I did in the constructor. I also set the **lastMouseState** field to the **mouseState** value. Then I set the **mouseState** field to the current mouse state.

The three new methods for game pads work the same way as the methods for the keyboard. The difference is that they take a **Buttons** parameter for the button you are interested in and a **PlayerIndex** parameter for the game pad. For the release you check if a button is up in this frame and down in the last. For a press you check if a button that was up in the last frame is down in this frame.

Mouse input works a little differently. It doesn't have a **ButtonUp** or **ButtonDown** method like the keyboard or game pad. Instead it has form properties the button in question that has a **Pressed** or **Released** state. So earlier in the code I added an enumeration **MouseButton** that has the three main mouse buttons: Left, Middle and Right. When checking for a button up or down you pass in one of those buttons. Then in a switch I assign a local variable **result** to if the button has been pressed or released. There are also properties for the position of the mouse as a **Point** and as a **Vector2**. There are also properties that do the same for the last mouse position.

What I'm going to add next are some GUI controls and a class to manage controls on the form. One thing about working with MonoGame is that you don't have all of the nice GUI controls that you're used to working with in Windows, or Mac or Linux. You have to make the controls yourself. It is a good idea to also have a class that manages all of the controls in a game state, or game screen. The first step is to create a base class for all controls. Right click the **MGRpgLibrary** project in the solution explorer, select **Add** and then **New Folder**. Name this new folder **Controls**. Now, right click the **Controls** folder, select **Add** and then **Class**. Name this new class **Control**. The code for that class follow next.

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

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

namespace MGRpgLibrary.Controls
{
    public abstract class Control
    {
        #region Field Region

        protected string name;
        protected string text;
        protected Vector2 size;
        protected Vector2 position;
        protected object value;
        protected bool hasFocus;
        protected bool enabled;
        protected bool visible;
        protected bool tabStop;
        protected SpriteFont spriteFont;
        protected Color color;
        protected string type;

        #endregion

        #region Event Region

        public event EventHandler Selected;

        #endregion
        #region Property Region

        public string Name
        {
            get { return name; }
            set { name = value; }
        }
        public string Text
        {
            get { return text; }
            set { text = value; }
        }
        public Vector2 Size
        {
            get { return size; }
            set { size = value; }
        }
        public Vector2 Position
        {
            get { return position; }
            set
            {
                position = value;
                position.Y = (int)position.Y;
            }
        }

        public object Value
        {
            get { return value; }
            set { this.value = value; }
```

```
}

public bool HasFocus
{
    get { return hasFocus; }
    set { hasFocus = value; }
}

public bool Enabled
{
    get { return enabled; }
    set { enabled = value; }
}

public bool Visible
{
    get { return visible; }
    set { visible = value; }
}

public bool TabStop
{
    get { return tabStop; }
    set { tabStop = value; }
}

public SpriteFont SpriteFont
{
    get { return spriteFont; }
    set { spriteFont = value; }
}

public Color Color
{
    get { return color; }
    set { color = value; }
}

public string Type
{
    get { return type; }
    set { type = value; }
}

#endregion

#region Constructor Region

public Control()
{
    Color = Color.White;
    Enabled = true;
    Visible = true;
    SpriteFont = ControlManager.SpriteFont;
}

#endregion

#region Abstract Methods
```

```
        public abstract void Update(GameTime gameTime);
        public abstract void Draw(SpriteBatch spriteBatch);
        public abstract void HandleInput(PlayerIndex playerIndex);

        #endregion

        #region Virtual Methods

        protected virtual void OnSelected(EventArgs e)
        {
            if (Selected != null)
            {
                Selected(this, e);
            }
        }
        #endregion
    }
}
```

This class has many protected fields that are common to controls. Public properties to expose these fields, so they can be overriden in inherited classes. There is a protected virtual method, **OnSelected**, that is used to fire the event **Selected** if it is subscribed to. There are also three abstract methods that any class that inherits from control has to implement. The one, **Update**, allows the control to be updated. The second, **Draw**, allows the control to be drawn. The last, **HandleInput**, is used to handle the input for the control. While these methods must be implemented they can be empty.

Controls have many things in common. I've picked a few of the more important ones. Controls have a **name** that will identify it. They have **text** that they may draw. They will have a **position** on the screen. They also have a **size**. The **value** field is a little more abstract. You can use this field to associate something with the control. Since the field is of type **object** you can assign any class to this field. One property of note is the **Position** property. I cast the **Y** component of the position to an integer. One thing I found with MonoGame is it doesn't like drawing text when the **Y** component of the position isn't an integer value.

Controls can also have focus, be visible or enabled, and be a tab stop. The last one is another peculiar field. You will be able to move through all of the controls on a screen and skip over ones that you may not want selected, like a label. The other field that a control will have is a type field that is a string. Controls also have a **SpriteFont** associated with them and a **Color**. For right now there is also an event associated with controls. This is the **Selected** event and will be triggered when the player selects the control.

The constructor of the **Control** class assigns the color of the control to white. It also sets its visible and enabled properties to true. It also sets the **SpriteFont** of the control to a static **SpriteFont** property of the **ControlManager**, a class that I will be designing next.

Right click the **Controls** folder in the **MGRpgLibrary** project, select **Add** and then **Class**. Name this new class **ControlManager**. This is the code for the **ControlManager** class.

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

```
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;

namespace MGRpgLibrary.Controls
{
    public class ControlManager : List<Control>
    {
        #region Fields and Properties

        int selectedControl = 0;
        static SpriteFont spriteFont;

        public static SpriteFont SpriteFont
        {
            get { return spriteFont; }
        }

        #endregion

        #region Constructors

        public ControlManager(SpriteFont spriteFont)
            : base()
        {
            ControlManager.spriteFont = spriteFont;
        }

        public ControlManager(SpriteFont spriteFont, int capacity)
            : base(capacity)
        {
            ControlManager.spriteFont = spriteFont;
        }

        public ControlManager(SpriteFont spriteFont, IEnumerable<Control> collection)
            : base(collection)
        {
            ControlManager.spriteFont = spriteFont;
        }

        #endregion

        #region Methods

        public void Update(GameTime gameTime, PlayerIndex playerIndex)
        {
            if (Count == 0)
                return;

            foreach (Control c in this)
            {
                if (c.Enabled)
                    c.Update(gameTime);
                if (c.HasFocus)
                    c.HandleInput(playerIndex);
            }

            if (InputHandler.ButtonPressed(Buttons.LeftThumbstickUp, playerIndex) ||
                InputHandler.ButtonPressed(Buttons.DPadUp, playerIndex) ||
```

```
            InputHandler.KeyPressed(Keys.Up))
            PreviousControl();

        if (InputHandler.ButtonPressed(Buttons.LeftThumbstickDown, playerIndex) ||
            InputHandler.ButtonPressed(Buttons.DPadDown, playerIndex) ||
            InputHandler.KeyPressed(Keys.Down))
            NextControl();
    }
    public void Draw(SpriteBatch spriteBatch)
    {
        foreach (Control c in this)
        {
            if (c.Visible)
                c.Draw(spriteBatch);
        }
    }

    public void NextControl()
    {
        if (Count == 0)
            return;

        int currentControl = selectedControl;
        this[selectedControl].HasFocus = false;

        do
        {
            selectedControl++;

            if (selectedControl == Count)
                selectedControl = 0;

            if (this[selectedControl].TabStop && this[selectedControl].Enabled)
                break;
        } while (currentControl != selectedControl);

        this[selectedControl].HasFocus = true;
    }
    public void PreviousControl()
    {
        if (Count == 0)
            return;

        int currentControl = selectedControl;
        this[selectedControl].HasFocus = false;

        do
        {
            selectedControl--;

            if (selectedControl < 0)
                selectedControl = Count - 1;
            if (this[selectedControl].TabStop && this[selectedControl].Enabled)
                break;
        } while (currentControl != selectedControl);

        this[selectedControl].HasFocus = true;
    }
    #endregion
}
```

}

There are a few using statements in this class to bring some of the MonoGame Framework classes into scope. The class inherits from **List<T>**. and will have all of the functionality of that class. This makes adding and removing controls to the manager exceedingly easy.

The **List<T>** class has three constructors so I have three constructors that will call the constructor of the **List<T>** class with the appropriate parameter. The constructors also take a **SpriteFont** parameter that all controls can use for their **SpriteFont** field. You can set the **SpriteFont** field of a control to use a different sprite font. This way when a control is first created it is easy to assign it a sprite font as the **spriteFont** field is static and can be accessed using a static property. The other field in the class is the **selectedIndex** field. This field holds which control is currently selected in the control manager.

There are a few public methods in this class. The **Update** method is used to update the controls and handle the input for the currently selected control. The **Draw** method is used to draw the controls on the screen. The other methods **NextControl** and **PreviousControl** are for moving from control to control.

The **Update** method takes as parameters the **GameTime** object from the game and the **PlayerIndex** of the game pad that you want to handle input from. The **Update** method first checks to see if there are controls on the screen. If there are none you can exit the method. There is next a foreach loop that loops through all of the controls on the screen. Inside the loop I check to see if the control is enabled using the **Enabled** property and if it is call the **Update** method of the control. Next there is a check to see if the control has focus and if it does calls the **HandleInput** method of the control. To move between controls you can use the Up and Down keys on the keyboard, the Up and Down directions on the direction pad, or the Up and Down directions of the left thumb stick. If any of the Up directions evaluate to true the **PreviousControl** method is called. Similarly, if any of the Down directions evaluate to true the **NextControl** method is called.

The **Draw** method takes the current **SpriteBatch** object as a parameter. There is a foreach loop that loops through all of the controls. Inside the loop there is a check to see if the control is visible. If it is, it calls the **Draw** method of the control.

The **NextControl** and **PreviousControl** methods aren't as robust as they should be. As I go I will update them so that they are more robust and to prevent exceptions from being thrown if something unexpected happens.

The **NextControl** method checks to make sure there are controls on the screen. If there are no controls there is nothing to do so it exits. I set a local variable to be the currently selected control. The reason is that I will loop through the controls and to know when to stop I needed a reference to the current control that had focus. I use the **this** keyword to reference the **List<Control>** part of the class and set the **HasFocus** method to false for the selected control.

There is next a do-while loop that loops through the controls until it finds a suitable control or reaches the starting control. The first step in moving the focus is to increase the **selectedControl** variable to move to the next control in the list. I check to see if the **selectedControl** field is equal to the **Count**

property. If it is you have reached that last control and I set the **selectedControl** field back to zero, the first control. The next if statement checks to see if the control referenced by **selectedControl** is a **TabStop** and is **Enabled**. If it is I break out of the loop. Finally I set the **HasFocus** property to true so the control is selected.

The **PreviousControl** method has the same format as the **NextControl** method but instead of incrementing the **selectedControl** field it decreases the **selectedControl** field. You first check to see if there are controls to work with. Save the value of the **selectedControl** field and set the **HasFocus** property of the control to false. In the do-while loop you first decrease the **selectedControl** field. If it is less than zero you set it to the number of controls minus one. If the control is a **TabStop** control and the control is **Enabled** I break out of the loop. Before exiting you set the **HasFocus** property to true.

With the control manager, it is now time to add in some specific controls. The control that I'm going to add in first is a simple **Label** control that can be used to draw text. The advantage of using the control manager and controls is you can group controls for easy access and you can loop through them using a foreach loop. Right click the **Controls** folder in the **MGRpgLibrary** project, select **Add** and then **Class**. Name this new class **Label**. This is the code for the **Label** class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
namespace MGRpgLibrary.Controls
{
    public class Label : Control
    {
        #region Constructor Region

        public Label()
        {
            tabStop = false;
        }

        #endregion

        #region Abstract Methods

        public override void Update(GameTime gameTime)
        {
        }

        public override void Draw(SpriteBatch spriteBatch)
        {
            spriteBatch.DrawString(SpriteFont, Text, Position, Color);
        }

        public override void HandleInput(PlayerIndex playerIndex)
        {
        }

        #endregion
    }
}
```

The class looks simple but coupled with the **ControlManager** it ends up being quite powerful. There are using statements to bring a couple of the XNA Framework name spaces into scope. The constructor of the **Label** class sets the **tabStop** field to **false** by default so **Labels** can't be selected by default. You can of course override this behavior if you need to. The **Update** and **HandleInput** methods do nothing at the moment but need to be implemented because we inherited from an abstract class. The **Draw** method calls the **DrawString** method of the **SpriteBatch** class to draw the text.

Another useful control to add is a **LinkLabel**. It is like a **Label** but I allow it to be selected. You could get away with adding this to the **Label** class but I like separating them into different controls. Right click the **Controls** folder in the **MGRpgLibrary** project, select **Add** and then **Class**. Name this new class **LinkLabel**. This is the code for the **LinkLabel** class.

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

namespace MGRpgLibrary.Controls
{
    public class LinkLabel : Control
    {
        #region Fields and Properties

        Color selectedColor = Color.Red;

        public Color SelectedColor
        {
            get { return selectedColor; }
            set { selectedColor = value; }
        }

        #endregion

        #region Constructor Region

        public LinkLabel()
        {
            TabStop = true;
            HasFocus = false;
            Position = Vector2.Zero;
        }

        #endregion

        #region Abstract Methods

        public override void Update(GameTime gameTime)
        {
        }

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

```
            if (hasFocus)
                spriteBatch.DrawString(SpriteFont, Text, Position, selectedColor);
            else
                spriteBatch.DrawString(SpriteFont, Text, Position, Color);
        }

        public override void HandleInput(PlayerIndex playerIndex)
        {
            if (!HasFocus)
                return;

            if (InputHandler.KeyReleased(Keys.Enter) ||
                InputHandler.ButtonReleased(Buttons.A, playerIndex))
                base.OnSelected(null);

            if (InputHandler.CheckMouseReleased(MouseButton.Left))
            {
                size = SpriteFont.MeasureString(Text);

                Rectangle r = new Rectangle(
                    (int)Position.X,
                    (int)Position.Y,
                    (int)size.X,
                    (int)size.Y);

                if (r.Contains(InputHandler.MouseAsPoint))
                    base.OnSelected(null);
            }
        }

        #endregion
    }
}
```

Again, the class is simplistic but combined with the control manager it can be quite powerful. There are a couple of using statements to bring some of the MonoGame Framework classes into scope. There is a new field and property associated with this control, **selectedColor** and **SelectedColor**. They are used in drawing the control in a different color if it is selected.

The constructor sets the **TabStop** property to true so it can receive focus. It also sets the **HasFocus** property to false initially so the control does not have focus and will not be updated or handle input. The **Draw** method draws the control in its regular colour if it is not select and in the selected colour if it does have focus. The **HandleInput** method returns if control does not have focus. If it does it checks to see if the **Enter** key or the **A** button on the game pad have been released. If they have, they call the **OnSelected** method of the **Control** class passing null for the **EventArgs** parameter. It then check to see that the left mouse button has been released. If it has it measures the size of the text using the MeasureString method of the SpriteFont. It then creates a rectangle around the text. If the mouse pointer is in the area it calls the **OnSelected** method.

In other tutorials I will be adding in other controls. For now though, there are enough controls to move between two screens. What I'm going to do is add in a game state, **StartMenuScreen**. This state will be a menu that the player will choose items from a menu on how they wish to proceed. To move from the **TitleScreen** to the **StartMenuScreen** there will be a link label on the **TitleScreen** that if selected will move to the **StartMenuScreen**.

To use the **ControlManager** you need a **SpriteFont**. Open the **MonoGame Pipeline Tool** by double clicking the **Content.mgcb** file in the **Content** folder of the **EyesOfTheDragon** project. In the **MonoGame Pipeline Tool** right click the **Content** node, select **Add** and then **New Folder**. Name this new folder **Fonts**. Right click the **Fonts** folder, select **Add** and then **New Item**. Select the **Sprite Font** entry and name it **ControlFont**. Save the project and close the **MonoGame Pipeline Tool**. In the **Fonts** folder in the **Content** folder open the **ControlFont.spritefont** file. Change the **Size** element to 20.

I first want to extend the **BaseGameState** a little. I want to add in a **ControlManager** to that state. Change the **BaseGameState** class to the following.

```
using MGRpgLibrary;
using MGRpgLibrary.Controls;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;

namespace EyesOfTheDragon.GameScreens
{
    public abstract partial class BaseGameState : GameState
    {
        #region Fields region

        protected Game1 GameRef;
        protected ControlManager ControlManager;
        protected PlayerIndex playerIndexInControl;

        #endregion

        #region Properties region
        #endregion

        #region Constructor Region

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

        #endregion

        #region XNA Method Region

        protected override void LoadContent()
        {
            ContentManager Content = Game.Content;
            SpriteFont menuFont = Content.Load<SpriteFont>(@"Fonts\ControlFont");
            ControlManager = new ControlManager(menuFont);
            base.LoadContent();
        }

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

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

        #endregion
    }
}
```

I added in the over rides for the **LoadContent**, **Update**, and **Draw** methods. The **Update** and **Draw** methods just call those methods of the base class. I create a new instance of the **ControlManager** class in the **LoadContent** method. I get the **ContentManager** from our game using the **Content** property. I load in the menu font and create the control manager instance.

I want to add another screen to the game before I show the use of the **ControlManager**. Right click the **GameScreens**, select **Add** and then **Class**. Name this new class **StartMenuScreen**. This is the code for that class.

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

namespace EyesOfTheDragon.GameScreens
{
    public class StartMenuScreen : BaseGameState
    {
        #region Field region
        #endregion

        #region Property Region
        #endregion

        #region Constructor Region

        public StartMenuScreen(Game game, GameStateManager manager)
        : base(game, manager)
        {
        }

        #endregion

        #region XNA Method Region

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

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

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

        public override void Draw(GameTime gameTime)
        {
            if (InputHandler.KeyReleased(Keys.Escape))
            {
                Game.Exit();
            }

            base.Draw(gameTime);
        }

        #endregion

        #region Game State Method Region
        #endregion
    }
}
```

This is state a skeleton, for now. In the next tutorial I will add more to it. It is here so I can show how to move from one game state to another. I did add in a condition that checks to see if the Escape key is released. If it is, the game exits. This isn't ideal behavior for a game. I will fix that in another tutorial. Switch back to the code for the **Game1** class. Add a property for this screen just below the **TitleScreen** property.

```
        public StartMenuScreen StartMenuScreen { get; private set; }
```

With that done, you will want to create a **StartMenuScreen** in the **Game1** constructor. Change that constructor to the following.

```
        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);
            StartMenuScreen = new StartMenuScreen(this, _gameStateManager);

            _gameStateManager.ChangeState(TitleScreen);
        }
```

All you are doing is creating an instance of the **StartMenuScreen** and assigning it to the field in the **Game1** class. Flip back to the code for the **TitleScreen**. The changes I made were extensive so I will give you the code for the entire class. Change the code for the **TitleScreen** class to the following.

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

namespace EyesOfTheDragon.GameScreens
{
    public class TitleScreen : BaseGameState
    {
        #region Field region

        Texture2D backgroundImage;
        LinkLabel startLabel;

        #endregion

        #region Constructor region

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

        #endregion

        #region XNA Method region

        protected override void LoadContent()
        {
            ContentManager Content = GameRef.Content;
            backgroundImage = Content.Load<Texture2D>(@"Backgrounds\titlescreen");

            base.LoadContent();

            startLabel = new LinkLabel();
            startLabel.Position = new Vector2(350, 600);
            startLabel.Text = "Press ENTER to begin";
            startLabel.Color = Color.White;
            startLabel.TabStop = true;
            startLabel.HasFocus = true;
            startLabel.Selected += new EventHandler(startLabel_Selected);

            ControlManager.Add(startLabel);
        }
        public override void Update(GameTime gameTime)
        {
            ControlManager.Update(gameTime, PlayerIndex.One);
            base.Update(gameTime);
        }

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

            base.Draw(gameTime);

            GameRef.SpriteBatch.Draw(
                backgroundImage,
```

```
                GameRef.ScreenRectangle,
                Color.White);

            ControlManager.Draw(GameRef.SpriteBatch);

            GameRef.SpriteBatch.End();
        }

        #endregion

        #region Title Screen Methods

        private void startLabel_Selected(object sender, EventArgs e)
        {
            StateManager.PushState(GameRef.StartMenuScreen);
        }

        #endregion
    }
}
```

The first change was the addition of a **LinkLabel** control to the field section. I will wire an event handler for the **Selected** event of the **LinkLabel**. In the **LoadContent** method I create the link label and add it to the control manager. It is important that you construct controls on game screens after the call to **base.LoadContent**. The reason is the control manager will not exist until after that. I set the position of the **LinkLabel**, I did this by trial and error, the text, colour, tab stop, and has focus properties. I also wire the event handler if the player presses either **Enter** or **A** on the controller. The last step is adding the **LinkLabel** to the control manager.

In the **Update** method I call the **Update** method of the **ControlManager**. For the parameters I pass in the **GameTime** parameter from the **Update** method and **PlayerIndex.One** for the player index. For now I will only be accepting input from the controller and **PlayerIndex.One**. In the **Draw** method I call the **Draw** method of the **ControlManager**. If you call this before you draw the background image, the background image will be drawn over top of the link label. I pass in the **SpriteBatch** from our game reference.

The last method is the **startLabel_Selected** method. This will be called automatically if the player selects the start label, you do not have to call this method explicitly. This is where events can be very nice to work with. If you are interested in responding to an event you subscribe, or wire an event handler to the event. This method just calls the **PushState** method of the **GameStateManager** passing in the **StartMenuScreen**.

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