Eyes of the Dragon Tutorials 4.0

Part 2

GUI Framework

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.

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 within Windows, Mac or Linux. You have to make the controls yourself. It is a good idea also to 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 **SharedProject** 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 follows 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 SharedProject.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.Black;
            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)
        {
            Selected?.Invoke(this, e);
        }

        #endregion
    }
}
```

This class has many protected fields that are common to controls. Public properties expose these

fields, so they can be overridden in inherited classes. There is a protected virtual method, **Selected**, 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 them. 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 that 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 to be 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 black. 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 **SharedProject** 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 SharedProject.Controls
{
    public class ControlManager : List<Control>
    {
        #region Fields and Properties

        int selectedControl = -1;
        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 ((Xin.IsKeyDown(Keys.LeftShift) || Xin.IsKeyDown(Keys.RightShift))
        && Xin.WasKeyPressed(Keys.Tab))
    {
        PreviousControl();
    }

    if (Xin.WasKeyPressed(Keys.Tab))
    {
        NextControl();
    }
}

public void Draw(SpriteBatch spriteBatch)
{
    foreach (Control c in this)
    {
        if (c.Visible)
```

```
            {
                c.Draw(spriteBatch);
            }
        }
    }

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

        if (selectedControl == -1)
        {
            selectedControl = 0;
        }

        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;
        }

        if (selectedControl == -1)
        {
            selectedControl = 0;
        }

        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. I haven't implemented game pad input yet, that will be in a future tutorial. 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 shift tab to move to the previous control or just the tab key to move to the next control. When I implement game pad input I will update this to use the D-pad. If either of the shift keys are down and the tab key has been pressed. The **PreviousControl** method is called. Similarly, if the tab key has been pressed 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 check to see if the selected index is -1 and if it is set it to zero. 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 **SharedProject** 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;
using SharedProject.Controls;
using static System.Net.Mime.MediaTypeNames;

namespace SharedProject.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 namespaces into scope. The constructor of the **Label** class sets the **tabStop** field to **false** by default so **Labels** can't be selected use tabs. You can, of course, override this behaviour 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 with tabs. It is also like a button without a background. 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 **SharedProject** project, select **Add** and then **Class**. Name this new class **LinkLabel**. This is the code for the **LinkLabel** class.

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

namespace SharedProject.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 (Xin.WasKeyReleased(Keys.Enter))
            {
                base.OnSelected(null);
                return;
            }

            if (Xin.WasMouseReleased(MouseButtons.Left))
            {
                size = SpriteFont.MeasureString(Text);

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

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

        #endregion
    }
}
```

Again, the class is simplistic but combined with the control manager; it can be pretty 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 colour 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 selected 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 has been released. If they have, they call the **OnSelected** method of the **Control** class passing null for the **EventArgs** parameter. It then checks 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 **SharedProject** project. In the **MGCB Editor** 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 **MGCB Editor**. 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 **GameState** a little. I want to add in a **ControlManager** to that state. Change the **GameState** class to the following.

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

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

        protected ControlManager controls;

        private readonly List<GameComponent> childComponents;

        protected SpriteBatch SpriteBatch { get; set; }

        public ControlManager ControlManager { get { return controls; } }
```

```
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>();
    controls = new(Game.Content.Load<SpriteFont>(@"Fonts/ControlFont"));
}

public override void Update(GameTime gameTime)
{
    foreach (GameComponent component in childComponents)
    {
        if (component.Enabled)
            component.Update(gameTime);
    }

    ControlManager.Update(gameTime, PlayerIndex.One);
    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);
            }
        }

        ControlManager.Draw(SpriteBatch);

        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
    }
}
```

I added a ControlManager field and a property to expose it. In the **LoadContent** method create the instance of the control manager. The **Update** method I call the Update method of the control manager and in the **Draw** method, I call its Draw method.

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

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

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

    public class StartMenuState : GameState, IStartMenuState
    {
        public StartMenuState(Game game) : base(game)
        {
            Game.Services.AddService<IStartMenuState>(this);
        }

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

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

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

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

First off, there is an interface like the **TitleState**. This is state a skeleton, for now. In the following tutorial, I will add more to it. It is here so I can show how to move from one game state to another. In the constructor, I register the state as a service. 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 behaviour for a game. I will fix that in another tutorial. Switch back to the code for the **Game1** class. Replace it with the following.

```
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 StartMenuState StartMenuState { 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);
            StartMenuState = 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);
        }
    }
}
```

What is new is the addition of a StartMenuState property. Then, in the constructor, I initialize the property.

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 SharedProject.Controls;

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");

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

            ControlManager.Add(startLabel);
        }

        private void StartLabel_Selected(object sender, EventArgs e)
        {
            StartMenuState state = (StartMenuState)Game.Services.GetService<IStartMen-
uState>().Tag;
            StateManager.ChangeState(state);
        }

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

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

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

            base.Draw(gameTime);

            SpriteBatch.End();
        }

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

            LoadContent();
        }

        #endregion
```

```
    }
}
```

The first change was the addition of a **LinkLabel** control to the **LoadContent**. Typically, I would make it a field. In this case, though, I will not be accessing properties after I create them. Also, it is accessible through the control manager. I then 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 vital that you construct controls on game screens after the call to **base.LoadContent**. The reason is that 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 do not need to call the **Update** method of the **ControlManager**. It will be called in the call to **base.Update**. In the **Draw** method, I call **Begin** on the **SpriteBatch** object. I then draw the background image because I want the other content drawn over the top of it. I then call the **Draw** method of the base class to draw the **ControlManager**. Finally, I call the **End** method to perform the rendering.

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. The first step is to retrieve a reference to the **StartMenuState** by retrieving it from the collection of services. The method then calls the **ChangeState** method of the **GameStateManager** passing in the **StartMenuState**.

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