A Summoner's Tale
Chapter 4
Graphical User Interface

Welcome to my Summoner's Tale tutorial series on creating a Pokémon-inspired game with MonoGame. I'm writing these tutorials for the MonoGame 3.8.1 framework using Visual Studio 2022. The code should work on previous versions of MonoGame and Visual Studio. I plan on creating the editors on macOS and Windows, except the map editor. The map editor will be Windows only.

The tutorials will make more sense if they are read in order. You can find the list of tutorials on my web blog, A Summoner's Tale page. In addition to the PDFs, I will make the code for each tutorial available on GitHub here: https://github.com/Synammon/summoners-tale. It will also be included on the page that links to the tutorials.

I want to get to the map editor soon. I also want to make it cross-platform. So, I will not use Windows Forms as I do in my game. Instead, I'm going to build it entirely with MonoGame. That means that we will need some controls to make our forms. I know there are libraries for both, but I feel it is essential to learn these things and not just use pre-built libraries.

To get started, right-click the SummonersTale project, select Add and the New Folder. Name this new folder Forms. Now, right-click the SummonersTale project, choose Add and then Class. Name this new class Control. Here is the code for the Control class.

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

namespace SummonersTale.Forms
{
    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;
        protected bool _mouseOver;
```

```
#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 virtual 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;
            _mouseOver = false;
        }

        #endregion

        #region Abstract Methods

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

        #endregion

        #region Virtual Methods

        protected virtual void OnSelected(EventArgs e)
        {
            Selected?.Invoke(this, e);
        }
```

```
        #endregion
    }
}
```

This class is the base class that all controls will inherit from. This will allow us to use polymorphism and group all controls together in one collection. We don't want to create instances of the base class, so it is abstract. There are a number of protected fields. There is one for the name and one for the text that belongs to the control. There are fields for the size and the position. The _value field is for holding different types of data. There are some bool fields next. There is one that tells if the control has focus. Next, there are fields that tell if the control is enabled and visible. The _tabStop field determines if focus will leave/enter the tab key has been pressed. For rendering text, there is a SpriteFont field. To draw the control, we also need colour. The _type field returns the type of control. Finally, there is a field that holds if the mouse cursor is over the control. There is also an event Selected that is first when the control is selected. There are properties to expose all of the fields.

There are a few fields initialized in the constructor. They are the colour which defaults to white. The are enabled and visible. The font is set to the font of the control manager, which we will add shortly. Finally, I initialize the _mouseOver field.

There are three abstract methods that must be implemented in any child classes. The first is Update. It will update the control. The second is Draw, and that will, of course, render the control. Finally, the HandleInput method will handle mouse and keyboard input. There is also a virtual method that will fire if the control is selected. It is virtual, so it can be overridden in classes that inherit from it.

We have a control class, but we need a class to manage them as a collection. Right-click the Forms folder, select Add and then Class. Name this new class ControlManager. Here is the code for that class.

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

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

        int _selectedControl = -1;
        bool _acceptInput = true;

        static SpriteFont _spriteFont;

        public static SpriteFont SpriteFont
        {
```

```
            get { return _spriteFont; }
        }

        public bool AcceptInput
        {
            get { return _acceptInput; }
            set { _acceptInput = value; }
        }

        #endregion

        #region Event Region

        public event EventHandler FocusChanged;

        #endregion

        #region Constructors

        private ControlManager()
            : base()
        {
        }

        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)
        {
            if (Count == 0)
            {
                return;
            }

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

            foreach (Control c in this)
            {
```

```
                if (c.HasFocus)
                {
                    c.HandleInput();
                    break;
                }
            }

            if (!AcceptInput)
            {
                return;
            }

            if (Xin.WasKeyReleased(Keys.Tab) && (Xin.IsKeyDown(Keys.LeftShift) ||
Xin.IsKeyDown(Keys.RightShift)))
            {
                PreviousControl();
            }

            if (Xin.WasKeyReleased(Keys.Tab) && !(Xin.IsKeyDown(Keys.LeftShift) ||
Xin.IsKeyDown(Keys.RightShift)))
            {
                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)
```

```
                {
                    FocusChanged?.Invoke(this[_selectedControl], null);

                    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)
                {
                    FocusChanged?.Invoke(this[_selectedControl], null);

                    break;
                }
            } while (currentControl != _selectedControl);

            this[_selectedControl].HasFocus = true;
        }

        #endregion
    }
}
```

This class inherits from the List<T> class, where T is Control so that it will act as a container, and there is no need for a field for a container. There are three fields: _selectedControl, _acceptInput and _spriteFont. The first is which control is currently selected. The first two determine which control is currently selected and if the control manager accepts input. That is useful in the event that you have multiple control managers and only want one active at a time. The SpriteFont field is static and will be shared by all control managers. There are properties that expose the fields, except for the _selectedControl field. There is also an event that will fire when focus leave a control for another control.

There are three constructors. The first is private and takes no parameters. It is private because I only want to accept parameters. The second takes two parameters, which are a SpriteFont and an integer, which is the initial capacity of the list. The third takes as parameter a SpritFont and an IEnumerable<Control> so that you can pass in a collection of controls.

There are four methods. They are Update, Draw, PrievousControl and NextControl. As the names imply, they update the control manager, draw the control manager, move focus to the previous control and move focus to the next control.

The Update method takes a GameTime parameter that will be passed to any controls added. If the Count property is zero, there are no controls that need to be updated the method is exited. In a foreach loop, I loop over all of the controls. If their Enabled property is true, I call their Update method. In a second foreach loop, I loop over the controls again. If a control has focus, I call the HandleInput method of the control. If the AcceptInput property is false, I exit the method. In order to move focus to the next control, l to the next control, I check to see if the tab key has been released and if either of the shift keys is down. Similarly, if the tab key has been pressed and neither shift keys are down, I call the method to move focus to the next control.

The next method is the Draw method, and that method takes a SpriteBatch parameter. In a foreach loop, it cycles all of the controls it contains. If calls their Draw method if the Visible property is true.

The NextControl method is next and, as mentioned, moves control to the next control. If first checks to make sure that is actually controls. If there are not, it exits the method. You need to know what the currently selected control is. The HasFocus property of the control is set to false. Next is a do-while loop. It increments the _selectedControl property by one. If the _selectedControl is the Count property of the class, it is then set to 0. Next is an if statement checks if the control has a TabStop property of true and the control has an Enabled property of true. It invokes the FocusChanged event handler if it subscribed to and breaks out of the loop. The loop will continue until the current control is back to the selected control. Finally, the HasFocus property of the select control is set to true.

The PreviousControl works basically the same way. The only difference is that it decrements the current control rather than incrementing it. Also, it checks to see if the current control is minus one and moves it to the number of items in the collection minus one.

Okay, off to a good start. I'm going to implement some simple controls: a label, a picture box, and a button. I won't be implementing text boxes at this point, as they are a little complicated. First, I will implement a label. Right-click on the Forms folder, select Add and then Class. Name this new class Label. Replace the boilerplate code with the following.

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

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

namespace SummonersTale.Forms
{
    public class Label : Control
    {
        public Label()
        {
            _visible = true;
            _enabled = true;
            _tabStop = false;
        }

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

        public override void HandleInput()
        {
        }

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

A rather simple class, almost not worth being part of controls as we could just draw the text manually. It could be extended with new properties. All it does is set the _visible and _enabled fields s to true. Also, it sets the _tabStop property to false. Then, in the Draw method, it uses the SpriteBatch object passed in to draw the Text.

Now, I will implement the PictureBox control. It is a little more complicated than the Label class. So, right-click the Forms folder in The SummonersTale project, select Add and then Class. Name the new class PictureBox. Replace the boilerplate code with the following.

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

namespace SummonersTale.Forms
{
    public enum FillMethod { Clip, Fill, Original, Center }

    public class PictureBox : Control
    {
        #region Field Region
```

```
        private Texture2D _image;
        private Rectangle _sourceRect;
        private Rectangle _destRect;
        private FillMethod _fillMethod;
        private int _width;
        private int _height;

        #endregion

        #region Property Region

        public Texture2D Image
        {
            get { return _image; }
            set { _image = value; }
        }

        public Rectangle SourceRectangle
        {
            get { return _sourceRect; }
            set { _sourceRect = value; }
        }

        public Rectangle DestinationRectangle
        {
            get { return _destRect; }
            set { _destRect = value; }
        }

        public FillMethod FillMethod
        {
            get { return _fillMethod; }
            set { _fillMethod = value; }
        }

        public int Width
        {
            get { return _width; }
            set { _width = value; }
        }

        public int Height
        {
            get { return _height; }
            set { _height = value; }
        }

        #endregion

        #region Constructors

        public PictureBox(Texture2D image, Rectangle destination)
        {
            Image = image;

            DestinationRectangle = destination;

            Width = destination.Width;
```

```
            Height = destination.Height;

            if (image != null)
                SourceRectangle = new Rectangle(0, 0, image.Width, image.Height);
            else
                SourceRectangle = new Rectangle(0, 0, 0, 0);

            Color = Color.White;

            _fillMethod = FillMethod.Original;

            if (SourceRectangle.Width > DestinationRectangle.Width)
            {
                _sourceRect.Width = DestinationRectangle.Width;
            }

            if (SourceRectangle.Height > DestinationRectangle.Height)
            {
                _sourceRect.Height = DestinationRectangle.Height;
            }
        }

        public PictureBox(Texture2D image, Rectangle destination, Rectangle source)
            : this(image, destination)
        {
            SourceRectangle = source;
            Color = Color.White;

            _fillMethod = FillMethod.Original;

            if (SourceRectangle.Width > DestinationRectangle.Width)
            {
                _sourceRect.Width = DestinationRectangle.Width;
            }

            if (SourceRectangle.Height > DestinationRectangle.Height)
            {
                _sourceRect.Height = DestinationRectangle.Height;
            }
        }

        #endregion

        #region Abstract Method Region

        public override void Update(GameTime gameTime)
        {
        }

        public override void Draw(SpriteBatch spriteBatch)
        {
            if (_image != null)
            {
                switch (_fillMethod)
                {
                    case FillMethod.Original:
                        _fillMethod = FillMethod.Original;
```

```
                        if (SourceRectangle.Width > DestinationRectangle.Width)
                        {
                            _sourceRect.Width = DestinationRectangle.Width;
                        }

                        if (SourceRectangle.Height > DestinationRectangle.Height)
                        {
                            _sourceRect.Height = DestinationRectangle.Height;
                        }
                        spriteBatch.Draw(Image, DestinationRectangle,
SourceRectangle, Color);
                        break;
                    case FillMethod.Clip:
                        if (DestinationRectangle.Width > SourceRectangle.Width)
                        {
                            _destRect.Width = SourceRectangle.Width;
                        }

                        if (_destRect.Height > DestinationRectangle.Height)
                        {
                            _destRect.Height = DestinationRectangle.Height;
                        }
                        spriteBatch.Draw(Image, DestinationRectangle,
SourceRectangle, Color);
                        break;
                    case FillMethod.Fill:
                        _sourceRect = new(0, 0, Image.Width, Image.Height);
                        spriteBatch.Draw(Image, DestinationRectangle,
SourceRectangle, Color);
                        _destRect.Width = Width;
                        _destRect.Height = Height;
                        spriteBatch.Draw(Image, DestinationRectangle,
SourceRectangle, Color);
                        break;
                    case FillMethod.Center:
                        _sourceRect.Width = Image.Width;
                        _sourceRect.Height = Image.Height;
                        _sourceRect.X = 0;
                        _sourceRect.Y = 0;

                        Rectangle dest = new(0, 0, Width, Height);

                        if (Image.Width >= Width)
                        {
                            dest.X = DestinationRectangle.X;
                        }
                        else
                        {
                            dest.X = DestinationRectangle.X + (Width - Image.Width)
/ 2;
                        }

                        if (Image.Height >= Height)
                        {
                            dest.Y = DestinationRectangle.Y;
                        }
                        else
                        {
```

```
                            dest.Y = DestinationRectangle.Y + (Height -
Image.Height) / 2;
                        }
                        spriteBatch.Draw(Image, dest, SourceRectangle, Color);
                        break;
                }
            }
        }

        public override void HandleInput()
        {
        }

        #endregion

        #region Picture Box Methods

        public void SetPosition(Vector2 newPosition)
        {
            _destRect = new Rectangle(
                (int)newPosition.X,
                (int)newPosition.Y,
                Width,
                Height);
        }

        #endregion
    }
}
```

There is an enumeration with four values: Clip, Fill, Original, and Center. They define how the picture box will be rendered. If the value is Clip, the image will be clipped if it overflows the size of the picture box. Fill has the image fill the picture box. If it is greater than the bounds of the picture box, the image is scaled to fill the picture box. The Original is similar to the Clip value. I probably could have gotten by with just Clip. Finally, Center with center the image inside the picture box. It clip the edges if they exceed the bounds of the picture box.

The class inherits from Control, so it can be added to a ControlManager instance. There are a total of six fields in the class: _image, _sourceRect, _destRect, _fillMethod, _width, and _height. The fist holds the image to be drawn. The second and third and the source rectangle to be drawn and the destination to be drawn. The _fillMethod field determines the way the image will be drawn. _width and _height are the height and width of the picture box. Also, there are properties to expose the values.

```
There are two constructors for the class. One takes the image and the desination
rectangle to draw the picture box. The second takes a source rectangle in addition
to the other parameters. The first constructor sets the Image property to the image
passed in. It also sets the destination rectangle. If the Image is not null, the
source rectangle is set to the entire image. If it is null, the source rectangle is
set to an empty rectangle. The colour is set to white so that it will be drawn
without tinting. The fill method is set to original it will be drawn at its original
size. If the width of the source rectangle is greater than the width of the
destination rectangle, it is set to the width of the destination rectangle, clipping
```

it. Similarly, if the height of the source rectangle is greater than the height of the destination, it is set to the height of the destination, clipping it.

The second constructor takes a third parameter, the source rectangle. It calls the first constructor. I think the only thing it does differently is set the source rectangle to the parameter passed in.

The Update method is implemented but does nothing. All of the magic happens in the Draw method. If the image is not null, there is a switch on the FillMethod passed in. If it is set to Original, it checks if the width of the images is greater than the width of the destination. If it is, the width of the source rectangle is set to the width of the destination. I do the same for the heights. I then draw the image. For Clip, I check to see if the destination is great than the source. If it is, the destination is set to the width. This probably isn't doing what I intended. I will fix it in a future tutorial. It then draws the image to the destination rectangle. Fill draws the entire image into the destination rectangle. So, the width and height of the image are set to the width and height of the image. Fill sets the source rectangle to be the entire image. The destination rectangle is set to the width and height of the image. The image is then rendered. The Center mode is a little tricky. First, the source rectangle is set to the size of the image. Next, a new rectangle is made that has the height and width of the control. If the width of the image is greater than the width of the picture box, the X coordinate of the destination is set to the X coordinate of the picture box. Otherwise, it is set to the X coordinate of the picture box plus the width of the picture box minus the width of the image divided by 2. I do something similar for the height of the image. I then draw the rectangle. Then I draw the image.

The HandleInput doesn't do anything but needs to be implemented because it inherits from Control. There is a method, SetPosition that positions the picture box. It takes a Vector2, which is the position to place the picture box. Then, it sets the _destRect field to a rectangle that is the position, width, and heigh of the picture box.

Whew, that was a long one. I will probably revisit the draw method at some point. It will be good enough for now. Let's add the last control, a button. Right-click the Forms folder in the SummonersTale project, select Add and then Class. Name this new class Button. Here is the code.

```
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input.Touch;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework;
using SummonersTale.Forms;
using System;
using System.Collections.Generic;
using System.Text;
using static System.Net.Mime.MediaTypeNames;

namespace SummonersTale
{
    public enum ButtonRole { Accept, Cancel, Menu }

    public class Button : Control
    {
        #region

        public event EventHandler Click;
```

```
        #endregion
        #region Field Region

        private readonly Texture2D _background;
        float _frames;

        public ButtonRole Role { get; set; }
        public int Width { get { return _background.Width; } }
        public int Height { get { return _background.Height; } }

        #endregion

        #region Property Region
        #endregion

        #region Constructor Region

        public Button(Texture2D background, ButtonRole role)
        {
            Role = role;
            _background = background;
        }

        #endregion

        #region Method Region

        public override void Draw(SpriteBatch spriteBatch)
        {
            Rectangle destination = new(
            (int)Position.X,
                (int)Position.Y,
                _background.Width,
                _background.Height);

            spriteBatch.Draw(_background, destination, Color.White);

            _spriteFont = ControlManager.SpriteFont;

            Vector2 size = _spriteFont.MeasureString(Text);
            Vector2 offset = new((_background.Width - size.X) / 2,
(_background.Height / 2));

            spriteBatch.DrawString(_spriteFont, Text, ((Position + offset)), Color);
        }

        public override void HandleInput()
        {
            MouseState mouse = Mouse.GetState();
            Point position = new(mouse.X, mouse.Y);

            Rectangle destination = new(
                (int)_position.X,
                (int)_position.Y,
                _background.Width,
                _background.Height);

            if ((Role == ButtonRole.Accept && Xin.WasKeyReleased(Keys.Enter)) ||
```

```
                (Role == ButtonRole.Accept && Xin.WasKeyReleased(Keys.Space)))
            {
                OnClick();
                return;
            }

            if (Role == ButtonRole.Cancel && Xin.WasKeyReleased(Keys.Escape))
            {
                OnClick();
                return;
            }

            if (destination.Contains(position) &&
Xin.WasMouseReleased(MouseButton.Left) && _frames >= 5)
            {
                OnClick();
            }
        }

        private void OnClick()
        {
            Click?.Invoke(this, null);
        }

        public override void Update(GameTime gameTime)
        {
            _frames++;
            HandleInput();
        }

        public void Show()
        {
            _frames = 0;
        }

        #endregion
    }
}
```

There is an enumeration included in this class. It defines the role of a button. If a button has the Accept role, its events will fire if the user hits the enter key or the space bar. Similarly, the event will fire if the role is Cancel and the Escape key is pressed. The Menu option is for normal buttons. I know. I probably could have come up with a better name. After ten hours of coding, I didn't make have much imagination in me. The control, of course, inherits from the Control class. It has an event for if the button was activated.

There are two fields in the class: _background and _frames. The _background is readonly and holds the image of the button. The next is _frames which is a little tricky. It counts the number of frames since the button was activated. That is because MonoGame stutters with input. It can trigger the event a few times after the event has happened. So, I count a few frames before allowing the button to be clicked again. There are properties for the role of the button and the height and width of the button.

The constructor takes as parameters the background for the button and the role of the buttons. It sets the Role property and _background field to the values passed in.

The Draw method draws the button. First, it creates a rectangle using the position of the button and the height and width of the texture. It then draws the texture. After drawing the texture, I grab the font from the control manager. I use the font to measure the size of the text. I then center it in the button by taking the width of the button minus the X value of the string and dividing that by two. This is very common in game programming, so I suggest memorizing the formula. Centering vertically is done by taking the height of the object you want to center in minus the height of the object to be centered. You then draw the object by adding the offset to the position you want to center at.

The HandleInput method grabs the mouse state for now. Eventually, I will add some more features to Xin. This tutorial is long enough as it is. It then creates a point that describes the position of the mouse to test if the mouse was clicked on the button. Next, I create a rectangle that defines the button's location. I then check to see if the enter key was pressed or the space bar and the role of the button is accept. If either combination is true, I call the OnClick event and exit the method. Similarly, if the escape key was pressed and the role is set to cancel, I call the OnClick method and exit the method. After the keyboard input, I check to see if the destination rectangle contains the mouse position and _frames is greater than or equal to 5. If that is true, I call the OnClick method. The OnClick method invokes the event handler if it is subscribed to.

The Update method increments the _frames field and calls the HandleInput method. Finally, the Show method resets the _frame field to zero.

That is a lot to throw on you at once. I think you can handle just a little more. I'm going to add a new project to the solution for the editor. Right-click the SummorsTale solution, select Add and then New Project. Search for MonoGame Cross-Platform Desktop Application and click Next. On the page that comes up, enter SummonersTaleEditor. Now, we need to add a few references to the project. Right-click the SummonersTaleEdit, select Add and the Shared Project Reference. Select the SummonersTale project. Right-click the SummonersTaleEdit project, select Add and the Project Reference. Select the Psilibrary project and the SummonersTaleGame as references.

This is going to break things. The problem is that BaseGameState has a field that references the Game1 object. This works well if you are targeting a single platform. Once you start adding other platforms, things break. So, I made a few modifications to the game and the editor. I will start with the editor, because it has the fewest changes. Right-click the Game1 class in the SummonersTaleEditor and select Rename. When you are asked if you want to rename elements choose yes. This will stop confusion between the editor and the game.

That leaves the game. What I did was add the states as services as well as the SpriteBatch. First, let's the changes to the BaseGameState. What I did was remove the GameRef field and add a SpriteBatch field.

Then, in the constructor, I retrieve the SpriteBach object. Replace the BaseGame state with the following code.

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

namespace SummonersTale.StateManagement
{
    public class BaseGameState : GameState
    {
        #region Field Region

        protected readonly static Random random = new();
        protected readonly SpriteBatch spriteBatch;

        #endregion

        #region Proptery Region

        public SpriteBatch SpriteBatch
        {
            get { return spriteBatch; }
        }

        #endregion

        #region Constructor Region

        public BaseGameState(Game game)
            : base(game)
        {
            spriteBatch =
(SpriteBatch)Game.Services.GetService(typeof(SpriteBatch));
        }

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

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

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

        #endregion
    }
}
```

So, just a little bit new there. The only real change is removing the Game1 reference, adding the SpriteBatch, and retrieving the SpriteBatch in the constructor using Game.Services.GetService. Now, I will tackle TitleState. Replace the TitleState with the following code.

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

namespace SummonersTale.StateManagement
{
    public interface ITitleState
    {
        GameState GameState { get; }
    }

    public class TitleState : BaseGameState, ITitleState
    {
        private SpriteFont _spriteFont;
        private double _timer;

        public GameState GameState => this;

        public TitleState(Game game) : base(game)
        {
            Game.Services.AddService((ITitleState)this);

            Initialize();
            LoadContent();
        }

        public override void Initialize()
        {
            _timer = 5;

            base.Initialize();
        }

        protected override void LoadContent()
        {
            _spriteFont = content.Load<SpriteFont>(@"Fonts/MainFont");

            base.LoadContent();
        }

        public override void Update(GameTime gameTime)
        {
            _timer -= gameTime.ElapsedGameTime.TotalSeconds;

            if (_timer <= 0)
            {

manager.ChangeState((GameState)Game.Services.GetService(typeof(IGamePlayState)));
            }

            base.Update(gameTime);
```

```
        }

        public override void Draw(GameTime gameTime)
        {
            string message = "Game with begin in " + ((int)_timer).ToString() + "
seconds.";
            Vector2 size = _spriteFont.MeasureString(message);

            SpriteBatch.Begin();

            SpriteBatch.DrawString(
                _spriteFont,
                message,
                new((1280 - size.X) / 2, 720 - (_spriteFont.LineSpacing * 5)),
                Color.White);

            SpriteBatch.End();

            base.Draw(gameTime);
        }
    }
}
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using System;
using System.Collections.Generic;
using System.Text;

namespace SummonersTale.StateManagement
{
    public interface ITitleState
    {
        GameState GameState { get; }
    }

    public class TitleState : BaseGameState, ITitleState
    {
        private SpriteFont _spriteFont;
        private double _timer;

        public GameState GameState => this;

        public TitleState(Game game) : base(game)
        {
            Game.Services.AddService((ITitleState)this);

            Initialize();
            LoadContent();
        }

        public override void Initialize()
        {
            _timer = 5;

            base.Initialize();
        }

        protected override void LoadContent()
```

```
        {
            _spriteFont = content.Load<SpriteFont>(@"Fonts/MainFont");

            base.LoadContent();
        }

        public override void Update(GameTime gameTime)
        {
            _timer -= gameTime.ElapsedGameTime.TotalSeconds;

            if (_timer <= 0)
            {

manager.ChangeState((GameState)Game.Services.GetService(typeof(IGamePlayState)));
            }

            base.Update(gameTime);
        }

        public override void Draw(GameTime gameTime)
        {
            string message = "Game with begin in " + ((int)_timer).ToString() + "
seconds.";
            Vector2 size = _spriteFont.MeasureString(message);

            SpriteBatch.Begin();

            SpriteBatch.DrawString(
                _spriteFont,
                message,
                new((1280 - size.X) / 2, 720 - (_spriteFont.LineSpacing * 5)),
                Color.White);

            SpriteBatch.End();

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

What has changed here? Well, I added an interface ITitleState that will return the class as GameState. I then implement the interface in the class. A property is added that returns this instance. Then, in the constructor, I register the class as a service cast to the interface. I then call the Initialize method and the LoadContent method. I need to do this because it is the first state placed on the stack. I now use the Game.Services.GetService method to get the reference to the GamePlayState instead of the game reference. Also, in the Draw method, I am using the new SpriteBatch property instead of the reference to the game object.

That leaves the GamePlayState and the Game1 class. Replace the code in the GamePlayState with the following code.

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

```
using Psilibrary.TileEngine;
using System;
using System.Collections.Generic;
using System.Reflection.Metadata;
using System.Text;

namespace SummonersTale.StateManagement
{
    public interface IGamePlayState
    {
        GameState GameState { get; }
    }

    public class GamePlayState : BaseGameState, IGamePlayState
    {
        private TileMap _tileMap;
        private Camera _camera;

        public GameState GameState => this;

        public GamePlayState(Game game) : base(game)
        {
            Game.Services.AddService((IGamePlayState)this);
        }

        public override void Initialize()
        {
            Engine.Reset(new(0, 0, 1280, 720), 32, 32);
            _camera = new();

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

            TileSheet sheet = new(content.Load<Texture2D>(@"Tiles/TX Tileset
Grass"), "test", new(8, 8, 32, 32));
            TileSet set = new(sheet);

            TileLayer ground = new(100, 100, 0, 0);
            TileLayer edge = new(100, 100, -1, -1);
            TileLayer building = new(100, 100, -1, -1);
            TileLayer decore = new(100, 100, -1, -1);

            for (int i = 0; i < 1000; i++)
            {
                ground.SetTile(random.Next(0, 100), random.Next(0, 100), 0,
random.Next(0, 64));
            }

            _tileMap = new(set, ground, edge, building, decore, "test");

            base.LoadContent();
        }

        public override void Update(GameTime gameTime)
        {
            Vector2 motion = Vector2.Zero;
```

```
            if (Xin.IsKeyDown(Microsoft.Xna.Framework.Input.Keys.A))
            {
                motion.X = -1;
            }
            else if (Xin.IsKeyDown(Microsoft.Xna.Framework.Input.Keys.D))
            {
                motion.X = 1;
            }

            if (Xin.IsKeyDown(Microsoft.Xna.Framework.Input.Keys.W))
            {
                motion.Y = -1;
            }
            else if (Xin.IsKeyDown(Microsoft.Xna.Framework.Input.Keys.S))
            {
                motion.Y = 1;
            }

            if (motion != Vector2.Zero)
                motion.Normalize();

            _camera.Position += motion * 240 *
(float)gameTime.ElapsedGameTime.TotalSeconds;

            if (_camera.Position.X < 0)
            {
                _camera.Position = new(0, _camera.Position.Y);
            }

            if (_camera.Position.X > _tileMap.WidthInPixels - 1280)
            {
                _camera.Position = new(_tileMap.WidthInPixels - 1280,
_camera.Position.Y);
            }

            if (_camera.Position.Y < 0)
            {
                _camera.Position = new(_camera.Position.X, 0);
            }

            if (_camera.Position.Y > _tileMap.HeightInPixels - 720)
            {
                _camera.Position = new(_camera.Position.X, _tileMap.HeightInPixels -
720);
            }

            base.Update(gameTime);
        }

        public override void Draw(GameTime gameTime)
        {
            base.Draw(gameTime);
            _tileMap.Draw(gameTime, SpriteBatch, _camera, false);
        }
    }
}
```

Like in the TitleState there is an interface that returns the instance of the GamePlayState. The only other change was in the Draw method. I use the SpriteBatch object that is retrieved in the BaseGameState.

Almost to the end. Just want to tackle the Game1 class. Replace the code of the Game1 class with the following code.

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Psilibrary.TileEngine;
using SummonersTale;
using SummonersTale.StateManagement;
using System;

namespace SummonersTaleGame
{
    public class Game1 : Game
    {
        private readonly GameStateManager _manager;
        private readonly GraphicsDeviceManager _graphics;
        private SpriteBatch _spriteBatch;

        private GamePlayState _playState;
        private TitleState _titleState;

        public SpriteBatch SpriteBatch => _spriteBatch;

        public TitleState TitleState => _titleState;
        public GamePlayState PlayState => _playState;

        public Game1()
        {
            _graphics = new GraphicsDeviceManager(this)
            {
                PreferredBackBufferWidth = 1280,
                PreferredBackBufferHeight = 720
            };
            _graphics.ApplyChanges();
            _manager = new GameStateManager(this);

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

            Components.Add(_manager);
        }

        protected override void Initialize()
        {
            Components.Add(new FramesPerSecond(this));
            Components.Add(new Xin(this));

            _graphics.ApplyChanges();

            base.Initialize();
        }
```

```
        protected override void LoadContent()
        {
            _spriteBatch = new SpriteBatch(GraphicsDevice);
            Services.AddService(typeof(SpriteBatch), _spriteBatch);

            _playState = new(this);
            _titleState = new(this);

            _manager.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);

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

The first change is that the states are no long readonly. That is because I moved their initialization to the LoadConent method after the creation of the SpriteBatch object. There is no problem sharing the SpriteBatch. It is a reference type, so it uses very little memory on the stack. It is stored on the heap and references point to it. In the LoadContent method, after the creation of the sprite batch, I register the SpriteBatch as a service. After it is registered, I create the states and push the TitleState on the stack.

If you build and run now, everything works as before. I'm not going to dive into the editor in this tutorial. I think I've fed you enough for one day. So, that is going to be it for this tutorial. I covered a lot, and I don't want to overload you. I encourage you to keep visiting my blog for the latest news on my tutorials.

Good luck with your game programming adventures.
Cynthia