A Summoner’s Tale
Chapter 3
Bringing it Together

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.

This tutorial aims to bring the previous two together. It will add two new game states, migrate the tile engine to one of them, and add a basic version of my input handler, Xin. Implementing a full version of Xin would be a tutorial in itself. So, I will implement the relevant pieces for this tutorial and add others later. The first thing I want to do is delete a few files. There are three Game1.cs files. One in each of the projects. The only one that I want is the SummonersTaleGame one. In the SummonersTale and Psilibrary projects, right click the Game1.cs file and select Delete.

So, let’s get started. First, I will add two new game states. The one is an abstraction from which all of our other game states will inherit. Yes, I know we already have the GameState class. I added this one to my game because I have the GameState in the Game Library. I included it here in the event that it needs to be moved in a future tutorial. Right-click the StateManagement folder, select Add and then Class. Name this new class BaseGameState. Here is the code for that class.

```
using Microsoft.Xna.Framework;
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 Game1 GameRef;

        #endregion

        #region Constructor Region

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

        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
    }
}
```

I don't know many games that don't use random numbers. For that reason, there is a protected readonly static Random field to generate random numbers. This is because you don't want it reinitialized, and you want all states to share it. There is very little as counterproductive as having multiple instances of a Random generator. It just makes for some strange number generation sequences. Next, there is a Game1 field that will be a reference to our main game class. This gives us access to other game states and our Sprite Batch. The constructor just sets the Game1 field to the value passed in.

The next thing that I'm going to implement is the input handler. I will start with a stripped-down version because the full-blown version that I use is a tutorial in itself. Right-click the SummonersTale project, select add, and then Class. Name this new class Xin. Here is the code for the Xin class.

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

namespace SummonersTale
{
    public enum MouseButton { Left, Right };

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

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

        public static KeyboardState LastKeyboardState { get { return 
lastKeyboardState; } }
        public static MouseState LastMouseState { get { return lastMouseState; } }
        public Xin(Game game) : base(game)
        {
            keyboardState = Keyboard.GetState();
            mouseState = Mouse.GetState();
        }

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

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

            base.Update(gameTime);
        }

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

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

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

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

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

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

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

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

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

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

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

The constructor initializes the current keyboard and mouse fields. The Update method sets the value of the lastKeyboardState field to the value of the keyboardState field and the lastMouseState value to the mouseState value. Next, it sets the keyboarState and mouseState values to the current state of the keyboard and mouse, respectively. What this does is allow us to compare the previous frame of the game to the current frame of the key. You will see why that is important shortly.

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

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

With the input manager in place, I am going to move to have a couple of game states in place. The first is a title-intro state that will be displayed when the game first launches. The other is the state where gameplay will take place. First, right-click the StateManagement folder in the SummonersTale project, select Add and then Class. Name this new class TitleState.cs. Next, repeat the process and name the class GamePlayState.cs.

Before I add the code to the states, I want to add a font. In the SummonersTale project, expand the Content folder. Then, double-click the Content.mgcb item to open the MonoGame Content Editor. Select the route node and click the Add New Folder button. Name the new folder Fonts. Select the Fonts folder and click the Add New Item button. From the list, choose the Sprite Description. Finally, name the new font MainFont.

So, how this is going to work is that the title state will stay active for five seconds. After five seconds, it will change to the gameplay state. In the future, I will implement methods that check for any key to be pressed or any mouse button. Then, if either is pressed, it will change states if the five seconds is not up.

To start, I am going to add the TitleState code. Replace that class 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 class TitleState : BaseGameState
    {
        private SpriteFont _spriteFont;
        private double _timer;

        public TitleState(Game game) : base(game)
        {
        }

        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(GameRef.PlayState);
            }

            base.Update(gameTime);
        }

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

            GameRef.SpriteBatch.Begin();

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

            GameRef.SpriteBatch.End();

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

The class inherits from BaseGameState, so it can be used with the state manager and have access to all of the public and protected members of the parent classes. There are two fields _spriteFont which is a SpriteFont for rendering text. The other is a double _timer that is used to count down the five seconds. There is an override of the Initialize method that sets the _timer field to the 5 seconds that we are going to count down. There is an override of the LoadContent method the loads the SpriteFont. Make sure you have saved your Content.mgcb file. In the Update method I decrease the _timer field by the elapsed time since the last frame. We can do this because the target time is one full second. If _timer is less than or equal to zero I call the ChangeState method of the StateManager to make the GamePlayState the active state.

In the override of the Draw method, I set a message to be a string with the current value of the _timer field cast to an integer. I then use the MeasureString method of the SpriteFont to see how big the message is. I call Begin on the SpriteBatch property of the Game1 class to start rendering. I then call the DrawString to render the message. The parameters of the DrawString that we are using are the SpriteFont, the text, the position, and the colour. To calculate the position, I take the width of the screen minus the width of the string divided by two to center it horizontally. To place it vertically, I take the height of the screen and subtract five times the line height property of the SpriteFont. Finally, I call the

End method of the SpriteBatch to end rendering. There will be a couple of errors because some properties have not been defined. I will fix that after I've implemented the GamePlayState.

For now, all I am going to do is move the code for rendering the map to the GamePlayState. Replace 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 class GamePlayState : BaseGameState
    {
        private TileMap _tileMap;
        private Camera _camera;

        public GamePlayState(Game game) : base(game)
        {
        }

        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)
        {
            base.Update(gameTime);
        }

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

I'm not going to explain the code, as you've already seen it all before. What I am going to do, is make the necessary changes to the Game1 class to pull it all together. Change the code for the Game1 class to the following.

```
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 readonly GamePlayState _playState;
        private readonly 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);

            _playState = new(this);
            _titleState = new(this);
            _manager.PushState(_titleState);
        }

        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);
        }

        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 the removal of the TileMap and Camera fields. I added two new fields, one for each of the two states that we added. They are marked readonly as they should not be set after the first time. I added properties to expose their values. I also added a property to expose the SpriteBatch object. In the constructor, I initialize the two new game states and push the title state onto the stack. In the Initialize method, I add a new instance of Xin to the list of game components. I removed the code related to the camera and the engine. I removed the code for creating a map from the LoadContent method. Finally, I removed the code for drawing the map from the Draw method.

If you build and run now, the title state will first be displayed. After five seconds, the game will switch to the game play state. As Flemeth would say, it's pretty but useless. It doesn't actually do something. Let's add some code to move the map. I will do it incrementally. The first step is to check if the W, A, S, or D keys are down and move the camera according to the keys being pressed. Change the Update method of the GamePlayState to the following.

```
        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;
            }

            _camera.Position += motion;
            base.Update(gameTime);
        }
```

What the new code does is create a Vector2 called motion that describes the desired motion to move the map. If A is down, the X motion is -1. If the A key is not down, but the D key is down, the X motion is one. Similarly, if the W key is down the Y motion is -1 and 1 if the S key is down. In math, moving up would be 1 and down -1. In MonoGame, it is reversed.

You can build and run now and move the map. The problem is that the map scrolls insanely fast, even at one pixel. Let's fix that. Change the Update method to the following.

```
        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;
            }
            _camera.Position += motion * 240 *
(float)gameTime.ElapsedGameTime.TotalSeconds;
            base.Update(gameTime);
        }
```

So, what I did was multiply motion by 240 and the number of seconds that have passed since the last frame. That effectively says that we want to move the camera by 240 pixels per second. There is another problem, though. If you move the map diagonally, it moves much faster than in a single direction. This is because of math. Remember the old equation: a^2 + b^2 = c^2. So, 1^2 + 1^2 = 2. The diagonal will be the square root of 2, which is greater than 1. How can we fix this? The answer is to normalize the vector. What that means is that no matter the direction, the vector will have a length of one. Change the Update method of the GamePlayState to the following.

```
        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;
            }

            motion.Normalize();
            _camera.Position += motion * 240 *
(float)gameTime.ElapsedGameTime.TotalSeconds;
            base.Update(gameTime);
        }
```

Hey! The game crashes. What gives? What happens is that you have the Zero vector, where both the X and Y values are zero. This is the equivalent of dividing a number by zero. We just need to check that it is not the zero vector before normalizing. Change the code for the Update method to the following code.

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

There is one last problem. The map scrolls off the edges of the screen. What we need to check is if the X and Y coordinates of the camera are within the bounds of the map. The left and top edges are easy. If either is less than zero, set them to zero. The bottom and right edges are a little trickier but easy enough too. Use the MapHeightInPixels and MapWidthInPixels properties. Just remember that you need to subtract the height and width of the screen. Change the Update method to the following.

```
        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);
        }
```

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