A Summoner's Tale
Chapter 1
Getting Started

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.

So, let's get started! When you are building a home, you first pour the foundation to build upon. When building a game, you require a good foundation as well. For that reason, the first few tutorials are going to focus on the foundation.

This game will be cross-platform, targeting multiple systems, Windows, macOS, Android, and possibly iOS. For that reason, I made the base project in the solution a shared library. Shared libraries are great for sharing content and code against multiple platforms. I especially like it for sharing content. You just put it in the shared library, and when Visual Studio builds the project, the content is automatically placed in each of the projects. The only exception I've found is XML-based content generated using the IntermediateSerializer. In case you are unfamiliar, MonoGame uses the content pipeline to convert your game assets, such as sprite sheets, sound effects and music, into a format that is read into MonoGame at runtime. In the search box on the window that comes up, enter monogame shared. The IntermediateSerailizer takes raw data, such as level maps, and transforms it into an XML document. The content pipeline builder takes the XML file and converts it into an XNB file that can be read at runtime.

The first step is to open a new instance of Visual Studio 2022. Then, in the dialog that comes up, choose the Create a new project button on the right side of the window. Next, select the MonoGame Shared Library option, and click the Next button.

On the following window, enter SummonersTale for the name of the project. Next, choose where you want the project to be created; I chose a folder on my E: drive. Finally, click the Create button to create the project.

Now, we want to add two new projects to the solution. The first is a game library that will hold the classes we will serialize for our game content. This is because the base project does not create a DLL that we can reference in the Content Pipeline Tool. The second will be the actual game project. In future tutorials, I will add projects for other platforms, such as Android. First, we will add the library. Right-click SummonersTale solution in the Solution Explorer, select Add and then New Project. In the window that

pops up, enter MonoGame Game Library in the search box. Select the MonoGame Game Library tile and click Next. In the Name text box, enter Psilibrary, then click Create.

With the library in place, it is time to add the game. Right-click SummonersTale in the solution, select Add and then New Project. In the window that comes up, search for MonoGame Cross-Platform. Click on the Next button. In the following window, enter SummonersTaleGame for the project's name and click on Create.

The next step is to add references to the SummonersTaleGame for the two libraries. Right-click on SummonersTaleGame in the Solution Explorer and select Add Project Reference. Select Psilibrary and click OK. Right-click on SummonersTaleGame again, but this time select Add Shared Project Reference. Select SummonersTale and click OK. The last step is to set the game as the start up project. This can be done using the drop-down in the ribbon in Visual Studio or by right-clicking the SummonersTaleGame and selecting Set a Startup Project. Build and run the game. You should be presented with a window with a blue screen.

The projects are all set up now. It is time to start adding some code! One of the first pieces I add to a game is a state manager to handle the different game states. In the library, there are two pieces. One piece represents a game state. The other piece manages the game states. I will start with the class that represents a game state. Right-click on SummonersTale in the Solution Explorer, select Add and then New Folder. Name this new folder StateManagement. Right-click the StateManagement folder, selectg add and then Class. Name this new class GameState. What I like to do in my tutorials is present code and then explain it. Add the following code to the GameState class.

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

namespace ShadowMonsters
{
    public interface IGameState
    {
        GameState Tag { get; }
    }

    public abstract partial class GameState : DrawableGameComponent, IGameState
    {
        #region Field Region

        protected GameState tag;
        protected readonly IStateManager manager;
        protected ContentManager content;
        protected readonly List<GameComponent> childComponents;

        #endregion

        #region Property Region

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

```
        public GameState Tag
        {
            get { return tag; }
        }

        #endregion

        #region Constructor Region

        public GameState(Game game)
            : base(game)
        {
            tag = this;

            childComponents = new List<GameComponent>();
            content = Game.Content;

            manager =
(IStateManager)Game.Services.GetService(typeof(IStateManager));
        }

        #endregion

        #region Method Region

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

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

            base.Update(gameTime);
        }

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

            foreach (GameComponent component in childComponents)
                if (component is DrawableGameComponent &&
((DrawableGameComponent)component).Visible)
                    ((DrawableGameComponent)component).Draw(gameTime);
        }

        protected internal virtual void StateChanged(object sender, EventArgs e)
        {
            if (manager.CurrentState == tag)
                Show();
            else
                Hide();
        }
```

```
        public virtual void Show()
        {
            Enabled = true;
            Visible = true;

            foreach (GameComponent component in childComponents)
            {
                component.Enabled = true;
                if (component is DrawableGameComponent)
                    ((DrawableGameComponent)component).Visible = true;
            }
        }

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

            foreach (GameComponent component in childComponents)
            {
                component.Enabled = false;
                if (component is DrawableGameComponent)
                    ((DrawableGameComponent)component).Visible = false;
            }
        }

        #endregion
    }
}
```

There is an interface that adds a readonly property Tag. This isn't really used yet, but it will be in the future, so I am including it now. All game states will implement this as it is a parent class so it is an abstract partial class, so we can use polymorphism, which is a big word that means a base class can act as a derived class. You will see this in action a lot in my code.

There are four protected fields in this class: tag, manager, content and childComponents. There are four access modifiers in C#: public, private, protected and internal. Public means that a member can be used anywhere. Private means that it can only be accessed inside of the class it is defined in. Protected means that the member can be used in that class and all classes that inherit from it. Finally, internal objects are accessible inside of a namespace.

The first field is used in the implementation of the interface. You can think of an interface as a contract that the class implementing it must implement. The field will return the instance of the class. The next is a reference to the state manager. Following that is a reference to the content manager of the game. It could be accessed without the field, but I find it handy to access it this way. Finally, there is a list of GameComponents. This will hold and components that the state uses. GameComponents are MonoGame constructs that are similar to the Game class. Like the Game class, they have Initialize, LoadContent, Update, Draw, and other methods. If they are added to the list of GameComponents for the game, their Update and Draw methods will be called automatically. That is, if their Enabled and Visible properties are set to true. For properties, there is one to expose the list of GameComponents and one required by the interface.

The constructor for the class takes a Game parameter. The main class of our Game is Game1 which inherits from the Game class. Again, an example of polymorphism in use. The constructor initializes the fields for the class. The tag field is set to this, the current instance. The childComponents field is initialized to a new List<GameComponent>. With .NET 6, you don't need to use the full type name to initialize an object. You can just use new(). The content field is set to the Content property of the Game object passed in. The state manager is registered as a service for the game. It can be retrieved using The GetService method of the Services property of the Game class. It is just a container that holds services.

There is an override of the LoadContent method that can be used to load shared content. We will cycle back to this later in the series.

The update method loops over all of the child components. If the component is enabled, its Update method is called. This is where in children of the GameState class, their Update methods will be called automatically without having to be explicitly called by the game state.

The Draw method is similar to the Update method. The difference is that it needs to check to see if the component is a DrawableGameComponent that has a Draw method or just a GameComponent that only has an Update method. What could be done here is use pattern matching rather than testing in casting. That is a change I may make in the future. If it is a DrawableGameComponet and Visible its Draw method is called.

There is an event handler, StateChanged, that is called when the state changes. What it does is use the tag field to see if this state is the current state. If this is the current state, the Show method is called. If it is not the current state, the Hide method is called.

The Show method sets the Enabled and Visible properties, which are inherited from the base class, to true. Then in a foreach loop, the Enabled and Visible properties of the child components are set to true. The Hide method works in reverse, setting Enabled and Visible to false.

That is it for this class. It is pretty solid, and I use it in all of my games. And as I said, it is one of the first things I implement in my games. Now, I will turn my attention to the StateManager class. Right click the StateManagement folder in the SummonersTale project, select Add and then Class. Name this new class StateManager; here is the code.

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

namespace SummonersTale.StateManagement
{
    public interface IStateManager
    {
        GameState CurrentState { get; }

        event EventHandler StateChanged;

        void PushTopMost(GameState state);
        void PushState(GameState state);
        void ChangeState(GameState state);
        void PopState();
        void PopTopMost();
```

```
        bool ContainsState(GameState state);
    }

    public class GameStateManager : GameComponent, IStateManager
    {
        #region Field Region

        private readonly Stack<GameState> gameStates = new();

        private const int startDrawOrder = 5000;
        private const int drawOrderInc = 50;
        private const int MaxDrawOrder = 5000;

        private int drawOrder;

        #endregion

        #region Event Handler Region

        public event EventHandler StateChanged;

        #endregion

        #region Property Region

        public GameState CurrentState
        {
            get { return gameStates.Peek(); }
        }

        #endregion

        #region Constructor Region

        public GameStateManager(Game game)
            : base(game)
        {
            Game.Services.AddService(typeof(IStateManager), this);
            drawOrder = startDrawOrder;
        }

        #endregion

        #region Method Region

        public void PushTopMost(GameState state)
        {
            drawOrder += MaxDrawOrder;
            state.DrawOrder = drawOrder;
            gameStates.Push(state);
            Game.Components.Add(state);
            StateChanged += state.StateChanged;
            OnStateChanged();
        }

        public void PushState(GameState state)
        {
            drawOrder += drawOrderInc;
```

```
        state.DrawOrder = drawOrder;
        AddState(state);
        OnStateChanged();
    }

    private void AddState(GameState state)
    {
        gameStates.Push(state);
        if (!Game.Components.Contains(state))
            Game.Components.Add(state);
        StateChanged += state.StateChanged;
    }

    public void PopState()
    {
        if (gameStates.Count != 0)
        {
            RemoveState();
            drawOrder -= drawOrderInc;
            OnStateChanged();
        }
    }

    public void PopTopMost()
    {
        if (gameStates.Count > 0)
        {
            RemoveState();
            drawOrder -= MaxDrawOrder;
            OnStateChanged();
        }
    }

    private void RemoveState()
    {
        GameState state = gameStates.Peek();

        StateChanged -= state.StateChanged;
        Game.Components.Remove(state);
        gameStates.Pop();
    }

    public void ChangeState(GameState state)
    {
        while (gameStates.Count > 0)
        {
            RemoveState();
        }

        drawOrder = startDrawOrder;
        state.DrawOrder = drawOrder;
        drawOrder += drawOrderInc;

        AddState(state);
        OnStateChanged();
    }

    public bool ContainsState(GameState state)
```

```
        {
            return gameStates.Contains(state);
        }

        protected internal virtual void OnStateChanged()
        {
            StateChanged?.Invoke(this, null);
        }

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

        #endregion
    }
}
```

There is an interface that is used for registering the class as a service and retrieving it later on. It has one property, CurrentState, one event, StateChanged, and six methods: PushTopMost, PushState, ChangeState, PopState, PopTopMost, and ContainsState. As the name implies, CurrentState returns the current game state. The event handler fires whenever the current state changes. PushTop and PushTopMost both place a state as the currently active state. ChangeState removes all states currently active and places the state passed in on top. PopState and Push state are similar methods that place a new state as the currently active state. Finally, ContainsState checks to see if a state is in the currently active states.

There is one field and three constants in this class. The field is a stack of GameState that holds the active states. The state on top of the stack is the one that is currently being updated and drawn. As you can see, I just used new() when initializing the stack rather than using the class name. The constants affect the order in which states are drawn. States with a higher draw order are rendered before states with lower draw orders. I start the draw order at a fairly high number, 5000. I will then increment the draw order using the increase constant. Finally, there is a constant that is for pushing a state on that is on top. Instead of pushing with the normal increase, I push with a much larger increase. Next, there is the event that will be fired when the game state changes. And there is the property that returns the state that is currently on top of the stack.

The constructor for the class takes a Game parameter because that is required by GameComponents. It first registers the component as a service. The opposite of what was done in the GameState class. It then initializes the drawOrder field to be the startDrawOrder constant.

The PushTopMost method and PushState methods are nearly identical. The first step is to increase the drawOrder field. The difference is the amount that they are increased. The DrawOrder property of the state is then set to the value of the drawOrder field. For the PushTopMost method I push the state on top of the stack of states. I add it to the list of components of the game, I wire the StateChanged event to the state to the StateChanged method of the GameState passed in. I then call the OnStateChanged method to fire the event. In the PushState method I call a method AddState instead of pushing the stack manually.

The AddState method is similar to what I did in the PushTopMost method. It first pushes the state on top of the stack. It then checks to see that the state is currently not in the list of GameComponents. If it is not, it adds it to the list. It then wires the event handler.

The PopState and PopTopMost are similar as well. In the PopState method I check to make sure that there is a state to pop off the stack. If there is, I call the RemoveState method to pop it off the stack. I decrease the drawOrder field and call the OnStateChanged method to notify other states that is had been popped off. The only difference between the two methods is the amount the drawOrder field is decremented.

The RemoveState method peeks at what state is on top of the stack. It unwires the state's event handler for the change event. It removes the state from the list of GameComponents. Finally, it pops the state off of the stack.

The ChangeState method is similar to the PushState and PushTopMost methods. First, it pops all existing states off of the stack. Next, it resets the drawOrder field, sets the DrawOrder property of the state, and increments the drawOrder field. It calls the AddState method to push the state on the stack. Finally, it calls the OnStateChanged method to notify the states of the change.

The ContainsState method is trivial. It just returns if the state passed in is on the stack. OnStateChanged is also trivial. It invokes the event handler if it is subscribed to.

So far, this is a pretty good foundation. It will allow us to break our game into smaller, more manageable chunks and seamlessly switch to and from them. There is one other core class that I want to add to this tutorial. That is, I want to add a class for measuring frame rates. This is important because it allows you to measure what sort of impact your changes have on the performance of your game. For example, I was working on a game and made a blunder when implementing some code. I caused the frame rate to draw to a crawl. I immediately knew what changes caused the decrease and removed them. Right-click the SummonersTale project, select Add and then Class. Name this new class FramesPerSecond. Here is the code for that class.

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

namespace SummonersTale
{
    public class FramesPerSecond : DrawableGameComponent
    {
        private float _fps;
        private readonly float _updateInterval = 1.0f;
        private float _timeSinceLastUpdate = 0.0f;
        private float _frameCount = 0;
        private float _totalSeconds;
        private float _afps;
        private float _totalFrames;
        private float _maxFps;
        private float _minFps = float.MaxValue;
```

```
public FramesPerSecond(Game game)
    : this(game, false, false, game.TargetElapsedTime)
{
}

public FramesPerSecond(Game game,
    bool synchWithVerticalRetrace,
    bool isFixedTimeStep,
    TimeSpan targetElapsedTime)
    : base(game)
{
    GraphicsDeviceManager graphics =
        (GraphicsDeviceManager)Game.Services.GetService(
            typeof(IGraphicsDeviceManager));

    graphics.SynchronizeWithVerticalRetrace =
        synchWithVerticalRetrace;
    Game.IsFixedTimeStep = isFixedTimeStep;
    Game.TargetElapsedTime = targetElapsedTime;
}

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

public sealed override void Update(GameTime gameTime)
{
    KeyboardState ks = Keyboard.GetState();

    if (ks.IsKeyDown(Keys.F1))
    {
        _fps = 0;
        _afps = 0;
        _timeSinceLastUpdate = 0;
        _totalFrames = 0;
        _frameCount = 0;
        _totalSeconds = 0;
        _minFps = float.MaxValue;
        _maxFps = 0;
    }

    base.Update(gameTime);
}

public sealed override void Draw(GameTime gameTime)
{
    float elapsed = (float)gameTime.ElapsedGameTime.TotalSeconds;

    _frameCount++;
    _timeSinceLastUpdate += elapsed;
    _totalFrames++;

    if (_timeSinceLastUpdate > _updateInterval)
    {
        _totalSeconds++;
        _fps = _frameCount / _timeSinceLastUpdate;
```

```
                if (_fps < _minFps)
                {
                    _minFps = _fps;
                }

                if (_fps > _maxFps)
                {
                    _maxFps = _fps;
                }

                _afps = _totalFrames / _totalSeconds;

                _frameCount = 0;
                _timeSinceLastUpdate -= _updateInterval;
                Debug.WriteLine($"DELTA: {_totalSeconds} FPS: {_fps:N6} - AFPS:
{_afps:N6} - " +
                    $"MIN FPS: {_minFps:N6} - MAX FPS: {_maxFps:N6}");
                Game.Window.Title = $"DELTA: {_totalSeconds} FPS: {_fps:N6} - AFPS:
{_afps:N6} - " +
                    $"MIN FPS: {_minFps:N6} - MAX FPS: {_maxFps:N6}";
            }

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

This class inherits from the DrawableGameComponent class because I want it to update and render automatically. Actually, I don't want it to render. I want it to count the number of times the Draw method of the Game class is called.

There are a number of fields in the class. The first, _fps is the current frames per second. Following that, is _updateInterval and that is the time period for which rendering is calculated, in seconds. After that is the number of frame rendered during the current update interval. Next, _totalSconds measures the number of seconds since the last update. The _afps field measures the average frame rate of the game. _totalFrames is the total number of frames the game has been running. The _maxFps and _minFps measure the maximum number of frames per second and the minimum number of frames per second. _minFps is initialized to the maximum float so that when the first time frames per second it measured it will be less than that value.

There are two constructors for this class. The first takes a single parameter that calls second passing in default values. The second takes as additional parameters the Game object, whether or not to synchronize with vertical refresh rate if we want to use a fixed time step, which would be sixty times per second, and the target elapsed time. It retrieves the current GraphicsDeviceManager similar to how we retrieved the StateManager. It sets the SynchronizeWithVerticalRetrace property to the parameter passed in. Similarly, IsFixedTimeStep andTargetElapsedTime properties are set based on the parameters passed in.

The Update method gets the current state of the keyboard. If the F1 key is down, I reset all of the counters back to zero. Other than _minFps, which is set to float.MaxValue like when the class is initialized.

The Draw method is where the real magic takes place. First, it grabs the amount of time passes since the last update. It increments the number of frames next. The _timeSinceLastUpdate is incremented using the elapsed time and the total number of frames is incremented as well. If total time since the last update is greater than the update interval the following steps are done:

1. _totalSeconds is incremented by one
2. _fps is calculated by taking _frameCount and dividing that by _updateInterval
3. If _fps is less than _minFps, _minFps is assigned _fps
4. If _fps is greater than _maxFps, _maxFps is set to _fps
5. _afps is set to the total number of frames divided by the total number of seconds
6. _frame cound is retured back to zero
7. _timeSinceLastUpdate is decremented by the _updateInterval
8. A string is written to the Debug window and the title of the window is set to the same value

Let's pull it all together in the Game1 class. Since most of the code is new, I will give you the code for the entire class. Replace the existing Game1 class with the following code.

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

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

        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));
            _graphics.ApplyChanges();

            base.Initialize();
        }
```

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

            // TODO: use this.Content to load your game content here
        }

        protected override void Update(GameTime gameTime)
        {
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back ==
ButtonState.Pressed || Keyboard.GetState().IsKeyDown(Keys.Escape))
                Exit();

            // TODO: Add your update logic here

            base.Update(gameTime);
        }

        protected override void Draw(GameTime gameTime)
        {
            GraphicsDevice.Clear(Color.CornflowerBlue);

            // TODO: Add your drawing code here

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

There are using statements to bring various namespaces into scope. Some for the MonoGame framework and others for our libraries. You may be wondering when the MonoGame framework classes are in the namespace Microsoft.Xna. That is because MonoGame is based on a Microsoft product called XNA. It was released to bring game development into the hands of all developers interested. While it was a simple platform, relatively speaking, it was also incredibly powerful. The MonoGame team aimed to make it an open-source project. Eventually, Microsoft abandoned XNA. While it does not have the same following as products such as Unity, it is a very viable product for developing games.

So, I did two things with the fields for this class. First, I added a GameStateManager field for our state manager. I made it and the GraphicsDeviceManager fields readonly because you don't want them to be reassigned once they have been initialized.

In the constructor, I set the width and height of the window to 720p, 1280 by 720 pixels. This will be the window using the PreferredBackBufferWidth and PreferredBackBufferHeight properties. After making a change to the graphics device manager it is best to call ApplyChanges. The last changes that I made were to initialize the GameStateManager and added it to the list of GameComponents.

In the Initialize method, I add a new instance of the FramesPerSecond class to the list of game components. Again, I call ApplyChanges because in the class I change a property of the graphics device manager.

If you build and run the game you will be presented with a blue window with the frame rate information in the title bar and the debug console. I know that I didn't get to anything graphical in this tutorial, and I

might not for a couple more. It is just so important to start with a strong foundation that I focus in on it from the get go.