A Summoner’s Tale
Chapter 8
Multiple Resolutions

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. I’m unfamiliar with Linux, so a few projects may not be done for that platform.

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 show how we will support multiple resolutions. There are a couple of approaches that you can take for this. The one that I am going to use is render target. This render target will have a fixed size, and all calculations will be performed on the resolution. When it comes to rendering, we will create a destination rectangle that fills the window and draw the render target.

So, let’s get started. First, we are going to create render targets and initialize them. I say multiple because a render target will be added to the BaseGameState class. This way, all game states will have access to a render target, and they will be initialized in the base class. Replace the BaseGamePlay 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 BaseGameState : GameState
    {
        #region Field Region

        protected const int TargetWidth = 1280;
        protected const int TargetHeight = 720;

        protected readonly static Random random = new();
        protected readonly SpriteBatch spriteBatch;
        protected readonly RenderTarget2D renderTarget;

        #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));
            renderTarget = new(GraphicsDevice, TargetWidth, TargetHeight);
        }

        #endregion
    }
}
```

There was an addition of two constants: TargetWidth and TargetHeight. They are the targets for our game. The game will be developed using 1280 by 720. This will then be flipped to the window size. In the constructor, I initialize the render target using the overload of the constructor that takes a graphics device, a width and a height.

What I'm going to do is demonstrate how to use the render target. To do that, I updated the Game1 class to use a resolution other than 1280 by 720. Replace the constructor of the Game1 class with the following.

```
public Game1()
{
    _graphics = new GraphicsDeviceManager(this)
    {
        PreferredBackBufferWidth = 1920,
        PreferredBackBufferHeight = 1080
    };
    _graphics.ApplyChanges();
    _manager = new GameStateManager(this);

    Services.AddService(typeof(GraphicsDeviceManager), _graphics);

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

    Components.Add(_manager);
}
```

All I did was change the resolution to HD: 1920 by 1080. After that, you can use any resolution that suits your needs and hardware.

Now, in the TitleState, I am going to render to the render target and flip it to the screen. So, replace the Draw method of the TitleState with the following code.

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

    GraphicsDevice.SetRenderTarget(renderTarget);
    GraphicsDevice.Clear(Color.Black);

    SpriteBatch.Begin();

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

    SpriteBatch.End();

    GraphicsDevice.SetRenderTarget(null);

    SpriteBatch.Begin();

    SpriteBatch.Draw(renderTarget, new Rectangle(new(0,0), new(1920,1080)),
Color.White);

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

The first step is to set that render target. That is done using the SetRenderTarget of the GraphicsDevice. Once it has been set, I call the Clear method passing in black. Next, I use the TargetWidth and TargetHeight constants to position the string. I then render the line as I did before. After rendering to the render target, you need to release it to return rendering back to the screen. You do that by calling the SetRenderTarget method and passing in null. Now that rendering is back on the screen. Finally, I call Begin, draw the render target, and call End.

If you build and run now, you will be presented with a black string with the message at the bottom of the screen. Just as if it was measured at 1920 by 1080 resolution. When it flips to the gameplay screen, you are presented with the map in the upper left corner of the screen and the typical cornflower blue. We fix that by rendering to a render target and presenting that to the screen. Update the Draw method of the GamePlayScreen class to the following.

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

    GraphicsDevice.SetRenderTarget(renderTarget);
    GraphicsDevice.Clear(Color.CornflowerBlue);

    _tileMap.Draw(gameTime, SpriteBatch, _camera, false);

    spriteBatch.Begin(
        SpriteSortMode.Deferred,
        BlendState.AlphaBlend,
        SamplerState.PointClamp,
        null,
        null,
        null,
        _camera.Transformation);

    sprite.Draw(SpriteBatch);
    SpriteBatch.End();

    GraphicsDevice.SetRenderTarget(null);

    SpriteBatch.Begin(SpriteSortMode.Immediate, BlendState.AlphaBlend,
SamplerState.LinearClamp);

    SpriteBatch.Draw(renderTarget, new Rectangle(new(0, 0), new(1920, 1080)),
Color.White);

    SpriteBatch.End();
}
```

What I did was call the SetRenderTarget method passing in our render target. I then call clear passing in the usual cornflower blue. Rendering then takes place as if nothing has changed. As far as the map is concerned, it is rendered on the screen. After rendering the map and sprite has occurred, I call SetRenderTarget passing in null to return rendering back to the screen. I then render the render target to the whole screen.

I made a few changes to the tile engine classes. I added constants to the Engine class for the target width and height. Replace the Engine class with the following code.

```
using Microsoft.Xna.Framework;

namespace Psilibrary.TileEngine
{
    public static class Engine
    {
        #region Field Region
        private static Rectangle viewPortRectangle;

        private static int tileWidth = 32;
        private static int tileHeight = 32;

        private static Camera camera;

        public const int TargetWidth = 1280;
        public const int TargetHeight = 720;

        #endregion

        #region Property Region

        public static int TileWidth
        {
            get { return tileWidth; }
            set { tileWidth = value; }
        }

        public static int TileHeight
        {
            get { return tileHeight; }
            set { tileHeight = value; }
        }

        public static Rectangle ViewportRectangle
        {
            get { return viewPortRectangle; }
            set { viewPortRectangle = value; }
        }


        public static Camera Camera
        {
            get { return camera; }
        }

        #endregion

        #region Constructors

        static Engine()
        {
            ViewportRectangle = new Rectangle(0, 0, 1280, 720);
            camera = new Camera();

            TileWidth = 64;
            TileHeight = 64;
        }

        #endregion

        #region Methods

        public static Point VectorToCell(Vector2 position)
        {
            return new Point((int)position.X / tileWidth, (int)position.Y /
tileHeight);
        }

        public static Vector2 VectorFromOrigin(Vector2 origin)
        {
            return new Vector2((int)origin.X / tileWidth * tileWidth, (int)origin.Y
/ tileHeight * tileHeight);
        }

        public static void Reset(Rectangle rectangle, int x, int y)
        {
            Engine.viewPortRectangle = rectangle;
            Engine.TileWidth = x;
            Engine.TileHeight = y;
        }
        #endregion
    }
}
```

I also made changes to the Camera class to use the constant rather than hard-coded numbers, which is always best practice. Update the Camera class to the following.

```
using Microsoft.Xna.Framework;
using SummonersTale.SpriteClasses;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Psilibrary.TileEngine
{
    public class Camera
    {
        #region Field Region

        Vector2 position;
        float speed;

        #endregion

        #region Property Region

        public Vector2 Position
        {
            get { return position; }
            set { position = value; }
        }

        public float Speed
        {
            get { return speed; }
            set { speed = (float)MathHelper.Clamp(speed, 1f, 16f); }
        }

        public Matrix Transformation
        {
            get
            {
                return Matrix.CreateTranslation(new Vector3(-Position, 0f));
            }
        }

        #endregion

        #region Constructor Region

        public Camera()
        {
            speed = 4f;
        }
        public Camera(Vector2 position)
        {
            speed = 4f;
            Position = position;
        }

        #endregion

        public void LockToSprite(AnimatedSprite sprite, TileMap map)
        {
            position.X = (sprite.Position.X + sprite.Width / 2)
                - (Engine.TargetWidth / 2);

            position.Y = (sprite.Position.Y + sprite.Height / 2)
                - (Engine.TargetHeight / 2);

            LockCamera(map);
        }


        public void LockCamera(TileMap map)
        {
            position.X = MathHelper.Clamp(position.X,
                0,
                map.WidthInPixels - Engine.TargetWidth);

            position.Y = MathHelper.Clamp(position.Y,
                0,
                map.HeightInPixels - Engine.TargetHeight);
        }
    }
}
```

Finally, I made some modifications to the TileMap class. Again, it was to use constants instead of hard-coded numbers. Replace the TileMap with the following code.

```
using Microsoft.Xna.Framework;
using SummonersTale.SpriteClasses;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Psilibrary.TileEngine
{
    public class Camera
    {
        #region Field Region

        Vector2 position;
        float speed;

        #endregion

        #region Property Region
        public Vector2 Position
        {
            get { return position; }
            set { position = value; }
        }

        public float Speed
        {
            get { return speed; }
            set { speed = (float)MathHelper.Clamp(speed, 1f, 16f); }
        }

        public Matrix Transformation
        {
            get
            {
                return Matrix.CreateTranslation(new Vector3(-Position, 0f));
            }
        }

        #endregion

        #region Constructor Region

        public Camera()
        {
            speed = 4f;
        }

        public Camera(Vector2 position)
        {
            speed = 4f;
            Position = position;
        }

        #endregion

        public void LockToSprite(AnimatedSprite sprite, TileMap map)
        {
            position.X = (sprite.Position.X + sprite.Width / 2)
                - (Engine.TargetWidth / 2);

            position.Y = (sprite.Position.Y + sprite.Height / 2)
                - (Engine.TargetHeight / 2);

            LockCamera(map);
        }


        public void LockCamera(TileMap map)
        {
            position.X = MathHelper.Clamp(position.X,
                0,
                map.WidthInPixels - Engine.TargetWidth);

            position.Y = MathHelper.Clamp(position.Y,
                0,
                map.HeightInPixels - Engine.TargetHeight);
        }
    }
}
```

Well, that went faster than I thought it would. It is probably because I tackled it earlier rather than later. Rather than end the tutorial here, what I'm going to do is get started on settings. Typically, I would implement a menu and then a settings screen. All I'm going to do is tackle saving settings and then reading them back in. I will handle the GUI in another tutorial.

What I am going to save is the desired resolution, music volume, and sound volume. I will be saving it into the user's AppData folder under the folder ASummonersTale. Okay, right-click the SummonersTale project in the Solution Explored, select Add and then Class. Name the new class Settings. Here is the code for the Settings class.

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

namespace SummonersTale
{
    public class Settings
    {
        private static float musicVolume = 0.5f;
        private static float soundVolume = 0.5f;
        private static Point resolution = new(1280, 720);

        public static float MusicVolume
        {
            get
            {
                return musicVolume;
            }

            set
            {
                musicVolume = MathHelper.Clamp(value, 0, 1f);
            }
        }

        public static float SoundVolume
        {
            get
            {
                return soundVolume;
            }

            set
            {
                soundVolume = MathHelper.Clamp(value, 0, 1f);
            }
        }

        public static Point Resolution
        {
            get { return resolution; }
            set { resolution = value; }
        }

        public static void Save()
        {
            string path = 
Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData);
            path += "/ASummonersTale/";

            if (!Directory.Exists(path))
            {
                Directory.CreateDirectory(path);
            }

            path += "settings.bin";

            using FileStream stream = new(path, FileMode.Create, FileAccess.Write);
            using BinaryWriter writer = new(stream);

            writer.Write(soundVolume);
            writer.Write(musicVolume);
            writer.Write(resolution.X);
            writer.Write(resolution.Y);
            writer.Close();
            stream.Close();
        }

        public static void Load()
        {
            string path = 
Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData);
            path += @"/ASummonersTale/settings.bin";

            if (!File.Exists(path))
            {
                Save();
            }

            using FileStream stream = new(path, FileMode.Open, FileAccess.Read);
            using BinaryReader reader = new(stream);

            soundVolume = reader.ReadSingle();
            musicVolume = reader.ReadSingle();
            resolution = new(reader.ReadInt32(), reader.ReadInt32());
            reader.Close();
            stream.Close();
        }
    }
}
```

So, what is going on here? There are three fields. One for each of the setting we will be saving. Next, there are properties to expose the fields. For music and sound, they are floating point numbers because in MonoGame volume of music and sound ranges between zero and one. Therefore, I use MathHelper.Clamp method to restrict the values between zero and one. The Save method grabs the location of the application data folder. You use the property Environment.SpecialFolder.ApplicationData to get the location and then Environment.GetFolderPath to convert it to a usable string. I append the suffix that we want to use to that folder. I check to see if the folder exists or not. If it does not exist, I create it. It would be best if this whole method is wrapped in a try-catch block. I then append the file name that we are going to save to. Next, in a using statement it will properly be disposed of when we are done with it, I create a FileStream. I pass in the path, create mode, and write access. I decided to write to a binary file. It is easier to read back, and no additional parsing is required. I then write the sound volume property, music volume property and the values of the resolution. I close the writer and the reader.

Like I did in the Save method, I get the path to the application data folder. I append the desired path. If it does not exist, I call the Save method to save the base settings. I then create a stream using the path, Open mode and Read access. I then create a BinaryReader to read the data. I use the ReadSingle method to read the sound volume and music value. Then, I use ReadInt32 to read the X and Y values of the setting and create a new point. Finally, I close the reader and stream.

When I am going to do next is implement the settings. That will be done in the Game1 class. First, I will do is in the constructor of the Game1 class set the default values I want to use. I will then call Save to save them. I will run the game to set the values. Then, I still stop the game, set it to load the settings and use those. First, change the constructor of the Game1 class to the following.

```
public Game1()
{
    Settings.MusicVolume = .5f;
    Settings.SoundVolume = .5f;
    Settings.Resolution = new(1920, 1080);
    Settings.Save();

    _graphics = new GraphicsDeviceManager(this)
    {
        PreferredBackBufferWidth = Settings.Resolution.X,
        PreferredBackBufferHeight = Settings.Resolution.Y,
    };

    _graphics.ApplyChanges();
    _manager = new GameStateManager(this);

    Services.AddService(typeof(GraphicsDeviceManager), _graphics);

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

    Components.Add(_manager);
}
```

Now, we want to reverse the process and load our settings at runtime. Replace the constructor of the Game1 class with the following.

```
public Game1()
{
    Settings.Load();

    _graphics = new GraphicsDeviceManager(this)
    {
        PreferredBackBufferWidth = Settings.Resolution.X,
        PreferredBackBufferHeight = Settings.Resolution.Y,
    };

    _graphics.ApplyChanges();
    _manager = new GameStateManager(this);

    Services.AddService(typeof(GraphicsDeviceManager), _graphics);

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

    Components.Add(_manager);
}
```

Hmmm, that didn't take as long as I thought would as well. I'm not going to dive any further into 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