Eyes of the Dragon Tutorials 4.0

Part 5

More on Mobile

I'm writing these tutorials for the MonoGame 3.8.1 framework using Visual Studio 2022. The tutorials will make more sense if they are read in order. You can find the list of tutorials on the Eyes of the Dragon 4.0 page of my web blog. I will be making the project available on GitHub here. It will be included on the page that links to the tutorials.

This is part five of a series of tutorials I plan to write on creating a role-playing game with MonoGame. I've worked on similar tutorials using XNA and the past version of MonoGame. In the process of writing more tutorials, I discovered better ways of doing some things and had to go back to fix things. I'm hoping in this series to avoid making those same mistakes. Also, I am going to make the game cross-platform. In my previous tutorials, they were focused on Windows only. I want to open things up for macOS, Linux, Android, iOS, and any platform that MonoGame supports.

This tutorial is going to focus on mobile development. In addition to Android, I am going to add iOS support. Keep in mind that you have to have a Mac to work with iOS. If you don't own a physical Mac, you can use Mac in Cloud. A good package for development is $25 USD a month to access one. It is an option if you don't want to dish out the money for a physical Mac. It offers the required SSH needed by Visual Studio to connect to a Mac.

iOS

I am using my PC for development. So, I need to connect to my MacBook in order to build and test iOS. The process for doing that is in Visual Studio to select Tools -> iOS -> Pair to Mac. This may not work if you are on macOS Ventura. There was a change in the way that macOS works with SSH. You need to make an adjustment to a configuration file: /etc/ssh/sshd_config. Add the following two lines at the end and restart your Mac.

```
HostkeyAlgorithms +ssh-rsa
PubkeyAcceptedAlgorithms +ssh-rsa
```

The other piece of the puzzle is that you need to grant Remote Login to your account on your Mac. On your Mac, press CMD-Space and then enter Remote Login. Grant permission to a group your account belongs to or your actual account.

From here, go back to Visual Studio on your PC. Go to Tools -> iOS -> Pair to Mac and follow the instructions. There is one last piece I did. I found that the remote simulator for iOS would often crash and not be able to start back up again on my PC. So, what I ended up doing was setting things so that the simulator would start up on my Mac. To do that, go to Tools -> Options -> Xamarin -> iOS Settings and deselect Remote Simulator to Windows.

Let's add an iOS project to the solution. If you don't have access to a Mac, skip over until you get to the next section. Right-click the EyesOfTheDragon solution, select Add and then New Project. In the Add New Project window that comes up, search for MonoGame iOS. Select the MonoGame iOS Application and click Next. Name this new project EyesOfTheDragoniOS.

Before you can build and run, there are four pieces of house keeping. First, you need to change the Deployment Target in the Info.plist for the project. MonoGame's project uses 11.2, but the Info.plist uses 11.0. We need to upgrade the Info.plist to 11.2. Double-click the Info.plist to open it. From the drop-down for Deployment Target, select 11.2. Second, you need to reference the libraries. Right-click the EyesOfTheDragoniOS project, select Add and then Project Reference. Select the RpgLibrary project. Before closing, select the Shared Project tab on the left side of the window, then select SharedProject. The next piece is simple. We need to set the project as the startup project. Right-click the EyesOfTheDragoniOS project and select Set as Startup Project.

The final piece is to rename the Game1.cs class and copy over the gameplay state to the iOS project. First, let's copy over the gameplay state. Right-click the EyesOfTheDragonAndroid project, right-click the GameStates folder and select Copy. Right-click the EyesOfTheDragoniOS project and select Paste. Now, right-click the Game1.cs class and select Rename. When prompted, select Yes to rename code elements. The last thing to do is to update the code in the iOS file with the following code.

```
using EyesOfTheDragoniOS.GameStates;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using SharedProject;
using SharedProject.GameScreens;
using SharedProject.GamesScreens;

namespace EyesOfTheDragoniOS
{
    public class iOS : Game
    {
        private GraphicsDeviceManager _graphics;
        private SpriteBatch _spriteBatch;
        public GameStateManager GameStateManager { get; private set; }
        public ITitleState TitleState { get; private set; }
        public IStartMenuState StartMenuState { get; private set; }
        public IGamePlayState GamePlayState { get; private set; }

        public iOS()
        {
            _graphics = new GraphicsDeviceManager(this);

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

            Components.Add(new Xin(this));

            GameStateManager = new GameStateManager(this);
            Components.Add(GameStateManager);
            Services.AddService(typeof(GameStateManager), GameStateManager);
            TitleState = new TitleState(this);
            StartMenuState = new StartMenuState(this);
            GamePlayState = new GamePlayState(this);
        }

        protected override void Initialize()
        {
            // TODO: Add your initialization logic here

            Settings.TargetHeight = _graphics.PreferredBackBufferHeight;
            Settings.TargetWidth = _graphics.PreferredBackBufferWidth;

            base.Initialize();
        }

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

            Services.AddService(typeof(SpriteBatch), _spriteBatch);
            GameStateManager.PushState((TitleState)TitleState);
        }

        protected void FixedUpdate()
        {
            _graphics.PreferredBackBufferWidth = GraphicsAdapter.DefaultAdapter.Cur-
rentDisplayMode.Width;
            _graphics.PreferredBackBufferHeight = GraphicsAdapter.DefaultAdapter.Cur-
rentDisplayMode.Height;
            _graphics.ApplyChanges();
        }

        protected override void Update(GameTime gameTime)
        {
            // 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);
        }
    }
}
```

Pretty much a straight copy and paste from the Android.cs file. The one thing to note is that I removed the code to exit the game. That is not supported on iOS, and you will have an error if you try to use it. If you build and run now, you will get the same results on iOS as on Android and desktop.

Keep on Moving

Now, I'm going to focus on moving. The first thing I will do is update Xin a little. What I want to do is replace the TouchLocation property. I want to change it to return (-1, -1) instead of (0, 0) if there is no touch event. Replace the TouchLocation property with the following version.

```
public static Vector2 TouchLocation
{
    get
    {
        Vector2 result = Vector2.Zero - Vector2.One;

        if (touchLocations.Count > 0)
        {
            if (touchLocations[0].State == TouchLocationState.Pressed ||
                touchLocations[0].State == TouchLocationState.Moved)
            {
                result = touchLocations[0].Position;
            }
        }

        return result;
    }
}
```

The thing is, in order to move, we need something to move. For that reason, I will be introducing sprites. So, what is a sprite? A sprite is basically a small(?) image that is drawn on the screen. We will be using an animated sprite, which is a sprite that is drawn repeatedly where each time it is drawn, a different image is drawn, much like a cartoon.

To start, we need to add a new folder for our sprite classes. Right-click the SharedProject folder, select Add and then New Folder. Name this new folder SpriteClasses. Now, I will start adding classes. The first class that I'm going to add is an abstract class that all sprites will inherit from. Right-click the SpriteClasses folder, select Add and then Class. Name this new class Sprite. Here is the code for that class.

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

namespace SharedProject.Sprites
{
    public enum Facing { Up, Down, Left, Right }

    public abstract class Sprite
    {
        protected float _speed;
        protected Vector2 _velocity;

        public Facing Facing { get; set; }
        public string Name { get; set; }
        public Vector2 Position { get; set; }
        public Vector2 Size { get; set; }
        public Point Tile { get; set; }

        public int Width { get; set; }
        public float Height { get; set; }
        public float Speed
        {
            get { return _speed; }
            set { _speed = MathHelper.Clamp(_speed, 1.0f, 400.0f); }
        }

        public Vector2 Velocity
        {
            get { return _velocity; }
            set { _velocity = value; }
        }

        public Vector2 Center
        {
            get { return Position + new Vector2(Width / 2, Height / 2); }
        }

        public Vector2 Origin
        {
            get { return new Vector2(Width / 2, Height / 2); }
        }

        public Sprite()
        {

        }

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

There is an enumeration here, Facing. This defines the direction that the sprite is facing. The class has two protected fields: _speed and _velocity. The _speed field is how many pixels the sprite travels per second. The _velocity field defines what direction the sprite is travelling. There are some properties in the class. Facing is what direction the sprite is facing. Name is the name of the sprite. Position is the position of the sprite. Size is the size of the sprite. Tile is which tile the sprite is in. Width and Height are the width and height of the sprite. Speed exposes the speed of the sprite. Velocity exposed the velocity of the sprite. The Center is the center of the sprite on the map. Finally, Origin is the center of the sprite. There is a base constructor that I may expand in the future. Finally, there are two abstract methods, Update and Draw, that classes inheriting from this class must implement.

The animation type that I will be implementing is frame animation. I mean that animation is a series of frames, much like animation in a cartoon. The frames are displayed in rapid succession. You start with the first frame, render it, move on to the second, render that frame and continue until you get to the last frame. When you get to the last frame, you go back to the first frame.

Before I get to the animated sprite, I need a class for animations. Right-click the SpriteClasses folder, select Add and then Class. Name this new class Animation. This is the code for that class.

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

namespace SharedProject.Sprites
{
    public class Animation
    {
        #region Field Region

        readonly Rectangle[] frames;
        int framesPerSecond;
        TimeSpan frameLength;
        TimeSpan frameTimer;
        int currentFrame;
        int frameWidth;
        int frameHeight;

        #endregion

        #region Property Region

        public int FramesPerSecond
        {
            get { return framesPerSecond; }
            set
            {
                if (value < 1)
                    framesPerSecond = 1;
                else if (value > 60)
                    framesPerSecond = 60;
                else
                    framesPerSecond = value;
                frameLength = TimeSpan.FromSeconds(1 / (double)framesPerSecond);
            }
        }

        public Rectangle CurrentFrameRect
        {
            get { return frames[currentFrame]; }
        }

        public int CurrentFrame
        {
            get { return currentFrame; }
            set
            {
                currentFrame = (int)MathHelper.Clamp(value, 0, frames.Length - 1);
            }
        }

        public int FrameWidth
        {
            get { return frameWidth; }
        }

        public int FrameHeight
        {
            get { return frameHeight; }
        }
        #endregion

        #region Constructor Region

        public Animation(int frameCount, int frameWidth, int frameHeight, int xOffset,
int yOffset)
        {
            frames = new Rectangle[frameCount];
            this.frameWidth = frameWidth;
            this.frameHeight = frameHeight;

            for (int i = 0; i < frameCount; i++)
            {
                frames[i] = new Rectangle(
                        xOffset + (frameWidth * i),
                        yOffset,
                        frameWidth,
                        frameHeight);
            }
            FramesPerSecond = 5;
            Reset();
        }

        private Animation(Animation animation)
        {
            this.frames = animation.frames;
            FramesPerSecond = 5;
        }

        #endregion

        #region Method Region

        public void Update(GameTime gameTime)
        {
            frameTimer += gameTime.ElapsedGameTime;

            if (frameTimer >= frameLength)
            {
                frameTimer = TimeSpan.Zero;
                currentFrame = (currentFrame + 1) % frames.Length;
            }
        }

        public void Reset()
        {
            currentFrame = 0;
            frameTimer = TimeSpan.Zero;
        }

        #endregion

        #region Interface Method Region

        public object Clone()
        {
            Animation animationClone = new(this)
            {
                frameWidth = this.frameWidth,
                frameHeight = this.frameHeight
            };

            animationClone.Reset();

            return animationClone;
        }

        #endregion
    }
}
```

There are some fields in the class. The first is an array of rectangles which describe the frames for the animation. The framesPerSecond field is how many frames should be rendered. I've found between 5 and 8 to be the best values. There is a TimeSpan which defines the length of a frame, and a TimeSpan field which times how long since the last frame change. There are also fields for the height and width of the frames.

There are properties to expose the framesPerSecond field. I probably could have gotten away with using the Clamp method of MathHelper instead of using an if statement in the set part. After setting the field, I calculate the frame length by using the FromSeconds method of the TimeSpan class. There is a method to return the current rectangle as well as the current frame. Setting the current frame does use the Clamp method to clamp it in the valid range. There are then properties to expose the frame width and frame height.

The constructor takes as parameters the frame count, width and height. It also takes the x offset and y offset. These are used for sprite sheets where there are multiple animations on the same image. This animation assumes that the animations are in rows, not columns. After initializing the fields, I create the source rectangles. The Y coordinate is the easy part. It is just the y offset. The X coordinate is the current frame times the frame width plus the X offset. The width and height are the width and height passed in. I initialize the frame rate to 5 and call the Reset method that resets an animation back to zero. There is a second constructor that is used to clone an animation.

The Update method is where the magic takes place. The first step is to increase the frame timer by the elapsed game time. Following that, you check to see if the frame timer is greater than the frame length. If it is, set the frame timer back to zero. Next, increase the frame timer by one and then take the remainder of that calculation. The has the frames ranging between zero and the number of frames due to the way remainders work. The Reset method just resets the current frame to zero and the frame time to zero as well.

Initially, this class implemented the IClonable interface, which is why there is a region called Interface Method Region. It is easier to make a clone of a class than to initialize one sometimes. This one creates a new instance of the Animation class using the second constructor. It then sets the frameWidth and frameHeight fields. Finally, it calls the Reset method to reset the animation.

Now it is time to implement the animated sprite class. Right-click the SpriteClasses folder, select Add and then Class. Name this new class AnimatedSprite. The code for that class follows next.

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

namespace SharedProject.Sprites
{
    public class AnimatedSprite : Sprite
    {
        #region Field Region

        readonly Dictionary<string, Animation> animations;
        string currentAnimation;
        bool isAnimating;
        readonly Texture2D texture;

        #endregion

        #region Property Region

        public bool IsActive { get; set; }

        public string CurrentAnimation
        {
            get { return currentAnimation; }
            set { currentAnimation = value; }
        }

        public bool IsAnimating
        {
            get { return isAnimating; }
            set { isAnimating = value; }
        }

        #endregion

        #region Constructor Region

        public AnimatedSprite(Texture2D sprite, Dictionary<string, Animation> animation)
        {
            texture = sprite;
            animations = new();

            foreach (string key in animation.Keys)
                animations.Add(key, (Animation)animation[key].Clone());
        }

        #endregion

        #region Method Region

        public void ResetAnimation()
        {
            animations[currentAnimation].Reset();
        }

        public override void Update(GameTime gameTime)
        {
            if (isAnimating)
                animations[currentAnimation].Update(gameTime);
        }

        public override void Draw(SpriteBatch spriteBatch)
        {
            spriteBatch.Draw(
                texture,
                new Rectangle(
                    (int)Position.X,
                    (int)Position.Y,
                    Engine.TileWidth,
                    Engine.TileHeight),
                animations[currentAnimation].CurrentFrameRect,
                Color.White);
        }

        public bool LockToMap(Point mapSize, ref Vector2 motion)
        {
            Position = new(
                MathHelper.Clamp(Position.X, 0, mapSize.X - Width),
                MathHelper.Clamp(Position.Y, 0, mapSize.Y - Height));

            if (Position.X == 0 && motion.X < 0 || Position.Y == 0 && motion.Y < 0)
            {
                motion = Vector2.Zero;
                return false;
            }

            if (Position.X == mapSize.X - Width && motion.X > 0)
            {
                motion = Vector2.Zero;
                return false;
            }

            if (Position.Y == mapSize.Y - Width && motion.Y > 0)
            {
                motion = Vector2.Zero;
                return false;
            }

            return true;
        }

        #endregion
    }
}
```

So, there are a few assumptions about animated sprites. One is that all of the animations are on a single image. The other is that all the frames for a single image are on the same row. There is a Dictionary with a key type of string and a value of Animation. There is a string for what the current animation is and a bool for if the sprite is animating or not. Next there is a Texture2D for the image of the sprite. In another tutorial, I will demonstrate how to work with multiple images. There are properties to expose the active animation, if the sprite is active or not, and if it is animating.

The constructor takes a Texture2D and Dictionary argument. It sets the texture field to the value passed in. Then, it creates a new Dictionary. In a foreach loop, I loop over all of the keys in the animation Dictionary passed in. For each animation, I create a clone of it and add it to the sprite's animations. There is a simple method that I will mention, ResetAnimation. What this method does is call the Reset method of the current animation.

Because this method inherits from Sprite, it must implement an Update and Draw method. All the Update method does is check to see if the sprite is animating and if it is, call the current animation's Update method.

The Draw method assumes that it is called between Begin and End of the sprite batch object. It uses the overload that takes a texture, a destination rectangle and a tint colour. For the destination rectangle, I cast the position to an integer and used the height and width of the engine for those values.

So, things are progressing well. The next step is to get the sprite onto the map. For that, we will need a sprite. I will use the eight from the previous Eyes of the Dragon tutorials. You can download them from this link. Once you have downloaded and extracted them, they need to be added as content. To do that, open the Content folder in the SharedProject project and double-click the Content.mgcb to open the MGCB Editor. Right-click the root node, select Add and then New Folder. Name this new folder PlayerSprites. Now, right-click that folder, select Add and then Existing Item. Navigate to the location of the sprites and select all eight of them and add them to the folder. When prompted, select to copy them and select to perform the action for all items.

Speaking of content, there is a little bug that we need to fix. For iOS content items are not copied to the output folder when debugging. So, when you run and try to load content, you will get a content file not found exception. The fix is to update the file:

C:\Users\{user name}\.nuget\packages\monogame.content.builder.task\3.8.1.303\build\MonoGame.Content.Builder.Task.targets.

In the first PropertyGroup, you need to add an item to include content. Update that PropertyGroup as follows.

```
<PropertyGroup>
  <DisableFastUpToDateCheck>true</DisableFastUpToDateCheck>
     <CollectBundleResourcesDependsOn> IncludeContent; </CollectBundleResourcesDependsOn>
</PropertyGroup>
```

Turns out I was mistaken. We can have a TileMap in the SharedProject project. What we can't do is build a TileMap in a shared project. So, I'm going to move the GamePlayState back to the SharedProject. What I did was delete the GamePlayState in the game projects and add a new one to the shared project. So, in each of the game projects, right-click the GameStates folder and select Delete. Now, this is going to cause a bit of upheaval, sheer bedlam, to be honest. There is just a problem with using statements in the games, and the fact that GamePlayState no longer exists, having been harshly wiped from existence. In the GamePlayState for the Android project, delete these two lines.

```
using Android.Views;
using EyesOfTheDragonAndroid.GameStates;
```

In the iOS project, delete the following line.

```
using EyesOfTheDragoniOS.GameStates;
```

Finally, from the Desktop game, delete the following line.

```
using EyesOfTheDragon.GameStates;
```

Now, let’s add back the game state with a plus. We will add an animated sprite. In the GameScreens folder, add a new class called GamePlayState. Here is the code for that class.

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Rpglibrary.TileEngine;
using RpgLibrary.TileEngine;
using SharedProject;
using SharedProject.GamesScreens;
using SharedProject.Sprites;
using System;
using System.Collections.Generic;
using System.Security.Cryptography;
using System.Text;

namespace SharedProject.GameScreens
{
    public class GamePlayState : GameState, IGamePlayState
    {
        readonly Camera camera;
        TileMap map;
        readonly Engine engine;
        RenderTarget2D renderTarget;
        AnimatedSprite sprite;

        public GamePlayState(Game game) : base(game)
        {
            Game.Services.AddService<IGamePlayState>(this);
            camera = new(Settings.BaseRectangle);
            engine = new(32, 32, Settings.BaseRectangle);
        }

        public GameState Tag => this;

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

            renderTarget = new(GraphicsDevice, Settings.BaseWidth, Settings.BaseHeight);

            Texture2D texture = Game.Content.Load<Texture2D>(@"Tiles/tileset1");

            List<Tileset> tilesets = new()
            {
                new(texture, 8, 8, 32, 32),
            };

            TileLayer layer = new(100, 100);

            map = new("test", tilesets[0], layer);

            Dictionary<string, Animation> animations = new();
            Animation animation = new(3, 32, 32, 0, 0) { CurrentFrame = 0, FramesPerSec-
ond = 8 };
            animations.Add("walkdown", animation);

            animation = new(3, 32, 32, 0, 32) { CurrentFrame = 0, FramesPerSecond = 8 };
            animations.Add("walkleft", animation);

            animation = new(3, 32, 32, 0, 64) { CurrentFrame = 0, FramesPerSecond = 8 };
            animations.Add("walkright", animation);

            animation = new(3, 32, 32, 0, 96) { CurrentFrame = 0, FramesPerSecond = 8 };
            animations.Add("walkup", animation);

            texture = Game.Content.Load<Texture2D>(@"PlayerSprites/femalepriest");

            sprite = new(texture, animations)
            {
                CurrentAnimation = "walkdown",
                IsActive = true,
                IsAnimating = true,
            };
        }

        public override void Update(GameTime gameTime)
        {
            map.Update(gameTime);
            sprite.Update(gameTime);

            base.Update(gameTime);
        }

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

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

            SpriteBatch.Begin(SpriteSortMode.Immediate,
                              BlendState.AlphaBlend,
                              SamplerState.PointClamp,
                              null,
                              null,
                              null,
                              Matrix.Identity);

            map.Draw(gameTime, SpriteBatch, camera);
            sprite.Draw(SpriteBatch);

            SpriteBatch.End();

            GraphicsDevice.SetRenderTarget(null);

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

            SpriteBatch.Draw(renderTarget, Settings.TargetRectangle, Color.White);

            SpriteBatch.End();
            }
        }
    }
```

In the new code that you haven’t seen before is the addition of an AnimatedSprite field. In the LoadContent method, after creating an empty map, I create a Dictionary<string, Animation> to hold the animations for the sprite. There are four animations, one on each row. Each row is made up of three images. So, for the parameters, I pass in 3 for the number of frames, 32 for the height and width of the frame and the offset. The X-offset is consistent at zero. The Y-offset starts and zero and increases by 32 in each row. Once a row is constructed, it is added to the dictionary. I then load the female priest texture. I create the sprite, initializing the animation to walk down, IsActive and IsAnimating to true. In the Update method, I call the Update method of the sprite. Similarly, I call the Draw method of the sprite in the Draw method, after drawing the map, or it would not show up.

If you build and run now, you will be prompted to tap to move on to the next screen. Once there, you will find the tile map with the sprite on top. I wanted to cover movement in this tutorial. Just getting the game up and running on all three platforms took more time and effort than I expected. So, I think that I am going to end this here. There is more that I could pack in, but I will save it for the following tutorial. I don't want you to have too much to digest at once. I encourage you to visit the news page of my site, https://cynthiamcmahon.ca/blog/, for the latest news on my tutorials.

Good luck with your game programming adventures!
Cynthia