Eyes of the Dragon Tutorials 4.0

Part 9

Who's That Girl?

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 nine 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 MonoGame supports.

This tutorial will add a non-player character to the game, a girl. Her name is Rio, and she dances on the sand. We have a bit of the chicken and the egg situation going on. The shared project needs to know about characters. The RpgLibrary needs to know about characters as well. The SharedProject gets full access to the classes in the RpgLibrary. The RpgLibrary, on the other hand, has no access to the SharedProject. So, how are we ever going to resolve this? With one very simple, yet very powerful, word: Interfaces. In the RPG Library, we are going to create an interface that holds the absolute bare minimum it requires. Okay, maybe a little bit more than the bare minimum. In the SharedProject, we implement the interface in any needed classes. The interface allows the sharing of member data. Well, how does that exactly work, Cynthia? Interfaces are only contracts. Yes, they are contracts, but they can act like fields. It will be so much clearer when you see it in action.

With the main theory lesson done, let's dig into the tutorial. As I said, the RpgLibrary needs an interface to access the details it needs to know about characters. So, I added an interface called ICharacter. First, I created a folder. Then, right-click the RpgLibrary project, select Add and then New Folder, and name this new folder Characters. Next, right-click the Characters folder, select Add and then New Item. Finally, navigate the list to Interface, and call the interface ICharacter. Here is the code for that interface.

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

namespace RpgLibrary.Characters
{
    public interface ICharacter
    {
        string Name { get; }
        bool Enabled { get; set; }
        bool Visible { get; set; }
        Vector2 Position { get; set; }
        Point Tile { get; set; }
        void Update(GameTime gameTime);
        void Draw(SpriteBatch spriteBatch);
    }
}
```

Okay, not that scar, huh? It is just a collection of properties and methods that must be implemented by any class that implements it. There is a get-only property for the name of the character. That doesn't mean that the class implementing it has to do that. It is just the minimum requirement. There are Enabled and Visible properties. A position in pixels and a position in tiles. Then comes the two methods, Update and Draw.

The next step is to add a character layer. It will hold all of the characters on the map. Right-click the TileEngine folder in the RpgLibrary, select Add and then Class. Name this new class CharacterLayer. Here is the code for that class.

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using RpgLibrary.Characters;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace RpgLibrary.TileEngine
{
    public class CharacterLayer : ILayer
    {
        public List<ICharacter> Characters { get; private set; } = new();

        public CharacterLayer() { }

        public void Update(GameTime gameTime)
        {
            foreach (var character in Characters.Where(x => x.Enabled) )
            {
                character.Update(gameTime);
            }
        }

        public void Draw(SpriteBatch spriteBatch, Camera camera, List<Tileset> tilesets)
        {
            foreach (var character in Characters.Where(x => x.Visible))
            {
                character.Draw(spriteBatch);
            }
        }
    }
}
```

Nothing big or scary here as well. The class implements the ILayer interface, so it has an Update and Draw method. There is a List<ICharacter> that will hold our characters. There is an explicit constructor. The Update method loops over the Enabled property in a LINQ Where clause. It does the same in the Draw method using the Visible property. So, as long as our characters implement the interface, they have Draw and Update methods that we can call, and they can be rendered on the map. Pretty neat, huh?

Wow. We are just speeding through this tutorial. I need to find something big and scary. I know! We can add the new layer to the map. That has to be done in the TileMap class, right? Nope, it is done in the SharedProject. That is because, again, we used interfaces. We can use the AddLayer method of the TileMap class to add a layer from anywhere. However, we need to create a class in the SharedProject to describe our characters. Right-click the SharedProject project, select Add and then Class. Name this new class Character. Here is the code for that class.

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using RpgLibrary.Characters;
using SharedProject.Sprites;
using System;
using System.Collections.Generic;
using System.Text;

namespace SharedProject
{
    public class Character : ICharacter
    {
        private string _name;
        private AnimatedSprite _sprite;
        private string _spriteName;

        public string Name => _name;

        public bool Enabled { get; set; }
        public bool Visible { get; set; }
        public Vector2 Position { get; set; }
        public Point Tile { get; set; }

        private Character()
        {
            Enabled = true;
            Visible = true;
            Position = new();
            Tile = new();
        }

        public Character(string name, AnimatedSprite animatedSprite, string spriteName)
        {
            _name = name;
            _sprite = animatedSprite;
            _spriteName = spriteName;
        }

        public void Draw(SpriteBatch spriteBatch)
        {
            _sprite.Draw(spriteBatch);
        }
        public void Update(GameTime gameTime)
        {
            _sprite.Position= Position;
            _sprite.Update(gameTime);
        }
    }
}
```

I added a few fields to the class. There is one for the name, an animated sprite, and a sprite name. There are then the properties that we talked about earlier. I added a private constructor that takes no parameters. It initializes the properties. There is a second constructor that takes a name, sprite, and sprite name. It assigns the properties to the values passed in. The Draw method calls the Draw method of the sprite. I needed to assign the Position property of the sprite as the Position property of the character to get it where I wanted it.

I did have to make an adjustment to the TileMap class. Currently, we only draw tile layers. We need to adjust it to draw CharacterLayers as well. It is just a simple matter of adding an else clause to the if. Replace the Draw method of the TileMap class with the following version.

```
public void Draw(GameTime gameTime, SpriteBatch spriteBatch, Camera camera)
{
    foreach (ILayer layer in _mapLayers)
    {
        if (layer is TileLayer layer1)
        {
            layer1.Draw(spriteBatch, camera, _tilesets);
        }
        else if (layer is CharacterLayer characters)
        {
            characters.Draw(spriteBatch, camera, _tilesets);
        }
    }
}
```

Nothing big and scary there either. The last change to get the characters on the screen is to create a character layer and add it to the map. You do that in the LoadContent method of the GamePlayState. Replace that method with the following version.

```
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, FramesPerSecond = 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,
};

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

AnimatedSprite rio = new(texture, animations)
{
    CurrentAnimation = "walkdown",
    IsAnimating = true,
};

CharacterLayer chars = new();

chars.Characters.Add(
    new Character("Rio", rio, "femalefighter")
    {
        Position = new(320, 320),
        Tile = new(10, 10),
        Visible= true,
        Enabled=true,
    });

map.AddLayer(chars);

rightButton = new(Game.Content.Load<Texture2D>("GUI/g21245"), ButtonRole.Menu)
{
    Position = new(80, Settings.BaseHeight - 80),
    Size = new(32, 32),
    Text = "",
    Color = Color.White,
};

rightButton.Down += RightButton_Down;
ControlManager.Add(rightButton);

upButton = new(Game.Content.Load<Texture2D>("GUI/g21263"), ButtonRole.Menu)
{
    Position = new(48, Settings.BaseHeight - 48 - 64),
    Size = new(32, 32),
        Text = "",
        Color = Color.White,
    };

    upButton.Down += UpButton_Down;
    ControlManager.Add(upButton);

    downButton = new(Game.Content.Load<Texture2D>("GUI/g21272"), ButtonRole.Menu)
    {
        Position = new(48, Settings.BaseHeight - 48),
        Size = new(32, 32),
        Text = "",
        Color = Color.White,
    };

    downButton.Down += DownButton_Down;
    ControlManager.Add(downButton);

    leftButton = new(Game.Content.Load<Texture2D>("GUI/g22987"), ButtonRole.Menu)
    {
        Position = new(16, Settings.BaseHeight - 80),
        Size = new(32, 32),
        Text = "",
        Color = Color.White,
    };

    leftButton.Down += LeftButton_Down;

    ControlManager.Add(leftButton);
}
```

What has changed here? Well, after creating the player's sprite, I load the female fighter texture. Then, I created an animated sprite for the character walking down and is animating to show that the layer is updating. Next, I create a new character. I pass in the name of the character, the sprite, and the name of the texture. I position it in tile (10, 10) and at position (320, 320). It is also Visible and Enabled. I then add the layer to the map.

If you build and run now, when you get to the game, you will see a second sprite on the map that is animating. Pretty cool, huh? Well, there is one problem. You can walk on top of her. Not cool! Let's fix that. Replace the Update method with the following version.

```
public override void Update(GameTime gameTime)
{
    ControlManager.Update(gameTime);

    sprite.Update(gameTime);
    map.Update(gameTime);

    if (Xin.IsKeyDown(Microsoft.Xna.Framework.Input.Keys.A) && !inMotion)
    {
        MoveLeft();
    }
    else if (Xin.IsKeyDown(Microsoft.Xna.Framework.Input.Keys.D) && !inMotion)
    {
        MoveRight();
    }
    if (Xin.IsKeyDown(Microsoft.Xna.Framework.Input.Keys.W) && !inMotion)
    {
        MoveUp();
    }
    else if (Xin.IsKeyDown(Microsoft.Xna.Framework.Input.Keys.S) && !inMotion)
    {
        MoveDown();
    }

    if (motion != Vector2.Zero)
    {
        motion.Normalize();
    }
    else
    {
        inMotion = false;
        return;
    }

    if (!sprite.LockToMap(new(99 * Engine.TileWidth, 99 * Engine.TileHeight), ref mo-
tion))
    {
        inMotion = false;
        return;
    }

    Vector2 newPosition = sprite.Position + motion * speed * (float)ga-
meTime.ElapsedGameTime.TotalSeconds;

    Rectangle nextPotition = new Rectangle(
        (int)newPosition.X,
        (int)newPosition.Y,
        Engine.TileWidth,
        Engine.TileHeight);

    if (nextPotition.Intersects(collision))
    {
        inMotion = false;
        motion = Vector2.Zero;
        sprite.Position = new((int)sprite.Position.X, (int)sprite.Position.Y);
        return;
    }

    if (map.PlayerCollides(nextPotition))
    {
        inMotion = false;
        motion = Vector2.Zero;
        return;
    }

    sprite.Position = newPosition;
    sprite.Tile = Engine.VectorToCell(newPosition);

    camera.LockToSprite(sprite, map);

    base.Update(gameTime);
}
```

What has changed here? Well, in an if statement, I call a new method PlayerCollides on the tile map that will check to see if there is a collision between the player and the characters. If there is, I set inMotion to false, the motion vector to the zero vector and exit the method. That leads us to the PlayerCollides method. Add this method to the TileMap class.

```
public bool PlayerCollides(Rectangle nextPotition)
{
    CharacterLayer layer = _mapLayers.Where(x => x is CharacterLayer).FirstOrDefault() as 
CharacterLayer;

    if (layer != null)
    {
        foreach (var character in layer.Characters)
        {
            Rectangle rectangle = new(
                new(character.Tile.X * Engine.TileWidth, character.Tile.Y * 
EngineH.TileHeight),
                new(Engine.TileWidth, Engine.TileHeight));
            if (rectangle.Intersects(nextPotition))
            {
                return true;
            }
        }
    }

    return false;
}
```

The first step is to get the character layer. I do that using a Where clause on the map layers, using the FirstOrDefault clause and casting it to a CharacterLayer. If that is not null, I cycle over all of the characters on the map. I then create a rectangle that represents the tile the character is on. If that intersects with the tile the player is trying to enter, I return true. If no collision is found, I return false.

That is it for getting a character on the map and collision detection between the player and the characters.

So, this has been a quick tutorial, but I am going to end it here. I don’t want to start a new topic at this point. 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. Also, I’m thinking of reviving my newsletter so you will be informed of new stuff rather than having to keep looking for new things.

Good luck with your game programming adventures!
Cynthia