Eyes of the Dragon Tutorials 4.0

Part 4

Tiles! Tiles! Tiles!

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 four 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 the MonoGame supports.

This tutorial is going to focus on the tile engine. It is my absolute favourite thing about game programming. I don't know what fascinates me so much about them. They are just so cool! There are a million different ways to do them. I will be using my tile engine from Silver Prophet, which is based on my original Eyes of the Dragon tutorial series.

Before we get to code, we need some tiles. I will be using some tiles from OpenGameArt.org. They are an excellent place for free game assets. You can download the tiles from my blog at the following link: https://cynthiamcmahon.ca/blog/downloads/tilesets.zip

After you have downloaded and extracted the tiles, they need to be added to the project. In the EyesOfTheDragon project, open the Content folder and double-click on the Content.mgcb node to open the MGCB Editor. Right-click on the root name, select Add and then New Folder. Name this new folder Tiles. Right-click the Tiles folder, select Add and then Existing Item. Navigate to the tile set and add it to the tile set. Save the project and exit the MCGB Editor.

There is one piece of housekeeping that needs to be addressed. In the previous tutorial, I missed adding a reference to the RpgLibrary to the Android project. I might have and just missed it in my project. In any event, let's just double-check to be sure. Right-click the EyesOfTheDragonAndroid project, and select Add Reference. If it is not there already, add the RpgLibrary project.

Before writing some code, let's talk a little about theory. What is a tile engine, and why would you want to use one? Consider if you want to build a map that is ten thousand pixels by ten thousand pixels. That is a total of one hundred million pixels. Each pixel is four bytes, so you are using four hundred million bytes of memory for the map. What the tile engine does is break down a map into smaller chunks of repeatable pieces or tiles. These tiles are drawn to the window rather than the large image. It is more efficient and uses less memory.

So, let's write some code already! I will be using a layered approach to the tile engine. We can have any number of layers and layer types. All layers will implement an interface. This allows us to group all layers into a single collection and use polymorphism to iterate over all of the layers. What was that word? Polywhatsism? That is a fancy word that amounts to a derived class acting as a member of a base class. Still not clear? I will try and explain more when it comes time to put it into effect. I promise you it is not that big and scary once you see it in action. I will also give a couple of examples.

To begin with, let's add the interface. Right-click the RpgLibrary project, select Add and then New Folder. Name this new folder TileEngine. Now, right-click the TileEngine folder, select Add and then New Item. From the list that comes up, select Interface. Name this new interface ILayer. Here is the code for the ILayer interface. Here is the code for that interface.

```
using System.Collections.Generic;

using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

namespace RpgLibrary.TileEngine
{
    public interface ILayer
    {
        void Update(GameTime gameTime);
        void Draw(SpriteBatch spriteBatch, Camera camera, List<Tileset> tilesets);
    }
}
```

Uh-oh! We have two missing pieces. There is no Camera class and no Tileset class. That's fine. We will add them shortly. Interfaces are not like classes. They are a contract that the class implementing them must fulfill. In our case, any class that implements the ILayer interface must implement an Update method that takes a GameTime parameter and a Draw method that takes a SpriteBatch, Camera, and Tileset object. This allows us to update and render any type of layer using polymorphism. There is that word again. Let's give an example.

Suppose you are trying to model animals. You have a base class Animal with the following code.

```
    public abstract class Animal
    {
        public Animal()
        {

        }

        public abstract string Speak();
    }
```

This is an abstract class that an instance that cannot be created directly. You must inherit the class and implement the abstract methods. In this trivial class, we have an animal that has a speak method. Let's add a couple other classes, a Cat and a Dog that inherit from Animal and create the abstract method.

```
    public class Cat : Animal
    {
        public override string Speak()
        {
            return "Meow! Meow!";
        }
    }

    public class Dog : Animal
    {
        public override string Speak()
        {
            return "Bark! Bark!";
        }
    }
```

Ah, so I understand the idea of inheritance. I don't see what is so special about it. What is special is that instead of creating a Dog or a Cat, we can create animals as the Animal class. This allows us to group them together in a collection and call their methods directly, and C# will pick the right class. For example, if we had a List<Animal> and added a Dog and Cat and iterated through the list, for the first object, the dog, the "Bark! Bark!" string would be returned, and for the second object, the cat, "Meow! Meow!" would be returned. See the following complete sample.

We will start with the animal classes. I have three of them a cat, a dog and a goldfish. Here is what the final code looks like.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp1
{
    public abstract class Animal
    {
        public abstract string Speak();
    }

    public class Cat : Animal
    {
        public override string Speak()
        {
            return "Meow! Meow!";
        }
    }

    public class Dog : Animal
    {
        public override string Speak()
        {
            return "Bark! Bark!";
        }
    }

    public class Goldfish : Animal
```

```
    {
        public override string Speak()
        {
            return "Glub! Glub!";
        }
    }
}
```

Now, let's create some animals and add them to a List<Animal>, iterate over the collection and print their speak to the console.

```
List<Animal> animals = new()
{
    new Cat(),
    new Goldfish(),
    new Dog()
};

foreach (Animal animal in animals) Console.WriteLine(animal.Speak());
```

And that is polymorphism in action. A variable created as a derived class is acting as a member of the base class is having its method called at run time. First, the cat will write, Meow! Meow! Then, the goldfish will write Glub! Glub! Finally, the dog will write Bark! Bark! All of them are assigned to the Animal base class.

Circling back to the tile engine, we need to add a few classes. The first is a class that represents the engine itself. Right-click the TileEngine folder in the RpgLibrary project, select Add and then Class. Name this class Engine. This is the code for the Engine class.

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

namespace RpgLibrary.TileEngine
{
    public class Engine
    {
        #region Field Region

        private static int _tileWidth;
        private static int _tileHeight;
        private static Rectangle _viewPortRectangle;

        #endregion

        #region Property Region

        public static int TileWidth
        {
            get { return _tileWidth; }
        }

        public static int TileHeight
```

```
        {
            get { return _tileHeight; }
        }

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

        #endregion

        #region Constructors

        public Engine(int tileWidth, int tileHeight, Rectangle viewportRectangle)
        {
            Engine._tileWidth = tileWidth;
            Engine._tileHeight = tileHeight;
            Engine._viewPortRectangle = viewportRectangle;
        }

        #endregion

        #region Methods

        public static Point VectorToCell(Vector2 position)
        {
            return new Point((int)position.X / _tileWidth, (int)position.Y / _tile-
Height);
        }

        public static Point PointToWorld(Point point)
        {
            return new Point(point.X * _tileWidth, point.Y * _tileHeight);
        }
        #endregion
    }
}
```

There are a few fields in this class. The first is _tileWidth which represents the width of a tile on the screen. The next is _tileHeight which is the height of a tile on the screen. The last is a rectangle which defines the dimensions of the window. There is a little bit of replication between this project and the SharedProject. This brings up the point. Why am I adding classes to the RpgLibrary instead of the SharedProject? It is because of how I plan to store maps. I plan to use the Intermediate Serializer to save maps. For it to work, you need a DLL, and shared projects do not generate DLLs. We could write our own content pipeline extension and use the shared project. I will not be doing that in this tutorial series, though. There are properties to expose the value of the fields. The viewport should technically by readonly as well, but I will leave it as is as it came from another project that doesn't use render targets.

There is a single constructor that takes as parameters the width of tiles on the screen, the height of tiles on the screen and a rectangle that describes the view port. It then sets the fields to the values of the arguments passed in.

There are two static methods. The first is VectorToCell. It takes a Vector2 as an argument. It then

returns which tile the Vector2 is in. That is calculated by casting the position to an integer and then dividing that by the width or height of a tile on the screen. The second is PointToWorld. Given a tile, it returns a point in screen coordinates.

Before I get to the Camera class, I want to implement a TileMap class, as they are interrelated. To get to the TileMap class, we need a Tileset and TileLayer. Because the TileLayer is dependent on a Tileset, I will start with a Tileset. Right-click the TileEngine folder in the RpgLibrary project, select Add and then Class. Name this new class Tileset. Here is the code for that class. Well, actually, it will be a struct. What's the difference? It defines where they are stored in memory and if they are a reference type or a value type. If you have a value type, ie struct, and you assign it to a new variable, you are creating a copy of that variable. If you have a reference type, ie class, and you assign it to a new variable, the new variable is pointing to the same memory as the original. Changing a value in either variable affects the other. I know you are asking, what is with all of the computer science tangents? Well, not everybody arriving at these tutorials is at the same point in their adventures. I don't want to lose those who are not so familiar with the inner workings of C# and programming.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

namespace RpgLibrary.TileEngine
{
    public struct Tileset
    {
        #region Fields and Properties

        private Texture2D _image;
        private int _tileWidthInPixels;
        private int _tileHeightInPixels;
        private int _tilesWide;
        private int _tilesHigh;
        private readonly Rectangle[] _sourceRectangles;

        #endregion

        #region Property Region

        public Texture2D Texture
        {
            get { return _image; }
            private set { _image = value; }
        }

        public int TileWidth
        {
            get { return _tileWidthInPixels; }
            private set { _tileWidthInPixels = value; }
        }

        public int TileHeight
        {
```

```
            get { return _tileHeightInPixels; }
            private set { _tileHeightInPixels = value; }
        }

        public int TilesWide
        {
            get { return _tilesWide; }
            private set { _tilesWide = value; }
        }

        public int TilesHigh
        {
            get { return _tilesHigh; }
            private set { _tilesHigh = value; }
        }

        public Rectangle[] SourceRectangles
        {
            get { return (Rectangle[])_sourceRectangles.Clone(); }
        }

        #endregion

        #region Constructor Region

        public Tileset()
        {
            _tilesWide = 0;
            _tilesHigh = 0;
            _tileWidthInPixels = 0;
            _tileHeightInPixels = 0;
            _sourceRectangles = Array.Empty<Rectangle>();
            _image = null;
        }

        public Tileset(Texture2D image, int tilesWide, int tilesHigh, int tileWidth, int
tileHeight)
        {
            _image = image;
            _tileWidthInPixels = tileWidth;
            _tileHeightInPixels = tileHeight;
            _tilesWide = tilesWide;
            _tilesHigh = tilesHigh;

            int tiles = tilesWide * tilesHigh;

            _sourceRectangles = new Rectangle[tiles];

            int tile = 0;

            for (int y = 0; y < tilesHigh; y++)
                for (int x = 0; x < tilesWide; x++)
                {
                    _sourceRectangles[tile] = new Rectangle(
                        x * tileWidth,
                        y * tileHeight,
                        tileWidth,
                        tileHeight);
                    tile++;
                }
```

```
        }

        #endregion

        #region Method Region
        #endregion
    }
}
```

There are six fields in the structure. The first is the image for the tile set. The next is the width and height of a tile in pixels. Following that is the number of tiles wide a tile set is, and the number of tiles high a tile set is. Finally, there is an array that defines the source rectangles for the tile set. There are properties to expose the value of all of the fields.

There are two constructors. The first is the parameterless constructor that is required by a structure. It initializes all of the fields to default values. To avoid creating a zero-element array, you can use the Array.Empty method. The second constructor takes the tile width in pixels, the tile height in pixels, the number of tiles high a tile set is, along with the number of tiles wide, and the texture of the tileset. It sets the fields to the parameters passed in. It then calculates the number of tiles in the tile set. It creates an array with that many elements and defines a counter that counts which source rectangle we are on. This is important. When creating the source rectangles, we start in the upper left corner, going across row by row, and finally arriving at the bottom right corner. You need to follow the same pattern any time you are creating source rectangles for tiles.

Third time writing this part. First, I want to add a class that represents a tile. Actually, it is going to be a structure. Right-click the TileEngine folder in the RpgLibary project, Select Add and then Class. Name this new structure Tile. Here is the code for that structure.

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

namespace RpgLibrary.TileEngine
{
    public struct Tile
    {
        #region Field Region

        private int _tileIndex;
        private int _tileset;
        private int _rotation;
        private bool _visible;

        #endregion

        #region Property Region

        [ContentSerializer]
        public int TileIndex
        {
            get { return _tileIndex; }
            private set { _tileIndex = value; }
```

```
        }

        [ContentSerializer]
        public int Tileset
        {
            get { return _tileset; }
            private set { _tileset = value; }
        }

        [ContentSerializer(Optional = true)]
        public bool Visible
        {
            get { return _visible; }
            set { _visible = value; }
        }

        [ContentSerializer(Optional = true)]
        public int TileRotation
        {
            get { return _rotation; }
            set { _rotation = value; }
        }

        #endregion

        #region Constructor Region

        public Tile()
        {
            _tileIndex = -1;
            _tileset = -1;
            _visible = true;
            _rotation = 0;
        }

        public Tile(int tileIndex, int tileset, bool visible = true, int rotation = 0)
        {
            _visible = visible;
            _tileIndex = tileIndex;
            _tileset = tileset;
            _rotation = rotation;
        }

        #endregion
    }
}
```

So, there are four fields in this class. Two that you would expect: _tileIndex and _tileset. There are two others that you might not have guessed: _rotation and _visible. _rotation is handy if you need to quickly, well, rotate a tile, such as rotate left or right 90 degrees. The other, _visible, is handy for things like secret doors, hidden items, etc. There are four properties to expose the values of the fields. The first two properties had private set accessors because you typically don't want them changed once they are set. They are also marked with the ContentSerializer attributes, so the Intermediate Serializer will serialize them. Just be aware that you can do this right now. I will discuss it more when we get to the editor. The Visible and Rotation properties are not readonly. They have ContentSerializer with an optional equal to true, so if they are not serialized to the document, they won't blow up. There are two constructors. The one is the required constructor that takes no parameters and initializes the

field. The second takes a parameter for each field, with the final two optional and their values set to true and zero.

Great, we have tiles that we can use to populate our maps. So, let's add a MapLayer class that will hold all of our tiles. Right-click the TileEngine folder in the RpgLibrary project, select Add and then Class, naming the class MapLayer. Here is the code for the MapLayer class.

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

namespace Rpglibrary.TileEngine
{
    public class MapLayer : ILayer
    {
        #region Field Region

        private readonly Tile[,] _layer;

        #endregion

        #region Property Region

        public int Width
        {
            get { return _layer.GetLength(1); }
        }

        public int Height
        {
            get { return _layer.GetLength(0); }
        }

        #endregion

        #region Constructor Region

        public MapLayer(Tile[,] map)
        {
            this._layer = (Tile[,])map.Clone();
        }

        public MapLayer(int width, int height)
        {
            _layer = new Tile[height, width];

            for (int y = 0; y < height; y++)
            {
                for (int x = 0; x < width; x++)
                {
                    _layer[y, x] = new Tile(0, 0);
                }
            }
        }

        #endregion
```

```
        #region Method Region

        public Tile GetTile(int x, int y)
        {
            return _layer[y, x];
        }

        public void SetTile(int x, int y, Tile tile)
        {
            _layer[y, x] = tile;
        }

        public void SetTile(int x, int y, int tileIndex, int tileset, bool visible =
true, int rotation = 0)
        {
            _layer[y, x] = new Tile(tileIndex, tileset, visible, rotation);
        }

        public void Update(GameTime gameTime)
        {
        }

        Rectangle destination = new(0, 0, Engine.TileWidth, Engine.TileHeight);
        Tile tile;

        Point min;
        Point max;
        Vector2 origin;

        public void Draw(SpriteBatch spriteBatch, Camera camera, List<Tileset> tilesets)
        {
            Point cameraPoint = Engine.VectorToCell(camera.Position);
            Point viewPoint = Engine.VectorToCell(
                new Vector2(
                    (camera.Position.X + camera.ViewportRectangle.Width),
                    (camera.Position.Y + camera.ViewportRectangle.Height)));

            min.X = Math.Max(0, cameraPoint.X - 1);
            min.Y = Math.Max(0, cameraPoint.Y - 1);
            max.X = Math.Min(viewPoint.X + 1, Width);
            max.Y = Math.Min(viewPoint.Y + 1, Height);

            for (int y = min.Y; y < max.Y; y++)
            {
                destination.Y = y * Engine.TileHeight;

                for (int x = min.X; x < max.X; x++)
                {
                    tile = GetTile(x, y);

                    if (tile.TileIndex == -1 || tile.Tileset == -1 || tile.TileIndex >=
tilesets[tile.Tileset].SourceRectangles.Length)
                        continue;

                    destination.X = x * Engine.TileWidth;

                    if (tile.TileRotation == 0)
                    {
                        spriteBatch.Draw(
```

```
                            tilesets[tile.Tileset].Texture,
                            destination,
                            tilesets[tile.Tileset].SourceRectangles[tile.TileIndex],
                            Color.White);
                    }
                    else
                    {
                        origin.X = (float)tilesets[tile.Tileset].SourceRectan-
gles[tile.TileIndex].Width / 2f;
                        origin.Y = (float)tilesets[tile.Tileset].SourceRectan-
gles[tile.TileIndex].Height / 2f;

                        Rectangle dest2 = new(destination.X, destination.Y, destina-
tion.Width, destination.Height);
                        dest2.X += Engine.TileWidth / 2;
                        dest2.Y += Engine.TileHeight / 2;

                        spriteBatch.Draw(
                            tilesets[tile.Tileset].Texture,
                            dest2,
                            tilesets[tile.Tileset].SourceRectangles[tile.TileIndex],
                            Color.White,
                            MathHelper.ToRadians(tile.TileRotation),
                            origin,
                            SpriteEffects.None,
                            1f);
                    }
                }
            }
        }

        #endregion
    }
}
```

This class is where a lot of the magic happens. It is where rendering actually occurs. There is just a single field in this class that is a two-dimensional array of Tile objects. There are a couple of other options that I could have gone with. I could have gone with a single-dimensional array that emulates a 2D array. Also, I could have gone with a Dictionary<Point, Tile>. Going along with the Dictionary<Point, Tile> idea, I could have gone with a List<Tile>. The 2D array is the most straightforward to implement.

There are a couple of properties that return the width and height of the map. C#, to me, is a little backward in that when dealing with a multidimensional array, if you want to get the width, you pass in 1 in the call to GetLength and 0 to get the height. You need to remember that when iterating over the array to render tiles.

There are two constructors for the class. The first takes a 2D array of Tile objects. It assigns the field to be a clone of the array. The clone is a shallow copy of the array. It is more appropriate than assigning the array to the field directly. The second constructor takes the width and height of the array. In nested for-loops, it iterates over all of the rows and columns. It sets each tile to (0, 0). It might have been more appropriate to use (-1, -1). The reason is that if either the tile index or tile set index is -1, the tile will not be drawn.

There are GetTile and SetTile, which do what you expect. They get the tile or set the tile. As I

mentioned earlier, I have the y and x coordinated reversed from what you would expect in math. There are two overloads of the SetTile method. The first takes as parameters the x and y coordinates along with a Tile object. The second takes as parameters the x and y coordinates, plus the tile index, the tile set index and optionally, if the tile is visible and its rotation. They default to true and 0, respectively.

Because the class implements the ILayer interface, it must implement the Update method. Currently, it does nothing.

So, I lied. There are actually more fields. They are here to increase efficiency slightly. You could just as well include them I n the Draw method. The first is the location where the output will be rendered. The second is a Tile object. Next are two Points that represent where in the map to begin rendering and where to end rendering. They are padded by plus or minus one tile. There is also a Vector2 that will hold the origin of rotated tiles. I don't remember why I implemented rotated tiles. I don't think I've ever actually used them. I think it was for a tutorial request long ago.

So, a lot going on in the Draw method. We don't just render the entire map. We only render what is visible, plus or minus one tile, to allow for edges. The first thing we need to do is calculate what tile the camera is in. Then, we need to determine the last tile its viewport is in. To determine where to start, you take the maximum of zero and the camera point minus one. To find where to stop drawing, you take the minimum of the viewpoint plus one and the width or height of the map. For the Y coordinate, you loop between min.Y and max.Y rendering from top to bottom. For the X coordinate, you loop from min.X to max.X for each Y value.

The Y coordinate of the destination only changes when the Y coordinate changes. For that reason, the Y value is set at each step through the outer loop. For the X coordinate, I set the tile field to the GetTile method, passing in the coordinates. I remember why I added the tile field. It is so that it can be set once and used multiple times without having to index it. If the tile index is -1, the tile set is -1, or the tile index is greater than the number of tiles, I continue to the next iteration of the loop. I now set the X coordinate of the destination.

Next, I check to see if the rotation is zero. If it is, I just draw the tile. In the event that it is not, I calculate the origin of the tile, which is half its width and half of its height. I create a new destination rectangle. I offset its X and Y coordinates by half the width of a tile on the screen and half the height of a tile on the screen, respectively. I then use the overload of the draw method that takes a texture, destination rectangle, source rectangle, tint colour, the angle of rotation cast to radians, the origin, no sprite effects and a layer depth of 1.

What we need now is a class to represent a map, hinted at way back when we created the Engine class. Again, right-click the TileEngine folder in the RpgLibrary project, select Add and then Class. Name this class TileMap. As per usual, here is the code.

```
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework;
using Rpglibrary.TileEngine;
using System;
```

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace RpgLibrary.TileEngine
{
    public class TileMap
    {
        #region Field Region

        private readonly string _name;
        private readonly List<Tileset> _tilesets;
        private readonly List<ILayer> _mapLayers;

        private int _mapWidth;
        private int _mapHeight;

        #endregion

        #region Property Region

        public string Name
        {
            get { return _name; }
        }

        public int WidthInPixels
        {
            get { return _mapWidth * Engine.TileWidth; }
        }

        public int HeightInPixels
        {
            get { return _mapHeight * Engine.TileHeight; }
        }

        #endregion

        #region Constructor Region

        public TileMap(string name, List<Tileset> tilesets, MapLayer baseLayer, MapLayer
buildingLayer, MapLayer splatterLayer)
        {
            this._name = name;
            this._tilesets = tilesets;
            this._mapLayers = new List<ILayer>();

            _mapLayers.Add(baseLayer);

            AddLayer(buildingLayer);
            AddLayer(splatterLayer);

            _mapWidth = baseLayer.Width;
            _mapHeight = baseLayer.Height;
        }

        public TileMap(string name, Tileset tileset, MapLayer baseLayer)
        {
            this._name = name;
```

```
        _tilesets = new List<Tileset>
        {
            tileset
        };

        _mapLayers = new List<ILayer>
        {
            baseLayer
        };

        _mapWidth = baseLayer.Width;
        _mapHeight = baseLayer.Height;
    }

    #endregion

    #region Method Region

    public void AddLayer(ILayer layer)
    {
        if (layer is MapLayer layer1)
        {
            if (!(layer1.Width == _mapWidth && layer1.Height == _mapHeight))
                throw new Exception("Map layer size exception");
        }

        _mapLayers.Add(layer);
    }

    public void AddTileset(Tileset tileset)
    {
        _tilesets.Add(tileset);
    }

    public void Update(GameTime gameTime)
    {
        foreach (ILayer layer in _mapLayers)
        {
            layer.Update(gameTime);
        }
    }

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

    #endregion

    public void Resize(List<ILayer> layers)
    {
        _mapWidth = ((MapLayer)layers[0]).Width;
        _mapHeight = ((MapLayer)layers[0]).Height;
```

```
            _mapLayers.Clear();

            foreach (ILayer layer in layers)
                _mapLayers.Add(layer);
        }
    }
}
```

Nearing the finish line for this tutorial. A map will have a name, so there is a field for that. A map will have a number of tile sets associated with it, so there is a field for that as well. There is also a field to hold the layers. You will notice that I use ILayer so I can iterate over the entire collection of layers using, you guessed it, polymorphism. There are also fields for the width and height of the map. There are properties for the name of the map, the width of the map in pixels and the height of the map in pixels.

There are two constructors for the class. The first takes the name, a list of tile sets, a base layer, a building layer and a splatter layer. So, the base layer is the ground, duh! The building layer is for buildings. It will mainly be empty tiles. Here is where a Dictionary<Point, Tile> would have been a good choice. It would save much memory. The splatter layer is curious. It just adds decoration, like flowers on grass, cracks in roads, etc. It will also be mostly empty tiles, so another form for storing the tiles would be appropriate. The code itself initializes the fields using the values passed in. The base layer is added to the list of layers directly. The other layers are added using the AddLayer method, which I will get to shortly, which checks to make sure the dimensions of layers are the same. The second constructor works basically the same as the first.

The AddLayer method uses pattern matching to see if the layer passed in is a MapLayer. If it is, it checks to see if the width of the layer matches the width of the map and that the height of the layer matches the height of the map. If they don't, I throw an exception. Otherwise, I add the layer to the list of layers. I'm not so picky when it comes to what type of tile set to add, so they are just added.

In the Update method, I iterate over all of the layers, calling their Update method. Then, in the Draw method, I loop over all of the layers again. This time, I call their Draw methods if they are a map layer. In the future, I will be adding layers that are not visible, like the collision layer. There is one last method in the class, Resize. I could have come up with a better name. What it does, is remove all of the layers on a map and add new layers.

I don't want to venture any further, but I also feel like I'd leave you hanging if I didn't demonstrate maps. Venturing further wins, so let's add a class for the gameplay state. This state is going to be platform-specific. Mainly because it requires a map, and maps are part of the RpgLibary. Such is the price of using the Intermediate Serializer instead of writing our own content pipeline extension.

Before I get to the platform-specific code, I want to add an interface to the SharedProject project. It will be for the gameplay state. It will allow us to define the gameplay state in the games and reference it in the library. Right-click the GameScreens folder in the SharedProject, select Add and then New Item. Name this new item IGamePlayState. Here is the code for that interface.

```
using System;
using System.Collections.Generic;
```

```
using System.Text;

namespace SharedProject.GamesScreens
{
    public interface IGamePlayState
    {
        GameState Tag { get; }
    }
}
```

Pretty much the same interface that we've used before. It will just be used like the others. Now, I will turn my attention toward the games. I will start with the Desktop platform. The steps will be almost identical for the Android platform. Right-click the EyesOfTheDragon project in the Solution Explorer, select Add and then New Folder. Name this new folder GameStates. Now, right-click the GameStates folder, select Add and then Class. Name this new class GamePlayState. The code for that class follows next.

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

namespace EyesOfTheDragon.GameStates
{
    public interface IGamePlayState
    {
        GameState State { get; }
    }

    public class GamePlayState : GameState, IGamePlayState
    {
        private readonly Camera camera;
        TileMap map;
        readonly Engine engine;
        RenderTarget2D renderTarget;

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

        public GameState State => 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);
        }

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

            SpriteBatch.End();

            GraphicsDevice.SetRenderTarget(null);

            SpriteBatch.Begin(SpriteSortMode.Immediate, BlendState.AlphaBlend, Sampler-
State.PointClamp);

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

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

Like the other game states, there is an interface for this state. It contains a single property that returns the current state. There are four fields. The first is a Camera needed for the map. The next is a TileMap. After that is an Engine. Finally, there is a render target to draw our base resolution to. In the constructor, I register the instance of the state. I also initialize the camera, passing in the base resolution from the settings class and the engine setting the tile size to 32 by 32 and the base resolution from the settings class. I also included the property required by the interface.

In the LoadContent method, after the call to base.LoadContent, I initialize the render target. I then

load the texture for the tile set. I create a new List<Tileset> and add a new tile set that is eight tiles wide, eight tiles high and with a tile width and height of 32 pixels. I create a new layer that is one hundred tiles by one hundred tiles. I create the map next.

In the Update method, I call the Update method of the map. Like I did in the previous tutorial, I set the render target and clear the render target to black. For now, I call Begin on the sprite batch passing in some default parameters. The exciting parameters are SamplerState.PointClamp and Matrix.Identity. The first deals with how the texture is sampled and prevents tearing when the tiles are not rendered on rounded pixels. The other, while only the identity matrix, will be used in translating the tiles drawn based on the camera. We will get to that in another tutorial. After the call to Begin, I call the Draw method of the map. I then end rendering and reset the render target back to the screen. I then draw the render target to the entire screen.

Before I get to implementing this in the game, I'm going to add the same change to the Android project. Right-click the EyesOfTheDragonAndroid project, select Add and then New Folder. Name this new folder GameStates. Right-click the GameStates folder, select Add and then Class. Name this new class GamePlayState. The code follows next.

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

namespace EyesOfTheDragonAndroid.GameStates
{
    public class GamePlayState : GameState, IGamePlayState
    {
        readonly Camera camera;
        TileMap map;
        readonly Engine engine;
        RenderTarget2D renderTarget;

        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);
        }

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

            SpriteBatch.End();

            GraphicsDevice.SetRenderTarget(null);

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

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

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

Virtually identical to the other class. In fact, the only difference is the namespace. While we are in the Android project, let's add the state to the game. Replace the Android class with the following code.

```
using Android.Views;
using EyesOfTheDragonAndroid.GameStates;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using SharedProject;
```

```
using SharedProject.GameScreens;
using SharedProject.GamesScreens;

namespace EyesOfTheDragonAndroid
{
    public class Android : 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 Android()
        {
            _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)
```

```
        {
            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);
        }
    }
}
```

The new code just defines a new property to hold the game state and initializes the new property. We need to do the same thing to the Desktop class. Replace that with the following code.

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

namespace EyesOfTheDragon
{
    public class Desktop : Game
    {
        private readonly 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 Desktop()
        {
            _graphics = new GraphicsDeviceManager(this);

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

            _graphics.PreferredBackBufferWidth = Settings.BaseWidth;
            _graphics.PreferredBackBufferHeight = Settings.BaseHeight;
            _graphics.ApplyChanges();

            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 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);
        }
    }
}
```

Thanks for hanging in there. There is only one last change that I want to make. That is, I want to flip to the GamePlayState from the StartMenuState. Wait a minute, you're saying. They're in different projects that don't talk to one another. Well, they do, through the interface. The interface is known to both projects, and they can talk to one another through that contract. They only know what the contract defines, though. So, the only common point is the Tag property. That is enough to allow us to switch states. Replace the TitleState to the following code.

```
using System;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;
using SharedProject.Controls;
```

```
using SharedProject.GamesScreens;

namespace SharedProject.GameScreens
{
    public interface ITitleState
    {
        GameState Tag { get; }
    }

    public class TitleState : GameState, ITitleState
    {
        #region Field region

        Texture2D backgroundImage;
        RenderTarget2D renderTarget;

        readonly GameState tag;

        public GameState Tag
        {
            get { return tag; }
        }

        #endregion

        #region Constructor region

        public TitleState(Game game)
            : base(game)
        {
            Game.Services.AddService<ITitleState>(this);
            SpriteBatch = Game.Services.GetService<SpriteBatch>();
        }

        #endregion

        #region XNA Method region

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

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

            ContentManager Content = Game.Content;
            backgroundImage = Content.Load<Texture2D>(@"Backgrounds\titlescreen");

            Label startLabel = new()
            {
                Position = new Vector2(350, 600),
                Text = "Tap to begin",
                Color = Color.White,
                TabStop = true,
                HasFocus = true
            };

            startLabel.Selected += StartLabel_Selected; ;

            ControlManager.Add(startLabel);
        }
```

```
        private void StartLabel_Selected(object sender, EventArgs e)
        {
            GameState state = Game.Services.GetService<IGamePlayState>().Tag;
            StateManager.ChangeState(state);
        }

        public override void Update(GameTime gameTime)
        {
            if (Xin.WasMouseReleased(MouseButtons.Left) || Xin.TouchReleased() ||
Xin.WasKeyPressed())
            {
                StartLabel_Selected(this, null);
            }
            base.Update(gameTime);
        }

        public override void Draw(GameTime gameTime)
        {
            GraphicsDevice.SetRenderTarget(renderTarget);
            renderTarget.GraphicsDevice.Clear(Color.Black);

            SpriteBatch.Begin();

            SpriteBatch.Draw(
                backgroundImage,
                Settings.BaseRectangle,
                Color.White);

            base.Draw(gameTime);

            SpriteBatch.End();

            GraphicsDevice.SetRenderTarget(null);

            SpriteBatch.Begin();

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

            SpriteBatch.End();
        }

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

            LoadContent();
        }

        #endregion
    }
}
```

What has changed here is the StartLabel_Selected method that is triggered when the mouse is clicked, a key is pressed, or the screen is tapped. Instead of grabbing the StartMenuState using the IStartMenuState interface, I grab the GamePlayState using IGamePlayState interface. This effectively allows us to interact with the GamePlayState without having a reference to it in our project. Pretty cool, right?

Wow, this certainly long enough. 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