A Summoner's Tale
Chapter 2
Tile Engine

Welcome to my Summoner's Tale tutorial series on creating a Pokémon-inspired game with MonoGame. I'm writing these tutorials for the MonoGame 3.8.1 framework using Visual Studio 2022. The code should work on previous versions of MonoGame and Visual Studio. I plan on creating the editors on macOS and Windows, except the map editor. The map editor will be Windows only.

The tutorials will make more sense if they are read in order. You can find the list of tutorials on my web blog, A Summoner's Tale page. In addition to the PDFs, I will make the code for each tutorial available on GitHub here: https://github.com/Synammon/summoners-tale. It will also be included on the page that links to the tutorials.

This tutorial will have some visible results. I will start work on the tile engine. First, what is a tile engine, and why do we 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.

I will be using a multilayer tile engine for these tutorials. For a map, there will be four layers: the ground layer, the edge layer, the building layer, and the decoration layer. As their names imply, the ground-related tiles will go on the ground layer. The edge layer adds transition tiles between ground tiles, for example, between grass and dirt. The building layer holds building tiles or furniture tiles for interior maps. Finally, the decoration layer is for tiles that add decorations to add variety to the map. Now, there can be any number of layers for a map, and the engine can be expanded for more. There will be other layers eventually, such as a character layer that would hold the NPCs on a map.

The first thing that we will need is some tiles. I happened to be looking at one of my favourite sites for game art, https://itch.io. They have tons of resources for game developers. I will be using tile sets created by Cainos. Please visit their page on itch.io and show your support for their work. You can download the tiles from itch.io here. Just click the download button and then click to download the file. If you have your own tile sets feel free to use them. You will just need to make a couple of changes when creating the tile sets. I will mention where those are in the tutorial.

Now that we have tiles, we can get started. First, let's add the tiles to the project. In the SummonersTale project, there is a folder called the Content folder. Expand that, and you will find a Content.mgcb file. This file holds a list of all the content in your game. Double-click that to open the MGCB Editor. In the editor, select the Content node. Use the Add New Folder button to add a folder named Tiles. Select the Tiles folder. Use the Add Existing Item button to add the tiles. Add the TX*.png images. Save the project and close the editor.

Now, it would be perfectly possible to add the tile engine to the SummonersTale project. The game would build and run. The only problem is that in order to compile the maps, we need access to a DLL, dynamic link library. This project does not generate a DLL. So, we need a project that does. That is why I added the Psilibrary project.

Right-click the Psilibrary project, select Add and then New Folder. Name this new folder TileEngine. Right-click the TileEngine folder, select Add and then class. Name this new class TileLayer. Here is the code for the TileLayer class.

```
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework;
using System.IO;
using System;

namespace Psilibrary.TileEngine
{
    public struct Tile
    {
        #region Property Region

        public int TileSet { get; set; }
        public int TileIndex { get; set; }

        #endregion

        #region Constructor Region

        public Tile()
        {
            TileSet = -1;
            TileIndex = -1;
        }

        public Tile(int set, int index)
        {
            TileSet = set;
            TileIndex = index;
        }

        #endregion
    }

    public class TileLayer
    {
        #region Field Region

        [ContentSerializer]
        readonly Tile[] tiles;
```

```
        int width;
        int height;

        Point cameraPoint;
        Point viewPoint;
        Point min;
        Point max;
        Rectangle destination;

        #endregion

        #region Property Region

        public bool Enabled { get; set; }

        public bool Visible { get; set; }

        [ContentSerializer]
        public int Width
        {
            get { return width; }
            private set { width = value; }
        }

        [ContentSerializer]
        public int Height
        {
            get { return height; }
            private set { height = value; }
        }

        #endregion

        #region Constructor Region

        private TileLayer()
        {
            Enabled = true;
            Visible = true;
        }

        public TileLayer(Tile[] tiles, int width, int height)
            : this()
        {
            this.tiles = (Tile[])tiles.Clone();
            this.width = width;
            this.height = height;
        }

        public TileLayer(int width, int height)
            : this()
```

```
{
    tiles = new Tile[height * width];
    this.width = width;
    this.height = height;

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

public TileLayer(int width, int height, int set, int index)
    : this()
{
    tiles = new Tile[height * width];
    this.width = width;
    this.height = height;

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

#endregion

#region Method Region

public Tile GetTile(int x, int y)
{
    if (x < 0 || y < 0)
        return new Tile();

    if (x >= width || y >= height)
        return new Tile();

    return tiles[y * width + x];
}

public void SetTile(int x, int y, int tileSet, int tileIndex)
{
    if (x < 0 || y < 0)
        return;

    if (x >= width || y >= height)
```

```
                return;

            tiles[y * width + x] = new Tile(tileSet, tileIndex);
        }

        public void Update(GameTime gameTime)
        {
            if (!Enabled)
                return;
        }

        public void Draw(GameTime gameTime, SpriteBatch spriteBatch, TileSet
tileSet, Camera camera)
        {
            if (!Visible)
                return;

            cameraPoint = Engine.VectorToCell(camera.Position);
            viewPoint = Engine.VectorToCell(
                new Vector2(
                    (camera.Position.X + Engine.ViewportRectangle.Width),
                    (camera.Position.Y + Engine.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);

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

            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.TileSet == -1 || tile.TileIndex == -1)
                        continue;

                    destination.X = x * Engine.TileWidth;

                    if (tileSet.TileSheets.Count > 0)
                    {
                        spriteBatch.Draw(
                            tileSet.TileSheets[tile.TileSet].Texture,
                            destination,

tileSet.TileSheets[tile.TileSet].SourceRectangles[tile.TileIndex],
                            Color.White);
                    }
                }
            }
        }
```

```
        #endregion
    }
}
```

There is a structure in this file that represents a tile. Structs are similar to classes. The big difference is that they are stored on the stack instead of the heap because they are value types instead of reference types. What that means is if you have two variables that are from classes that are set to the same object, any change to the object affects both variables. In contrast, if create a struct A and assign another variable to A and changes to B are not applied to A. A tile has a TileSet property and a TileIndex property. This is because a map can be made up of multiple tilesets. A tile with a TileSet property or a TileIndex property set to -1 is ignored during the rendering process. There are two constructors for the struct. The first is the required constructor that initializes the tile to -1 and the other to the parameters that are passed in.

In the TileLayer class, there is a readonly array of Tiles. This will hold the tiles on the map. It is marked with the attribute ContentSerializer which lets MonoGame know that we want this field written during the serialization process. Which, as you will remember from the last tutorial, writes an object to XML to be converted into an XNB file and read using the Content Pipeline.

I went with a single-dimensional array rather than a two-dimensional array because it is a little easier to iterate over it. It is readonly because I don't want it reassigned after it has been initialized. There are fields width and height that are the width and height of the layer. There are four Point fields. The first, cameraPoint, is where the 2D camera is. I will get to that in a bit. The viewPoint field is also used in rendering the layer. The min and max Points are used to figure out what should be visible in drawing the map. The destination is where the tile will be drawn.

There are two public bool properties, Enabled and Visible, which determine if the layer should be updated and rendered, respectively. BY design, only public members a serialized by the IntermediateSerializer. Because I don't want the width or height of the map being changed once set, I expose the set part of their fields as private. I also mark them with the ContentSerializer attribute.

One thing about using the intermediate serializer is it requires a parameterless constructor. I only want to create layers by passing in parameters. So, I made a constructor that takes zero parameters that sets the layer to enabled and visible to true. The second parameter takes an array of tiles, the width of the layer and the height of the layer. It first calls the constructor that takes no parameters. It then creates a shallow copy of the array passed in using the Clone method of the array class. It then sets the width and height fields using parameters passed in.

The last constructor takes the height and width of the layer. It also calls the constructor that takes zero parameters. It creates a new array with height times width tiles. We can emulate a 2D array in a 1D array using a formula. We will get to that shortly. It then sets the width and height fields to the parameters passed in. Next, there are two nested loops. The outer loop uses the variable y, and the inner loop uses the variable x. Inside to find which index in the array to we use the formula y * width + x.

So, if the height and width of the layer are five, elements zero to four and the first row of the map. Elements five to nine are the second row. Elements ten to fourteen are the third row, and so on.

The last constructor is like the previous constructor. The only difference is that it fills the layer with a tile set and a tile index.

There are two public methods, GetTile and SetTile, that get and set a tile at the given index, respectively. In the GetTile method, I check to see if the x and y coordinates are within the layer. If they are not in the layer, I return an empty tile. If not, I return the tile using the formula that was used in building the map. Currently, the Update method just checks to see if Enabled is false and exits.

The last method takes four parameters a GameTime parameter, a SpriteBatch parameter, TileSet parameter and a Camera parameter. I will add the TileSet and Camera classes shortly. If the layer is not visible, I exit the method. We only want to draw what is visible, plus or minus one tile. The camera's position will be a Vector2, and we need it as a Point. So, I use a method of the Engine class to convert it to a point. The viewPoint is the tile to stop rendering at. It is found by taking the camera's position and adding the width of the viewport to the X coordinate and the height of the viewport to the Y coordinate. This Vector is similarly converted to a Point.

Next, I calculate the minimum and maximum tile to draw. The minimum X coordinate is the maximum of 0 and the X coordinate of the camera minus one. Similarly, the minimum Y coordinate is the maximum of the camera's Y coordinate minus one and 0. This is where the minus one comes across. To determine the last X tile to draw, I get the minimum of the viewPoint's Y coordinate plus one and the width of the map. The Y value is calculated using the minimum of the viewPoint's Y coordinate plus the height of the map.

I set the destination rectangle to the be zero for the X and Y coordinate and the width and height to the width and height of a tile on the screen. There is a local variable, tile, that will hold the tile to be drawn.

To loop over the tiles, I start from the top left corner and render to the bottom right inside nested for loops. The Y coordinates will range from min.Y to max.Y. The X coordinates will range from min.X to max.X. Inside the outer loop. I set the Y property of the destination rectangle to the Y tile times the height of a tile. In the inner loop, I get the tile using the GetTile method. If either of the TileSet or TileIndex properties is set to -1, I go to the next tile. I set the X property of the destination tile to be the x coordinate of the tile times the width of the tile on the screen. The last thing is to check if there are tile sets. If there are, I call Draw passing in the tile set texture, the destination rectangle, the source rectangle in the tile set's tile sheet, and no tinting.

So, there are a few pieces missing. First, there is the Engine class, followed by the Camera class and the TileSet and TileSheet classes. I will add them in that order. The engine and camera classes do not need to be serialized. For that reason, they could be added to the shared library. I will keep the in Psilibrary to keep related classes together. Right-click the TileEngine folder in the Psilibrary project, select Add and then Class. Name this new class Engine. Repeat the process two times, naming the classes Camera and TileSet.

The Engine class is a static class that will have methods and properties that describe the viewport. Add the following code to the Engine class.

```
using Microsoft.Xna.Framework;

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

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

        private static Camera camera;

        #endregion

        #region Property Region

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

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

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


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

        #endregion

        #region Constructors

        static Engine()
        {
```

```
            ViewportRectangle = new Rectangle(0, 0, 1280, 720);
            camera = new Camera();

            TileWidth = 64;
            TileHeight = 64;
        }

        #endregion

        #region Methods

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

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

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

        #endregion
    }
}
```

The class describes how the tile engine is rendered on the screen. It is static because there should only ever be one instance. First, there is a Rectangle field that describes the viewport. Next, there are fields the describe the width and height of a tile on the screen. There should only ever be one Camera so it is included here as well. There are properties to expose the fields. The Camera property is readonly because I don't want it assigned outside of the class. After the properties is the constructor. It initializes the viewport rectangle, camera and the height and width of tiles.

There are some static methods. The first, Vector2Cell is used to return what cell a Vector2 is in. That is calculated by taking the X coordinate of the vector and dividing it by the width of a tile on the screen. The Y coordinate is calculated by dividing the Y coordinate of the vector by the tile height. The next method finds Vector2 that is the origin of a vector on the map. It is a vector with an X coordinate of the origin divided by the tile width times the tile width and the Y coordinate of the origin divided by the tile height times the tile height. The last method, Reset, is used to reset the viewport. It will be used in the

future when I tackle handling multiple resolutions. It sets the viewPortRectangle field to the rectangle passed in and the TileWidth and TileHeight parameters passed in.

That brings up to the Camera class. The points to what part of the map is to be rendered. Replace the code of the Camera class with the following.

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

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

        Vector2 position;
        float speed;

        #endregion

        #region Property Region

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

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

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

        #endregion

        #region Constructor Region

        public Camera()
```

```
        {
            speed = 4f;
        }

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

        #endregion
    }
}
```

There are two fields in the class. The first field is position, and it is the position of the camera. The other, speed, is a legacy field that I keep intact in case I want to implement a camera that does not follow the player. It is helpful in strategy games where you are commanding multiple units and need to scan the map. There are properties that expose the properties. There is a third property, Transformation. This property is used to translate world space to screen space. There is a static method of the Matrix class that creates a matrix that translates an object based on a Vector3 that is passed in. All that means is that what will be drawn in world space in screen space. So, if the camera's position is (80, 80), the world's coordinates will be shifted to 80 coordinates left, and 80 coordinates up because we are subtracting the camera's position. You need to remember that. You subtract the camera's position instead of adding it. That is why when I convert Vector2 to Vector3; I multiply Vector2 by -1.

There are two constructors for the class. The first takes no parameters and initializes the camera's speed property to 4 pixels. In the other constructor, I pass in the speed and position of the camera and set the properties to those values. You are safe to do that with Vector2 because it is a value type, not a reference type.

The next missing piece is a tile set. A tile set can be defined in multiple ways. The most common is as a collection of textures that define individual tiles or as a single image with the individual tiles as rectangles. I will be using the single image based on the tile sets that we downloaded earlier. One side note, it is best if your images are in powers of two. This is because it is more efficient for the graphics card. It is also more efficient to have a single image rather than multiple images. Replace the TileSet class with the following code.

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Runtime.CompilerServices;
using System.Text;
using System.Threading.Tasks;
```

```
namespace Psilibrary.TileEngine
{
    public class TileSheet
    {
        private Rectangle dimensions;
        private readonly Texture2D texture;
        private string name;
        private Rectangle[] sourceRectangles;

        [ContentSerializerIgnore]
        public Texture2D Texture
        {
            get { return texture; }
        }

        public Rectangle Dimensions
        {
            get { return dimensions; }
            set { dimensions = value; }
        }

        public string TextureName
        {
            get { return name; }
            set { name = value; }
        }

        [ContentSerializer]
        public Rectangle[] SourceRectangles
        {
            get { return sourceRectangles; }
            private set { sourceRectangles = value; }
        }

        private TileSheet()
        {
        }

        public TileSheet(Texture2D texture, string name, Rectangle dimensions)
        {
            this.texture = texture;
            this.name = name;
            this.dimensions = dimensions;

            SourceRectangles = new Rectangle[dimensions.X * dimensions.Y];

            for (int y = 0; y < dimensions.Y; y++)
            {
                for (int x = 0; x < dimensions.X; x++)
                {
                    SourceRectangles[y * dimensions.X + x] = new()
                    {
                        X = x * dimensions.Width,
                        Y = y * dimensions.Height,
                        Height = dimensions.Height,
                        Width = dimensions.Width
                    };
```

```
                }
            }
        }
    }

    public class TileSet
    {
        #region Fields and Properties

        private List<TileSheet> tileSheets;

        #endregion

        #region Property Region

        public List<TileSheet> TileSheets
        {
            get { return tileSheets; }
            set { tileSheets = value; }
        }

        #endregion

        #region Constructor Region

        private TileSet()
        {
            tileSheets = new();
        }

        public TileSet(TileSheet sheet)
            : this()
        {
            tileSheets.Add(sheet);
        }

        #endregion

    }
}
```

I included tile sheets and tile sets in the same file as they are related. A tile set is a collection of tile sheets. A tile sheet is an image with a number of tiles. I did it this way to allow for tile sets having different dimensions.

In a tile sheet, there are private fields that are the texture for the sheet, the name of the texture, a rectangle that defines the dimensions of the tile sheet, and the source rectangles for the tiles. There are properties to expose the fields. The texture is read-only and will be ignored by the intermediate serializer. The source rectangles property has a private set accessor, so it is marked with a ContentSerializer attribute. There are two constructors for a tile sheet. The private one is required for the intermediate serializer to deserialize the content. The second constructor takes as parameters a Texture for the sheet, the name of the sheet and the dimensions of the sheet. For the dimensions, the X

property is the number of tiles wide the sheet is, the Y property is the number of tiles high, and the Width and Height properties are the width and height of a tile, respectively.

There are then properties to expose fields. The first is Textures which returns the list of textures. It is marked with the ContentSerializerIgnore attribute because we don't want that exported to the XML. Following that is the list of image names. These are the names of the tile set image. This needs to be serialized, so there is no attribute because it is public, and the get and set properties are public as well. Next is the array of source rectangles. I made it with a private set because I didn't want new ones created. We do want these serialized so it is marked with the ContentSerializer attribute.

There are a couple of constructors for the class. The first takes no parameters and is included for the IntermediateSerializer. The second takes as parameters the texture of the tile sheet, the name of the tile sheet, and the dimensions of the tile sheet. Inside I construct the source rectangles. That is done in nested for loops. The outer loop will loop from the top to the bottom and the inner from left to right. A rectangle is created similarly to the destination rectangle of the tile layer. The X value is the x index of the inner loop times the Width property of the dimension. The Y value is the y index of the outer for loop times the Height property of the dimension. The Width and Height properties are the Width and Height properties of the dimension.

The TileSet class is trivial. It is just a container to hold the tile sheets. It has a private field that is a list of tile sheets and a public property to expose the field. There is a private constructor for the intermediate serializer, along with a public one that takes a TileSheet object as a parameter.

There is one last class that I want to add to this tutorial. That is a class that represents a tile map. A tile map is a collection of layers. Right now, we only have tile layers. In future tutorials, we will discuss other types of layers. Right-click the TileEngine folder in the Solution Explorer, select Add and then Class. Name this new class TileMap. Here is the code for that class.

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

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

        string mapName;
        TileLayer groundLayer;
        TileLayer edgeLayer;
```

```
        TileLayer buildingLayer;
        TileLayer decorationLayer;

        int mapWidth;
        int mapHeight;

        TileSet tileSet;

        #endregion

        #region Property Region

        [ContentSerializer(Optional = true)]
        public string MapName
        {
            get { return mapName; }
            private set { mapName = value; }
        }

        [ContentSerializer]
        public TileSet TileSet
        {
            get { return tileSet; }
            set { tileSet = value; }
        }

        [ContentSerializer]
        public TileLayer GroundLayer
        {
            get { return groundLayer; }
            set { groundLayer = value; }
        }

        [ContentSerializer]
        public TileLayer DecorationLayer
        {
            get { return decorationLayer; }
            set { decorationLayer = value; }
        }

        [ContentSerializer]
        public TileLayer EdgeLayer
        {
            get { return edgeLayer; }
            set { edgeLayer = value; }
        }

        [ContentSerializer]
        public TileLayer BuildingLayer
        {
            get { return buildingLayer; }
            set { buildingLayer = value; }
        }

        [ContentSerializer]
        public int MapWidth
        {
            get { return mapWidth; }
```

```
        private set { mapWidth = value; }
    }

    [ContentSerializer]
    public int MapHeight
    {
        get { return mapHeight; }
        private set { mapHeight = value; }
    }

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

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

    #endregion

    #region Constructor Region

    private TileMap()
    {
    }

    private TileMap(TileSet tileSet, string mapName)
        : this()
    {
        this.tileSet = tileSet;
        this.mapName = mapName;
    }

    public TileMap(
        TileSet tileSet,
        TileLayer groundLayer,
        TileLayer edgeLayer,
        TileLayer buildingLayer,
        TileLayer decorationLayer,
        string mapName)
        : this(tileSet, mapName)
    {
        this.groundLayer = groundLayer;
        this.edgeLayer = edgeLayer;
        this.buildingLayer = buildingLayer;
        this.decorationLayer = decorationLayer;

        mapWidth = groundLayer.Width;
        mapHeight = groundLayer.Height;
    }

    #endregion

    #region Method Region

    public void SetGroundTile(int x, int y, int set, int index)
```

```
{
    groundLayer.SetTile(x, y, set, index);
}

public Tile GetGroundTile(int x, int y)
{
    return groundLayer.GetTile(x, y);
}

public void SetEdgeTile(int x, int y, int set, int index)
{
    edgeLayer.SetTile(x, y, set, index);
}

public Tile GetEdgeTile(int x, int y)
{
    return edgeLayer.GetTile(x, y);
}

public void SetBuildingTile(int x, int y, int set, int index)
{
    buildingLayer.SetTile(x, y, set, index);
}

public Tile GetBuildingTile(int x, int y)
{
    return buildingLayer.GetTile(x, y);
}

public void SetDecorationTile(int x, int y, int set, int index)
{
    decorationLayer.SetTile(x, y, set, index);
}

public Tile GetDecorationTile(int x, int y)
{
    return decorationLayer.GetTile(x, y);
}

public void FillEdges()
{
    for (int y = 0; y < mapHeight; y++)
    {
        for (int x = 0; x < mapWidth; x++)
        {
            edgeLayer.SetTile(x, y, -1, -1);
        }
    }
}

public void FillBuilding()
{
    for (int y = 0; y < mapHeight; y++)
    {
        for (int x = 0; x < mapWidth; x++)
        {
            buildingLayer.SetTile(x, y, -1, -1);
        }
```

```
            }
        }

        public void FillDecoration()
        {
            for (int y = 0; y < mapHeight; y++)
            {
                for (int x = 0; x < mapWidth; x++)
                {
                    decorationLayer.SetTile(x, y, -1, -1);
                }
            }
        }

        public void Update(GameTime gameTime)
        {
            if (groundLayer != null)
                groundLayer.Update(gameTime);

            if (edgeLayer != null)
                edgeLayer.Update(gameTime);

            if (buildingLayer != null)
                buildingLayer.Update(gameTime);

            if (decorationLayer != null)
                decorationLayer.Update(gameTime);
        }

        public void Draw(GameTime gameTime, SpriteBatch spriteBatch, Camera camera,
bool debug = false)
        {
            if (WidthInPixels >= 1280)
            {
                spriteBatch.Begin(
                    SpriteSortMode.Deferred,
                    BlendState.AlphaBlend,
                    SamplerState.PointClamp,
                    null,
                    null,
                    null,
                    camera.Transformation);
            }
            else
            {
                Matrix m = Matrix.CreateTranslation(
                    new Vector3((1280) / 2, (720 - HeightInPixels) / 2, 0));
                spriteBatch.Begin(
                    SpriteSortMode.Deferred,
                    BlendState.AlphaBlend,
                    SamplerState.PointClamp,
                    null,
                    null,
                    null,
                    m);
            }

            if (groundLayer != null)
```

```
                groundLayer.Draw(gameTime, spriteBatch, tileSet, camera);

            if (edgeLayer != null)
                edgeLayer.Draw(gameTime, spriteBatch, tileSet, camera);

            if (decorationLayer != null)
                decorationLayer.Draw(gameTime, spriteBatch, tileSet, camera);

            if (buildingLayer != null)
                buildingLayer.Draw(gameTime, spriteBatch, tileSet, camera);

            spriteBatch.End();
        }

        #endregion
    }
}
```

That is a lot of code to digest at once. A lot of it is repetitions. First, maps have a mapName field. As I mentioned earlier in the tutorial, maps have four TileLayers. So, there is a field for each layer. Maps also have a with and height. Maps also have a TileSet. There are properties to expose each of the fields. The MapName property is marked as optional. This is an artifact from my games. It could safely be removed. The set properties MapWidth and MapHeight are private, so they are marked with the ContentSerializer attribute. There are two properties that return the width and height of the map in pixels.

There are three constructors in the class. The first is private and takes no parameters. As you guessed, it is there for the intermediate serializer. The second is also private and takes a tile set and name as arguments. The last takes the tile set, the four layers, and the name of the map. The third calls the second, which calls the first. They all just set the fields to the values passed in.

There are public methods to set or get the tile of a layer. In addition to those methods, there are methods to fill the edge, building and decoration layers with empty tiles. The Update method calls the Update method of the layers if they are not null.

The Draw method is where the rendering will take place. It takes as parameters a GameTime object, a SpriteBatch object, a Camera object and a bool parameter. The bool parameter is optional and will default to false. Inside the Draw method, I check to see if the width of the map in pixels is greater than or equal to 1280. For now, this is a hard-coded value. When we move to support multiple resolutions, it will be replaced. If that is true, I call the Begin method on the SpriteBatch passing in as value, SpriteSortMode.Deferred, BlendState.AlphaBlend, SamplerState.PointClamp, null, null, null, and the transformation matrix of the camera. Otherwise, I pass in the same parameters except for the transformation matrix. I pass in a matrix that will center the map on the screen. Next, there are calls that will call the draw method of the layer if it is not null.

Let's pull it all together in the game and render a random map. Update the Game1 class of the game to the following.

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Psilibrary.TileEngine;
using SummonersTale;
using SummonersTale.StateManagement;
using System;

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

        public Game1()
        {
            _graphics = new GraphicsDeviceManager(this)
            {
                PreferredBackBufferWidth = 1280,
                PreferredBackBufferHeight = 720
            };
            _graphics.ApplyChanges();
            _manager = new GameStateManager(this);

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

            Components.Add(_manager);
        }

        protected override void Initialize()
        {
            Components.Add(new FramesPerSecond(this));
            _graphics.ApplyChanges();

            Engine.Reset(new(0, 0, 1280, 720), 32, 32);
            _camera = new Camera();

            base.Initialize();
        }

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

            TileSheet sheet = new(Content.Load<Texture2D>(@"Tiles/TX Tileset
Grass"), "test", new(8, 8, 32, 32));
            TileSet set = new(sheet);

            TileLayer ground = new(100, 100, 0, 0);
            TileLayer edge = new(100, 100, -1, -1);
            TileLayer building = new(100, 100, -1, -1);
            TileLayer decore = new(100, 100, -1, -1);
```

```
            Random random = new Random();
            for (int i = 0; i < 1000; i++)
            {
                ground.SetTile(random.Next(0, 100), random.Next(0, 100), 0,
random.Next(0, 64));
            }

            _tileMap = new(set, ground, edge, building, decore, "test");
        }

        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);

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

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

What changed here? First, I added two new fields. A field for a TileMap and a field for a Camera as both are needed to render. Next, in the Initialize method, I call the Reset method on the Engine class, passing in a rectangle that describes the window and a tile height and width of sixty-four pixels. I also initialize the Camera field. In the LoadContent method, I create a TileSet using one of the textures that we downloaded earlier in the tutorial. If you used different textures, substitute their names here, along with the rectangle that describes the dimensions. Next, I create a TileSet using the TileSheet. Following that, I create the layers. The ground layer is set to tile set and tile index of 0, 0. The others are set to -1, -1. Just to show how it works, I set one thousand of the tiles to random values. Finally, I create the TileMap object using the values. The last change is to draw the map in the Draw method. I call the Draw method of the TileMap, passing in the appropriate parameters.

So, that is going to be it for this tutorial. I covered a lot, and I don't want to overload you. I encourage you to keep visiting my blog for the latest news on my tutorials.

Good luck with your game programming adventures.
Cynthia