Shadow Monsters – MonoGame Tutorial Series
Chapter 1
Tile Engine

This tutorial series is about creating a Pokemon style game with the MonoGame Framework called Shadow Monsters. The tutorials will make more sense if you read them in order as each tutorial builds on the previous tutorials. You can find the list of tutorials on my blog: Shadow Monsters. The source code for each tutorial will be available as well. I will be using Visual Studio 2019 Community for the series. The code should compile on the 2013, 2015 and 2017 versions as well.

I want to mention though that the series is released as Creative Commons 3.0 Attribution. It means that you are free to use any of the code or graphics in your own game, even for commercial use, with attribution. Just give credit to Cynthia McMahon and add a link to my site, https://mygameprogrammingadventures.blogspot,com. Screenshots of your project and/or a video of game play would be appreciated.

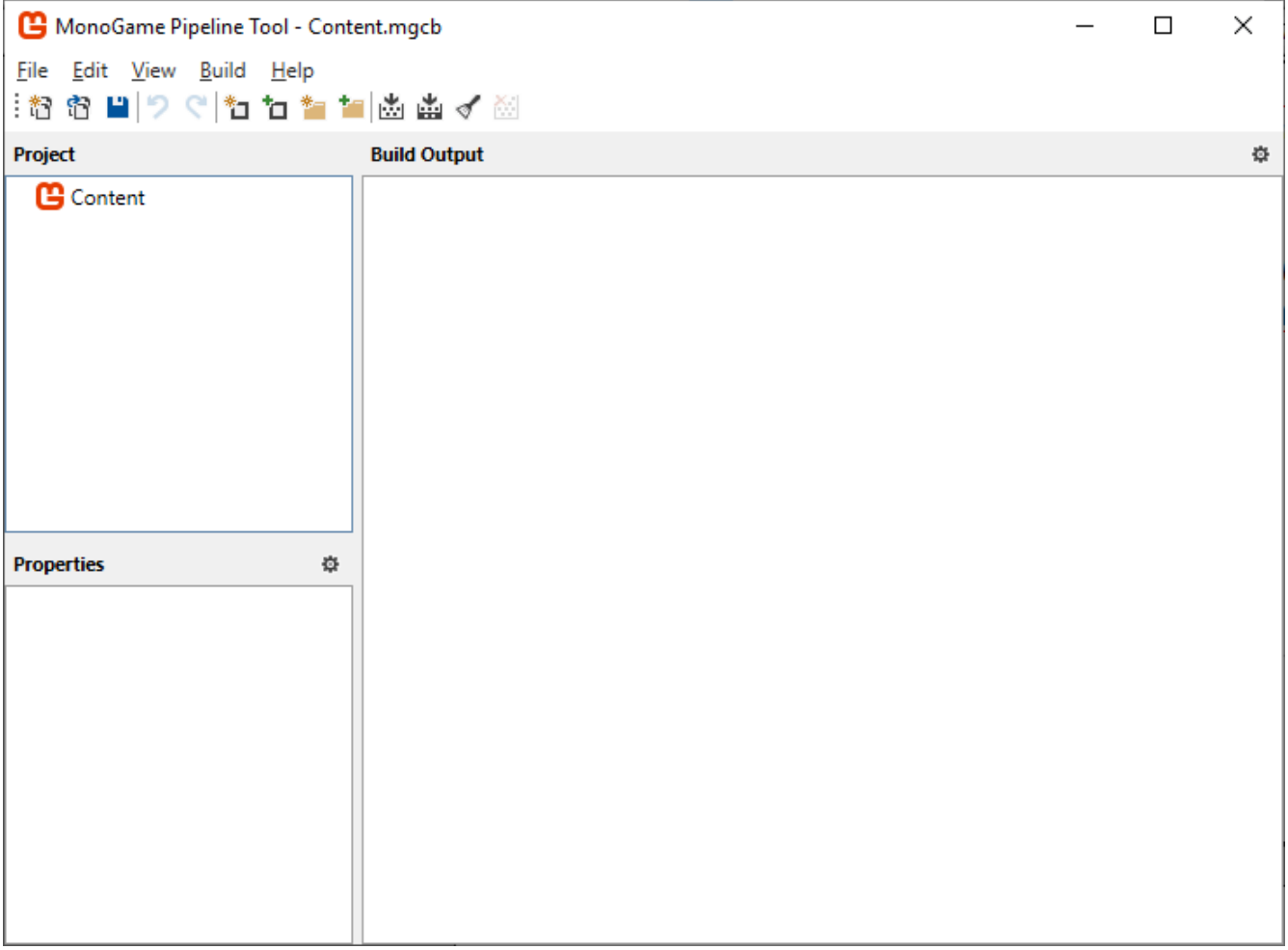
I also want to mention that I assume you have a basic understanding of C# and MonoGame. If you don't I recommend that you learn basic C# and work with MonoGame a little. Enough ti know the basics of fields, properties, methods, classes and the MonoGame framework.

In this tutorial I will be covering the tile engine. What exactly is a tile engine and why do we use one for games like this? First, a large map would be thousands of pixels wide and high if it was hand painted. Loading this onto the graphics processor is impossible above certain resolutions, typically 8192x8192 though it can be smaller. Drawing that image over and over again is also very inefficient. It's not as bad with today's hardware but still should be avoided.

This is where a tile engine comes in. A map is broken down into smaller pieces, called tiles. These tiles are much more efficient to draw. A tile engine is what draws the tiles on the screen.

Let's begin the tutorial. First, open Visual Studio if it is not already open. If you're opening it for the first time you will get a project dialog, On the right scroll down to Create a new project. On the Create a new project dialog enter MonoGame in the search box. Scroll down to MonoGame Cross Platform Desktop Project. Click the Next button. On this screen enter ShadowMonsters for the name of the project and click OK.

Next up is to add some tiles. I'm using some public domain tiles that I found on the internet at https://opengameart.org/content/outdoors-tileset-16x16. Download that tileset and save it to the desktop. Once you've saved them head back to Visual Studio. In the Solution Explorer there is Content folder, double click that folder. You will find a Content.mgcb in there with the MonoGame logo. Right click that and select Open With. In the dialog that comes up scroll down to the MonoGame Pipeline Tool and click the Set Default then click OK. The MonoGame Pipeline Tool will open and you will be presented with this dialog.



This is where you will add game content to your project. First, click the New Folder button in the tool bar. Name the folder Tiles and click Okay. Select the Tiles folder in the Project pane and then click the Existing Item button. Browse to the tile set that you downloaded earlier. When you click the button to add the tile set you will be prompted to copy the file or create a link. It is best if you copy the file. Click the Save button to save the project.

Now it is time to add some code. First, right click the ShadowMonsters project in the Solution Explorer, select Add and the New Folder. Name the new folder TileEngine. Now right click the TileEngine folder, select Add and then Class. Name this new class TileSet. Change the code for that class to the following. Don't worry I will explain it after you've read it.

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

namespace ShadowMonsters.TileEngine
{
    public class TileSet
    {
        public int TilesWide = 8;
        public int TilesHigh = 10;
        public int TileWidth = 16;
```

```
public int TileHeight = 16;

#region Fields and Properties

private List<Texture2D> image;
private List<string> imageName;
private Rectangle[] sourceRectangles;

#endregion

#region Property Region

public List<Texture2D> Textures
{
    get { return image; }
}

public List<string> TextureNames
{
    get { return imageName; }
}

public Rectangle[] SourceRectangles
{
    get { return sourceRectangles; }
}

#endregion

#region Constructor Region

public TileSet()
{
    image = new List<Texture2D>();
    imageName = new List<string>();

    sourceRectangles = new Rectangle[TilesWide * TilesHigh];

    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++;
        }
}

public TileSet(int tilesWide, int tilesHigh, int tileWidth, int tileHeight)
    : this()
{
    TilesWide = tilesWide;
    TilesHigh = tilesHigh;
    TileWidth = tileWidth;
    TileHeight = tileHeight;
```

```
            sourceRectangles = new Rectangle[TilesWide * TilesHigh];

            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 four public fields that describe a tile set, the number of tiles wide, the number of tiles high, the width of the tiles and the height of the tiles. Next are some generic List fields. One holds the images, or textures, of the tiles and the other their name. I included the name at this time but it is not really used until later on in the game. There is also an array of Rectangles. These describe the source tiles inside the textures. One thing to note is that tile set images must be the same dimensions and the tiles the same height and width. It is also best that they be a power of two. It is much kinder on the GPU to have images being loaded a power of two. It also enables graphic acceleration.

After the fields are three properties to expose them. You could make the fields public or use auto properties instead of doing it this way. It is my personal preference to create fields and then expose them using properties.

You will also notice that I used the #region and #endregion directives. They are there purely for organizational purposes and do not affect the code at all. It makes it easier to find code because you can narrow down your search. They can also be collapsed in Visual Studio so you can limit what you see. In larger files it makes sense to have more regions to keep things tidy,

There are two constructors next that are nearly identical. The first creates new lists for the tile set texture and their names and an array for the source rectangles that describe the tiles in the tile set. The number of tiles is the tiles wide times the tiles high. I have a local variable, tile, that will be the index of the source rectangle being created. Next are loops that loop over the rows of the tile set and then the columns of the tile set. That means the tiles will start at the top left corner and go left to right then top to bottom. This will become more important when we start work on the editor. The second constructor calls the first. It then sets the TilesWide, TilesHiigh, TileWidth and TileHeight fields. After setting the fields it creates the

source rectangles as before,

So, a map will be made up of four layers. They will be the ground, edge or transition, decoration and building or object layers. So, the next class we'll need describes a layer. Right click the TileEngine folder in the Solution Explorer, select Add and then Class. Name this new class TileLayer. Here is the code.

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

namespace ShadowMonsters.TileEngine
{
    public class 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

        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; }

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

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;
        }

        #endregion

    }
}
```

So, quite a bit of code at once, It's not that scary though. I actually tucked a second class in this one that describes a tile. It has two fields, TileIndex and TileSet. TileSet is the index of the image to draw and TileIndex is the source rectangle ti be used. If they are both -1 then nothing should be drawn. There is a constructor that initializes the fields to -1 and one that initializes them to the values passed in.

There are a few fields in the TileLayer class. The first is an array of Tile that will describe the layer. I went with a single dimensional array instead of two dimensional array because it can be serlialized. This will be important later on in the series. Next are fields for the width and height of the map. The next two fields will be used for drawing the layer which we will add in later. There are also fields min and max that will be used to limit what is drawn, instead of drawing the entire map. Finally is a field that says where to draw the tile.
Next up are the properties for the class. Yes, I broke my preference and used autopropreties. The first two will control of a layer should update itself and if a layer should draw itself. Next

are properties that expose the width and height of the layer.

You will notice that my classes flow in the same order: fields, properties, constructors and then methods. This is again personal preference and typically the way C# files are laid out. The constructors allow for different ways of creating layers. All the public constructors call the private constructor that makes the layer enabled and visible. The first public constructor takes an array of tiles, width and height. It creates a shallow copy of the array and sets the width and height. The next takes the width and height f the layer. It creates the array, sets the width and height and then loops over the height and width to set the tile. The formula for setting the tile is the row of tile times the width plus the column. The last constructor is like the previous except it takes the tile set and tile index. The parameters are used when creating the tile.

There are three methods in this class. The first, GetTile, will return the tile at the coordinates passed in. If either coordinate passed in is less than 0 an empty tile is returned. Similarly if either coordinate is outside the width or height of the map an empty tile is returned. The SetTile method also checks to make sure that the coordinates passed in are on the map or just return, It then sets the tile using the tile set and tile index passed in. The last method, Update, just checks if Enabled is false and if it is it exists. This method will be fleshed out more later. You will notice that there is no Draw method. That is because we are missing components. I will get to them now.

The first class that we need is a camera. The camera controls what part of the map is drawn. Right click the TileEngine folder, select Add and then Class. Name this new class Camera. Here is the code for the Camera class.

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

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

        Vector2 position;
        float speed;

        #endregion

        #region Property Region

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

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

```
        }

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

        #endregion

        #region Constructor Region

        public Camera()
        {
            speed = 4f;
        }

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

        #endregion

        public void LockCamera(TileMap map, Rectangle viewport)
        {
            position.X = MathHelper.Clamp(position.X,
                0,
                map.WidthInPixels - viewport.Width);
            position.Y = MathHelper.Clamp(position.Y,
                0,
                map.HeightInPixels - viewport.Height);
        }
    }
}
```

This class is pretty simple. There is a position field and a speed field and properties to expose them. The interesting property is the Transformation property. It creates a transformation matrix that will translate the map coordinates to screen coordinates. The Speed property uses MathHelper.Clamp to restrict the speed between 1 and 16. If the value is less than 1 it will be bumped up to one. Similarly, if it is greater than 16 it will be capped at 16.

There are two constructors. The first just sets the speed. The second takes the position of the camera.

There is a method in this class. It won't make complete sense because we are missing the TileMap class. What it does is clamp the position of the camera so that the background of the screen never shows.

There is a utility class that TileMap uses but it also uses TileMap. I will give the code for the utility class first. Right click the TileEngine folder, select Add and then Class. Name this new class Engine. Here is the code.

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

```
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;

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

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

        private TileMap map;

        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 TileMap Map
        {
            get { return map; }
        }

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


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

        #endregion

        #region Constructors

        public Engine(Rectangle viewPort)
        {
            ViewportRectangle = viewPort;
            camera = new Camera();
```

```
            TileWidth = 64;
            TileHeight = 64;
        }

        public Engine(Rectangle viewPort, int tileWidth, int tileHeight)
            : this(viewPort)
        {
            TileWidth = tileWidth;
            TileHeight = tileHeight;
        }

        #endregion

        #region Methods

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

        public void SetMap(TileMap newMap)
        {
            map = newMap ?? throw new ArgumentNullException("newMap");
        }

        public void Update(GameTime gameTime)
        {
            Map.Update(gameTime);
        }

        public void Draw(GameTime gameTime, SpriteBatch spriteBatch)
        {
            Map.Draw(gameTime, spriteBatch, camera);
        }

        #endregion
    }
}
```

There are a few fields, most of them static. That is so 'i don't need an instance to access the values. The first describes the area of the screen that we will be drawing to. ""next is the height ad width of the tiles on the screen. There is a non-static TileMap Finally there is a static Camera field. There are properties to expose the fields.

There are two constructors in this class. The first takes a Rectangle that defines the view port that we will be drawing to. It sets the view port and the tile width and height. The second constructor takes the view port but also takes the width of the tiles and the height of the tiles. It calls the other constructor and sets the values.

There is a utility method that takes a Vector2 and returns a Point. This can be useful in different scenarios. There is a method that sets the map for the engine. There is an Update method that calls the Update method of the TileMap class. Finally, there is a Draw method that calls the Draw method of the TileLayer class.

Before I get to the TileMap class I want to add the Draw method to the TileLayer class. Open that class and add the following method to the Method region.

```
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;

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

    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;

            spriteBatch.Draw(
                tileSet.Textures[tile.TileSet],
                destination,
                tileSet.SourceRectangles[tile.TileIndex],
                Color.White);

        }
    }

    spriteBatch.End();
}
```

This is the meat of the tile engine. It is where the tiles are drawn to the screen. First, it will only draw if it is Visible. It then grabs the position of the camera as a point and the position of the end of the screen based on the position of the camera. I then calculate then minimum tile which is the Max of 0 and the position of the X or Y coordinate minus one. This is because we don't want to draw below 0. The maximum tile we want to draw is the minimum of the view

port plus one and the width or height of the map. Limiting the coordinates this way increases the efficiency of the tile engine.

Next I create a Rectangle for the destination and a Tile for drawing with. Now I call Begin to start drawing. The sort mode is set to Deferred. BlendState.AlphaBlend which allows for transparency. The next one, SamplerState.PointClamp, controls how the texture is sampled. It prevents tearing and bleeding when drawing the tiles. The next three parameters are unused. The last is the transformation matrix that will translate the tiles from tile space to screen space.

Following that are two nested for loops that loop over the visible tiles. I set the Y coordinate of the destination tile before going into the inner for loop, a minor optimization. Inside the inner loop I get the tile using the GetTile method. If either are -1 I go to the next tile. I then set the X coordinate of the destination. Finally I draw the tile. The last thing to do is end the SpriteBatch.

That leaves the TileMap class. Right click the TileEngine folder, select Add and then Class. Name the new class TileMap. Here is the code.

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

namespace ShadowMonsters.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

        public string MapName
        {
            get { return mapName; }
            private set { mapName = value; }
        }
```

```
public TileSet TileSet
{
    get { return tileSet; }
    set { tileSet = value; }
}

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

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


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

public int MapWidth
{
    get { return mapWidth; }
}

public int MapHeight
{
    get { return mapHeight; }
}

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

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

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

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

        }
        #endregion

}
```

There are several fields in the class. There is one for the name of the map. There is then four TileLayer fields, one for each of the layers. There are also fields for the width and height of the map and the tile set.

There are properties to expose the fields. I also added a property that returns the width of the map in pixels as well as one that returns the height of the map in pixels. They are calculated using the Engine class that we created earlier.

There are three constructors for this class. The first is for future use and currently does nothing, The second is private and sets the tile set and map name to the values passed in. It will be used in the future as well. The last constructor takes as parameters the tile set, the ground layer, edge layer, building layer, decoration layer and map name. It sets the fields to the values passed in. It also sets the map width and height using the ground layer. Currently the tile engine assumes that all layers are the same size. You can change this but it is best to keep them the same.

There are Set and Get methods for each of the layers. They are used to get the tile at the selected index or set it, There are also Fill methods for each of the layers that will fill them with the tile information passed in. The Update method checks if each of the layers is null. If it is not null it calls the update method of the layer. The same is true for the Draw method but for drawing instead of updating.

The last thing I'm going to tackle in this tutorial is creating a map and rendering it. That will take place in the Game1 class that is created by MonoGame. Change the Game1 class to the following.

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using ShadowMonsters.TileEngine;
using System;

namespace ShadowMonsters
{
    /// <summary>
    /// This is the main type for your game.
    /// </summary>
    public class Game1 : Game
    {
        public static Random Random = new Random();
        private readonly GraphicsDeviceManager graphics;
        private SpriteBatch spriteBatch;
        private readonly Engine engine = new Engine(new Rectangle(0, 0, 1280, 720));
        private TileMap map;
```

```
        public Game1()
        {
            graphics = new GraphicsDeviceManager(this)
            {
                PreferredBackBufferWidth = 1280,
                PreferredBackBufferHeight = 720
            };

            graphics.ApplyChanges();

            Content.RootDirectory = "Content";
        }

        /// <summary>
        /// Allows the game to perform any initialization it needs to before starting to run.
        /// This is where it can query for any required services and load any non-graphic
        /// related content.  Calling base.Initialize will enumerate through any components
        /// and initialize them as well.
        /// </summary>
        protected override void Initialize()
        {
            // TODO: Add your initialization logic here

            base.Initialize();
        }

        /// <summary>
        /// LoadContent will be called once per game and is the place to load
        /// all of your content.
        /// </summary>
        protected override void LoadContent()
        {
            // Create a new SpriteBatch, which can be used to draw textures.
            spriteBatch = new SpriteBatch(GraphicsDevice);

            // TODO: use this.Content to load your game content here
            TileSet set = new TileSet();
            set.TextureNames.Add("tileset1");
            set.Textures.Add(Content.Load<Texture2D>(@"Tiles\tileset16-outdoors"));

            TileLayer groundLayer = new TileLayer(100, 100, 0, 1);
            TileLayer edgeLayer = new TileLayer(100, 100);
            TileLayer buildingLayer = new TileLayer(100, 100);
            TileLayer decorationLayer = new TileLayer(100, 100);

            for (int i = 0; i < 1000; i++)
            {
                decorationLayer.SetTile(Random.Next(0, 100), Random.Next(0, 100), 0,
Random.Next(2, 4));
            }

            map = new TileMap(set, groundLayer, edgeLayer, buildingLayer, decorationLayer,
"level1");

            engine.SetMap(map);
        }

        /// <summary>
        /// UnloadContent will be called once per game and is the place to unload
        /// game-specific content.
```

```
        /// </summary>
        protected override void UnloadContent()
        {
            // TODO: Unload any non ContentManager content here
        }

        /// <summary>
        /// Allows the game to run logic such as updating the world,
        /// checking for collisions, gathering input, and playing audio.
        /// </summary>
        /// <param name="gameTime">Provides a snapshot of timing values.</param>
        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
            engine.Update(gameTime);
            base.Update(gameTime);
        }

        /// <summary>
        /// This is called when the game should draw itself.
        /// </summary>
        /// <param name="gameTime">Provides a snapshot of timing values.</param>
        protected override void Draw(GameTime gameTime)
        {
            GraphicsDevice.Clear(Color.CornflowerBlue);

            // TODO: Add your drawing code here

            base.Draw(gameTime);
            engine.Draw(gameTime, spriteBatch);
        }
    }
}
```

The first addition to the class was a using statement for the System name space. I added it so that I could create a Random field for generating random numbers. The field I created is static so I can share it with other classes. The other fields that I added are an Engine field and a TileMap field. The Engine is set to 720p resolution. I did that because I'm cross platform and want to reach as many devices as possible. It's true that most devices are 1080p now but that is way too big for this game.

I updated the constructor to set the resolution for the game to 720p as well, It's important to remember to call ApplyChanges after you make any resolution change.

In the LoadContent method is where the big changes are. I create a TileSet object and add the texture name and text to the tile set. I then create the four layers. For the ground layer I use the constructor that accepts a tile index and tile set for additional parameters. I set them to 0, 1 which is a grass tile. For the others I just pass the height and width. Next I loop a thousand times and call SetTile on the decoration layer passing in random X and Y coordinates and random tile index.

The Update method just calls the Update method of the Engine field. The Draw method just calls the Draw method of the Engine field as well, If you build and run the game now you should see something similar to the following.

I'm going to call it here, this is a lot to digest at once. I'm going to start work on the next tutorial. I'm not sure what the topic will be.

Keep checking back on the blog for news on that tutorial. I hope to have it up in the next week or so. If you don't want to have to keep visiting the site to check for new tutorials you can sign up for my newsletter on the site and get a weekly status update of all the news from Game Programming Adventures.

I wish you the best in your MonoGame Programming Adventures!