A Summoner's Tale – MonoGame Tutorial Series

Chapter 3

Tile Engine and Game Play State

This tutorial series is about creating a Pokemon style game with the MonoGame Framework called A Summoner's Tale. 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 web site: A Summoner's Tale. The source code for each tutorial will be available as well. I will be using Visual Studio 2013 Premium for the series. The code should compile on the 2013 Express version and Visual Studio 2015 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 add a link to my site, http://gameprogrammingadventures.org, and credit to Jamie McMahon.

In this tutorial I will be adding the game state for the player exploring the map and the tile engine to render the map. Why do you need a tile engine though? You could just draw the map in an image editor, load that and draw it. The reason is memory and efficiency. Most of the map will be made up of the same background images. Creating a tile based on that and rendering just that saves a lot of memory. It is also more efficient to load a few hundred small images than one extremely large image. The same is true for rendering.

First, right click the Avatars project, select Add and then New Folder. Name this new folder TileEngine. Now right click the TileEngine folder, select Add and then Class. Name this new class TileSet. Here is the code for that class.

```
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;

namespace Avatars.TileEngine
{
    public class TileSet
    {
        public int TilesWide = 8;
        public int TilesHigh = 8;
        public int TileWidth = 64;
        public int TileHeight = 64;

        #region Fields and Properties

        Texture2D image;
```

```
        string imageName;
        Rectangle[] sourceRectangles;

        #endregion

        #region Property Region

        [ContentSerializerIgnore]
        public Texture2D Texture
        {
            get { return image; }
            set { image = value; }
        }

        [ContentSerializer]
        public string TextureName
        {
            get { return imageName; }
            set { imageName = value; }
        }

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

        #endregion

        #region Constructor Region

        public TileSet()
        {
            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)
        {
            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
        }
    }
```

First question then is then “What is a tile set?”. A tile set is an image that is made up of the individual tiles that will be placed on the map. When rending the map you will pick out the individual tile using the image and a source rectangle. This source rectangle defines the X and Y coordinates of the tile and the height and width of the tile. You will see this in practice shortly.

I first have 4 public member variables in this class that hold the basics of the tile set. They are TilesWide for the number of tiles across the image, TilesHigh for the number of tiles down the image, TileWidth for the width of each tile and TileHeight for the height of each tile. There are private member variables for the texture for the image, the name of the image and the source rectangles that describe each tile.

You will see that I marked some of the properties with attributes. These attributes control if the member variable will be serialized or not. For serialization to work correctly you may need the actual XNA Framework installed, depending on the version of MonoGame that you are using. We will cross that bridge in the future when we get to that point. These attributes control if the property will be serialized or not when using IntermediateSerializer. Again, I'll explain this better when we actually use it.

There is a property that exposes the name of the texture and the actual texture. There is also a readonly property that exposes the source rectangles for the tile set.

I've included two constructors in this class. The parameterless one is meant for deserializing the tile set when importing it into the game. It uses the default values that I assigned to the member variables to create the source rectangles. It will also read these values from a serialized tile set and use those instead. The rectangles are calculate inside of a nested for loop. The X coordinate is found using the width of each tile and the index for the inner loop. The Y coordinate is found using the height of each tile and the index of the outer loop. It is important to remember that Y is the outer loop and X is in the inner loop or the source rectangles will come out rotated.

The next class that I'm going to add is for a basic 2D camera that will assist in rendering only the viewable portion of that map. By drawing only the visible portion of the map we are increasing the efficiency or the rendering process. For example, say we have a map that is 200 tiles by 200 tiles. That is 40000 tiles that would need to be drawn each frame of the game. If the screen can only display 40 by 30 plus one in each direction you can get by by drawing just 1200 tiles, actually a little more because we need to pad for the right and bottom edges. That is much quicker to render than the entire map as you can see.

Now, 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 Avatars.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);
        }
    }
}
```

There are two member variables in this class. They are position and speed. Position, as you've probably already gathered, is the position of the camera on the map. Speed is the speed at which the camera moves, in pixels. I also added properties that expose both of these member variables. In the Speed property I clamp the value between a minimum and maximum values, 1px and 16px. 4px would probably be better than 1px because that would be extremely slow.

There is a third property, Transformation, that returns a translation matrix. This matrix will be used to adjust where the map is drawn on the screen. You will notice that it uses -Position. This is because the camera's position is subtract from the map coordinates being drawn.

There are two public constructors. The first is parameterless and just sets the speed to a fixed value. The second takes as a parameter the position of the camera. It then sets the position of the camera using the value and sets the default speed of the camera.

There is also one method called LockCamera. What it does is clamp the X and Y coordinates between 0 and the width of the map minus the width of the viewport for width and 0 and the height of the map minus the height of the viewport for height. I subtract the height and width of the viewport so that we do not go past the right or bottom edge. Otherwise the camera would move to the width or height of the map and the default background color would show.

The next thing that I will add is a class that represents a layer in the map. Using layers allows you to separate duties. I typically include 4 layers on a map. The first layer is the base terrain for the map. The second layer is used for transition tiles. What I mean by this is that if you have a grass and a mud tile side by side there is an image that blends the two together. These can be made up of both tiles or one of the tiles with transparent areas. I then have a layer for solid objects, such as buildings, trees, and other objects. The last layer is for decorations. These tiles are used to add interest to the map. You of course are not limited to just these four layers but they are what I've included in this tutorial.

Now, right click the Tile Engine folder, select Add and then Class. Name this new class TileLayer. Here is the code for that class.

```
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;

namespace Avatars.TileEngine
{
    public class TileLayer
    {
        #region Field Region

        [ContentSerializer(CollectionItemName = "Tiles")]
        int[] tiles;

        int width;
        int height;

        Point cameraPoint;
        Point viewPoint;
        Point min;
        Point max;

        Rectangle destination;
```

```
        #endregion

        #region Property Region

        [ContentSerializerIgnore]
        public bool Enabled { get; set; }

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

        public TileLayer(int width, int height)
            : this()
        {
            tiles = new int[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] = 0;
                }
            }
        }

        public TileLayer(int width, int height, int fill)
            : this()
        {
            tiles = new int[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] = fill;
                }
            }
        }

        #endregion

        #region Method Region

        public int GetTile(int x, int y)
        {
            if (x < 0 || y < 0)
                return -1;

            if (x >= width || y >= height)
                return -1;

            return tiles[y * width + x];
        }

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

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

            tiles[y * width + x] = 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);
            int 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 == -1)
                        continue;

                    destination.X = x * Engine.TileWidth;

                    spriteBatch.Draw(
                        tileSet.Texture,
                        destination,
                        tileSet.SourceRectangles[tile],
                        Color.White);

                }
            }

            spriteBatch.End();
        }

        #endregion
    }
}
```

A few member variables here. First, is an array of integers that holds the tiles for the layer. Instead of creating a two dimensional array I created a one dimensional array. I will use a formula to find the tile at the requested X and Y coordinates. I will also use the convention that if a tile has a value of -1 then that tile will not be drawn. Next there are width and height member variables that describe the height and width of the map.

Next up for member variables are four Point variables: cameraPoint, viewPoint, min and max. I created these at the class level because I will be using them each time that the map is drawn. Doing it this way just means that we are not constantly creating and destroying variables each frame. It is not a great optimization but even minor optimizations will help in a game. I will discuss their purpose further when I get to rendering.

The last member variable is a Rectangle variable and did it this way for the same reason as the others. The thing to note though is that this will be used every time a tile is drawn as it represents the destination the tile will be drawn in screen space. This is more effective than the other optimization. The reason is we will be drawing 4 layers with approximately 800 tiles a layer 60 times per second. Compared to 4 layers 60 times per second.

There are two auto implemented properties Enabled and Visible that control if the layer is enabled and visible. I added Enabled because I'm considering adding in animated tiles and to support that I would require an Update method. Visible will determine if the layer should be drawn or not. I also added readonly properties to expose the width and height of the layer.

There are four constructors in this class. The first has no parameters and just sets the auto properties. It is also private and will not be called outside of the class. Its main purpose is for use with the

IntermediateSerializer class what is used for serializing and deserializing content. The other three constructors are used for creating layers. The one takes an array as well as height and width parameters, the second just takes height and width parameters and the third takes height, width and fill parameters. Each of them also calls this() to initialize the Visible and Enabled properties.

The constructor that takes an array as a parameter creates a clone of the array, which is a shallow copy of the array. It then sets the width and height parameters. The constructor that takes just the width and height of the layer first creates a new array that is height * width. Next there are nested loops where y is the outer loop and x is the inner loop. To calculate where a tile is places I use the formula y * width + x. So the first row will be from 0 to 1 * width - 1, the second from 1 * width to 2 * width – 1 and then 2 * width to 3 * width -1 and so on. This formula will has to be applied consistently or you will have very strange behaviour. The third works the same as the first but rather than setting the tile to a default value it sets the tile to the value passed in.

Next are two method GetTile and SetTile. Their name definitely describes their purpose as they get and set tiles respectively. GetTile returns a value where as SetTile receives and additional value. There are if statements in each method to check that the x and y values that are passed in are within the bounds of the map. Both also use the same formula as in the constructor to get and set the tile.

I included an Update method that accepts a GameTime parameter. This parameter holds the amount of time passed between frames and has some useful purposes. Currently it checks the Enabled property is false and if it is exits the method.

The Draw method is where the meat of this class is as it actually draws the tiles. You will get some errors here because it relies on some static methods for a class I have not implemented yet, Engine. Engine just holds some common properties for the tile engine.

First, I check to see if the layer is visible or not. If it is not visible I exit the function. I then set the cameraPoint and viewPoint vectors using Engine.VectorToCell. What this call does is takes a point in pixels and returns the tile that the pixel is in. These vectors will be used to determine where to start and stop drawing tiles. As I mentioned we will only be drawing the visible tiles. Actually it will be plus and minus one tile to account for the scenario where the pixel is not the X and Y coordinates of a tile. The viewPoint vector is found by taking the camera position and adding the size of the view port the map is being drawn to. The view port is typically the entire screen in these types of games but occasionally you will find a side bar or bottom bar that holds information that reduces the map space.

The min vector is clamped between 0 and the camera position minus 1 tile. Similarly max is clamped between the view port X and Y plus 1 tile. After that I create a new instance for the destination Rectangle member variable.

The call to SpriteBatch.Begin is interesting. I use SpriteSortMode.Deferred because I am drawing a lot of images within the batch. BlendState.AlphaBlend allows for tiles that have transparency with in them. This include fully transparent and partially transparent, if the exported image supports it. SamplerState.PointClamp is used to prevent strange behaviour when drawing the map. This includes line around the tiles when scrolling the map. This SamplerState is always recommended when using source rectangles when drawing.

Next are the next for loops that actually render the layer. The tiles are drawn from the top left corner across the screen then down to the bottom right corner. You can experiment with different loops to see how it affects the way the layer is rendered.

Outside of the inner loop I set the Y coordinate of the destination rectangle as it remains the same for the entire row. Inside the loop I call GetTile passing in the X and Y coordinates of the tile. If the tile is -1 I just move to the next iteration of the inner loop. I then set the X coordinate of the tile. Finally I draw the tile using the overload that requires a texture, source rectangle, destination rectangle and tint color.

That, at a very high level, is how a tile layer is rendered. There are likely a few more optimizations that could be made such as accessing the array directly rather than relying on GetTile. I just used that method to make sure that I calculate the tile in the one dimensional array correctly.

Now I'm going to add a class that represents the entire map for the game. Right click the TileEngine folder, select Add and then Class. Name this new class TileMap. Add the following code to the TileMap class.

```
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;

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

        string mapName;
        TileLayer groundLayer;
        TileLayer edgeLayer;
        TileLayer buildingLayer;
        TileLayer decorationLayer;
        Dictionary<string, Point> characters;

        [ContentSerializer]
        int mapWidth;

        [ContentSerializer]
        int mapHeight;

        TileSet tileSet;

        #endregion

        #region Property Region

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

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

        [ContentSerializer]
        public Dictionary<string, Point> Characters
        {
            get { return characters; }
            private set { characters = 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.characters = new Dictionary<string, Point>();
            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 index)
        {
            groundLayer.SetTile(x, y, index);
        }

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

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

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

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

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

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

        public int 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);
                }
```

```
            }
        }

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

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

The class is pretty simple. It is has a member variable for the name of the map, member variables for the layers for the map, the height and width of the map along with a Dictionary<string, Point> that holds the names of NPCs on the map and their coordinates, in tiles. I will be getting to NPCs in a future tutorial. There is also a member variable for the tile set that is used for drawing the map.

There is a property for each of the member variables that exposes them to other classes. Most of these are public getters with private setters. They are also marked so that they will be serialized using the IntermediateSerializer.

There are two public get only properties that expose the width of the map in pixels and the height of the map in pixels. They are used in different places to determine if an object is inside the map or outside the map.

There are three constructors for this class. The first is a private constructor with no parameters. The no parameter constructor is required for deserializing objects. The second constructor takes as parameters the tile set for the map and the name of the map. It just initializes the character dictionary and member variables with the values passed in.

The third takes those parameters as well as four layers. It calls the two parameter constructor so that it will initialize those member variables rather than replicate the code in this constructor. It then sets the width and height members using the height and width of the ground layer.

There is a GetTile and SetTile method for each of the layers. They just provide the functionality to update the layers without exposing the layers themselves. This is done to promote good object-oriented programming. The user of the class does not need to know about the objects with in the class to work with them. They instead use the public interface that is exposed. I don't use the term as the keyword interface. I use it to mean a contract that is published to user so they can interact with the class.

There are also Fill methods for the layers that fill the building, edge and decoration layers with -1. This is useful when creating maps in the editor. I will be providing the editor as part of the series but will not be writing tutorials on how to create it.

Finally, the Update and Draw methods check to make sure the layers are not null and then call the Update and Draw method of that layer. We should probably be a little more diligent when calling the other methods that will through a null value exception when the layer does not have a value. I will leave that as an exercise for you to implement in your game.

Now I'm going to implement the the Engine class. So, right click the TileEngine folder, select Add and then Class. Name this new class Engine. Here is the code for that class.

```
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;
using Avatars.TileEngine;

namespace Avatars.TileEngine
{
    public class Engine
    {
        private static Rectangle viewPortRectangle;

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

```
        private TileMap map;

        private static float scrollSpeed = 500f;

        private static Camera camera;

        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; }
        }

        #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)
        {
            if (newMap == null)
            {
```

```
                throw new ArgumentNullException("newMap");
            }

            map = newMap;
        }

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

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

        #endregion
    }
}
```

For this I'm going on the premise that there is only one map and one instance of Engine at a time. For that reason I've included some public static members to expose values to other classes outside of the tile engine.

The first three member variables hold a Rectangle that describes the view port, the width of tiles on the screen and the height of tiles on the screen. The next member holds the map that is currently in use. The last two member variables hold scrollSpeed that determines how fast the map scrolls. You'll think that 500 pixels is really fast. This is just a multiplier though and not an actual speed. It is not used in this tutorial but will be in a future tutorial so I've left it in.

Static properties expose the tile width and height, the view port rectangle and the camera. There is also a property that exposes the map as well.

There are two constructors for this class. The first accepts a Rectangle that represents the visible area of the screen the map will be drawn to. It sets that value to the appropriate member variable and then sets the tile width and height on the screen to be 64 pixels.

The second constructor takes the same Rectangle for the screen space but also takes the width and height of the tiles on the screen. It calls the first constructor to set the view port and then sets the tile width and tile height member variables. That is done simply by dividing the X value by the tile width and the Y value by the tile height.

Next you will find the public static method VectorToCell that you saw earlier that takes a vector which represents a point on the map measured in pixels that returns what tile the pixel is in.

There is also a method called SetMap that accepts a TileMap parameter. It checks to see if it is null and if it is throws an exception as the map cannot be null. If it isn't null it sets the map member variable to be the map passed in.

The last two methods are the Update and Draw methods. The Update method takes a GameTime parameter and just calls the Update method of the map. The Draw method takes GameTime as well as a SpriteBatch object that will be used to render the map.

That wraps up the tile engine. It is rather plain but it will get our job done nicely. The last thing that I'm going to add today is a basic game play state. It won't do much as the tutorial is already pretty

long but I will pick up with it in the next tutorial.

Now, right click the GameStates folder, select Add and then Class. Name this new class GamePlayState. Here is the code for that state.

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

namespace Avatars.GameStates
{
    public interface IGamePlayState
    {

    }

    public class GamePlayState : BaseGameState, IGamePlayState
    {
        Engine engine = new Engine(Game1.ScreenRectangle, 64, 64);

        public GamePlayState(Game game)
            : base(game)
        {
            game.Services.AddService(typeof(IGamePlayState), this);
        }

        public override void Initialize()
        {
            base.Initialize();
        }

        protected override void LoadContent()
        {
        }

        public override void Update(GameTime gameTime)
        {
            base.Update(gameTime);
        }

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

It is a pretty stripped down version of a game state. I included the interface that this class will implement. Currently there is nothing in the interface but that will most definitely change in this for this class. I next included an Engine member variable that I initialize using the static ScreenRectangle property that is exposed by the Game1 class.

The constructor just registers this instance of the GamePlayState as a service using the interface that was included at the start of the class. Currently it does not do anything but will be required in the near future so I made sure to keep it in.

The constructor then registers the instance as a service with the game that can be retrieved as

needed by other states. I also added method stubs for the main DrawableGameComponent methods, Initialize, Load, Update and Draw.

I'm going to end the tutorial here because we've covered a lot in this one. In the next tutorial I will wire the game play state to open from the main menu state and create an render a map. Please stay tuned for the next tutorial in this series. If you don't want to have to keep checking for new tutorials you can sign up for my newsletter on the site and get a weekly status update of all the news for Game Programming Adventures

I wish you the best in your MonoGame Programming Adventures!
Jamie McMahon