A Summoner's Tale – MonoGame Tutorial Series

Chapter 4

Exploring the Map

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'm going to be working on the game play state that we added in the last tutorial. First, I'm going to cover wiring the game so that we can change from the menu state to the game state. I will then have the game play state render a map that we will code on the fly. Once the map is rendering I will add in scrolling the map.

To get started open up the Game1 class. Replace the fields, properties and constructor with the following.

```
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

        GameStateManager gameStateManager;

        ITitleIntroState titleIntroState;
        IMainMenuState startMenuState;
        IGamePlayState gamePlayState;

        static Rectangle screenRectangle;

        public SpriteBatch SpriteBatch
        {
            get { return spriteBatch; }
        }

        public static Rectangle ScreenRectangle
        {
            get { return screenRectangle; }
        }

        public ITitleIntroState TitleIntroState
        {
            get { return titleIntroState; }
```

```
        }

        public IMainMenuState StartMenuState
        {
            get { return startMenuState; }
        }

        public IGamePlayState GamePlayState
        {
            get { return gamePlayState; }
        }

        public Game1()
        {
            graphics = new GraphicsDeviceManager(this);
            Content.RootDirectory = "Content";

            screenRectangle = new Rectangle(0, 0, 1280, 720);

            graphics.PreferredBackBufferWidth = ScreenRectangle.Width;
            graphics.PreferredBackBufferHeight = ScreenRectangle.Height;

            gameStateManager = new GameStateManager(this);
            Components.Add(gameStateManager);

            this.IsMouseVisible = true;

            titleIntroState = new TitleIntroState(this);
            startMenuState = new MainMenuState(this);
            gamePlayState = new GamePlayState(this);

            gameStateManager.ChangeState((TitleIntroState)titleIntroState, PlayerIndex.One);
        }
```

The first change is that I've added a field for the game play state. I then added a property that exposes the IGamePlayState interface. Finally, in the constructor I create the state.

I'm going to make a few tweeks to the game play state. Open the GamePlayState file and update it to the following code.

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

namespace Avatars.GameStates
{
    public interface IGamePlayState
    {
        void SetUpNewGame();
        void LoadExistingGame();
        void StartGame();
    }

    public class GamePlayState : BaseGameState, IGamePlayState
    {
        Engine engine = new Engine(Game1.ScreenRectangle, 64, 64);
        TileMap map;
        Camera camera;
```

```
        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);
        }

        public void SetUpNewGame()
        {
        }

        public void LoadExistingGame()
        {
        }

        public void StartGame()
        {
        }
    }
}
```

First, I added some using statements to bring a few of MonoGame/XNA classes into scope in this class. I then updated the interface to include three method signatures, SetUpNewGame, LoadExistingGame and StartGame. The first two are called to create a new game or load an existing game. The third will be called once the other two have finished to start a game. I added in two new fields, map and camera, for a TileMap and Camera respectively. I also implemented the interface method, SetUpNewGame, LoadExistingGame and StartGame.

Next, I'm going to add the transition from the menu to the game play. Open the MainMenuState file and replace the Update method with the following version.

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

            if (Xin.CheckKeyReleased(Keys.Space) || Xin.CheckKeyReleased(Keys.Enter) ||
(menuComponent.MouseOver && Xin.CheckMouseReleased(MouseButtons.Left)))
            {
                if (menuComponent.SelectedIndex == 0)
                {
                    Xin.FlushInput();

                    GameRef.GamePlayState.SetUpNewGame();
                    GameRef.GamePlayState.StartGame();
```

```
                    manager.PushState((GamePlayState)GameRef.GamePlayState,
PlayerIndexInControl);
                }
                else if (menuComponent.SelectedIndex == 1)
                {
                    Xin.FlushInput();

                    GameRef.GamePlayState.LoadExistingGame();
                    GameRef.GamePlayState.StartGame();
                    manager.PushState((GamePlayState)GameRef.GamePlayState,
PlayerIndexInControl);
                }
                else if (menuComponent.SelectedIndex == 2)
                {
                    Xin.FlushInput();
                }
                else if (menuComponent.SelectedIndex == 3)
                {
                    Game.Exit();
                }
            }

            base.Update(gameTime);
        }
```

What changed is I updated the if statements where the selected index is 0 or if the selected index is 1. Since 0 is the new game option I call the SetUpNewGame and StartGame methods on the game play state. I then push the game play state onto the game state manager. Similarly, for the other state I call LoadExistingGame instead of SetUpNewGame. Otherwise, the code is the same for both.

What we are going to need next is a tile set. I've uploaded one to my site here. I ended up merging two sets into one. The building tiles were grabbed from a public domain tile set that I found on OpenGameArt.org here, http://opengameart.org/content/town-tiles. I recommend that you visit this site where your developing games. You can find some awesome art work with various licenses. You can even some of the assets as placeholders until you find exactly what it is you are looking for.

Once you've downloaded to the tile set open the content manager. First add a new folder to the content folder called Tiles. Now select the Tiles folder and select Add Existing Item. Navigate to the tileset1.png file and add it. Before closing the content manager make sure that you rebuild the content.

Now, go back to the GamePlayState.cs file in the solution. Update the Draw and SetUpNewGame methods as follows.

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

            if (map != null && camera != null)
                map.Draw(gameTime, GameRef.SpriteBatch, camera);
        }

        public void SetUpNewGame()
        {
            Texture2D tiles = GameRef.Content.Load<Texture2D>(@"Tiles\tileset1");
            TileSet set = new TileSet(8, 8, 32, 32);
            set.Texture = tiles;

            TileLayer background = new TileLayer(200, 200);
            TileLayer edge = new TileLayer(200, 200);
```

```
            TileLayer building = new TileLayer(200, 200);
            TileLayer decor = new TileLayer(200, 200);

            map = new TileMap(set, background, edge, building, decor, "test-map");

            map.FillEdges();
            map.FillBuilding();
            map.FillDecoration();

            camera = new Camera();
        }
```

In the Draw method I check if the map and camera parameters are both not null because they are required to draw the map. If they are not null I call the Draw method of the TileMap class passing in the GameTime parameter to this Draw method, the map member variable and camera member variable.

In the SetUpNewGame method I load the image for the tile set into the tiles variable. I then create a new tile set using the parameters 8, 8, 32 and 32 because the tile set is 8 tiles width, 8 tiles high with a tile width and height of 32 pixels. A quick note here. When you are creating assets like this it is best if your images have the same height and width with the same dimensions and they are a power of 2. This allows for loading on the GPU which speeds up rendering. After creating the tile set I assign the Texture property the tile set that I just loaded.

Next I create the four layers each map has with the same height and width. I then create a map object using the tile set, the four layers and call it test-map. After that I call the Fill methods to set all of the tiles for the layers above the background tiles to -1 so nothing will be drawn for those layers. Since we need a camera to draw a map I create a new instance of the camera as well.

If you build and run the game you will be shown the title screen. Dismissing the title screen will bring you to the menu. Now you can select the New Game option to be taken to the game play screen with a field of grass tiles being drawn. Since the destination tiles are bigger than the source tiles there is a bit of distortion in the rendered tiles. You can fix that by changing the engine to have width and height of 32 instead of 64.

Next I'm going to tackle scrolling the map. Still in the GamePlayState add the following using statement to bring the input manager into scope and and replace the Update method with the following.

```
using Avatars.Components;

        public override void Update(GameTime gameTime)
        {
            Vector2 motion = Vector2.Zero;

            if (Xin.KeyboardState.IsKeyDown(Keys.W) && Xin.KeyboardState.IsKeyDown(Keys.A))
            {
                motion.X = -1;
                motion.Y = -1;
            }
            else if (Xin.KeyboardState.IsKeyDown(Keys.W) &&
Xin.KeyboardState.IsKeyDown(Keys.D))
            {
                motion.X = 1;
                motion.Y = -1;
            }
            else if (Xin.KeyboardState.IsKeyDown(Keys.S) &&
Xin.KeyboardState.IsKeyDown(Keys.A))
```

```
            {
                motion.X = -1;
                motion.Y = 1;
            }
            else if (Xin.KeyboardState.IsKeyDown(Keys.S) &&
Xin.KeyboardState.IsKeyDown(Keys.D))
            {
                motion.X = 1;
                motion.Y = 1;
            }
            else if (Xin.KeyboardState.IsKeyDown(Keys.W))
            {
                motion.Y = -1;
            }
            else if (Xin.KeyboardState.IsKeyDown(Keys.S))
            {
                motion.Y = 1;
            }
            else if (Xin.KeyboardState.IsKeyDown(Keys.A))
            {
                motion.X = -1;
            }
            else if (Xin.KeyboardState.IsKeyDown(Keys.D))
            {
                motion.X = 1;
            }

            if (motion != Vector2.Zero)
            {
                motion *= camera.Speed;
                camera.Position += motion;
                camera.LockCamera(map, Game1.ScreenRectangle);
            }

            base.Update(gameTime);
        }
```

I have a local variable, motion, that will hold what direction the player wants to try and move the map. I've implemented moving the map using the W, A, S and D keys. It allows for scrolling left, right, up, down and diagonals. That is why there are a series of if and else if statements, one for each of the directions. Before I cover the if statements a brief introduction to screen space. Screen space starts with the coordinates (0, 0) in the upper right corner increasing going down and right. To move up you subtract from the Y coordinate and add to the Y coordinate to move down. Similarly, to move left you subtract from the X coordinate and add to the X coordinate to move right.

The first direction that I check is up and left, the W and A keys. If that is true I set the X and Y value of motion to -1 and -1. Next up is W and D keys which is up and right so the X value 1 and the Y value is -1. Now down to the right is A and S with X set to -1 and Y set to 1. The last diagonal is S with D and the X and Y values of 1 and 1. For the cardinal directions: up, down, left and right. For those directions I check the W, S, A and D keys respectively. For W and S the X values are 0 and the Y values are -1 and 1 respectively. Similarly, A and D the Y values are 0 and the X values are -1 and 1 respectively.

Next there is an if statement that checks to see if the motion vector is not the zero vector. If it is not I multiply the motion vector by the camera speed and then add it to the position of the camera. Finally I call LockCamera method of the camera passing in the map object and the ScreenRectangle static property from the Game1 class.

If you build and run the game now once you reach the game play screen you will be able to move the map in all directions and the map will not scroll off the screen. You are going to notice a strange behavior on the diagonals. The map scrolls faster than in the cardinal directions. Why is that?

It has to do with vectors and their magnitudes. A two dimensional vector's magnitude is calculated by taking the square root of the sum of the squares, SQRT(X^2 + Y ^2). For a cardinal vector it will always evaluate to 1. For one of the diagonal vectors it evaluations as SQRT(2) which is greater than 1 so the map scrolls faster. How do you resolve this as the map should move at the same speed in all directions. The answer is you normalize the vector. What that means is the vector will still have the same direction but the magnitude of the vector will be 1. Fortunately the Vector2 class gives us a Normalize method and takes care of the for us. Update the if statement where I check for motion as follows.

```
            if (motion != Vector2.Zero)
            {
                motion.Normalize();
                motion *= camera.Speed;
                camera.Position += motion;
                camera.LockCamera(map, Game1.ScreenRectangle);
            }
```

So, the reason I check that the motion vector is not the zero vector is that the zero vector has no magnitude and you will be dividing by zero in the calculation and causing an exception to be thrown.

I'm going to end the tutorial here though. That is because we made good progress but the topics I want to cover now are fairly long and time consuming. I'd like to make sure that in each tutorial there is clearly defined progress now and there is something demonstrable at the end of each one.

In the next tutorial I'm going to add a component that will represent the player in the game with an animated sprite for the player. Please stay tuned for the next tutorial in this series. 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!
Jamie McMahon