Shadow Monsters – MonoGame Tutorial Series
Chapter 5
Non-Player Characters

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 to know the basics of fields, properties, methods, classes and the MonoGame framework.

In this tutorial I will be adding non-player characters for the player to interact with. First, I want to fix the issue from the last tutorial. What we need is some sprites for the shadow monsters. I'm going to use some sprites that I found on OpenGameArt.org. I highly recommend browsing their archives for resources for your own game, even if they are just place holders.

To get started browse to https://opengameart.org/content/fantasy-characters and download the characters. Decompress them to a folder. In Visual Studio open the Content Pipeline Tool. Create a new folder ShadowMonsterImages. To the ShadowMonsterImages folder add in the characters from the pack. Save and build the project.

Back in Visual Studio open the ShadowMonsters.txt file and replace the contents with the following.

```
Dark1,Brownie,Dark,100,1,9,12,10,50,0,0,Tackle:1,Block:1
Earth1,Beast,Earth,100,1,10,10,9,60,0,0,Tackle:1,Block:1
Fire1,Efreet,Fire,100,1,12,8,10,50,0,0,Tackle:1,Block:1
Light1,Paladin,Light,100,1,12,9,10,50,0,0,Tackle:1,Block:1
Water1,Siren,Water,100,1,9,12,10,50,0,0,Tackle:1,Block:1
Wind1,Harpy,Wind,100,1,10,10,12,50,0,0,Tackle:1,Block:1
```

For the shadow monsters I made some random choices based on the name. You are free to use which ever images suit you. The last thing to fix is in the LoadContent method of the GamePlayState. I missed filling the moves. Add the following line at the start of the LoadContent method.

```
            MoveManager.FillMoves();
```

Now we can focus on non-player characters. To get started right click the ShadowMonsters project in the Solution Explorer, select Add and then New Folder. Name this new folder Characters. Now right click the Characters folder, select Add and then Class. Name this new class Character. Here is the code for that class.

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

namespace ShadowMonsters.Characters
{
    public class Character
    {
        #region Constant

        public const float SpeakingRadius = 40f;
        public const int MonsterLimit = 6;

        #endregion

        #region Field Region

        protected string name;
        protected string textureName;
        protected readonly ShadowMonster[] monsters = new ShadowMonster[MonsterLimit];
        protected int currentMonster;
        protected ShadowMonster givingMonster;
        protected AnimatedSprite sprite;

        protected string conversation;

        protected static Game gameRef;

        #endregion

        #region Property Region

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

        public string SpriteName
        {
            get { return textureName; }
        }
        public AnimatedSprite Sprite
        {
            get { return sprite; }
        }

        public ShadowMonster BattleMonster
        {
            get { return monsters[currentMonster]; }
```

```
}

public ShadowMonster GiveMonster
{
    get { return givingMonster; }
}

public string Conversation
{
    get { return conversation; }
}

public bool Battled
{
    get;
    set;
}

public List<ShadowMonster> BattleMonsters => monsters.ToList<ShadowMonster>();

#endregion

#region Constructor Region

protected Character()
{
}

#endregion

#region Method Region

public bool NextMonster()
{
    currentMonster++;

    return currentMonster < MonsterLimit && monsters[currentMonster] != null;
}

public static Character FromString(Game game, string characterString)
{
    if (gameRef == null)
    {
        gameRef = game;
    }

    Character character = new Character();
    string[] parts = characterString.Split(',');

    character.name = parts[0];
    character.textureName = parts[1];
    character.sprite = new AnimatedSprite(
        game.Content.Load<Texture2D>(@"CharacterSprites\" + parts[1]),
        Game1.Animations)
    {
        CurrentAnimation = (AnimationKey)Enum.Parse(typeof(AnimationKey), parts[2])
    };
    character.conversation = parts[3];
    character.currentMonster = int.Parse(parts[4]);
```

```
            for (int i = 5; i < 11 && i < parts.Length - 1; i++)
            {
                character.BattleMonsters[i - 5] =
ShadowMonsterManager.GetShadowMonster(parts[i].ToLowerInvariant());
            }

            character.givingMonster = ShadowMonsterManager.GetShadowMonster(parts[parts.Length
- 1].ToLowerInvariant());
            return character;
        }

        public void ChangeMonster(int index)
        {
            if (index < 0 || index >= MonsterLimit)
            {
                currentMonster = index;
            }
        }

        public void SetConversation(string newConversation)
        {
            this.conversation = newConversation;
        }

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

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

        public bool Alive()
        {
            for (int i = 0; i < MonsterLimit; i++)
            {
                if (BattleMonsters[i] != null && BattleMonsters[i].CurrentHealth > 0)
                {
                    return true;
                }
            }

            return false;
        }

        #endregion
    }
}
```

There are two constants in this class. The first is, in pixels, how close the player has to be to start a conversation with the character. The second is the number of shadow monsters the character can have. I decided to go with the six from Pokemon.

There are several fields in this class. They are name, the name of the character, textureName, the name of the texture for the sprite, monsters, the shadow monsters owned by the character, currentMonster is the current monster that is battling, givingMonster is for if

the character gives or sells a monster to the player, sprite is the sprite for the character and conversation is the name of the conversation the character is currently in. More on that in a future tutorial. There is also a static Game field that is useful later on. The fields are all protected because I will be implementing other types of characters and inherit them from this class. There are properties to expose all of the fields.

There is a single protected constructor that is used to create new Character objects. Actual instantiation is done in a FromString method the same as in the ShadowMonster class.

The first method is NextMonster and is called when battling the character to move to the next shadow monster they have. It returns true if there is a next shadow monster and false if there isn't.

FromString works in the same way as ShadowMonster.FromString. It takes a string that describes the character and returns that character. The string is comma delimited and is the format character name, texture name, current animation, conversation, current monster, battle monsters and giving monster. It takes a Game parameter and a string parameter. If the gameRef field is null I set it to be the parameter passed in. I split the string on the comma into its parts. The name is set to the first part and textureName to the second. I use the Enum.Parse method to convert the string to an AnimationKey. I then create an AnimatedSprite using the texture name and a Dictionary<AnimationKey, Animation> that I haven't created yet in the Game1 class. I set the current animation to the walk down animation. The next field that is set is the conversation field followed by currentMonster. I loop 6 times setting the BattleMonster at i - 5 to be the shadow monster of the name. I do the same thing for givingMonster.

After FromString is ChangeMonster that changes the battle monster that is currently selected. It checks to make sure the index passed in is in bounds. SetConversation is used to change the current conversation. More on that later on in the series. The Update method calls the Update method of the sprite field and similarly the Draw method calls the Draw method of the sprite field. Alive checks to see if there are any shadow monsters that have health left. If there is it returns true, otherwise false.

To the Game1 class add the following field and property. Also, update the Initialize method to the following.

```
        private static Dictionary<AnimationKey, Animation> animations = new
Dictionary<AnimationKey, Animation>();
        public static Dictionary<AnimationKey, Animation> Animations => animations;

        protected override void Initialize()
        {
            // TODO: Add your initialization logic here

            Animation animation = new Animation(3, 32, 36, 0, 0);
            animations.Add(AnimationKey.WalkUp, animation);

            animation = new Animation(3, 32, 36, 0, 36);
            animations.Add(AnimationKey.WalkRight, animation);

            animation = new Animation(3, 32, 36, 0, 72);
            animations.Add(AnimationKey.WalkDown, animation);
```

```
            animation = new Animation(3, 32, 36, 0, 108);
            animations.Add(AnimationKey.WalkLeft, animation);

            Components.Add(new Xin(this));
            base.Initialize();
        }
```

The code in the Initialize method should be familiar, its the same code we've been using in the previous tutorials. It just creates the animations and adds them to the dictionary of animations.

Next we are going to extend the tile map. We are going to add a new layer that will hold the characters on the map. Right click the TileEngine folder, select Add and then class. Name this new class CharacterLayer. Here is the code for that class.

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

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

        private readonly Dictionary<Point, Character> characters;

        #endregion

        #region Property Region

        public Dictionary<Point, Character> Characters => characters;

        #endregion

        #region Constructor Region

        public CharacterLayer()
        {
            characters = new Dictionary<Point, Character>();
        }
        #endregion

        #region Method Region

        public void Update(GameTime gameTime)
        {
            foreach (Character c in characters.Values)
            {
                c.Update(gameTime);
            }
        }
```

```
        public void Draw(GameTime gameTime, SpriteBatch spriteBatch, Camera camera)
        {
            spriteBatch.Begin(
                SpriteSortMode.Deferred,
                BlendState.AlphaBlend,
                SamplerState.PointClamp,
                null,
                null,
                null,
                camera.Transformation);

            foreach (Character c in characters.Values)
            {
                c.Draw(gameTime, spriteBatch);
            }

            spriteBatch.End();
        }

        #endregion
    }
}
```

There is a single field, characters, that holds the characters on the layer. The field is a Dictionary<Point, Character> where the point is the tile the character is on. It will be used for collision detection later. There is a property to expose the field. The constructor initializes the field..

The Update method takes a GameTime parameter. It loops over all of the characters on the layer. Inside the loop it calls the Update method of the character.

The Draw method takes a GameTime parameter, SpriteBatch and Camera. It calls Begin on the SpriteBatch the same as in the TileLayer class which is why we needed a Camera. It then loops over the values in the characters field. Inside the loop I call the Draw method of the character.

The next step is to add the layer to the TileMap class. The changes were moderate but I will give the code for the entire 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;
using System.IO;

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

        string mapName;
```

```
TileLayer groundLayer;
TileLayer edgeLayer;
TileLayer buildingLayer;
TileLayer decorationLayer;
CharacterLayer characterLayer;

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; }
}

public CharacterLayer CharacterLayer => characterLayer;

#endregion

#region Constructor Region

private TileMap()
{
    characterLayer = new CharacterLayer();
}

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);

            characterLayer.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);

            characterLayer.Draw(gameTime, spriteBatch, camera);

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

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

        }
        #endregion
    }
}
```

The first change was the addition of a CharacterLayer field, characterLayer. There is a property to expose the field. I updated the private constructor to initialize the field. The second constructor now calls the first constructor. The Update method calls the Update method of the layer passing in the gameTime parameter. It doesn't matter what order the Update method is called so it is called after all of the tile layers. The Draw method calls the Draw method of the layer passing in the required parameters. I call the Draw method after the edge layer but before the building layer. This will give the illusion that the character can walk behind buildings. They also walk under decorations like walking under tall grass in Pokemon.

Let's add some sprites. Open the Content Pipeline Tool. Create a new folder called CharacterSprites. To this folder add the remaining sprites from Tutorial Two. Save the project and build it.

Remove the animations field from the GamePlayState class because we will be using the static one in the Game1 class. Update the LoadContent method to the following to add a character to the map.

```
        protected override void LoadContent()
        {
```

```
            MoveManager.FillMoves();
            ShadowMonsterManager.FromFile(@".\Content\ShadowMonsters.txt", content);
            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");

            Character c = Character.FromString(GameRef,
"Paul,ninja_m,WalkDown,PaulHello,0,fire1,,,,,,,");
            c.Sprite.Position = new Vector2(2 * Engine.TileWidth, 2 * Engine.TileHeight);

            map.CharacterLayer.Characters.Add(new Point(2, 2), c);

            engine.SetMap(map);

            sprite = new AnimatedSprite(content.Load<Texture2D>(@"Sprites\mage_f"),
Game1.Animations)
            {
                CurrentAnimation = AnimationKey.WalkDown
            };
            base.LoadContent();
        }
```

The changes were the removal of creating the animations. I create a new character using the FromString method named Paul using the ninja_m sprite. The current animation is set to WalkDown and the conversation is set to PaulHello. The current shadow monster is 0 and the only battle monster is fire1. There is no giving shadow monster.

I then set the position of the NPC to 2 times tile width and 2 time tile height, the tile at (2, 2). I then add the NPC to the collection using tile (2, 2). The reason why I use Point is for collision detection. It will be clearer shortly when I add collision detection because right now if you build and run you can walk right through Paul.

Now I'm going to implement tile based movement and collision detection with characters on the map. First, open the AnimatedSprite class to add an origin property to it. Since we are binding the sprite to the tile width and height I'm using those to calculate the origin.

```
        public Vector2 Origin
        {
            get { return new Vector2(Engine.TileWidth / 2, Engine.TileHeight / 2); }
        }
```

Now open the Engine class and add the following method. What it does is round return what

position the origin passed in is. It is used to round values to the nearest tile.

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

Now to implement the tile based movement and collision with characters. Add these two fields to the GamePlayState and change the Update method to the following.

```
        private bool inMotion;
        private Rectangle collision;

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

            if (Xin.KeyboardState.IsKeyDown(Keys.W) && !inMotion)
            {
                motion.Y = -1;
                sprite.CurrentAnimation = AnimationKey.WalkUp;
                sprite.IsAnimating = true;
                inMotion = true;
                collision = new Rectangle(
                    (int)sprite.Position.X,
                    (int)sprite.Position.Y - Engine.TileHeight * 2,
                    Engine.TileWidth,
                    Engine.TileHeight);
            }
            else if (Xin.KeyboardState.IsKeyDown(Keys.S) && !inMotion)
            {
                motion.Y = 1;
                sprite.CurrentAnimation = AnimationKey.WalkDown;
                sprite.IsAnimating = true;
                inMotion = true;
                collision = new Rectangle(
                    (int)sprite.Position.X,
                    (int)sprite.Position.Y + Engine.TileHeight * 2,
                    Engine.TileWidth,
                    Engine.TileHeight);
            }
            else if (Xin.KeyboardState.IsKeyDown(Keys.A) && !inMotion)
            {
                motion.X = -1;
                sprite.CurrentAnimation = AnimationKey.WalkLeft;
                sprite.IsAnimating = true;
                inMotion = true;
                collision = new Rectangle(
                    (int)sprite.Position.X - Engine.TileWidth * 2,
                    (int)sprite.Position.Y,
                    Engine.TileWidth,
                    Engine.TileHeight);
            }
            else if (Xin.KeyboardState.IsKeyDown(Keys.D) && !inMotion)
            {
                motion.X = 1;
                sprite.CurrentAnimation = AnimationKey.WalkRight;
                sprite.IsAnimating = true;
```

```
            inMotion = true;
            collision = new Rectangle(
                (int)sprite.Position.X + Engine.TileWidth * 2,
                (int)sprite.Position.Y,
                Engine.TileWidth,
                Engine.TileHeight);
        }

        if (motion != Vector2.Zero)
        {
            motion.Normalize();
            motion *= (sprite.Speed * (float)gameTime.ElapsedGameTime.TotalSeconds);
            Rectangle pRect = new Rectangle(
                    (int)(sprite.Position.X + motion.X),
                    (int)(sprite.Position.Y + motion.Y),
                    Engine.TileWidth,
                    Engine.TileHeight);

            if (pRect.Intersects(collision))
            {
                sprite.IsAnimating = false;
                inMotion = false;
                motion = Vector2.Zero;
            }

            foreach (Point p in engine.Map.CharacterLayer.Characters.Keys)
            {
                Rectangle r = new Rectangle(
                    p.X * Engine.TileWidth,
                    p.Y * Engine.TileHeight,
                    Engine.TileWidth,
                    Engine.TileHeight);

                if (r.Intersects(pRect))
                {
                    motion = Vector2.Zero;
                    sprite.IsAnimating = false;
                    inMotion = false;
                }
            }

            Vector2 newPosition = sprite.Position + motion;
            newPosition.X = (int)newPosition.X;
            newPosition.Y = (int)newPosition.Y;

            sprite.Position = newPosition;
            motion = sprite.LockToMap(
                new Point(
                    map.WidthInPixels,
                    map.HeightInPixels),
                motion);

            if (motion == Vector2.Zero)
            {
                Vector2 origin = new Vector2(
                        sprite.Position.X + sprite.Origin.X,
                        sprite.Position.Y + sprite.Origin.Y);
                sprite.Position = Engine.VectorFromOrigin(origin);
                inMotion = false;
                sprite.IsAnimating = false;
```

```
            }
        }

        Engine.Camera.LockToSprite(map, sprite, new Rectangle(0, 0, 1280, 720));
        sprite.Update(gameTime);

        base.Update(gameTime);
    }
```

There are lots of ways to implement tile based movement. The way I chose to do it is by creating a collision rectangle when movement starts and stop movement when the player collides with the rectangle. This is called bounding box collision detection. If the two rectangles intersect then there is a collision and movement is negated. I use the field inMotion to determine if the sprite is already moving.

I removed the old line that set motion to Vector.Zero because I will be handling that somewhere else. In the if statements that check if a key is down I added a check to see if inMotion is true because if it is we want to ignore the request. Inside the ifs I set the inMotion field to true and create a collision rectangle. If movement is up the collision is two tiles above where the player wants to go. Similarly, down is two tiles down, left is two tiles left and right is two tiles right. This is because we want the player to be able to move through the next tile, if possible, but not the tile afterwards.

If motion is not zero, meaning the player is trying to move, I create a rectangle that describes the player using the position of the sprite plus the desired direction and the tile width and height. If this rectangle intersects with the collision rectangle we are about to move into the second tile away and movement is negates.

In a foreach loop I loop over all of the characters on the map. I then create a rectangle that describes the character's position. I use the tile width and height to calculate the rectangle. I do this for the following reason. If your game supports different resolutions you might change the tile width and height. For example, at 1280 by 720 I use 64 by 64 but at 1920 by 1080 I use 128 by 128. Doing it this way allows for the differing tile width and height. If the two rectangles intersect then the movement is cancelled.

After locking to the map there is a check if motion is Vector2.Zero. If it is movement has been cancelled. I use the new VectorFromOrigin method on the Engine class to calculate the coordinates of the tile the origin of the sprite is in. The sprite's position is then set to that value. The inMotion field is set to false and the IsAnimating property of the sprite is set to false.

If you build and run now you can move around the map tile by tile and not run over Paul. I;m sure he appreciates that. Since the decoration tile is not transparent it might render over top of Paul.

I'm going to wrap this tutorial up here. I will be starting work on the next tutorial shortly. Keep checking back on the blog for news on that tutorial. I hope to have it up in the next week or so.

I wish you the best in your MonoGame Programming Adventures!