Shadow Monsters – MonoGame Tutorial Series
Chapter 11
Saving the Game

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.

Thia tutorial is going to cover saving the game. I will be taking a shot gun approach and save everything. First though there are two things that I want to add to the tile engine. I want to implement a collision layer and a portal layer. As the name implies the collision layer is to keep the player off of a tile, say one with a tree on it. The portal layer holds tiles where the player can change maps.

To get started right click the TileEngine folder, select Add and then Class. Name this new class CollisionLayer, here is the code.

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

namespace ShadowMonsters.TileEngine
{
    public enum CollisionType { Passable, Impassable, Water }

    public class CollisionLayer
    {
        private readonly Dictionary<Point, CollisionType> collisions = new Dictionary<Point, 
CollisionType>();
        private static Texture2D texture;
        public static Texture2D Texture { get => texture; set => texture = value; }
        public Dictionary<Point, CollisionType> Collisions
        {
            get { return collisions; }
        }

        public CollisionLayer()
        {
        }

        internal void Save(BinaryWriter writer)
        {
            writer.Write(Collisions.Count);
            foreach (Point p in collisions.Keys)
            {
                writer.Write((int)Collisions[p]);
                writer.Write(p.X);
                writer.Write(p.Y);
            }
        }

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

            foreach (Point p in collisions.Keys)
            {
                spriteBatch.Draw(
                    texture,
                    new Rectangle(
                        p.X * Engine.TileWidth,
                        p.Y * Engine.TileHeight,
                        Engine.TileWidth,
                        Engine.TileHeight),
                    Color.White);
            }

            spriteBatch.End();
        }
    }
}
```

There is an enumeration with the basic collision types, passable, impassable and water. You can add others such as mud that slows the player or swamp which damages the player. There are two fields: collisions and texture. The collisions field is a Dictionary<Point, CollisionType> that is the tile index of the collision and what the collision is. I went with point to support multiple resolutions. The texture field is for drawing the collision, used in the editor. There are properties to expose the value of the fields.

I included the Save method now rather than add it later. It takes a BinaryWriter that is an open writer for saving the layer. To save the layer I writer the number of collisions so I know how many to read when I load. I then write the type of collision cast as an integer. Next I write the

X coordinate of the tile followed by the Y coordinate of the tile. That is it for saving the collision layer.

The Draw method takes a SpriteBatch object and a Camera object. The call to Begin of the SpriteBatch is the same as drawing a TileLayer. It takes a transformation matrix that will translate the world coordinates to screen coordinates. It then draws the texture by creating a destination rectangle.

Now I will implement the portal layer. The first step is to create a class that represents a portal. Right click the TileEngine folder, select Add and then Class. Name this new class Portal. Here is the code.

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

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

        Point sourceTile;
        Point destinationTile;
        string destinationLevel;

        #endregion

        #region Property Region

        public Point SourceTile
        {
            get { return sourceTile; }
            private set { sourceTile = value; }
        }

        public Point DestinationTile
        {
            get { return destinationTile; }
            private set { destinationTile = value; }
        }

        public string DestinationLevel
        {
            get { return destinationLevel; }
            private set { destinationLevel = value; }
        }

        #endregion

        #region Constructor Region

        private Portal()
        {
        }

        public Portal(Point sourceTile, Point destinationTile, string destinationLevel)
        {
            SourceTile = sourceTile;
            DestinationTile = destinationTile;
            DestinationLevel = destinationLevel;
        }

        #endregion

        #region Method Region

        public void Save(BinaryWriter writer)
        {
            writer.Write(destinationLevel);
            writer.Write(sourceTile.X);
            writer.Write(sourceTile.Y);
            writer.Write(destinationTile.X);
            writer.Write(destinationTile.Y);
        }

        #endregion
    }
}
```

For fields there is the source tile, destination tile and destination level. There are properties to expose the values. There are two constructors. The parameterless private one is for loading portals. The second takes the source tile, destination tile and destination level.

The Save method takes a BinaryWriter for writing the portal. It writes the destination level, X coordinate of the source tile, Y coordinate of the source tile, X coordinate of the destination tile and then the Y coordinate of the source tile.

Now that we have a representation of a portal we can implement the layer. Right click the TileEngine folder, select Add and then Class. Name this new class PortalLayer. Here is the code.

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

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

        private Dictionary<string, Portal> portals;
        private static Texture2D texture;

        #endregion
```

```
#region Property Region
public static Texture2D Texture { get => texture; set => texture = value; }

public Dictionary<string, Portal> Portals
{
    get { return portals; }
    private set { portals = value; }
}

#endregion

#region Constructor Region

public PortalLayer()
{
    portals = new Dictionary<string, Portal>();
}

#endregion

public void AddPortal(string name, Portal portal)
{
    if (!portals.ContainsKey(name))
    {
        portals.Add(name, portal);
    }
}

public void Save(BinaryWriter writer)
{
    writer.Write(portals.Count);

    foreach (var r in portals.Keys)
    {
        writer.Write(r);
        portals[r].Save(writer);
    }
}

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

    foreach (var s in portals.Keys)
    {
        Rectangle r = new Rectangle(
            portals[s].SourceTile.X * Engine.TileWidth,
            portals[s].SourceTile.Y * Engine.TileHeight,
            Engine.TileWidth,
            Engine.TileHeight);
        spriteBatch.Draw(texture, r, Color.Red);
    }
```

```
            spriteBatch.End();

        }
    }
}
```

Like in the collision layer there a two fields, one for the portals and one for a texture for drawing the layer. There are properties to expose the values. There is a public constructor that initializes the portals field.

The Save method first writes the number of portals. It then loops over all of the keys in the portals dictionary. Inside the loop I write the name of the portal then call the Save method of the portal.

The Draw method calls begin like before to translate world coordinates to screen coordinates. It then loops over the keys in the dictionary. Inside the loop I create a destination rectangle using the source tile. I draw the texture tinting it red to indicate a portal.

Before I get to adding in the new layers to the map I want to implement some more save methods. The first is for tile sets. Add the following method to the TileSet class. Also add the using statement for System.IO.

```
        using System.IO;
        public void Save(BinaryWriter writer)
        {
            writer.Write(imageName.Count);

            foreach (string s in imageName)
            {
                writer.Write(s);
                writer.Write(-1);
            }
            writer.Write(TilesWide);
            writer.Write(TilesHigh);
            writer.Write(TileWidth);
            writer.Write(TileHeight);
        }
```

So, it writes the number of tile sets. It then loops over the image names, writes them followed by a minus one. I like to terminate strings with a minus one to prevent overflow when reading strings. It then writes the TilesWide, TilesHigh, TileWidth and TileHeight properties.

The next Save that I will implement is the TileLayer. Add this method to the TileLayer class along with the using statement for System.IO.

```
        using System.IO;
        public void Save(BinaryWriter writer)
        {
            writer.Write(width);
            writer.Write(height);

            for (int y = 0; y < height; y++)
            {
                for (int x = 0; x < width; x++)
                {
                    writer.Write(tiles[y * width + x].TileSet);
                    writer.Write(tiles[y * width + x].TileIndex);
                }
            }
        }
```

What this method does is write the width and the height of the layer so they can be read in to load the map later. It then loops over all of the tiles and writes the tile set and the tile index.

The next save that I'm going to implement is for the collision layer. Add the using statement for System.IO and the following method to the CollisionLayer class.

```
        using System.IO;
        internal void Save(BinaryWriter writer)
        {
            writer.Write(Collisions.Count);
            foreach (Point p in collisions.Keys)
            {
                writer.Write((int)Collisions[p]);
                writer.Write(p.X);
                writer.Write(p.Y);
            }
        }
```

Similar to the Save method of the PortalLayer class. It writes the number of collisions then loops over the collisions. Inside the loop it writes the collision cast as an integer then the X coordinate and Y coordinate for the collision.

The next save that I want to implement is the ShadowMonster class. Add the following using statements and method to the ShadowMonster class.

```
        using System.IO;
        using System.Text;
        public bool Save(BinaryWriter writer)
        {
            StringBuilder b = new StringBuilder();

            b.Append(name);
            b.Append(",");
            b.Append(displayName);
            b.Append(",");
            b.Append(element);
            b.Append(",");
            b.Append(experience);
            b.Append(",");
            b.Append(costToBuy);
            b.Append(",");
            b.Append(level);
            b.Append(",");
            b.Append(attack);
            b.Append(",");
            b.Append(defense);
            b.Append(",");
            b.Append(speed);
            b.Append(",");
            b.Append(health);
            b.Append(",");
            b.Append(currentHealth);
            b.Append(",");
            b.Append(IsAsleep);
            b.Append(",");
            b.Append(IsConfused);
            b.Append(",");
            b.Append(IsParalyzed);
            b.Append(",");
            b.Append(IsPoisoned);
            b.Append(",");
            b.Append(Source.X);
            b.Append(",");
            b.Append(Source.Y);

            foreach (string s in knownMoves.Keys)
            {
                b.Append(",");
                b.Append(s);
                b.Append(":");
                b.Append(knownMoves[s].UnlockedAt);
            }

            writer.Write(b.ToString());
            return true;
        }
```

Instead of writing each individual value I concatenate the shadow monster into a string using a StringBuilder and appending the values separated by commas. I then write the string builder as a string.

The next save method that I want to save is the Character class. Add this method to the Character class.

```
        public virtual bool Save(BinaryWriter writer)
        {
            StringBuilder b = new StringBuilder();

            b.Append(name);
            b.Append(",");
            b.Append(textureName);
            b.Append(",");
            b.Append(sprite.CurrentAnimation);
            b.Append(",");
            b.Append(conversation);
            b.Append(",");
            b.Append(currentMonster);
            b.Append(",");
            b.Append(Battled);

            writer.Write(b.ToString());
            writer.Write(-1);

            foreach (ShadowMonster a in monsters)
            {
                if (a != null)
                {
                    a.Save(writer);
                    writer.Write(-1);
                }
                else
                {
                    writer.Write("*");
                    writer.Write(-1);
                }
            }

            if (givingMonster != null)
            {
                givingMonster.Save(writer);
                writer.Write(-1);
            }
            else
            {
                writer.Write("*");
                writer.Write(-1);
            }

            return true;
        }
```

This save method is similar to the ShadowMonster save method. There is a string builder to build a comma separated string. Looping over the shadow monsters I save the shadow monster if it is not null followed by a -1. If it is null I write an asterisk and a -1. I do something similar for the giving monster.

Before implementing the Save method for the Merchant class I want to add a Save method in the Item class. Add the following using statement and Save method to the Item class, it is part of the Backpack.cs file.

```
    using System.IO;

    public void Save(BinaryWriter writer)
    {
        writer.Write(Name);
        writer.Write(Count);
    }
```

This is a simple method that just writes the name of the item and the count. The next save method that I want to implement is the Backpack. Add the following code to the Backpack class.

```
    public void Save(BinaryWriter writer)
    {
        writer.Write(Items.Count);

        foreach (Item i in Items)
            i.Save(writer);
    }
```

The method writes the number of items in the backpack then iterates over the items and calls the save method of the item. The next save that I will implement is the Merchant class. Add the following using statement and method to the Merchant class.

```
using System.IO;

public override bool Save(BinaryWriter writer)
{
    base.Save(writer);
    backpack.Save(writer);

    return true;
}
```

This method overrides the method of the base class Character. It calls the base method to save the character data then calls the Save method of the Backpack. Finally it returns true.

The next thing to save is the CharacterLayer. Add the following method to the CharacterLayer.

```
public bool Save(BinaryWriter writer)
{
    writer.Write(characters.Count);

    foreach (Point p in characters.Keys)
    {
        if (characters[p] is Merchant)
        {
            writer.Write(2);
        }
        else
        {
            writer.Write(1);
        }

        writer.Write(p.X);
        writer.Write(p.Y);

        characters[p].Save(writer);
    }

    return true;
}
```

The first thing the method does is write the number of characters on the layer. It then loops over the keys in the characters dictionary. I then check if the character is a Merchant and if it is I write a 2. Otherwise I write a 1. These will be used when reading the character back in. I then write the X coordinate of the key followed by the Y coordinate of the key.

That is everything we need to save a map. Make sure you have the following using statement and add this Save method to the TileMap class.

```
using System.IO;

public bool Save(BinaryWriter writer)
{
    writer.Write(mapName);

    characterLayer.Save(writer);
    tileSet.Save(writer);
    edgeLayer.Save(writer);
    groundLayer.Save(writer);
```

```
            decorationLayer.Save(writer);
            buildingLayer.Save(writer);
            portalLayer.Save(writer);
            collisionLayer.Save(writer);

            return true;
        }
```

What the method does is write the map name then call the Save methods of the layers and the tile set. Finally it returns true.

Next up is saving the player. Add the following code to the Player class.

```
        internal void Save(BinaryWriter writer)
        {
            StringBuilder b = new StringBuilder();

            b.Append(name);
            b.Append(",");
            b.Append(gender);
            b.Append(",");
            b.Append(mapName);
            b.Append(",");
            b.Append(tile.X);
            b.Append(",");
            b.Append(tile.Y);
            b.Append(",");
            b.Append(textureName);
            b.Append(",");
            b.Append(speed);
            b.Append(",");
            b.Append(sprite.CurrentAnimation);
            b.Append(",");
            b.Append(sprite.Position.X);
            b.Append(",");
            b.Append(sprite.Position.Y);

            writer.Write(b.ToString());
            writer.Write(shadowMonsters.Count);

            foreach (ShadowMonster a in shadowMonsters)
            {
                a.Save(writer);
                writer.Write(-1);
            }

            writer.Write(selected);

            foreach (ShadowMonster a in battleShadowMonsters)
            {
                if (a != null)
                {
                    a.Save(writer);
                    writer.Write(-1);
                }
                else
                {
                    writer.Write("*");
                    writer.Write(-1);
                }
            }

            backpack.Save(writer);
        }
```

This is similar to the Character Save method. There is a StringBuilder to save character fields as a comma delimited string. Next it writes the number of shadow monsters the player has then the shadow monsters. Following that it writes the battle shadow monsters.

The last thing to do is trigger a save. That will be done in the Update method of the GamePlayState. Add the following using statement and code to the GamePlayState.

```
    using System.Security.Cryptography;
    using System.IO;

    private readonly byte[] IV = new byte[]
    {
        067, 197, 032, 010, 211, 090, 192, 076,
        054, 154, 111, 023, 243, 071, 132, 090
    };

    private readonly byte[] Key = new byte[]
    {
        067, 090, 197, 043, 049, 029, 178, 211,
        127, 255, 097, 233, 162, 067, 111, 022,
    };

    public override void Update(GameTime gameTime)
    {
        engine.Update(gameTime);
        frameCount++;
        if (Xin.KeyboardState.IsKeyDown(Keys.W) && !inMotion)
        {
            motion.Y = -1;
            Game1.Player.Sprite.CurrentAnimation = AnimationKey.WalkUp;
            Game1.Player.Sprite.IsAnimating = true;
            inMotion = true;
            collision = new Rectangle(
                (int)Game1.Player.Sprite.Position.X,
                (int)Game1.Player.Sprite.Position.Y - Engine.TileHeight * 2,
                Engine.TileWidth,
                Engine.TileHeight);
        }
        else if (Xin.KeyboardState.IsKeyDown(Keys.S) && !inMotion)
        {
            motion.Y = 1;
            Game1.Player.Sprite.CurrentAnimation = AnimationKey.WalkDown;
            Game1.Player.Sprite.IsAnimating = true;
            inMotion = true;
            collision = new Rectangle(
                (int)Game1.Player.Sprite.Position.X,
                (int)Game1.Player.Sprite.Position.Y + Engine.TileHeight * 2,
                Engine.TileWidth,
                Engine.TileHeight);
        }
        else if (Xin.KeyboardState.IsKeyDown(Keys.A) && !inMotion)
        {
            motion.X = -1;
```

```
                Game1.Player.Sprite.CurrentAnimation = AnimationKey.WalkLeft;
                Game1.Player.Sprite.IsAnimating = true;
                inMotion = true;
                collision = new Rectangle(
                    (int)Game1.Player.Sprite.Position.X - Engine.TileWidth * 2,
                    (int)Game1.Player.Sprite.Position.Y,
                    Engine.TileWidth,
                    Engine.TileHeight);
            }
            else if (Xin.KeyboardState.IsKeyDown(Keys.D) && !inMotion)
            {
                motion.X = 1;
                Game1.Player.Sprite.CurrentAnimation = AnimationKey.WalkRight;
                Game1.Player.Sprite.IsAnimating = true;
                inMotion = true;
                collision = new Rectangle(
                    (int)Game1.Player.Sprite.Position.X + Engine.TileWidth * 2,
                    (int)Game1.Player.Sprite.Position.Y,
                    Engine.TileWidth,
                    Engine.TileHeight);
            }

            if (motion != Vector2.Zero)
            {
                motion.Normalize();
                motion *= (Game1.Player.Sprite.Speed *
(float)gameTime.ElapsedGameTime.TotalSeconds);
                Rectangle pRect = new Rectangle(
                        (int)(Game1.Player.Sprite.Position.X + motion.X),
                        (int)(Game1.Player.Sprite.Position.Y + motion.Y),
                        Engine.TileWidth,
                        Engine.TileHeight);

                if (pRect.Intersects(collision))
                {
                    Game1.Player.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;
                        Game1.Player.Sprite.IsAnimating = false;
                        inMotion = false;
                    }
                }

                Vector2 newPosition = Game1.Player.Sprite.Position + motion;
                newPosition.X = (int)newPosition.X;
                newPosition.Y = (int)newPosition.Y;
```

```
        Game1.Player.Sprite.Position = newPosition;
        motion = Game1.Player.Sprite.LockToMap(
            new Point(
                map.WidthInPixels,
                map.HeightInPixels),
            motion);

        if (motion == Vector2.Zero)
        {
            Vector2 origin = new Vector2(
                    Game1.Player.Sprite.Position.X + Game1.Player.Sprite.Origin.X,
                    Game1.Player.Sprite.Position.Y + Game1.Player.Sprite.Origin.Y);
            Game1.Player.Sprite.Position = Engine.VectorFromOrigin(origin);
            inMotion = false;
            Game1.Player.Sprite.IsAnimating = false;
        }
    }

    if ((Xin.CheckKeyReleased(Keys.Space) ||
        Xin.CheckKeyReleased(Keys.Enter)) && frameCount >= 5)
    {
        frameCount = 0;
        foreach (Point s in engine.Map.CharacterLayer.Characters.Keys)
        {
            Character c = engine.Map.CharacterLayer.Characters[s];

            AnimationKey animation = Game1.Player.Sprite.CurrentAnimation;

            if (animation == AnimationKey.WalkLeft &&
                ((int)c.Sprite.Position.X > (int)Game1.Player.Sprite.Position.X ||
                    (int)c.Sprite.Position.Y != (int)Game1.Player.Sprite.Position.Y))
            {
                continue;
            }

            if (animation == AnimationKey.WalkUp &&
                ((int)c.Sprite.Position.X != (int)Game1.Player.Sprite.Position.X ||
                    (int)c.Sprite.Position.Y > (int)Game1.Player.Sprite.Position.Y))
            {
                continue;
            }

            if (animation == AnimationKey.WalkRight &&
                ((int)c.Sprite.Position.X < (int)Game1.Player.Sprite.Position.X ||
                    (int)c.Sprite.Position.Y != (int)Game1.Player.Sprite.Position.Y))
            {
                continue;
            }

            if (animation == AnimationKey.WalkDown &&
                ((int)c.Sprite.Position.X != (int)Game1.Player.Sprite.Position.X ||
                    (int)c.Sprite.Position.Y < (int)Game1.Player.Sprite.Position.Y))
            {
                continue;
            }

            float distance = Vector2.Distance(
                Game1.Player.Sprite.Origin + Game1.Player.Sprite.Position,
                c.Sprite.Origin + c.Sprite.Position);
```

```
                    if (Math.Abs(distance) < Engine.TileWidth + Engine.TileWidth / 2)
                    {
                        manager.PushState(
                            (ConversationState)GameRef.ConversationState);

                        GameRef.ConversationState.SetConversation(c);
                        GameRef.ConversationState.StartConversation();
                        break;
                    }
                }
            }

            if (Xin.CheckKeyReleased(Keys.I))
            {
                manager.PushState(GameRef.ItemSelectionState);
            }

            if ((Xin.CheckKeyReleased(Keys.B)) && frameCount >= 5)
            {
                frameCount = 0;
                foreach (Point s in engine.Map.CharacterLayer.Characters.Keys)
                {
                    Character c = engine.Map.CharacterLayer.Characters[s];

                    AnimationKey animation = Game1.Player.Sprite.CurrentAnimation;

                    if (animation == AnimationKey.WalkLeft &&
                        ((int)c.Sprite.Position.X > (int)Game1.Player.Sprite.Position.X ||
                            (int)c.Sprite.Position.Y != (int)Game1.Player.Sprite.Position.Y))
                    {
                        continue;
                    }

                    if (animation == AnimationKey.WalkUp &&
                        ((int)c.Sprite.Position.X != (int)Game1.Player.Sprite.Position.X ||
                            (int)c.Sprite.Position.Y > (int)Game1.Player.Sprite.Position.Y))
                    {
                        continue;
                    }

                    if (animation == AnimationKey.WalkRight &&
                        ((int)c.Sprite.Position.X < (int)Game1.Player.Sprite.Position.X ||
                            (int)c.Sprite.Position.Y != (int)Game1.Player.Sprite.Position.Y))
                    {
                        continue;
                    }

                    if (animation == AnimationKey.WalkDown &&
                        ((int)c.Sprite.Position.X != (int)Game1.Player.Sprite.Position.X ||
                            (int)c.Sprite.Position.Y < (int)Game1.Player.Sprite.Position.Y))
                    {
                        continue;
                    }

                    float distance = Vector2.Distance(
                        Game1.Player.Sprite.Origin + Game1.Player.Sprite.Position,
                        c.Sprite.Origin + c.Sprite.Position);

                    if (Math.Abs(distance) < Engine.TileWidth + Engine.TileWidth / 2 &&
c.Alive())
```

```
                {
                    GameRef.StartBattleState.SetCombatants(Game1.Player, c);
                    manager.PushState(GameRef.StartBattleState);
                    break;
                }
            }
        }

        if (Xin.CheckKeyReleased(Keys.F1))
        {
            using (Aes aes = Aes.Create())
            {
                aes.IV = IV;
                aes.Key = Key;

                string path = 
Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData);
                path += "\\ShadowMonsters\\";

                try
                {
                    if (!Directory.Exists(path))
                    {
                        Directory.CreateDirectory(path);
                    }
                }
                catch
                {
                    // uh oh
                }

                try
                {
                    ICryptoTransform encryptor = aes.CreateEncryptor(Key, IV);
                    FileStream stream = new FileStream(
                        path + "ShadowMonsters.sav",
                        FileMode.Create,
                        FileAccess.Write);
                    using (CryptoStream cryptoStream = new CryptoStream(
                        stream,
                        encryptor,
                        CryptoStreamMode.Write))
                    {
                        BinaryWriter writer = new BinaryWriter(cryptoStream);
                        map.Save(writer);
                        Game1.Player.Save(writer);
                        writer.Close();
                    }
                    stream.Close();
                    stream.Dispose();
                }
                catch
                {
                    // uh oh
                }
            }
        }

        Engine.Camera.LockToSprite(map, Game1.Player.Sprite, new Rectangle(0, 0, 1280, 
720));
```

```
            Game1.Player.Update(gameTime);

            base.Update(gameTime);
        }
```

We are encrypting the file so there is a using statement for System.Security.Cryptography. AES requires an initialization vector and a key so there are byte arrays for both. For your game you will want to use different values.

In the Update method I add a check to see if the F1 key has been released. I create an Aes instance. I then assign the IV and Key properties to the IV and key that I created earlier. I then generate a path to the user's application data folder for saving the game. In a try catc.h block I check to see if the path exists or not. If it does not exist I create it. In the catch you should do something but I theory this should never happen.

In a try catch block I create an encryptor to encrypt the data. I then create a FileStream with Create and Write properties. I then create a CryptoStream using the stream and encryptor for writing. I then create the BinaryWriter for saving the data and call the Save methods of the map and player. I close the writer and dispose it. In the catch we need to handle the failure saving the game for now we do nothing.

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!