Eyes of the Dragon Tutorials
Part 22
Reading in Data

I'm writing these tutorials for the MonoGame 3.8 framework using Visual Studio 2019. The tutorials will make more sense if they are read in order. You can find the list of tutorials on the Eyes of the Dragon page of my web blog. I will be making each version of the project available on GitHub here. It will be included on the page that links to the tutorials.

In this tutorial I'm going to work on reading data into the game and expand a few things. We have all of those wonderful manager classes in the **RpgLibrary** but I decided against using them. I instead added a class to the **EyesOfTheDragon** project to manage all data in the game called **DataManager**. Right click the **EyesOfTheDragon** project, select **Add** and then **Class**. Name this new class **DataManager**. The code for that class follows next.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Content;
using RpgLibrary.CharacterClasses;
using RpgLibrary.ItemClasses;
using RpgLibrary.SkillClasses;
using RpgLibrary.SpellClasses;
using RpgLibrary.TalentClasses;
using RpgLibrary.TrapClasses;

namespace EyesOfTheDragon.Components
{
    static class DataManager
    {
        #region Field Region

        static Dictionary<string, ArmorData> armor = new Dictionary<string, ArmorData>();
        static Dictionary<string, WeaponData> weapons = new Dictionary<string, WeaponData>();
        static Dictionary<string, ShieldData> shields = new Dictionary<string, ShieldData>();
        static Dictionary<string, KeyData> keys = new Dictionary<string, KeyData>();
        static Dictionary<string, ChestData> chests = new Dictionary<string, ChestData>();
        static Dictionary<string, EntityData> entities = new Dictionary<string, EntityData>();
        static Dictionary<string, SkillData> skills = new Dictionary<string, SkillData>();

        #endregion

        #region Property Region

        public static Dictionary<string, ArmorData> ArmorData
        {
            get { return armor; }
        }

        public static Dictionary<string, WeaponData> WeaponData
        {
```

```
            get { return weapons; }
        }

        public static Dictionary<string, ShieldData> ShieldData
        {
            get { return shields; }
        }

        public static Dictionary<string, EntityData> EntityData
        {
            get { return entities; }
        }

        public static Dictionary<string, ChestData> ChestData
        {
            get { return chests; }
        }

        public static Dictionary<string, KeyData> KeyData
        {
            get { return keys; }
        }

        public static Dictionary<string, SkillData> SkillData
        {
            get { return skills; }
        }

        #endregion

        #region Constructor Region
        #endregion

        #region Method Region

        public static void ReadEntityData(ContentManager Content)
        {
            string[] filenames = Directory.GetFiles(@"Content\Game\Classes", "*.xnb");

            foreach (string name in filenames)
            {
                string filename = @"Game\Classes\" + Path.GetFileNameWithoutExtension(name);

                EntityData data = Content.Load<EntityData>(filename);
                EntityData.Add(data.EntityName, data);
            }
        }

        public static void ReadArmorData(ContentManager Content)
        {
            string[] filenames = Directory.GetFiles(@"Content\Game\Items\Armor", "*.xnb");

            foreach (string name in filenames)
            {
                string filename = @"Game\Items\Armor\" +
Path.GetFileNameWithoutExtension(name);

                ArmorData data = Content.Load<ArmorData>(filename);
                ArmorData.Add(data.Name, data);
            }
```

```
        }

        public static void ReadWeaponData(ContentManager Content)
        {
            string[] filenames = Directory.GetFiles(@"Content\Game\Items\Weapon", "*.xnb");

            foreach (string name in filenames)
            {
                string filename = @"Game\Items\Weapon\" +
Path.GetFileNameWithoutExtension(name);

                WeaponData data = Content.Load<WeaponData>(filename);
                WeaponData.Add(data.Name, data);
            }
        }

        public static void ReadShieldData(ContentManager Content)
        {
            string[] filenames = Directory.GetFiles(@"Content\Game\Items\Shield", "*.xnb");

            foreach (string name in filenames)
            {
                string filename = @"Game\Items\Shield\" +
Path.GetFileNameWithoutExtension(name);

                ShieldData data = Content.Load<ShieldData>(filename);
                ShieldData.Add(data.Name, data);
            }
        }

        public static void ReadKeyData(ContentManager Content)
        {
            string[] filenames = Directory.GetFiles(@"Content\Game\Keys", "*.xnb");

            foreach (string name in filenames)
            {
                string filename = @"Game\Keys\" + Path.GetFileNameWithoutExtension(name);

                KeyData data = Content.Load<KeyData>(filename);
                KeyData.Add(data.Name, data);
            }
        }

        public static void ReadChestData(ContentManager Content)
        {
            string[] filenames = Directory.GetFiles(@"Content\Game\Chests", "*.xnb");

            foreach (string name in filenames)
            {
                string filename = @"Game\Chests\" + Path.GetFileNameWithoutExtension(name);

                ChestData data = Content.Load<ChestData>(filename);
                ChestData.Add(data.Name, data);
            }
        }

        public static void ReadSkillData(ContentManager Content)
        {
            string[] filenames = Directory.GetFiles(@"Content\Skills", "*.xnb");
```

```
            foreach (string name in filenames)
            {
                string filename = @"Game\Skills\" + Path.GetFileNameWithoutExtension(name);
                SkillData data = Content.Load<SkillData>(filename);
                SkillData.Add(data.Name, data);
            }
        }

        #endregion

        #region Virtual Method region
        #endregion
    }
}
```

There is a lot of code there but most of it is the same with just a few minor differences. First off, this is a static class. When you make a class static it is created the first time it is referenced, you don't need to call its constructor to create it. You can provide a static constructor for a static class but I instead created the instances of the fields explicitly.

I added in a using statement for the **System.IO** name space to bring a few classes into scope for manipulating files and paths. I also added a using statement for a couple XNA framework name spaces and for many of the **RpgLibrary** name spaces as well.

There are a number of **Dictionary<string, T>** fields and properties where **T** is the type of data associated with the field or property. **ArmorData** works with armor for example. The fields are private and the properties are public. The properties are also read only, or get only. There are also a number of methods **ReadT(ContentManager Content)** where **T** is again a type of data. I need the **ContentManager** associated with the game to read in the data as it has been compiled by the **Content Pipeline**.

Each of the **ReadT** methods works the same way. I first get all of the files in a specific directory with an **xnb** extension, the extension given to content when it is processed by the **Content Pipeline**, that is read into a local variable **filenames**. In a foreach loop I loop through of the file names. I then create a string that is the same as the asset name given to the file when it is processed by the **Content Pipeline**.

I then use the **Load** method of the **ContentManager** to load the asset into the variable **data**. The **data** variable is then added to the appropriate **Dictionary<string, T>** where the string is the name of the data and **T** is the actual data.

Reading in the data in the game is relatively painless. First, make sure you have a using statement for the **Components** name space of **EyesOfTheDragon**. Then in the **LoadContent** method of the **Game1** class call each of the **Read** methods. Add the following using statement to the **Game1** class and change the **LoadContent** method to the following.

```
using EyesOfTheDragon.Components;

        protected override void LoadContent()
        {
```

```
            _spriteBatch = new SpriteBatch(GraphicsDevice);

            DataManager.ReadEntityData(Content);
            DataManager.ReadArmorData(Content);
            DataManager.ReadShieldData(Content);
            DataManager.ReadWeaponData(Content);
            DataManager.ReadChestData(Content);
            DataManager.ReadKeyData(Content);
            DataManager.ReadSkillData(Content);
        }
```

Now we have access to all of the data we created in the editor. There is one small drawback to the way things work right now. Every time you add new items using the editor you need to add it to the **MonoGame Pipeline Tool**. Unfortunately there is no easy way to do that. You will need to run the tool and add the new content manually.

Now that we have all of our data it is time to update a few things. The first thing I want to update is the **Player** class. Instead of having an **AnimatedSprite** field, I want to have a **Character** field. This will break a few things but now is the best time to tackle this. Change the code of the **Player** class of the **EyesOfTheDragon** project to the following.

```
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 MGRpgLibrary;
using MGRpgLibrary.TileEngine;
using MGRpgLibrary.SpriteClasses;
using MGRpgLibrary.CharacterClasses;

namespace EyesOfTheDragon.Components
{
    public class Player
    {
        #region Field Region

        Camera camera;
        Game1 gameRef;
        readonly Character character;

        #endregion

        #region Property Region

        public Camera Camera
        {
            get { return camera; }
            set { camera = value; }
        }

        public AnimatedSprite Sprite
        {
            get { return character.Sprite; }
        }
```

```
public Character Character
{
    get { return character; }
}

#endregion

#region Constructor Region

public Player(Game game, Character character)
{
    gameRef = (Game1)game;
    camera = new Camera(gameRef.ScreenRectangle);
    this.character = character;
}

#endregion

#region Method Region

public void Update(GameTime gameTime)
{
    camera.Update(gameTime);
    Sprite.Update(gameTime);

    if (InputHandler.KeyReleased(Keys.PageUp) ||
        InputHandler.ButtonReleased(Buttons.LeftShoulder, PlayerIndex.One))
    {
        camera.ZoomIn();
        if (camera.CameraMode == CameraMode.Follow)
            camera.LockToSprite(Sprite);
    }
    else if (InputHandler.KeyReleased(Keys.PageDown) ||
        InputHandler.ButtonReleased(Buttons.RightShoulder, PlayerIndex.One))
    {
        camera.ZoomOut();
        if (camera.CameraMode == CameraMode.Follow)
            camera.LockToSprite(Sprite);
    }

    Vector2 motion = new Vector2();

    if (InputHandler.KeyDown(Keys.W) ||
        InputHandler.ButtonDown(Buttons.LeftThumbstickUp, PlayerIndex.One))
    {
        Sprite.CurrentAnimation = AnimationKey.Up;
        motion.Y = -1;
    }
    else if (InputHandler.KeyDown(Keys.S) ||
        InputHandler.ButtonDown(Buttons.LeftThumbstickDown, PlayerIndex.One))
    {
        Sprite.CurrentAnimation = AnimationKey.Down;
        motion.Y = 1;
    }

    if (InputHandler.KeyDown(Keys.A) ||
        InputHandler.ButtonDown(Buttons.LeftThumbstickLeft, PlayerIndex.One))
    {
        Sprite.CurrentAnimation = AnimationKey.Left;
```

```
                motion.X = -1;
            }
            else if (InputHandler.KeyDown(Keys.D) ||
                InputHandler.ButtonDown(Buttons.LeftThumbstickRight, PlayerIndex.One))
            {
                Sprite.CurrentAnimation = AnimationKey.Right;
                motion.X = 1;
            }

            if (motion != Vector2.Zero)
            {
                motion.Normalize();

                Sprite.IsAnimating = true;
                Sprite.Position += motion * Sprite.Speed;
                Sprite.LockToMap();

                if (camera.CameraMode == CameraMode.Follow)
                    camera.LockToSprite(Sprite);
            }
            else
            {
                Sprite.IsAnimating = false;
            }

            if (InputHandler.KeyReleased(Keys.F) ||
                InputHandler.ButtonReleased(Buttons.RightStick, PlayerIndex.One))
            {
                camera.ToggleCameraMode();

                if (camera.CameraMode == CameraMode.Follow)
                    camera.LockToSprite(Sprite);
            }

            if (camera.CameraMode != CameraMode.Follow)
            {
                if (InputHandler.KeyReleased(Keys.C) ||
                    InputHandler.ButtonReleased(Buttons.LeftStick, PlayerIndex.One))
                {
                    camera.LockToSprite(Sprite);
                }
            }
        }

        public void Draw(GameTime gameTime, SpriteBatch spriteBatch)

        {
            character.Draw(gameTime, spriteBatch);
        }
        #endregion
    }
}
```

What has changed here is first I added in a using statement for the **CharacterClasses** name space from the **MGRpgLibrary**. I then replaced the **AnimatedSprite** field with a **Character** field **character**. In the **Sprite** property instead of returning the **AnimatedSprite** field I return the **Sprite** property of the **character** field. I change the constructor to take a **Character** argument rather than an **AnimatedSprite** argument. I set the **character** field to the **character** parameter. In the **Update** method I replace the

sprite field with the **Sprite** property. In the **Draw** method I call the **Draw** method of the **character** field.

That is going to break the **LoadGameScreen** and the **CharacterGeneratorScreen** because when they created the **Player** component they were passing in an **AnimatedSprite** and now it is expecting a **Character**. I will fix the **LoadGameScreen** first. What you will want to do is to create an **Entity** in the **CreatePlayer** method and then a **Character** object. When you create the **Player** object pass in the **Character** object. Change the **CreatePlayer** method of the **LoadGameScreen** class to the following.

```
        private void CreatePlayer()
        {
            Dictionary<AnimationKey, Animation> animations = new Dictionary<AnimationKey,
Animation>();

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

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

            animation = new Animation(3, 32, 32, 0, 64);
            animations.Add(AnimationKey.Right, animation);

            animation = new Animation(3, 32, 32, 0, 96);
            animations.Add(AnimationKey.Up, animation);

            AnimatedSprite sprite = new AnimatedSprite(
                GameRef.Content.Load<Texture2D>(@"PlayerSprites\malefighter"),
                animations);

            Entity entity = new Entity(
                "Encelwyn",
                DataManager.EntityData["Fighter"],
                EntityGender.Male,
                EntityType.Character);
            Character character = new Character(entity, sprite);

            GamePlayScreen.Player = new Player(GameRef, character);
        }
```

When I created the **Entity** I used some magic values. For the name I used, **Encelwyn**. I also specified that the character will be a fighter and a male. You do want the **Entity** to be a **Character** though so we are fine there. Eventually we will be writing out the player and reading the player back in. It does make handling creating the entity in the **CharacterGeneratorScreen** is a little more problematic. You would like selections on the **CharacterGeneratorScreen** to reflect the data that you have created. The name of the character will be the hardest to deal with as you may want the player to be able to choose a name for their character.For the name we will just use a fixed value for now. Filling the class selector with the available classes won't be that hard. Loading the sprites also won't be that difficult because of the way I named the sprites.

Let's get to the changes. The first change is you want to remove the initialization of the **classItems** field in the **CharacterGeneratorScreen**. Change that field to the following and add a using statement for the **CharacterClasses** name space of the **RpgLibrary** and **MGRpgLibrary**.

```
using RpgLibrary.CharacterClasses;
using MGRpgLibrary.CharacterClasses;

string[] classItems;
```

Now, in the **LoadContent** method you will want create the array of strings for **classItems** and fill it. You will do that in the **LoadContent** method. Change the **LoadContent** method to the following.

```
protected override void LoadContent()
{
    base.LoadContent();

    classItems = new string[DataManager.EntityData.Count];

    int counter = 0;

    foreach (string className in DataManager.EntityData.Keys)
    {
        classItems[counter] = className;
        counter++;
    }

    LoadImages();
    CreateControls();

    containers = Game.Content.Load<Texture2D>(@"ObjectSprites\containers");
}
```

What the new code does is create a new array that is the size of the **Count** property of **EntityData** in the **DataManager**. You create the **CharacterGeneratorScreen** in the constructor of **Game1** before the **LoadContent** method is called. How can you use those values here? The reason is that because you created the instance of **CharacterGeneratorScreen** in the constructor of **Game1** its **LoadContent** method isn't called until its **Initialize** method is called. That will be when it is added to the list of components in the game by the state manager. If you set a break point at the call to **base.LoadContent** in Visual Studio you will see that the method is not called until after the player selects the menu item for creating a new game.

What the new code is doing is creating a new string array the size of the **Count** property of the **EntityData** property of the **DataManager** class. There is then a variable I called **counter** set to zero initially that will hold what index I am on. In a foreach loop I loop through all of the keys in the **Keys** collection of **EntityData**. I assign the **classItems** array value at index **counter** to the current key and then I increment the **counter** variable.

The last thing to do is to update the **CreatePlayer** method of **CharacterGeneratorScreen**. Like in the **LoadGameScreen** class you want to create a **Character** object and pass it to the call to the constructor of the **Player** class. The difference is that you will use the values from the screen. Change the **CreatePlayer** method of the **CharacterGeneratorScreen** to the following.

```
private void CreatePlayer()
{
    Dictionary<AnimationKey, Animation> animations =
        new Dictionary<AnimationKey, Animation>();
```

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

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

    animation = new Animation(3, 32, 32, 0, 64);
    animations.Add(AnimationKey.Right, animation);

    animation = new Animation(3, 32, 32, 0, 96);
    animations.Add(AnimationKey.Up, animation);

    int gender = genderSelector.SelectedIndex < 2 ? genderSelector.SelectedIndex : 1;

    AnimatedSprite sprite = new AnimatedSprite(
        characterImages[gender, classSelector.SelectedIndex],
        animations);
    EntityGender g = EntityGender.Unknown;

    switch (genderSelector.SelectedIndex)
    {
        case 0:
            g = EntityGender.Male;
            break;
        case 1:
            g = EntityGender.Female;
            break;
        case 2:
            g = EntityGender.NonBinary;
            break;
    }
    Entity entity = new Entity(
        nameSelector.SelectedItem,
        DataManager.EntityData[classSelector.SelectedItem],
        g,
        EntityType.Character);
    Character character = new Character(entity, sprite);
    GamePlayScreen.Player = new Player(GameRef, character);
}
```

What the new code does is first create a local variable of type **EntityGender** and assign it to be **Unkown** by default. Then in a switch statement I check the **SelectedIndex** of the **genderSelector**. If it is 0, I set the local variable **g** to **Male**. If it is 1, I set the local variable to **Female**. Finally, if it is three is set **g** to **NonBinary**. I then create a new **Entity** passing in, **SelectedItem** of the **nameSelector**, for the name, the **EntityData** selected in the **classSelectror**, the **g** variable, and **Character** for the **EntityType**.

I then create a new **Character** and using it in the call to the constructor of the **Player** class. I know selecting a name from a list isn't very RPGish but it will work for now. A better solution will be after the player has confirmed their choices of gender and class to move to a screen they can select a name from.

I think that this is enough for this tutorial. It covers the original tutorial with a few modifications. So, I encourage you to visit my blog at, https://cynthiamcmahon.ca/blog/, for the latest news on my tutorials and other goodness.