Eyes of the Dragon Tutorials
Part 21
Items in the Game

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.

We have been working mostly with data for items and the editors. In this tutorial I'm going to work on how items will be handled in the game. So a brief discussion on how I intend to implement items is needed.

In the **RpgLibrary** there are classes that hold the basic data of items and classes that represent the actual items. I'm going to add a class to the **MGRpgLibrary** that represent items in the game. This class will have a **Texture2D** associated with it so you can have an icon associated with an item. A character can have these items equipped in different slots. They can have armor in the different armor locations for example. You are going to want to manage all of the items in the game as well as the items the party is carrying. This will be a party based engine but you can easily use it with just a single character if you so desire. There will be a party backpack that will hold items the party is carrying.

The first step is to create a class that will represent an item in the game. Right click the **ItemClasses** folder in the **MGRpgLibrary**, select **Add** and then **Class**. Name this new class **GameItem**. The code for this class follows next.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using RpgLibrary.ItemClasses;

namespace MGRpgLibrary.ItemClasses
{
    public class GameItem
    {
        #region Field Region

        public Vector2 Position;
        private Texture2D image;
        private Rectangle? sourceRectangle;
        private readonly BaseItem baseItem;
        private Type type;

        #endregion

        #region Property Region

        public Texture2D Image
        {
```

```
            get { return image; }
        }

        public Rectangle? SourceRectangle
        {
            get { return sourceRectangle; }
            set { sourceRectangle = value; }
        }

        public BaseItem Item
        {
            get { return baseItem; }
        }

        public Type Type
        {
            get { return type; }
        }

        #endregion

        #region Constructor Region

        public GameItem(BaseItem item, Texture2D texture, Rectangle? source)
        {
            baseItem = item;
            image = texture;
            sourceRectangle = source;
            type = item.GetType();
        }

        #endregion

        #region Method Region

        public void Draw(SpriteBatch spriteBatch)
        {
            spriteBatch.Draw(image, Position, sourceRectangle, Color.White);
        }

        #endregion

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

I will be adding in more to this class as the game evolves. It is just to get things started. The reason that this class is in the **MGRpgLibrary** rather than the **RpgLibrary** is that you may want to draw the item. In that case you will want access to the MonoGame framework classes. We could move the **RpgLibrary** into the **MGRpgLibrary**, that would certainly be a viable option. I'm trying to keep the data separate and at this time it would be a major headache to try and fix everything. The way things are organized will work well enough.

In the class itself I added a couple using statements. There are two to bring some MonoGame framework classes into scope and one for the **ItemClasses** from the **RpgLibrary**.

There are a few fields in the class. The first is public and is a **Vector2** called **Position**. We really don't care where the item will be drawn so I decided to simplify things and just make it public. Another reason is how C# deals with properties that are structures, like **Vector2**. If you have a **Vector2** property you can't assign to the individual properties of the **Vector2**, the **X** and **Y** values for example. It has to do with how C# handles value types and reference types. Since **Vector2** is a struct rather than a class it is a value type. The other four fields are all private. The **image** field is a **Texture2D**. The next field is a nullable field, indicated by the ? after the type, and is the source rectangle of the **image** field. In the overload of the **Draw** method of **SpriteBatch** that I use it will accept null for the source rectangle. There is then a **BaseItem** field for the actual item. This field is marked **readonly** so it can only be assigned to as a class initializer or in the constructor. There is also a **Type** field for the type of the item. This isn't to be confused with the string **type** field from **BaseItem**. This is the actual type associated with the **BaseItem** like **Weapon**, **Armor**, or **Shield**.

There are properties to expose some of the fields. The **Image** property is read only, or just has a get part, and returns the **image** property. I included it as it may be useful. The property is a read and write, get and set, property and is also nullable. It either returns or sets the **sourceRectangle** field. The **Item** property is also read only. The reason is the **baseItem** field is marked **readonly** and can't be set using the property anyway. The **Type** property is read only and returns the **type** field.

There is just one constructor and it takes three parameters. The first is a **BaseItem** which is the actual item, a **Texture2D** for the texture of the image, and a nullable **Rectangle** for the source rectangle. The constructor sets the fields based on the values passed in. To set the **type** field I use the **GetType** method that returns the type associated with a variable. It won't return **BaseItem** if you pass a **Weapon** in. It will return **Weapon**. Even if the variable is a **BaseItem** variable that you assign a **Weapon** to. C# is smart enough to know the inherited type when you're using polymorphism.

At the moment there is just the one method in this class, the **Draw** method that takes as a parameter an active **SpriteBatch**. What I mean by active is in between calls to **Begin** and **End**. It calls the **Draw** method of the **SpriteBatch** with the **Texture2D**, **Vector2**, **Rectangle?**, and **Color**.

I'm going to add a global item manager first that holds all items in the game. Right click the **ItemClasses** folder in the **MGRpgLibrary** project, select **Add** and then class. Name this new class **GameItemManager**. The code for that class follows next.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using RpgLibrary.ItemClasses;

namespace MGRpgLibrary.ItemClasses
{
    public class GameItemManager
    {
        #region Field Region

        readonly Dictionary<string, GameItem> gameItems = new Dictionary<string, GameItem>();
        static SpriteFont spriteFont;
```

```
        #endregion

        #region Property Region

        public Dictionary<string, GameItem> GameItems
        {
            get { return gameItems; }
        }

        public static SpriteFont SpriteFont
        {
            get { return spriteFont; }
            private set { spriteFont = value; }
        }

        #endregion

        #region Constructor Region

        public GameItemManager(SpriteFont spriteFont)
        {
            SpriteFont = spriteFont;
        }

        #endregion

        #region Method Region
        #endregion

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

It is a fairly basic class. I added in using statement for a couple MonoGame framework name spaces and for the **ItemClasses** name space of the **RpgLibrary**. There are two fields in the class. The first is a **Dictionary<string, GameItem>** and is readonly. The second is a **SpriteFont** and it is static. There are properties to expose both fields. For the **SpriteFont** property I include a private set, or write. The constructor takes a **SpriteFont** parameter and sets the **spriteFont** field using the private write property.

This class is not for inventory. It is to hold all of the basic items in the game. Items in inventory will be handled a little differently. What this class allows is you can read in all items at run time and if you need to add an item to the player's inventory you can retrieve it from this class. I've been considering allowing the player to be able to upgrade items. Some items can contain sockets that can be filled to create interesting effects. The **SpriteFont** field will be useful if you want to draw text related to an item.

You are going to need images for item icons. I went to one of my favourite spots when looking for place holder graphics, http://opengameart.org. They have an excellent collection of free art for open source games. I found an image by an artist there, Jetrel, that I modified to take the background color out of the image. You can find my image at this link. Download the file and extract it to a directory.

Open the **MonoGame Pipeline Tool** by double clicking the **Content.mgcb** node in the **Content** folder of the **EyesOfTheDragon** project. Right click the **ObjectSprites** folder in the **MonoGame Pipeline Tool**. folder, select **Add** and then **Existing Item**. Navigate to where you extracted the file and add in the **itemimages.png** file.

I'm going to expand the **Character** class so that a character can have some items equipped as well being able to equip and unequip items. Change the **Character** class in the **MGRpgLibrary** project to the following.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using RpgLibrary.CharacterClasses;
using MGRpgLibrary.SpriteClasses;
using MGRpgLibrary.ItemClasses;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

namespace MGRpgLibrary.CharacterClasses
{
    public class Character
    {
        #region Field Region

        protected Entity entity;
        protected AnimatedSprite sprite;

        // Armor fields
        protected GameItem head;
        protected GameItem body;
        protected GameItem hands;
        protected GameItem feet;

        // Weapon/Shield fields
        protected GameItem mainHand;
        protected GameItem offHand;
        protected int handsFree;

        #endregion

        #region Property Region

        public Entity Entity
        {
            get { return entity; }
        }

        public AnimatedSprite Sprite
        {
            get { return sprite; }
        }

        // Armor properties
        public GameItem Head
        {
            get { return head; }
        }
```

```
public GameItem Body
{
    get { return body; }
}

public GameItem Hands
{
    get { return hands; }
}

public GameItem Feet
{
    get { return feet; }
}

// Weapon/Shield properties

public GameItem MainHand
{
    get { return mainHand; }
}

public GameItem OffHand
{
    get { return offHand; }
}

public int HandsFree
{
    get { return handsFree; }
}

#endregion

#region Constructor Region

public Character(Entity entity, AnimatedSprite sprite)
{
    this.entity = entity;
    this.sprite = sprite;
}

#endregion

#region Method Region
#endregion

#region Virtual Method region

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

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

```
        public virtual bool Equip(GameItem gameItem)
        {
            bool success = false;
            return success;
        }

        public virtual bool Unequip(GameItem gameItem)
        {
            bool success = false;
            return success;
        }

        #endregion
    }
}
```

So, what is new here. I added in a using statement to bring the **ItemClasses** of the **MGRpgLibrary** into scope. I included several new **GameItem** fields and properties to expose the fields. I place comments above the two sets to divide them logically. The first set of fields has to do with armor. There a four regions that armor will fit: head, body, hands, and feet. So, I have four fields for those regions named **head**, **body**, **hands**, and **feet**. Since characters have two hands I have two fields for hands, **mainHand** and **offHand**. I could have used right and left hand here but decided to go with **main** and **off**. There is also another field, **handsFree**, that is an integer that represents the number of free hands. There are read only, get only, properties to expose all of the new fields. I also included two new virtual methods **Equip** and **Unequip** that return a bool and take a **GameItem** parameter. These method can be called to equip and unequip items. They will be expanded on later. Since they are virtual can override them in any class that inherits from **Character** if needed. I have them returning a bool value that tells if equipping or unequipping the item was successful. By default it will be unsuccessful as you can see I set the **success** local variable to **false** and return it at the end of the method.

The last class I want to add is a class for the items the party is carrying. You may want to limit the number of items in inventory. I'm not going to but it could easily be added. For this I'm going to create a class called **Backpack**. The **Backpack** will be added to the class that will represent the party of characters in the game. This is going to be a basic class for the moment and is more of a place holder that will be expanded upon. The **Backpack** will allow for unique items, like potions that are made by a character with a high herbalism skill will be more potent than a less skilled character. It also allows for items with sockets. Right click the **ItemClasses** folder in the **MGRpgLibrary** folder, select **Add** and then **Class**. Name this new class **Backpack**. The code for that class follows next.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace MGRpgLibrary.ItemClasses
{
    public class Backpack
    {
        #region Field Region

        readonly List<GameItem> items;
```

```
        #endregion

        #region Property Region

        public List<GameItem> Items
        {
            get { return items; }
        }

        public int Capacity
        {
            get { return items.Count; }
        }

        #endregion

        #region Constructor Region

        public Backpack()
        {
            items = new List<GameItem>();
        }

        #endregion

        #region Method Region

        public void AddItem(GameItem gameItem)
        {
            items.Add(gameItem);
        }

        public void RemoveItem(GameItem gameItem)
        {
            items.Remove(gameItem);
        }

        #endregion

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

This is just a basic class to get us up and going. There is just the one field in the class that is a **List<GameItem>** that represents the items in the backpack. I used a generic **List** rather than a **Dictionary** as you can have multiple items by the same name. These multiple items can also have different properties. I will explain more in a bit. The **items** field is readonly so that it can only be assigned to as an initializer or in the constructor. There are two properties in the class. The first is used to return the **items** field. The second may be a bit of a misnomer. I added a property, **Capacity**, that returns the number of items in the **items** field. You would expect capacity to be the amount an object can hold, not the amount in the object. It will work well enough though. The constructor just creates a new **List<GameItem>**. I added in two methods: **GetItem** and **RemoveItem**. Their purpose is to get and remove items from the backpack. Here you don't want to work on copies of the items. You want to work on the items themselves. For example, if you fill a socket on a weapon you want to affect the actual weapon in the backpack and not a copy of the weapon. I will get into that more in a future

tutorial.

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.

Good luck in your game programming adventures!
Cynthia