Eyes of the Dragon Tutorials
Part 16
Quests and Conversations

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.

This tutorial is going to cover adding in loot for the player to find and pick up be it in chests, boxes, barrels, whatever. After I handle the combat engine I will add in mobs, any enemy the player kills, dropping items.

The first thing we are going to need is a basic sprite class that we can use to draw sprites with. So, I added a class called **BaseSprite** to the **SpriteClasses** folder in the **MGRprLibrary** project. Right click the **SpriteClasses** folder, select **Add** and then **Class**. Call the class **BaseSprite**. This is the code.

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

namespace MGRpgLibrary.SpriteClasses
{
    public class BaseSprite
    {
        #region Field Region

        protected Texture2D texture;
        protected Rectangle sourceRectangle;
        protected Vector2 position;
        protected Vector2 velocity;
        protected float speed = 2.0f;

        #endregion
        #region Property Region

        public int Width
        {
            get { return sourceRectangle.Width; }
        }
        public int Height
        {
            get { return sourceRectangle.Height; }
        }
        public Rectangle Rectangle
        {
            get
            {
                return new Rectangle(
                    (int)position.X,
```

```
            (int)position.Y,
            Width,
            Height);
    }
}
public float Speed
{
    get { return speed; }
    set { speed = MathHelper.Clamp(speed, 1.0f, 16.0f); }
}

public Vector2 Position
{
    get { return position; }

    set
    {
        position = value;
    }
}
public Vector2 Velocity
{
    get { return velocity; }

    set
    {
        velocity = value;

        if (velocity != Vector2.Zero)
            velocity.Normalize();
    }
}

#endregion

#region Constructor Region

public BaseSprite(Texture2D image, Rectangle? sourceRectangle)
{
    this.texture = image;

    if (sourceRectangle.HasValue)
        this.sourceRectangle = sourceRectangle.Value;
    else
        this.sourceRectangle = new Rectangle(
            0,
            0,
            image.Width,
            image.Height);

    this.position = Vector2.Zero;
    this.velocity = Vector2.Zero;
}

public BaseSprite(Texture2D image, Rectangle? sourceRectangle, Point tile)
    : this(image, sourceRectangle)
{
    this.position = new Vector2(
        tile.X * Engine.TileWidth,
        tile.Y * Engine.TileHeight);
```

```
        }

        #endregion

        #region Method Region
        #endregion

        #region Virtual Method region

        public virtual void Update(GameTime gameTime)
        {
        }

        public virtual void Draw(GameTime gameTime, SpriteBatch spriteBatch)
        {
            spriteBatch.Draw(
                texture,
                Position,
                sourceRectangle,
                Color.White);
        }
        #endregion
    }
}
```

Compared to some of the classes in the game it is pretty straight forward. The fields came from the **AnimatedSprite** class. They are included as down the road they could be useful. I added a few using statements to bring some of the MonoGame framework classes into scope, as well as the **TileEngine** classes.

When you create a sprite you can specify what tile it is in, rather than saying it is at pixel (x,y) in the map. To do that I need the **Engine** class.

There are a number of protected fields in the class. They are protected because down the road you might want a special kind of sprite. The fields are a **Texture2D** for the sprite. A **Rectangle** that is a source rectangle, if you want to use sprite sheets. There are **Vector2** fields for the position of the sprite and the velocity of the sprite. For items sprites will remain in the same place. But, you could have a sprite that bobs up and down and including a velocity was a good idea. There is also a **float** field for the speed of the sprite.

There are a few properties that expose some information about the sprite. The **Width** and **Height** properties return the **Width** and **Height** properties of the **sourceRectangle** field. The **Speed**, **Position**, and **Velocity** properties were taken from the **AnimatedSprite** class.

I added in two constructors for this class. The first takes a **Texture2D** for the sprite and a **Rectangle** for the source rectangle of the sprite. The **Rectangle** parameter is a nullable parameter, indicated by the ? after **Rectangle**. So, if you don't want to specify a source rectangle you can pass in null for that parameter and the class will use the entire image. The second constructor takes a third parameter, a **Point** that represents the tile the sprite is in.

The first constructor sets the **texture** field with the **image** parameter. It checks to see if the **Rectangle** parameter has a value using the **HasValue** property. If it does I set the **sourceRectangle** field to be the

sourceRectangle parameter. If it doesn't have a value I create a new **Rectangle** using the width and height of the image passed in. I then set the **position** and **velocity** fields to be the **Zero** vector.

The second constructor calls the first constructor using **this** to reference that constructor with the **image** and **sourceRectangle** parameters passed in. To set the position of the sprite, in tiles, I multiply the **X** property of the **Point** by the **TileWidth** and the the **Y** property of the **Point** by the **TileHeight** from the **Engine** class. What is important here is that you create sprites after creating an instance of the **Engine** class. Otherwise the **TileWidth** and **TileHeight** will both be 0 and the sprites will all line up in the top left corner of the map. You are not going to have to worry about it as when the **GamePlayScreen** is created an **Engine** class is constructed. Just be aware that limitation is there.

I'm going to add a class that holds a **BaseItem** from the **RpgLibrary** and has a **BaseSprite** as well. This will allow you to have icons for your items. This will be helpful when I get to inventory and you can have icons on the screen in your game, for a chest for example. This is a hybrid class and is a cross between an item and a sprite. I decided the best course of action with to create an **ItemClasses** folder in the **MGRpgLibrary** to represent items like this. Right click the **MGRpgLibrary**, select **Add** and then **New Folder**. Name this new folder **ItemClasses**. Right click the **ItemClasses** folder, select **Add** and then **Class**. Name this new class **ItemSprite**. This is the code for that class.

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

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

        BaseSprite sprite;
        BaseItem item;

        #endregion

        #region Property Region

        public BaseSprite Sprite
        {
            get { return sprite; }
        }

        public BaseItem Item
        {
            get { return item; }
        }

        #endregion

        #region Constructor Region
```

```
        public ItemSprite(BaseItem item, BaseSprite sprite)
        {
            this.item = item;
            this.sprite = sprite;
        }

        #endregion

        #region Method Region
        #endregion

        #region Virtual Method region

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

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

        #endregion
    }
}
```

This is another straight forward class. There are using statements to bring a few name space into scope.

There are two fields and two properties to expose the fields. The first field is a **BaseSprite** and the second is a **BaseItem**. The property **Sprite** exposes the **sprite** field and the property **Item** exposes the **item** field. The constructor takes two parameters. A **BaseItem** for the item and a **BaseSprite** for the sprite. It just sets the fields to the values passed in. I added **Update** and **Draw** methods that are virtual so you can create classes that inherit from this one and override those properties if needed. They have the same parameters as the **BaseSprite** class. They just call the **Update** and **Draw** methods of the **sprite** field.

Before I get to chests I thought I'd add in a couple place holder classes that will be needed for chests. What fun is having a chests if you can't trap them! You need to make your thief characters feel useful after all. To handle that I added in a few place holders for traps. I added them to the **RpgLibrary** as they deal more with mechanics than with MonoGame. Right click the **RpgLibrary**, select **Add** and then **New Folder**. Name this new folder **TrapClasses**. To this folder you are going to want to add three empty classes, I included the regions in mine. Right click the **TrapClasses** folder three times, selecting **Add** and then **Class** each time. Name the classes **Trap**, **TrapData**, and **TrapDataManager**. The code for my classes follows next in that order.

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

namespace RpgLibrary.TrapClasses
```

```
{
    public class Trap
    {
        #region Field Region
        #endregion

        #region Property Region
        #endregion

        #region Constructor Region
        #endregion

        #region Method Region
        #endregion

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

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

namespace RpgLibrary.TrapClasses
{
    public class TrapData
    {
        #region Field Region
        #endregion

        #region Property Region
        #endregion

        #region Constructor Region
        #endregion

        #region Method Region
        #endregion

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

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

namespace RpgLibrary.TrapClasses
{
    public class TrapDataManager
    {
        #region Field Region
        #endregion

        #region Property Region
        #endregion
```

```
        #region Constructor Region
        #endregion

        #region Method Region
        #endregion

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

Containers that the player can interact with that hold items, and gold, are all going to be lumped into one class, a **Chest** class. The containers can be anything you can imagine though and you're only limited by the graphics you have. To get started, right click the **ItemClasses** in the **RpgLibrary**, select **Add** and then **Class**. Name this class **ChestData**. This is the code for that class.

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

namespace RpgLibrary.ItemClasses
{
    public class ChestData
    {
        public string Name;
        public string TextureName;
        public bool IsTrapped;
        public bool IsLocked;
        public string TrapName;
        public string KeyName;
        public int MinGold;
        public int MaxGold;

        public Dictionary<string, string> ItemCollection;

        public ChestData()
        {
            ItemCollection = new Dictionary<string, string>();
        }

        public override string ToString()
        {
            string toString = Name + ", ";

            toString += TextureName + ", ";
            toString += IsTrapped.ToString() + ", ";
            toString += IsLocked.ToString() + ", ";
            toString += TrapName + ", ";
            toString += KeyName + ", ";
            toString += MinGold.ToString() + ", ";
            toString += MaxGold.ToString();

            foreach (KeyValuePair<string, string> pair in ItemCollection)
            {
                toString += ", " + pair.Key + "+" + pair.Value;
            }
```

```
            return toString;
        }
    }
}
```

This is a data class that will be used to generate chests on the fly and in the editor. There are a number of fields. The **Name** field is the name of the chest. The **TextureName** field is the name of the texture for the chest. The Boolean fields, **IsTrapped** and **IsLocked**, tell if a chest is trapped or if it locked. The **TrapName** and **KeyName** are used if the chest has a trap and requires a special key to open it. If they are set to **none** then there is no trap and a special key isn't needed to open the chest. The **MinGold** and **MaxGold** fields represent the minimum and maximum gold a chest can hold. If both are set to zero then the chest holds no gold. The **ItemCollection** field is a **Dictionary<string, string>** that holds the items in the chest. The first string is the name of the item and the second string is type of the item.

There is a limitation here though. A chest can't hold more than one of any particular item. It is good enough for right now though.

Like the other data classes for items there is an override of the **ToString** method. I append the fields, plus a comma and a space, using the **ToString** method if needed to convert a field to a string. The **ItemCollection** is the interesting item here though. In a foreach loop I loop through all of the **KeyValuePairs** in the dictionary. I append a comma and a space and for the pair I append the **Key** a + and the **Value**. This does mean that your item names can't use a + sign. If that is an issue, say you want a longsword +5, you can change to + to another symbol that you don't use. It then returns the string that was created.

I'm also going to add a basic class for chests. It will need to be fleshed out more but it will be enough to get chests into the game. Right click the **ItemClasses** folder in the **RpgLibrary**, select **Add** and then **Class**. Name this new class **Chest**. This is the code so far.

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

namespace RpgLibrary.ItemClasses
{
    public class Chest : BaseItem
    {
        #region Field Region

        static Random Random = new Random();
        ChestData chestData;

        #endregion
        #region Property Region

        public bool IsLocked
        {
            get { return chestData.IsLocked; }
```

```
}

public bool IsTrapped
{
    get { return chestData.IsTrapped; }
}

public int Gold
{
    get
    {
        if (chestData.MinGold == 0 && chestData.MaxGold == 0)
            return 0;

        int gold = Random.Next(chestData.MinGold, chestData.MaxGold);

        chestData.MinGold = 0;
        chestData.MaxGold = 0;

        return gold;
    }
}

#endregion

#region Constructor Region
public Chest(ChestData data)
    : base(data.Name, "", 0, 0)
{
    this.chestData = data;
}

#endregion

#region Method Region
#endregion

#region Virtual Method region

public override object Clone()
{
    ChestData data = new ChestData();

    data.Name = chestData.Name;
    data.IsLocked = chestData.IsLocked;
    data.IsTrapped = chestData.IsTrapped;
    data.TextureName = chestData.TextureName;
    data.TrapName = chestData.TrapName;
    data.KeyName = chestData.KeyName;
    data.MinGold = chestData.MinGold;
    data.MaxGold = chestData.MaxGold;

    foreach (KeyValuePair<string, string> pair in chestData.ItemCollection)
        data.ItemCollection.Add(pair.Key, pair.Value);

    Chest chest = new Chest(data);

    return chest;
}
#endregion
```

```
    }
}
```

There is one using statement that I added for the **TrapClasses** in the **RpgLibrary**. It's not used right now but it will be required. There is a static field that is a **Random** object. It will be used to generate the gold that a chest contains. There is a **ChestData** field that holds the **ChestData** associated with the chest. The **IsLocked** property exposes the **IsLocked** field in the **ChestData** object. The **IsTrapped** property works with the **IsTrapped** field in the **ChestData** object. The **Gold** property is interesting. It checks to see if **MinGold** and **MaxGold** are both 0 and if they the property returns 0. It then generates a random number between **MinGold** and **MaxGold**. They are then set to be 0 and the number generated is returned. What that boils down to is that the chest will remain after it is opened, in case the player can't carry all of the items for example, and if the player goes back to the chest and opens it again they won't get gold from it multiple times. That would quickly make your most expensive items in a shop purchasable and blow your game balance out of the water.

The constructor of the **Chest** class takes a **ChestData** parameter. For the call to the constructor of the base class it passes in the **Name** field, an empty string for the type, and 0 for the price and weight. It then sets the **ChestData** field.

The **Clone** method first creates a new **ChestData** object. It then assigns the fields from the field. To handle the **Dictionary<string, string>** I loop through all of the **KeyValuePair<string, string>** in the collection. I then add a new item to the **ItemCollection** of the new **ChestData** object. I then create a new chest using the new **ChestData** object and return it. It might not be necessary to create a new **ChestData** object but it is good programming practise to reduce side effects. I say that because **ChestData** is a class and that makes it a reference type. Passing around references directly you can inadvertently change the original reference and that would be a hard bug to track down. When you are passing around reference types and you think the original could be changed it is always better to pass a copy.

To actually create chests you will really want to update the editor, and add a class for keys. I'm going to go a head and add chests to the **Level** class though. You are going to need a few graphics for chests and other containers then. I created a **PNG** file from a tile set off the web that has a chest in it. I will add other containers as I find them and when I use them in future tutorials I'll link to the graphics. You can find my chest at http://cynthiamcmahon,ca/blog/downloads/containers1.zip. Download and extract the files.

Open the **MonoGame Pipeline Tool** in the **EyesOfTheDragon** project by expanding the **Content** folder and double clicking the **Content.mgcb** file. Right click the **Content** node, select **Add** and then **New Folder**, Name this new folder **ObjectSprites**. Right click the **ObjectSprites** folder, select **Add** and then **Existing Item**. Navigate to the **containers.png** file and selected. If prompted choose to copy the file to the **Content** folder. Save the project and close the **MonoGame Pipeline Tool**.

Chests are tied to a specific map, and maps are tied to specific levels. So chests, like characters, belong in the **Level** class. Update the **Level** in the **MGRpgLibrary** class to the following.

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Text;
using MGRpgLibrary.TileEngine;
using MGRpgLibrary.CharacterClasses;
using MGRpgLibrary.ItemClasses;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

namespace MGRpgLibrary.WorldClasses
{
    public class Level
    {
        #region Field Region

        readonly TileMap map;
        readonly List<Character> characters;
        readonly List<ItemSprite> chests;

        #endregion
        #region Property Region

        public TileMap Map
        {
            get { return map; }
        }

        public List<Character> Characters
        {
            get { return characters; }
        }

        public List<ItemSprite> Chests
        {
            get { return chests; }
        }

        #endregion

        #region Constructor Region

        public Level(TileMap tileMap)
        {
            map = tileMap;
            characters = new List<Character>();
            chests = new List<ItemSprite>();
        }

        #endregion

        #region Method Region

        public void Update(GameTime gameTime)
        {
            foreach (Character character in characters)
                character.Update(gameTime);

            foreach (ItemSprite sprite in chests)
                sprite.Update(gameTime);
        }
```

```
        public void Draw(GameTime gameTime, SpriteBatch spriteBatch, Camera camera)
        {
            map.Draw(spriteBatch, camera);

            foreach (Character character in characters)
                character.Draw(gameTime, spriteBatch);

            foreach (ItemSprite sprite in chests)
                sprite.Draw(gameTime, spriteBatch);
        }

        #endregion
    }
}
```

The first new thing is that there is a using statement bring the **ItemClasses** of the **MGRpgLibrary** into scope. I added a **List<ItemSprite>** called chests to hold the chests in a level. It is readonly so it can't be assigned to outside of the constructor or as an initializer in the class. In the constructor I create the **List<ItemSprite>**. In the **Update** method I loop through all of the **ItemSprite** objects in the list of **ItemSprites** and call their **Update** method passing in the **GameTime** parameter. I do the same in the **Draw** method, except I **Draw** instead of **Update**, passing in the **SpriteBatch** parameter as well.
The last thing I want to do in this tutorial is to actually have a chest show up. I thought the best place to do that would be in the **CreateWorld** method of the **CharacterGeneratorScreen**. You could also make the same changes to the **LoadGameScreen** but I'm not going to do that officially. The first thing you are going to want to do is to add using statements for both **ItemClasses** name spaces of the libraries. Add these two using statements to the **CharacterGeneratorScreen**.

```
using MGRpgLibrary.ItemClasses;
using RpgLibrary.ItemClasses;
```

It would be a good idea to load the texture for containers in the **LoadContent** method. Add the following field and change the **LoadContent** method to the following.

```
Texture2D containers;

protected override void LoadContent()
{
    base.LoadContent();
    LoadImages();
    CreateControls();
    containers = Game.Content.Load<Texture2D>(@"ObjectSprites\containers");
}
```

Now, in the **CreateWorld** method you want to create a chest and added it to the list of chests for the level. Change the **CreateWorld** method to the following.

```
        private void CreateWorld()
        {
            Texture2D tilesetTexture = Game.Content.Load<Texture2D>(@"Tilesets\tileset1");
            Tileset tileset1 = new Tileset(tilesetTexture, 8, 8, 32, 32);

            tilesetTexture = Game.Content.Load<Texture2D>(@"Tilesets\tileset2");

            Tileset tileset2 = new Tileset(tilesetTexture, 8, 8, 32, 32);
```

```
List<Tileset> tilesets = new List<Tileset>();

tilesets.Add(tileset1);
tilesets.Add(tileset2);

MapLayer layer = new MapLayer(100, 100);

for (int y = 0; y < layer.Height; y++)
{
    for (int x = 0; x < layer.Width; x++)
    {
        Tile tile = new Tile(0, 0);
        layer.SetTile(x, y, tile);
    }
}

MapLayer splatter = new MapLayer(100, 100);

Random random = new Random();

for (int i = 0; i < 100; i++)
{
    int x = random.Next(0, 100);
    int y = random.Next(0, 100);
    int index = random.Next(2, 14);

    Tile tile = new Tile(index, 0);
    splatter.SetTile(x, y, tile);
}

splatter.SetTile(1, 0, new Tile(0, 1));
splatter.SetTile(2, 0, new Tile(2, 1));
splatter.SetTile(3, 0, new Tile(0, 1));

List<MapLayer> mapLayers = new List<MapLayer>();

mapLayers.Add(layer);
mapLayers.Add(splatter);

TileMap map = new TileMap(tilesets, mapLayers);
Level level = new Level(map);

ChestData chestData = new ChestData();

chestData.Name = "Some Chest";
chestData.MinGold = 10;
chestData.MaxGold = 101;

Chest chest = new Chest(chestData);

BaseSprite chestSprite = new BaseSprite(
    containers,
    new Rectangle(0, 0, 32, 32),
    new Point(10, 10));

ItemSprite itemSprite = new ItemSprite(
    chest,
    chestSprite);

level.Chests.Add(itemSprite);
```

```
            World world = new World(GameRef, GameRef.ScreenRectangle);

            world.Levels.Add(level);
            world.CurrentLevel = 0;

            GamePlayScreen.World = world;
        }
```

The new code starts just after creating the **Level** object. What I did was create a **ChestData** object and set the **Name** property to **“Some Chest”** and **MinGold** to 10 and **MaxGold** to 101. I then created a new **Chest** object. The next step was to create a **BaseSprite** for the chest. For the parameters to the constructor I use the **containers Textur2D** for, a **Rectangle** with 0 for X and Y, and 32 for the width and height. I also placed it at tile (10, 10). I then create an **ItemSprite** passing in the chest I created and the base sprite. It is then added to the **Chests** field of the **Level**.

So, if you build and run your game and start a new game you will see a chest! The player isn't interacting with the chest but it is a start.

I think that this is enough for this tutorial. It covers the original tutorial with little modification. 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