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.

In this tutorial I'm going to get started with adding quests and conversations to the game. I will be adding in place holders that will be filled in as the game progresses, like I did for skills, spells, and talents. I will then fill out the **Entity** class to use the new classes and add in a couple classes to the **MGRpgLibrary** as well.

To get started, right click the **RpgLibrary** project, select **Add** and then **New Folder**. Name this new folder **QuestClasses**. I'm going to add three classes to this folder. For quests I'm going to implement quests that have multiple steps. So the first class to add is a class for the steps. Right click the **QuestClasses** folder, select **Add** and then **Class**. Name this new class **QuestStep**. For now this is just a place holder. Later on down the road it will be added to. This is the code for that class so far.

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

namespace RpgLibrary.QuestClasses
{
    public class QuestStep
    {
        #region Field Region
        #endregion

        #region Property Region
        #endregion

        #region Constructor Region
        #endregion

        #region Method Region
        #endregion

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

As I mentioned, it is a place holder class that will be filled over time. It is just needed for a few things and I get started with them now. You will want to right click the **QuestClasses** folder again, select **Add** and then **Class**. Name this new class **Quest**. The code for the class is the same, just a bare class split up with regions.

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

namespace RpgLibrary.QuestClasses
{
    public class Quest
    {
        #region Field Region
        #endregion

        #region Property Region
        #endregion

        #region Constructor Region
        #endregion

        #region Method Region
        #endregion

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

As will all of the other classes it is a good idea to have a class to manage quests in the game, especially for the editor. Right click the **QuestClasses** folder, select **Add** and then **Class**. Name this new class **QuestManager**. There is actually something to this class. It should look familiar as I've used the same format several times now.

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

namespace RpgLibrary.QuestClasses
{
    public class QuestManager
    {
        #region Field Region

        readonly Dictionary<string, Quest> quests;

        #endregion

        #region Property Region

        public Dictionary<string, Quest> Quests
        {
            get { return quests; }
        }

        #endregion

        #region Constructor Region

        public QuestManager()
```

```
        {
            quests = new Dictionary<string, Quest>();
        }

        #endregion

        #region Method Region
        #endregion

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

The next set of classes that I'm going to add are for conversations. I like the approach that Jim Perry took in his book, ***RPG Programming using XNA Game Studio 3.0***, for working with conversations using nodes. In my XNA 3.0 RPG tutorials I used a system similar to the one that Nick Gravlyn used in his tile engine video tutorials. I'm trying to reduce the complexity of handling conversations.

I'm going to give the instructions for adding all the classes, then the code. Right click the **RpgLibrary**, select **Add** and then **New Folder**. Name this new folder **ConversationClasses**. Right click that folder, select **Add** and then **Class**. Name this new class **ConversationNode**. Right click the folder, select **Add** and then **Class**. Name the class **Conversation**. Right click the folder again, select **Add** and then **Class**. Name this new class **ConversationManager**. The code for all three classes follows next.

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

namespace RpgLibrary.ConversationClasses
{
    public class ConversationNode
    {
        #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.ConversationClasses
{
    public class Conversation
    {
        #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.ConversationClasses
{
    public class ConversationManager
    {
        #region Field Region

        readonly Dictionary<string, Conversation> conversations;

        #endregion

        #region Property Region

        public Dictionary<string, Conversation> Conversations
        {
            get { return conversations; }
        }

        #endregion

        #region Constructor Region

        public ConversationManager()
        {
            conversations = new Dictionary<string, Conversation>();
        }

        #endregion

        #region Method Region
        #endregion

        #region Virtual Method region
        #endregion
    }
```

```
}
```

Nothing really new there either. It is all code that you've seen before. Now I'm going to get in to some new stuff. The first thing I want to do is to update the **Entity** class. What I want to do is to add fields and properties for skills, spells, and talents. I added in new regions to the **Entity** class. I added in a **Skill Field and Property** region, a **Spell Field and Property** region, and a **Talent Field and Property** region. To these regions I added a field for each of the items and a field for modifiers of the items. Add the following regions to the **Entity** class. Also, add these using statements for the **Skill**, **Spell** and **Talent** classes.

```
using RpgLibrary.SkillClasses;
using RpgLibrary.SpellClasses;
using RpgLibrary.TalentClasses;

#region Skill Field and Property Region

readonly Dictionary<string, Skill> skills;
readonly List<Modifier> skillModifiers;

public Dictionary<string, Skill> Skills
{
    get { return skills; }
}

public List<Modifier> SkillModifiers
{
    get { return skillModifiers; }
}

#endregion

#region Spell Field and Property Region

readonly Dictionary<string, Spell> spells;
readonly List<Modifier> spellModifiers;

public Dictionary<string, Spell> Spells
{
    get { return spells; }
}

public List<Modifier> SpellModifiers
{
    get { return spellModifiers; }
}

#endregion

#region Talent Field and Property Region

readonly Dictionary<string, Talent> talents;
readonly List<Modifier> talentModifiers;

public Dictionary<string, Talent> Talents
{
    get { return talents; }
}
```

```
public List<Modifier> TalentModifiers
{
    get { return talentModifiers; }
}

#endregion
```

The next step is to create the fields. I did that in the private constructor for the class. I also made a change to the public constructor to call the private constructor. You can do that using **this** like you would use **base** to call a base constructor. Change the constructor region of the **Entity** class to the following.

```
        #region Constructor Region

        private Entity()
        {
            Strength = 10;
            Dexterity = 10;
            Cunning = 10;
            Willpower = 10;
            Magic = 10;
            Constitution = 10;
            health = new AttributePair(0);
            stamina = new AttributePair(0);
            mana = new AttributePair(0);
            skills = new Dictionary<string, Skill>();
            spells = new Dictionary<string, Spell>();
            talents = new Dictionary<string, Talent>();
            skillModifiers = new List<Modifier>();
            spellModifiers = new List<Modifier>();
            talentModifiers = new List<Modifier>();
        }

        public Entity(
            string name,
            EntityData entityData,
            EntityGender gender,
            EntityType entityType) : this()
        {
            EntityName = name;
            EntityClass = entityData.EntityName;
            Gender = gender;
            EntityType = entityType;
            Strength = entityData.Strength;
            Dexterity = entityData.Dexterity;
            Cunning = entityData.Cunning;
            Willpower = entityData.Willpower;
            Magic = entityData.Magic;
            Constitution = entityData.Constitution;
            health = new AttributePair(0);
            stamina = new AttributePair(0);
            mana = new AttributePair(0);
        }

        #endregion
```

The private constructor will be called before the public constructor. So the code there will be executed before the other constructor's code, the one that takes an **EntityData** parameter.

Another thing that I added to the **Entity** class was a method to update modifiers, called **Update**, of course. Change the **Method** region of the **Entity** class to the following. You may have to add it come to think of it.

```
#region Method Region

public void Update(TimeSpan elapsedTime)
{
    foreach (Modifier modifier in skillModifiers)
        modifier.Update(elapsedTime);

    foreach (Modifier modifier in spellModifiers)
        modifier.Update(elapsedTime);

    foreach (Modifier modifier in talentModifiers)
        modifier.Update(elapsedTime);
}

#endregion
```

The **Update** method takes a **TimeSpan** parameter that represents the **ElapsedGameTime** from the **gameTime** parameter in the **Update** method of MonoGame. In a foreach loop I loop through all of the modifiers in each of the lists of modifiers. I call their **Update** methods passing in the **elapsedTime** passed to the **Update** method. You're not going to want a lot of characters at a time as it will bog things down. Fortunately for the most part modifiers will only be needed by party members and in combat.

Since I moved to using a transformation matrix for the tile engine to control where things are rendered you no longer need to pass a **Camera** object to the **Draw** method of the **AnimatedSprite** class. Change the **Draw** method of the **AnimatedSprite** class to the following.

```
        public void Draw(GameTime gameTime, SpriteBatch spriteBatch)
        {
            spriteBatch.Draw(
                texture,
                Position,
                animations[currentAnimation].CurrentFrameRect,
                Color.White);
        }
```

You will also need to update the **Draw** method of the **Player** class. You can change that method to the following.

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

I want to add a new folder to the **MGRpgLibrary** project classes that are related to characters. Right click the **MGRpgLibrary**, select **Add** and then **New Folder**. Name this new folder **CharacterClasses**. To this folder I'm going to add two classes. Right click the **CharacterClasses** folder in the **MGRpgLibrary** select **Add**, and then **Class**. Name this new class **Character**. A **Character** will be a character in the game that doesn't have a conversation or quest associated with them. They can also be monsters or shop keepers. This is the code for that class.

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

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

        protected Entity entity;
        protected AnimatedSprite sprite;

        #endregion

        #region Property Region

        public Entity Entity
        {
            get { return entity; }
        }
        public AnimatedSprite Sprite
        {
            get { return sprite; }
        }

        #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);
        }

        #endregion
    }
}
```

I added a couple using statements to bring some of the MonoGame framework classes into scope. As well as the **CharacterClasses** from the **RpgLibrary** and the **SpriteClasses** of the **MGRpgLibrary**. There are two protected fields, **Entity** and **AnimatedSprite**. As you can guess the **Entity** field is the entity for the character and the **AnimatedSprite** is the sprite for the character. There are two public properties to expose their values that are read only. The constructor for this class takes two parameters. The first is an **Entity** object and the second is an **AnimatedSprite** object. I also added in two virtual methods that can be overridden in a class that inherits from **Character**. The **Update** method takes a **GameTime** parameter. It calls the **Update** method of the **entity** field passing in the **ElapsedGameTime** property. It also calls the **Update** method of the **sprite** field. The **Draw** method takes as parameters a **GameTime** parameter and a **SpriteBatch** parameter. The **Draw** method calls the **Draw** method of the **sprite** field. The other class I want to add will be called **NonPlayerCharacter**. It is a bit of a misnomer as the other class is for NPCs. These are just special NPCs that have conversations and quests associated with them. Right click the **CharacterClasses** folder of the **MGRpgLibrary** project, select **Add** and then **Class**. Name this new class **NonPlayerCharacter**. This is the code for that class so far.

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

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

        readonly List<Conversation> conversations;
        readonly List<Quest> quests;

        #endregion

        #region Property Region

        public List<Conversation> Conversations
        {
            get { return conversations; }
        }
```

```
        public List<Quest> Quests
        {
            get { return quests; }
        }

        public bool HasConversation
        {
            get { return (conversations.Count > 0); }
        }

        public bool HasQuest
        {
            get { return (quests.Count > 0); }
        }

        #endregion

        #region Constructor Region

        public NonPlayerCharacter(Entity entity, AnimatedSprite sprite)
            : base(entity, sprite)
        {
            conversations = new List<Conversation>();
            quests = new List<Quest>();
        }

        #endregion

        #region Method Region
        #endregion

        #region Virtual Method region

        public override void Update(GameTime gameTime)
        {
            base.Update(gameTime);
        }

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

        #endregion
    }
}
```

There are extra using statements to bring some classes into scope for this class. There are two new fields in this class. The first is a **List<Conversation>** called **conversations** that holds the conversations related to the NPC. The second is a **List<Quest>** called **quests** that holds the quests associated with theNPC. They are readonly properties so they can be assigned to as an initializer in a class or in the constructor of the class. That just keeps them from being changed unintentionally. There are get only properties to expose them to other classes. There are two other properties that are associated with an NPC. They are **HasConversation** and **HasQuest** that return Boolean values. To determine their values I compare the **Count** property of the associated list with 0. If it is greater than 0 then the NPC has a conversation or quest associated with it. The class inherits from **Character** so the constructor requires

an **Entity** and an **AnimatedSprite** for parameter to pass to the base class **Character**. In the constructor I initialize the **conversations** and **quests** fields. I went a head and added in overrides to the **Update** and **Draw** methods of the base class. All they do right now is call the base **Update** and **Draw** methods. The last thing I want to do is add characters to the **Level** class. I'm also going to update the **Draw** method to take a **GameTime** parameter as it is needed in the **Draw** method of the **Character** class. Change the **Level** class to the following.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using MGRpgLibrary.TileEngine;
using MGRpgLibrary.CharacterClasses;
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;

        #endregion

        #region Property Region

        public TileMap Map
        {
            get { return map; }
        }

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

        #endregion

        #region Constructor Region

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

        #endregion

        #region Method Region

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

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

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

        #endregion
    }
}
```

The first thing I did was add a using statement to bring the **Character** class into scope for this class. I then added a readonly field, **characters**, that is a **List<Character>**. It maybe a good idea to use a **Dictionary<string, Character>** instead and I may change that. I also added in a get only property to expose the **characters** field. The constructor create a new **List<Character>** for the level. In the **Update** method I do something that isn't the most efficient thing in the world and it will be updated later on down the road. I loop through all of the characters in the **characters** field and call their **Update** methods. This isn't very efficient but for now it is okay. It wouldn't be a bad thing if you don't have a lot of characters on a level. You will want to balance your levels so that you aren't bogging down your game with a lot of entities, or game objects is a better term, and updating them needlessly.

I'm also drawing all of the characters in the level. You would more than likely only want to draw characters that are visible. That is another bridge we can cross when we get to it.

The last thing we need to do is to update the **DrawLevel** method of the **World** class and the **Draw** method of the **GamePlayScreen** because I added a **GameTime** parameter to the **Draw** method of the **Level** class. Change the **DrawLevel** method of the **World** class to the following.

```
public void DrawLevel(GameTime gameTime, SpriteBatch spriteBatch, Camera camera)
{
    levels[currentLevel].Draw(gameTime, spriteBatch, camera);
}
```

Finally, change the **Draw** method of the **GamePlayScreen** to the following.

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

    base.Draw(gameTime);

    world.DrawLevel(gameTime, GameRef.SpriteBatch, player.Camera);
    player.Draw(gameTime, GameRef.SpriteBatch);

    GameRef.SpriteBatch.End();
```

```
}
```

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