XNA 4.0 RPG Tutorials

Part 27

Updating Components and Talents

I'm writing these tutorials for the new XNA 4.0 framework. The tutorials will make more sense if they are read in order. You can find the list of tutorials on the XNA 4.0 RPG tutorials page of my web site. I will be making my version of the project available for download at the end of each tutorial. It will be included on the page that links to the tutorials.

The first thing that I want to update is the **Player** component. Right now the sprite moves at a constant speed. If you were to run your game on a slower computer than yours the sprite will move slower. It is also possible that the sprite will move faster on a faster computer. This is a little more unlikely because an XNA game will not, by default, run at more than 60 frames per second. If the game was only running at 30 frames per second the slow down would be much more noticeable. Luckily this is an easy fix. You determine how many units you want the sprite to travel in 1 second and you the multiply that by the elapsed time in each frame there is movement. The first step is to update the **Update** method of the **Player** component. Modify that method as follows.

```
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)
    {
        Sprite.IsAnimating = true;
        motion.Normalize();

        Sprite.Position += motion * Sprite.Speed * (float)gameTime.ElapsedGameTime.TotalSeconds;
        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);
        }
    }

}
```

The change is in the if statement where I check to see if the **motion** is not the zero vector. I cast the **TotalSeconds** property of **gameTime.ElapsedGameTime** to a float and multiply the **Speed** property of **Sprite** by it. If you were to run the game now the sprite would really crawl. You need to update the **Speed** property of the sprite to an appropriate value. I did that in the **AnimatedSprite** class. What I did was increase the **speed** field and the **Speed** property. Change the **speed** field and **Speed** property to the following.

```
float speed = 200.0f;

public float Speed
{
    get { return speed; }
    set { speed = MathHelper.Clamp(speed, 1.0f, 400.0f); }
}
```

I decided on those values after trying different values. At 200 units per second the sprite moves at a good pace and seems appropriate for the game. At 400 units per second the sprite was really moving. You probably don't want to go much higher than that. Testing that the sprite is moving at a constant rate can be achieved easily. What you can do is set the **IsFixedTimeStep** property of the **Game1** class to false and the **SynchronizeWithVertivalRetrace** property of the **GraphicsDevice** to false. Change the constructor of the **Game1** class to the following.

```
public Game1()
{
    graphics = new GraphicsDeviceManager(this);

    graphics.PreferredBackBufferWidth = screenWidth;
    graphics.PreferredBackBufferHeight = screenHeight;

    ScreenRectangle = new Rectangle(
        0,
        0,
        screenWidth,
        screenHeight);

    Content.RootDirectory = "Content";

    Components.Add(new InputHandler(this));

    stateManager = new GameStateManager(this);
    Components.Add(stateManager);

    TitleScreen = new TitleScreen(this, stateManager);
    StartMenuScreen = new StartMenuScreen(this, stateManager);
    GamePlayScreen = new GamePlayScreen(this, stateManager);
    CharacterGeneratorScreen = new CharacterGeneratorScreen(this, stateManager);
    LoadGameScreen = new LoadGameScreen(this, stateManager);
    SkillScreen = new GameScreens.SkillScreen(this, stateManager);

    stateManager.ChangeState(TitleScreen);

    this.IsFixedTimeStep = false;
    graphics.SynchronizeWithVerticalRetrace = false;
}
```

Now if you build and run the game the sprite moves at the same rate as before the change. It is good practice to try and have your game run at the same speed across all computers. You may find if you create an XBOX 360 project from a Windows game that the game is very sluggish. That is because your computer is usually much faster than an XBOX 360, especially if you have a newer computer. I would suggest when you are creating a game to use this. It is a good way to have your game run the same across all computers.

I've been considering the next change for a while now. I've been considering moving the classes in the **RpgLibrary** into the **XRpgLibrary** project. The reason I didn't want to is that there would be a lot of classes in one project and organization would suffer. The reason I wanted to was to reduce the amount of dependencies. In the end I think that reducing the dependencies is worth the hit in organization.

The first step is to copy things from the **RpgLibrary** to the **XRpgLibrary**. Right click the **ConversationClasses** folder in the **RpgLibary** and select **Copy**. Right click the **XRpgLibrary** and select **Paste**. Repeat those steps with the **QuestClasses**, **SkillClasses**, **SpellClasses**, **TalentClasses** and **TrapClasses** folders. Now repeat the process with the **CharacterClasses**, **ItemClasses**, and **WorldClasses** folders. In the pop up box that appears for these select **Yes**. Select the **Mechanic.cs**, **Modifier.cs**, and **RolePlayingGame.cs** files in the **RpgLibrary** and copy them to the **XRpgLibrary** project as well.

Now expand the **References** node in the **RpgEditor** project and delete the **RpgLibrary** reference.Do the same for the **References** node in the **EyesOfTheDragonContent** project. Right click the

References node in the **EyesOfTheDragonContent** project and select **Add Reference**. From the **Project** tab select the **XRpgLibrary**. The last step is to right click the **RpgLibrary** project and select **Remove** to remove the project.

What I'm going to add next is add some code to count the frames per second. This will give you an idea of how changes you make to the game affect performance. If the frames per second drop to a really low value you will know that a change you made has made a performance hit. The first step is to add a new field region with the other field regions in the **Game1** class.

```
#region Frames Per Second Field Region

private float fps;
private float updateInterval = 1.0f;
private float timeSinceLastUpdate = 0.0f;
private float frameCount = 0;

#endregion
```

You measure frames per second in the **Draw** method. Change the **Draw** method of the **Game1** class to the following.

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    base.Draw(gameTime);

    float elapsed = (float)gameTime.ElapsedGameTime.TotalSeconds;
    frameCount++;
    timeSinceLastUpdate += elapsed;

    if (timeSinceLastUpdate > updateInterval)
    {
        fps = frameCount / timeSinceLastUpdate;
#if XBOX360
        System.Diagnostics.Debug.WriteLine("FPS: " + fps.ToString());
#else
        this.Window.Title = "FPS: " + fps.ToString();
#endif
        frameCount = 0;
        timeSinceLastUpdate -= updateInterval;
    }
}
```

The preprocessor directives may not line up like they are in the code that I pasted. The first thing the new code does is get the amount of time that has elapsed in seconds. This is going to be a small value that will eventually add up to approximately 1. I then increase the **frameCount** variable that holds the number of frames that have elapsed. The **timeSinceLastUpdate** holds the amount of time since the last update of the frames per second. The if statement check if **timeSinceLastUpdate** is greater than the **updateInterval** field, which is set to **1.0f**. If it is I set the **fps** field to **frameCount** divided by **timeSinceLastUpdate**. Then if the target platform is the Xbox 360 I write the **fps** value using the **System.Diagnostics.Debug.WriteLine** method. This will write the **fps** value to the output window in Visual Studio. If it is not the Xbox 360 I set the **Title** property of the window to be the **fps** value. The last step is to set **frameCount** back to 0 and subtract **updateInterval** from **timeSinceLastUpdate**.

The last thing I'm going to work on in this tutorial is flesh out the talents a little. There will be three types of talents. Talents will be passive, sustained, or activated. A passive talent is a talent that requires no stamina to be available and is always active. A sustained talent is a talent that requires stamina to activate and once activated is active until it is deactivated. An activated talent costs a certain amount of stamina to activate produces a specific effect. Spells will work in a similar way except they require mana instead of stamina.

For this we are going to need some new classes for effects. There will be different types of effects. So, what types of effects are there. An effect may modify a primary attribute or a secondary attribute. Secondary attributes are attributes that are derived from the primary attributes like hit points. Besides affecting attributes spells and talents may change the characters status like paralyzing them for a while. Right click the **XRpgLibrary** projects, select **Add** and then **New Folder**. Name this new folder **EffectClasses**. To this folder you are going to add three class. Right click the **EffectClasses** folder, select **Add** and then **Class**. Name this class **BaseEffect**. Repeat the process and name the classes **BaseEffectData** and **BaseEffectDataManager**. For now these are template classes for now and the code for them follows next.

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

namespace RpgLibrary.EffectClasses
{
    public enum EffectType { Damage, Heal }

    public class BaseEffect
    {
        #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.EffectClasses
{
    public class BaseEffectData
    {
        #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.EffectClasses
{
    public class BaseEffectDataManager
    {
        #region Field Region
        #endregion

        #region Property Region
        #endregion

        #region Constructor Region
        #endregion

        #region Method Region
        #endregion

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

I used **BaseEffect** rather than **Effect** because XNA already contains a classed called **Effect** that is used in 3D rendering. I just wanted to avoid confusion with that class. You will notice that I changed the namespace from **XRpgLibrary** to just **RpgLibrary**. I did that to try and keep the XNA part and the mechanics part separate. I added an enumeration to the **BaseEffect** class called **EffectType**. This is also a place holder but I added two values to it: **Damage** and **Heal**. There purpose I think is obvious. **Damage** will cause damage to a target and **Heal** will heal a target. I will be adding in more down the road.

The next class that I'm going to work on is the **TalentData** class. Change the **TalentData** class to the following. Update the code for the **TalentData** class to the following.

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

namespace RpgLibrary.TalentClasses
{
    public enum TalentType { Passive, Sustained, Activated }

    public class TalentData
    {
        #region Field Region

        public string Name;
        public string[] AllowedClasses;
        public Dictionary<string, int> AttributeRequirements;
        public string[] TalentPrerequisites;
        public int LevelRequirement;
        public TalentType TalentType;
        public int ActivationCost;
        public string[] Effects;

        #endregion

        #region Property Region
        #endregion

        #region Constructor Region

        public TalentData()
        {
            AttributeRequirements = new Dictionary<string, int>();
        }

        #endregion

        #region Method Region
        #endregion

        #region Virtual Method region

        public override string ToString()
        {
            string toString = Name;

            foreach (string s in AllowedClasses)
                toString += ", " + s;

            foreach (string s in AttributeRequirements.Keys)
                toString += ", " + s + "+" + AttributeRequirements[s].ToString();

            foreach (string s in TalentPrerequisites)
                toString += ", " + s;

            toString += ", " + LevelRequirement.ToString();

            toString += ", " + TalentType.ToString();
            toString += ", " + ActivationCost.ToString();

            foreach (string s in Effects)
                toString += ", " + s;

            return toString;
        }

        #endregion
    }
}
```

There are several fields associated with the **TalentData** class. The first is **Name**, the name of the talent. Next is an array of strings, **AllowedClasses**, that holds the classes that can learn the talent. I did this because a rogue or a fighter can, for instance, learn archery talents. Next there is a dictionary with string keys and integer values, **AttributeRequirements**, that holds any attribute values that a character must have to learn the talent. There is then an array of strings, **TalentPrerequisites**, that will hold any talents that must be learned before this talent can be learned. The next field, **LevelRequirement**, will hold what level a character must be to learn the talent. In this way a low level character can't learn a very powerful talent that will unbalance your game, whereas a high level character may need that talent against the stronger foes. The field **TalentType** is the type of talent, whether it is passive, sustained, or activated. The next one, **ActivationCost**, is the cost required to activate the talent. The last, **Effects**, is an array of strings that holds the effects that the talent may cause.

The constructor just creates a new **Dictionary<string, int>** for the **AttributeRequirements** field. The one method, **ToString**, just creates a string that represents a **TalentData** object and returns it.

Before I get to the **Talent** class I want to add a static method to the **Mechanics** class. This method will return an attribute from an entity based on the name of the attribute. This will be helpful in many places. Add the following static method to the **Mechanics** class.

```
public static int GetAttributeByString(Entity entity, string attribute)
{
    int value = 0;

    switch (attribute.ToLower())
    {
        case "strength":
            value = entity.Strength;
            break;
        case "dexterity":
            value = entity.Dexterity;
            break;
        case "cunning":
            value = entity.Cunning;
            break;
        case "willpower":
            value = entity.Willpower;
            break;
        case "magic":
            value = entity.Magic;
            break;
        case "constitution":
            value = entity.Constitution;
            break;
    }

    return value;
}
```

It is a rather simple method. I set the **value** variable to be 0. There is then a switch statement on the value of the **attribute** parameter converted to a lower case string. There is then a case for each of the primary attributes. The case sets the **value** variable to be the value of the appropriate attribute. Finally I return the value.

Now I'm going to add a few things to the **Talent** class. Change the code for the **Talent** class to the following.

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

namespace RpgLibrary.TalentClasses
{
    public class Talent
    {
        #region Field Region

        string name;
        List<string> allowedClasses;
        Dictionary<string, int> attributeRequirements;
        List<string> talentPrerequisites;
        int levelRequirement;
        TalentType talentType;
        int activationCost;
        List<string> effects;

        #endregion

        #region Property Region

        public string Name
        {
            get { return name; }
        }

        public List<string> AllowedClasses
        {
            get { return allowedClasses; }
        }

        public Dictionary<string, int> AttributeRequirements
        {
            get { return attributeRequirements; }
        }

        public List<string> TalentPrerequisites
        {
            get { return talentPrerequisites; }
        }

        public int LevelRequirement
        {
            get { return levelRequirement; }
        }

        public TalentType TalentType
        {
            get { return talentType; }
        }

        public int ActivationCost
        {
            get { return activationCost; }
        }

        public List<string> Effects
        {
            get { return effects; }
        }

        #endregion

        #region Constructor Region

        private Talent()
        {
            allowedClasses = new List<string>();
            attributeRequirements = new Dictionary<string, int>();
            talentPrerequisites = new List<string>();
            effects = new List<string>();
        }

        #endregion

        #region Method Region

        public static Talent FromTalentData(TalentData data)
        {
            Talent talent = new Talent();

            talent.name = data.Name;

            foreach (string s in data.AllowedClasses)
                talent.allowedClasses.Add(s.ToLower());

            foreach (string s in data.AttributeRequirements.Keys)
                talent.attributeRequirements.Add(
                    s.ToLower(),
                    data.AttributeRequirements[s]);

            foreach (string s in data.TalentPrerequisites)
```

```
                talent.talentPrerequisites.Add(s);

            talent.levelRequirement = data.LevelRequirement;
            talent.talentType = data.TalentType;
            talent.activationCost = data.ActivationCost;

            foreach (string s in data.Effects)
                talent.Effects.Add(s);

            return talent;
        }

        public static bool CanLearn(Entity entity, Talent talent)
        {
            bool canLearn = true;

            if (entity.Level < talent.LevelRequirement)
                canLearn = false;

            string entityClass = entity.EntityClass.ToLower();

            if (!talent.AllowedClasses.Contains(entityClass))
                canLearn = false;

            foreach (string s in talent.AttributeRequirements.Keys)
            {
                if (Mechanics.GetAttributeByString(entity, s) < talent.AttributeRequirements[s])
                {
                    canLearn = false;
                    break;
                }
            }

            foreach (string s in talent.TalentPrerequisites)
            {
                if (!entity.Talents.ContainsKey(s))
                {
                    canLearn = false;
                    break;
                }
            }

            return canLearn;
        }

        #endregion

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

First thing that I did was include a using statement to bring the **Entity** class into scope in this class. There are fields that match the fields from the **TalentData** class as well as get only properties to expose their values. Instead of arrays I use the generic **List<T>** collection to hold the values. There is a private constructor that takes no parameters. It creates new collections for the fields are collections. You don't use a constructor to create **Talent** objects you instead use the static method **FromTalentData**.

The **FromTalentData** method takes a **TalentData** parameter called **data**. The first step is to create a new **Talent** object using the private constructor. In a foreach loop I then loop through all of the entries in the **AllowedClasses** field of the **TalentData** and add each item converted to a lower case string to the **allowedClasses** field of the **Talent** object. I loop over the keys in the **AttributeRequirements** in **data** and add the key converted to a lower case string with the value to **attributeRequirements** in

talent. I do the same with **TalentPrerequisites** and **talentPrerequisites** for **data** and **talent**. There are then three straight assignments from **data** to **talent**. There is one last loop that assigns values from the **Effects** in **data** to **effects** in **talent**. I then return the **talent** variable.

The last method, **CanLearn**, is also a static method that takes an **Entity** that is trying to learn the talent and a **Talent** that represents the talent to be learned. There is a local variable **canLearn** that is set to true initially. There are then a number of checks to see if all of the prerequisites for the talent have been learned. If one of the checks fails **canLearn** will be set to false. The first check makes sure that the level of the entity passed in is not less than the required level. I then get the name of the class for the entity and convert it to a lower case string. I then use the **Contains** method of **List<T>** to see if the class is not in the list of allowed classes. There is next a foreach loop that loops through all of the keys in **AttributeRequirements**. I then use the **GetAttributeByString** method passing in the entity and the key to get the value of the attribute of the entity and compare it to the value in the dictionary for the key. If it is less than the minimum level I set can learn to false and break out of the loop. The last step is to loop through all of the prerequisites for the talent. If the talent is not I set **canLearn** to false and break out of the loop.

I think I'm going to end this tutorial here. I wanted to update a few things and add more on talents to the game. Things are starting to take form but there is a long way to go yet. I encourage you to visit the news page of my site, XNA Game Programming Adventures, for the latest news on my tutorials.

Good luck in your game programming adventures!

Jamie McMahon