Eyes of the Dragon Tutorials
Part 48
Combat Engine – Part Three

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 will be continuing on with the combat engine. The focus of this tutorial will be using the spells and talents that we have created. The first thing that we need to do is expose the spell and talent data in the DataManager. Add the following two properties to the DataManager class.

```
public static SpellDataManager SpellData
{
    get { return SpellDataManager; }
}

public static TalentDataManager TalentData
{
    get { return TalentDataManager; }
}
```

Nothing special here, it just returns the fields. I could have just made the fields public but I prefer to expose fields as properties, or use auto properties.

The next step is to update the CombatScreen. I made a number of changes so I will give you the code for the entire class. This excludes activating the spell or talent. Replace the CombatScreen with the following.

```
using EyesOfTheDragon.Components;
using MGRpgLibrary;
using MGRpgLibrary.ConversationComponents;
using MGRpgLibrary.Mobs;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using RpgLibrary.CharacterClasses;
using RpgLibrary.SpellClasses;
using RpgLibrary.TalentClasses;
using System;
using System.Collections.Generic;
using System.Text;

namespace EyesOfTheDragon.GameScreens
{
    public class CombatScreen : BaseGameState
    {
        private Texture2D _menuTexture;
        private Texture2D _actionTexture;
        private Texture2D _selected;
        private bool _displayActionTexture = false;
        private Mob _mob;
```

```
        private GameScene _scene;
        private readonly List<string> _actions = new List<string>();
        private int _action;

        public CombatScreen(Game game, GameStateManager manager) : base(game, manager)
        {
        }

        public void SetMob(Mob mob)
        {
            _mob = mob;

            _scene = new CombatScene(Game, "Combat Scene", new List<SceneOption>());

            SceneOption option = new SceneOption("Attack", "Attack", new SceneAction());

            _scene.Options.Add(option);

            if (GamePlayScreen.Player.Character.Entity.Mana.MaximumValue > 0)
            {
                option = new SceneOption("Spell", "Spell", new SceneAction());
            }
            else
            {
                option = new SceneOption("Talent", "Talent", new SceneAction());
            }
            _scene.Options.Add(option);

            option = new SceneOption("Run for your life!", "Run for your life!", new
SceneAction());
            _scene.Options.Add(option);

            ((CombatScene)_scene).SetCaption($"You face death itself in the form of a
{_mob.Entity.EntityName.ToLower()}.");
        }

        public override void Initialize()
        {
            base.Initialize();
        }

        protected override void LoadContent()
        {
            Color[] data = new Color[1280 * 200];

            for (int i = 0; i < data.Length; i++)
            {
                data[i] = Color.Black;
            }

            _menuTexture = new Texture2D(GraphicsDevice, 1280, 200);
            _menuTexture.SetData(data);

            data = new Color[200 * 720];

            for (int i = 0; i < data.Length; i++)
            {
                data[i] = Color.Blue;
            }
```

```
            _actionTexture = new Texture2D(GraphicsDevice, 200, 720);
            _actionTexture.SetData(data);

            base.LoadContent();
        }

        public override void Update(GameTime gameTime)
        {
            if (!_displayActionTexture)
            {
                _scene.Update(gameTime, PlayerIndex.One);
            }

            if (GamePlayScreen.Player.Character.Entity.Health.CurrentValue <= 0)
            {
                StateManager.ChangeState(GameRef.GameOverScreen);
            }

            if (InputHandler.KeyReleased(Microsoft.Xna.Framework.Input.Keys.Space) && !
_displayActionTexture)
            {
                switch (_scene.SelectedIndex)
                {
                    case 0:
                        if (GamePlayScreen.Player.Character.Entity.Dexterity >=
_mob.Entity.Dexterity)
                        {
                            _mob.Attack(GamePlayScreen.Player.Character.Entity);

                            if (_mob.Entity.Health.CurrentValue <= 0)
                            {
                                StateManager.PopState();
                                return;
                            }

                            _mob.DoAttack(GamePlayScreen.Player.Character.Entity);
                        }
                        else
                        {
                            _mob.DoAttack(GamePlayScreen.Player.Character.Entity);
                            _mob.Attack(GamePlayScreen.Player.Character.Entity);

                            if (_mob.Entity.Health.CurrentValue <= 0)
                            {
                                StateManager.PopState();
                                return;
                            }
                        }
                        break;
                    case 1:
                        _displayActionTexture = true;

                        _actions.Clear();
                        Entity entity = GamePlayScreen.Player.Character.Entity;

                        if (entity.Mana.MaximumValue > 0)
                        {
                            foreach (SpellData s in DataManager.SpellData.SpellData.Values)
                            {
                                foreach (string c in s.AllowedClasses)
```

```
                                {
                                    if (c == entity.EntityClass && entity.Level >=
s.LevelRequirement)
                                    {
                                        _actions.Add(s.Name);
                                    }
                                }
                            }
                        }
                        else
                        {
                            foreach (TalentData s in DataManager.TalentData.TalentData.Values)
                            {
                                foreach (string c in s.AllowedClasses)
                                {
                                    if (c == entity.EntityClass && entity.Level >=
s.LevelRequirement)
                                    {
                                        _actions.Add(s.Name);
                                    }
                                }
                            }
                        }
                        break;
                    case 2:
                        Vector2 center = GamePlayScreen.Player.Sprite.Center;
                        Vector2 mobCenter = _mob.Sprite.Center;
                        _action = 0;

                        if (center.Y < mobCenter.Y)
                        {
                            GamePlayScreen.Player.Sprite.Position.Y -= 8;
                        }
                        else if (center.Y > mobCenter.Y)
                        {
                            GamePlayScreen.Player.Sprite.Position.Y += 8;
                        }
                        else if (center.X < mobCenter.X)
                        {
                            GamePlayScreen.Player.Sprite.Position.X -= 8;
                        }
                        else
                        {
                            GamePlayScreen.Player.Sprite.Position.X += 8;
                        }

                        StateManager.PopState();
                        break;
                }
            }

            if (_displayActionTexture)
            {
                if (InputHandler.KeyReleased(Microsoft.Xna.Framework.Input.Keys.Escape))
                {
                    _displayActionTexture = false;
                    return;
                }
                else if (InputHandler.KeyReleased(Microsoft.Xna.Framework.Input.Keys.Space))
                {
```

```
                    // Use spell or talent
                }
                else if (InputHandler.KeyReleased(Microsoft.Xna.Framework.Input.Keys.Down))
                {
                    _action++;

                    if (_action >= _actions.Count)
                    {
                        _action = 0;
                    }
                }
                else if (InputHandler.KeyReleased(Microsoft.Xna.Framework.Input.Keys.Up))
                {
                    _action--;

                    if (_action < 0)
                    {
                        _action = _actions.Count - 1;
                    }
                }
            }

            base.Update(gameTime);
        }

        public override void Draw(GameTime gameTime)
        {
            base.Draw(gameTime);

            if (_selected == null)
                _selected = Game.Content.Load<Texture2D>(@"GUI\rightarrowUp");

            GameRef.SpriteBatch.Begin();

            GameRef.SpriteBatch.Draw(_menuTexture, new Rectangle(0, 720 - _menuTexture.Height,
_menuTexture.Width, _menuTexture.Height), Color.White);

            _scene.Draw(gameTime, GameRef.SpriteBatch);

            if (_displayActionTexture)
            {
                GameRef.SpriteBatch.Draw(_actionTexture, new Rectangle(_actionTexture.Width, 0,
_actionTexture.Width, _actionTexture.Height), Color.White);
                Vector2 position = new Vector2(_actionTexture.Width + 50, 5);


                int count = 0;

                foreach (string s in _actions)
                {
                    Color myColor = Color.Black;

                    if (count == _action)
                    {
                        myColor = Color.White;
                        GameRef.SpriteBatch.Draw(_selected, new Rectangle((int)position.X - 35,
(int)position.Y, _selected.Width, _selected.Height), Color.White);
                    }

                    GameRef.SpriteBatch.DrawString(FontManager.GetFont("testfont"), s,
```

```
position, myColor);
                    position.Y += FontManager.GetFont("testfont").LineSpacing + 5;

                    count++;
                }
            }

            GameRef.SpriteBatch.End();
        }
    }
}
```

The first change is the addition of two new fields: _selected and _action. _selected is a texture for what spell or talent is currently selected. It is similar to the conversation state and the combat scene that are already in use. _action is what action is currently selected.

There were a lot of changes to the Update method. First, I check to see if the action texture is not currently being displayed. If it is not I call the Update method of the combat scene. It flows as before until it gets to where it checks to see if the space bar has been released. If it has and the action texture is not being displayed it enters into the switch.

The switch has a new case added to it, for if the spell or talent has been selected. If it has I set the _displayActionTexture to true. I clear the _actions list. I assign the entity for the player to a local variable to keep from having to type it over and over again. I check to see if the entity is a spell caster or not by checking to see if the MaximumValue property of the Mana field is greater than zero. I then loop over all of the SpellData values in the DataManager using the property that I added earlier. I then loop over the classes that have the spell. If the class name for the entity equals the current value and the entity's level is greater than or equal to the level requirement I add it to the list of actions that are available. I do basically the same thing for talents by replacing spell with talent.

The next addition is a check to see if _displayActionTexture is true. If it is I check to see if the escape key has been release. If it has I set _displayActionTexture to false and exit the method. I then have a stub for if the space bar has been released. Here is where we will cast the spell or use the talent, provided there is enough mana or stamina.

If the space bar wasn't released I check to see if the down key has been released. If it has I increment _action to the next available action. If that is greater than or equal to the number of actions _action is set to zero. Then, if the up key has been released I do the opposite. I decrement _action by one. If it is less than zero I set it to the number of items minus one. This is basically the same as the menu and conversation class.

In the Draw method I check to see if _selected is null. If it is I load the texture to be displayed for selected items. As before, I check to see if the action texture should be displayed. If it is I draw the texture. Next is a Vector2 that determines where to draw the actions. I set it 50 pixels to the right of the edge of the texture and 5 pixels down. Next there is a local variable to tracks what items we are on. Then in a foreach loop I iterate over the list of available actions. Since the texture is blue I set a local color to be black, similar to game scene and menu. I then check to see if the count variable is the _action field meaning it is selected. If that is true I set the color to white and draw the selected texture

to the left of item. I then draw the action. I increment the Y component of the position to the height of the font plus five pixels. I also increment the count variable to move to the next item.

Before I get to casting spells or using talents, I need to update the spell and talent classes. I did not include an important attribute for spells and talents, the target. Unfortunately, this breaks a lot of things so there are a lot of code changes. Bare with me, this is important. I will start with the SpellData class. Update it to the following code.

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

namespace RpgLibrary.SpellClasses
{
    public enum SpellType { Passive, Sustained, Activated }
    public enum TargetType { Self, Enemy, Special }

    public class SpellData
    {
        #region Field Region

        public string Name;
        public string[] AllowedClasses;
        public Dictionary<string, int> AttributeRequirements;
        public string[] SpellPrerequisites;
        public int LevelRequirement;
        public SpellType SpellType;
        public int ActivationCost;
        public int CoolDown;
        public List<BaseEffect> Effects;

        #endregion

        #region Property Region
        #endregion

        #region Constructor Region

        public SpellData()
        {
            AttributeRequirements = new Dictionary<string, int>();
            Effects = new List<BaseEffect>();
        }

        #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 SpellPrerequisites)
                toString += ", " + s;

            toString += ", " + LevelRequirement.ToString();
            toString += ", " + SpellType.ToString();
            toString += ", " + ActivationCost.ToString();
            toString += ", " + CoolDown.ToString();

            foreach (BaseEffect e in Effects)
                toString += ", " + e.ToString();

            return toString;
        }

        #endregion
    }
}
```

The change is I added a new enumeration, TargetType, that has as values: Self, Enemy and Special. Special will be added later but Self and Enemy are self-explanitory. I then updated the Effects field to be a List<BaseEffect>. In the constructor I initialize the Effects field. In the override of the ToString method I now loop over the list of effects and append the effect as a string.

The BaseEffect and BaseEffectData classes need to be updated to use this new TargetType enumeration. Replace the BaseEffectData with the following.

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

namespace RpgLibrary.EffectClasses
{
    public class BaseEffectData
    {
        #region Field Region

        public string Name;
        public TargetType TargetType;

        #endregion

        #region Property Region
        #endregion

        #region Constructor Region
        protected BaseEffectData()
        {
        }
        #endregion

        #region Method Region
```

```
        #endregion

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

I just added a public field. The same with BaseEffect. Here is the new class.

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

namespace RpgLibrary.EffectClasses
{
    public abstract class BaseEffect
    {
        #region Field Region

        protected string name;
        public TargetType TargetType;

        #endregion

        #region Property Region

        public string Name
        {
            get { return name; }
            protected set { name = value; }
        }

        #endregion

        #region Constructor Region
        protected BaseEffect()
        {
        }

        #endregion

        #region Method Region
        #endregion

        #region Virtual Method Region

        public abstract void Apply(Entity entity);

        #endregion
    }
}
```

Next I updated the SpellDataManager to account for the change in BaseEffect. This is the code for the new SpellDataManager.

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

namespace RpgLibrary.SpellClasses
{
    public class SpellDataManager
    {
        #region Field Region

        readonly Dictionary<string, SpellData> spellData;

        #endregion

        #region Property Region

        public Dictionary<string, SpellData> SpellData
        {
            get { return spellData; }
        }

        public object EffectData { get; private set; }

        #endregion

        #region Constructor Region

        public SpellDataManager()
        {
            spellData = new Dictionary<string, SpellData>();
        }

        #endregion

        #region Method Region

        public void FillSpellData()
        {
            SpellData data = new SpellData()
            {
                Name = "Spark Jolt",
                SpellPrerequisites = new string[0],
                SpellType = SpellType.Activated,
                LevelRequirement = 1,
                ActivationCost = 8,
                AllowedClasses = new string[] { "Wizard" },
                AttributeRequirements = new Dictionary<string, int>() { { "Magic", 10 } }
            };

            BaseEffect effect = DamageEffect.FromDamageEffectData(new DamageEffectData
            {
                Name = "Spark Jolt",
                TargetType = TargetType.Enemy,
                AttackType = AttackType.Health,
                DamageType = DamageType.Air,
                DieType = DieType.D6,
                NumberOfDice = 3,
                Modifier = 2
```

```
            });

            data.Effects.Add(effect);

            spellData.Add("Spark Jolt", data);

            data = new SpellData()
            {
                Name = "Mend",
                SpellPrerequisites = new string[0],
                SpellType = SpellType.Activated,
                LevelRequirement = 1,
                ActivationCost = 6,
                AllowedClasses = new string[] { "Priest" },
                AttributeRequirements = new Dictionary<string, int>() { { "Magic", 10 } }
            };

            BaseEffect healEffect = HealEffect.FromHealEffectData(new HealEffectData
            {
                Name = "Mend",
                TargetType = TargetType.Self,
                HealType = HealType.Health,
                DieType = DieType.D8,
                NumberOfDice = 2,
                Modifier = 2
            });

            data.Effects.Add(healEffect);
            spellData.Add("Mend", data);
        }

        #endregion

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

Instead of creating a DamageEffectData for the Spark Jolt spell I create a BaseEffect for the SpellData. The BaseEffect is create using the DamageEffect class passing in a new DamageEffectData with the TargetType set to Enemy. The effect is then added to the list of effects. I follow the same flow using the HealEffect instead of a DamageEffect for the Mend spell.

I also changed the Spell class. It now has a List<BaseEffect> for effects instead of a List<string> for effects. I also updated the FromSpellData to reflect the change. Update the Spell class to the following code.

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

namespace RpgLibrary.SpellClasses
{
    public class Spell
```

```
{
    #region Field Region

    string name;
    List<string> allowedClasses;
    Dictionary<string, int> attributeRequirements;
    List<string> spellPrerequisites;
    int levelRequirement;
    SpellType spellType;
    int activationCost;
    int coolDown;
    List<BaseEffect> 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> SpellPrerequisites
    {
        get { return spellPrerequisites; }
    }

    public int LevelRequirement
    {
        get { return levelRequirement; }
    }

    public SpellType SpellType
    {
        get { return spellType; }
    }

    public int ActivationCost
    {
        get { return activationCost; }
    }

    public int CoolDown
    {
        get { return coolDown; }
    }

    public List<BaseEffect> Effects
    {
        get { return effects; }
```

```
}

#endregion

#region Constructor Region

private Spell()
{
    allowedClasses = new List<string>();
    attributeRequirements = new Dictionary<string, int>();
    spellPrerequisites = new List<string>();
    effects = new List<BaseEffect>();
}

#endregion

#region Method Region

public static Spell FromSpellData(SpellData data)
{
    Spell spell = new Spell
    {
        name = data.Name,
        levelRequirement = data.LevelRequirement,
        spellType = data.SpellType,
        activationCost = data.ActivationCost,
        coolDown = data.CoolDown
    };

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

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

    foreach (string s in data.SpellPrerequisites)
        spell.SpellPrerequisites.Add(s);

    foreach (BaseEffect s in data.Effects)
        spell.Effects.Add(s);

    return spell;
}

public static bool CanLearn(Entity entity, Spell spell)
{
    bool canLearn = true;

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

    string entityClass = entity.EntityClass.ToLower();

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

    foreach (string s in spell.AttributeRequirements.Keys)
    {
```

```
                if (Mechanics.GetAttributeByString(entity, s) < spell.AttributeRequirements[s])
                {
                    canLearn = false;
                    break;
                }
            }

            foreach (string s in spell.SpellPrerequisites)
            {
                if (!entity.Spells.ContainsKey(s))
                {
                    canLearn = false;
                    break;
                }
            }

            return canLearn;
        }

        #endregion

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

The same needs to be done to the Talent classes. First, update the TalentData class to the following code.

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

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;
        int coolDown;
        List<BaseEffect> 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 int CoolDown
{
    get { return coolDown; }
}

public List<BaseEffect> 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<BaseEffect>();
}

#endregion

#region Method Region

public static Talent FromTalentData(TalentData data)
{
```

```
            Talent talent = new Talent
            {
                name = data.Name,
                levelRequirement = data.LevelRequirement,
                talentType = data.TalentType,
                activationCost = data.ActivationCost,
                coolDown = data.CoolDown
            };

            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);

            foreach (BaseEffect 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
    }
}
```

Virtually identical to the Spell class. I really could have used one class instead of two but it is too late to go back now. It is broken, though, because the TalentData is still using a list of strings for effects. Replace the TalentData class with the following code.

```
using RpgLibrary.EffectClasses;
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 int CoolDown;
        public List<BaseEffect> Effects = new List<BaseEffect>();

        #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();
            toString += ", " + CoolDown.ToString();

            foreach (BaseEffect s in Effects)
                toString += ", " + s.ToString();

            return toString;
        }

        #endregion
    }
}
```

Finally, we need to update the TalentDataManager to account for the use of BaseEffects instead of strings. Here is the new code for the TalentDataManager.

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

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

        readonly Dictionary<string, TalentData> talentData;

        #endregion

        #region Property Region

        public Dictionary<string, TalentData> TalentData
        {
            get { return talentData; }
        }

        #endregion

        #region Constructor Region

        public TalentDataManager()
        {
            talentData = new Dictionary<string, TalentData>();
        }

        #endregion
```

```
    #region Method Region

    public void FillTalents()
    {
        TalentData data = new TalentData
        {
            Name = "Bash",
            TalentPrerequisites = new string[0],
            LevelRequirement = 1,
            TalentType = TalentType.Activated,
            ActivationCost = 5,
            AllowedClasses = new string[] { "Fighter" },
            AttributeRequirements = new Dictionary<string, int>() { { "Strength", 10 } }
        };

        DamageEffect effect = DamageEffect.FromDamageEffectData(new DamageEffectData
        {
            AttackType = AttackType.Health,
            DamageType = DamageType.Crushing,
            DieType = DieType.D8,
            Modifier = 2,
            Name = "Bash",
            NumberOfDice = 2
        });

        data.Effects.Add(effect);

        talentData.Add("Bash", data);

        data = new TalentData()
        {
            Name = "Below The Belt",
            TalentPrerequisites = new string[0],
            LevelRequirement = 1,
            TalentType = TalentType.Activated,
            ActivationCost = 5,
            AllowedClasses = new string[] { "Rogue" },
            AttributeRequirements = new Dictionary<string, int>() { { "Dexterity", 10 } }
        };

        effect = DamageEffect.FromDamageEffectData(new DamageEffectData()
        {
            AttackType = AttackType.Health,
            DamageType = DamageType.Piercing,
            DieType = DieType.D4,
            Modifier = 2,
            Name = "Below The Belt",
            NumberOfDice = 3
        });

        data.Effects.Add(effect);

        talentData.Add("Below The Belt", data);
    }

    #endregion

    #region Virtual Method region
    #endregion
}
```

```
}
```

Virtually the same as the SpellDataManager so I am not going to go over it. Now all of the pieces are in place to actually cast a spell or use a talent. It was a bit challenging so I left it out, with all the dependencies. What you need to do is replace the comment in the Update method with the following method call. For the Bash talent I had to replace Warrior with Fighter because everywhere else I use Fighter.

```
DoAction();
```

Now, we need to add the DoAction method. Add it between the Update and Draw methods. Here is the code.

```
private void DoAction()
{
    Entity entity = GamePlayScreen.Player.Character.Entity;

    if (entity.Mana.MaximumValue > 0)
    {
        SpellData spell = DataManager.SpellData.SpellData[_actions[_action]];

        if (spell.ActivationCost > entity.Mana.CurrentValue)
        {
            return;
        }

        entity.Mana.Damage((ushort)spell.ActivationCost);

        Spell s = Spell.FromSpellData(spell);

        if (s.SpellType == SpellType.Activated)
        {
            foreach (BaseEffect e in s.Effects)
            {
                switch (e.TargetType)
                {
                    case TargetType.Enemy:
                        e.Apply(_mob.Entity);
                        break;
                    case TargetType.Self:
                        e.Apply(entity);
                        break;
                }
            }
        }
    }
    else
    {
        TalentData talent = DataManager.TalentData.TalentData[_actions[_action]];

        if (talent.ActivationCost > entity.Stamina.CurrentValue)
        {
            return;
        }

        entity.Stamina.Damage((ushort)talent.ActivationCost);
```

```
        Talent s = Talent.FromTalentData(talent);

        if (s.TalentType == TalentType.Activated)
        {
            foreach (BaseEffect e in s.Effects)
            {
                switch (e.TargetType)
                {
                    case TargetType.Enemy:
                        e.Apply(_mob.Entity);
                        break;
                    case TargetType.Self:
                        e.Apply(entity);
                        break;
                }
            }
        }
    }

    _mob.DoAttack(entity);
}
```

The first thing I do is save the entity into a local variable to save myself from typing that long path to the entity property. I then check to see if the entity is a spell caster by checking the maximum mana. If it is a spell caster I get the spell data for the selected spell from the list. If the activation cost is greater than the current mana I exit the method. I then call the Damage method on the Mana property of the entity to reduce the mana. I then create a Spell object using the FromSpellData method. Next I check to see if the spell is an activated spell. If it is I loop over all of the BaseEffects. Inside that there is a switch on the target type. If it is enemy I call the Apply method passing in the Entity property of the mob. If it is self I call the Apply method passing in the player's Entity property. In either case, I call the DoAttack method of the entity to hit back. I'm being generous and saying that spells and talents are faster than basic attacks. I should really update the entire logic because a dagger should hit before a maul each round. That is a future tutorial, though.

If the character is not a spell caster I do the same using talents and stamina rather than spells and mana. If you try to build and run now you will be greeted by a big fat exception when trying to deserialize weapons. Similarly, the editors for weapons are broken. This is because of the changes to the BaseEffectData class. It is complaining because the attribute TargetType is missing. The simplest solution, and the one I am going to go with, is to edit the XML files and add a TargetType element to them. Replace the XML files with these new versions from: http://cynthiamcmahon.ca/blog/downloads/weapons_48.zip.

If you build and run now you should be able to use spells and talents in combat. So, that is going to be it for this tutorial. It is longer than I had planned but it was necessary to cover all of the topics, and I accomplished what I intended and I don't want to venture further in this tutorial. So, please continue to visit my blog, https://cynthiamcmahon.ca/blog/, for the latest news on my tutorials and other goodness.

Good luck with your Game Programming Adventures!

Cynthia