Eyes of the Dragon Tutorials
Part 57
Going Real-Time 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 with real-time combat. I will be enabling activating spells and talents in real-time. To begin, I created a base class that the Talent and Spell class will inherit from, because they are essentially the same. Add the following class named Activation to the MGRpgLibrary project.

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

namespace MGRpgLibrary
{
    public abstract class Activation
    {
        public bool InUse { get; set; }
        public double CastTime { get; set; }
        public double Duration { get; set; }
        public double Range { get; set; }
        public double AreaOfEffect { get; set; }
        public double AngleOfEffect { get; set; }
    }
}
```

As you can see, this is an abstract class that the classes that inherit it will instantiate. I added a few properties to the class. The first, InUse, is a boolean that tells if the spell or talent is currently in use. Next is CastTime, which is how long it takes to activate the spell or talent. Duration tells how long a spell or talent takes. A duration of 0 means the spell or talent is instantaneous. Following that is TimeLeft, which is how much longer the activation has until it is complete. Range is how far the spell or talent can be cast. AreaOfEffect is a measure of the distance the spell or talent reaches. Finally, AngleOfEffect is a measure in radians the angle which the activation has.

These properties need to be implemented in the data classes for spells and talents. Other than the InUse property, which is specific to an activation. Also, I want a finer degree of control when it comes to cool downs. For that reason, I changed the CoolDown field from an integer to a double. Change the SpellData and TalentData classes to the following.

SpellData

```
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 double CoolDown;
        public List<BaseEffect> Effects;

        #endregion

        #region Property Region

        public double CastTime { get; set; }
        public double Duration { get; set; }
        public double Range { get; set; }
        public double AreaOfEffect { get; set; }
        public double AngleOfEffect { get; set; }

        #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
    }
}
```

TalentData

```
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 double CoolDown;
        public List<BaseEffect> Effects = new List<BaseEffect>();

        #endregion

        #region Property Region

        public double CastTime { get; set; }
        public double Duration { get; set; }
        public double Range { get; set; }
        public double AreaOfEffect { get; set; }
        public double AngleOfEffect { get; set; }

        #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
    }
}
```

So, the next step is to update the spell data and talent data instances that we have to include these new properties. First, change the SpellDataManager to the following.

```
using Microsoft.Xna.Framework;
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 } },
        AreaOfEffect = 32,
        AngleOfEffect = MathHelper.TwoPi,
        CastTime = 0,
        Duration = 0,
        CoolDown = 2,
        Range = 128,
    };

    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 } },
```

```
                Range = 0,
                AreaOfEffect = 0,
                AngleOfEffect = 0,
                CastTime = 0,
                CoolDown = 2
            };

            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
    }
}
```

So, what is new is that I updated the creation of the data classes to include the new properties, and add a cool down. For Spark Jolt, I set the AreaOfEffect to 32 pixels and the AngleOfEffect to TwoPi so it will affect a circle. Next, I set the cast time to 0 for instantaneous. The duration is 0, so it is instantaneous. It can be cast every 2 seconds. The range is 128 pixels, or 4 tiles. The Mend spell has a range of 0 pixels because it is cast on one's self. The AreaOfEffect is 0 as well. The CastTime is also zero with a CoolDown of 2 seconds.

We also have to do the TalentDataManager so that the TalentData instances will use the new properties. Replace the TalentDataManager class with the following code.

```
using Microsoft.Xna.Framework;
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 } },
        AreaOfEffect = 8,
        AngleOfEffect = MathHelper.TwoPi,
        Range = 48,
        CastTime = 0,
        CoolDown = 1.5
    };

    DamageEffect effect = DamageEffect.FromDamageEffectData(new DamageEffectData
    {
        TargetType = SpellClasses.TargetType.Enemy,
        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 } },
```

```
                AreaOfEffect = 8,
                AngleOfEffect = MathHelper.TwoPi,
                Range = 48,
                CastTime = 0,
                CoolDown = 1.5
            };

            effect = DamageEffect.FromDamageEffectData(new DamageEffectData()
            {
                TargetType = SpellClasses.TargetType.Enemy,
                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
    }
}
```

The changes are in the FillTalents method. In Bash I set the AreaOfEffect to 8 pixels. AngleOfEffect is set to TwoPi so it will affect anything within that area. Range is set to 48 pixels. CastTime is a quarter of a second, and CoolDown is one and a half seconds. Below The Belt also has an AreaOfEffect equal to 8 pixels and an AngleOfEffect of TwoPi so it is circular. The Range is 48 pixels as well. The CastTime is a quareter of a second and the CoolDown is 1.5 seconds.

I want to have a finer degree over control of the cool down of spells and talents. For that reason, I switched from having an integer for the cool down fields and properties. Also, I changed the Spell and Talent classes' FromSpellData and FromTalentData methods to initialize the new properties. Change the Spell and Talent classes to the following code.

Talent

```
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;
double 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 double 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,
                Range = data.Range,
                AreaOfEffect = data.AreaOfEffect,
                AngleOfEffect = data.AngleOfEffect,
                CastTime = data.CastTime
            };

            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
    }
}
```

Spell

```
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;
        double 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 double 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,
            Range = data.Range,
            AreaOfEffect = data.AreaOfEffect,
            AngleOfEffect = data.AngleOfEffect,
            CastTime = data.CastTime
        };

        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
    }
}
```

So, all of the mechanics are in place now. The next step is to be able to activate spells and talents. What I am going to do is add a new quasi-state that allows for targeting. If the range is 0 then it won't appear. If it is greater than zero a targeting bubble will appear. I won't be dealing with cones at this point, they will be a future tutorial. We need a targeting graphic. Download and add the following graphic to the GUI folder of the Content project, https://cynthiamcmahon.ca/blog/target.png.

Now that we have the graphic it is time to change the GamePlayScreen. The changes are extensive so I will give you the code for the base ones first. Then, I will update the methods involved as we go. Update the GamePlayScreen to the following code.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using EyesOfTheDragon.Components;
using MGRpgLibrary;
using MGRpgLibrary.TileEngine;
using MGRpgLibrary.SpriteClasses;
using MGRpgLibrary.WorldClasses;
using MGRpgLibrary.CharacterClasses;
using MGRpgLibrary.Mobs;
using RpgLibrary;
using RpgLibrary.ItemClasses;

namespace EyesOfTheDragon.GameScreens
{
    public class GamePlayScreen : BaseGameState
    {
        #region Field Region

        Engine engine = new Engine(32, 32);
        static Player player;
        static World world;

        Texture2D swordUp;
        Texture2D swordRight;
        Texture2D swordDown;
        Texture2D swordLeft;

        Texture2D target;

        bool targeting;
        double castTime;

        Activation currentActivation;
        public readonly static Activation[] HotKeys = new Activation[10];

        Rectangle playerSword;
```

```
bool playerAttacking = false;
double playerTimer = 0;
int attackDirection = -1;

double _healthTimer;

#endregion

#region Property Region

public static World World
{
    get { return world; }
    set { world = value; }
}

public static Player Player
{
    get { return player; }
    set { player = value; }
}

#endregion

#region Constructor Region

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

#endregion

#region XNA Method Region

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

protected override void LoadContent()
{
    swordUp = GameRef.Content.Load<Texture2D>("ObjectSprites/Sword-Up");
    swordRight = GameRef.Content.Load<Texture2D>("ObjectSprites/Sword-Right");
    swordDown = GameRef.Content.Load<Texture2D>("ObjectSprites/Sword-Down");
    swordLeft = GameRef.Content.Load<Texture2D>("ObjectSprites/Sword-Left");
    target = GameRef.Content.Load<Texture2D>("GUI/target");

    base.LoadContent();
}

public override void Update(GameTime gameTime)
{
    if (targeting)
    {
        if (InputHandler.KeyPressed(Keys.Escape))
        {
            targeting = false;
        }
```

```
                if (InputHandler.CheckMousePress(MouseButton.Left))
                {
                    castTime = 0;
                }

                return;
            }

            world.Update(gameTime);
            player.Update(gameTime);

            player.Camera.LockToSprite(player.Sprite);

            HandleHotKeyInput();

            HandleConversation();

            HandleMobs(gameTime);

            if (InputHandler.KeyReleased(Keys.I))
            {
                StateManager.PushState(GameRef.InventoryScreen);
            }

            if (InputHandler.KeyReleased(Keys.C))
            {
                StateManager.PushState(GameRef.StatsScreen);
                Visible = true;
            }

            if (Player.Character.Entity.Level < Mechanics.Experiences.Length)
            {
                if (Player.Character.Entity.Experience >=
Mechanics.Experiences[Player.Character.Entity.Level])
                {
                    Player.Character.Entity.LevelUp();
                    StateManager.PushState(GameRef.LevelScreen);
                    Visible = true;
                }
            }

            if (InputHandler.CheckMousePress(MouseButton.Left) && playerTimer > 0.25 && !
playerAttacking)
            {
                playerAttacking = true;
                playerTimer = 0;

                if (player.Sprite.CurrentAnimation == AnimationKey.Up)
                {
                    attackDirection = 0;
                }
                else if (player.Sprite.CurrentAnimation == AnimationKey.Right)
                {
                    attackDirection = 1;
                }
                else if (player.Sprite.CurrentAnimation == AnimationKey.Down)
                {
                    attackDirection = 2;
                }
                else
```

```
                {
                    attackDirection = 3;
                }
            }

            if (playerTimer >= 0.25)
            {
                playerAttacking = false;
            }

            playerTimer += gameTime.ElapsedGameTime.TotalSeconds;

            if (player.Character.Entity.Health.CurrentValue <= 0)
            {
                StateManager.PushState(GameRef.GameOverScreen);
            }

            if (player.Character.Entity.Health.CurrentValue <
player.Character.Entity.Health.MaximumValue)
            {
                _healthTimer += gameTime.ElapsedGameTime.TotalSeconds;

                if (_healthTimer > 1)
                {
                    _healthTimer = 0;
                    player.Character.Entity.Health.Heal(2);
                    player.Character.Entity.Mana.Heal(2);
                    player.Character.Entity.Stamina.Heal(2);
                }
            }

            base.Update(gameTime);
        }

        private void HandleMobs(GameTime gameTime)
        {
            MobLayer mobLayer = World.Levels[World.CurrentLevel].Map.Layers.Find(x => x is
MobLayer) as MobLayer;

            if (playerAttacking)
            {
                foreach (var mob in mobLayer.Mobs.Values)
                {
                    if (playerSword.Intersects(mob.Sprite.Bounds))
                    {
                        if (player.Character.Entity.MainHand != null &&
                            player.Character.Entity.MainHand.Item is Weapon)
                        {
                            mob.Entity.ApplyDamage(player.Character.Entity.MainHand);
                            playerAttacking = false;

                            if (mob.Entity.Health.CurrentValue <= 0)
                            {
                                StateManager.PushState(GameRef.LootScreen);

GamePlayScreen.Player.Character.Entity.AddExperience(mob.XPValue);
                                GameRef.LootScreen.Gold = mob.GoldDrop;

                                foreach (var i in mob.Drops)
                                {
```

```
                                GameRef.LootScreen.Items.Add(i);
                            }
                        }
                    }
                }
            }

            foreach (var mob in mobLayer.Mobs.Where(kv => kv.Value.Entity.Health.CurrentValue
<= 0).ToList())
            {
                mobLayer.Mobs.Remove(mob.Key);
            }

            foreach (var mob in mobLayer.Mobs.Values)
            {
                mob.DoAttack(player.Sprite, player.Character.Entity);
                mob.ShouldAttack(player.Sprite);
            }

            foreach (var mob in mobLayer.Mobs.Values)
            {
                float distance = Vector2.Distance(player.Sprite.Center, mob.Sprite.Center);

                if (distance < mob.Sprite.Width * 4)
                {
                    Vector2 motion = Vector2.Zero;

                    if (mob.Sprite.Position.X < player.Sprite.Position.X)
                    {
                        motion.X = 1;
                        mob.Sprite.CurrentAnimation = AnimationKey.Right;
                    }

                    if (mob.Sprite.Position.X > player.Sprite.Position.X)
                    {
                        motion.X = -1;
                        mob.Sprite.CurrentAnimation = AnimationKey.Left;
                    }

                    if (mob.Sprite.Position.Y < player.Sprite.Position.Y)
                    {
                        motion.Y = 1;
                        mob.Sprite.CurrentAnimation = AnimationKey.Down;
                    }

                    if (mob.Sprite.Position.Y > player.Sprite.Position.Y)
                    {
                        motion.Y = -1;
                        mob.Sprite.CurrentAnimation = AnimationKey.Up;
                    }

                    if (motion != Vector2.Zero)
                    {
                        motion.Normalize();
                    }

                    float speed = 200f;

                    motion *= speed * (float)gameTime.ElapsedGameTime.TotalSeconds;
```

```
            mob.Sprite.Position += motion;
            mob.Sprite.IsAnimating = true;

            if (mob.Sprite.Bounds.Intersects(player.Sprite.Bounds))
            {
                mob.Sprite.Position -= motion;
            }
        }
        else
        {
            mob.Sprite.IsAnimating = false;
        }
    }
}

private void HandleConversation()
{
    if (InputHandler.KeyReleased(Keys.Space) ||
        InputHandler.ButtonReleased(Buttons.A, PlayerIndex.One))
    {
        foreach (ILayer layer in World.Levels[World.CurrentLevel].Map.Layers)
        {
            if (layer is CharacterLayer)
            {
                foreach (Character c in ((CharacterLayer)layer).Characters.Values)
                {
                    float distance = Vector2.Distance(
                        player.Sprite.Center,
                        c.Sprite.Center);

                    if (distance < Character.SpeakingRadius && c is NonPlayerCharacter)
                    {
                        NonPlayerCharacter npc = (NonPlayerCharacter)c;

                        if (npc.HasConversation)
                        {
                            StateManager.PushState(GameRef.ConversationScreen);

                            GameRef.ConversationScreen.SetConversation(
                                player,
                                npc,
                                npc.CurrentConversation);

                            GameRef.ConversationScreen.StartConversation();
                        }
                    }
                    else if (distance < Character.SpeakingRadius && c is Merchant)
                    {
                        StateManager.PushState(GameRef.ShopScreen);
                        GameRef.ShopScreen.SetMerchant(c as Merchant);
                    }
                }
            }
        }
    }
}

private void HandleHotKeyInput()
{
```

```
if (InputHandler.KeyPressed(Keys.D1))
{
    if (HotKeys[0] != null)
    {
        if (HotKeys[0].Range > 0)
        {
            targeting = true;
            currentActivation = HotKeys[0];
            castTime = 0;
        }
    }
}

if (InputHandler.KeyPressed(Keys.D2))
{
    if (HotKeys[1] != null)
    {
        if (HotKeys[1].Range > 0)
        {
            targeting = true;
            currentActivation = HotKeys[1];
            castTime = 0;
        }
    }
}

if (InputHandler.KeyPressed(Keys.D3))
{
    if (HotKeys[2] != null)
    {
        if (HotKeys[2].Range > 0)
        {
            targeting = true;
            currentActivation = HotKeys[2];
            castTime = 0;
        }
    }
}

if (InputHandler.KeyPressed(Keys.D4))
{
    if (HotKeys[3] != null)
    {
        if (HotKeys[3].Range > 0)
        {
            targeting = true;
            currentActivation = HotKeys[3];
            castTime = 0;
        }
    }
}

if (InputHandler.KeyPressed(Keys.D5))
{
    if (HotKeys[4] != null)
    {
        if (HotKeys[4].Range > 0)
        {
            targeting = true;
            currentActivation = HotKeys[4];
```

```
            castTime = 0;
        }
    }
}

if (InputHandler.KeyPressed(Keys.D6))
{
    if (HotKeys[5] != null)
    {
        if (HotKeys[5].Range > 0)
        {
            targeting = true;
            currentActivation = HotKeys[5];
            castTime = 0;
        }
    }
}

if (InputHandler.KeyPressed(Keys.D7))
{
    if (HotKeys[6] != null)
    {
        if (HotKeys[6].Range > 0)
        {
            targeting = true;
            currentActivation = HotKeys[6];
            castTime = 0;
        }
    }
}

if (InputHandler.KeyPressed(Keys.D8))
{
    if (HotKeys[7] != null)
    {
        if (HotKeys[7].Range > 0)
        {
            targeting = true;
            currentActivation = HotKeys[7];
            castTime = 0;
        }
    }
}

if (InputHandler.KeyPressed(Keys.D9))
{
    if (HotKeys[8] != null)
    {
        if (HotKeys[8].Range > 0)
        {
            targeting = true;
            currentActivation = HotKeys[8];
            castTime = 0;
        }
    }
}

if (InputHandler.KeyPressed(Keys.D0))
{
    if (HotKeys[9] != null)
```

```
        {
            if (HotKeys[9].Range > 0)
            {
                targeting = true;
                currentActivation = HotKeys[9];
                castTime = 0;
            }
        }
    }
}

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);

    if (playerAttacking)
    {
        switch (attackDirection)
        {
            case 0:
                playerSword = new Rectangle(
                    (int)player.Sprite.Position.X + (32 - swordUp.Width) / 2,
                    (int)player.Sprite.Position.Y - swordUp.Height,
                    swordUp.Width,
                    swordUp.Height);
                GameRef.SpriteBatch.Draw(swordUp, playerSword, Color.White);
                break;
            case 2:
                playerSword = new Rectangle(
                    (int)player.Sprite.Position.X + (32 - swordDown.Width) / 2,
                    (int)player.Sprite.Position.Y + 32,
                    swordDown.Width,
                    swordDown.Height);
                GameRef.SpriteBatch.Draw(swordDown, playerSword, Color.White);
                break;
            case 1:
                playerSword = new Rectangle(
                    (int)player.Sprite.Position.X + 32,
                    (int)player.Sprite.Position.Y + (32 - swordRight.Height) / 2,
                    swordRight.Width,
                    swordRight.Height);
                GameRef.SpriteBatch.Draw(swordRight, playerSword, Color.White);
                break;
            case 3:
                playerSword = new Rectangle(
                    (int)player.Sprite.Position.X - swordLeft.Width,
                    (int)player.Sprite.Position.Y + (32 - swordLeft.Height) / 2,
                    swordLeft.Width,
                    swordLeft.Height);
```

```
                        GameRef.SpriteBatch.Draw(swordLeft, playerSword, Color.White);
                        break;
                }
            }

            player.Draw(gameTime, GameRef.SpriteBatch);

            if (targeting)
            {
                Vector2 targetPosition = Player.Camera.Position + InputHandler.MouseAsVector2;
                Color tint = Color.White;
                float scale = (float)currentActivation.AreaOfEffect / target.Width;

                if (Vector2.Distance(Player.Sprite.Center, targetPosition) >
currentActivation.Range)
                {
                    tint = Color.Black;
                }

                GameRef.SpriteBatch.Draw(
                    target,
                    targetPosition,
                    null,
                    tint,
                    0f,
                    new Vector2(),
                    scale,
                    SpriteEffects.None,
                    1f);
            }
            GameRef.SpriteBatch.End();
        }

        #endregion

        #region Abstract Method Region
        #endregion
    }
}
```

Wow, that was a lot of code. I will break it down, though. First, I added a new Texture2D field that is the texture for where the player is targeting. The next new field is a boolean, targeting, that say if the player is currently targeting a spell. I am going easy on the player and pausing the game while they are targeting. The next field is castTime and it measures how much time has passed since the spell or talent was started. There is a field, currentActivation, which tells what is currently being used. The final new field is HotKeys and it is an array of 10 Activation objects, one for each digit. As you would suspect, the LoadContent method loads the texture for targeting.

There were a lot of changes to the Update method. It was getting quite large, so I split it into different methods. What I do first is check to see if targeting is true. If it is and the Escape key has been pressed, I cancel out of targeting the spell or talent. The plan is to actual invoke the action if the left mouse button has been pressed, so there is a check for that. I am pausing the game while the player is casting so I return from the method.

After calling the Update method of the world and player, I call a new method HandleHotKeyInput,

which as the name implies, handles the press of the digit keys 1 through 0. The code for handling the check for starting a conversation has moved to a new method. The code for handling mobs has also been move to its own method.

The rest of the method flows as before, up until the code that heals the player over time. Instead of just healing the player's health, it also heals the players stamina and mana. I just call that out right instead of checking to see if the character is a magic user or a mundane.

I'm going to skip over the HandleMobs method as it is the same code with no modification. The same is true for HandleConversation. The HandleHotKeyInput method is new. It is a series of checks to see if one of the digit keys have been pressed. Again, I go with pressed instead of released because I want it to trigger the millisecond that the player has pressed the key. In each case I check to see if the hot key at the index of the digit minus one is not null. If it is not null I check to see if the Range property of the activation is greater than zero. If it is I set the targeting field to true so that the targeting circle will be drawn. I set the currentActivation field to the associated activation. Finally, I set castTime to zero so that the count down to casting the spell will start. I guess it is count up because we go from zero up until it is time to cast.

The Draw method has a new if statement that checks to see if the targeting field is true. If it is, I create a Vector2 that is the position of the camera plus the mouse as a Vector2. That translates the mouse coordinates from screen coordinates to world coordinates. I then introduce a local variable, tint, that is the color to tint the targeting texture. I then create another local variable, scale, that is how much to scale the targeting texture based on its original size and the area of effect. Next, if the distance between the center of the player's sprite and the position of the mouse is greater than the range I set the tint to black so the texture will be greyed out. I then call Draw on the SpriteBatch passing in the target, position, null, tint color, 0, Vector2.Zero (in essence), scale, SpriteEffects.None, and finally 1f.

We need to circle back to the CharacterGeneratorScreen again. No, not the LoadWorld method again. It stays the same for a change. We need to update creating the character to assign their first skill to the first hot key. Replace the CreatePlayer method with the following code.

```
private void CreatePlayer()
{
    int gender = genderSelector.SelectedIndex < 2 ? genderSelector.SelectedIndex : 1;

    AnimatedSprite sprite = new AnimatedSprite(
        characterImages[gender, classSelector.SelectedIndex],
        AnimationManager.Instance.Animations);
    EntityGender g = EntityGender.Unknown;

    switch (genderSelector.SelectedIndex)
    {
        case 0:
            g = EntityGender.Male;
            break;
        case 1:
            g = EntityGender.Female;
            break;
        case 2:
            g = EntityGender.NonBinary;
            break;
```

```
    }

    Entity entity = new Entity(
        nameSelector.SelectedItem,
        DataManager.EntityData[classSelector.SelectedItem],
        g,
        EntityType.Character);

    foreach (string s in DataManager.SkillData.Keys)
    {
        Skill skill = Skill.FromSkillData(DataManager.SkillData[s]);
        entity.Skills.Add(s, skill);
    }

    Character character = new Character(entity, sprite);
    GamePlayScreen.Player = new Player(GameRef, character)
    {
        Gold = 200
    };

    if (GamePlayScreen.Player.Character.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)
                {
                    GamePlayScreen.HotKeys[0] = Spell.FromSpellData(s);
                    break;
                }
            }
        }
    }
    else
    {
        foreach (TalentData s in DataManager.TalentData.TalentData.Values)
        {
            foreach (string c in s.AllowedClasses)
            {
                if (c == entity.EntityClass && entity.Level >= s.LevelRequirement)
                {
                    GamePlayScreen.HotKeys[0] = Talent.FromTalentData(s);
                    break;
                }
            }
        }
    }
}
```

After creating the player object there is a familiar if statement that checks to see if the MaximumValue of the Mana property is greater than zero to see if the character is a spell-caster or a mundane. If the character is a spell-caster I loop over all of the SpellData objects in the SpellData in the DataManager. For each string in the AllowedClasses collection I check to see if the value is the player's selected class. If it is, I set the first hot key to be that spell from the spell data. Talents work the same as spells, just with TalentData instead of SpellData.

If you build and run now you can activate the first hot key. You can dismiss it by pressing the escape key. The rest of the game will work as before. You still can't actually cast the spell or use the talent. I am going to hold off on that for the next tutorial, I think. This tutorial is very long and I don't want to get into something new at this point. I accomplished most of 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