Eyes of the Dragon Tutorials
Part 26
Level Editor – More On Skills

I'm writing these tutorials for the MonoGame 3.8 framework using Visual Studio 2019. The tutorials will make more sense if they are read in order. You can find the list of tutorials on the Eyes of the Dragon page of my web blog. I will be making each version of the project available on GitHub here. It will be included on the page that links to the tutorials.

In this tutorial I'm going to work on the game to give you a break from the editors that I think you deserve. What I want to do first is to make a change to the way the screen manager works. Instead of switching directly to a new screen I want to add a transition period. This transition period will give the state a chance to finish off a few tasks that will leave it in a better state and help prevent cascading. It has to do with the control manager and the fact that I'm using events. The control manager should really be threaded but I don't want to go into that in these tutorials. Having a transition state will allow it to finish off tasks before switching to a new state.

First, make the **EyesOfTheDragon** project the start up project by right clicking it in the solution explorer and selecting **Set As StartUp Project**. The first thing that I did was add an enumeration to the **GameStateManager** class at the namespace level. Doing things makes it accessible with out having to reference a class name. Add this enumeration just above the class declaration for **GameStateManager**.

```
public enum ChangeType { Change, Pop, Push }
```

The enumeration has values for each of the three types of state changes. You can change states, pop the current state off the stack, and push a new state on top of the stack. It will be used to determine how to respond to switching states.

I added changing states to the **BaseGameState** class. The changes are rather extensive so I will give you the code for the entire class. Change the **BaseGameState** to the following.

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

namespace EyesOfTheDragon.GameScreens
{
    public abstract partial class BaseGameState : GameState
    {
        #region Fields region

        protected Game1 GameRef;
        protected ControlManager ControlManager;
        protected PlayerIndex playerIndexInControl;
```

```
protected BaseGameState TransitionTo;
protected bool Transitioning;
protected ChangeType changeType;
protected TimeSpan transitionTimer;
protected TimeSpan transitionInterval = TimeSpan.FromSeconds(0.5);

#endregion

#region Properties region
#endregion

#region Constructor Region

public BaseGameState(Game game, GameStateManager manager)
    : base(game, manager)
{
    GameRef = (Game1)game;
    playerIndexInControl = PlayerIndex.One;
}

#endregion

#region XNA Method Region

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

protected override void LoadContent()
{
    ContentManager Content = Game.Content;
    SpriteFont menuFont = Content.Load<SpriteFont>(@"Fonts\ControlFont");

    ControlManager = new ControlManager(menuFont);

    base.LoadContent();
}


public override void Update(GameTime gameTime)
{
    if (Transitioning)
    {
        transitionTimer += gameTime.ElapsedGameTime;

        if (transitionTimer >= transitionInterval)
        {
            Transitioning = false;

            switch (changeType)
            {
                case ChangeType.Change:
                    StateManager.ChangeState(TransitionTo);
                    break;
                case ChangeType.Pop:
                    StateManager.PopState();
                    break;
                case ChangeType.Push:
                    StateManager.PushState(TransitionTo);
```

```
                        break;
                    }
                }
            }

            base.Update(gameTime);
        }

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

        #endregion

        #region Method Region

        public virtual void Transition(ChangeType change, BaseGameState gameState)
        {
            Transitioning = true;
            changeType = change;
            TransitionTo = gameState;
            transitionTimer = TimeSpan.Zero;
        }

        #endregion
    }
}
```

What I did was add a few fields to **BaseGameState**. The first is of type **BaseGameState** and is the state that you are transitioning to. It can be set to null if you are popping a state. It isn't set directly though, I set it with a new virtual method that I created that I will get to in a bit. There is also a bool field, **Transitioning**, that determines if the screen is transitioning or not. The **changeType** field will hold what the change is. The **transitionTimer** field will be used to tell when the transition period is over and is a **TimeSpan**. The last field, **transitionInterval**, holds how long until the transition happens. I set it to be half a second initially.

In the **Update** method is where I handle the transition logic. I check to see if the **Transitioning** field is true. If it is I increment the **transitionTimer** field and then check to see if it is great than or equal to the **transitionInterval** field. If it greater than the transition interval I set the **Transitioning** field to be false. There is then a switch statement on the **changeType** field. For each of the cases I call an appropriate method of the **GameStateManager** to change the state.

The new method is the **Transition** method and it takes two parameters. The first is a **ChangeType** parameter and the second is a **BaseGameState** parameter. If the **ChangeType** is **Pop** you can pass in null for the **BaseGameState** parameter. I set the **Transitioning** field to true, **changeType** to the value passed in, **transitionTo** to the value passed in, and set **transitionTimer** to **TimeSpan.Zero**. Implementing this in the game will require changing the calls that change the state to call the new **Transition** method. I will start with the **TitleScreen** class. Change the **startLabel_Selected** method to the following.

```
private void startLabel_Selected(object sender, EventArgs e)
{
```

```
    Transition(ChangeType.Push, GameRef.StartMenuScreen);
}
```

As you can see before I was pushing the **StartMenuScreen** onto the stack using the game state manager. Instead I call the **Transition** method with **ChangeType.Push** and the **StartMenuScreen** from the **Game1** class.

In the **StartMenuScreen** class you need to change the **menuItem_Selected** method. It will be called when a menu item is selected. Change that method to the following.

```
private void menuItem_Selected(object sender, EventArgs e)
{
    if (sender == startGame)
        Transition(ChangeType.Push, GameRef.CharacterGeneratorScreen);

    if (sender == loadGame)
        Transition(ChangeType.Push, GameRef.LoadGameScreen);

    if (sender == exitGame)
        GameRef.Exit();
}
```

There is only one change in the **CharacterGeneratorScreen** and that is in the **linkLabel1_Selected** method. Change that method to the following.

```
void LinkLabel1_Selected(object sender, EventArgs e)
{
    InputHandler.Flush();

    CreatePlayer();
    CreateWorld();

    GameRef.SkillScreen.SkillPoints = 25;

    Transition(ChangeType.Change, GameRef.SkillScreen);
}
```

In the **SkillScreen** class you need to change the **acceptLabel_Selected** method. Change that method to the following.

```
void acceptLabel_Selected(object sender, EventArgs e)
{
    undoSkill.Clear();

    Transition(ChangeType.Change, GameRef.GamePlayScreen);
}
```

That leaves the **LoadGameScreen**. In that class there are two methods that need to be updated. They are the **exitLinkLabel_Selected** and **loadListBox_Selected** method. Change those methods to the following.

```
void exitLinkLabel_Selected(object sender, EventArgs e)
{
    Transition(ChangeType.Pop, null);
```

```
}

void loadListBox_Selected(object sender, EventArgs e)
{
    loadLinkLabel.HasFocus = true;
    ControlManager.AcceptInput = true;

    Transition(ChangeType.Change, GameRef.GamePlayScreen);

    CreatePlayer();
    CreateWorld();
}
```

For some reason the **LoadGameScreen** is missing in the **Game1** class. Add the following property and update the constructor of the **Game1** class.

```
public LoadGameScreen LoadGameScreen { get; private set; }

public Game1()
{
    _graphics = new GraphicsDeviceManager(this);
    ScreenRectangle = new Rectangle(
        0,
        0,
        ScreenWidth,
        ScreenHeight);

    Content.RootDirectory = "Content";

    Components.Add(new InputHandler(this));

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

    TitleScreen = new TitleScreen(this, _gameStateManager);
    StartMenuScreen = new StartMenuScreen(this, _gameStateManager);
    GamePlayScreen = new GamePlayScreen(this, _gameStateManager);
    CharacterGeneratorScreen = new CharacterGeneratorScreen(this, _gameStateManager);
    SkillScreen = new SkillScreen(this, _gameStateManager);
    LoadGameScreen = new LoadGameScreen(this, _gameStateManager);

    _gameStateManager.ChangeState(TitleScreen);
}
```

I'm now going to revisit skills for the rest of this tutorial. After looking at the numbers I was using it would be too easy for a character to get a 100 ranking in a skill. What I intend to do is instead of starting with 25 skill points and 10 every level drop that down to 10 skill points to start with and 5 more every level. I've also been thinking about adding a configuration file that you can create to set values like this in an editor and read them in at run time. It will make personalizing things a little easier.

To get started I'm going to modify the **Modifier** structure. I'm going to make it a class and add in a string field that will tell what is being modified. I was going to place a **Modifier** field in each of the **Skill**, **Talent**, and **Spell** classes. Instead in the **Entity** class I will have a fields that contains all of the modifiers for one type. There will be a **List<Modifier>** for the skills, talents, and spells an entity has

learned. Change the code of the **Modifier** structure to the following.

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

namespace RpgLibrary
{
    public class Modifier
    {
        #region Field Region

        public string Modifying;
        public int Amount;
        public int Duration;
        public TimeSpan TimeLeft;

        #endregion

        #region Constructor Region

        public Modifier(string modifying, int amount)
        {
            Modifying = modifying;
            Amount = amount;
            Duration = -1;
            TimeLeft = TimeSpan.Zero;
        }

        public Modifier(string modifying, int amount, int duration)
        {
            Modifying = modifying;
            Amount = amount;
            Duration = duration;
            TimeLeft = TimeSpan.FromSeconds(duration);
        }
        #endregion

        #region Method Region

        public void Update(TimeSpan elapsedTime)
        {
            if (Duration == -1)
                return;

            TimeLeft -= elapsedTime;

            if (TimeLeft.TotalMilliseconds < 0)
            {
                TimeLeft = TimeSpan.Zero;
                Amount = 0;
            }
        }
        #endregion

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

I just added in a new field, **Modifying**, that says what is being modified. I change the two constructors to take a string parameter for what is being modified and set the field to the value passed in.
The next thing to do is to flesh out the **Skill** class. Update the **Skill** class to the following.

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

namespace RpgLibrary.SkillClasses
{
    public enum DifficultyLevel
    {
        Master = -25,
        Expert = -10,
        Improved = -5,
        Normal = 0,
        Easy = 10,
    }

    public class Skill
    {
        #region Field Region

        string skillName;
        string primaryAttribute;
        readonly Dictionary<string, int> classModifiers;
        int skillValue;

        #endregion

        #region Property Region

        public string SkillName
        {
            get { return skillName; }
        }

        public int SkillValue
        {
            get { return skillValue; }
        }

        public string PrimaryAttribute
        {
            get { return primaryAttribute; }
        }

        public Dictionary<string, int> ClassModifiers
        {
            get { return classModifiers; }
        }

        #endregion
        #region Constructor Region

        private Skill()
        {
```

```
    skillName = "";
    primaryAttribute = "";
    classModifiers = new Dictionary<string, int>();
    skillValue = 0;
}

#endregion

#region Method Region

public void IncreaseSkill(int value)
{
    skillValue += value;

    if (skillValue > 100)
        skillValue = 100;
}

public void DecreaseSkill(int value)
{
    skillValue -= value;
    if (skillValue < 0)
        skillValue = 0;
}

public static Skill FromSkillData(SkillData data)
{
    Skill skill = new Skill
    {
        skillName = data.Name,
        skillValue = 0
    };

    foreach (string s in data.ClassModifiers.Keys)
    {
        skill.classModifiers.Add(s, data.ClassModifiers[s]);
    }

    return skill;
}

public static int AttributeModifier(int attribute)
{
    int result = 0;
    if (attribute < 25)
        result = 1;
    else if (attribute < 50)
        result = 2;
    else if (attribute < 75)
        result = 3;
    else if (attribute < 90)
        result = 4;
    else if (attribute < 95)
        result = 5;
    else
        result = 10;

    return result;
}
#endregion
```

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

The first thing I did was tweak the numbers for the **DifficultyLevel** enum. With the values I had it could be impossible for any character to perform a skill that was ranked at **Master** level. I added in four fields. Two string fields, one for the name of the skill and one for the primary attribute associated with the skill. I also added in a **Dictionary<string, int>** that holds the modifiers for the different classes and the skill. The last is an integer field for the value of the skill. There a public read only, or get only, properties to expose the values of the fields.

There is a private constructor for this class. It initializes the values of the fields to known states. Instead of a public constructor I will create **Skill** objects using a public static method and the **SkillData** class.

I included two public methods, **IncreaseSkill** and **DecreaseSkill**, that increase and decrease the value of a skill respectively. The **IncreaseSkill** method makes sure that the base value of a skill never goes above 100. Similarly, the **DecreaseSkill** method makes sure that the base value of a skill never goes below 0.

The next method is the static method, **FromSkillData**, that takes a **SkillData** object. This method creates a new **Skill** object. It sets the **skillName** field to the **Name** of the **SkillData** object. I initialize the **skillValue** field to 0. In a foreach loop I loop through all of the keys in the **ClassModifiers** dictionary. I add each entry to the **Dictionary<string, int>** in the **Skill** class. I finally return the **Skill** object.

The last method that I added is also a public static method, **AttributeModifier**, that takes an integer value. This method returns a modifier for a skill based on a character's attribute. I have a local variable, **result**, that is initially assigned to 0. Then there is a set of chained if-else-if statements that will set the **result** local variable. The values will need to be tweaked more than likely to make the game balanced. At the end of the method I return the **result** variable.

I add a static method to the **Mechanics** class to determine if using a skill is successful or not. The way I evaluate it is like this. You take the base value of the skill, add in the difficulty associated with trying to use the skill, and add any modifiers to the attempt. If the roll of a d100 is less than or equal to that number the attempt succeeds. Add the following method to the **Mechanics** class as well as the using statement for the skill classes.

```
using RpgLibrary.SkillClasses;

public static bool UseSkill(Skill skill, Entity entity, DifficultyLevel difficulty)
{
    bool result = false;
    int target = skill.SkillValue + (int)difficulty;

    foreach (string s in skill.ClassModifiers.Keys)
    {
        if (s == entity.EntityClass)
```

```
        {
            target += skill.ClassModifiers[s];
        }
    }

    foreach (Modifier m in entity.SkillModifiers)
    {
        if (m.Modifying == skill.SkillName)
        {
            target += m.Amount;
        }
    }

    string lower = skill.PrimaryAttribute.ToLower();

    switch (lower)
    {
        case "strength":
            target += Skill.AttributeModifier(entity.Strength);
            break;
        case "dexterity":
            target += Skill.AttributeModifier(entity.Dexterity);
            break;
        case "cunning":
            target += Skill.AttributeModifier(entity.Cunning);
            break;
        case "willpower":
            target += Skill.AttributeModifier(entity.Willpower);
            break;
        case "magic":
            target += Skill.AttributeModifier(entity.Magic);
            break;
        case "constitution":
            target += Skill.AttributeModifier(entity.Constitution);
            break;
    }

    if (Mechanics.RollDie(DieType.D100) <= target)
        result = true;

    return result;
}
```

The method takes as parameters the **Skill** being used, the **Entity** using the skill, and a **DifficultyLevel** for the attempt. The first thing I do is declare a local variable **result** set to false that determines if the attempt was successful or not. I then declare a local variable **target** that will be the target number of the attempt. It is initially set to the **SkillValue** property of the skill plus the **DifficultyLevel** passed in. There is then a foreach loop that loops through the **Keys** of the **ClassModifiers** dictionary of the skill. If the **Entity**'s class is one of the keys I add the value of the **ClassModifiers** entry in the dictionary. There is a second foreach loop that loops through all of the modifiers in the **SkillModifiers** collection of the entity. If the **Modifying** property of the **Modifier** is the **SkillName** property of the skill I add the **Amount** property of the modifier to **target**. I then get the **PrimaryAttribute** of the skill as a lower case string. In a switch I add the **AttributeModifier** result of the **Skill** class to **target** based on the attribute of the **Entity**. I use the **RollDie** method of the **Mechanics** class to roll a D100 and compare it to **target**. If it is less than or equal to **target** then **result** is set to true. Since **result** was initially set to false there is no need to include an else clause to set **result** to false.

The last thing I'm going to do in this tutorial is update the **SkillScreen** class for assigning points to a skill. I first want to update the **CreatePlayer** method of the **CharacterGeneratorScreen**. What I want to do is add skills to the entity. Update the **CreatePlayer** method to the following. Also add a using statement for the **RpgLibrary.Skills** classes.

```
using RpgLibrary.SkillClasses;

private void CreatePlayer()
{
    Dictionary<AnimationKey, Animation> animations =
        new Dictionary<AnimationKey, Animation>();

    Animation animation = new Animation(3, 32, 32, 0, 0);
    animations.Add(AnimationKey.Down, animation);

    animation = new Animation(3, 32, 32, 0, 32);
    animations.Add(AnimationKey.Left, animation);

    animation = new Animation(3, 32, 32, 0, 64);
    animations.Add(AnimationKey.Right, animation);

    animation = new Animation(3, 32, 32, 0, 96);
    animations.Add(AnimationKey.Up, animation);

    int gender = genderSelector.SelectedIndex < 2 ? genderSelector.SelectedIndex : 1;

    AnimatedSprite sprite = new AnimatedSprite(
        characterImages[gender, classSelector.SelectedIndex],
        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);
}
```

What I added was a foreach loop that loops through all the keys in the **SkillData** dictionary in the **DataManager** component. I create a new **Skill** using the **FromSkillData** method that I created earlier in the tutorial. I then add the skill to the **Skills** dictionary of the entity.

There were a lot of changes to the **SkillScreen** so I'm going to give you the code for the entire class. Update the **SkillScreen** to the following.

```
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 Microsoft.Xna.Framework.Content;
using MGRpgLibrary;
using MGRpgLibrary.Controls;
using MGRpgLibrary.CharacterClasses;
using RpgLibrary.SkillClasses;
using EyesOfTheDragon.Components;

namespace EyesOfTheDragon.GameScreens
{
    internal class SkillLabelSet
    {
        internal Label Label;
        internal Label SkillLabel;
        internal LinkLabel LinkLabel;
        internal int SkillValue;

        internal SkillLabelSet(Label label, Label skillLabel, LinkLabel linkLabel)
        {
            Label = label;
            SkillLabel = skillLabel;
            LinkLabel = linkLabel;
            SkillValue = 0;
        }
    }

    public class SkillScreen : BaseGameState
    {
        #region Field Region

        int skillPoints;
        int unassignedPoints;
        Character target;
        PictureBox backgroundImage;
        Label pointsRemaining;
        List<SkillLabelSet> skillLabels = new List<SkillLabelSet>();
        Stack<string> undoSkill = new Stack<string>();
        EventHandler linkLabelHandler;

        #endregion

        #region Property Region

        public int SkillPoints
        {
```

```
        get { return skillPoints; }
        set
        {
            skillPoints = value;
            unassignedPoints = value;
        }
    }

    #endregion

    #region Constructor Region

    public SkillScreen(Game game, GameStateManager manager)
        : base(game, manager)
    {
        linkLabelHandler = new EventHandler(addSkillLabel_Selected);
    }

    #endregion

    #region Method Region

    public void SetTarget(Character character)
    {
        target = character;

        foreach (SkillLabelSet set in skillLabels)
        {
            set.SkillValue = character.Entity.Skills[set.Label.Text].SkillValue;
            set.SkillLabel.Text = set.SkillValue.ToString();
        }
    }

    #endregion

    #region Virtual Method region

    #endregion

    #region XNA Method Region

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

    protected override void LoadContent()
    {
        base.LoadContent();

        ContentManager Content = GameRef.Content;
        CreateControls(Content);
    }

    private void CreateControls(ContentManager Content)
    {
        backgroundImage = new PictureBox(
        Game.Content.Load<Texture2D>(@"Backgrounds\titlescreen"),
        GameRef.ScreenRectangle);
        ControlManager.Add(backgroundImage);
```

```
Vector2 nextControlPosition = new Vector2(300, 150);

pointsRemaining = new Label
{
    Text = "Skill Points: " + unassignedPoints.ToString(),
    Position = nextControlPosition
};
nextControlPosition.Y += ControlManager.SpriteFont.LineSpacing + 10f;
ControlManager.Add(pointsRemaining);

foreach (string s in DataManager.SkillData.Keys)
{
    SkillData data = DataManager.SkillData[s];
    Label label = new Label
    {
        Text = data.Name,
        Type = data.Name,
        Position = nextControlPosition
    };

    Label sLabel = new Label
    {
        Text = "0",
        Position = new Vector2(
            nextControlPosition.X + 300,
            nextControlPosition.Y)
    };

    LinkLabel linkLabel = new LinkLabel
    {
        TabStop = true,
        Text = "Add",
        Type = data.Name,
        Position = new Vector2(
            nextControlPosition.X + 390,
            nextControlPosition.Y)
    };

    nextControlPosition.Y += ControlManager.SpriteFont.LineSpacing + 10f;
    linkLabel.Selected += addSkillLabel_Selected;

    ControlManager.Add(label);
    ControlManager.Add(sLabel);
    ControlManager.Add(linkLabel);

    skillLabels.Add(new SkillLabelSet(label, sLabel, linkLabel));
}

nextControlPosition.Y += ControlManager.SpriteFont.LineSpacing + 10f;

LinkLabel undoLabel = new LinkLabel
{
    Text = "Undo",
    Position = nextControlPosition,
    TabStop = true
};

undoLabel.Selected += new EventHandler(undoLabel_Selected);
nextControlPosition.Y += ControlManager.SpriteFont.LineSpacing + 10f;
```

```
    ControlManager.Add(undoLabel);

    LinkLabel acceptLabel = new LinkLabel
    {
        Text = "Accept Changes",
        Position = nextControlPosition,
        TabStop = true
    };

    acceptLabel.Selected += new EventHandler(acceptLabel_Selected);

    ControlManager.Add(acceptLabel);
    ControlManager.NextControl();
}
void acceptLabel_Selected(object sender, EventArgs e)
{
    undoSkill.Clear();
    Transition(ChangeType.Change, GameRef.GamePlayScreen);
}

void undoLabel_Selected(object sender, EventArgs e)
{
    if (unassignedPoints == skillPoints)
        return;

    string skillName = undoSkill.Peek();

    undoSkill.Pop();
    unassignedPoints++;

    foreach (SkillLabelSet set in skillLabels)
    {
        if (set.LinkLabel.Type == skillName)
        {
            set.SkillValue--;
            set.SkillLabel.Text = set.SkillValue.ToString();

            target.Entity.Skills[skillName].DecreaseSkill(1);
        }
    }

    pointsRemaining.Text = "Skill Points: " + unassignedPoints.ToString();
}

void addSkillLabel_Selected(object sender, EventArgs e)
{
    if (unassignedPoints <= 0)
        return;

    string skillName = ((LinkLabel)sender).Type;

    undoSkill.Push(skillName);
    unassignedPoints--;

    // Update the skill points for the appropriate skill
    foreach (SkillLabelSet set in skillLabels)
    {
        if (set.LinkLabel.Type == skillName)
        {
```

```
                    set.SkillValue++;
                    set.SkillLabel.Text = set.SkillValue.ToString();

                    target.Entity.Skills[skillName].IncreaseSkill(1);
                }
            }

            pointsRemaining.Text = "Skill Points: " + unassignedPoints.ToString();
        }

        public override void Update(GameTime gameTime)
        {
            ControlManager.Update(gameTime, PlayerIndex.One);

            base.Update(gameTime);
        }

        public override void Draw(GameTime gameTime)
        {
            GameRef.SpriteBatch.Begin();

            base.Draw(gameTime);

            ControlManager.Draw(GameRef.SpriteBatch);

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

I removed the using statement for the System.IO name space as I no longer read in the file names for the skills. I instead use the **DataManager** that I added to get the skill names. I did add a using statement for the **EyesOfTheDragon.Components** and **XRpgLibrary** name spaces.
I updated the **SkillLabelSet** class. I added another **Label** to the class that is used to draw the points the character has allocated. I also added an integer to hold the skill points that are allocated to a skill. I modified the constructor to take another **Label**. The constructor assigns fields to values passed in and sets **SkillValue** to 0.

I added on new field to the **SkillScreen** class, **target**, which is a **Character**. You will set this field with the target character that you want to assign skill points to. At the moment there is just the one character but eventually I will be moving to a party system where other characters may join the player character. When one of those characters levels up you will be able to assign skill points to them.

I added a public method, **SetTarget**, that is used to set the **target** field. I first set the **target** field to the value passed in. In a foreach loop I loop over all of the **SkillLabelSet** items. I set the **SkillValue** to the **SkillValue** of the appropriate skill of the target passed in. I then update the **Text** property of **SkillLabel** to be the **SkillValue** passed in.

There were many changes to the **CreateControls** method. I removed the code that loaded in the skills using the Content Pipeline. I instead get the skills from the **DataManager**. It still creates a background image and a **Label** that displays the unassigned points. The biggest difference is in the foreach loop. It loops through all of the keys in the **SkillData** dictionary of the **DataManager**. There is a variable to

hold the appropriate **SkillData** called **data**. I create a new **Label** and set its **Text** and **Type** property to be the **Name** of the **SkillData** item. I position it like before. I then create another **Label** and set its **Text** property to "0" initially. I position it 300 pixels to the right of the first label. I then create a **LinkLabel** like before but position it 390 pixels to the right of the first label. I subscribe to the **Selected** property of the **LinkLabel** as well. I then add all three controls to the **ControlManager**. I also add an item to the **SkillLabelSet** collection. The rest of the method continues on as before.

I also updated the **undoLabel_Selected** and **addSkillLabel_Selected** methods. Before there was a comment in each method that you needed to update the skill points of the target. I now actually update the skill points assigned to a skill.

In the **undoLabel_Selected** method I handle removing a previously assigned skill point. The new code is a foreach loop that loops through all of the **SkillLabelSet** items in the collection. If the **Type** property of the **LinkLabel** in the set is equal to the **skillName** that I got off the **undoSkill** stack I reduce the **SkillValue** of the set item by 1. I update the **Text** property of the **SkillLabel** to show the change. I also call the **DecreaseSkill** method of the appropriate skill of the **target** field.
The **addSkillLabel_Selected** method works the same way. I just increment instead of decrement.

There is a foreach loop that loops through all of the **SkillLabelSet** items in the collection. If the **Type** property of the **LinkLabel** of the set is equal to the **skillName** variable, that I got from the **LinkLabel** triggered the event, I increment **SkillValue** by 1, update the **Text** property, and call the **IncreaseSkill** method passing in 1. Ideally you would want to add a check here that **SkillValue** doesn't go over the maximum of 100. I will also mention that we are dealing with raw values. The character can assign up to 100 raw points to a skill. Any modifiers for their class or primary attribute are extras and are not applied in the raw skill points.

I will mention that the **target** field is a reference to the actual character. Any changes you do the the **target** field will apply to the actual character. You really need to be careful when you are passing references around. There is the potential for unwanted side effects. When you assign a different character to the **target** field you are not working with the old value you are working with the new value.

The last thing to do is to update the **linkLabel1_Selected** method in the **CharacterGeneratorScreen** to pass in the character to the **SkillScreen**. Change that method to the following.

```
void LinkLabel1_Selected(object sender, EventArgs e)
{
    InputHandler.Flush();

    CreatePlayer();
    CreateWorld();

    GameRef.SkillScreen.SkillPoints = 10;

    Transition(ChangeType.Change, GameRef.SkillScreen);

    GameRef.SkillScreen.SetTarget(GamePlayScreen.Player.Character);
}
```

I also updated the method to have 10 skill points to start with rather than 25. After calling **Transition** I call the **SetTarget** method of the **SkillScreen** passing in **GamePlayScreen.Player.Character**, the player's character.

That's it for this tutorial. It covers the original tutorial with some modifications. So, I encourage you to visit my blog at, https://cynthiamcmahon.ca/blog/, for the latest news on my tutorials and other goodness.

Good luck in your Game Programming Adventures!

Cynthia