XNA 4.0 RPG Tutorials

Part 20

More on Skills

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

In this tutorial I'm going to do a little more work on skills. What I plan to do is to add in a screen to allow the player to spend their skill points. Initially I will move to this screen from the main screen for creating characters, **CharacterGeneratorScreen**. First thing though, I kind of goofed. I shouldn't have had you navigate to the **EyesOfTheDragonContent** folder for entering skills. You should have added a **Skills** folder inside of the **Game** folder in the **EyesOfTheDragonContent** folder. Easy enough to fix though. Like you did for adding the **Game** folder to the **EyesOfTheDragonContent** folder open up a windows explorer window and navigate to the **Game** folder in the **EyesOfTheDragonContent** folder. Select the **Skills** folder from the **Game** folder in windows explorer and drag it onto the **Game** folder in the solution explorer.

I want to first add a new control similar to the **LeftRightSelector**. The difference is that this control will work with numeric values instead of strings, much like the **Numeric Up Down** control I've been using in the editor, which is also called a **SpinBox**. In the **XRpgLibrary** right click the **Controls** folder, select **Add** and then **Class**. Name this new class **SpinBox**. This is the code for that class.

```
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;

namespace XRpgLibrary.Controls
{
    public class SpinBox : Control
    {
        #region Event Region

        public event EventHandler SelectionChanged;

        #endregion

        #region Field Region

        int current;
        int minValue;
        int maxValue;
        int increment;

        Texture2D leftTexture;
        Texture2D rightTexture;
```

```
        Texture2D stopTexture;

        Color selectedColor = Color.Red;
        int width;

        #endregion

        #region Property Region

        public int MinimumValue
        {
            get { return minValue; }
            set
            {
                if (value > maxValue)
                    minValue = maxValue;
                else
                    minValue = value;
            }
        }

        public int MaximumValue
        {
            get { return maxValue; }
            set
            {
                if (value < minValue)
                    maxValue = minValue;
                else
                    maxValue = value;
            }
        }

        public int Value
        {
            get { return current; }
            set
            {
                if (value < minValue)
                    current = minValue;
                else if (value > maxValue)
                    current = maxValue;
                else
                    current = value;
            }
        }

        public int Increment
        {
            get { return increment; }
            set { increment = value; }
        }

        public int Width
        {
            get { return width; }
            set { width = value; }
        }

        public Color SelectedColor
        {
            get { return selectedColor; }
            set { selectedColor = value; }
        }

        #endregion

        #region Constructor Region
```

```
        public SpinBox(Texture2D leftArrow, Texture2D rightArrow, Texture2D stop)
        {
            minValue = 0;
            maxValue = 100;
            increment = 1;
            width = 50;

            leftTexture = leftArrow;
            rightTexture = rightArrow;
            stopTexture = stop;

            TabStop = true;
            Color = Color.White;
        }

        #endregion

        #region Method Region
        #endregion

        #region Virtual Method region


        public override void Update(GameTime gameTime)
        {
        }

        public override void Draw(SpriteBatch spriteBatch)
        {
            Vector2 drawTo = position;

            if (current != minValue)
                spriteBatch.Draw(leftTexture, drawTo, Color.White);
            else
                spriteBatch.Draw(stopTexture, drawTo, Color.White);

            drawTo.X += leftTexture.Width + 5f;
            string currentValue = current.ToString();
            float itemWidth = spriteFont.MeasureString(currentValue).X;
            float offset = (width - itemWidth) / 2;

            drawTo.X += offset;

            if (hasFocus)
                spriteBatch.DrawString(spriteFont, currentValue, drawTo, selectedColor);
            else
                spriteBatch.DrawString(spriteFont, currentValue, drawTo, Color);

            drawTo.X += -1 * offset + width + 5f;

            if (current != maxValue)
                spriteBatch.Draw(rightTexture, drawTo, Color.White);
            else
                spriteBatch.Draw(stopTexture, drawTo, Color.White);
        }

        public override void HandleInput(PlayerIndex playerIndex)
        {
            if (InputHandler.ButtonReleased(Buttons.LeftThumbstickLeft, playerIndex) ||
                InputHandler.ButtonReleased(Buttons.DPadLeft, playerIndex) ||
                InputHandler.KeyReleased(Keys.Left))
            {
                current -= increment;
                if (current < minValue)
                    current = minValue;
                OnSelectionChanged();
            }

            if (InputHandler.ButtonReleased(Buttons.LeftThumbstickRight, playerIndex) ||
                InputHandler.ButtonReleased(Buttons.DPadRight, playerIndex) ||
```

```
                InputHandler.KeyReleased(Keys.Right))
            {
                current += increment;
                if (current > maxValue)
                    current = maxValue;
                OnSelectionChanged();
            }
        }

        protected virtual void OnSelectionChanged()
        {
            if (SelectionChanged != null)
                SelectionChanged(this, null);
        }

        #endregion
    }
}
```

This should look a little familiar. The biggest difference is that you are working with numbers rather than strings. There are using statements to bring some of the XNA framework classes into scope. I included an event like in the **LeftRightSelector** called **SelectionChanged**. This event will be triggered if the selection changed, much like the **LeftRightSelector**.

There are four integer fields: **current**, **minValue**, **maxValue**, and **increment**. They hold the current value of the **SpinBox**, the minimum and maximum values and how much to increment when right is pressed or decrement when left is pressed. The fields **leftTexture**, **rightTexture**, and **stopTexture** are from the **LeftRightSelector**. The left and right ones will let the player know they can move left or right and the stop they can't move in a specific direction. The last field, **width**, is the width of the **SpinBox**.

There are a number of properties to expose the values and many do error checking. **MinimumValue** is for the minimum value. The set part checks to see if the value passed to the property is greater than the maximum value. If it is it sets the minimum value to the maximum value. **MaximumValue** is for the maximum value. The set part checks if the value passed in is less than the minimum value. If it is it sets the maximum value to be the minimum value. Otherwise they set the minimum or maximum to the value passed in. The **Value** property exposed the **current** field. If the value is less than the minimum value **current** is set to be the minimum value. If it is greater than the maximum value it is set to be the maximum value. Otherwise it is set to the value passed in. The **Increment** property should do some sort of validation. You don't want the increment to be greater than the difference between the minimum and maximum values. You also don't want it to be less than or equal to zero. For now it is fine but something that should probably be looked into. The **Width** property exposes the **width** field. Again, you should probably do a little validation here but we will just be careful. The last property is for the selected color and is called **SelectedColor**. It is just a simple get and set property.

The constructor for this class takes three **Texture2D** parameter. The first is a left pointing arrow, the second a right pointing arrow, and the third a stop symbol. I set a few default values for the **SpinBox** first. The minimum and maximum values are set to 0 and 100 respectively. The **increment** field is set to be 1 and the **width** field to 50. I set the **Texture2D** fields to the values passed to the constructor. I also set **TabStop** to true and **Color** to white.

The **Draw** method is similar to that of the **LeftRightSelector**. There is a local variable to hold where to draw an element of the **SpinBox**. I check to see if **current** is not equal to **minValue**. If it isn't then I draw the left arrow. Otherwise I draw the stop bar. I increment the **X** property of **drawTo** by the width of the left arrow and 5 pixels for padding. I create a string from **current** called **currentValue**. I then

measure the width of **currentValue** using the **MeasureString** method. I then create an offset that will center the item and increment the **X** value of **drawTo**. I check the **hasFocus** field to determine what color to draw the item in and draw it in the appropriate color. By comparing **current** to **maxValue** I determine if I should draw the right arrow or the stop bar.

The **HandleInput** method first checks to see if the player released the left thumb stick left, direction pad left, or left arrow key left. I then decrement **current** by **increment**. If **current** is less than **minValue** I set **current** to be **minValue**. I also call the **OnSelectionChanged** method that will check if the event **SelectionChanged** is subscribed to and should be fired. I then check if the player released the left thumb stick right, the direction pad right, or the right arrow key. If so, I increment **current** by the **icrement** field. If **current** is greater than **maxValue** I set current to be **maxValue**. I then call the **OnSelectionChanged** method to see if the event should be fired.

I had planned on using the **SpinBox** control in this tutorial. After trying to work with it for a bit I found it wasn't really the best choice for the job and I decided to go a different route that was simpler to implement. The control will be useful down the road so I kept it in the tutorial.

The next step is to add a screen to handle distributing skill points. Right click the **GameScreens** in the **EyesOfTheDragon** project, select **Add** and then **Class**. Name this new class **SkillScreen**. The code for that screen follows next.

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

using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Content;

using XRpgLibrary;
using XRpgLibrary.Controls;
using RpgLibrary.SkillClasses;

namespace EyesOfTheDragon.GameScreens
{
    internal class SkillLabelSet
    {
        internal Label Label;
        internal LinkLabel LinkLabel;

        internal SkillLabelSet(Label label, LinkLabel linkLabel)
        {
            Label = label;
            LinkLabel = linkLabel;
        }
    }

    public class SkillScreen : BaseGameState
    {
        #region Field Region

        int skillPoints;
        int unassignedPoints;

        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
        #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);

            string skillPath = Content.RootDirectory + @"\Game\Skills\";
            string[] skillFiles = Directory.GetFiles(skillPath, "*.xnb");

            for (int i = 0; i < skillFiles.Length; i++)
                skillFiles[i] = @"Game\Skills\" +
Path.GetFileNameWithoutExtension(skillFiles[i]);

            List<SkillData> skillData = new List<SkillData>();

            Vector2 nextControlPosition = new Vector2(300, 150);

            pointsRemaining = new Label();
```

```
            pointsRemaining.Text = "Skill Points: " + unassignedPoints.ToString();
            pointsRemaining.Position = nextControlPosition;

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

            ControlManager.Add(pointsRemaining);

            foreach (string s in skillFiles)
            {
                SkillData data = Content.Load<SkillData>(s);

                Label label = new Label();
                label.Text = data.Name;
                label.Type = data.Name;

                label.Position = nextControlPosition;

                LinkLabel linkLabel = new LinkLabel();
                linkLabel.TabStop = true;
                linkLabel.Text = "+";
                linkLabel.Type = data.Name;

                linkLabel.Position = new Vector2(
                    nextControlPosition.X + 350,
                    nextControlPosition.Y);

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

                linkLabel.Selected += addSkillLabel_Selected;

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

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

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

            LinkLabel undoLabel = new LinkLabel();
            undoLabel.Text = "Undo";
            undoLabel.Position = nextControlPosition;
            undoLabel.TabStop = true;
            undoLabel.Selected += new EventHandler(undoLabel_Selected);
            nextControlPosition.Y += ControlManager.SpriteFont.LineSpacing + 10f;

            ControlManager.Add(undoLabel);

            LinkLabel acceptLabel = new LinkLabel();
            acceptLabel.Text = "Accept Changes";
            acceptLabel.Position = nextControlPosition;
            acceptLabel.TabStop = true;
            acceptLabel.Selected += new EventHandler(acceptLabel_Selected);

            ControlManager.Add(acceptLabel);
            ControlManager.NextControl();
        }

        void acceptLabel_Selected(object sender, EventArgs e)
        {
            undoSkill.Clear();
            StateManager.ChangeState(GameRef.GamePlayScreen);
        }

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

            string skillName = undoSkill.Peek();
            undoSkill.Pop();
```

```
            unassignedPoints++;

            // Update the skill points for the appropriate skill
            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

            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
    }
}
```

There are several using statements that I added to this class. I needed class from the **System.IO** name space for finding all of the skills in the game. There are also using statements to bring a few of the **XNA** framework classes into scope. I also added using statements to bring some of the **XRpgLibrary** and **RpgLibrary** classes into scope as well.

There is a second class in the code for the **SkillScreen** called **SkillLabelSet**. Its purpose is to hold a pair of values: a **Label** and a **LinkLabel**. For this tutorial it isn't too important but it will be down the road. It is an internal class. The internal access modifier means that the member is public inside of the assembly and private outside of the assembly. So, what is an assembly? Example of assemblies are our game and our libraries. They are a collection of types and resources that are built to work together and form a logical unit of functionality.

The **SkillScreen** class itself inherits from the **BaseGameState** class so it can be used in the state manager and gives us access to a few protected fields. There are several fields in this class. The first, **skillPoints**, is the total number of skill points the player has to spend. The next, **unassignedPoints**, is the number of points the player has left to spend. The **backgroundImage** field is for the background image of the screen, exactly like the other screens that use a background image. The **pointsRemaining Label** will be used to let the player know how many points they have left to spend on skills.
The next field is a **List<SkillLabelSet>** called **skillLabels**. The reason this field is hear is that we are reading in our skills at run time rather than having them hard coded. If they were hard coded our job

would be simpler as we could just design the screen statically rather than dynamically. This is an example of how we are making more of an engine than a game. The engine can be easily modified using data files to make the game you want rather than modifying the code. The next field is a **Stack<string>** called **undoSkill**. I decided it would be nice to include an undo feature when spending skill points. Using a stack is an easy way to implement an undo system. When you perform an action you push that action onto the stack. If you want to undo the action, you pop it off the stack and reverse the action.

The last field may have many of you scratching your heads. It is an **EventHandler** field named **linkLabelHandler**. This field will be used to wire event handlers for the **LinkLabels** in the **SkillLabelSet**. The **LinkLabels** will be generated dynamically. You don't know how many of them there will be. You want to wire events to them but how do you create an event handler dynamically? The way I decided to handle it was to have an **EventHandler** field and create an instance of that event handler in the constructor. Then for any control that wants to implement that event you can assign the field.

I'm sure there are a lot of you scratching your heads here. This is an advanced topic and even people who've been programming for years struggle with it when they see it. I'm not going into events and event handlers in this tutorial. I'm going to be writing a tutorial, out side of the RPG tutorials, that deals with events and event handlers. As I said, they cam be a complex topic and deserve a tutorial of their own.

I also included a single property call **SkillPoints**. This property is used to get the **skillPoints** field and set the **skillPoints** and **unassignedPoints**. I don't want to allow setting the **unassignedPoints** directly. That could lead to extra skill points being available. Using the property to set them makes sure that initially they are the same as the skill points.

What the constructor does is create a new **EventHandler** called **addSkillLabel_Selected**. Later on in the class you will see there is a method **addSkillLabel_Selected**. That is the implementation of the event handling code.

The **LoadContent** method creates a local variable **Content** of type **ContentManager**. I did this as I will be using it a lot. It then calls a method, **CreateControls**, passing in the variable **Content**.

CreateControls, as the name applies, creates the controls on the screen. Like in other screens it creates a **PictureBox** for the background of the screen and adds it to the **ControlManager**. There is next a string that I set to be the **RootDirectory** property of **Content** plus **\Game\Skills** which is where the skills are compiled to by the **Content Pipeline**. There is next an array of strings called **skillFiles** that will hold all of the files in the **Skills** directory. I use the **GetFiles** method of the **Directory** class to get the file name passing in the path and ***.xnb** for the type of files. **xnb** is the extension that the **Content Pipeline** gave our content when it compiled it. Using **xnb** files protects your assets from being meddled with. If you were storing things as straight text an industrious person could get into the text and make changes to the text. Your game could either be, at best, broken, at worst perverted and passed around with your name attached to it. That is why I called the second state the worst state as it is almost a form of identity theft. There is next a for loop that loops through all of the file names that were returned. I create a new path to the file with out the extension using the **GetFileNameWithoutExtension** method of the **Path** class.

Next I have a local variable that is a **List<SkillData>** called **skillData** that will hold all of the skills temporarily. The **nextControlPostion** variable is a **Vector2** and is used for positioning controls on the screen. I then create the **Label** for **pointsRemaining**. It set its **Text** property to be **Skill Points:** plus the value of **unassignedPoints** converted to a string. I set its **Position** property to **nextControlPosition** and then increase the **Y** value of **nextControlPosition** by the **LineSpacing** property of the **SpriteFont** of **ControlManager** plus 10 pixels. I then add the **Label** to the **ControlManager**.

There is then a foreach loop that loops through all of the strings in **skillFiles**. I use the **Load** method of the **Content Manager** to load in the **SkillData** with that name. I create a **Label** and set its **Text** and **Type** properties to the **Name** property of the **SkillData** object. I set the **Position** property of the **Label** to be **nextControlPosition**. I then create a **LinkLabel**, set its **TabStop** property to true, its **Text** property to + and its **Type** property to the **Name** of the **SkillData** object. I set the **Position** property of the **LinkLabel** to be the **X** value of **nextControlPosition** plus 350 pixel and the same **Y** value of **nextControlPosition**. I then increment the **Y** value of **nextControPosition** by **LineSpacing** and 10 pixels. I then wire the event handler of the **Selected** event to be **addSkillLabel_Selected**. I don't have to use new as I already created a handler in the constructor. I can just assign the handler that I created earlier. I add the **Label** and **LinkLabel** to the **Control Manager**.

After the loop I increment the **Y** value of **nextControlPosition** by **LineSpacing** plus 10 pixels. I then create another **LinkLabel** called **undoLabel** that will handle if the player changes their mind about assigning skill points. I set its **Text** property to **Undo**, its **Position** to **nextControlPosition**, and **TabStop** to true. I then wire a new event handler for the **Selected** event. I increase the **Y** value of **nextControlPosition** by **LineSpacing** plus 10 pixels. I then add the **LinkLabel** to the **Control Manager**.

I then create one more **LinkLabel** called **acceptLabel** that allows the player to accept their choices. I set **Text** to **Accept Changes**, **Position** to **nextControlPosition** and **TabStop** to true. I then wire the handler for the **Selected** Event. I then add the control to the **Control Manager** and call the **NextControl** method of the **Control Manager** to move to the first control.

Next there are the event handlers. The **acceptLabel_Selected** method handles if the player has accepted changes. If they have, I clear the **undoSkill** stack so there is nothing to be undone. I then call the **ChangeState** method passing in the **GamePlayScreen**.

The **undoLabel_Selected** method handles the player wanting to undo an assignment of skill points. If the **unassignedPoints** and **skillPoints** fields are the same there is nothing to undo so I exit the method. I use the **Peek** method of the **Stack** class to get the value on top of the stack. I then pop the top member off of the stack. I increment the **unassignedPoints** field. There is a comment next that you will want to update the skill points for the skill of the player. I haven't added the data for the player yet so I can't do that. I then update the **Text** property of the **pointsRemaining** label to be the points left.

The **addSkillLabel_Selected** method handles the player wanting to assign a skill point to a skill. If **unassignedPoints** is less than or equal to 0 you don't want to assign any points. So I exit the method. I get the name of the skill using the **Type** property of the **LinkLabel**. The **sender** parameter is what triggered the event, in this case a **LinkLabel**. So, the order of operations is first to cast **sender** to be a **LinkLabel** and then use that to get the **Type** property. The inner brackets will be assessed before the other one. I then push **skillName** onto the stack and decrement **unassignedPoints**. There is another comment about work having to be done like in the previous method. I then update the **Text** property of the **pointsRemaining** label to be the points left.

Let's add this new screen to the game. The first thing to do is to add a field of **SkillScreen** to the **Game1** class and create it in the constructor. Add this field to the **Game State Region** and change the constructor to the following.

```
public SkillScreen SkillScreen;

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

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

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

    Content.RootDirectory = "Content";

    Components.Add(new InputHandler(this));

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

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

    stateManager.ChangeState(TitleScreen);
}
```

Nothing really new there. There is just a public field that can be accessed using the **GameRef** field in the **BaseGameState**. In the constructor I just create a new instance of **SkillScreen**.

The last thing is that you want to jump from the **CharacterGeneratorScreen** to the **SkillScreen** rather than to the **GamePlayScreen**. I did that 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;
    StateManager.ChangeState(GameRef.SkillScreen);
}
```

What I did here was call the **CreatePlayer** and **CreateWorld** methods to create a **Player** and **World** object. If you recall from the beginning I give players 25 points to spend on skills when they are first created. I assign the **SkillPoints** property of the **SkillScreen** class to 25. I then call the **ChangeState** method of the **GameStateManager** passing in the **SkillScreen**.

I'm going to end this tutorial here. The plan was to add in more to dealing with skills in the game. I encourage you to visit the news page of my site, XNA Game Programming Adventures, for the latest news on my tutorials.

Good luck in your game programming adventures!

Jamie McMahon