Eyes of the Dragon Tutorials
Part 33
Non-Player Character Conversations

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.

So, the game is coming along well but the player cannot interact with anything, making it a very boring game. In this tutorial I'm going to cover adding in NPCs for the player to interact with. The player will be able to converse with the NPCs. I will be implementing a different system than I had originally intended as it is much easier to follow.

The way that conversations are going to work is as follows. An NPC can have one or more conversations associated with them. A conversation is made up of scenes. Each scene has one or more options associated with them. So, this makes a bit of a tree like structure for conversations where the scene options drive what happens during the conversation.

Now, onto implementing this in the game. First, right click the **ConversationClasses** folder in the **MGRpgLibrary** folder and select **Delete** to remove those existing classes. Now, right click the **MGRpgLibrary** project, select Add and then New Folder. Name this new folder **ConversationComponents**. Right click the **ConversationComponents** folder, select **Add** and then **Class**. Name this new class **SceneOption**. The code for that class follows.

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

namespace MGRpgLibrary.ConversationComponents
{
    public enum ActionType
    {
        Talk,
        End,
        Change,
        Quest,
        Buy,
        Sell
    }

    public class SceneAction
    {
        public ActionType Action;
        public string Parameter;
    }

    public class SceneOption
    {
        private string optionText;
```

```
        private string optionScene;
        private SceneAction optionAction;

        private SceneOption()
        {
        }

        public string OptionText
        {
            get { return optionText; }
            set { optionText = value; }
        }

        public string OptionScene
        {
            get { return optionScene; }
            set { optionScene = value; }
        }

        public SceneAction OptionAction
        {
            get { return optionAction; }
            set { optionAction = value; }
        }

        public SceneOption(string text, string scene, SceneAction action)
        {
            optionText = text;
            optionScene = scene;
            optionAction = action;
        }
    }
}
```

The first thing that I added to this class is an enumeration called ActionType. This defines the various actions that we will implementing in the game when the play selects a scene option. Talk move the current scene to another scene. End ends the current conversation. Change changes the conversation to a new conversation. Quest adds a quest to the player's quest list. Buy will be used to open a merchant's inventory for buying items and Sell will allow the player to sell items to merchants. SceneAction is very simple class that combines an ActionType with a Parameter. I did this to make it a bit cleaner when defining scenes and serializing/deserializing scenes.

SceneOption contains three member variable, optionText, optionScene and optionAction. The first, optionText is what will be displayed to the player. The next, optionScene, is the name of a scene to transition to for this option. Finally, optionAction is the action to take when this option is selected by the player.

I next added a private constructor that takes no parameters. This constructor is included so that we can use the IntermediateSerializer to serialize and deserialize conversations to load them into the game.

Next are public properties that expose the properties to outside classes. Typically I refrain from using public setters in a property as if you do not validate the setting it can cause unexpected behaviours in your game.

Finally there is a public constructor that takes as parameters the text to display for the scene, the scene to transition to and the action for the option. Inside it just sets the members using the parameters.

Next I'm going to add the scene that is made up of options. Right click the ConversationComponents folder, select Add and then Class. Name this new class GameScene. The code for that class follows next.

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

namespace MGRpgLibrary.ConversationComponents
{
    public class GameScene
    {
        #region Field Region

        protected Game game;
        protected string textureName;
        protected Texture2D texture;
        protected SpriteFont font;
        protected string text;
        private List<SceneOption> options;
        private int selectedIndex;
        private Color highLight;
        private Color normal;
        private Vector2 textPosition;
        private static Texture2D selected;
        private Vector2 menuPosition = new Vector2(50, 475);

        #endregion

        #region Property Region

        public string Text
        {
            get { return text; }
            set { text = value; }
        }

        public static Texture2D Selected
        {
            get { return selected; }
        }

        public List<SceneOption> Options
        {
            get { return options; }
            set { options = value; }
        }

        [ContentSerializerIgnore]
```

```
        public SceneAction OptionAction
        {
            get { return options[selectedIndex].OptionAction; }
        }

        public string OptionScene
        {
            get { return options[selectedIndex].OptionScene; }
        }

        public string OptionText
        {
            get { return options[selectedIndex].OptionText; }
        }

        public int SelectedIndex
        {
            get { return selectedIndex; }
        }

        [ContentSerializerIgnore]
        public Color NormalColor
        {
            get { return normal; }
            set { normal = value; }
        }

        [ContentSerializerIgnore]
        public Color HighLightColor
        {
            get { return highLight; }
            set { highLight = value; }
        }

        public Vector2 MenuPosition
        {
            get { return menuPosition; }
        }

        #endregion

        #region Constructor Region

        private GameScene()
        {
            NormalColor = Color.Blue;
            HighLightColor = Color.Red;
        }

        public GameScene(string text, List<SceneOption> options, string textureName =
"basic_scene")
        {
            this.text = text;
            this.options = options;
            this.textureName = textureName;

            textPosition = Vector2.Zero;
        }

        public GameScene(Game game, string textureName, string text, List<SceneOption> options)
```

```
{
    this.game = game;
    this.textureName = textureName;

    LoadContent(textureName);

    this.options = new List<SceneOption>();
    this.highLight = Color.Red;
    this.normal = Color.Black;
    this.options = options;
}

#endregion

#region Method Region

public void SetText(string text)
{
    textPosition = new Vector2(450, 50);
    StringBuilder sb = new StringBuilder();

    float currentLength = 0f;

    if (font == null)
    {
        this.text = text;
        return;
    }

    string[] parts = text.Split(' ');

    foreach (string s in parts)
    {
        Vector2 size = font.MeasureString(s);

        if (currentLength + size.X < 500f)
        {
            sb.Append(s);
            sb.Append(" ");

            currentLength += size.X;
        }
        else
        {
            sb.Append("\n\r");
            sb.Append(s);
            sb.Append(" ");

            currentLength = size.X;
        }
    }

    this.text = sb.ToString();
}

public void Initialize()
{
}

protected void LoadContent(string textureName)
```

```
{
    texture = game.Content.Load<Texture2D>(@"Backgrounds\" + textureName);
    selected = game.Content.Load<Texture2D>(@"GUI\rightarrowUp");
    font = game.Content.Load<SpriteFont>(@"Fonts\scenefont");
}

public void Update(GameTime gameTime, PlayerIndex index)
{
    if (InputHandler.KeyReleased(Keys.Up) ||
        InputHandler.ButtonReleased(Buttons.LeftThumbstickUp, index))
    {
        selectedIndex--;
        if (selectedIndex < 0)
            selectedIndex = options.Count - 1;
    }
    else if (InputHandler.KeyReleased(Keys.Down) ||
        InputHandler.ButtonReleased(Buttons.LeftThumbstickDown, index))
    {
        selectedIndex++;
        if (selectedIndex > options.Count - 1)
            selectedIndex = 0;
    }
}

public void Draw(GameTime gameTime, SpriteBatch spriteBatch, Texture2D portrait)
{
    Vector2 selectedPosition = new Vector2();
    Rectangle portraitRect = new Rectangle(25, 25, 425, 425);
    Color myColor;

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

    if (textPosition == Vector2.Zero)
        SetText(text);

    if (texture != null)
        texture = game.Content.Load<Texture2D>(@"Backgrounds\" + textureName);

    if (portrait != null)
        spriteBatch.Draw(portrait, portraitRect, Color.White);

    spriteBatch.DrawString(font,
        text,
        textPosition,
        Color.White);

    Vector2 position = menuPosition;

    for (int i = 0; i < options.Count; i++)
    {
        if (i == SelectedIndex)
        {
            myColor = HighLightColor;
            selectedPosition.X = position.X - 35;
            selectedPosition.Y = position.Y;
            spriteBatch.Draw(selected, selectedPosition, Color.White);
        }
        else
            myColor = NormalColor;
```

```
                spriteBatch.DrawString(font,
                    options[i].OptionText,
                    position,
                    myColor);

                position.Y += font.LineSpacing + 5;
            }
        }
        #endregion
    }
}
```

There is a lot going on in this class. I'll tackle member variables first. There is a **Game** type field that is the reference to the game. It is used for loading content using the content manager. Next there is a string field **textureName** and a **Texture2D** field texture. These fields hold the name of the texture to draw when displaying the scene. The next member variable, **font**, is the font used for drawing the scene text. Next is **text** and it is built in a method further on in the class so that the text for the scene wraps in the screen area. Next is a **List<SceneOption>** that is the options for the scene. The **selectedInde**x member is what scene option is currently selected. The two **Color** fields hold the color to draw unselected and selected options. There is also a **Vector2** that controls where the scene text is rendered and a **Texture2D** selected that will be drawn beside the currently selected scene option.

Next are a number of properties to expose the member variables to other classes. The only thing out of the normal is that I've marked a few with attributes that define how the class is serialized using the **IntermediateSerializer** because I don't want some of the members serialized so that they are set at runtime rather than at build time.

Next up are the three constructors for this class. The first requires no parameters and is required to deserialize and load the exported XML content. The second is used in the editor to create scenes. You will notice that there is a default parameter that is set to **basic_scene**. This allows you to have a default background image and custom background image for certain scenes. The third is used in the game to load a scene based on the XML generated. This constructor sets and initializes the member variables and calls **LoadContent** to load the content associated with the scene.

Next is **SetText** and it takes a parameter **text**. First, I set the position of where to draw the text. You will notice that the position is almost half way over to the right. This is because when I call **Draw** to draw the scene it accepts a **Texture2D** parameter called **portrait** that represents the portrait of the character the player is speaking to.

After setting the **position** I create a **StringBuilder** that will be used to convert the single line of text to multiple lines of text. There is then a local variable, **currentLength**, that holds the length of the current line. I then check to make sure the **font** member variable is not null. It if is I just set the member variable to the parameter and exit the method.

I then use the **Split** method of the string class to split the string into parts on the space character. Next in a foreach loop I iterate over all of the parts. I then use **MeasureString** to determine the length of that word. If the length of the word is less than the maximum length of text on the screen I append the part to the string builder with a space and update the line length.

If the length is greater than the maximum length I append a carriage return, append the text and then append a space. I can do that because rendering text with **DrawString** allows for escape characters like \n and \r. I then reset **currentLength** to **size.X**. The last thing to do in this method is to set the text member variable to the string builder as a string.

Next there is an empty method, **Initialize**, that will be updated to initialize the scene if necessary. I included it now as I do use it my games.

LoadContent is used to load the scene content. I pass a **textureName** variable to the method that is the background that will be used to draw the scene. While we are in this method lets add a font for text in conversations. Open the **MonoGame Pipeline Tool**. Right click the Fonts folder and select **Add** and then **New Item**. Select the **Sprite Font** item and name the font scenefont. Save the project and close the **MonoGame Pipeline Tool**. As well as loading the font and the background I also load an image that will be displayed in front of the currently selected item.

The **Update** method takes a **GameTime** parameter and a **PlayerIndex** parameter. The index parameter is the index of the current game pad. It can be excluded if you do not want to support game pads in your game. In the **Update** method I check to see if the player has requested to move the selected item up or down. I check if moving the item up or down exceeds the bounds of the list of options and if does I wrap to either the first or last item in the list.

The last thing to do is draw the scene. The **Draw** method takes a **GameTime** parameter, **SpriteBatch** parameter and a **Texture2D** for the speaker's portrait. There are local variables that determine where to draw the selected item indicator, the speaker's portrait and the color to draw scene options with. I check to see if selected texture is null. If it is null I load it. If **textPosition** is **Vector2.Zero** then the text has not been set so I set it. Next if the background texture is null I load it as well. Order for drawing is important. The first item to be drawn is the background. Next, if there was a portrait parameter passed in instead of null I draw the portrait. Next I draw the speaker's text.

Next I set the position of where to draw the scene options to the base position. Next I loop over all of the options. Inside the loop I check to see if the current loop variable is the **SelectedIndex** for the options. If it is I set the color to draw the text in to be the highlight color. Next I position the location of the selected item texture to the left of the item and draw the selected item texture. Otherwise I set the color to the base color for the options. Finally I draw the text for the option. At the end of the loop I update the Y value for the position to be the line spacing for the font plus 5 pixels.

The last component to add for conversations is a class that represents a conversation. In the **MGRpgLibrary** project right click the **ConversationComponents** folder, select Add and then Class. Name this new class **Conversation**. Here is the code for the **Conversation** 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.Content;
```

```
namespace MGRpgLibrary.ConversationComponents
{
    public class Conversation
    {
        #region Field Region

        private string name;
        private string firstScene;
        private string currentScene;
        private Dictionary<string, GameScene> scenes;

        #endregion

        #region Property Region

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

        public string FirstScene
        {
            get { return firstScene; }
            set { firstScene = value; }
        }

        [ContentSerializerIgnore]
        public GameScene CurrentScene
        {
            get { return scenes[currentScene]; }
        }

        public Dictionary<string, GameScene> GameScenes
        {
            get { return scenes; }
            set { scenes = value; }
        }

        #endregion

        #region Constructor Region

        private Conversation()
        {
        }

        public Conversation(string name, string firstScene)
        {
            this.scenes = new Dictionary<string, GameScene>();
            this.name = name;
            this.firstScene = this.currentScene = firstScene;
        }

        #endregion

        #region Method Region

        public void Update(GameTime gameTime)
        {
```

```
            CurrentScene.Update(gameTime, PlayerIndex.One);
        }

        public void Draw(GameTime gameTime, SpriteBatch spriteBatch, Texture2D portrait)
        {
            CurrentScene.Draw(gameTime, spriteBatch, portrait);
        }

        public void AddScene(string sceneName, GameScene scene)
        {
            if (!scenes.ContainsKey(sceneName))
                scenes.Add(sceneName, scene);
        }

        public GameScene GetScene(string sceneName)
        {
            if (scenes.ContainsKey(sceneName))
                return scenes[sceneName];

            return null;
        }

        public void StartConversation()
        {
            currentScene = firstScene;
        }

        public void ChangeScene(string sceneName)
        {
            currentScene = sceneName;
        }

        #endregion
    }
}
```

This is really just a container for **GameScenes** and determines which scene is drawn when. There are a few member variables in this class. The first, **name**, represents the name of the conversation and is typically set to the name of the NPC. The next fields **firstScene** and **currentScene** represent which scene the conversation begins with and what scene in the conversation we are currently on. Finally there is a **Dictionary<string, GameScene>** that is a list of scenes in the current conversation.

There are public properties to expose all of the scenes to other classes. I marked the **CurrentScene** property with an attribute so that it will not be serialized when using the **IntermediateSerializer**.

I included a private constructor with no parameters for this class that will be used by the **IntermediateSerializer** when deserializing the conversation. As I'm sure I've mentioned having a parameterless constructor is a requirement for **IntermediateSerializer** being able to deserialize content.

There is a second public constructor that takes a name and **firstScene** parameter that can be used for generating conversations on the fly or in an editor. Inside the constructor I just initialize all member variables.

The **Update** method just calls the **Update** method of the current scene. Similarly, the **Draw** method just calls the **Draw** method of the current scene.

There are two method, **AddScene** and **GetScene**, that are used to add and retrieve conversations from the dictionary. They are mainly meant for an editor but they can have uses outside of an editor if you wanted to generate conversations on the fly.

When a conversation first starts it must be reset to the first scene. For that reason I included a **StartConversation** method that will just set the current scene to the first scene. I also included a **ChangeScene** method that changes the scene to the value passed as an argument.

I'm also going to include a manager class that will be used to manage all of the conversations for characters on the current map. In the **MGRpgLibrary** folder right click the **ConversationComponents** folder, select Add and then Class. Name this new class **ConversationManager.** The code for that class is next.

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

namespace MGRpgLibrary.ConversationComponents
{
    public sealed class ConversationManager
    {
        #region Field Region

        private static readonly ConversationManager instance = new ConversationManager();
        private Dictionary<string, Conversation> conversationList = new Dictionary<string,
            Conversation>();

        #endregion
        #region Property Region

        [ContentSerializer]
        public Dictionary<string, Conversation> ConversationList
        {
            get { return conversationList; }
            private set { conversationList = value; }
        }

        public static ConversationManager Instance
        {
            get { return instance; }
        }

        #endregion

        #region Constructor Region

        private ConversationManager()
        {
        }
```

```
        #endregion

        #region Method Region

        public void AddConversation(string name, Conversation conversation)
        {
            if (!conversationList.ContainsKey(name))
            {
                conversationList.Add(name, conversation);
            }
        }

        public Conversation GetConversation(string name)
        {
            return conversationList.ContainsKey(name) ? conversationList[name] : null;
        }

        public bool ContainsConversation(string name)
        {
            return conversationList.ContainsKey(name);
        }

        public void ClearConversations()
        {
            conversationList = new Dictionary<string, Conversation>();
        }

        #endregion
    }
}
```

This class uses the singleton design pattern. In this design pattern there is only ever one instance of the class that provides access to it through out the system. In this game there should only be 1 conversation manager. This manager gives access to all classes that need to know about conversations in the game. I am also implementing this in such a way that only conversations required for the current map are loaded into memory. Each time the map changes the conversations in the manager change.

Singleton classes should not be inherited from so I marked the class as sealed. Inside I create a private, static readonly **ConversationManager** object called instance. This is done with the private constructor that takes no parameters, and that constructor will be used for deserializing conversations. The only other field in this class is a **Dictionary<string, Conversation>**. The key is the conversation name and the value is the associated conversation.

The two properties have public getters and the one for the conversations has a private setter. I marked that one with an attribute that will have the **IntermediateSerializer** serialize and deserialize it. The static property gives other classes to access the singleton instance.

I've also add a few methods to this class. The first, **AddConversation** accepts a name parameter and a **Conversation** parameter. It checks to see if a key with that name already exists. If it does not exist it adds the conversation.

The **GetConversation** method accepts as a parameter the conversation to be retrieved. It checks to

see if that key exists and if it does it returns the conversation. If it does not exist it returns null. Instead of returning null you could raise an exception and handle it in your code.

The next method is **ContainsConversation** that accepts the name of the conversation to be checked and returns true or false.

The last method, **ClearConversations**, is used to remove all conversations from the conversation manager. It should only be called when switching maps.

If you try to build and run now there will be an error. That is because the **NonPlayerCharacter** class references the old **ConversationClasses** namespace that we deleted. To fix that open the **NonPlayerCharacter** class and replace the **RpgLibrary.ConverstaionClasses** using statement with **MGRpgLibrary.ConverstaionComponents**. At this point you should be able to build and run your game as normal.

Now we've got a set of classes that define and manage conversations in the game. The next step will be to be able to display them in the game. For that I will create a new game state. In the **EyesOfTheDragon** project right click the **GameScreens** folder, select **Add** and then **Class**. Name this new class **ConversationScreen**. Here is the code for that screen, to begin with, I will be filling it out more in future tutorials.

```
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.CharacterClasses;
using MGRpgLibrary.ConversationComponents;

namespace EyesOfTheDragon.GameScreens
{
    public class ConversationScreen : BaseGameState
    {
        private ConversationManager conversations = ConversationManager.Instance;
        private Conversation conversation;
        private Player player;
        private NonPlayerCharacter npc;

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

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

        protected override void LoadContent()
        {
```

```
            base.LoadContent();
        }

        public override void Update(GameTime gameTime)
        {
            conversation.Update(gameTime);

            base.Update(gameTime);
        }

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

            GameRef.SpriteBatch.Begin();

            conversation.Draw(gameTime, GameRef.SpriteBatch, null);

            GameRef.SpriteBatch.End();
        }

        public void SetConversation(Player player, NonPlayerCharacter npc, string
            conversation)
        {
            this.player = player;
            this.npc = npc;
            this.conversation = conversations.GetConversation(conversation);
        }

        public void StartConversation()
        {
            conversation.StartConversation();
        }
    }
}
```

I had to add several using statements to bring classes in other namespaces into scope. They are for the MonoGane Framework and our libraries.

I added in a few private fields. The first, **conversations**, is the instance of **ConversationManager** that is the singleton class. Next is a field **conversation** that represents the current conversation. **Player** which is our player object. The last is a **NonPlayerCharacter** and is the character that the player is currently talking to.

Currently the constructor just has the two base parameters that are required from the class that we inherit all game states from, **BaseGameState**. Currently the **Initialize** and **LoadContent** methods are empty but have been included because they are standard MonoGame Framework methods for **DrawableGameComponents**.

The **Update** method calls the **Update** method of the current **conversation**. It then calls **base.Update** to update other game components. The **Draw** method calls **SpriteBatch.Begin** to start rendering 2D objects, calls the **Draw** method of the current **conversation** and the calls **SpriteBatch.End** to stop rendering.

The next method is **SetConversation**. This method accepts a **Player** parameter, a **NonPlayerCharacter** parameter and a string parameter. It sets the **Player** and **NonPlayerCharacter** fields using the parameters. It then uses the **ConversationManager** instance to retrieve the conversation with the name passed in.

So, the plumbing is now in place for the player to have conversations with non-player characters in the game. In the next tutorial I will add a non-player character to the game with a conversation attached to it so that we can see these changes in progress. I'm going to wrap the tutorial here because I'd like to try and keep the tutorials to a reasonable length so that you don't have too much to digest at once. 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