A Summoner's Tale – MonoGame Tutorial Series

Chapter 8

Conversations

This tutorial series is about creating a Pokemon style game with the MonoGame Framework called A Summoner's Tale. The tutorials will make more sense if you read them in order as each tutorial builds on the previous tutorials. You can find the list of tutorials on my web site: A Summoner's Tale. The source code for each tutorial will be available as well. I will be using Visual Studio 2013 Premium for the series. The code should compile on the 2013 Express version and Visual Studio 2015 versions as well.

I want to mention though that the series is released as Creative Commons 3.0 Attribution. It means that you are free to use any of the code or graphics in your own game, even for commercial use, with attribution. Just add a link to my site, http://gameprogrammingadventures.org, and credit to Jamie McMahon.

This tutorial is about attaching conversations to the characters that the player will with. At a very high level a conversation is a tree of nodes that can be traversed in different ways depending on the player's choices. In the game I called a node a GameScene. The scene contains the text to be displayed to the player and one or more SceneOptions. A SceneOption has a SceneAction which determines what action is taken if the player selects that action. A full conversation is made up a number of GameScenes.

So, let's get started. First, right click the Avatars project, select Add and then New Folder. Name this new folder ConversationComponents. Now right click the ConversationComponents folder, select Add and then class name this new class SceneOption. Here is the code for that class.

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

namespace Avatars.ConversationComponents
{
    public enum ActionType
    {
        Talk,
        End,
        Change,
        Quest,
        Buy,
        Sell,
        GiveItems,
        GiveKey,
    }
```

```
    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;
        }
    }
}
```

I added an enumeration called ActionType that defines what action to take when the player selects that option. Talk moves the conversation to another scene in the same conversation. End ends the conversation. Change changes the current conversation to another conversation. Quest gives the player a quest if the node is selected. The Buy and Sell options were added for speaking with shopkeepers. The GiveItem and GiveKey give an item or a key to the player. Next up is a really basic class, SceneAction. This holds the action to take and any parameters that will be used based on the action chosen.

The actual SceneOption class holds the details for the option. For that there are three private member variables. The optionText field holds what is drawn on the screen. Then optionScene is what scene to change to based on the option. Finally there is a SceneAction field that defines the action taken with any parameters. There are three public properties that expose these values to other classes. There is also a private constructor that takes no parameters that will be used for writing conversations and reading them back in. There is a second constructor that takes three parameters which are the text displayed, the scene being transitioned to and the action.

With the SceneOption in place I can now create the GameScene class. Right click the

ConversationComponents folder, select Add and then Class. Name this new class GameScene. This is the code for that class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Avatars.Components;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Input;

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

        protected Game game;
        protected string text;
        private List<SceneOption> options;
        private int selectedIndex;
        private Color highLight;
        private Color normal;
        private Vector2 textPosition;
        private static Texture2D selected;
        private bool isMouseOver;

        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; }
        }

        public bool IsMouseOver
        {
            get { return isMouseOver; }
        }

        [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)
        {
            this.text = text;
            this.options = options;
            textPosition = Vector2.Zero;
        }

        public GameScene(Game game, string text, List<SceneOption> options)
        {
            this.game = game;

            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, SpriteFont font)
```

```
        {
            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 = 0;
                }
            }

            this.text = sb.ToString();
        }

        public void Initialize()
        {
        }

        public void Update(GameTime gameTime, PlayerIndex index)
        {
            if (Xin.CheckKeyReleased(Keys.Down))
            {
                selectedIndex--;
                if (selectedIndex < 0)
                    selectedIndex = options.Count - 1;
            }
            else if (Xin.CheckKeyReleased(Keys.Down))
            {
                selectedIndex++;
                if (selectedIndex > options.Count - 1)
                    selectedIndex = 0;
            }
        }

        public void Draw(GameTime gameTime, SpriteBatch spriteBatch, Texture2D background,
SpriteFont font)
        {
            Vector2 selectedPosition = new Vector2();
            Color myColor;

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

            if (textPosition == Vector2.Zero)
```

```
                SetText(text, font);

            if (background != null)
                spriteBatch.Draw(background, Vector2.Zero, Color.White);

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

            Vector2 position = menuPosition;

            Rectangle optionRect = new Rectangle(0, (int)position.Y, 1280,
font.LineSpacing);
            isMouseOver = false;

            for (int i = 0; i < options.Count; i++)
            {
                if (optionRect.Contains(Xin.MouseState.Position))
                {
                    selectedIndex = i;
                    isMouseOver = true;
                }

                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;
                optionRect.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 is text and it is used 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 selectedIndex 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. This one is static and will be shared by all instances of GameScene. The next field I included is isMouse over and will be used to test if the mouse is over an SceneOption. The last member variable is menuPosition and controls where the scene options are drawn.

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. The third is used in the game when creating scenes on the fly.

I added a method called SetText and it takes as parameters text and font. 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 accepted a Texture2D parameter called portrait that represented the portrait of the character the player is speaking to. I'm not implementing that for a while yet but I want it available if needed. 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 an will be used in the future.

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 background image and a SpriteFont to draw the text with. 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 not null I draw the background.

After drawing the background I draw the speaker's text in white. You can include a property in the game scene for what color to draw the speaker's text in rather than hard coding it.

There is then a Vector2 local variable that I assign the position of the scene options to be drawn. I then create a rectangle based on the position that is the width of the screen. I then loop over all of the options for the player to choose from as a reply to the current scene. Inside that loop I check if the mouse is included in the rectangle. If it is I set the selectedIndex member variable to the current loop index. Next I check to see if the current loop index is the selectedIndex. If it is I draw the selected item texture to the left of that option and I set the local color variable to the highlight color. If it is not then the local color variable is set to the base option color. I then draw the scene 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.

Now I'm going to add in the class that represents a conversation. Right click the ConversationComponents folder, select Add and then Class. Name this new class Conversation. Here 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;

namespace Avatars.ConversationComponents
{
    public class Conversation
    {
        #region Field Region

        private string name;
        private string firstScene;
        private string currentScene;
        private Dictionary<string, GameScene> scenes;
        private string bsckgroundName;
        private Texture2D background;
        private string fontName;
        private SpriteFont spriteFont;

        #endregion

        #region Property Region

        public string Name
        {
            get { return name; }
        }

        public string FirstScene
        {
            get { return firstScene; }
        }

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

        public Dictionary<string, GameScene> Scenes
```

```
        {
            get { return scenes; }
        }

        public Texture2D Background
        {
            get { return background; }
        }

        public SpriteFont SpriteFont
        {
            get { return spriteFont; }
        }

        public string BackgroundName
        {
            get { return backgroundName; }
            set { backgroundName = value; }
        }

        public string FontName
        {
            get { return fontName; }
            set { fontName = value; }
        }

        #endregion

        #region Constructor Region

        public Conversation(string name, string firstScene, Texture2D background, SpriteFont
font)
        {
            this.scenes = new Dictionary<string, GameScene>();
            this.name = name;
            this.firstScene = firstScene;
            this.background = background;
            this.spriteFont = font;
        }

        #endregion

        #region Method Region

        public void Update(GameTime gameTime)
        {
            CurrentScene.Update(gameTime, PlayerIndex.One);
        }

        public void Draw(GameTime gameTime, SpriteBatch spriteBatch)
        {
            CurrentScene.Draw(gameTime, spriteBatch, background, spriteFont);
        }

        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
    }
}
```

I added in member variables for the name of the conversation, the first scene of the conversation, the current scene of the conversation, a dictionary of scenes, a texture for the conversation and a font for the scene. I also include member variables for the name of the background for the conversation and the name of the font the text will be drawn with.

Next there are properties that expose the member variables. The ones for the name fields are both getters and setters. Most of the others are simple getters only. The one that is more than just a simple getter is the one that returns the current scene. Rather than returning the key for the scene I return the scene using the key.

The constructor for this class takes four parameters: name, firstScene, background and font. They represent the name of the conversation, the first scene to be displayed, the background for the conversation and the font the conversation is drawn with. I just set the fields with the parameters that are passed in.

The scene needs to have its Update method called so I included an Update method in this class. It calls the Update method of the current scene for the conversation. The conversation needs to be drawn so there is a Draw method for the conversation. It just calls the Draw method of the current scene.

You can just use the raw dictionary to add and retrieve scenes but I included a method for adding a scene and a method for getting a scene. The reason being is that I can validate the values and prevent the game from crashing if something unexpected is passed in.

The last two methods on this class are StartConversation and ChangeScene. StartConversation sets the currentScene member variable to the firstScene member variable. ChangeScene changes the scene to the parameter that is passed in.

The last thing that I'm going to add in this tutorial is a class that manages the conversations in the game. Right click the ConversationComponents folder, select Add and then Class. Name this new class ConversationManager. Here is the code for that class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using System.Xml;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
```

```
namespace Avatars.ConversationComponents
{
    public class ConversationManager
    {
        #region Field Region

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

        #endregion

        #region Property Region

        public static Dictionary<string, Conversation> ConversationList
        {
            get { return conversationList; }
        }

        #endregion

        #region Constructor Region

        public ConversationManager()
        {
        }

        #endregion

        #region Method Region
        public static void AddConversation(string name, Conversation conversation)
        {
            if (!conversationList.ContainsKey(name))
                conversationList.Add(name, conversation);
        }

        public static Conversation GetConversation(string name)
        {
            if (conversationList.ContainsKey(name))
                return conversationList[name];

            return null;
        }

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

        public static void ToFile(string fileName)
        {
            XmlDocument xmlDoc = new XmlDocument();

            XmlElement root = xmlDoc.CreateElement("Conversations");
            xmlDoc.AppendChild(root);

            foreach (string s in ConversationManager.ConversationList.Keys)
            {
                Conversation c = ConversationManager.GetConversation(s);

                XmlElement conversation = xmlDoc.CreateElement("Conversation");

                XmlAttribute name = xmlDoc.CreateAttribute("Name");
                name.Value = s;
                conversation.Attributes.Append(name);
```

```
            XmlAttribute firstScene = xmlDoc.CreateAttribute("FirstScene");
            firstScene.Value = c.FirstScene;
            conversation.Attributes.Append(firstScene);

            XmlAttribute backgroundName = xmlDoc.CreateAttribute("BackgroundName");
            backgroundName.Value = c.BackgroundName;
            conversation.Attributes.Append(backgroundName);

            XmlAttribute fontName = xmlDoc.CreateAttribute("FontName");
            fontName.Value = c.FontName;
            conversation.Attributes.Append(fontName);

            foreach (string sc in c.Scenes.Keys)
            {
                GameScene g = c.Scenes[sc];

                XmlElement scene = xmlDoc.CreateElement("GameScene");

                XmlAttribute sceneName = xmlDoc.CreateAttribute("Name");
                sceneName.Value = sc;

                scene.Attributes.Append(sceneName);

                XmlElement text = xmlDoc.CreateElement("Text");
                text.InnerText = c.Scenes[sc].Text;

                foreach (SceneOption option in g.Options)
                {
                    XmlElement sceneOption = xmlDoc.CreateElement("GameSceneOption");

                    XmlAttribute oText = xmlDoc.CreateAttribute("Text");
                    oText.Value = option.OptionText;
                    sceneOption.Attributes.Append(oText);

                    XmlAttribute oOption = xmlDoc.CreateAttribute("Option");
                    oOption.Value = option.OptionScene;
                    sceneOption.Attributes.Append(oOption);

                    XmlAttribute oAction = xmlDoc.CreateAttribute("Action");
                    oAction.Value = option.OptionAction.ToString();
                    sceneOption.Attributes.Append(oAction);

                    XmlAttribute oParam = xmlDoc.CreateAttribute("Parameter");
                    oParam.Value = option.OptionAction.Parameter;

                    scene.AppendChild(sceneOption);
                }

                conversation.AppendChild(scene);
            }

            root.AppendChild(conversation);
        }

        XmlWriterSettings settings = new XmlWriterSettings();
        settings.Indent = true;
        settings.Encoding = Encoding.UTF8;

        FileStream stream = new FileStream(fileName, FileMode.Create, FileAccess.Write);
        XmlWriter writer = XmlWriter.Create(stream, settings);
        xmlDoc.Save(writer);
    }

    public static void FromFile(string fileName, Game gameRef, bool editor = false)
    {
        XmlDocument xmlDoc = new XmlDocument();
```

```
            try
            {
                xmlDoc.Load(fileName);

                XmlNode root = xmlDoc.FirstChild;

                if (root.Name == "xml")
                    root = root.NextSibling;

                if (root.Name != "Conversations")
                    throw new Exception("Invalid conversation file!");

                foreach (XmlNode node in root.ChildNodes)
                {
                    if (node.Name == "#comment")
                        continue;

                    if (node.Name != "Conversation")
                        throw new Exception("Invalid conversation file!");

                    string conversationName = node.Attributes["Name"].Value;
                    string firstScene = node.Attributes["FirstScene"].Value;
                    string backgroundName = node.Attributes["BackgroundName"].Value;
                    string fontName = node.Attributes["FontName"].Value;

                    Texture2D background = gameRef.Content.Load<Texture2D>(@"Backgrounds\" +
backgroundName);
                    SpriteFont font = gameRef.Content.Load<SpriteFont>(@"Fonts\" +
fontName);

                    Conversation conversation = new Conversation(conversationName,
firstScene, background, font);
                    conversation.BackgroundName = backgroundName;
                    conversation.FontName = fontName;

                    foreach (XmlNode sceneNode in node.ChildNodes)
                    {
                        string text = "";
                        string optionText = "";
                        string optionScene = "";
                        string optionAction = "";
                        string optionParam = "";
                        string sceneName = "";

                        if (sceneNode.Name != "GameScene")
                            throw new Exception("Invalid conversation file!");

                        sceneName = sceneNode.Attributes["Name"].Value;

                        List<SceneOption> sceneOptions = new List<SceneOption>();

                        foreach (XmlNode innerNode in sceneNode.ChildNodes)
                        {
                            if (innerNode.Name == "Text")
                                text = innerNode.InnerText;

                            if (innerNode.Name == "GameSceneOption")
                            {
                                optionText = innerNode.Attributes["Text"].Value;
                                optionScene = innerNode.Attributes["Option"].Value;
                                optionAction = innerNode.Attributes["Action"].Value;
                                optionParam = innerNode.Attributes["Parameter"].Value;

                                SceneAction action = new SceneAction();
                                action.Parameter = optionParam;
```

```
                                action.Action = (ActionType)Enum.Parse(typeof(ActionType),
optionAction);

                                SceneOption option = new SceneOption(optionText,
optionScene, action);
                                sceneOptions.Add(option);
                            }
                        }

                        GameScene scene = null;

                        if (editor)
                            scene = new GameScene(text, sceneOptions);
                        else
                            scene = new GameScene(gameRef, text, sceneOptions);


                        conversation.AddScene(sceneName, scene);
                    }

                    conversationList.Add(conversationName, conversation);
                }
            }
            catch
            {
            }
            finally
            {
                xmlDoc = null;
            }
        }

        #endregion

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

All of the members, other than the constructor, are all static. I did this because I am sharing this component with other classes. It is not really "best practices" to do this though. In this case it is for a demo and will suit our purposes. You would probably want to update the manager to be align with best object-oriented programming practices in a production game.

The only member variable holds the conversations in the manager. There is also a property that exposes the member variable. The constructor takes no parameters and does no actions. It was included for use in the future.

There are a number of static methods next. The first is AddConversation is and called to add a conversation to the manager. You can also use the static property and add the conversation that way. This just adds a little error checking to make sure a conversation with the given key does not already exists. This was added mostly for the editor so that if you try to add the same conversation twice you will get an error message.

The next static method is GetConversation and is used to retrieve a conversation from the manager. This could also be done with the property as well. I added it because it first checks to see that the conversation exists before trying to retrieve it. This would prevent a crash if the conversation was not present. You would have to handle the null value that is returned or that could crash the game.

ContainsConverstaion just checks to see if the give key is present in the dictionary and returns that back. This was also added to try and have better error checking before adding or retrieving a conversation.

Next is the method ToFile. This method is used to write the conversation manager to an XML document. I went this route to show one of the few ways that I use to read and write content without using the content manager. The method takes as a parameter the name of the file to write the conversation manager to. Creating XML documents through code is process of creating a root node and then appending children to that node. Those children can also have child nodes under them.

The first thing to do is to create a new XmlDocument. To that I add the root element for the document, Conversations. Next that is appended to the document. The next step is to loop over all of the conversations and create a node for them to append them to the root node of the document.

The first step is to get the conversation using the key from the foreach loop that iterates over the collection of conversations. Next I create a new element/node. I then create attributes for the element for the name, firstScene, backgroundName and fontName members. There is then another foreach loop to go over the scene collection.

Inside that loop I first get the scene using the key. I then create an element for that scene. I then create an attribute for the name of the scene and the text for the scene.

There is then another foreach loop to iterate over the options for that scene. I then create an element for that option. I then create attributes for the three scene properties, text, option and parameter. I then append that element to the scene element. Next the scene is appended to the conversation. The conversation is then appended to the root.

Now that the document is created it needs to be written out. I use the XmlWriter class for that. It requires and XmlWriterSettings parameter so I create that with standard XML attributes, indentation and encoding as UTF8. Next I create a FileStream that is required using Create and Write options. The Create option will create a new file if the file does not exist or overwrite the existing file if does exist. I then create the writer and write the file

The next static method is FromFile that will parse the XML document that was written out earlier. The first thing to do is to create an XmlDocument object. I then do everything inside of a try-catch-finally block to prevent crashes.

The XmlDocument class has a method Load that will load the document into that object. The object can then be parsed and the data be pulled out. I grab the root node of the document using the FirstChild property. If the name is xml then it is the header and we need to go down a level so I set the root node to its next sibling. I then compare that to Conversations. If it is not Conversations then the file is not in the format that we are expecting and I throw an exception.

I then iterate over all of the child nodes using the ChildNodes collection. I check to see if the name is comment, if it is I move onto the next node. Next I check to see if it is Conversation. If it is not conversation the document is not in the right format so I throw an exception. I then grab the name, first scene, background and font attributes. Next up I use the game object passed in to load the background texture and the sprite font. I then create a conversation using the attributes and assign the background and font properties.

Since that node should have child nodes I have a foreach loop that will iterate over them. Inside that loop I have some local variables to hold values that are required for scenes and scene options. If the name of the node is not GameScene I throw an exception. I then grab the Name attribute and assign it to the sceneName variable. Next I create a list of scene options that is required for creating the game scene. I then iterate over the child nodes. If the name is Text then I set the text variable to the inner text of that node. If it is GameSceneOption I assign the local variables to the attributes of the node. Afterwards I create an action and then the option using the text and the action. I then create a GameScene object and initialize it, so the compiler will not complain it has not been initialized. If we are in the editor I use the first constructor, otherwise is use the second constructor. I then add the scene to the conversation. The conversation is then added to the list of conversations.

I don't do anything in the catch but it is a good idea to handle it in some way. I will cover that in another tutorial. In the finally, which is always called, I set the xmlDoc to null.

There is one other static method, ClearConversations, that creates a new list of conversations. This could tax the garbage collector a bit so it might be best to use the Clear method to remove the elements instead.

I'm going to end the tutorial here as it is a lot to digest in one sitting. In the next tutorial I will cover updating the game to allow for conversations with other characters in the game. Please stay tuned for the next tutorial in this series. If you don't want to have to keep visiting the site to check for new tutorials you can sign up for my newsletter on the site and get a weekly status update of all the news from Game Programming Adventures. You can also follow my tutorials on Twitter at https://twitter.com/GPAAdmi77640534.

I wish you the best in your MonoGame Programming Adventures!
Jamie McMahon