A Summoner's Tale
Chapter F
Look Who's Talking

Welcome to my Summoner's Tale tutorial series on creating a Pokémon-inspired game with MonoGame. I'm writing these tutorials for the MonoGame 3.8.1 framework using Visual Studio 2022. The code should work on previous versions of MonoGame and Visual Studio. I plan on creating the editors on macOS and Windows. I'm unfamiliar with Linux, so a few projects may not be done for that platform.

The tutorials will make more sense if they are read in order. You can find the list of tutorials on my web blog, A Summoner's Tale page. In addition to the PDFs, I will make the code for each tutorial available on GitHub here: https://github.com/Synammon/summoners-tale. It will also be included on the page that links to the tutorials.

We have a character on the map, and they are pretty but useless. They don't actually do anything. It would be better if they could talk. For that reason, we will be adding conversations in this tutorial.

So, let's begin. I will be using a slightly modified version of my conversation classes. The only difference is going to be the way the rendering is done. I will render the conversation in a bubble at the bottom of the screen with the map visible rather than drawing to the right of a portrait, as we don't have portraits.

Conversations have two main components. The first is the scene, the text that is rendered. There are also options. Options are the choices the player can choose from. First, I will add actions. Right-click the Psilibrary project in the Solution Explorer, Select Add and then New Folder ConversationComponents. Now, right-click the new 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;

namespace Psilibrary.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;
        }
    }
}
```

There is an enumeration here of the various actions an option can have. First, there is Talk to move to the next scene. Next, there is End to stop talking to the character. Change will change to a new conversation branch. After that, is Quest to accept or turn in a quest. Next, buy to start buying items from the character. Finally, Sell to start selling items to the character.

There is a short class that is an action. An action has an ActionType and a Parameter. Parameter. I did this to make it a bit cleaner when defining scenes and serializing/deserializing scenes. SceneOption contains three member variables, 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. What exactly is the IntermediateSerializer? Well, it is used to serialize content to XML and read it back in again. It is the mechanism we will use in editors to save and load our content.

Next are some 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 classes that make up a scene that is made up of options. There are two classes. One is a class that goes into the Psilibrary that holds the data. The other goes into SummonersTale and holds the functionality. Right-click the ConversationComponents folder in Psilibrary, select Add and then Class. Name this new class GameSceneData. 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 Psilibrary.ConversationComponents
{
    public partial class GameSceneData
    {
        #region Field Region

        protected string text;
        protected List<SceneOption> options;

        #endregion

        #region Property Region


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

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

        #endregion

        #region Constructor Region
        #endregion

        #region Method Region
        #endregion
    }
}
```

A simple class that has the text for the scene and a list of options. Everything is public or protected because we will be inheriting from this in the other library. Speaking of, it is time to implement it. Right-click the SummonersTale project, select Add and then New Folder. Name this new folder ConversationComponents. Now, right-click this new folder, select Add and then Class. Name this new class GameScene. Here is the code for that class. Also, I'm really sorry about the wall of code.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Input;
using SummonersTale;
using SummonersTale.Forms;

namespace Psilibrary.ConversationComponents
{
    public class ButtonGroup
    {
        public Button Button { get; set; }
        public string Text { get; set; }
        public SceneAction Action { get; set; }
    }

    public class GameScene : GameSceneData
    {
        public event EventHandler<SelectedIndexEventArgs> ItemSelected;

        #region Field Region

        protected Game game;
        protected Texture2D border;
        protected Texture2D texture;
        protected SpriteFont font;
        protected Texture2D button;
        protected int selectedIndex;
        protected Color highLight;
```

```
        protected Color normal;
        protected Vector2 textPosition;
        protected static Texture2D selected;
        protected Vector2 menuPosition = new(50, 475);
        protected bool isOver;
        protected List<ButtonGroup> buttons = new();

        #endregion

        #region Property Region

        public bool IsOver { get; private set; }

        [ContentSerializerIgnore]
        protected Vector2 Size { get; set; }

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

        [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()
    {
    }

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

        LoadContent();

        Size = MeasureText(out this.text, text);
        Size += new Vector2(0, (options.Count + 5) * font.LineSpacing);

        this.options = new List<SceneOption>();

        int index = 0;

        foreach (var option in options)
        {
            this.options.Add(option);
            ButtonGroup buttonGroup = new()
            {
                Button = new(button, ButtonRole.Menu)
                {
                    Size = new(font.LineSpacing, font.LineSpacing),
                    Text = "",
                    Index = index++,
                },
                Text = option.OptionText,
            };

            buttonGroup.Button.Click += Button_Click;
            buttons.Add(buttonGroup);
        }

        highLight = Color.Red;
        normal = Color.Black;
    }

    private void Button_Click(object sender, EventArgs e)
    {
        Button btn = (Button)sender;

        if (btn.Index.HasValue)
        {
            SelectedIndexEventArgs item = new() { Index= btn.Index.Value };
            ItemSelected?.Invoke(sender, item);
        }
    }
```

```
        #endregion

        #region Method Region

        public Vector2 MeasureText(out string parsedText, string text)
        {
            Vector2 measured = Vector2.Zero;
            StringBuilder sb = new();
            string textOut = string.Empty;

            float currentLength = 0f;

            if (font == null)
            {
                parsedText = string.Empty;
                return measured;
            }

            measured.Y = font.LineSpacing;

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

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

                if (currentLength + size.X < Settings.BaseWidth - 100f)
                {
                    sb.Append(s);
                    sb.Append(' ');

                    currentLength += size.X + font.MeasureString(" ").X;
                }
                else
                {
                    measured.Y += font.LineSpacing;
                    sb.Append('\n');
                    sb.Append(s);
                    sb.Append(' ');

                    currentLength = size.X + font.MeasureString(" ").X;
                }
            }

            parsedText = sb.ToString();

            measured.X = Settings.BaseWidth;
            return measured;
        }

        protected void LoadContent()
        {
            border = new(game.GraphicsDevice, Settings.BaseWidth, 100);
            texture = new(game.GraphicsDevice, Settings.BaseWidth, 100);

            border.Fill(Color.Blue);
            texture.Fill(Color.White);
```

```
            font = game.Content.Load<SpriteFont>(@"Fonts/scenefont");
            button = game.Content.Load<Texture2D>(@"GUI/g21245");
        }

        public virtual void Update(GameTime gameTime)
        {
            foreach (ButtonGroup group in buttons)
            {
                group.Button.Update(gameTime);
            }
        }

        public virtual void Draw(GameTime gameTime, SpriteBatch spriteBatch,
Texture2D portrait = null)
        {
            Rectangle portraitRect = new(25, 25, 425, 425);
            string parsedText = string.Empty;

            if (texture == null) LoadContent();

            Size = MeasureText(out parsedText, text);
            Size += new Vector2(0, buttons.Count * (font.LineSpacing + 20) + 50);

            textPosition = new(25, Settings.BaseHeight - Size.Y);

            Rectangle dest = new(
                0,
                (int)textPosition.Y,
                Settings.BaseWidth,
                (int)Size.Y);

            spriteBatch.Draw(border, dest, Color.White);
            spriteBatch.Draw(texture, dest.Grow(-2), Color.White);

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

            spriteBatch.DrawString(font,
                parsedText,
                textPosition,
                Color.Black);

            Vector2 location = new(25, Settings.BaseHeight -
buttons[0].Button.Size.Y - 20);

            for (int i = buttons.Count - 1; i >= 0; i--)
            {
                buttons[i].Button.Position = location;

                buttons[i].Button.Draw(spriteBatch);

                spriteBatch.DrawString(font,
                    options[i].OptionText,
                    location + new Vector2(40, 0),
                    Color.Black);

                location.Y -= buttons[0].Button.Size.Y + 20;
            }
```

```
        }
        #endregion
    }
}
```

There is a lot going on in this class. Actually, it is two classes. The first is a logical construct that bundles a button, action and text together. To select an option, you click or tap the button to the left. I did that to make things seamless between our target platforms.

Now, for the second class. There is an event in this class that will fire when a scene option is selected. Now, I'll tackle member variables. There is a Game type field that is the reference to the game. It is used for loading content using the content manager. Following, the game field is fields for the background of the state. There is one for the border and one for the background. The next member variable, font, is the font used for drawing the scene text. Next is button, and is the texture for the button. The next five fields are currently unused but are included because they were useful in the past. 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<ButtonGroup> which is the options for the scene.

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 would be serialized using the IntermediateSerializer. If it was in Psilibrary. Mostly, they are no longer relevant. because I didn't want some of the members serialized so that they are set at runtime rather than at build time.

Next up are the two 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 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. Where this class deviates from previous tutorials I was is when it loops over the options passed to the constructor. It adds the option to the list of options. It then creates a new ButtonGroup, creating a Button using the button texture and setting its size to the LineSpacing property of the font, both height and width. To prevent possible null value exceptions, the Text property is set to the empty string, and the action to the scene action. The text for the button group is set to the text property of the option group. Finally, a new property for buttons, Index, is set to a local variable. The button is added to the list of buttons, and it is assigned to an event handler. Wait a minute! You're assigning multiple objects to the same event handler? You're a crazy woman. Actually, it works because we are passed a sender object. We can grab a unique property and figure out what the index is.

In the event handler for the click event of the button, I cast the sender object to a Button. Technically, this is dangerous because we have no idea what the sender may be. You could use pattern matching instead. I will leave that to you as an exercise. Since the property is nullable, I check to see if it has a value. If it does, I create event arguments that are the index of the button. I then invoke the event handler if it is subscribed to.

So, we need to add the property to the Button class. Add the following line of code to the Button class with the other properties. As you can see, it is nullable, so it doesn't have to have a value associated with it.

```
public int? Index { get; set; } = null;
```

There were initially two similar methods, MeasureText and SetText. In fact, they were similar enough that I merged them into one method, MeasureText. What it does is calculate how much room they take to render the text and convert the text from one long string into a string that is escaped so that it renders on multiple lines. First, I create a zero vector to hold the size. In actuality, the width will never be more than the base width of the game. It is more efficient to use a string build than a string, so there is a string builder, and the current length is set to zero. If there isn't a font, I exit the method. Since there will always be one line, I set the height to the line spacing property of the font. I then split the string into parts on the space character.

Next, I loop over the parts. Inside that loop, I measure the size of the part. If the current length plus the width of the part is less than the base width minus one hundred pixels, I add it with a space to the string builder. I'm writing this thinking I could have done this better. Anyway, I then update the current width to be the current width to the current width plus the width of the part plus a space. If it is greater than the base width of the window minus one hundred pixels, I increase the current height by the line spacing of the font. I append a new line and a carriage return, then the string and a space. I then set the current length to the width of the part. Finally, I returned the size.

In the LoadContent method, I create the border and foreground textures as I did in the other tutorials where I create textures. I then fill the border with blue and the foreground with white. I then load the font that I will add now. Open the MGCB Editor, right-click the Fonts folder, select Add and then New Item. From the list that comes up, select Sprite Font Description. Save and close the MGCB Editor. Now, from the menu, select File -> Open. Navigate to the font that we just added. Change the size to 16. I then load a button texture with a caret pointing right.

The Update and Draw methods are virtual because we will be overriding them when we get to combat. In the Update method, I loop over the buttons and call the button's Update method to call their HandleInput methods so that we can capture clicks or taps.

In the Draw method, there is an optional parameter portrait that we won't be using. It is included in the event that we have portraits for characters in the future. I then set the position of the portrait. There is a string the holds the text for the conversation that has been parsed to fit the width of the window. If the texture is null, I call the LoadContent method. Ideally, you do not want to be constantly parsing the text. However, the text can be dynamic, so we have no choice. I call the MeasureText method to parse and measure the height of the text. The method takes as parameters an out parameter for the parsed text and the text to be parsed. It returns a Vector2 that is the size of the text. I then add zero to the X property, but to the Y property I add

the line spacing plus twenty pixels for each scene option, then an extra fifty pixels. The text position is twenty-five pixels for the X property, and the Y property is the height of the window minus the size of the text.

Next, I create a destination rectangle for the border around the background. I draw the border, then I draw the background two pixels shorter. If the portrait is not null, I draw that. Next, I draw the text for the scene.

Now, I create a Vector2, which is the position of the lowest button. I then loop backwards over the buttons. I set the Position property of the button to the location. I draw the button, then draw the text beside it. Then, I move the location up twenty pixels and the size of a button.

Wow, that was a monster. It is one of the most complicated classes that I will be implementing in this tutorial. Now that we have scenes, we can focus on full conversations. This will also be a two-class deal, a data and function. Let's start with the data half. In the Psilibrary project, right-click the ConversationComponents folder, select Add and then Class. Name this new class ConversationData. 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.Content;

namespace Psilibrary.ConversationComponents
{
    public class ConversationData
    {
        #region Field Region

        protected string name;
        protected string firstScene;
        protected Dictionary<string, GameSceneData> scenes;

        #endregion

        #region Property Region

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

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

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

        #endregion

        #region Constructor Region

        protected ConversationData()
        {
        }

        public ConversationData(string name, string firstScene)
        {
            this.scenes = new();
            this.name = name;
            this.firstScene = firstScene;
        }

        #endregion

        #region Method Region
        #endregion
    }
}
```

A very simple class. It is comprised of three fields and properties. There is a name field, what scene is the first scene, and the scenes. There are then properties to expose the fields. Everything is either protected or public, so we have access to it in the conversation class. There is a parameterless constructor that is required for serialization.

That brings us to the conversation class. Right-click the ConversationComponents in the SummonersTale project, 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;
using Microsoft.Xna.Framework.Content;

namespace Psilibrary.ConversationComponents
{
    public class Conversation : ConversationData
    {
        #region Field Region

        private string currentScene;
        private readonly Dictionary<string, GameScene> scenes = new();

        #endregion
```

```
#region Property Region

public Dictionary<string, GameScene> Scenes
{
    get { return scenes; }
}

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

#endregion

#region Constructor Region

public Conversation()
    : base()
{
    GameScenes = new();
}

#endregion

#region Method Region

public void Update(GameTime gameTime)
{
    if (Scenes.ContainsKey(currentScene))
    {
        Scenes[currentScene].Update(gameTime);
    }
}

public void Draw(GameTime gameTime, SpriteBatch spriteBatch)
{
    if (Scenes.ContainsKey(currentScene))
    {
        Scenes[currentScene].Draw(gameTime, spriteBatch);
    }
}

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

public GameSceneData GetScene(string sceneName)
{
    if (GameScenes.ContainsKey(sceneName))
        return GameScenes[sceneName];

    return null;
```

```
        }

        public void StartConversation()
        {
            currentScene = FirstScene;
        }

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

        #endregion
    }
}
```

Not a complicated class like GameScene is. It inherits from ConversationData, so we will have access to all of its public and protected members. It has two fields, the current scene and a dictionary of scenes. Both the current scene and keys for the dictionary are strings. There are properties to expose the dictionary and current scene, not the key. There is a single constructor that initializes the GameScenes property.

In the Update method, I check to see if the dictionary contains the current key. If it does, I call the Update method of the scene. I do something similar for the Draw method.

You can add scenes to the dictionary using the property. The preferred method is to use the AddScene method. It won't throw an exception if there is an existing key like the property would. If it does not exist, I add a scene data to the GameScenes property and game scene to the Scenes property. The GetScene method gets the game scene data for the scene passed in if it exists. Otherwise, it returns null.

There are two other methods in this class. The first is StartConversation which resets the current scene to the first scene. The other method is ChangeScene which changes the current scene. It would be a good idea to check in both methods that the keys exist in the dictionary before setting the values. I will leave that as an exercise for you.

Still with me? Good, we are reaching the end of the race with one lap to go. There is more that I will be adding in the future. For now, I just want to get these core components working. They won't require much in the way of maintenance. The only heavy lifting left is the conversation state. Right-click the StateManagement folder in the SummonersTale project, select Add, and then Class. Name this new class ConversationState. This is the code for that class.

```
using Microsoft.Xna.Framework;
using Psilibrary.ConversationComponents;
using SummonersTale.Forms;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
namespace SummonersTale.StateManagement
{
    public interface IConversationState
    {
        GameState GameState { get; }
    }

    public class ConversationState : BaseGameState, IConversationState
    {
        private Conversation conversation;
        private Player player;

        public GameState GameState => this;

        public ConversationState(Game game)
            : base(game)
        {
            Game.Services.AddService<IConversationState>(this);

            conversation = new();

            SceneAction action = new()
            {
                Action = ActionType.Talk,
                Parameter = "Help"
            };

            List<SceneOption> options = new()
            {
                new SceneOption("Help!", "Help", action)
            };

            action = new()
            {
                Action = ActionType.End,
                Parameter = ""
            };

            options.Add(new("Goodbye.", "Goodbye", action));

            GameScene scene = new(game, "Oh no! The unthinkable has happened! A
thief has stolen Greynar's eyes. With out them he will not be able to animated and
defend us. You have to do something or the monsters outside the village will crush
us.", options);
            conversation.AddScene("Hello", scene);

            options = new();
            options.Add(new("Goodbye.", "Goooodbye", new() { Action = ActionType.End,
Parameter = "" }));

            scene = new(game, "Oh thank the heavens for you!", options);
            conversation.AddScene("Help", scene);

            conversation.FirstScene = "Hello";
            conversation.StartConversation();
        }
```

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

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

            foreach (GameScene scene in conversation.Scenes.Values)
            {
                scene.ItemSelected += Scene_ItemSelected;
            }
        }

        private void Scene_ItemSelected(object sender, SelectedIndexEventArgs e)
        {
            ButtonGroup btn = (ButtonGroup)sender;

            switch (btn.Action.Action)
            {
                case ActionType.End:
                    manager.PopState();
                    break;
                case ActionType.Talk:
                    conversation.ChangeScene(btn.Action.Parameter);
                    break;
            }
        }

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

            base.Update(gameTime);
        }

        public override void Draw(GameTime gameTime)
        {

            GraphicsDevice.SetRenderTarget(renderTarget);
            GraphicsDevice.Clear(Color.Transparent);

            SpriteBatch.Begin();

            base.Draw(gameTime);

            conversation.Draw(gameTime, SpriteBatch);

            SpriteBatch.End();

            GraphicsDevice.SetRenderTarget(null);

            SpriteBatch.Begin();

            SpriteBatch.Draw(renderTarget, new Rectangle(Point.Zero,
Settings.Resolution), Color.White);
```

```
            SpriteBatch.End();
        }

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

        public void StartConversation()
        {
            conversation.StartConversation();
        }
    }
}
using Microsoft.Xna.Framework;
using Psilibrary.ConversationComponents;
using SummonersTale.Forms;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SummonersTale.StateManagement
{
    public interface IConversationState
    {
        GameState GameState { get; }
    }

    public class ConversationState : BaseGameState, IConversationState
    {
        private Conversation conversation;
        private Player player;

        public GameState GameState => this;

        public ConversationState(Game game)
            : base(game)
        {
            Game.Services.AddService<IConversationState>(this);

            conversation = new();

            SceneAction action = new()
            {
                Action = ActionType.Talk,
                Parameter = "Help"
            };

            List<SceneOption> options = new()
            {
                new SceneOption("Help!", "Help", action)
            };

            action = new()
            {
                Action = ActionType.End,
```

```
                Parameter = ""
            };

            options.Add(new("Goodbye.", "Goodbye", action));

            GameScene scene = new(game, "Oh no! The unthinkable has happened! A
thief has stolen Greynar's eyes. With out them he will not be able to animated and
defend us. You have to do something or the monsters outside the village will crush
us.", options);
            conversation.AddScene("Hello", scene);

            options = new();
            options.Add(new("Goodbye.", "Gooodbye", new() { Action = ActionType.End,
Parameter = "" }));

            scene = new(game, "Oh thank the heavens for you!", options);
            conversation.AddScene("Help", scene);

            conversation.FirstScene = "Hello";
            conversation.StartConversation();
        }

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

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

            foreach (GameScene scene in conversation.Scenes.Values)
            {
                scene.ItemSelected += Scene_ItemSelected;
            }
        }

        private void Scene_ItemSelected(object sender, SelectedIndexEventArgs e)
        {
            ButtonGroup btn = (ButtonGroup)sender;

            switch (btn.Action.Action)
            {
                case ActionType.End:
                    manager.PopState();
                    break;
                case ActionType.Talk:
                    conversation.ChangeScene(btn.Action.Parameter);
                    break;
            }
        }

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

            base.Update(gameTime);
        }
```

```
        public override void Draw(GameTime gameTime)
        {

            GraphicsDevice.SetRenderTarget(renderTarget);
            GraphicsDevice.Clear(Color.Transparent);

            SpriteBatch.Begin();

            base.Draw(gameTime);

            conversation.Draw(gameTime, SpriteBatch);

            SpriteBatch.End();

            GraphicsDevice.SetRenderTarget(null);

            SpriteBatch.Begin();

            SpriteBatch.Draw(renderTarget, new Rectangle(Point.Zero,
Settings.Resolution), Color.White);

            SpriteBatch.End();
        }

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

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

Since this is a game state, there is an interface for it. Like most of the others, this one has a single property. It inherits from BaseGameState and implements the interface. For fields, there is a conversation and the player. And, for properties, there is one required by the interface.

The constructor registers the state as a service, as we've done before. It then creates a new conversation. After that, I create a conversation through code. The flow is you create a list for the scene options. You create the options and add them to the list. Once you have the options, you create a scene and add it to the conversation. You add scenes until the conversation is finished. For now, I wire the handlers for the scenes of the conversation in the LoadContent method. They will eventually need to be wired elsewhere. This is a demonstration.

In the event handler, I cast the sender to a ButtonGroup. I then do a switch on the action the conversation will perform. If the action is End, I pop the state off the stack. If it is Talk, I change the scene to the parameter.

In the Update method I call the Update method of the conversation. The Draw method I haven't completely worked it out yet. What I want is for the state below to be visible. However, because I am using render targets, we get a black background. I want to try a few different options before I settle on the solution. The actual rendering is the same as the other states. You set the render target, clear the buffer, and then render. Once rendering is done, reset the render target back to the screen and then render the render target at the current resolution.

There is a method, SetConversation, that will eventually set the conversation that is being started. It takes as parameters the player object and the name of the conversation. The other method is StartConversation, it resets a conversation to the beginning.

There are two things left to do. One is to trigger the state somehow. Two is to update the games to create a conversation state. Because this tutorial is long enough, I will opt for the quickest solution to triggering the state. That is, in the MainMenuState, for the second menu item, I will switch states.

Let's tackle the MainMenuState first. All I did was update the event handler for the button to load a game replace the OldGameButton_Click method with the following.

```
private void OldGameButton_Click(object sender, EventArgs e)
{
    manager.PushState(Game.Services.GetService<IConversationState>().GameState);
    Visible = true;
}
```

That leaves the Desktop, iOS and Android classes. Other than the namespace, the three are identical. I will start with Desktop and explain the changes. For the other two, I will just give you the code. Replace the Desktop class with the following code.

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Psilibrary.TileEngine;
using SummonersTale;
using SummonersTale.StateManagement;
using System;

namespace SummonersTaleGame
{
    public class Desktop : Game
    {
        private readonly GameStateManager _manager;
        private readonly GraphicsDeviceManager _graphics;
        private SpriteBatch _spriteBatch;

        private GamePlayState _playState;
        private TitleState _titleState;
        private MainMenuState _mainMenuState;
        private NewGameState _newGameState;
        private ConversationState _conversationState;
```

```
        public SpriteBatch SpriteBatch => _spriteBatch;

        public TitleState TitleState => _titleState;
        public GamePlayState PlayState => _playState;
        public MainMenuState MainMenuState=> _mainMenuState;
        public NewGameState NewGameState => _newGameState;
        public ConversationState ConversationState => _conversationState;

        public Desktop()
        {
            Settings.Load();

            _graphics = new GraphicsDeviceManager(this)
            {
                PreferredBackBufferWidth = Settings.Resolution.X,
                PreferredBackBufferHeight = Settings.Resolution.Y,
            };

            _graphics.ApplyChanges();
            _manager = new GameStateManager(this);

            Services.AddService(typeof(GraphicsDeviceManager), _graphics);

            Content.RootDirectory = "Content";
            IsMouseVisible = true;

            Components.Add(_manager);
        }

        protected override void Initialize()
        {
            Components.Add(new FramesPerSecond(this));
            Components.Add(new Xin(this));

            _graphics.ApplyChanges();

            base.Initialize();
        }

        protected override void LoadContent()
        {
            _spriteBatch = new SpriteBatch(GraphicsDevice);
            Services.AddService(typeof(SpriteBatch), _spriteBatch);

            _playState = new(this);
            _titleState = new(this);
            _mainMenuState = new(this);
            _newGameState = new(this);
            _conversationState = new(this);

            _manager.PushState(_titleState);
        }

        protected override void Update(GameTime gameTime)
        {
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back ==
ButtonState.Pressed || Keyboard.GetState().IsKeyDown(Keys.Escape))
```

```
                Exit();

            // TODO: Add your update logic here

            base.Update(gameTime);
        }

        protected override void Draw(GameTime gameTime)
        {
            GraphicsDevice.Clear(Color.CornflowerBlue);

            base.Draw(gameTime);
        }
    }
}
```

That I did was add a new ConversationState field and a property to expose it. I then initialize it in the LoadContent method. The other two are identical, other than the root namespace they are in. Replace the Android class with the following version.

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using SummonersTale;
using SummonersTale.StateManagement;

namespace SummonersTaleGameAndroid
{
    public class Android : Game
    {
        private readonly GameStateManager _manager;
        private readonly GraphicsDeviceManager _graphics;
        private SpriteBatch _spriteBatch;

        private GamePlayState _playState;
        private TitleState _titleState;
        private MainMenuState _mainMenuState;
        private NewGameState _newGameState;
        private ConversationState _conversationState;

        public SpriteBatch SpriteBatch => _spriteBatch;

        public TitleState TitleState => _titleState;
        public GamePlayState PlayState => _playState;
        public NewGameState NewGameState => _newGameState;
        public MainMenuState MainMenuState => _mainMenuState;
        public ConversationState ConversationState => _conversationState;

        public Android()
        {
            SummonersTale.Settings.Load();

            _graphics = new GraphicsDeviceManager(this);
            _manager = new GameStateManager(this);

            Services.AddService(typeof(GraphicsDeviceManager), _graphics);
```

```
            Content.RootDirectory = "Content";
            IsMouseVisible = true;

            Components.Add(_manager);
        }

        protected override void Initialize()
        {
            SummonersTale.Settings.Resolution = new(
                GraphicsDevice.DisplayMode.Width,
                GraphicsDevice.DisplayMode.Height);

            Components.Add(new Xin(this));

            _graphics.ApplyChanges();

            base.Initialize();
        }

        protected override void LoadContent()
        {
            _spriteBatch = new SpriteBatch(GraphicsDevice);
            Services.AddService(typeof(SpriteBatch), _spriteBatch);

            _playState = new(this);
            _titleState = new(this);
            _mainMenuState = new(this);
            _newGameState = new(this);
            _conversationState = new(this);

            _manager.PushState(_titleState);
        }

        protected override void Update(GameTime gameTime)
        {
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back ==
ButtonState.Pressed || Keyboard.GetState().IsKeyDown(Keys.Escape))
                Exit();

            // TODO: Add your update logic here

            base.Update(gameTime);
        }

        protected override void Draw(GameTime gameTime)
        {
            GraphicsDevice.Clear(Color.CornflowerBlue);

            base.Draw(gameTime);
        }
    }
}
```

Now, the iOS class.

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
```

```
using Microsoft.Xna.Framework.Input;
using SummonersTale;
using SummonersTale.StateManagement;

namespace SummonersTaleGameiOS
{
    public class iOS : Game
    {
        private readonly GameStateManager _manager;
        private readonly GraphicsDeviceManager _graphics;
        private SpriteBatch _spriteBatch;

        private GamePlayState _playState;
        private TitleState _titleState;
        private MainMenuState _mainMenuState;
        private NewGameState _newGameState;
        private ConversationState _conversationState;

        public SpriteBatch SpriteBatch => _spriteBatch;

        public TitleState TitleState => _titleState;
        public GamePlayState PlayState => _playState;
        public MainMenuState MainMenuState => _mainMenuState;
        public NewGameState NewGameState => _newGameState;
        public ConversationState ConversationState => _conversationState;

        public iOS()
        {
            Settings.Load();

            _graphics = new GraphicsDeviceManager(this)
            {
                PreferredBackBufferWidth = Settings.Resolution.X,
                PreferredBackBufferHeight = Settings.Resolution.Y,
            };

            _graphics.ApplyChanges();
            _manager = new GameStateManager(this);

            Services.AddService(typeof(GraphicsDeviceManager), _graphics);

            Content.RootDirectory = "Content";
            IsMouseVisible = true;

            Components.Add(_manager);
        }

        protected override void Initialize()
        {
            Components.Add(new FramesPerSecond(this));
            Components.Add(new Xin(this));

            _graphics.ApplyChanges();

            base.Initialize();
        }

        protected override void LoadContent()
```

```
        {
            _spriteBatch = new SpriteBatch(GraphicsDevice);
            Services.AddService(typeof(SpriteBatch), _spriteBatch);

            _playState = new(this);
            _titleState = new(this);
            _mainMenuState = new(this);
            _newGameState = new(this);
            _conversationState = new(this);

            _manager.PushState(_titleState);
        }

        protected override void Update(GameTime gameTime)
        {
            // TODO: Add your update logic here

            base.Update(gameTime);
        }

        protected override void Draw(GameTime gameTime)
        {
            GraphicsDevice.Clear(Color.CornflowerBlue);

            base.Draw(gameTime);
        }
    }
}
```

Now, you can build and run on all platforms. The touch input on the emulator is a little wonky. I will have to check my archives to see how I resolved that. I want to get this tutorial out there, though, as it covers an important aspect of the game. Especially considering that it is the basis for the combat engine.

So, I'm most definitely not going to dive any further into this tutorial. I think I've fed you more than enough for one day. I covered a lot, and I don't want to overload you. I encourage you to keep visiting my blog for the latest news on my tutorials.

Good luck with your game programming adventures.
Cynthia