Shadow Monsters – MonoGame Tutorial Series
Chapter 6
Conversation

This tutorial series is about creating a Pokemon style game with the MonoGame Framework called Shadow Monsters. 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 blog: Shadow Monsters. The source code for each tutorial will be available as well. I will be using Visual Studio 2019 Community for the series. The code should compile on the 2013, 2015 and 2017 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 give credit to Cynthia McMahon and add a link to my site, https://mygameprogrammingadventures.blogspot.com. Screenshots of your project and/or a video of game play would be appreciated.

I also want to mention that I assume you have a basic understanding of C# and MonoGame. If you don't I recommend that you learn basic C# and work with MonoGame a little. Enough to know the basics of fields, properties, methods, classes and the MonoGame framework.

Let's face it. Paul is pretty but useless. He does nothing but stand there. In this tutorial we will make Paul talk. Let's talk a bit about conversations. Conversations are made up of scenes. A scene has a number of options associated with it. Each option has an action. In implementing conversations I will be taking a bottom up approach.

The first thing we are going to need is fonts. Let's talk a little bit about fonts. Fonts are not royalty free, mostly. You can only include in your game royalty free fonts or fonts that you have purchased the rights to. I will be using Kootenay, a royalty free font.

Right click the Content folder, select Add and then New Folder. Name this new folder Fonts. Right click the Fonts folder, select Add and then New Item. On the left select General then scroll down to Text File. Name the file testfont.spritefont. Paste the following code into that file.

```
<?xml version="1.0" encoding="utf-8"?>
<!--
This file contains an xml description of a font, and will be read by the XNA
Framework Content Pipeline. Follow the comments to customize the appearance
of the font in your game, and to change the characters which are available to draw
with.
-->
<XnaContent xmlns:Graphics="Microsoft.Xna.Framework.Content.Pipeline.Graphics">
  <Asset Type="Graphics:FontDescription">

    <!--
    Modify this string to change the font that will be imported.
    -->
    <FontName>Kootenay</FontName>
```

```
    <!--
    Size is a float value, measured in points. Modify this value to change
    the size of the font.
    -->
    <Size>18</Size>

    <!--
    Spacing is a float value, measured in pixels. Modify this value to change
    the amount of spacing in between characters.
    -->
    <Spacing>0</Spacing>

    <!--
    UseKerning controls the layout of the font. If this value is true, kerning information
    will be used when placing characters.
    -->
    <UseKerning>true</UseKerning>

    <!--
    Style controls the style of the font. Valid entries are "Regular", "Bold", "Italic",
    and "Bold, Italic", and are case sensitive.
    -->
    <Style>Regular</Style>

    <!--
    If you uncomment this line, the default character will be substituted if you draw
    or measure text that contains characters which were not included in the font.
    -->
    <!-- <DefaultCharacter>*</DefaultCharacter> -->

    <!--
    CharacterRegions control what letters are available in the font. Every
    character from Start to End will be built and made available for drawing. The
    default range is from 32, (ASCII space), to 126, ('~'), covering the basic Latin
    character set. The characters are ordered according to the Unicode standard.
    See the documentation for more information.
    -->
    <CharacterRegions>
      <CharacterRegion>
        <Start>&#32;</Start>
        <End>&#126;</End>
      </CharacterRegion>
    </CharacterRegions>
  </Asset>
</XnaContent>
```

Now open the Content Pipeline Tool. From the menu bar select Add Existing Folder. Choose the Fonts folder that we just created. This will add the folder we just created and its contents. Save the project and close the tool.

We will be using multiple fonts so I created a font manager. Right click the ShadowMonsters project, select Add and then Class. Name this new class FontManager. Here is the code.

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using System.Collections.Generic;

namespace ShadowMonsters
```

```
{
    public class FontManager : DrawableGameComponent
    {
        private readonly static Dictionary<string, SpriteFont> _fonts =
            new Dictionary<string, SpriteFont>();
        private static Game gameRef;

        public FontManager(Game game) : base(game)
        {
            gameRef = game;
        }

        protected override void LoadContent()
        {
            _fonts.Add("testfont", Game.Content.Load<SpriteFont>(@"Fonts\testfont"));
        }

        public static SpriteFont GetFont(string name)
        {
            if (gameRef.GraphicsDevice.Viewport.Height == 1080)
            {
                name += "1080";
            }

            return _fonts[name];
        }

        public static bool ContainsFont(string name)
        {
            if (gameRef.GraphicsDevice.Viewport.Height == 1080)
            {
                name += "1080";
            }

            return _fonts.ContainsKey(name);
        }
    }
}
```

This class is a DrawableGameComponent. I inherited it from that insted of GameComponent to have access to the LoadContent method that will load the fonts. It will also be added to the list of components so if I need direct access to it later I will have it.

There are two static fields. The first, _fonts, holds the loaded fonts. The second, gameRef, will be used for switching resolutions. More on that in a future tutorial. They are static because I want to be able to reference them with out accessing the component.

The constructor takes a Game parameter because it inherits from DrawableGameComponent. It sets the gameRef field. LoadContent loads our testfont that we just created.

GetFont takes a string parameter that is the name of the font that we want. I use the gameRef field to check the height of the window the game is hosted in. If it is 1080 I append 1080 to the name. This will be used in a future tutorial when we cover having multiple resolutions.

ContainsFont is used to check if a font exists. It again checks the height of the window to see

what font to check.

To get started on conversations right click the ShadowMonsters 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 ShadowMonsters.ConversationComponents
{
    public enum ActionType
    {
        Talk,
        End,
        Change,
        Quest,
        Teach,
        Shop,
        GiveItems,
        GiveKey,
        Battle,
        Rest,
    }

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

    }

    public class SceneOption
    {
        private string optionText;
        private string optionScene;
        private SceneAction optionAction;

        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; }
        }
```

```
        private SceneOption()
        {
        }

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

    }
}
```

There is an enumeration in this class that holds all the possible actions that a scene option might have. Talk moves to the next scene. End exits out of the conversation. Change changes to a different conversation. Quest gives the player a quest. Teach gives the player a shadow monster. Shop opens the shop interface. GiveItems gives the player one or more items. GiveKey gives the player a key, Battle starts the battle interface and Rest heals the player's shadow monsters.

In this class there is a class SceneAction that defines the action a scene option takes. It has two public fields: Action and Parameter. Action is what the SceneAction is and Parameter is used for passing information about the action, like the name of the quest to be given or the names of the items being given.

The SceneOption class has three fields: optionText, optionScene and optionAction. The optionText field is the text to be displayed. The optionScene is what scene is displayed if the option is selected. The optionAction field is the action taken if the option is selected. There are properties to expose the fields.

There are two constructors in this class. The first is a private constructor that takes no parameters. It is used in saving and loading conversations later on. The second takes as parameters the text for the option, the scene to change to and the action.

That is all there is to scene options and scene actions, The next thing to be implemented is game scenes. Right click the ConversationComponents folder, select Add and then Class. Name this new class GameScene. Here is the code for that class.

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

namespace ShadowMonsters.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 bool isMouseOver;

private Vector2 menuPosition = new Vector2(50, 475);

#endregion

#region Property Region

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

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

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; }
    set { selectedIndex = MathHelper.Clamp(value, 0, options.Count - 1); }
}

public bool IsMouseOver
{
    get { return isMouseOver; }
}

public Color NormalColor
{
    get { return normal; }
    set { normal = value; }
}
```

```
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;
    options = new List<SceneOption>();
}

public GameScene(string text, List<SceneOption> options)
    : this()
{
    this.text = text;
    this.options = options;
    textPosition = Vector2.Zero;
}

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

#endregion

#region Method Region

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

    StringBuilder sb = new StringBuilder();
    float currentLength = 0f;

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

    foreach (string s in parts)
    {
        Vector2 size = FontManager.GetFont("testfont").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()
        {
            if (Xin.CheckKeyReleased(Keys.Up) || Xin.CheckKeyReleased(Keys.W))
            {
                selectedIndex--;
                if (selectedIndex < 0)
                {
                    selectedIndex = options.Count - 1;
                }
            }
            else if (Xin.CheckKeyReleased(Keys.Down) || Xin.CheckKeyReleased(Keys.S))
            {
                selectedIndex++;
                if (selectedIndex > options.Count - 1)
                {
                    selectedIndex = 0;
                }
            }
        }

        public void Draw(SpriteBatch spriteBatch, Texture2D background)
        {
            Color myColor;

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

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

            spriteBatch.DrawString(FontManager.GetFont("testfont"),
                text,
                textPosition,
                Color.White);

            Vector2 position = menuPosition;

            Rectangle optionRect = new Rectangle(0, (int)position.Y, 1280,
FontManager.GetFont("testfontont").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;
                }
                else
                {
                    myColor = NormalColor;
                }

                spriteBatch.DrawString(FontManager.GetFont("testfont"),
                    options[i].OptionText,
                    position,
                    myColor);

                position.Y += FontManager.GetFont("testfont").LineSpacing + 5;
                optionRect.Y += FontManager.GetFont("testfont").LineSpacing + 5;
            }
        }

        #endregion
    }
}
```

First, what is a scene. A scene has text and options or responses to the text. The play selects the option that they want and the conversation moves to the next scene, or ends. There is a protected Game field because we will be reusing the class in other places. Next is the text field. There is a List<SceneOption> that holds the responses the player can give. The next two fields determine what color the text will be drawn in. textPosition is the position the text will be drawn. isMouseOver tells if the mouse is over a given line. MenuPosition is where to start drawing the options. There are properties to expose the fields.

There are three constructors in the class. The first is private and is used for loading and saving scenes which we will cover in a future tutorial. The next takes the text for the scene and the options for the scene. It calls the first constructor. The final constructor takes a Game parameter, the text for the scene and the options for the scene. It then calls the second constructor which will call the first. All three set fields to the parameters passed in or their default values.

SetText is used to add line breaks to the text so that it will wrap in the desired area. Here we should have some code to handle multiple resolutions be we will get to that in a future tutorial. The first thing that happens is it initializes the textPosition field. This is one of the two places where we should handle multiple resolutions. It then creates a StringBuilder to build the broken down text. The currentLength variable is for determining how far across the row we are. I then break the string down into its parts based on a space. I then loop over all of the words. Inside of the loop I measure the size of the word using the FontManager and the MeasureString method. If the currentLength plus the width of the word is less then 500 I append the word and a space and increment currentLength with the width. If it is not I append

a carriage return, the word, a space and reset current length to 0. Finally I set the text field to be the string builder as a string.

The Update method is used to move from one scene to another scene. To move up a scene you use the W or up arrow and to move down the S or down arrow. I included W and S instead of just arrows because the player's hands are already over WASD for movement and they won't have to move their hands. Also, the mouse will work but that is handled in the Draw method. The check for key releases is done using the Xin class that we created previously. To move up we decrement the selectedIndex field. If it is less than zero we move to the last entry in the list. Similarly, to move down we increment the selectedIndex field. If it is greater than or equal to the number of entries we move to the first element.

The Draw method is where we draw the scene. It takes as parameters a SpriteBatch object that is between calls to Begin and End and a Texture2D for the background that can be null. There is a local variable, myColor that tells what color to draw an option in. In the event that SetText was not called I check to see if textPosition is Vector2.Zero. If it is I call the SetText method. If the background is not null I draw it at Vector2.Zero. This is another place where we should handle multiple resolutions. We should have different textures for each screen resolution that we want to support or handle scaling in some manner. Next I draw the text using the testfont that we created earlier. Now I set a local variable to be the position of the menuPosition field and create a rectangle that describes a line of text. Here we should use the width of the window instead of hard coding a value. I then set the mouseOver field to false. Next I loop over the options. If the rectangle contains the mouse as a point I change the selected index to that menu item and mouseOver to true. If the loop counter and the selectedIndex are the same myColor is set to HighLightColor, other wise to NormalColor. Next I draw the option text and then move to the next line.

We have our action, option and scene now we can move onto the entire conversation. Right click the ConversationComponents folder, select Add and then Class. Name this new class Conversation. Here is the code for that class.

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

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

        private string name;
        private string firstScene;
        private string currentScene;
        private readonly Dictionary<string, GameScene> scenes;
        private string backgroundName;
        private Texture2D background;

        #endregion
```

```
#region Property Region

public string Name
{
    get { return name; }
}

public string FirstScene
{
    get { return firstScene; }
}

public GameScene CurrentScene
{
    get
    {
        if (!string.IsNullOrEmpty(currentScene) && scenes.ContainsKey(currentScene))
            return scenes[currentScene];
        return null;
    }
}

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

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

public Texture2D Background
{
    get { return background; }
}

#endregion

#region Constructor Region

private Conversation()
{
    scenes = new Dictionary<string, GameScene>();
}

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

#endregion

#region Method Region

public void LoadContent(ContentManager conetnt)
```

```
        {
            background = conetnt.Load<Texture2D>(@"Textures\" + BackgroundName);
        }

        public void Update()
        {
            if (CurrentScene != null)
            {
                CurrentScene.Update();
            }
        }

        public void Draw(SpriteBatch spriteBatch)
        {
            if (CurrentScene != null)
            {
                CurrentScene.Draw(spriteBatch, background);
            }
        }

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

        public void RemoveScene(string optionText)
        {
            scenes.Remove(optionText);
        }

        public GameScene GetScene(string sceneName)
        {
            return scenes.ContainsKey(sceneName) ? scenes[sceneName] : null;
        }

        public void StartConversation()
        {
            currentScene = firstScene;
        }

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

This class has six fields, the name of the conversation, the firstScene, the currentScene, the scenes, the name of the background and the actual background. There are fields to expose the values of the fields. The CurrentScene property checks to make sure the currentScene field is not null that the scene is in the dictionary of scenes.

There are two constructors in the class. The first is a private one that will be used for loading and saving conversations. The second takes as parameters the name of the conversation and

the first scene.

The LoadContent method will load the background using the BackgroundName property. Update checks to see if the current scene is not null and if it isn't calls its Update method. The same is true for the Draw method.

AddScene is used to dynamically add a scene to the conversation. It is useful if a quest is finished and it opens up a new scene. RemoveScene is useful for the same reason. If the player accepts a quest the scene may no longer be valid. GetScene checks if the scene name passed in exists and returns it if it does. Otherwise it returns null. StartConversation resets a conversation back to the first scene. Finally ChangeScene will change the current scene to the scene name passed in. You might consider checking if the scene exists before setting the value.

The next thing that I want to implement is a conversation manager. Right click the ConversationComponents folder, select Add and then Class. Name this new class ConversationManager. Here is the code.

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

namespace ShadowMonsters.ConversationComponents
{
    public class ConversationManager
    {
        #region Field Region
        private static ConversationManager instance = new ConversationManager();
        private Dictionary<string, Conversation> conversationList = new Dictionary<string,
Conversation>();

        #endregion

        #region Property Region

        public static ConversationManager Instance
        {
            get { return instance; }
            set { instance = value; }
        }

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

        #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)
{
    if (conversationList.ContainsKey(name))
        return conversationList[name];

    return null;
}

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

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

public void CreateConversations(Game gameRef)
{
    ConversationList.Clear();
    Conversation c = new Conversation("PaulHello", "Hello");

    List<SceneOption> options = new List<SceneOption>();
    SceneOption teach = new SceneOption(
        "Teach",
        "Teach",
        new SceneAction() { Action = ActionType.Teach, Parameter = "none" });
    options.Add(teach);

    SceneOption option = new SceneOption(
        "Good bye.",
        "",
        new SceneAction() { Action = ActionType.End, Parameter = "none" });
    options.Add(option);

    GameScene scene = new GameScene(
        gameRef,
        "Hello, my name is Paul. I'm still learning about training shadow monsters.",
        options);

    c.AddScene("Hello", scene);

    options = new List<SceneOption>();

    scene = new GameScene(
        gameRef,
        "I have given you Brownie!",
        options);
```

```
            option = new SceneOption(
                "Goodbye",
                "",
                new SceneAction() { Action = ActionType.End, Parameter = "none" });

            options.Add(option);

            c.AddScene("Teach", scene);
            ConversationList.Add("PaulHello", c);
        }

        #endregion
    }
}
```

This class is a singleton because you only ever want one conversation manager. For that reason there is a static field instance of type ConversationManager. There is also a field conversationList that holds all of the conversations in the game. There are also read/write properties to expose the fields. The Instance one will be used when we get to the editor for creating conversations. There is also a private constructor so that you have to use the instance in order to access the class.

The AddConversation method is used to add a new conversation to the conversationList field. It checks to make sure a conversation by that name does not already exist. GetConversation is used to get a conversation from the dictionary of conversations. ContainsConversation checks to see if there is a conversation with the given name. ClearConversations removes all conversations from the dictionary.

CreateConversations programmatically creates the conversations. Right now there is only Paul's one conversation. The first step is to clear the old conversations, if any exist. Next create the conversation. Next create a list for the scene options. Then each scene option. The first scene option is Teach which will give a new shadow monster to the player. The action is Teach with no parameter. If there is no parameter you should pass in none. The option is then added to the list. A new option is created called Good bye that will end the current conversation. So the action is End and the parameter is none. The option is then added to the list of options. I then create a scene passing in some text appropriate to a low level NPC. The scene is then added to the scenes for the conversation. I follow a similar process to create another scene and add it to the conversation. The conversation is then added to the collection of conversations.

Now I'm going to add a game state for conversations. Right click the GameStates folder in the Solution Explorer, select Add and then Class. Name this new class ConversationState. Here is the code.

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using ShadowMonsters.Characters;
using ShadowMonsters.ConversationComponents;
using System;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Text;
using System.Threading.Tasks;

namespace ShadowMonsters.GameStates
{
    public interface IConversationState
    {
        void SetConversation(Character character);
        void StartConversation();
    }

    public class ConversationState : BaseGameState, IConversationState
    {
        #region Field Region

        private Conversation conversation;
        private Character speaker;

        #endregion

        #region Property Region
        #endregion

        #region Constructor Region

        public ConversationState(Game game)
            : base(game)
        {
            game.Services.AddService(typeof(IConversationState), this);
        }

        #endregion

        #region Method Region

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

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

        public override void Update(GameTime gameTime)
        {
            if (conversation.CurrentScene == null)
            {
                return;
            }

            if (Xin.CheckKeyReleased(Keys.Space) ||
                Xin.CheckKeyReleased(Keys.Enter) ||
                (Xin.CheckMouseReleased(MouseButtons.Left) &&
                conversation.CurrentScene.IsMouseOver))
            {
                switch (conversation.CurrentScene.OptionAction.Action)
                {
                    case ActionType.Teach:
```

```
                        BuyShadowMonster();
                        break;
                    case ActionType.Change:
                        speaker.SetConversation(conversation.CurrentScene.OptionScene);
                        manager.PopState();
                        break;
                    case ActionType.End:
                        manager.PopState();
                        break;
                    case ActionType.GiveItems:
                        break;
                    case ActionType.GiveKey:
                        if (conversation.CurrentScene.OptionAction.Parameter != null)
                        {
                            bool success = int.TryParse(
                                conversation.CurrentScene.OptionAction.Parameter,
                                out int key);

                            if (success)
                            {
                            }

                            conversation.ChangeScene(conversation.CurrentScene.OptionScene);
                        }
                        break;
                    case ActionType.Quest:
                        CheckQuest();
                        break;
                    case ActionType.Rest:
                        conversation.ChangeScene(conversation.CurrentScene.OptionScene);
                        break;
                    case ActionType.Shop:
                        break;
                    case ActionType.Talk:
                        conversation.ChangeScene(conversation.CurrentScene.OptionScene);
                        break;
                }
            }

            conversation.Update();
            base.Update(gameTime);
        }

        private void BuyShadowMonster()
        {
            string scene = conversation.CurrentScene.OptionScene;


conversation.CurrentScene.Options.RemoveAt(conversation.CurrentScene.SelectedIndex);
            conversation.CurrentScene.SelectedIndex--;
            conversation.ChangeScene(scene);
        }

        private void CheckQuest()
        {
        }

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

```
            GameRef.SpriteBatch.Begin();
            conversation.Draw(GameRef.SpriteBatch);
            GameRef.SpriteBatch.End();
        }

        public void SetConversation(Character character)
        {
            speaker = character;

            if
(ConversationManager.Instance.ConversationList.ContainsKey(character.Conversation))
            {
                conversation =
ConversationManager.Instance.ConversationList[character.Conversation];
            }
            else
            {
                manager.PopState();
            }
        }

        public void StartConversation()
        {
            if (conversation != null)
            {
                conversation.StartConversation();
            }
        }

        #endregion
    }
}
```

There is a public interface that the class will implement. It has two methods that must be implemented. The SetConversation method is used to set the current speaker and grab the current conversation with the speak. The other starts the conversation.

The class inherits from BaseGameState and implements the IConversationState interface. For fields there is the current conversation and the speaker. There are no properties in this class because I don't need to share any values with other classes. The constructor takes a Game parameter because it inherits from BaseGameState. It registers the class as a service using the interface. There is a stub for LoadContent for when it will be necessary in a future tutorial.

There is a lot going on in the Update method. First, there is a check to see if conversation is null. If it is we exit the method. Next there is a check to see if the enter or space key have been released or if the mouse is over a scene option and the left mouse button has been released. This means the player has selected an option. There is a switch on the action type for the selected option. Some of the cases are stubs or partially implemented and will be fully implemented later on. The Teach action calls a method BuyShadowMonster that will buy the giving shadow monster. The Change action changes the conversation for the speaker to the scene option then pops the conversation state off the stack. End just pops the conversation state off of the stack. GiveItems is just a stub for now. GiveKey checks if the scene option parameter is not null if it is not it uses int.TryParse to try and parse the string into an integer. If

that succeeds we would add the key to the key manager that will be implemented in a future tutorial. The Quest action type calls a method stub CheckQuest that will check the state of a quest or give a new quest. The Rest action just calls the ChangeScene method to switch to the next scene. In a future tutorial, most likely the next one, I will cover creating a player component that will have shadow monsters. The Shop action is just a stub for now. Finally Talk moves to the next scene based on the selected option. After the switch I call the Update method of the conversation and the base class.

The Draw method calls Begin on the sprite batch object. It then calls the Draw method of the conversation. It then calls the End method of sprite batch object.

SetConversation sets the speaker field to the character that was passed in. It then checks to see if the desired conversation exists. If it does it sets the conversation field. Otherwise it pops the state off the stack. StartConversation checks if the conversation is null. If it is not it calls the StartConversation method of the conversation field.

Now it is time to implement the new state. First, we need to create a state. That will be done in the Game1 class like we did for the GamePlayState. First, add the following field and property.

```
private readonly ConversationState conversationState;
public ConversationState ConversationState => conversationState;
```

Next update the constructor to the following.

```
public Game1()
{
    graphics = new GraphicsDeviceManager(this)
    {
        PreferredBackBufferWidth = 1280,
        PreferredBackBufferHeight = 720
    };

    graphics.ApplyChanges();

    Content.RootDirectory = "Content";

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

    gamePlayState = new GamePlayState(this);
    conversationState = new ConversationState(this);

    stateManager.PushState(gamePlayState);
    IsMouseVisible = true;
}
```

The last thing to do is to trigger the conversation. That will be done in the GamePlayState's Update method. We will be a little mean and make it so that the player is facing the character to start the conversation. We could be really mean and make it so that they are both facing each other. Add the following field and change the Update method of the GamePlayState to the following.

```
int frameCount = 0;

public override void Update(GameTime gameTime)
{
    engine.Update(gameTime);
    frameCount++;
    if (Xin.KeyboardState.IsKeyDown(Keys.W) && !inMotion)
    {
        motion.Y = -1;
        sprite.CurrentAnimation = AnimationKey.WalkUp;
        sprite.IsAnimating = true;
        inMotion = true;
        collision = new Rectangle(
            (int)sprite.Position.X,
            (int)sprite.Position.Y - Engine.TileHeight * 2,
            Engine.TileWidth,
            Engine.TileHeight);
    }
    else if (Xin.KeyboardState.IsKeyDown(Keys.S) && !inMotion)
    {
        motion.Y = 1;
        sprite.CurrentAnimation = AnimationKey.WalkDown;
        sprite.IsAnimating = true;
        inMotion = true;
        collision = new Rectangle(
            (int)sprite.Position.X,
            (int)sprite.Position.Y + Engine.TileHeight * 2,
            Engine.TileWidth,
            Engine.TileHeight);
    }
    else if (Xin.KeyboardState.IsKeyDown(Keys.A) && !inMotion)
    {
        motion.X = -1;
        sprite.CurrentAnimation = AnimationKey.WalkLeft;
        sprite.IsAnimating = true;
        inMotion = true;
        collision = new Rectangle(
            (int)sprite.Position.X - Engine.TileWidth * 2,
            (int)sprite.Position.Y,
            Engine.TileWidth,
            Engine.TileHeight);
    }
    else if (Xin.KeyboardState.IsKeyDown(Keys.D) && !inMotion)
    {
        motion.X = 1;
        sprite.CurrentAnimation = AnimationKey.WalkRight;
        sprite.IsAnimating = true;
        inMotion = true;
        collision = new Rectangle(
            (int)sprite.Position.X + Engine.TileWidth * 2,
            (int)sprite.Position.Y,
            Engine.TileWidth,
            Engine.TileHeight);
    }


    if (motion != Vector2.Zero)
    {
        motion.Normalize();
        motion *= (sprite.Speed * (float)gameTime.ElapsedGameTime.TotalSeconds);
```

```
    Rectangle pRect = new Rectangle(
            (int)(sprite.Position.X + motion.X),
            (int)(sprite.Position.Y + motion.Y),
            Engine.TileWidth,
            Engine.TileHeight);

    if (pRect.Intersects(collision))
    {
        sprite.IsAnimating = false;
        inMotion = false;
        motion = Vector2.Zero;
    }

    foreach (Point p in engine.Map.CharacterLayer.Characters.Keys)
    {
        Rectangle r = new Rectangle(
            p.X * Engine.TileWidth,
            p.Y * Engine.TileHeight,
            Engine.TileWidth,
            Engine.TileHeight);

        if (r.Intersects(pRect))
        {
            motion = Vector2.Zero;
            sprite.IsAnimating = false;
            inMotion = false;
        }
    }

    Vector2 newPosition = sprite.Position + motion;
    newPosition.X = (int)newPosition.X;
    newPosition.Y = (int)newPosition.Y;

    sprite.Position = newPosition;
    motion = sprite.LockToMap(
        new Point(
            map.WidthInPixels,
            map.HeightInPixels),
        motion);

    if (motion == Vector2.Zero)
    {
        Vector2 origin = new Vector2(
                sprite.Position.X + sprite.Origin.X,
                sprite.Position.Y + sprite.Origin.Y);
        sprite.Position = Engine.VectorFromOrigin(origin);
        inMotion = false;
        sprite.IsAnimating = false;
    }
}

if ((Xin.CheckKeyReleased(Keys.Space) ||
    Xin.CheckKeyReleased(Keys.Enter)) && frameCount >= 5)
{
    frameCount = 0;
    foreach (Point s in engine.Map.CharacterLayer.Characters.Keys)
    {
        Character c = engine.Map.CharacterLayer.Characters[s];

        AnimationKey animation = sprite.CurrentAnimation;
```

```
            if (animation == AnimationKey.WalkLeft &&
                ((int)c.Sprite.Position.X > (int)sprite.Position.X ||
                    (int)c.Sprite.Position.Y != (int)sprite.Position.Y))
            {
                continue;
            }

            if (animation == AnimationKey.WalkUp &&
                ((int)c.Sprite.Position.X != (int)sprite.Position.X ||
                    (int)c.Sprite.Position.Y > (int)sprite.Position.Y))
            {
                continue;
            }

            if (animation == AnimationKey.WalkRight &&
                ((int)c.Sprite.Position.X < (int)sprite.Position.X ||
                    (int)c.Sprite.Position.Y != (int)sprite.Position.Y))
            {
                continue;
            }

            if (animation == AnimationKey.WalkDown &&
                ((int)c.Sprite.Position.X != (int)sprite.Position.X ||
                    (int)c.Sprite.Position.Y < (int)sprite.Position.Y))
            {
                continue;
            }

            float distance = Vector2.Distance(
                sprite.Origin + sprite.Position,
                c.Sprite.Origin + c.Sprite.Position);

            if (Math.Abs(distance) < Engine.TileWidth + Engine.TileWidth / 2)
            {
                manager.PushState(
                    (ConversationState)GameRef.ConversationState);

                GameRef.ConversationState.SetConversation(c);
                GameRef.ConversationState.StartConversation();
                break;
            }
        }
    }

    Engine.Camera.LockToSprite(map, sprite, new Rectangle(0, 0, 1280, 720));
    sprite.Update(gameTime);

    base.Update(gameTime);
}
```

The frameCount field counts the number of frames since the last time the space bar or enter key have been pressed. This is because when exiting a conversation there is the potential that the key press will be picked up by the game play state and things will be thrown back into the conversation state.

For new code there is a check to see if either the space bar or enter key have been pressed and the number of frames is greater than 5. 5 is a little high but it works so we'll go with it. The

frameCount field is then reset back to 0. I then loop over all of the characters on the map. Inside I grab the current character and the current animation of the player. Next there are a series of if statements that check to see if the player is facing the character. That is done by comparing the X and Y coordinates. If the X coordinates are the same then the two sprites are in the same column. If the Y coordinates are the same they are on the same row. If the player's X coordinate is left than the X coordinate of the character's X coordinate then it is to the left and you check if the player is facing right. The other cases are similar and I will leave it to you to figure them out.

Next I calculate the distance between the origins of the two sprites. I compare the absolute value of the two with the width of a tile plus half a tile. This ensures that the two sprites are beside each other. If they are I push the conversation state onto of the stack, set the conversation to be the character's conversation and start the conversation.

I'm going to wrap this tutorial up here. I will be starting work on the next tutorial shortly. Keep checking back on the blog for news on that tutorial. I hope to have it up in the next week or so.

I wish you the best in your MonoGame Programming Adventures!