Creating a Role Playing Game with XNA Game Studio 3.0
Part 41
NPC Dialogs - Part 2

To follow along with this tutorial you will have to have read the previous tutorials to understand much of what it going on. You can find a list of tutorials here: XNA 3.0 Role Playing Game Tutorials You will also find the latest version of the project on the web site on that page. If you want to follow along and type in the code from this PDF as you go, you can find the previous project at this link: http://www.jtmbooks.com/rpgtutorials/New2DRPG40.zip You can download the graphics from this link: Graphics.zip

I this tutorial I will continue on with adding in dialogs for the NPCs in the game. I'm using a slight variation of what Nick Gravelyn did in his tile engine series because I liked the way he did it. I will be enhancing it in future tutorials to add in more features.

The first thing that you will want to do is download the simple script that I made for this tutorial. You can find the tutorial at script download. After you have downloaded it and unzipped it you will want to add it to the **Content** folder of the game. To do that right click the **Content** folder and select **Add** and then **New Item**. Navigate to where you unzipped it and add **script1.script**. You will have to make sure and set the **Objects of type:** combo box is set to all. Make sure that the **Build Action** for the script is set to **None** and the **Copy To Output Directory** is set to **Copy always**.

I guess the best way to start is to start with the format of a script file. For things like this I really like using XML documents. The classes in the .NET framework make working with XML files very easy. This document will be more complex than anything you have seen so far though. I will show you the document that I'm using and try and explain it.

```
<Dialogs>
  <Dialog>
    <Name Text="SkeletonDialog" />
    <Text>Leave me alone. I'm not bothering anybody!</Text>
    <Handlers>
      <Handler Text="Continue" Actions="StopDialog" />
    </Handlers>
  </Dialog>
  <Dialog>
    <Name Text="KnightDialog" />
    <Text>I lost my favorite sword. Will you help me find it?</Text>
    <Handlers>
      <Handler Text="Yes" Actions="StartDialog:ThankYou1" />
      <Handler Text="No" Actions="StartDialog:ThanksAnyway1" />
    </Handlers>
  </Dialog>
  <Dialog>
    <Name Text="ThankYou1" />
    <Text>Thank you so very much for agreeing to help me!</Text>
    <Handlers>
      <Handler Text="Continue" Actions="StopDialog" />
    </Handlers>
  </Dialog>
  <Dialog>
    <Name Text="ThanksAnyway1" />
    <Text>If you change your mind I will be here later.</Text>
    <Handlers>
      <Handler Text="Continue" Actions="StopDialog" />
    </Handlers>
  </Dialog>
```

```
  <Dialog>
    <Name Text="NinjaDialog" />
    <Text>That skeleton keeps bothering me! Will you go and ask it to leave me alone?</Text>
    <Handlers>
      <Handler Text="Yes" Actions="StartDialog:ThankYou2" />
      <Handler Text="No" Actions="StartDialog:ThanksAnyway2" />
    </Handlers>
  </Dialog>
  <Dialog>
    <Name Text="ThankYou2" />
    <Text>Thank you so very much for agreeing to help me!</Text>
    <Handlers>
      <Handler Text="Continue" Actions="StopDialog" />
    </Handlers>
  </Dialog>
  <Dialog>
    <Name Text="ThanksAnyway2" />
    <Text>If you change your mind I will be here later.</Text>
    <Handlers>
      <Handler Text="Continue" Actions="StopDialog" />
    </Handlers>
  </Dialog>
</Dialogs>
```

I decided that dialog is shorter than conversation so I will be using dialog instead of conversation. The root node of this document is called **Dialogs**. Inside of it will be a number of nodes called **Dialog**. These nodes will hold the actual dialogs for the NPCs. A dialog should have a **Name** node that will hold the name of the dialog to be used in the game in the **Text** attribute. These have to be unique inside a script file. You can not have duplicates. That is why if you look I have a **ThankYou1**, **ThankYou2**, **ThanksAnyway1** and **ThanksAnyway2**. There is also a **Text** node. This node holds the text that you want to appear when that dialog is active as its inner text.

The other node is called **Handlers**. Inside this you can have any number of nodes named **Handler**. These nodes work like a menu. The **Text** attribute is the text that will be displayed for the player to choose the different answers. If you look at the dialog with the name **KnightDialog** the knight is asking a yes or no question. The player can decide if they want to help or don't want to help so there are **Handler** entries for both **Yes** and **No**. The other attribute is **Actions**. This one is a little complicated to explain. The **Actions** attribute holds the methods you want to call if that option is selected. This will be done with the use of reflection.

I know that some of you might be trembling after reading that and think that reflection is a really hard thing to work with. Just using it to invoke methods is easy enough. The format of the value of the **Actions** attribute is as follows: "**methodName[:parameter [,parameter]] [;methodName[:parameter [,parameter]]]**". That might look a little complex but I will try my best to simplify it to help you understand it.

All actions must have at least one method associated with it. That is why it starts out with **methodName**. What **methodName** is the name of the method you want to call. Next there is a square bracket followed by a colon and the word **parameter** then another square bracket followed by a comma and the word **parameter**. What that means is if the method you are calling requires parameters you place a colon after the name of the method and list the parameters for the method separated by commas. Next there is another square bracket followed by a semicolon with the same thing ending with another square bracket. That means if you want to call more than one method you separate them with semicolons.

So, if you look at the handlers with their text being **Continue** they only call one method, the **StopDialog** method. These methods must be inside the **NPC** class to be used. In those methods you are

free to call methods in other classes, etc. The handlers that have **Yes** or **No** for their text call the method **StartDialog** with a parameter. The parameter is the dialog you want to start next. So if you are in the **KnightDialog** and the player chooses the **Yes** option the **StartDialog** method of the **NPC** class is called passing in the name of the dialog to start **ThankYou1**.

To the **DialogClasses** folder you will want to add a new class named **Script**. There will be several classes in this file. I will explain them after you have read the code. This is the code for the **Script** classes.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Reflection;
using System.Collections.ObjectModel;
using New2DRPG.SpriteClasses;

namespace New2DRPG.DialogClasses
{
    public class Script
    {
        Dictionary<string, Dialog> dialogs;

        public Script(params Dialog[] dialogs)
        {
            this.dialogs = new Dictionary<string, Dialog>();
            foreach (Dialog dialog in dialogs)
            {
                this.dialogs.Add(dialog.Name, dialog);
            }
        }

        public Dialog this[string name]
        {
            get { return dialogs[name]; ; }
        }
    }

    public class Dialog
    {
        public Collection<DialogHandler> handlers;
        string name;
        string text;

        public Dialog(string name, string text,
            params DialogHandler[] handlers)
        {
            this.handlers = new Collection<DialogHandler>();
            this.name = name;
            this.text = text;
            foreach (DialogHandler handler in handlers)
                this.handlers.Add(handler);
        }

        public int HandlerCount
        {
            get { return handlers.Count; }
        }
```

```
        public string Name
        {
            get { return name; }
        }

        public string Text
        {
            get { return text; }
        }

        public void InvokeHandler(NPC npc, int currentHandler)
        {
            handlers[currentHandler].Invoke(npc);
        }
    }

    public class DialogHandler
    {
        string caption;
        DialogAction[] actions;

        public DialogHandler(
           string caption,
           params DialogAction[] actions)
        {
            this.caption = caption;
            this.actions = actions;
        }

        public string Caption
        {
            get { return caption; }
        }

        public void Invoke(NPC npc)
        {
            foreach (DialogAction action in actions)
                action.Invoke(npc);
        }
    }

    public class DialogAction
    {
        MethodInfo method;
        object[] parameters;

        public DialogAction(string methodName, object[] parameters)
        {
            method = typeof(NPC).GetMethod(methodName);
            this.parameters = parameters;
        }

        public void Invoke(NPC npc)
        {
            method.Invoke(npc, parameters);
        }
    }
}
```

This class requires using statements for the **System.Reflection** , **System.Collections.ObjectModel** and **New2DRPG.SpriteClasses** namespaces. There are four classes in this class that are dependent upon one another: **Script**, **Dialog**, **DialogHandler** and **DialogAction**.

The class **Script** will hold all of the dialogs associated with the script as a dictionary with a string as the key and an object of type **Dialog** as the value. The constructor for the **Script** classes uses the params keyword with what looks like an array of **Dialog** objects. If you are unfamiliar with the params keyword you can use it to say that you are accepting 0 or more parameters of the type associated with it. You can only use the params keyword once inside of a method. The constructor then creates a new dictionary to work with it. In a foreach loop it loops through all of the dialogs passed to the constructor. It then adds the dialog using the name of the dialog as the key and the actual dialog as the value. There is an indexer for the **Script** class that uses a string. Indexers let you use a class as if it was an array. So in the game you can set a dialog by using the **Script** object and placing the name of the script in square brackets.

The next class is the **Dialog** class. This is the class that needed the **System.Collections.ObjectModel** namespace. It needed it for the **Collection** class. The **Collection** class should be used if you are using a collection to hold information about methods, assemblies, properties, etc. There is a field called **handlers** that is a **Collection<DialogHandler>**. There are also fields for the name of the dialog and the text in the dialog. The constructor for the **Dialog** class creates a new **Collection<DialogHandler>**, sets the **text** and **name** fields, and in a foreach loop it goes through all of the handlers for the conversation and adds them to the collection of handlers. There are get only properties to get the number of handlers in the collection, the name of the dialog and the text for the dialog. The method **InvokeHandler** will be used call the methods that were added in the dialogs in the scripts. It takes as a parameter the NPC the player is talking to and the currently selected handler.

The **DialogHandler** class holds the hand the handlers for the dialogs and is used to invoke the handlers using the **DialogAction** class. The **DialogHandler** class has two fields. The first is **caption** that will hold the text to display for the handler and the second is **actions** which is an array of **DialogAction** which will be used to actually call the appropriate method. The constructor takes as parameters the caption of the handler and the actions for the handler. The constructor then sets the fields. There is a get only property to get the caption of the dialog. There is also a method **Invoke** that takes as a parameter the **NPC** to call the method on. The method then in a foreach loop calls the **Invoke** method of the actions of the handler passing in the **NPC**.

The last class is the **DialogAction** class that actually will call the appropriate method in the **NPC**. This is the class that uses reflection. The field **MethodInfo** will hold the information about the method that you will call. The other field is an array of objects. This will hold the parameters to pass to the method. When the game is run and the method is called the array of objects will be cast to the proper type. To get the information of the method you use reflection as well. By getting the type of the class, using typeof, you can call the **GetMethod** method passing in the name of the method that you want to get the information about. If the method does not exist in the class you will get an exception. Since C# is case sensitive the method name must have the same case as in the string as in the class. The constructor of the class use the **GetMethod** method to get the information about the method name passed in. It also sets the array of parameters passed to the class.

The **Invoke** method is where the method is actually called. You use the **Invoke** method of the **MethodInfo** class to call the method you want to call passing in the class the method belongs to with the parameters for the method.

Now you need to implement the script classes. I will start with the **DialogComponent** class.

There were enough changes, and it is still short enough, that I will give you the entire code for the class and then go over the changes to the **DialogComponent** class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Media;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;
using New2DRPG.SpriteClasses;
using System.Text;

namespace New2DRPG.DialogClasses
{
    public class DialogComponent : Microsoft.Xna.Framework.DrawableGameComponent
    {
        SpriteBatch spriteBatch;
        SpriteFont spriteFont;
        ContentManager Content;
        Texture2D dialogTexture;
        Rectangle position;

        Color normal = Color.White;
        Color hilite = Color.Yellow;

        public Dialog dialog = null;
        public NPC npc = null;
        int currentHandler = 0;

        KeyboardState oldState, newState;

        public DialogComponent(Game game)
            : base(game)
        {
            spriteBatch =
                (SpriteBatch)Game.Services.GetService(typeof(SpriteBatch));
            Content =
                (ContentManager)Game.Services.GetService(typeof(ContentManager));
            LoadContent();
        }

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

        protected override void LoadContent()
        {
            base.LoadContent();
            dialogTexture = Content.Load<Texture2D>(@"GUI\conversationbox");
            position = new Rectangle(
                (TileEngine.ViewPortWidth - dialogTexture.Width) / 2,
                (TileEngine.ViewPortHeight - dialogTexture.Height) / 2,
```

```
                dialogTexture.Width,
                dialogTexture.Height);
            spriteFont = Content.Load<SpriteFont>(@"normal");
        }

        public override void Update(GameTime gameTime)
        {
            newState = Keyboard.GetState();

            if (dialog != null && npc != null)
            {
                if (CheckKey(Keys.Up))
                {
                    currentHandler--;
                    if (currentHandler < 0)
                        currentHandler = dialog.HandlerCount - 1;
                }
                if (CheckKey(Keys.Down))
                {
                    currentHandler++;
                    if (currentHandler == dialog.HandlerCount)
                        currentHandler = 0;
                }
                if (CheckKey(Keys.Space) || CheckKey(Keys.Enter))
                {
                    dialog.InvokeHandler(npc, currentHandler);
                    currentHandler = 0;
                }
            }
            base.Update(gameTime);

            oldState = newState;
        }

        private bool CheckKey(Keys theKey)
        {
            return oldState.IsKeyDown(theKey) && newState.IsKeyUp(theKey);
        }

        public override void Draw(GameTime gameTime)
        {
            base.Draw(gameTime);
            spriteBatch.Draw(dialogTexture, position, Color.White);

            if (dialog == null)
                return;

            string text = WrapText(dialog.Text);
            Vector2 textPosition = new Vector2(position.X + 10, position.Y + 10);
            spriteBatch.DrawString(spriteFont, text, textPosition, Color.White);

            int menuY = (int)spriteFont.MeasureString(text).Y +
spriteFont.LineSpacing;
            textPosition.Y = position.Y + menuY + 10;
            for (int i = 0; i < dialog.handlers.Count; i++)
            {
                Color textColor = normal;
                text = dialog.handlers[i].Caption;
                if (i == currentHandler)
```

```
                    textColor = hilite;
                spriteBatch.DrawString(spriteFont, text, textPosition, textColor);
                textPosition.Y += spriteFont.LineSpacing;
            }
        }

        private string WrapText(string text)
        {
            StringBuilder sb = new StringBuilder();

            float spaceWidth = spriteFont.MeasureString(" ").X;
            float length = 0f;

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

            foreach (string word in words)
            {
                float size = spriteFont.MeasureString(word).X;

                if (length + size < position.Width - 20)
                {
                    sb.Append(word + " ");
                    length += size + spaceWidth;
                }
                else
                {
                    sb.Append("\n" + word + " ");
                    length = size + spaceWidth;
                }
            }

            return sb.ToString();
        }

        public void Show()
        {
            Enabled = true;
            Visible = true;
        }

        public void Hide()
        {
            Enabled = false;
            Visible = false;
        }
    }
}
```

The first change was that I added a using statement for the **System.Text** namespace. That namespace has the **StringBuilder** class in it which I will use to have the text in the dialogs wrap around the conversation box. There are several new fields in the class. The two color fields will be used to draw the menu for the conversation. The **normal** field will draw the text that color if is not selected and the **hilite** field will draw the text that color if it is selected. The class has two public fields: **dialog** and **npc**. The **dialog** field will hold the active dialog and the **npc** field will hold the **NPC** that the player is talking to. There is a field **currentHandler** which will hold the handler that is currently selected. Since the user will be interacting with the component there are two **KeyboardState** fields: **newState** and **oldState** to hold the state of the keyboard. The constructor, **Initialize** and **LoadConent**

methods are the same as before.

The **Update** method handles selecting the player's response to the **NPC**. The method first gets the state of the keyboard into the **newState** field. It checks to make sure that both the **dialog** and **npc** fields are not null. If they were null and you tried to do something with them you would get an exception. The code inside the if statement will look very familiar. There is an if statement that calls the **CheckKey** method passing in **Keys.Up**. The **CheckKey** method works the same as you have seen in other classes. It was copied and pasted into this class. If it was pressed it subtracts one from the **currentHandler** field. If the field is less than zero it sets it to be the number of handlers minus one. This keeps the field in the bounds of the number of handlers when the player is scrolling up. It then checks to see if the down key was pressed. If it was it adds one to **currentHandler**, checks to see if it is equal to the number of handlers. If it is equal to that it sets **currentHandler** to be zero. It then checks to see if the space bar or enter keys have been pressed. If either of them have been pressed it calls the **InvokeHandler** method of the **Dialog** class passing in the **npc** and **currentHandler** fields. It then sets **currentHandler** to zero so that if there were more handlers in the previous conversation than the present conversation an exception will not be thrown. It then sets the **oldState** field to the **newState** field. The **CheckKey** method should be familiar to you by now.

The **Draw** method will draw the image for the dialog, the text of the dialog and the text of the handlers for the dialog. The **Draw** method first calls **base.Draw** to call the **Draw** method of the other classes to draw the rest of the game. It then draws the image for the dialog to appear in. The **Draw** method then checks to make sure that there is actually a dialog to display. If there isn't it exits the method. I then set a local string variable **text** to be the return of a method I wrote called **WrapText** that will have the text go to the next line if adding another word will make it go off the image. I then create a **Vector2** called **textPosition** to hold the position where I will draw the text. I set it initially to be the position of the image plus 10 pixels for the X and Y values of the **Vector2**. This is because there is a border around the image and it has rounded corners. After drawing the text there is a local variable called **menuY** that is set to be the **Y** component of the size of **text** plus the **LineSpacing** property of the font adding a blank line. I then set the **Y** value of **textPosition** to be the **Y** value of the image plus **menuY** plus 10 to add the padding for the border of the image. In a for loop I loop through all of the handlers for the dialog. I then loop I set **textColor** to be **normal**. I set **text** to be the **Caption** property of the handler at the appropriate index. If **currentHandler** is the value of the index of the loop I then set **textColor** to be **hilite** so the text will be drawn with that color. I then draw the text using the **textColor** variable for the color to draw the text. I then add the **LineSpacing** property of the font to the **Y** value of **textPosition** to have the next handler appear on the next line.

That leaves the **WrapText, Show** and **Hide** methods for this class. The **Show** and **Hide** methods are the same as before. The **WrapText** method will have the dialog continue to the next line if adding the word will have the text go outside of the image for the dialog. I create a new instance of the **StringBuilder** class. The reason I used **StringBuilder** instead of just a string is if you are doing a lot of manipulation of a string like this you are creating and releasing a lot of strings. You only have to create the one instance of **StringBuilder**. I will be using the width of a space a lot in the method so I have a variable that holds the width of a space using the **MeasureString** method of the **SpriteFont** class. I also need to know the length of the current line so there is a variable **length** that will hold the length of the current line. I then create an array of strings called **words** and use the **Split** method of the string class passing in a space on what I want to split the string on. This gives us all of the words in the text. In a foreach loop I go through all of the words in the array. I get the width of the word into the variable **size** using the **X** value of the **MeasureString** method of the **SpriteFont** class passing in the word. I then compare the length of the current line plus the size of the word to the width of the image minus 20

pixels. The reason I subtracted 20 pixels is that there is a border around the image. If the length plus the size of the word is less than that it is safe to add that word to the current line. I use the **Append** method of the string builder class passing in the word plus a space and add the size of the word plus the size of a space to the text. If the comparison failed the word belongs on the next line. To do that I append **\n** plus the word plus a space and set the length to the size of the word plus the size of the space. If you are familiar with the **WriteLine** method of the **Console** class you know that **\n** will have what you are writing go onto the next line. The **DrawString** method of the **SpriteBatch** class has that same

functionality. You can use **\n** to have the text you are drawing go to the next line with the same **Y** value as the starting position of the text.

You also need to change the **NPC** class to implement the dialog classes. There were enough changes that I will give you the code for the entire class. This is the new code for the **NPC** class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using New2DRPG.DialogClasses;

namespace New2DRPG.SpriteClasses
{
    public class NPC : AnimatedSprite
    {
        DialogComponent dialog;
        Script script;
        float speakingRadius;
        bool isTalking;
        string dialogName;

        public NPC(Game game,
            Texture2D texture,
            List<Animation> animations,
            DialogComponent dialog,
            Script script)
            : base(game, texture, animations)
        {
            speakingRadius = 80f;
            this.dialog = dialog;
            this.script = script;
            dialog.Hide();
        }

        public bool IsTalking
        {
            get { return isTalking; }
        }

        public float SpeakingRadius
        {
            get { return speakingRadius; }
        }

        public string DialogName
        {
            get { return dialogName; }
            set { dialogName = value; }
        }

        public override void Update(GameTime gameTime)
        {
                base.Update(gameTime);
                if (dialog.Enabled)
                    dialog.Update(gameTime);
```

```
        }

        public override void Draw(GameTime gameTime)
        {
            base.Draw(gameTime);
            if (dialog.Enabled)
                dialog.Draw(gameTime);
        }

        public void StartDialog(string dialogname)
        {
            if (script == null || dialog == null)
                return;
            dialog.npc = this;
            dialog.dialog = script[dialogname];
            isTalking = true;
        }

        public void StopDialog()
        {
            if (dialog == null)
                return;

            dialog.Hide();
            isTalking = false;
        }
    }
}
```

I added in two fields to the class. The first is a **Script** object that will hold the script associated with the **NPC**. The second field is a string called **dialogName** and it will be used to set the name of the dialog associated with the **NPC**. The constructor now takes five parameters: a **Game** object, a **Texture2D**, a **List<Animation>**, a **DialogComponent** and a **Script**. The constructor then sets the **speakingRadius** field, the **dialog** field, the **script** field and calls the **Hide** method of the **dialog** field. I added in a property to get and set the **dialogName** field.

The **Update** and **Draw** methods are the same as before. The **Update** method will call the **Update** method of the **dialog** field if it is active. The same is true for the **Draw** method calling the **Draw** method of the **dialog** field. The **StartDialog** method now takes a string as a parameter **dialogname**, the name of the dialog you want to start. If either the **script** field or the **dialog** field are null the dialog can not be started and the method exits. The method then sets the **npc** field of the **DialogComponent** to be the current **NPC** using the **this** keyword. It sets the **dialog** field of the **DialogComponent** using the indexer of the **Script** class and sets the **isTalking** field to true. The **StopDialog** method checks to see if **dialog** is null. If it is it just exits the method. It then calls the **Hide** method of the **DialogComponent** and sets **isTalking** to be false.

All of the other changes were in the **Game1** class. If you look at the code that I've written for that class I have added in **#region** preprocessor directives to separate the code into regions. The reason is that the entire class is getting a little long and it is hard to find things in the class. Separating it into regions helps to organize the code and make it easier to find things. For example, I have a region called **Content Methods** that hold all of the methods relating to loading content into the game. On the left beside the **#region** directive is a little plus sign. If you click that plus sign it will collapse the region. You only have to open it if you are going to be working with methods in that region. That is just a little organizational tip for you. I would have liked to have written a tutorial on adding regions to your code

but that wouldn't be very useful tutorial as it wouldn't add anything to the game.

The first thing that I did was add a few fields to the class. There is a field that is an array of strings that holds the dialogs for the **NPCs**. The dialogs are in the same order as the asset names. The first dialog is **KnightDialog** and the first asset is **knt1**. The other dialogs are **NinjaDialog** and **SkeletonDialog** and the assets are **nja1** and **skl1**. There is also a **DialogComponent** field and a **Script** field. I've also added all of the using statements that you will need in the **Game1** class as well.

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Media;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;
using New2DRPG.CoreComponents;
using New2DRPG.SpriteClasses;
using New2DRPG.DialogClasses;
using System.Xml;
using System.Reflection;
using System.Collections.ObjectModel;

DialogComponent dialog;
Script script;

string[] dialogNames =
    {
        "KnightDialog",
        "NinjaDialog",
        "SkeletonDialog" };
```

To use the dialogs you need to read them in. What I did is in the **LoadContent** method is after adding the **SpriteBatch** and **ContentManager** to the list of services for the game is create a new instance of the **DialogComponent** and add it to the list of components for the game. This way you don't have to call the **Update** or **Draw** methods of the component. Also just before calling **CreateNPC** I call the method **ReadScript** passing in the the path to the script which is **Content\script1.script**. The **ReadScript** method has a helper method **ReadHandlers** that will read in the handlers for the script. I will give you the new code for the **LoadContent** method and the **ReadScript** and **ReadHandlers** methods. I will explain the new methods after you have read them.

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    tileSpriteBatch = new SpriteBatch(GraphicsDevice);

    Services.AddService(typeof(SpriteBatch), spriteBatch);
    Services.AddService(typeof(ContentManager), Content);

    dialog = new DialogComponent(this);
    Components.Add(dialog);
```

```
    normalFont = Content.Load<SpriteFont>("normal");
    LoadCreatePCScreen();
    LoadStartScreen();
    LoadHelpScreen();
    LoadActionScreen();
    LoadQuitPopUpScreen();
    LoadGenderPopUpScreen();
    LoadClassPopUpScreen();
    LoadDifficultyPopUpScreen();
    LoadNameInputScreen();
    LoadCreditScreen();
    LoadIntroScreen();
    LoadViewCharacterScreen();
    LoadQuitActionScreen();

    creditScreen.Hide();
    startScreen.Hide();
    helpScreen.Hide();
    createPCScreen.Hide();

    activeScreen = introScreen;
    activeScreen.Show();

    List<Animation> animations = new List<Animation>();

    Animation tempAnimation = new Animation(2, 64, 64, 0, 0);
    animations.Add(tempAnimation);

    tempAnimation = new Animation(2, 64, 64, 128, 0);
    animations.Add(tempAnimation);

    tempAnimation = new Animation(2, 64, 64, 256, 0);
    animations.Add(tempAnimation);

    tempAnimation = new Animation(2, 64, 64, 384, 0);
    animations.Add(tempAnimation);

    Texture2D playerSpriteTexture = Content.Load<Texture2D>(@"Sprites\amg1large");

    playerSprite = new AnimatedSprite(this, playerSpriteTexture, animations);
    playerSprite.CurrentAnimation = AnimationKey.Down;
    playerSprite.IsAnimating = true;

    spriteTextures = new Texture2D[assetNames.Length];
    for (int i = 0; i < assetNames.Length; i++)
        spriteTextures[i] = Content.Load<Texture2D>(assetNames[i]);
    script = ReadScript(@"Content\script1.script");
    CreateNPCS();
}

private Script ReadScript(string scriptName)
{
    Script newScript;
    XmlDocument input = new XmlDocument();
    input.Load(scriptName);

    XmlNodeList dialogs = input.GetElementsByTagName("Dialog");

    Collection<Dialog> newDialogs = new Collection<Dialog>();
```

```
    foreach (XmlNode node in dialogs)
    {
        string name = "";
        string text = "";
        List<DialogHandler> handlers = new List<DialogHandler>();
        Dialog newDialog;
        foreach (XmlNode innerNode in node.ChildNodes)
        {
            if (innerNode.Name == "Name")
                name = innerNode.Attributes["Text"].Value;
            if (innerNode.Name == "Text")
                text = innerNode.InnerText;
            if (innerNode.Name == "Handlers")
            {
                handlers = ReadHandlers(innerNode);
            }
        }
        newDialog = new Dialog(name, text, handlers.ToArray());
        newDialogs.Add(newDialog);
    }

    Dialog[] dialogsToAdd = new Dialog[newDialogs.Count];

    for (int i = 0; i < newDialogs.Count; i++)
        dialogsToAdd[i] = newDialogs[i];

    newScript = new Script(dialogsToAdd);
    return newScript;
}

private List<DialogHandler> ReadHandlers(XmlNode innerNode)
{
    List<DialogHandler> handlers = new List<DialogHandler>();
    foreach (XmlNode node in innerNode)
    {
        string text = "";
        string actions = "";
        if (node.Name == "Handler")
        {
            text = node.Attributes["Text"].Value;
            actions = node.Attributes["Actions"].Value;
            List<DialogAction> dialogActions = new List<DialogAction>();
            string[] methods = actions.Split(';');
            string methodName = "";
            object[] parameters = null;
            foreach (string m in methods)
            {
                if (m.Contains(':'))
                {
                    string[] newActions = m.Split(':');
                    methodName = newActions[0];
                    parameters = (object[])newActions[1].Split(',');
                }
                else
                {
                    methodName = m;
                    parameters = null;
                }
                dialogActions.Add(new DialogAction(methodName, parameters));
```

```
                }
                DialogHandler handler = new DialogHandler(text,
                    (DialogAction[])dialogActions.ToArray());
                handlers.Add(handler);
            }
        }
        return handlers;
    }
```

The **ReadScript** method returns a **Script** object. The reason I did that was when I add the **Content Pipeline** classes for the script you can just replace **ReadScript** with **Content.Load<Script>** in your code and modify the parameter so that it refers to the script for your game. There is a **Script** variable that will hold the script and there is an **XmlDocument** variable named **input**. I named it input so it will be one less change when I add the **Content Pipeline** class for the scripts.

I use the **Load** method of the **XmlDocument** class to read the script into **input**. There is then a **XmlNodeList** variable that will hold all of the nodes that you are interested in processing. I use **GetElementsByTagName** which will get all of the nodes in the document with that have the name passed in. I passed in **Dialog** because I wanted all of the dialog nodes in the document. There is a **Collection<Dialogs>** called **newDialogs** that will hold all of the dialogs. Then in a foreach loop I loop through all of the dialog nodes to process them.

In the foreach loop there are four variables: **name**, **text**, **handlers** and **newDialog**. The first two will hold the name and text of the conversation respectively. **handlers** is a **List<DialogHandler>** that will hold the handlers for the dialog. **newDialog** will hold the actual dialog after it is processed.

There is another foreach loop that will loop through all of the child nodes in the current **Dialog** node. Inside that loop there are three if statements. The first if statement checks to see if the name of the current node is **Name**. If it is it then sets the **name** variable to the value of the **Text** attribute. The second if statement checks to see if the name of the current node is **Text**. If it is it sets the **text** variable to be the inner text of the node. Finally if the name of the current node is **Handlers** it sets **handlers** to be the return of the **ReadHandlers** method passing in the current node. When the loop is finished it creates a new **Dialog** object and adds it to the list of dialogs.

Once the loops have finished there is an array of **Dialog** called **dialogsToAdd**. It is set to the number of dialogs in the **newDialogs** variable. I will pass this array to the constructor of the **Script** class. To fill the array I use a for loop to loop through all of the dialogs in **newDialogs**. I then set the value of the array to the appropriate value in **newDialogs**. I then create a new script object and return it.

The **ReadHandlers** method returns a **List<DialogHandler>** which will be all of the handlers for the dialog. It first creates a **List<DialogHandler>** to hold of the dialog handlers for the dialog. Then in a foreach loop it loops through all of the dialogs.

Inside the foreach loop there are two variables **text** and **actions**. **text** will hold the text for the handler and **actions** will hold the actions for the handler. There is an if statement that checks to make sure the current node is a **Handler** node. If it is I set **text** to be the value of the **Text** attribute. I also set **action** to be the value of the **Actions** attribute. I then create a **List<DialogAction>** to hold the actions for the handler.

What I do now is parse the **action** string. Parsing is taking an object and breaking it up into its

components and it can be a complicated topic. When I was talking about the format of a handler I said that handlers would be separated by semicolons. To get all of the actions from the string I first call the **Split** method passing in a semicolon to get an array with all of the methods in the string. There is next a variable **methodName** that will hold the name of the method and an array of **object** called **parameters** that will hold the parameters for the method.

In a foreach loop I go through all of the methods that were returned from the call to **Split**. I then check to see if the method contains a colon using **Contains(":")**. If it contains a colon then there are parameters associated with the method. I then call the **Split** method passing in a colon to split the string to be first the name of the method and second the parameters of the method. I set **methodName** to be the first element in the array and I again called **Split** but this time on the second element in the array passing in a comma because the parameters for the action are separated by commas. If the string did not contain a colon then it is just the name of a method so I set **methodName** to the string and **parameters** to null.

Outside of the if statement I create a new **DialogAction** object and add it to the list of actions to be preformed. Then outside of the inner foreach loop I create a new **DialogHandler** passing in **text** and casting the return value of the **ToArray** method of the **List** class to an array of **DialogAction**. I then add the handler to **List<DialogHandler>**. At the end of the method I return **handler**, the **List<DialogHandler>**.

I also had to change the **CreateNPCS** method because the constructor now takes a **DialogComponent** object and a **Script** object. I also set the dialog for the NPC based on the index of the array. I do that right after creating the NPC. This is the code for that method.

```
private void CreateNPCS()
{
    List<Animation> listAnimation = new List<Animation>();

    Animation tempAnimation = new Animation(2, 64, 64, 0, 0);
    listAnimation.Add(tempAnimation);

    tempAnimation = new Animation(2, 64, 64, 128, 0);
    listAnimation.Add(tempAnimation);

    tempAnimation = new Animation(2, 64, 64, 256, 0);
    listAnimation.Add(tempAnimation);

    tempAnimation = new Animation(2, 64, 64, 384, 0);
    listAnimation.Add(tempAnimation);

    for (int i = 0; i < assetNames.Length; i++)
    {
        List<Animation> animations = new List<Animation>();

        foreach (Animation a in listAnimation)
        {
            Animation clonedAnimation = (Animation)a.Clone();
            animations.Add(clonedAnimation);
        }

        npcs.Add(
            new NPC(this,
                spriteTextures[i],
```

```
                animations,
                dialog,
                script));

            npcs[i].DialogName = dialogNames[i];
            npcs[i].IsAnimating = true;
            int direction = random.Next(0, 4);

            switch (direction)
            {
                case 0:
                    npcs[i].CurrentAnimation = AnimationKey.Up;
                    break;
                case 1:
                    npcs[i].CurrentAnimation = AnimationKey.Down;
                    break;
                case 2:
                    npcs[i].CurrentAnimation = AnimationKey.Left;
                    break;
                case 3:
                    npcs[i].CurrentAnimation = AnimationKey.Right;
                    break;
            }

            Vector2 position = new Vector2();

            position.X = TileEngine.TileWidth * random.Next(1, 10);
            position.Y = TileEngine.TileHeight * random.Next(1, 10);

            npcs[i].Position = position;
        }
    }
```

Now you have to actually have the NPC use the dialogs. All of that will happen in the **HandlePlayerInput** method. It was basically rewritten so I will explain it after you have read it. This is the new code for the **HandlePlayerInput** method.

```
private void HandlePlayerInput(GameTime gameTime)
{
    player.Update(gameTime);
    if (!inDialog)
    {
        foreach (NPC npc in npcs)
            npc.Update(gameTime);
        if (CheckKey(Keys.Space))
        {
            foreach (NPC npc in npcs)
            {
                float distance = Vector2.Distance(
                    npc.Origin,
                    playerSprite.Origin);
                if (distance < npc.SpeakingRadius)
                {
                    dialog.Show();
                    npc.StartDialog(npc.DialogName);
                    inDialog = true;
                    break;
                }
            }
```

```
    }
}
if (!inDialog)
{
    Vector2 motion = new Vector2();
    playerSprite.IsAnimating = true;
    if (newState.IsKeyDown(Keys.Up) || newState.IsKeyDown(Keys.NumPad8))
    {
        motion.Y--;
        playerSprite.CurrentAnimation = AnimationKey.Up;
    }
    if (newState.IsKeyDown(Keys.Down) || newState.IsKeyDown(Keys.NumPad2))
    {
        motion.Y++;
        playerSprite.CurrentAnimation = AnimationKey.Down;
    }
    if (newState.IsKeyDown(Keys.Right) || newState.IsKeyDown(Keys.NumPad6))
    {
        motion.X++;
        playerSprite.CurrentAnimation = AnimationKey.Right;
    }
    if (newState.IsKeyDown(Keys.Left) || newState.IsKeyDown(Keys.NumPad4))
    {
        motion.X--;
        playerSprite.CurrentAnimation = AnimationKey.Left;
    }
    if (newState.IsKeyDown(Keys.NumPad9))
    {
        motion.X++;
        motion.Y--;
        playerSprite.CurrentAnimation = AnimationKey.Right;
    }
    if (newState.IsKeyDown(Keys.NumPad3))
    {
        motion.X++;
        motion.Y++;
        playerSprite.CurrentAnimation = AnimationKey.Right;
    }
    if (newState.IsKeyDown(Keys.NumPad1))
    {
        motion.X--;
        motion.Y++;
        playerSprite.CurrentAnimation = AnimationKey.Left;
    }
    if (newState.IsKeyDown(Keys.NumPad7))
    {
        motion.X--;
        motion.Y--;
        playerSprite.CurrentAnimation = AnimationKey.Left;
    }
    if (motion != Vector2.Zero)
    {
        motion.Normalize();
        motion *= playerSprite.Speed;

        if (!actionScreen.CheckUnWalkableTile(playerSprite.Bounds, motion))
            playerSprite.Position += motion;

    }
```

```
            else
            {
                playerSprite.IsAnimating = false;
            }

            playerSprite.LockToMap();

            camera.LockToSprite(playerSprite);
            camera.LockCamera();
        }
    }
```

The first thing I do is check to see if the game is not in a dialog already. If it isn't in a dialog I call the **Update** method for all of the **NPCs** in the game. I then check to see if the space key was pressed. If it was I get the distance between the player and the NPC and check to see if the player is with in speaking radius of the NPC. If it is I call the **Show** method of the **DialogComponent** to show it and call the **StartDialog** method of the **NPC** passing in the **DialogName** property of the **NPC** because the **StartDialog** method requires a string parameter that is the name of the dialog you want to start.

If you don't do the following step when the player is in a dialog the sprite for the player will be able to wander all over the place and you don't want that. To stop that I again check to make sure that no dialog is in progress. If there is no dialog in progress it is okay to check for the movement of the player's sprite in the game.

There is just one more thing that you will need to do. You need to have a way to return the game back to the state where it is not in a dialog. The easiest way is to do it in the **Update** method. The **DialogComponent** has an **Enabled** property that you can check to see if it is currently active. What you can do is in the **Update** method, near the top, is check to see if the **DialogComponent** is currently enabled. If it is not you can set **inDialog** to false so the game can continue. This is the updated code for the **Update** method.

```
protected override void Update(GameTime gameTime)
{
    newState = Keyboard.GetState();

    if (!dialog.Enabled)
        inDialog = false;

    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    if (activeScreen == startScreen)
    {
        HandleStartScreenInput();
    }
    else if (activeScreen == helpScreen)
    {
        HandleHelpScreenInput();
    }
    else if (activeScreen == createPCScreen)
    {
        HandleCreatePCScreenInput();
    }
    else if (activeScreen == quitPopUpScreen)
    {
        HandleQuitPopUpScreenInput();
```

```
    }
    else if (activeScreen == genderPopUpScreen)
    {
        HandleGenderPopUpScreenInput();
    }
    else if (activeScreen == classPopUpScreen)
    {
        HandleClassPopUpScreenInput();
    }
    else if (activeScreen == difficultyPopUpScreen)
    {
        HandleDifficultyPopUpScreenInput();
    }
    else if (activeScreen == nameInputScreen)
    {
        HandleNameInputScreenInput();
    }
    else if (activeScreen == introScreen)
    {
        HandleIntroScreenInput();
    }
    else if (activeScreen == creditScreen)
    {
        HandleCreditScreenInput();
    }
    else if (activeScreen == actionScreen)
    {
        HandleActionScreeenInput();
        HandlePlayerInput(gameTime);
    }
    else if (activeScreen == viewCharacterScreen)
    {
        HandleViewCharacterScreenInput();
    }
    else if (activeScreen == quitActionScreen)
    {
        HandleQuitActionScreenInput();
    }
    oldState = newState;

    base.Update(gameTime);
}
```

Well, that was a bit of a long tutorial. I will be starting the next part of Eyes of the Dragon shortly. What I am thinking of adding is getting started with the combat system for the game. I encourage you to keep either visiting my site http://xna.jtmbooks.com or my blog, http://xna-rpg.blogspot.com for the latest news on my tutorials.