A Speech Library Helper for Cortana

Rob Miles

Using Speech in a Windows Phone application is quite easy. But it could be easier. You have to create the grammar file and then hook it up to your program code. Being an inventive soul I've made a little helper library. This has been created for use in an adventure which uses traditional two word commands:

```
Go west
Take chainsaw
Use chainsaw
```

The helper library lets you create as many verb-noun combinations as you like, and automatically builds and installs the Voice Commands file for you. You can use it to build an adventure, but you could also use it in any situation where you want to implement simple command processing using voice input.

Getting Started

The best way to get started is to load up the AdventureDemo project and take a look at how it works. It contains the AdventureEngine class along with a sample adventure which does very little. The project also contains customised code in App.xaml.cs which sets up the speech command file and decodes and despatches spoken commands. There is also code in MainPage.xaml.cs that performs the voice commands. You can just copy these files into your project.

Creating your own Adventure

The GameDesign class is the base class for your adventures. It serves as the container for the adventure commands and provides the methods that will generate the XML voice command files. It also provides the basis of adventure save and load behaviours. As supplied it just stores the text of all the previous commands and responses but you can extend the load and save pattern to include saving the position of the player and the state of game objects.

Creating an Adventure class of your own

You create your own adventure by extending the GameDesign class:

```
public class ScaryFairgroundGame : GameDesign
{
}
```

I've made a very simple game which I've called "ScaryFairgroundGame". You can make yours as complicated as you like. Once you've made your class you can give it a constructor. This sets up the voice commands for the game:

```
public ScaryFairgroundGame()
{
    Name = "Scary Fairground";                    // adventure name
    VoiceCommandPrefix = "Fairground";            // Cortana trigger word
    VoiceCommandExample = "Fairground";           // Cortana example word
```

```
    VoiceCommands.Add(new VoiceCommandNoOptions(
        new string[] { "Look", "View" },            // command names
        "MainPage.xaml",                            // page to navigate to
        new VoiceCommandDespatcherDelegate(Look),   // method to call
        "Looking"));                                // spoken feedback

    VoiceCommands.Add(new VoiceCommandWithOptions(
        new string[] { "Move", "Go" },              // command names
        "MainPage.xaml",                            // page to navigate to
        new VoiceCommandDespatcherDelegate(Go),     // method to call
        "Going",                                    // spoken feedback
        "direction",                                // phraselist name
        new string[] { "North", "South", "East",    // phraselist options
            "West", "Up", "Down" }));
}
```

This constructor sets the name of the game and the Voice Command prefix and example strings. These are used when the Voice Command file is built. In the code above the player must say "Fairground" when they want to give a game command, for example if they want to go east they will say:

```
Fairground Go East
```

The Example string should be the same text, this is what Cortana will show as help to the user.

Setting up Adventure Commands

The next thing you need to do is set up the commands that you want the adventure to use. There are two kinds of commands, one is a single command with no options, for example "Look", and the other command is followed by an option, for example "Go West". In the sample code above you can see how I've set these two commands up.

```
    VoiceCommands.Add(new VoiceCommandNoOptions(
        new string[] { "Look", "View" },            // command names
        "MainPage.xaml",                            // page to navigate to
        new VoiceCommandDespatcherDelegate(Look),   // method to call
        "Looking"));                                // spoken feedback
```

This call adds a Look command by making an instance of VoiceCommandNoOptions. When the user gives the look command the program will navigate to the MainPage page and then call the method Look, which is a member of the ScaryFairground class:

```
string Look(string option)
{
    return "You are in a Scary Fairground. Ooh look, a clown with a chainsaw.";
}
```

Each time the user speaks the Look command this method will be called automatically. Look is a simple command which doesn't have any options. The action methods for simple commands are called with an option string which is empty.

```
    VoiceCommands.Add(new VoiceCommandWithOptions(
        new string[] { "Move", "Go" },              // command names
        "MainPage.xaml",                            // page to navigate to
        new VoiceCommandDespatcherDelegate(Move),   // method to call
        "Going",                                    // spoken feedback
        "direction",                                // phraselist name
        new string[] { "North", "South", "East",    // phraselist options
```

```
            "West", "Up", "Down" }));
```

The framework also supports two word operations, where the first word is the command and the second the option for that command. For example the Move command is followed by one of a number of possible directions which are held in the VoiceCommandWithOptions instance that is created to represent the command.

```
string Move(string option)
{
    return "There is no way to travel " + option + ". Yet.";
}
```

When the method for the Move command is called it is passed the direction that was given by the user. You can use this to create any number of two-word commands in this format.

Creating the Voice Command file

The VoiceCommands list in the GameDesign class holds all the commands that have been created. These are automatically stored in a file and then loaded as voice commands when the program starts. You don't have to do anything else. The GameDesign class provides a method that will generate a voice command file for a number of language options. The voice command file is actually generated in the App.xaml.cs file when an application is launched. Take a look for the method setupVoiceCommands if you are interested in how it all works. The language locale for the voice commands is set to the current language locale for the phone, so the commands should always work correctly.

Decoding the Voice commands

The spoken commands are decoded in the App.xaml.cs file. This also automatically determines the destination page to be activated when the program is activated by a voice command.

Using Multiple Pages in your Adventure

If your adventure all happens on the MainPage of your application then you don't have to do anything to the program to make it work. But if you have different pages (perhaps you have an "inventory" page and a "shop" page) you need to configure a lookup table in the program so that these can be automatically located:

```
Type[] destinationFrameTypesValue = null;

Type[] destinationFrameTypes
{
    get
    {
        if (destinationFrameTypesValue == null)
            destinationFrameTypesValue = new Type[] {
            typeof(MainPage) };  // add types for your pages here
        return destinationFrameTypesValue;
    }
}
```

This code is in App.xaml.cs. If you want your voice commands to deliver the user to different pages you need to add the types of these pages to the list that is created. Otherwise the program will always display the MainPage when the user starts it from a voice command.

Decoding the Commands

The GameDesign class provides a method that will decode and act on game commands issued by the player. The method is used in the OnNavigatedTo method for the MainPage in the sample application. The method loads the game status, performs the command and then saves the game status. It is important that your game works in this way, since you can't make any assumptions about when the game might be in a position to save data.

If the program was started from a voice command it will also speak the response for the game. The GameDesign class can also decode commands which are typed in from the keyboard. When the user types commands it is possible that they might give invalid ones, and so your game needs to provide a method that can deliver a sensible response in this situation.

```
public override string InvalidCommand(string option)
{
    return "I don't know how to do that.";
}
```

Note that there is a potential problem here, in that if the user tries to speak the name of an item which is not in a phraselist the result is that Cortana will ignore the command completely. There is no way that your game can react to a spoken command that contains invalid content, since it never gets control in that situation. The way to solve this is to make sure that users know the existence of an item before they are required to use it, otherwise their failed guesses will not start the program running.

Rob Miles
www.robmiles.com
April 2015