Gesture Based UI Development Project

Conor Rabbitte - G00365218 - Github

April 2021

Contents

1 Purpose of the Application **1**

2 Gestures identified as appropriate for this application **4**
2.1 Menu Grammer 4
2.2 Basic Grammar 6
2.3 Password Grammar 8

3 Hardware used in creating the application **10**

4 Architecture for the Solution **10**

5 Conclusions Recommendations **11**

1 Purpose of the Application

In this project I have created a simple 2D platformer game entitled *King and Pigs*. When loaded the game presents a simple main menu (see Fig 1.1) containing a background image, a title, three buttons for navigation and a theme song. When the player selects the 'Play Game' button they will be taken to the first level in the game.
***Note: Github is in title above is hyperlink*

Fig 1.1

Level 1 is loaded in and a new Heads-Up Display (HUD) appears. In the top-right corner the players health is displayed as 3 hearts on a banner. In the top-left corner is the score. This represents the number of diamonds the player has collect in current level. The top-middle consists of four X's (**X X X X**) which represent a password the player will need to collect in order to complete the level. The password consists of four letters that make up the name of a fruit. Each letter of the password is hidden within all the diamonds found throughout the level. The player breaks open the diamonds to reveal the letter hidden, which then updates one of the four X's with that letter and its order within the password. Once the player collects and completes the password they can approach the exit door (the door furthest away from their starting location). They can speak the phrase "The password is **X X X X**", where **X** represent the needed password.

Fig 1.2

Levels 2 and 3 contain the same HUD, however the level design and password are different. (see Fig 1.3 & 1.4)

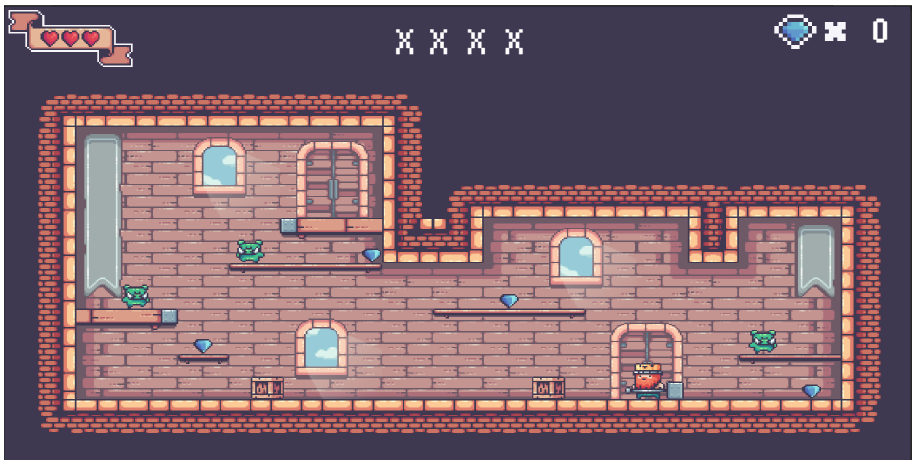


Fig 1.3

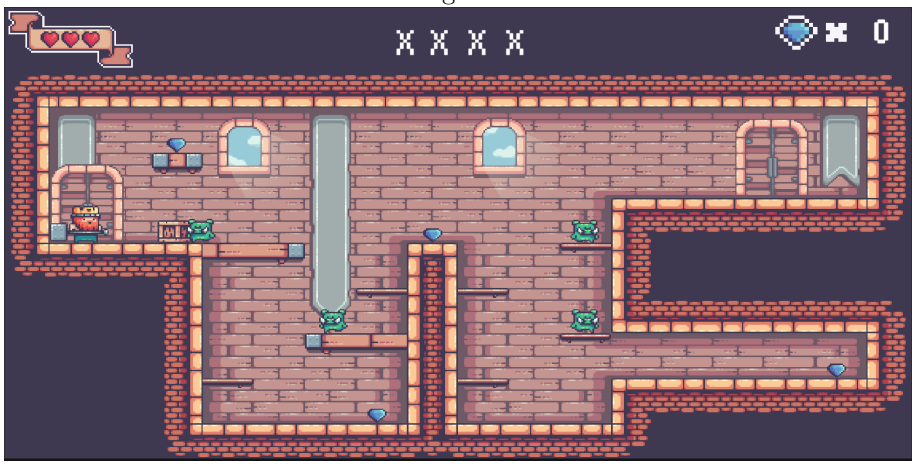


Fig 1.4

The idea of the game was to have a design that allows the player to optionally use voice for navigation and control but not be dependent on them. However, in the design the progression of the player does depend on them using the voice gestures to speak the levels password to the door in order to proceed. This was a gameplay design choice to incorporate a more complex system then simple controls.

Finally, when the player either dies or completes the 3 levels they will be presented with a simple Game Over screen and can click on a button which will take then back to the main men (see Fig 1.5).

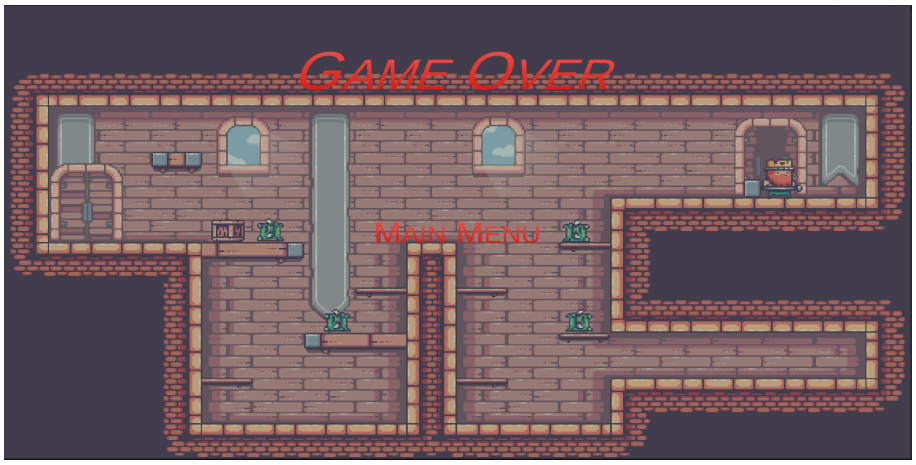


Fig 1.5

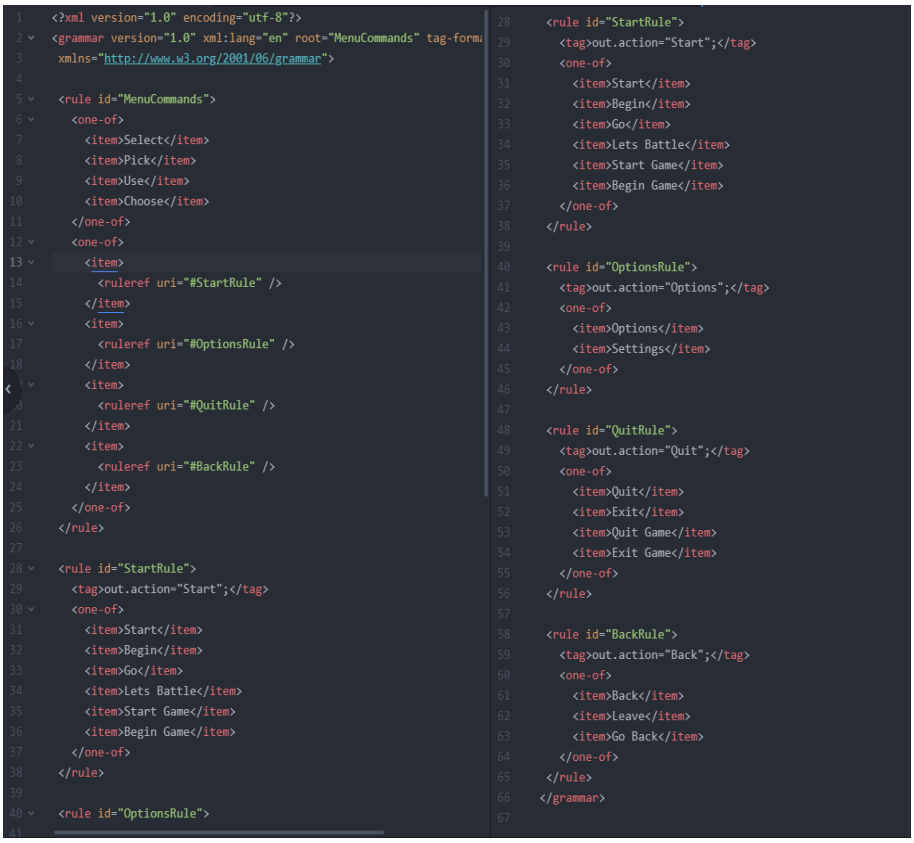
2 Gestures identified as appropriate for this application

The voice controls are split into three different scripts with each script performing its own tasks.

2.1 Menu Grammer

The MenuGrammar script controls the main menu voice operations (see Fig 2.1). This file consists of five different rules for speech recognition. The rules *StartRule, OptionsRule, QuitRule* all correspond to the three options given to the player at the main menu screen. The last rule the *BackRule* is used when the player is in the options menu to return to the previous menu. These are all encapsulated with the MenuCommands rule which contains the rulerefs for each of the aforementioned rules.

- **Start Game** - 'Choose Begin', 'Select Start Game'
- **Options** - 'Pick Options', 'Select Settings'
- **Quit Game** - 'Use Quit', 'Pick Exit Game'
- **Back (Inside Options)** - 'Select Leave', 'Choose Back'



```
<?xml version="1.0" encoding="utf-8"?>
<grammar version="1.0" xml:lang="en" root="MenuCommands" tag-forma
 xmlns="http://www.w3.org/2001/06/grammar">

 <rule id="MenuCommands">
   <one-of>
     <item>Select</item>
     <item>Pick</item>
     <item>Use</item>
     <item>Choose</item>
   </one-of>
   <one-of>
     <item>
       <ruleref uri="#StartRule" />
     </item>
     <item>
       <ruleref uri="#OptionsRule" />
     </item>
     <item>
       <ruleref uri="#QuitRule" />
     </item>
     <item>
       <ruleref uri="#BackRule" />
     </item>
   </one-of>
 </rule>

 <rule id="StartRule">
   <tag>out.action="Start";</tag>
   <one-of>
     <item>Start</item>
     <item>Begin</item>
     <item>Go</item>
     <item>Lets Battle</item>
     <item>Start Game</item>
     <item>Begin Game</item>
   </one-of>
 </rule>

 <rule id="OptionsRule">
   <rule id="StartRule">
     <tag>out.action="Start";</tag>
     <one-of>
       <item>Start</item>
       <item>Begin</item>
       <item>Go</item>
       <item>Lets Battle</item>
       <item>Start Game</item>
       <item>Begin Game</item>
     </one-of>
   </rule>

   <rule id="OptionsRule">
     <tag>out.action="Options";</tag>
     <one-of>
       <item>Options</item>
       <item>Settings</item>
     </one-of>
   </rule>

   <rule id="QuitRule">
     <tag>out.action="Quit";</tag>
     <one-of>
       <item>Quit</item>
       <item>Exit</item>
       <item>Quit Game</item>
       <item>Exit Game</item>
     </one-of>
   </rule>

   <rule id="BackRule">
     <tag>out.action="Back";</tag>
     <one-of>
       <item>Back</item>
       <item>Leave</item>
       <item>Go Back</item>
     </one-of>
   </rule>
 </grammar>
```

Fig 2.1 MenuGrammer.xml

Located inside the MainMenu.cs file we can use a simple for each loop to catch all the phrases recognized by the computer. Then we can use a switch statement to preform functionality on which phrases we want. You can see in the MenuGrammar.xml Fig. 2.4 the use of the tag header. This allows use to mark the rule and return a string. We do this simple by using the out.action"string", which returns the "string" to the program whenever the rule is recognized. We can now use this returned string in our switch statement to catch all the rules possibilities in one simple string.

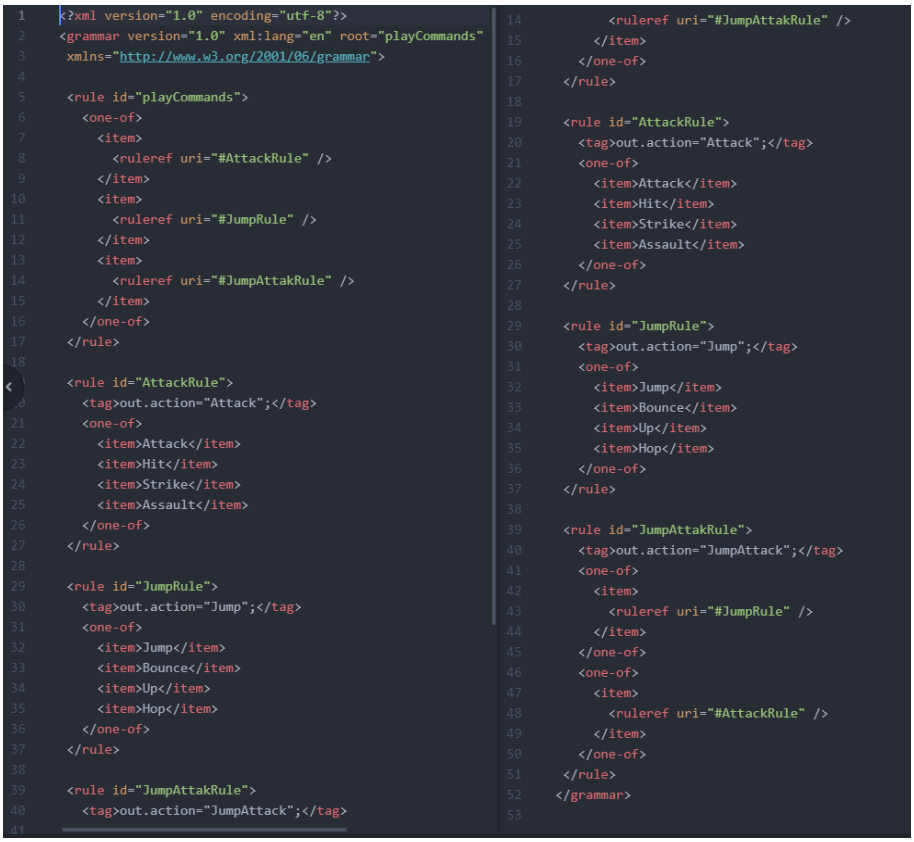
```
2 references
private void grammarRec_OnPhraseRecognized(PhraseRecognizedEventArgs args) {
    var message = new StringBuilder();
    var meanings = args.semanticMeanings;
    // Loops through phrases and applies logic based on switch
    foreach (var meaning in meanings) {
        var keyString = meaning.key.Trim();
        var valueString = meaning.values[0].Trim();
        message.Append("Key: " + keyString + ", Out Action: " + valueString + " \n");
        _outAction = valueString;
        // Checks for keywords setup in xml file out.action=""
        switch (_outAction) {
            case "Start":
                PlayGame();
                break;
            case "Options":
                Options();
                break;
            case "Quit":
                QuitGame();
                break;
            case "Back":
                Back();
                break;
            default:
                break;
        }
    }
```

Fig 2.2 MainMenu.cs

2.2 Basic Grammar

Inside Levels 1, 2, and 3 of the game the player has the ability to attack and jump using voice recognition. The player can jump using the W key and attack by using the Spacebar. This can also be achieved using phrases like 'Jump' or 'Hop' for the jump function and using phrases like 'Hit' or 'Attack' for the attack function. These can be said together to create a combination of jump and attack called under the rule JumpAttack.

- **Jump** - 'Jump', 'Bounce', 'Hop', 'Up'
- **Attack** - 'Attack', 'Hit', 'Strike', 'Assault'
- **Jump Attack** - 'Jump Attack', 'Hop Hit', 'Bounce Strike'



```xml
<?xml version="1.0" encoding="utf-8"?>
<grammar version="1.0" xml:lang="en" root="playCommands"
 xmlns="http://www.w3.org/2001/06/grammar">

 <rule id="playCommands">
   <one-of>
     <item>
       <ruleref uri="#AttackRule" />
     </item>
     <item>
       <ruleref uri="#JumpRule" />
     </item>
     <item>
       <ruleref uri="#JumpAttakRule" />
     </item>
   </one-of>
 </rule>

 <rule id="AttackRule">
   <tag>out.action="Attack";</tag>
   <one-of>
     <item>Attack</item>
     <item>Hit</item>
     <item>Strike</item>
     <item>Assault</item>
   </one-of>
 </rule>

 <rule id="JumpRule">
   <tag>out.action="Jump";</tag>
   <one-of>
     <item>Jump</item>
     <item>Bounce</item>
     <item>Up</item>
     <item>Hop</item>
   </one-of>
 </rule>

 <rule id="JumpAttakRule">
   <tag>out.action="JumpAttack";</tag>
     <ruleref uri="#JumpAttakRule" />
   </item>
   </one-of>
 </rule>

 <rule id="AttackRule">
   <tag>out.action="Attack";</tag>
   <one-of>
     <item>Attack</item>
     <item>Hit</item>
     <item>Strike</item>
     <item>Assault</item>
   </one-of>
 </rule>

 <rule id="JumpRule">
   <tag>out.action="Jump";</tag>
   <one-of>
     <item>Jump</item>
     <item>Bounce</item>
     <item>Up</item>
     <item>Hop</item>
   </one-of>
 </rule>

 <rule id="JumpAttakRule">
   <tag>out.action="JumpAttack";</tag>
   <one-of>
     <item>
       <ruleref uri="#JumpRule" />
     </item>
   </one-of>
   <one-of>
     <item>
       <ruleref uri="#AttackRule" />
     </item>
   </one-of>
 </rule>
</grammar>
```

Fig 3.1 BasicGrammer.xml

In the VoiceController.cs file we perform a similar action as mentioned above. The voice gesture recognition is used to pick out the key phrases the player uses and performs the appropriate action.

```
2 references
private void grammarRec_OnPhraseRecognized(PhraseRecognizedEventArgs args) {
    var message = new StringBuilder();
    var meanings = args.semanticMeanings;
    // Loops through phrases and applies logic based on switch
    foreach (var meaning in meanings) {
        var keyString = meaning.key.Trim();
        var valueString = meaning.values[0].Trim();
        message.Append("Key: " + keyString + ", Out Action: " + valueString + " \n");
        _outAction = valueString;
        // Checks for keywords setup in xml file out.action=""
        switch (_outAction)
        {
            case "Attack":
                Debug.Log("You said Attack!");
                playerCombat.Attack();
                break;
            case "Jump":
                Debug.Log("You said Jump!");
                playerMovement.Jump();
                break;
            case "JumpAttack":
                Debug.Log("[-----You said JumpAttack!-----]");
                playerMovement.Jump();
                playerCombat.Attack();
                break;
            default:
                break;
        }
```

Fig 3.1 VoiceController.cs

2.3 Password Grammar

Inside Levels 1, 2, and 3 of the game the player will have to interact with an exit door to move onto the next level or complete the game. It is here, that the PasswordGrammer.xml file is used. The file is simply made up of four rules. ThePasswordRule creates a chaining of items and rulerefs to create a specific phrase the player will have to used in order to open the door in-game. Once the player has collected all the letters of the hidden password they can then approach the exit door. Once there the player can say the phrase "The *Password* Is *Answer*". The password and answer can be replace with any of the following words.

- **Password** - 'Password', 'Code', 'Message', 'Secret'
- **Answer** - 'Kiwi', 'Pear', 'Date', 'Lime'

```xml
<?xml version="1.0" encoding="utf-8"?>
<grammar version="1.0" xml:lang="en" root="playCommands" tag-format="semantics/1.0"
 xmlns="http://www.w3.org/2001/06/grammar">
 <rule id="playCommands">
   <one-of>
     <item>
       <ruleref uri="#ThePasswordRule" />
     </item>
   </one-of>
 </rule>
 <rule id="ThePasswordRule">
   <tag>out.action="ThePassword";</tag>
   <item>The</item>
   <item>
       <ruleref uri="#AnswerRule" />
     </item>
   <item>Is</item>
   <item>
       <ruleref uri="#PasswordRule" />
     </item>
 </rule>
 <rule id="AnswerRule">
   <tag>out.action="Answer";</tag>
   <one-of>
     <item>Password</item>
     <item>Code</item>
     <item>Message</item>
     <item>Secret</item>
   </one-of>
 </rule>
 <rule id="PasswordRule">
   <tag>out.action="Password";</tag>
   <one-of>
     <item>Kiwi</item>
     <item>Pear</item>
     <item>Date</item>
     <item>Lime</item>
   </one-of>
 </rule>
</grammar>
```

Fig 4.1 PasswordGrammar.xml

The answer to the hidden password for the door is located within all the diamonds for that given level. Once the player has all letters the password is revealed in the top-center of their HUD. They can then approach the door and speak the command phrase filling in the Answer with the appropriate word they have now formed.

```
2 references
private void grammarRecPas_OnPhraseRecognized(PhraseRecognizedEventArgs args) {
    var message = new StringBuilder();
    var meanings = args.semanticMeanings;
    // Loops through phrases and applies logic based on switch
    foreach (var meaning in meanings) {
        var keyString = meaning.key.Trim();
        var valueString = meaning.values[0].Trim();
        message.Append("Key: " + keyString + ", Out Action: " + valueString + " \n");
        _outAction = valueString;
        // Checks for keywords setup in xml file out.action=""
        switch (_outAction)
        {
            case "Answer":
                Debug.Log("You said Answer!");
                break;
            case "Password":
                Debug.Log("You said Password!");
                break;
            case "ThePassword":
                Debug.Log("[------You have said The Password!------]");
                hasSaidPassword = true;
                break;
            default:
                break;
        }
    }
    Debug.Log(message);
```

Fig 4.2 VoiceController.cs

Once again in the VoiceController.cs file we perform actions based on the voice gesture recognition.

3 Hardware used in creating the application

The hardware used in this project was limited to Dell XPS 15 Laptop. This project focused more on using software such as Unity, Aseprite (pixel art creator), Atom (text editor). The project was designed to only use Voice Recognition as its main gesture.

4 Architecture for the Solution

The game was made in the Unity Engine and the architecture was put together through an interactive approach. This was done to help achieve GameObjects and CSharp files in the engine be reusable. This help created consist looking levels containing Prefabs that controlled the games major functionality. These were all stored in a file structure hierarchy that was reduced to eight major

files: Animation; Art_Assets; Audio; Prefabs; Scenes; Scripts; StreamingAssets; TextMeshPro;

- **Animation** : Contains all the animation work for the 2D art assets and their animator controllers.
- **Art_Assets** : Contains the Fonts, Materials, Palettes, Sprites and Tiles used to create the game.
- **Audio** : Contains all audio files for music and sound effects.
- **Prefabs** : Prefabs contain all reusable game objects such as hearts, diamonds, boxes, doors, pigs, and player. Also contains high-level functionality controllers like AudioManager, GameManager, and VoiceManager.
- **Scenes** : All 3 levels and main menu scene.
- **Scripts** : Contains all the scripting files for the game in CSharp.
- **StreamingAssets** : Contains all .xml files for the voice recognition.
- **TextMeshPro** : Contains all files need to import and use TextMeshPro Text.

5 Conclusions Recommendations

In this project I learned a lot about game design and the enormous task of creating even the smallest of games. I developed further understanding of gesture-based controls particularly in voice recognition. If I was to try again or given more time to develop I would have like to improve on the voice recognition as a combat mechanic. I had sketched through a possibility of having the pigs throw boxes at the player and words would appear over them. The player would then have to say the correct words in the order they were thrown to create a coherent sentence. I feel this would have greatly improved the combat and gameplay. I enjoyed the entire experience of creating my own idea of a game and building on previous knowledge gained from the Mobile Application Development Modules. I set out to create a full game loop which has optional and dependent voice recognition to proceed. I accomplished this and was very happy with my result.