VIRTUAL REALITY CONFLICT RESOLUTION APPLICATION

Final Year Project
B.Sc.(Hons) in Software Development

By
Aaron Hannon
Matthew Sloyan

GitHub Link:
https://github.com/MatthewSloyan/final-year-applied-project-and-minor-dissertation

May 1, 2020

Advised by Damien Costello
Department of Computer Science and Applied Physics
Galway-Mayo Institute of Technology (GMIT)

Contents

List of Figures

Chapter 1

Introduction

1.1 Application Description

This application is a virtual reality (VR) training simulation for ICSE security. The function of the application is to reduce training costs while training ticket inspectors on the Luas in Dublin. Currently they have to hire actors and close off a Luas route to train new inspectors and this project helps eliminate that. Once you launch the application on the virtual reality headset you are in a virtual train station. As soon as you hop on a train it disembarks then commencing the training session. The goal is to check everyone's ticket on board the train by conversing with all the non-player characters (NPCs) using your actual voice and they will reply to you using a text-to-speech engine. All the NPCs have different personalities so this is where the conflict resolution aspect of the project comes in. You may come across someone who may be very rude and you must coerce them into giving you their ticket or you may be fortunate to talk to someone who gives you their ticket straight away. Once all the NPCs are checked you may leave the train, check your score and end the simulation. After the purchase of a virtual reality headset there is very little cost involve and the training simulation can be replayed over and over again completely removing the need to hire actors and shutdown a Luas route for an entire day.

1.2 Client

Mark Toner who works for ICSE (International Centre for Security Excellence), contacted our supervisor to propose a project. The project entailed creating a training simulation for ticket inspectors for the Luas in Dublin. The problem with their current training system is that they have to hire out a large number of actors to act like passengers. Also, they must close off one of the Luas's routes to ensure the setting for the training session is as accurate as possible. The problem with this method was it is extremely costly to recreate this every time new people need training as they would have to hire actors again and close

of a Luas route. So, we were tasked with creating an application that was as useful in training ticket inspectors to deal with conflict resolution as it was cost effective. After meeting with Mark, we came up with some application ideas. In the end we settled on a virtual reality training simulation that allowed you to speak to bots, they would interpret what you are saying and respond. Also, these bots had to have different personalities or personas. The simulation had to have variance with each bot so to not make it too easy for the trainee. Some bots could be nice, and some could be extremely rude. After this meeting we started research and development straight away.

1.3 Objectives

Here are the main objectives that were decided after having a meeting with our client.

- To make sure that we, as developers, understood the requirements of the client to help us develop a system that the client was content with.
- To research the latest technology to aid in creating an application that would not be out-dated and that it was as efficient as possible for the time.
- To utilise our chosen research and development methodologies to full advantage.
- To implement a robust system architecture what we have learnt from our research of the latest technologies.
- To create an application that we, as developers, were happy with but more importantly, creating a product that met the requirements of our client.

1.4 Scope

This project was from the start treated as industry standard so we knew that the scope was large however, manageable and we knew that we could meet all the requirements in the time frame that was given to us. During the course of the project we had to:

- Create a virtual realistic environment for the training session to take place with life-like models and animations.
- Use speech services that would convert speech to text then text to speech in a realistic fashion.
- Create chat-bots that would recognise what you were saying with the aid of an artificial intelligence.

- Create a back-end server where the 3D engine/client could communicate with the chat-bot.
- Set-up a hosting service for the server so the 3D engine/client did not have to be on the same network to function.
- Develop a database that could securely store the training information.

Although there was a lot here to research and implement we were confident that we would get the project done in the given time frame.

1.5 Chapter Overviews

Here is a brief overview of the all the following chapters. These chapters include Methodologies, Technology Review, System Design, System Evaluation and the conclusion to the entire project.

1.5.1 Methodologies

In chapter 2 we discuss all the various methodologies we used throughout the research and development of the project. We also mention how we interacted with our client and project supervisor. There is also a big emphasis on testing, management and the tools used to aid us in development.

1.5.2 Technology Review

In chapter 3 we outline all the technologies we either used in development or that we researched and tested however did not suite our requirements. To summarise what kind of technologies we research, we looked at 3D Engines/ Gaming Engines, Speech services that would allow us to convert Speech to text and vice versa, relevant chat-bot technologies, database technologies, various servers that we could use to aid us in developing an efficient back-end to process HTTP requests and finally hosting services for our back-end server.

1.5.3 System Design

In chapter 4 we examine the entire implementation of the application in detail. We go through all the technologies that met our requirements from the Technology Review and what roll they played in creating our project. Code snippets are also used to aid our explanation of the implementation and to show the inner workings for each section in the application.

1.5.4 System Evaluation

In chapter 5 we mainly focus on the actual performance and robustness of the system which are described in the Objectives section of this chapter. We answer

questions like how well does the system perform when tested. We also look at any technical limitations that we encountered during the projects life-cycle.

1.5.5 Conclusion

In the final chapter we conclude this document as a whole. We highlight any findings from the system evaluations section. We also look at objectives and see how well we met them.

1.6 Source Code

The entirety of the this project can be located on GitHub. Upon clicking the link below you should be able to see the project's repository containing all source code and documentation related to the project. The "Chatbot" folder contains the AIML bot's source code that was used in the project and a sample bot using the Keras machine learning library. The "Research" folder contains a "Initial Research.md" file that contains all the research that was completed before starting the development stage. The "UnityEngine" folder contains the entire Unity project including all scripts and assets. Finally, there is a "README.md" file that shows all information regarding trying to run the project. The "Presentation" folder contains the presentation and video demo we created and finally the "Dissertation" folder contains all the files required to create this document.

GitHub Link: https://github.com/MatthewSloyan/final-year-applied-project-and-minor-dissertation

1.7 Deliverables

Listed below are all the components to be included in the project derived from the clients requirements:

- A Virtual reality application with a 3D train station environment and realistic non-player characters that you can converse with, developed using Unity3D that is to be deployed on the Oculus Quest.
- A live Flask server hosted on PythonAnywhere so that the Unity3D can connect and access the chat bot to generate replies and return it to the user to give the illusion of a conversation.
- A Mongo database that contains all the data of previous training sessions so that the trainees progress or score can be reviewed to see how well they deal with conflict resolution.

1.8 Workload Breakdown

In this section we will break down all the work we individually did that made it into the final build throughout the projects life cycle. Below is the list of the work done (Not necessarily in chronological order):

1.8.1 Aaron Hannon's Work

- Full AIML bot implementation, with multiple personas.
- Keras neural network implementation for chatbot. This is what we initially used, but moved to AIML.
- Flask server with AIML implementation and hosted on PythonAnywhere.
- Created Unity3D environment With non-playable characters(NPCs).
- Designed 3D environments (Trains, city and station).
- NPCs have genders and different personas.
- Created client that allowed Unity to communicate to the Flask server hosted on PythonAnywhere.
- Add animations all aspects of the training experience.
- Added character controller allowing the user to navigate through the scene for testing when not built to the Oculus.

1.8.2 Matthew Sloyan's Work

- Implemented full Azure speech services on Windows, Android and VR allowing us to convert speech to text and text to speech for NPC interactions.
- Keras neural network implementation for chatbot. This is what we initially used, but moved to AIML.
- Full Oculus support added allowing the user to navigate the scene using the Oculus Quest and interact with objects.
- Added various voices types to the NPCs.
- Improved NPC interaction system (Colour ring and satisfaction meter).
- Added sound and user options.
- Designed and implemented all menus/user interface (UI) that work in VR and Windows.
- Completed full unit testing of Flask server, for AIML and database requests (HTTP).

- Implemented scoring system.
- Added database support for user training data (MongoDB).

Chapter 2

Methodology

This chapter will look at the methodologies we used throughout the project regarding research, software development, client & supervisor meetings, testing, project management and the tools used.

2.1 Research Methodology

For this project we chose a form of a Mixed Research Methodology which involves both Qualitative and Quantitative research.

2.1.1 Qualitative

Qualitative research is focused on exploring and finding out thoughts, ideas and opinions [1]. This was suitable for our project at an early stage such as meeting the client initially. From this we gained invaluable insights into his goals, thoughts and ideas for the prototype. We also were given possible scenarios for the training application. An example was a ticket inspector (The player) would board the train and would be able to interact with any passenger to check if they had a ticket. From this scenario we conducted research into multiple areas such as chatbot technology, gamification and much more.

2.1.2 Quantitative

Quantitative research is less theoretical than qualitative research and is focused more on numerical results and statistics [2]. This research was used later in development when we had developed and deployed a test build. These builds were shown to the client and our supervisor. In return we collected the feedback, thoughts and ideas they had which was used to further make changes and achieve the initial outlined goals.

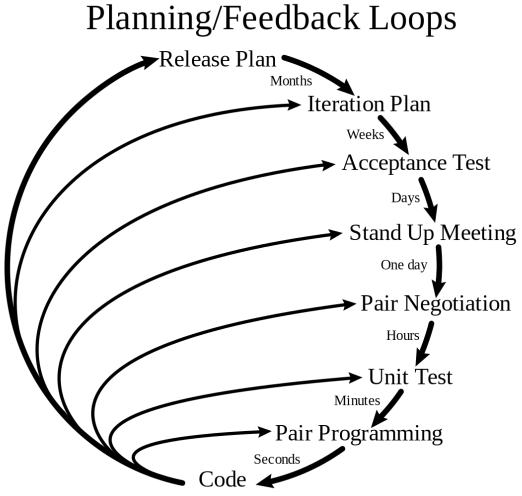
2.2 Software Development Methodology

We looked at several different approaches regarding software development methodologies, which included Waterfall and Agile methodologies. Waterfall is a sequential series steps or processes that must be completed before moving onto the following steps. These processes include (Requirements, Design, Implementation, Verification and Maintenance) [3]. However, from initial research we decided it wouldn't suit our projects as it can exclude the client from the process, causing changes to be difficult to make and delay testing until the end of the project which would decrease software quality. Another methodology we looked at was Agile. Agile promotes continuous development, deployment and testing throughout the software development life cycle (SDLC) of the project to help increase productivity, software quality and decrease development time [4]. We felt this would be most suitable to our project so we began to look at how Agile could be implemented to coincide with our initial goals.

2.2.1 Agile - Extreme Programming

The Agile methodology we have chosen for this project was Extreme Programming (XP). Extreme Programming is a form of Agile, which focuses on small development teams, quick iterative development cycles and a focus on test driven development. It is based on five values simplicity, communication, feedback, courage and respect. [5, p. 4]. This iterative development cycle is outlined in figure 2.1

Figure 2.1: Extreme Programming Life Cycle.



Advantages

- Cuts down cost and time of software projects, due to a focus on timely delivery of final products and less time on cumbersome documentation. Also, problems are solved through group discussions and errors are spotted earlier with Peer reviews. Peer reviews involve two programmers sitting side by side working on a feature. This was helpful for us working in college as we could fix bugs quickly and efficiently.
- Feedback is constant, so changes can be made quickly and efficiently.
- Software is developed faster, due to constant testing and deployment.
- Simplicity is key, so code can be developed quickly to pass tests and meet goals, and then improved later.
- Works well with small and large teams, which would be suitable for our project.

Disadvantages

- Not suitable for teams that are separated geographically.
- There can be a lack of defect documentation, so bugs can go unnoticed or not tested. However, we improved on this with GitHub issue tracking.
- The lack of full documentation can cause problems especially in larger projects or if team members leave/join midway.

As the advantages outweigh the disadvantages it was clear XP was the correct methodology to choose. Also, since it is more suited to the team size, workflow, project type and client.

2.3 Meetings

Initially we met with both our supervisor and our client. In this meeting we discussed initial goals, thoughts and ideas for the project (Qualitative Research). We also outlined the minimum requirements for the prototype which are described in the introduction chapter. Each week we would then meet our supervisor to discuss the following.

- Our progress from the last meeting.
- Feedback on our progress.
- Our next steps and planning.
- Answer any questions from our client or supervisor.

The frequency of meetings fitted well with XP perfectly as we could make any changes that arose from the meeting, then test and deploy the project again for the following week to show our supervisor.

2.4 Testing

Throughout the project we incorporated multiple testing methodologies such as Functional and Non-functional testing. More information on how these methodologies were implemented can found in the System Evaluation chapter.

2.4.1 Functional Testing

Functional testing focuses the functional aspects of a system such as unit, integration and system testing.

Unit testing

Unit testing is the first form of testing to be completed and is usually undertaken by the developer themselves. It involves testing a class level, E.g. testing functions in a class if they behave correctly and don't produce errors [6]. We implemented tests using NUnit framework to test basic functions of the Unity game. We also used pytest to run unit tests on the server. To test HTTP methods from the application to the server we used Postman to check the packets and debug errors.

Integration testing

Integration testing involves testing the system to insure it still works when adding new features, so it doesn't break other areas of the system [6]. We implemented integration by testing for all aspects of the system to see if they still worked correctly on all iterative builds. This was especially useful when adding Oculus VR support as it caused multiple errors with the Azure speech services.

System testing

System testing is a black box testing methodology completed after integration testing to ensure the system meets the requirements [7]. We implemented system testing by ensuring the application met the initial requirements outlined by the client in the Introduction Chapter.

2.4.2 Non-functional Testing

Non-functional testing focuses on the non-functional aspects of a system such as performance, usability and compatibility.

Performance testing

Compatibility testing is a measure of how the system performs under pressure, or heavy load [7]. We tested the system with a high number of requests to the server to ensure it didn't buckle under load. We also recorded the time requests

took to process which was largely dependant on internet speed. This was one of the technical limitations of the project.

Security testing

Security testing involves testing how secure the system is against unauthorized attacks [7]. We implemented security measures regarding the server, deployment and database access.

Usability testing

Usability testing involves measuring the ease of use from and end user's perspective [6]. This was vital to our game as it's a training experience, so we spent a lot of time working on improving usability of all controls, menus and game play. We conducted user tests with multiple users and our supervisor to collect data.

Compatibility testing

Compatibility testing is a measure of how well a system works in different environments (Browsers, devices etc.) [6]. This was essential as we were developing the application to work on Android, Windows, and Oculus Quest VR.

2.5 Project Management

We have used Git with GitHub to track our changes, log error and work effectively using our chosen Agile methodology. Git is an open source and free version control system which allows small or large projects to be stored and accessed efficiently [8]. Features include commits, branching, staging and workflows. GitHub allows a project to be stored remotely and for multiple people to work together seamlessly.

Using GitHub and Git allowed us to work simultaneously on the project to increase our productivity. We also used the inbuilt features on GitHub such as issue tracking. If an issue occurred or a bug was found during testing, we would make a GitHub issue, and log all our commits using the issue id. This would track our progress in fixing it and help us keep an account of the work undertaken.

2.6 Development Tools

For this project we used a multitude of tools to develop each section. The main tool that was used was the Unity Editor and Visual Studio. The Unity version used was 2019.2.6f1 which includes all the new features and provided support for the Azure Speech Services used. All the Unity script were written using Visual Studio in C#. Visual Studio was useful as it included the required packages

and IntelliSense for creating Unity scripts. The flask server was developed using Visual Studio code and was initially deployed to an AWS EC2 (Amazon Elastic Computer Cloud). Later it was deployed and modified using Heroku, and then finally PythonAnywhere.

Chapter 3

Technology Review

3.1 Overview

This chapter looks at some of the technologies used throughout the project with a focus on why we choose them along with the benefits they possess. Also, covered is some of the technology researched and reviewed which was not used but lead us to our final implementation. We initially began with a number of questions that needed answering, which helped up with our research.

- What 3D Engine to use?
- What Speech Service to use?
- What technology would we use to create a chat-bot?
- What Back-end to use?
- What Database to use?

3.2 Application

We knew we would need some form of gaming engine for the application itself. This would be the part of the project that the user would actually interact with. There are number of free gaming engines that we could have used however, we chose the Unity Engine as it is very well documented, powerful enough for our needs and provides multi-platform support [9].

3.2.1 Unity

This is the main part of the application where the user interacts with NPC's and the environment. After briefly looking at other game engines we chose Unity for the following reasons:

- Unity uses C#. C# is very versatile and supports many different programming styles [10]. It can be used to created objects, make network requests and it is also Unity's default scripting language.
- It was apparent that 3D models and animations would be key to improving the experience of the application. Unity allowed us to add 3D models with ease. Unity also has a built in animator that allows you to animate these models by key-framing them and moving them as desired. All of this was very important to create a realistic training experience for the user.
- Another aspect that was important was to be able to connect to our back-end Flask server. Unity has built-in HTTP support allowing us to send necessary JSON data from the Speech-to-Text service to the server to be processed and a response sent back. [11].
- As mentioned Speech-to-Text (STT) and Text-to-Speech (TTS) is another important feature that was needed to enhance the realism for the user. After some research we found out that a lot of the STT and TTS services support Unity and supply their own packages for the engine, giving us options for when we decided on which STT and TTS service we were going to choose.
- Unity provides full cross platform support for all the major development platforms. This was useful to useas we were developing and testing on Windows and Android with a focus on VR development.

3.3 HTTP

HTTP or HyperText Transfer Protocol, is a protocol used by the world wide web to send messages from a client to a server or vice verse. It also defines how these messages are structured to inform the server or web browser what actions to take upon receiving a request [12]. The reason we chose this approach is because for one, Unity has libraries that handle incoming and outgoing HTTP requests. Also, Flask uses HTTP making it very easy to communicate with our back-end server.

3.4 JSON

JavaScript Object Notation (JSON) is a widely used data interchange format for sending human readable messages across a network [13]. It consists of attribute value pairs and array types which can be seen below in listing 3.1. This simple example describing a person object with a number of attributes. We will use JSON to send all our web requests between our application and web server to communicate with our chat-bot and database.

Listing 3.1: JSON example.

```
{
  "firstName": "John",
  "lastName": "Smith",
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

3.5 Virtual Reality (VR)

As one of the main goals outlined in the project was to create an immersive training experience for the end user, we felt we just couldn't get that level required from a conventional screen and controls. With this in mind we researched new technologies and deciding on using Virtual Reality (VR) to achieve this. VR can be sometimes mixed up with Augmented reality (AR). AR involves displaying virtual objects in a real-world environment, usually displayed on a mobile phone display through a camera. This had been implemented in some training experiences we looked at but wouldn't have met our initial goals or scenarios provided. However, a VR environment consists of a virtual three-dimensional (3D) space that a user can interact with objects and move around. [14]. This 3D space is encapsulated in a head-mounted display (HMD) or headset that the user can wear. Inside the headset, the user can become immersed and feel they are experiencing the real or the next best thing which was exactly what we required. [14].

3.5.1 Google Cardboard

Even though VR seemed like the solution for our immersive experience we still wanted to test it to ensure it was worth the time and investment. A solution

to this was Google Cardboard (GC). GC is an affordable, simple to use cardboard frame that holds your smartphone device [15]. The user then holds the cardboard headset up to their eyes to experience VR videos and games. The cardboard frame as seen in figure 3.1 and retails for 15 EUR so it relatively cheap and it works with most phones [15]. Also, Google provide a Unity SDK for development which was useful to us when building and deploying the Android build. GMIT provided test units for us to try out with our application. From our initial tests we found the technology to be quite impressive and immersive but with a low resolution on our phones it could be improved. Another downside is that there is no controller to move the player around so a teleport movement functionality would have to be implemented which would break immersion.

Figure 3.1: Google Cardboard VR HMD.

3.5.2 Oculus Quest

The second VR device we looked at was the Oculus Quest. The Oculus Quest is an all in one VR headset that runs a modified version of the Android operating system (OS) [16]. It is built using Android 7.1 (Nougat) but can support as far back as Android 4.4 (KitKat). This device is considerably more expensive than the Google Cardboard retailing at 599 EUR for the 128GB model but serves several distinct advantages which are listed below [16].

- It has two 1440 x 1600-pixel screens with a refresh rate of 72 Hz, which provides a clearer and more realistic viewing experience.
- Runs on Android so previous builds for Google Cardboard work.
- Touch controller support as seen in figure 3.2, so the player can see their hands in the virtual environment and move the player.
- Unity support for Oculus Quest.

Figure 3.2: Oculus Quest VR HMD.

- More comfortable to wear for longer periods of time due to dedicated strap, which is also shown in figure 3.2.

3.6 Speech Services

As the application needed to include a dynamic chatbot we looked at some Text-to-Speech (TTS) and Speech-to-Text (STT) services to achieve this. Our initial thought was to include a text input system or a multiple choice dialog tree but from testing and further research we felt it would be pre-programmed and substantially less interactive for a training environment. This spurred us to look at other areas and the possibility of taking in audio from the microphone and parsing it to text using TTS. TTS involves converting human speech to a text format so it can be read by a machine in real time [17]. On the other hand, STT is the opposite in which text is converted to human like synthesized speech, which would be used to give our chatbot life, personality and evoke possible emotions [18]. Below we will look at some of the technologies we reviewed and tested in these two areas.

3.6.1 Windows

All Windows devices have STT functionality built into its Cortana virtual assistant. This allows the user to talk directly to the device and it will pick up what you said and decide from this. As described above we decided to use Unity as the main technology for our application which would be deployed to and Oculus Quest, so any service we tested must work with Unity and Android respectively. As the Windows STT services provided a Unity package, we thought it could work well, however from testing it was quite slow at depicting speech despite being accurate [19]. Also, the text predicted was lower case and contained no punctuation, or any useful characters such as question marks etc. which would

be useful for determining context in Natural Language processing (NLP). Another unfortunate downside was that it didn't work on Android when testing so we decided to look at other options, this was since its developed solely for Windows.

3.6.2 Google Cloud

The second speech service we looked at was Google Cloud Speech Synthesis. Compared to the inbuilt system provided by Windows this service works using an application programming interface (API) where audio data is sent to a remote server and a result is returned [20]. Because of this a constant internet connection is required which is another issue to look at. Using the documentation provided a solution was implemented despite it not working as desired due to the fact there is no specific Unity package available. The only way to implement it is to add the DLL files in Unity as a source which worked but unfortunately not as desired. Another area where Google Cloud was quite beneficial was the fact that allowed the developer to tailor the Speech Services to their own need. Some options included the ability to change the voice of the response from TTS or changing the pitch and tone of the voice [20]. This would be useful for depicting emotion in characters. The cost to use Google's services was also quite reasonable and provided a good allowance of free usage which would suffice for our needs but as it didn't work as expected we decided to test other services.

3.6.3 IBM Watson

Another service we researched was IBM Watson speech services, but with only a very limited amount of free characters (10,000) available for TTS and 250 minutes free with STT we found it to be a costly service to use [21]. Another downside again to this service was the fact that there was no Unity package available either, so ultimately, we decided to look into other service providers.

3.6.4 Azure

Much like Google Cloud services Azure works in much the same way and provides the same features regarding multiple voices, tone, pitch etc. which would allow a realistic experience. There is also over 140 different voices provided with 9 specific Neural voices built using machine learning that are specifically designed to provide a realistic human like response. Also, there is support for over 45 languages which could be used for future research to make the application available in multiple countries [22]. Other benefits include the fact there is a Unity package available for both TTS and STT, along with Android, IOS and Windows support. Another feature which improves on the inbuilt Windows STT is the service can depict punctuation and other useful characters such as question marks etc. which would be useful for Natural Language processing

(NLP). Below we will look a more in-dept look at the costs for each service in relation to virtual training.

Costs - Azure Text-to-Speech

There would be 250-400 replies an hour on average, with around 12500-20000 characters used per hour (average sentence around 50 characters). Based on the above the free tier would allow up to 25 hours of training for free per month [23]. Additional characters can be purchased for 14 EUR (1 million characters) which would allow for an additional 50 hours of training.

Costs - Azure Speech-to-Text

Every reply is on average 3-10 seconds. In one hour of training, it will use around 30 minutes. Based on the above the free tier would allow for up to 10 hours of training for free per month (5 hours free) [23]. A further hour can be purchased for 0.80 EUR, which equates to 2 hours of training. So, an additional 10 hours of training would cost 4.00 EUR.

From testing, research and analysis of the benefits we decided to use Microsoft Azure services for our application for the following reasons.

- Azure provided the best training cost, which would be useful for our client.
- Azure provided the most useful custom options in regard to multiple voices, language, pitch tone etc. which would required to scale the project in the future.
- Azure STT works extremely well at depicting human speech and includes punctuation. It is also very efficient and accurate.
- A Unity package is provided which would allow it to be easily deployed to a Unity environment and run efficiently.
- Android and IOS are supported which would allow the service to work on the multiple VR devices reviewed such as the Oculus Quest and Google VR.

3.7 Chatbot

It was a given that a we needed to create a chat-bot so this was one of the first areas we started to research. We looked at all the chat-bots created in the past and they all followed a trend. This trend was that they either used machine learning or AIML also known as Artificial Intelligence Markup Language. Before deciding on one from the beginning we tested both technologies to see which was suited best to our requirements. All we needed the bot to do at the start was take in an input and return an accurate and relevant response. Before implementing

Speech-to-Text and Text-to-Speech we worked with the most simple form of conveying dialog which was strings of text. We got this working with both the AIML and machine learning approach. Below are the two technologies in more detail and why when chose AIML for our final implementation.

3.7.1 Keras

Keras is a deep learning library that allows you to create neural networks quickly with very little implementation[24]. It also has a dedicated library for python which would run seamlessly on our flask server meaning we could make requests to the neural network using HTTP. To train the model, we processed a JSON intents file that included objects that looked like this:[25]

Listing 3.2: Keras training data example.

```
{
  "intents": [
    {
      "tag": "greeting",
      "patterns": ["Hi.", "Is anyone there?", "Hello.", "Good day"
          , "Whats up?"],
      "responses": ["Hello!", "Good to see you again!", "Hi there,
           how can I help?"],
      "context_set": ""
    },
    {
      "tag": "greetingResponse",
      "patterns": ["I am great thanks!", "Great, thank you!", "Not
           too bad!"],
      "responses": ["That's good to hear.", "I'm glad."],
      "context_set": ""
     }
   ]
}
```

The pre-processing mapped certain words to the tag. For example in listing 3.2 you can see the tag "greeting", and one of the patterns is "Is anyone there?". What the pre-processing does is it creates a bag for words related to the tag removing any unnecessary words like "Is" in this case. Then it trains the model using this tag and that bag of words. It also has a list of responses that is added to the output of the neural network. After training the model Keras loaded it and it was ready for use. Pre-processing would have to be done for every input too. This was done using the the method of creating a bag for words. Then that bag of words was used as an input to predict an output/response. The neural network was very accurate however it had some issues:

- The model would have to be rebuilt every time more responses where appended to the intents file making it tedious and bulky to maintain.

- There was no simple way for creating independent sessions for each bot. This was a feature that we really needed as we knew the application would have many different instances of the bot and that progress/state needed to be saved and there was no simple way of doing it.

Initially the we thought the safest approach in creating a chat-bot would be to use a neural network however, after doing some testing and implementing a working build using keras we found out that it was not the right route to take. Also due to the nature of neural networks it can be quite difficult to unit test, as the same result might not be returned each time. This would increase testing difficulty and decrease overall robustness of the project.

3.7.2 AIML

Artificial Intelligence Markup Language (AIML) is markup language that allows you to create a set of conversation rules[26]. These rules contain various tags and symbols that allow you to generate outputs or responses based on the input. AIML's tags are similar to another markup language called Extensible Markup Language(XML). The reason these markup languages are so similar is due to the fact that AIML is derived from XML. The main reason we chose AIML over the Keras approach was for the simple reason that we needed to store session data. This would have been very difficult to do using keras and would have forced us to do a lot of string manipulation to try and extract the relevant data from the input. AIML had built in tags and methods that allowed us to save various bits of data that we could store on a session by session basis. This was essential to the project and that is why we fundamentally chose AIML over Keras.

Below is a table of the tags that are most used in AIML and a description of what they do:[27][28]

AIML TAGS	
Tag	Description
< aiml >	Defines the beginning and end of a AIML document.
< category >	Defines the unit of knowledge in a bot's knowledge base.
< pattern >	Defines the pattern to match what a user may input to a bot.
< template >	Defines the response of a bot to user's input.
< star >	Used to match wild card * character(s) in the <pattern> Tag.
< srai >	Multipurpose tag, used to call/match the other categories.
< random >	Used <random>to get random responses.
< li >	Used to represent multiple responses.
< set >	Used to set value in an AIML variable.
< get >	Used to get value stored in an AIML variable.
< that >	Used in AIML to respond based on the context.
< topic >	Used in AIML to store a context so that later conversation can be done based on that context.
< think >	Used in AIML to store a variable without notifying the user.
< condition >	Similar to switch statements in programming language. It helps the bot to respond to matching input.

In listing 3.3 is a snippet of the simplest input and output that can be made in AIML. When the user says "HELLO BOT", AIML looks for patterns that contain this string. If the pattern is found, the bot replies with the contents of the template tags which is "Hello User!". However, if the input is not found in the patterns, you can have a category that acts as a catch all using the "*" symbol which will become the default response if no other input was found which in this case would be "Sorry, I do not understand...".

Listing 3.3: AIML example.

```
<aiml version="1.0.1" encoding="UTF-8"?>
    <category>
    <pattern> HELLO BOT </pattern>
        <template>
            Hello User!
        </template>
    </category>

    <category>
    <pattern> * </pattern>
        <template>
            Sorry, I do not understand...
        </template>
    </category>
</aiml>
```

With the AIML file set up, all we had to do was access it through our flask server. We were fortunate that python has an AIML library that handles reading the file and saving the state for the chat-bots. Saving the states of every bot was essential for this project and that is why we picked AIML. Also, using AIML would allow us to unit test the chatbot more efficiently, as the returned response can be predicted.

3.8 Back-end

To create a fully fledged application we needed a suitable back-end technology to handle the requests from the Unity application over HTTP.

3.8.1 Node

Node.js is an open-source server-side platform that runs and executes using JavaScript. It allows the developer to easily build a fast and scalable network server to run a website, make HTTP requests, etc. [29]. From initial tests we had it running a simple Python file as our chatbot had to be built in Python due to its nature, but that was problematic for Node.js so we investigated other options.

3.8.2 Flask

Flask is a web framework that allows you to build web applications with ease. Flask is actually classed as micro-framework meaning it has very little dependencies to external libraries [30]. This means Flask is extremely light-weight therefore, very efficient. Because we were working with STT and TTS efficiency was key. We needed to be able to generate responses as quickly as possible to give the illusion that you were having an actual in person conversation with the bot. Flask had this capability along with it being simple to set up so that is why we used it to host the back-end of this application. Listing 3.4 describes a snippet of python code to set up a simple working server:

Listing 3.4: Flask example.

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello World!"

if __name__ == "__main__":
    app.run()
```

3.9 Back-end - Deployment

After developing and deciding on using Flask as our back-end, we needed to host the server and deploy it in a production build so it could be accessible to devices outside of the local network. A debug server for testing in Flask can only take one request at a time so it's not scalable or secure, so a production build is required. Below is a list of the different options for deploying a Flask server.

3.9.1 Self-hosted Options

Self-hosted solution can be deployed using Web Server Gateway Interface (WSGI) containers. This specification is used to describe how a web service communicates with web applications. A few examples of WSGI containers include Gunicorn, uWSGI, Gevent and Twisted Web [31]. We attempted to implement a WSGI on a Windows Amazon Web Service (AWS) virtual machine (VM) but from research these types of servers are best suited to Linux machines, so we decided to look into other options.

3.9.2 Hosted Options - Cloud Services

There is various platform as a service (PaaS) solutions available to deploy a production server to, which we will look at below. PaaS is a model provided by third party vendors, where the vendor hosts the hardware and software on their own virtual machines and provides access at an affordable cost, which is usually much cheaper than setting up your own machines [32]. Below we will review two options that we assessed, tested and used for development.

Heroku

Heroku is a PasS which allows developers to build, run and deploy applications easily and efficiently to the cloud [32]. Git version control software is used to update, modify and deploy quickly. Heroku also provides a free tier services for student so it is a cost-effective approach to allowing applications to be easily accessible around the world, which is what we required to access our server from an Oculus Quest VR device. There was a few downsides to Heroku though. The first one being an issue with AIML. AIML allows you to store session data/variables which would enable you to save names and other data relevant to each NPC. Heroku had a problem storing this session data making it very difficult to save the states of all the different NPCs/bots in the application so we had to find a different deployment service. Another issue we found was that Heroku would timeout quickly if there was no incoming requests. This made the initial request slow to respond because you would have to wait for Heroku to launch the server.

PythonAnywhere

PythonAnywhere is what we decided to use after testing Heroku. It is very similar to Heroku as it allow developers to build, run and deploy their applications on the cloud. However, it differs in a few ways. The free tier of PythonAnywhere allowed the server to stay live for 3 months at a time making initial requests to the flask server much quicker [33]. You could also opt for a paid tier which keeps it live based on a certain amount of usage but the free tier was perfect for testing. Another way that it differs from Heroku is that there is no Git version control instead using an online IDE on their site which allows you to edit all your files from anywhere on any device [33]. Finally the main reason we choose PythonAnywhere, it did not have the issue with saved AIML sessions that Heroku did. This was a deal breaker for us because we needed these sessions for the bots and without them it would have been very difficult to save the states for the bots.

3.10 Databases

The results of the training simulation needed to be stored somewhere so they could be studied by the testing centre at a later date, so we began to look into possible options to achieve this. For this project we have chosen MongoDB as our database storage technology.

3.10.1 MongoDB

MongoDB is a document oriented distributed database, that allows users to store data quickly, efficiently and securely [34]. The data stored in binary format called BSON, but it is JSON in its human readable form as seen below in listing 3.5. Includes is an "_id" attribute which allows documents to be found efficiently.

Listing 3.5: MongoDB document example.

```
{
  "_id": "5cf0029caff5056591b0ce7d",
  "firstname": "Jane",
  "lastname": "Wu",
  "address": {
    "street": "1 Circle Rd",
    "city": "Los Angeles",
    "state": "CA",
    "zip": "90404"
  },
  "hobbies": ["surfing", "coding"]
}
```

Advantages

- High speed - MongoDB is a document-orientated database. It is easy to access documents by indexing. Hence, it provides a fast query response. It is estimated to be 100 times faster than a relational database.
- Sharding - Allows storage of large pieces of data by distributing to several servers, this also improves read times.
- Flexible - MongoDB is a schema-less database. That means we can have any type of data in a separate document. This allows us to store data of different types.
- Easy environment setup - Easier to setup than a relational database. Also provides Flask plugin called PyMongo which allows easy connection to the database.

3.10.2 mLab

mLab is a Database-as-a-Service (DAAS) for MongoDB, it is a free to use provider that allows for cloud hosting for your MongoDB database [35]. We have decided to use this technology as we have experience using it and wanted to learn more. There are also useful Flask plugins provided which allow a connection easily to mLab.

3.11 Other areas of research

As outlined in our introduction the focus of this application was to provide a realistic and gamified training experience that would be useful to the user specifically in the area of conflict resolution. To succeed with this we researched areas such as Digital Twins, Sara de Freitas, Michael J Sutton, Serious Fun, Gamification and Digital Training/VR Training. This research can be found in our "Research" section on GitHub repository.

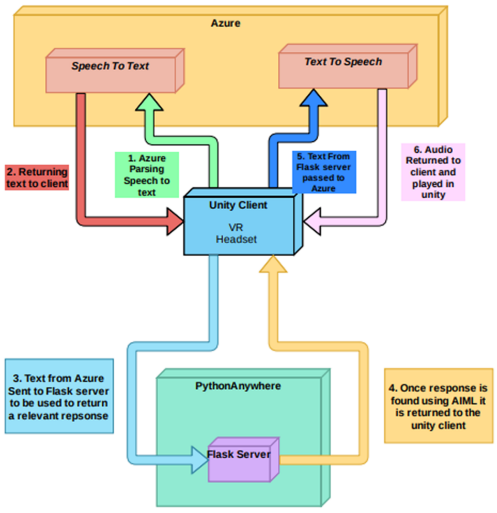
Chapter 4

System Design

4.1 Overview

In this section, we will illustrate the entire system in detail. We will go through all the various technologies that were finalized from the technical review, where they are located in the project structure and what role they have in creating a fluid and realistic experience for the user. In Figure 4.1 we can see the high-level design of the project and how all the technologies we have developed connect. Each component has a major role to play in creating this fluid experience and we will discuss how they were implemented in detail.

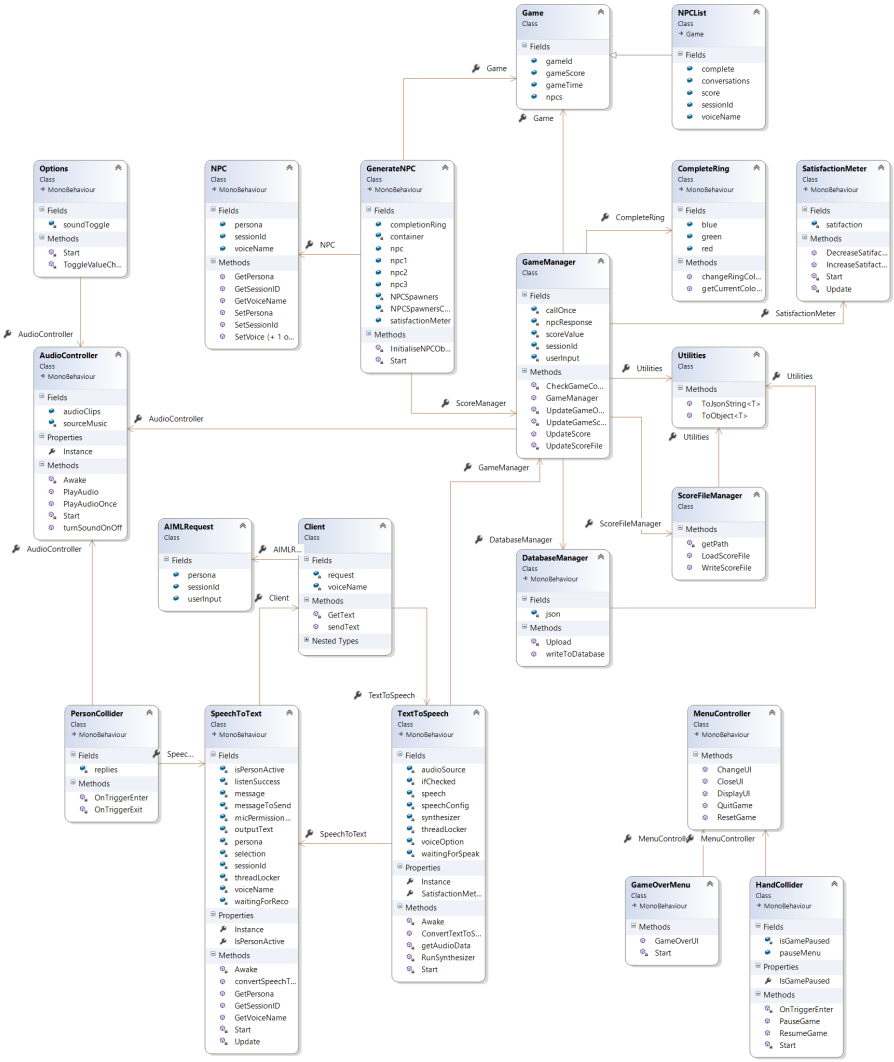
Figure 4.1: System Architecture.



4.2 Unity

As seen in figure 4.1 Unity is the central component to this application. It is the part that the user interacts with and the part where all the data flows in and out of. Figure 4.2 describes the whole Unity system at a class level. In this section and whole chapter we will look at this class diagram in more detail and describe how some of the features work. We will also explore the graphical side of the implementation and how we achieved that.

Figure 4.2: Unity Project Class Diagram.



4.2.1 3D Models

Generating realistic bots with realistic models was a problem that was easily solved with the aid of a website called Mixamo[36]. This website has an plethora of high quality 3d character models along with animations. Once the model was imported into Unity all that was left to do was give these bots a session ID,whether they had a ticket or not, a random voice, a gender based on that voice and a random persona E.g. Polite, neutral and rude. These personas would aid us later in deciding which AIML file to use server-side. As seen in the below code snippet this is how we generate all these traits. The session ID is a random number between 1 and 10000. The persona is an integer between zero and three, each number meaning a different persona:

NPC Personas	
Number	Persona
0	Rude NPC
1	Neutral NPC
2	Polite NPC

Table 4.1: NPC Personas

Listing 4.1: Random NPC persona generation.

```
public void SetSessionId()
{
   sessionId = UnityEngine.Random.Range(1, 10000);
}
public void SetPersona()
{
   persona = UnityEngine.Random.Range(0, 3);
}
public void SetVoice()
{
   //0 = male 1= female
   int gender = UnityEngine.Random.Range(0, 2);

   string[] voicesMale = {"en-US-GuyNeural", "en-IE-Sean"};
   string[] voicesFemale = { "en-US-JessaNeural", "de-DE-KatjaNeural" };
   if (gender == 0)
   {
      int rand = UnityEngine.Random.Range(0, voicesMale.Length);
      voiceName = voicesMale[rand];
   }
   else if(gender == 1)
   {
      int rand = UnityEngine.Random.Range(0, voicesFemale.Length);
      voiceName = voicesFemale[rand];
   }
}
```

Once these traits have been generated all that is left to be done is to instantiate them in the virtual 3D environment. The solution to this can be seen in the below snippet. A random NPC model is chosen and oriented correctly in the environment. This snippet is also wrapped in a for loop to generate multiple random NPCs. There is an array of manually place GameObjects in the virtual world. These GameObjects act as spawning points for a single NPC. Based on the NPCs voice a Male or Female model is loaded then it is spawned in the position and rotation of one of the spawn points according to the index of the loop. After this process is complete, we see NPCs standing in all the spawn points. This is how all the NPCs/Bots are spawned in the application.

Listing 4.2: NPC generation.

```
if (copy.GetComponent<NPC>().GetVoiceName() == "en-US-JessaNeural" || copy.GetComponent<
    NPC>().GetVoiceName() == "de-DE-KatjaNeural")
{
    int rand = UnityEngine.Random.Range(0, 2);

    // Instantiate random woman model
    if(rand == 0)
    {
        copy = Instantiate(npc2, new Vector3(NPCSpawners[i].position.x, NPCSpawners[i].position.y
    , NPCSpawners[i].position.z), Quaternion.Euler(0, NPCSpawners[i].rotation.eulerAngles.y, 0));
        copy.GetComponent<NPC>().SetVoice(npcVoice);
        copy.transform.parent = container.transform;
    }
    else if(rand == 1)
    {
        copy = Instantiate(npc3, new Vector3(NPCSpawners[i].position.x, NPCSpawners[i].position.y
    , NPCSpawners[i].position.z), Quaternion.Euler(0, NPCSpawners[i].rotation.eulerAngles.y, 0));
        copy.GetComponent<NPC>().SetVoice(npcVoice);
        copy.transform.parent = container.transform;
    }
}
else
{
    // Instantiate male model
    copy = Instantiate(npc1, new Vector3(NPCSpawners[i].position.x, NPCSpawners[i].position.y,
    NPCSpawners[i].position.z), Quaternion.Euler(0, NPCSpawners[i].rotation.eulerAngles.y, 0));

    // Set the voice and add to container object.
    copy.GetComponent<NPC>().SetVoice(npcVoice);
    copy.transform.parent = container.transform;
}
```

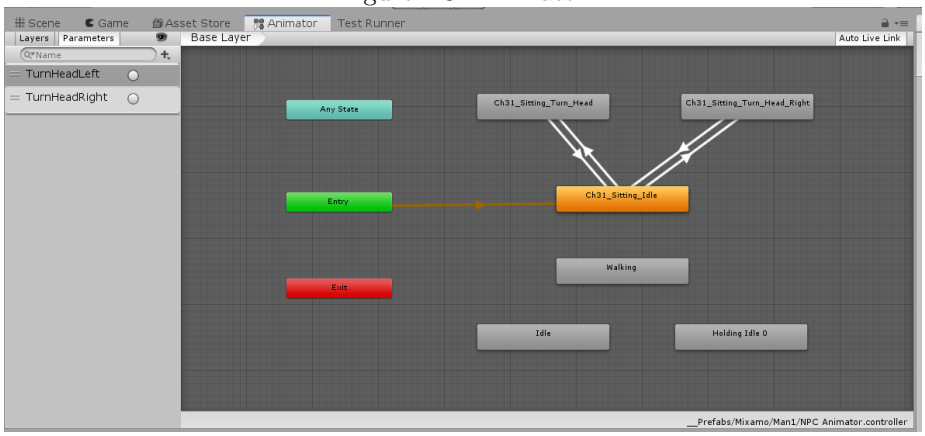
Due to us not being trained in 3D modeling and animation here are some links to the models we used and altered for this project:

- Train Station Model: https://assetstore.unity.com/packages/3d/environments/industrial/train-stations-bundle-161989
- City Chunk Model: https://assetstore.unity.com/packages/3d/environments/urban/simple-city-pack-plain-100348

4.2.2 Animation

Animations in our opinion were important in this application. We needed the user to feel as if they were speaking to a person. If the NPCs where static it would have ruined the realism, completely disconnecting the user from this virtual world. As mentioned earlier, we were fortunate to find a website called Mixamo[36]. As well as realistic models it also had animations to go alongside them. The process of applying the animation to the model is simple but tedious. Firstly, you must add an Animator component to the model/NPC this allows you to manage your animation clips. Once this is done, animations can be added to the animator using a graph-like interface as seen in figure 4.3. Each animation is considered as a state. If certain parameters are met you can change the state, therefore changing the animation.

Figure 4.3: Animator



4.2.3 Usability/User Experience

Throughout the development of this project we focused a lot of effort on ensuring the training application was intuitive to the user. We looked a number of areas in regards to user experience and usability which we will look at below.

Passenger (NPC) Interaction

From our initial meetings with our supervisor and the initial scope we laid out we felt getting the interaction to feel as real and immersive as possible was important. In real life, a ticket inspector can walk up to passenger and start a conversation simply by saying "Hello". We wanted our training experience to reflect this simple nature. There were a number of interaction in our development process such as initially having the player press a button to start speaking with an NPC, but we felt it was quite cumbersome and took away from the experience especially in VR. Through user testing we developed a simple way of interacting with an NPC. It involved adding a collider to each NPC which would act as a barrier. Once this barrier was passed, E.g the player moves close to the NPC it would trigger a conversation and the Azure Speech to Text service would start listening for input. Another aspect we looked at was keeping the flow of conversation intuitive. Similar to the initial interaction we felt that pressing a button wasn't suitable, so again we developed an automatic system. Once the result is returned from the Text to Speech and spoken by the passenger, the STT service will automatically start listening again for the player to reply to the passenger. For example if the player asked to see the passengers ticket, and the NPC replied with "I don't have a ticket", then the service will start listing for possible reply like "Why don't you have a ticket?". This simple flow of interaction and conversations improved the user experience drastically. More information on STT and TTS technologies can be found below in their

respective sections.

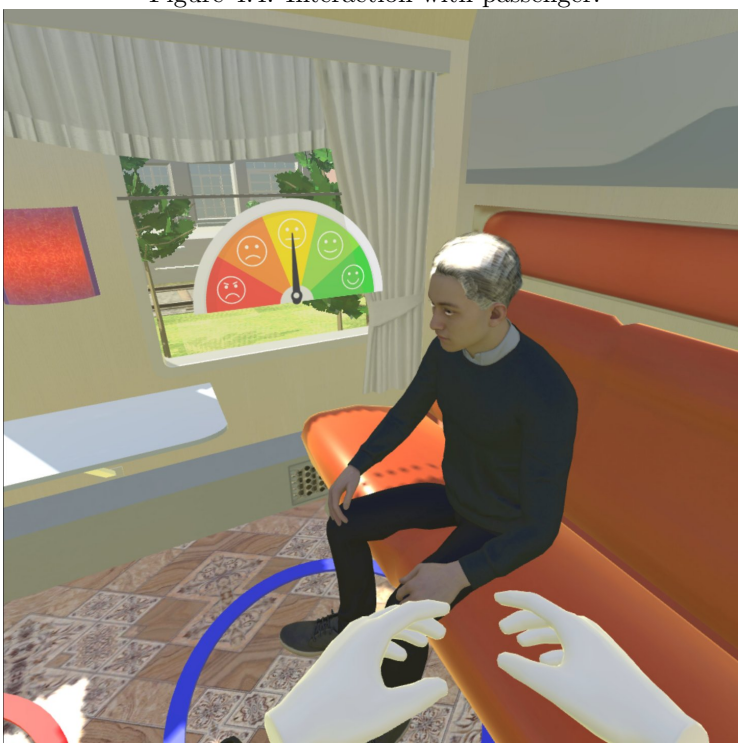
Player Input

Again from our meetings another area to improve immersion was to allow the player to psychically speak to a passenger. Our initial tests involved a keyboard input however it was very slow and cumbersome to type out each reply. We also looked at a multiple choice decisions where the player could reply with three options, however from initial user tests it was compared to a role playing decision based game rather than a immersive training experience. Instead we developed a Speech service that would listen for your voice, parse it to text to send to our chatbot for a reply. Of course this was more of a challenge as the player could say anything and in any way, however it is something that was very successful in our final build and really improved the overall experience.

Passenger Information and Scoring

There were a number of things that we wanted to display to the player about each passenger they interact with. Firstly, whether the passengers ticket had been checked on not. We decided this should be simple and researched multiple methods such as displaying a green tick beside their head and outputting a completion sound to the user amongst others. We decided that a ring around the passengers feet that would initially be red and then turn to green would work best. The ring turns to green when the player has checked the passengers ticket or dealt with situation where they don't have a ticket. It also turns blue when the player is within the range to start a conversation. This can be seen in figure 4.4. This red, blue, green loop is very simple to grasp and worked well in the various user tests. Also it worked well for the VR experience as it is very subtle but effective in displaying the required information.

Figure 4.4: Interaction with passenger.



VR Support

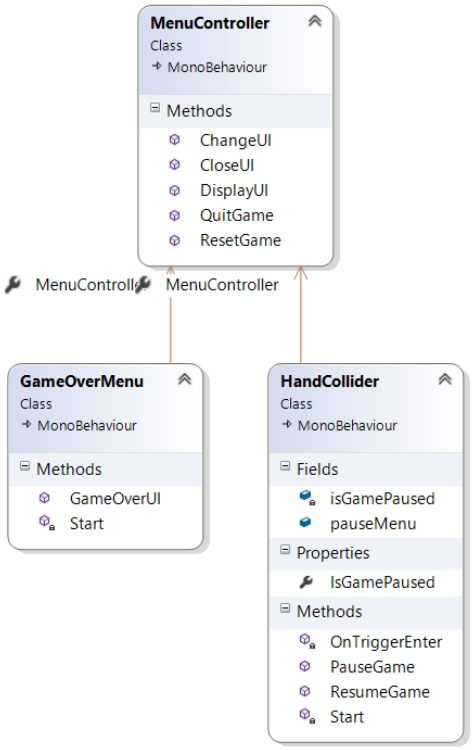
As our main platform focus was VR support on the Oculus Quest we wanted to ensure it was the most immersive experience possible. This was taken into account when designing the various environments (The train, the station, city etc), the in game sound, the player interactions, and controls. The controls were designed to require the least requirement of buttons possible, so the user didn't feel like they were holding controllers. All menu interactions are done by looking around in the virtual space and pulling the triggers, which is a simple fluid movement. One area we felt really improves the experience was the inclusion of the in-game watch. It was a very simple thing but it really was success in our user tests. In the virtual space we wanted to feel like the user is really in the train station to the best of our ability, and if they look down at anytime they can view their hands in real time and move their fingers. On the left hand there is a simple dialog watch that displays the real world time. If the user touches this watch with their right hand the game is paused and a virtual menu is displayed. Unfortunately, this was only a feature that could work on the VR build, but we implemented the pause menu support by displaying the menu on front of the player when the pause button is pressed.

User Interface (UI) Design

For this project we used a simple design scheme for all menus and UI. It involved a simple grey background, strong contrasting text and blue buttons. This made it easy to read and view all elements throughout the application. All menus were also designed with ergonomics in mind, so that it would be easy for the user to move between menus, close the menus or select options. For example on the pause menu implemented in VR the QUIT and RESET button were placed at the top as from testing they could be accidentally pressed when touching the watch. This can be seen in figure 4.5 The game time is also placed accordingly along with the option to turn he sound on and off. We wanted to ensure everything required by the player is all in one place and easily accessible. This idea was reflected in the start menu and game over menu too. Figure 4.6 shows the layout of classes developed to handle all UI.

Figure 4.5: UI displayed on watch.

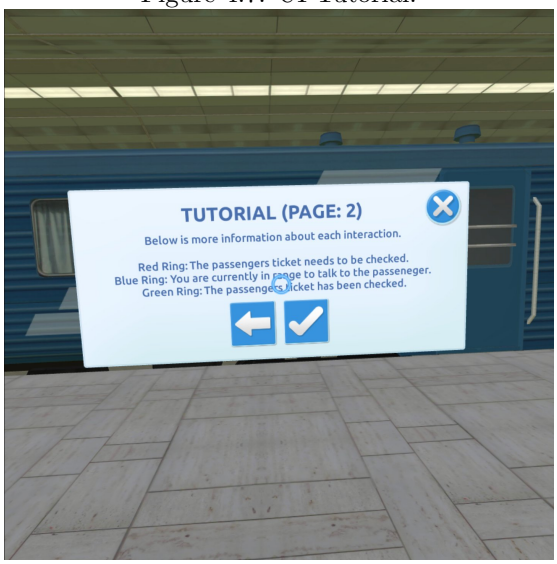
Figure 4.6: Class Diagram - Menus



Tutorial

For this application we have also designed a tutorial scene. This scene is simply used for the trainee to learn more about the application and how it works before they get into the training environment. We used this scene to display our presentation in the project demo video required. We have also developed simple to use tutorial UI, which allows the user to scroll between pages and close when ready. An example of this can be seen in figure 4.7.

Figure 4.7: UI Tutorial.



4.2.4 Scoring System

As one of the main goals of this project was to create a useful training experience to train ticket inspectors in relation to conflict resolution, we decided it would need a scoring system to help both the trainee and the training centre. A simple system was devised for each interaction and was scored depending on user input and the way it's said. Table 4.2 shows example of how the result is scored, based on the user input. If the user is deemed aggressive then they will get a -1 score, if they are deemed neutral their score won't change and if they are deemed positive they will gain one point to their total score. This simple system proved beneficial in early tests, so we implemented same idea with the personalities of each NPC.

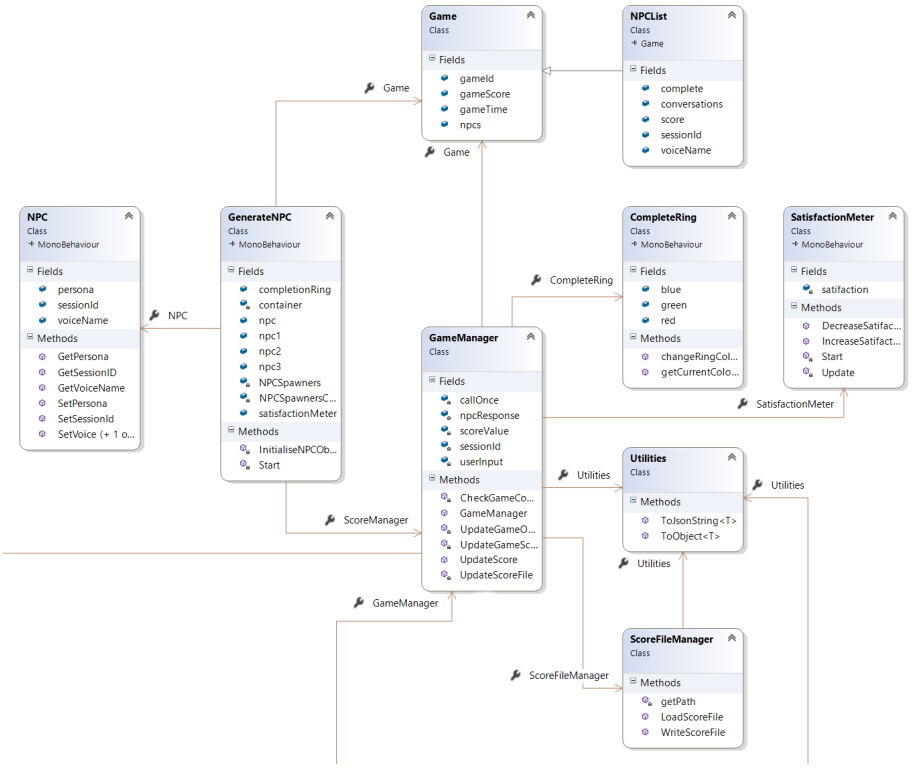
Interaction Scoring	
User Input	Score
Show me your ticket!	-1
Can I see your ticket?	0
Can I see your ticket, please?	+1

Table 4.2: Scores

The scoring systems works as follows as seen in Figure 4.8. Once the game is started and each NPC is being generated and template "Game" object is setup, this object contains a nested list "NPCList" which contains information about each NPC such as the conversations, current score, and sessionID. Once created it is written to a local file on the device, the decision to use file storage rather than have the object in memory was chosen for several reasons. Firstly, it allowed the file to be accessed at any time from any class easily, quickly and efficiently. It also would be stored if the game crashed, or if the user quit however results retrieval weren't implemented in the final build, it would be something to look at for future development.

The next steps occurs after each successful interaction with a NPC. The GameManager is called from the Client to update the game and score. The UpdateScore() method updates the score by reading in the current file, finding the NPC and updating their current score respectively. This also handles the check if the NPC is complete E.g. if their ticket has been checked. If the NPC is on the last state a boolean "complete" will be set to true which is used to check if the training experience is complete. This is written back out to a local file as a JSON string using the Utilities class methods. The Utilities methods were designed using generics so they can handle any type for usability. The end game condition is also checked here to see if the game is complete, if so, the end game menu is displayed with the players results.

Figure 4.8: Scoring System Class Diagram.



4.3 Azure Speech Services

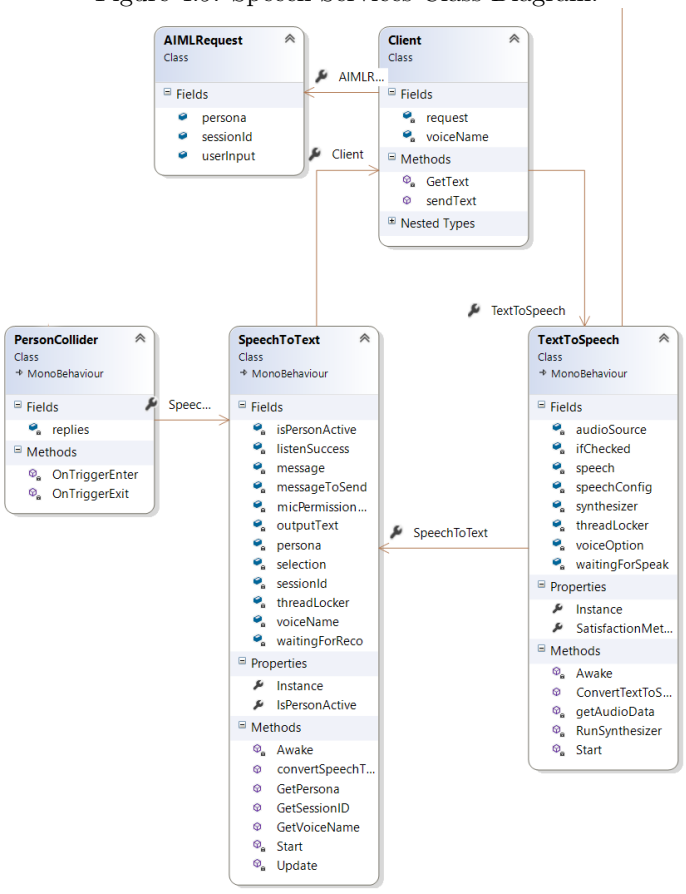
From our research and testing efforts as laid out in the technology review Azure was chosen as the provider for the speech services due to its many benefits. This provided the voice of our human like NPC's (Non-player character) and converted the players speech to machine readable diction which is sent to the Flask server for processing. Azure provides a Unity plugin to allow access to their services, with this implemented we could access the classes required.

The Speech to Text and Text to Speech functionality were the first sections to get working as it was initially decided that we would use voice commands and human speech to improve the user experience and usability rather than having the user type in their response. Using the sample Unity project provided by Microsoft as a guide we tested the implementation to learn how it works. Using this an initial prototype, the player could walk up to a cylinder which would turn blue on interaction. Once the collision occurs the speech engine begins listening and the result is returned from an Azure server as text output to the screen. We also developed a similar test build for Text to Speech where a user could input text into a text box which would play back their input as if it was

spoken by a human voice. This was one of the bare essential goals of our project.

Figure 4.9 shows the class diagram of the whole speech services in the project. The process starts in the PersonCollider class, once the player interacts with the NPC OnTriggeredEntered is fired, which sends a request to the SpeechToText class. This begins listening and displays this to the user. Once speech is recognised it is sent to Azure for processing and a result is returned as text. This is then sent to the Client which sends a HTTP request to the server as an AIMLRequest object. Once the result is processed the text is returned with a score for the chatbot to speak and the game will be updated. The TextToSpeech class handles this asynchronously and the process restarts again once the chatbot is finished talking (TTS). We will now go into more detail of how this three elements work.

Figure 4.9: Speech Services Class Diagram.



4.3.1 Speech-to-Text

Once this class is called a connection to the Azure services is made, then with a using statement to cut down memory usage the configuration information is passed into the imported classes (SpeechRecognizer). This begins listening for a total of 15 seconds in which it will timeout and display an error to the user. If the speech recognition is successful, the message is saved and will be sent to the client on the next update. This can be seen in Listing 4.3.

Throughout the duration of the project the SpeechToText code was modified heavily to improve it in many ways, which is described below.

- We had to implement a web client so it could contact the server via HTTP. More information can be found in the HTTP section in regard to the web client. We ran into problems trying to call the web client on result returned (Line 24 in Listing 4.3) so instead a check was added to the Update method which checks if there is a message to be sent on each frame. If so then the returned result is sent to the server for processing.
- We also worked on making this Speech To Text class threaded so it would let the application continue to play when waiting for a result, as seen on line 10-13 in Listing 4.3.
- Security was implemented by protecting the key from external access, so it is local to the device.
- Singleton design pattern was used as there should only be one instance of SpeechToText and it allows easy access from other classes.

Listing 4.3: Speech To Text - Setup and request

```
// Creates an instance of a speech config with specified subscription key and service region.
// API_Key for security this is read in from a text file and is not included on Github.
var config = SpeechConfig.FromSubscription(API_Key, "westeurope");

using (var recognizer = new SpeechRecognizer(config))
{
    listenSuccess = false;
    messageToSend = "";

    lock (threadLocker)
    {
        waitingForReco = true;
    }

    // Starts speech recognition and returns after a single utterance is recognized. The end of a
    // single utterance is determined by listening for silence at the end or until a maximum of 15
    // seconds of audio is processed.  The task returns the recognition text as result.
    var result = await recognizer.RecognizeOnceAsync().ConfigureAwait(false);

    // Checks result.
    string newMessage = string.Empty;
    if (result.Reason == ResultReason.RecognizedSpeech)
    {
        newMessage = result.Text;

        // Set independant variables for sending the message to the server.
        listenSuccess = true;
        messageToSend = newMessage;
    }
    else if (result.Reason == ResultReason.NoMatch)
    {
        newMessage = "NOMATCH: Speech could not be recognized.";
    }
    else if (result.Reason == ResultReason.Canceled)
    {
        var cancellation = CancellationDetails.FromResult(result);
        newMessage = $"CANCELED: Reason={cancellation.Reason} ErrorDetails={cancellation.
        ErrorDetails}";
    }

    lock (threadLocker)
    {
        message = newMessage;
        waitingForReco = false;

        // Dispose of recognizer correctly.
        recognizer.Dispose();
    }
}
```

4.3.2 Text-to-Speech

Once this class is called a connection to the Azure services is made using the configuration information. This includes data such as the voice name which is passed from the Client. There are several voices available both male and female for different countries in the world to give a more realistic and unique experience. We choose two male and female voices from the list of Neural voices which are created using machine learning, from testing they provided the most realistic sounding voice. This can be seen in Listing 4.4. Once setup the text is passed into the synthesizer.SpeakTextAsync() function. This is implemented

using a task (Task¡SpeechSynthesisResult¿) so it doesn't block and wait for a response.

Listing 4.4: Text to Speech - Setup and request

```
1  public void ConvertTextToSpeech(string inputText, string voiceName, bool ifChecked)
2  {
3      this.ifChecked = ifChecked;
4
5      // For security this is read in from a text file and is not included on Github.
6      string API_Key = System.IO.File.ReadAllText("../../API_Key.txt");
7
8      speechConfig = SpeechConfig.FromSubscription(API_Key, "westeurope");
9
10     // The default format is Riff16Khz16BitMonoPcm.
11     // We are playing the audio in memory as audio clip, which does not require riff header.
12     // So we need to set the format to Raw16Khz16BitMonoPcm.
13     speechConfig.SetSpeechSynthesisOutputFormat(SpeechSynthesisOutputFormat.
           Raw16Khz16BitMonoPcm);
14
15     // Change voice depending on option selected for multiple characters of different genders and
           ethnicities.
16     speechConfig.SpeechSynthesisVoiceName = voiceName;
17
18     lock (threadLocker)
19     {
20         waitingForSpeak = true;
21     }
22
23     // Creates a speech synthesizer.
24     synthesizer = new SpeechSynthesizer(speechConfig, null);
25
26     // Starts speech synthesis and returns after a single utterance is synthesized.
27     // There was an issue where the game would freeze at this point "synthesizer.SpeakTextAsync(
           inputText)" blocks until the task is complete.
28     // To solve this, I have implemented a task which runs using a Coroutine that waits until the
           result from Azure is returned.
29     // Code is adapted from: https://stackoverflow.com/questions/57897464/unity-freezes-for-2-
           seconds-while-microsoft-azure-text-to-speech-processes-input
30     Task<SpeechSynthesisResult> task = synthesizer.SpeakTextAsync(inputText);
31
32     StartCoroutine(RunSynthesizer(task, speechConfig, synthesizer));
33
34     lock (threadLocker)
35     {
36         waitingForSpeak = false;
37     }
38 }
```

The audio response received from Azure is 16-bit mono format. The 16 bit is the bit depth, or resolution of audio. A common compact disc (CD) uses 16 bits per sample. As it's returned as bits it must be converted first to a float array to be output to the device speaker. A loop was implemented to shift the bits left on each iteration using the formula supplied in the documentation. This can be seen on line 7 in Listing 4.5. Once completed the audio is ready for output.

Listing 4.5: Text to Speech - Converting audio from bits to array

```
private Task<float[]> getAudioData(SpeechSynthesisResult result)
   {
      var sampleCount = result.AudioData.Length / 2;
      var audioData = new float[sampleCount];

      for (var i = 0; i < sampleCount; ++i) {
         audioData[i] = (short)(result.AudioData[i * 2 + 1] << 8 | result.AudioData[i * 2]) /
      32768.0F;
      }

      return Task.FromResult(audioData);
   }
}
```

The last step of the process is to output the processed audio directly to the device speaker. A clip is first created using the data and applied to the AudioSource of the NPC and then this is played as seen in Listing 4.6. Lastly, the SpeechToText class is called again to start the NPC listening for a new user input. However, if the NPC's ticket has already been checked this step is skipped and a predefined output is spoken, E.g "Sorry, but you have already checked my ticket.".

Listing 4.6: Text to Speech - Asynchronous Audio Processing

```
private IEnumerator RunSynthesizer(Task<SpeechSynthesisResult> task, SpeechConfig config,
     SpeechSynthesizer synthesizer)
{
   // Wait until the task is complete E.g Azure Text to Speech returns a result.
   yield return new WaitUntil(() => task.IsCompleted);

   var result = task.Result;

   // Check result.
   if (result.Reason == ResultReason.SynthesizingAudioCompleted)
   {
      // Create a task to create audio data and wait till its completed.
      Task<float[]> audioTask = getAudioData(result);
      yield return new WaitUntil(() => audioTask.IsCompleted);

      // Synthesize the Audio and play to the speaker.
      // The output audio format is 16K 16bit mono
      var sampleCount = result.AudioData.Length / 2;
      var audioClip = AudioClip.Create("SynthesizedAudio", sampleCount, 1, 16000, false);
      audioClip.SetData(audioTask.Result, 0);
      audioSource.clip = audioClip;
      audioSource.Play();

      // true if ticket has already been checked, otherwise start listening again until ticket
      has been checked.
      if (!ifChecked)
      {
         // Wait until the audio is completed and start listening again.
         // Code adapted from: https://answers.unity.com/questions/1111236/wait-for-audio-to-
      finish-and-then-load-scene.html
         yield return new WaitWhile(() => audioSource.isPlaying);
         speech.convertSpeechToText(speech.GetSessionID(), speech.GetPersona(), speech.
      GetVoiceName());
      }
   }

   else if (result.Reason == ResultReason.Canceled)
   {
      var cancellation = SpeechSynthesisCancellationDetails.FromResult(result);
```

```
36          Debug.Log($"CANCELED:\nReason=[{cancellation.Reason}]\nErrorDetails=[{cancellation.
            ErrorDetails}]");
37      }
38
39      // Dispose of synthesizer correctly.
40      synthesizer.Dispose();
41  }
```

Throughout the project the Text To Speech code was modified and improved. Some of these improvements are outlined below.

- We encountered a problem where if the the internet connection was slow the application would hang and freeze until the result was returned. From research of the documentation it was found that Microsoft developed their sample code with this problem not fixed. This problem was outlined here [37] which led it to our attention in the documentation. If "synthesizer.SpeakTextAsync()" is called it blocks and waits for a result, which was disastrous for our project as it could hang for 2/3 seconds. To fix this a asynchronous task was implemented which would allow the application to run smoothly until the audio result was played to the user. This Task¡¿ can be seen in Listing 4.5.
- Singleton design pattern was used as there should only be one instance of TextToSpeech and it allows easy access from other classes.
- Security was implemented by protecting the API key from external access, so it is local to the device.

4.3.3 HTTP Web Client

In order to communicate with our Flask server, we used HTTP. We did so by using the UnityEngine.Networking library. Before sending the data, we created an object to store the session ID, persona and the user input from the STT service. Once this object was created it was converted to a JSON string. Then this string of json was sent, as seen in the code snippet below, to the Flask server hosted on PythonAnywhere to be processed and return a predicted response from the bot. That response is then sent to the TTS service.

```
        string json = new Utilities().ToJsonString(request);
        UnityWebRequest www = UnityWebRequest.Put("http://aaronchannon1.pythonanywhere.com/
        request", json);
        www.SetRequestHeader("Content-Type", "application/json");
        yield return www.SendWebRequest();

        if (www.isNetworkError || www.isHttpError)
        {
            Debug.Log(www.error);
        }
        else
        {
            string reponse = www.downloadHandler.text;

            try
            {
                string[] reponses = reponse.Split('=');

                TextToSpeech.Instance.ConvertTextToSpeech(reponses[0], voiceName, false);

                new ScoreManager(request.sessionId, Int32.Parse(reponses[1]), request.userInput,
        reponses[0]).UpdateScore();
            }
            catch
            {
                TextToSpeech.Instance.ConvertTextToSpeech(reponse, voiceName, false);
            }
        }
```

4.4 Chatbot - AIML

As mentioned in the technical review, we decided to use AIML instead of a keras neural network. In listing 4.7 we can see the Flask GET request that also contains the AIML kernel. The kernel object controls the entire AIML library. Once the request data has been parsed, the kernel decides what aiml file to load based on their persona and if the have a ticket or not.

Listing 4.7: Generate AIML response

```
@app.route('/request', methods=['PUT'])
def predictResponse():
    # Get json from request.
    sessionId = request.get_json()['sessionId']
    persona = request.get_json()['persona']
    userInput = request.get_json()['userInput']
    hasTicket = request.get_json()['hasTicket']

    # Load specific aiml file depending on persona and if they have a ticket.
    if hasTicket == True:
        kernel.respond("load aiml " + str(persona))
    else:
        kernel.respond("load aiml " + str(persona)+ " NO TICKET")

    print("DATA: ", kernel.getPredicate("usersName", sessionId))

    result = re.sub(r'([^\s\w]|_)+', '', userInput)

    # Predict reponse for specific session using user input.
    response = kernel.respond(result, sessionId)

    return response
```

Once loaded, any illegal characters are removed from the users input, making it easier to pass it into the AIML kernel. That edited input is then passed into the .respond method to get a response. From listing 4.8 we can see a stripped down version of the conversation you can have with a RUDE bot with NO ticket. As mentioned, this aiml file is loaded from the request data. The user can initiate the conversation by saying "Ticket Please". The bot then looks for a category that contains the phrase. In the case that the phrase does not exist in the AIML we have added a catch all that can be seen at the bottom of the file to generate a response if the bot does not understand what you are trying to say. In the case of saying "Ticket please", once found, the bot has a few random options it can response with. These options can be seen under the random tag. The bot clearly doe not want to talk to you right now. You can continue by saying " I'm sorry you need a ticket". Even though this phrase is not directly in the list of categories however, a form of it is: "<pattern>* YOU NEED A TICKET< /pattern>". The star symbol(*) allows you to say any number of words before "YOU NEED A TICKET" making it AIML great in catching most scenarios. The bot says that it does not have a ticket. The next part is up to the user. Do they let them away with not having a ticket or tell them that they have to pay a fine? If the user tells them to "BRING ONE NEXT TIME" the rude bot will reply with "Yeah I will. Can you leave me alone now?=3". The "=3" at the end of the reply indicates to us on the Unity client side that the conversation with that bot is now complete and deactivates them. In the case that you ask them to pay the fine, they respond rudely and you must tell then to calm down. After this the bot can decide whether they'd rather pay the fine or make sure to bring one next time.

Listing 4.8: Generate AIML response

```
<category>
  <pattern>TICKET PLEASE</pattern>
  <template>
     <random>
        <li> I'm busy.</li>
        <li> Can't you see I'm busy</li>
        <li> I don't have time for this!</li>
     </random>
  </template>
</category>

<category>
  <pattern>* YOU NEED A TICKET</pattern>
  <template>
    <random>
        <li> Well I don't have one</li>
        <li> Go away, I don't have one</li>
        <li> Stop bothering me I don't have one!</li>
     </random>
```

```
    </template>
  </category>

  <category>
    <pattern>BRING ONE NEXT TIME</pattern>
    <template>
         Yeah I will. Can you leave me alone now?=3
    </template>
  </category>

  <category>
    <pattern>* PAY THE FINE</pattern>
    <template>
         Piss off! I'm not paying the fine!
    </template>
  </category>

  <category>
    <pattern>* CALM DOWN</pattern>
    <template>
      <random>
          <li> Okay, I'll make sure to bring one next time=3</li
              >
          <li> I'd rather pay the fine=3</li>
       </random>
    </template>
  </category>

  <!-- CATCH ALL -->
 <category>
  <pattern>*</pattern>
      <template>
       <random>
          <li>Can you repeat that?=1</li>
          <li>What?=1</li>
          <li>I don't understand=1</li>
          <li>Pardon?=1</li>
          <li>Excuse me?=1</li>
       </random>
    </template>
  </category>
```

The listing 4.8 is a simplified version of a rude bot with no ticket. An AIML file has been constructed for each persona (polite, neutral and rude). Also, another three AIML files were created for those personas that do not have a ticket. Totaling to six AIML files which all can be found on the Github repository.

4.5 Back-end - Flask

After some consideration as described in the Technical review, we decided a Flask server would be best to handle the back-end. Firstly we set up a simple Flask server that could handle a simple GET request and return some text which can be seen in Listing 4.9. It was important to create this simple GET request for testing purposes. We could just test that we could make a HTTP request and if it returned a response, we knew the server was working correctly.

Listing 4.9: Basic Flask GET request

```
1  from flask import Flask, jsonify, request, json
2  from flask_pymongo import PyMongo
3  import aiml
4  import os, requests, time
5  from xml.etree import ElementTree
6
7  @app.route('/')
8  def index():
9     return "<h1>Welcome!!</h1>"
10
11 if __name__ == "__main__":
12    app.run(debug=True)
```

Once the simple Flask server was setup it was time to implement the AIML request. This request would contain data like the session ID, NPC person and the user input from the STT service. Because of this it had to be a PUT method as seen in Listing 4.10. We can also see in lines 4-6, the JSON data from the request being converted back into their respectable types. As mentioned in Section 4.4 here is where all the AIML code is located too. Once AIML has generated an output from the input this output is sent back to the Unity client to be converted back into speech using the Azure TTS service.

Listing 4.10: Flask PUT request to generate AIML response.

```
1  @app.route('/request', methods=['PUT'])
2  def predictResponse():
3      # Get json from request.
4     sessionId = request.get_json()['sessionId']
5     persona = request.get_json()['persona']
6     userInput = request.get_json()['userInput']
7     hasTicket = request.get_json()['hasTicket']
8
9     # Load specific aiml file depending on persona and if they have a ticket.
10    if hasTicket == True:
11       kernel.respond("load aiml " + str(persona))
12    else:
13       kernel.respond("load aiml " + str(persona)+ " NO TICKET")
14
15    print("DATA: ", kernel.getPredicate("usersName", sessionId))
16
17    result = re.sub(r'([^\s\w]|_)+', '', userInput)
18
19    # Predict reponse for specific session using user input.
20    response = kernel.respond(result, sessionId)
21
22    return response
```

The final part of the Flask server is another PUT request that, once the training session is finished, pushes all the session's data to a Mongo database. This can be seen in Listing 4.11. Simpler to the AIML put request, all that JSON data is converted back into types. This data includes the session/game's ID, the score of the user, the time it took and all the NPCs. Once this data has being inserted into the Mongo database a response is returned stating that the insertion was successful.

Listing 4.11: Flask PUT request to Push Data to MongoDB

```
@app.route('/api/results', methods=['PUT'])
def uploadResult():
    # Get json from request.
    gameId = request.get_json()['gameId']
    gameScore = request.get_json()['gameScore']
    gameTime = request.get_json()['gameTime']
    npcs = request.get_json()['npcs']

    # Get collection from database.
    result = mongo.db.results

    # Write json object to MongoDB database.
    result.insert({
        'gameId': gameId,
        'gameScore': gameScore,
        'gameTime': gameTime,
        'npcs': npcs
    })

    return jsonify(data="Result sucessfully uploaded.")
```

4.6 Hosting - PythonAnywhere

After creating an account, the steps to hosting a flask server on PythonAnywhere pretty tedious. Especially if you did not create the Flask server from scratch on the site.

- Firstly, you create a web app. This will generate all the necessary files you need to start.
- Then we added all our server files to the web app fold that was generated.
- Unfortunately, we had not developed the flask server on PythonAnywhere from the beginning so we had to install all the extra libraries at once before we could even get a simple get request working. To install the relevant libraries, we had to open a bash console and pip3.7 install –user "LIB-NAME" for every import we had.
- Then you must add every import to the WSGI config file.

After all this everything worked flawlessly, and we were able to get predictions from the Flask server.

4.7 Database - MongoDB

We have used a MongoDB database to store the final game information securely and efficiently. More information on how this is implemented can be found in the "Backend - Flask" section. We choose mLab to host this database. mLab is a fully fledged cloud provider that allows users to easily access and host their database without the need for a virtual machine. Once an account was created and the hosting service was setup using a Amazon Web Services virtual machine, the flask server could easily be connected to using credentials. These credentials are stored in an environments file and read in. This environments file is not published to our GitHub repository for security reasons.
There were a number of issues getting MongoDB with mLab to work with our hosting service PythonAnywhere but with the correct imports and a number of tests we got it work correctly.

4.7.1 Database Schema

The database schema was designed to only include the information that would be useful to the client for training purposes. Below is a descriptions of each of the fields and why they were included. The design can be seen in listing 4.12.

- gameId - unique ID for each training session. This would be used to pull information about the session when required.
- gameScore - An overall game score to rank sessions.
- gameTime - The time taken by the trainee. If a trainee was rushing or took too long it could give a negative outlook.
- npcs - A list of all passenger interactions in one training session. Each NPC has an array index which contains information such as their session id, if there ticket was checked, their score, voice name and finally a list of the whole conversation thread with said passenger. The voice name could be used to see how a trainee would react to different genders or mixed races. The list of conversations is very useful so that the trainer can see the way the trainee deals with situation based on their reply. These fields can be seen in listing 4.12.

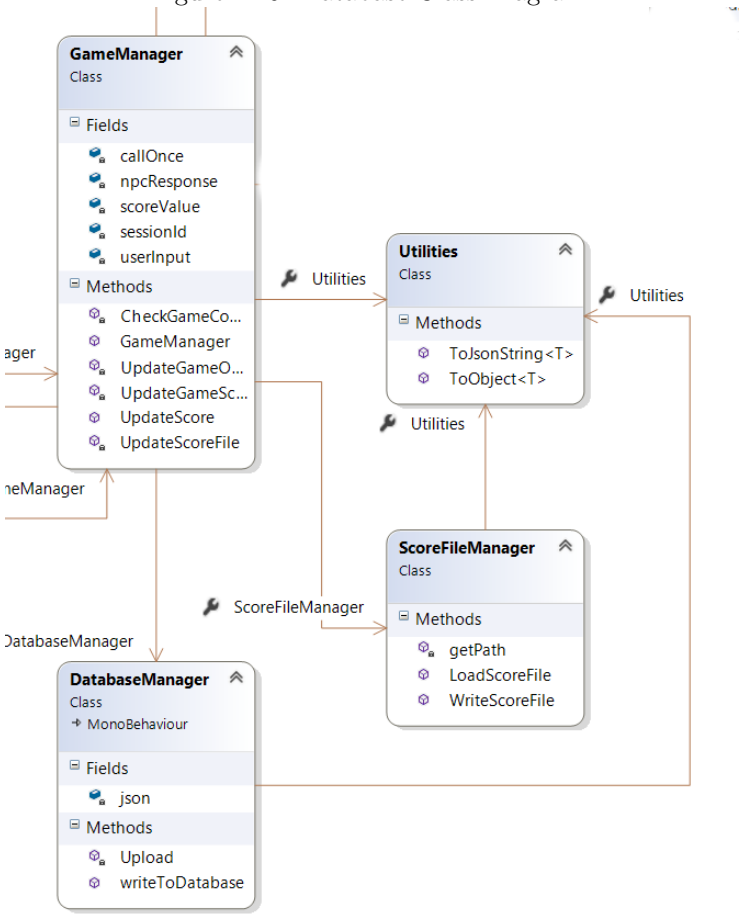
Listing 4.12: MongoDB database schema

```
    "gameId": 200,
    "gameScore": 1,
    "gameTime": "Time:\n0 Mins 20 Sec",
    "npcs": [
       {
          "sessionId": 6591,
          "complete": True,
          "score": 1,
          "voiceName": "de-DE-KatjaNeural",
          "conversations": [
             "Do you have a ticket?",
             "Here you go!"
          ]
       }
    ]
```

4.7.2 Unity Connection

How this works on the Unity side can be seen in Figure 4.10. It works very similar to how the client connects to the server for AIML requests and uses the same methods. A connection is made using a HTTP request that passes the data defined in listing 4.12 to flask server. This data is then stored using our MongoDB database hosted through mLab.

Figure 4.10: Database Class Diagram.



4.8 Code Design

With our experience from a number of modules throughout our Software Development course, we have implemented our best knowledge in the following areas to provide a well written and efficient piece of software.

4.8.1 Design Patterns and Principles

A design pattern is blueprint used to solve common problems in software development. They are used to improve code readability and efficiency.

Singleton

The Singleton pattern has been implemented in a number of cases that require one instance of it. For example this pattern can be seen in the AudioController script and in listing 4.13. This would ensure there was only one instance of the class created as there should only be one audio system. It also allowed it to work easily across multiple scenes such as the tutorial and start menu.
A Singleton is created by creating a private constructor so that no other classes can create an instance of that class. The second step is to create a static instance of the class which acts as the constructor. If the instance is null then create and instance once and store it, so further calls access this cached instance. This can be seen again in listing 4.13

Listing 4.13: Singleton - implemented in the AudioController script.

```
public static AudioController Instance { get; private set; }

private void Awake()
{
   if (Instance == null)
   {
      Instance = this;
   }
}
```

Single Responsibility Principle (SRP)

SRP ensures that every class has one single purpose. E.g so there isn't classes that are handling multiple tasks. At all levels throughout the design SRP is upheld as every class has specific purpose that delegates to other classes as required.

Open Close Principle (OCP)

OCP allows classes to be open for extension but closed for modification. So multiple clients, databases, speech engines etc could be added without breaking the overall implementation.

Dependency Inversion Principle

No higher-level modules or scripts depends on a low-level module.

Law of Demeter

No single function knows the whole navigation structure of the system. All common functionality is subjugated into multiple methods and reused where possible.

4.8.2 Object Pooling

To save on system memory in regarding the 3D objects in the Unity engine we decided to use object pooling. What this entails is loading one model into memory and it is just duplicated preventing us from having to instantiate a new model every time. An example of this in the project is the City chunk model that spawns when you are on the train. One chunk is instantiate and every other chunk is a copy of that first one. Once the copy is out of view it is then deleted, therefore saving more system memory. Another example of this can be seen in start screen with the NPC models walking in the station. There is only three models and the rest of them are copies of the initial models. This was extreme important from a system design point of view. The reason being, we were developing this application for the Oculus Quest. The Quest is currently running an old enough version of Android, Android 7.1.1 to be exact. Because the Quest is running on an mobile operating system with mobile level specifications we needed to make sure it ran smoothly and Object pooling helped us achieve this.

4.8.3 Modularity

Even though at a glance the project seems pretty glued down, meaning it is a training simulation for ticket inspectors and that it. However, the system is designed in such a way that certain components can be swapped and changed to give the user a different training experience. For example, to change the dialog of the NPCs all you have to do is simply change their AIML scripts. Now you have bots that are not just passengers on a train. To further the modularity, the entire scene can be changed in Unity with different models to give a different feel. Those are the only two things you would have to change in the entire project to create a training simulation for any other line of work.

4.9 Platforms

As we were developing for Windows, Android and VR with a focus on VR we had to ensure everything we developed would work on all platforms. This was a big challenge but we developed in such a way that all features would port across easily. However some features did have to be designed slightly different such as the watch menu implementation described above which could only be possible in VR. However, we have successfully developed a system that works correctly on all platforms.

Chapter 5

System Evaluation

5.1 Overview

In this chapter we will look at actual performance and robustness of the system. We will also review how it compares to our initial goals and scope outlined in the Introduction chapter. We will also look at any technical limitations that we encountered during the project's life-cycle.

5.2 Testing

Throughout the project we incorporated multiple testing methodologies such as Functional and Non-functional testing. We will now look at the testing efforts and the results of said tests. More information and descriptions of what these methodologies involve can been found in the Methodology chapter.

5.2.1 Functional Testing

Functional testing focuses on the functional aspects of a system such as unit, integration and system testing.

Unit Testing

Unit testing was one of the first forms of testing we completed and was undertaken at a class level. It ensured the code, methods and classes written worked correctly individually. This was completed for both Unity and our Flask server.

Server Unit Testing To test the flask server which handles the AIML chatbot requests and sending data to the MongoDB server we used pytest. pytest is automatically imported in Python 3.6, and to run tests you simply put "test_" before the method name and run the test file using "tests.py" in the command line. This allowed us to rapidly tests our code. To implement the testing some

template test methods were setup which could be used multiple times by different tests. One was setup for the AIML requests and a second for the MongDB database uploads. Listing 5.1 outlines the test method for AIML requests. This method takes in two parameters "test_data" and "bot_response". test_data is the request to be sent to the server shown in listing 5.2. bot_response is the response the AIML bot should return depending on the request sent. The template method shown in listing 5.1, uses these parameters, makes a call to the server and gets the response. The HTTP response code and AIML response is then checked using an assertion. If both assertions are true then the test passes. Response code 200 is a successful HTTP request. This testing allowed us to create a whole suite of tests for each type of persona and allowed us to test multiple responses at once. This improved our testing drastically as we didn't need to deploy the full application if we made changes to the server or AIML files.

Listing 5.1: pytest template method for testing AIML chatbot.

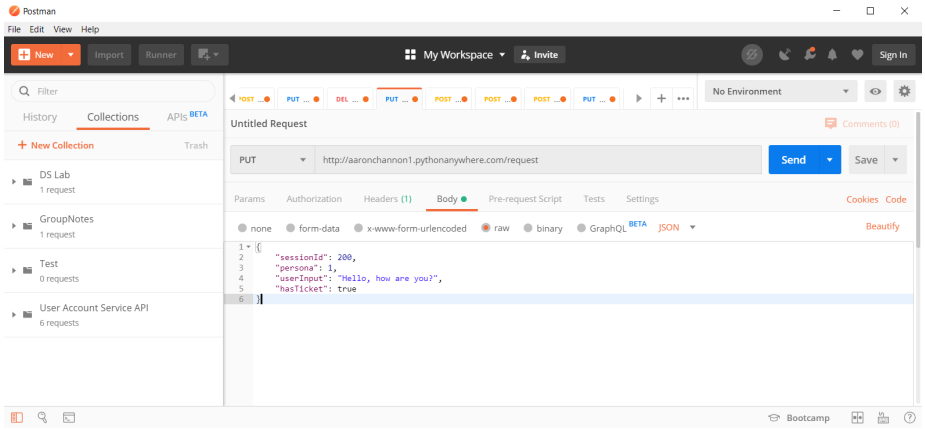
```
1
2  def test_predictResponse(test_data, bot_response):
3      response = app.test_client().post(
4          '/request',
5          data=json.dumps(test_data),
6          content_type='application/json',
7      )
8
9      data = response.get_data()
10
11     assert response.status_code == 200
12     assert data == bot_response
```

Listing 5.2: pytest AIML runner methods which call template methods.

```
1
2  # AIML test runners.
3  def test_aiml_polite():
4      test_data = {'sessionId': 1234, 'persona': 2, 'userInput': 'HELLO'}
5      tests.test_predictResponse(test_data, "Hello Pleased to meet you!")
6
7  def test_aiml_neutral():
8      test_data = {'sessionId': 1234, 'persona': 1, 'userInput': 'DO YOU HAVE A TICKET'}
9      tests.test_predictResponse(test_data, "Here it is=1")
```

Postman was also used to to check the packets and debug errors. Postman is a piece of software focused on developing API's rapidly, and allows the user to send API requests to a server and check the response. We used this to unit test the server specifically without integrating it with the whole system. This contains the server URL hosted on PythonAnywhere with the method parameter "/request" defined in the flask server implementation. Attached with the request is form data which includes the session id, persona, speech input and whether the passenger has a ticket. An example of this can be seen in figure 5.1.

Figure 5.1: Postman request example.



Integration Testing

Integration testing involves testing the system to insure it still works when adding new features, so it doesn't break other areas of the system. We implemented integration by testing for all aspects of the system to see if they still worked correctly on all iterative builds. As we have a lot of technologies working together seamlessly, integration tests were essential. After each new feature was added we would also run our unit tests again which are defined in listing 5.1 and 5.2. If they didn't pass then something was affecting the rest of the system. Due to the nature of our project user tests were also completed as integration tests, as bugs could develop from new features or when a new technology is added. One example of such an error was when adding Oculus Quest support and integrating it with the whole system. The Azure Speech Services would stop working once the application was closed and opened again. This was a major problem as the application could only be used once. From research of possible solutions and checking the error logs output by the Oculus Quest using LogCat, we came to a solution which we had thought would fix it. The Quest was throwing an error in regard to a .dll file it couldn't find. A number of people had a similar issue on a forum post, and the suggestion was to change the Unity version. To test this we built a new basic version of the project in Unity 2018, however the error still persisted when everything was added in. Going through each addition it was found that the error was caused by an Avatar game object that was added to the VR camera. This Avatar game object allows the user to see their own personal avatar in game connected to an Oculus account. It is useful for multiplayer games but not required for our application, but it is a standard addition to the supplied VR camera that Oculus provides so we would have never found this error without going through all possible options and integrating the Oculus support with the speech services.

System Testing

System testing is a black box testing methodology completed after integration testing to ensure the system meets the requirements. We implemented system testing by ensuring the application met the initial requirements outlined by the client in the Introduction Chapter. After each user, unit and integration test we would compare to our initial goals and see if we were meeting them. If it was a success we would move onto the next feature, but if not we would continue to develop the feature until it was meeting the required standard.

5.2.2 Non-functional Testing

Non-functional testing focuses the non-functional aspects of a system such as performance, usability and compatibility.

Network/System Performance Testing

Frame rate and response time where important for this project. We understood that the Oculus Quest would not perform as well as a gaming desktop but VR was a component that we could not remove from the project. To test the average frame rate we ran the project on both the Oculus and on a Gaming desktop. The average results can be seen in Figure 5.3. The Oculus ran at around 55fps on average and the desktop ran at 110fps on average. We found that this was an appropriate amount of frames per second to ensure the user has a smooth experience. Regarding the response time, this can be seen in Figure 5.2. To obtain these results we interacted with an NPC once to see how fast it took them to respond. Both of us did this to get accurate results as our network speed are drastically different. Even though it took roughly 1.5 seconds to respond on a really bad internet connection we found that it did not take too much from the users experience.

Figure 5.2: Response Time Graph.

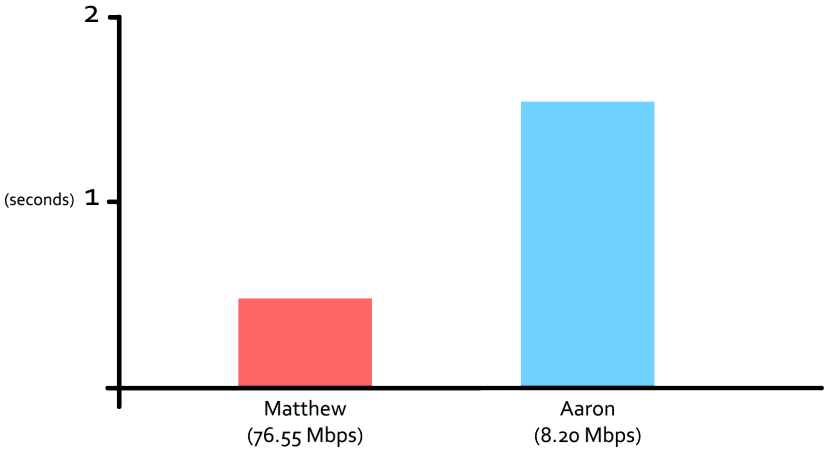
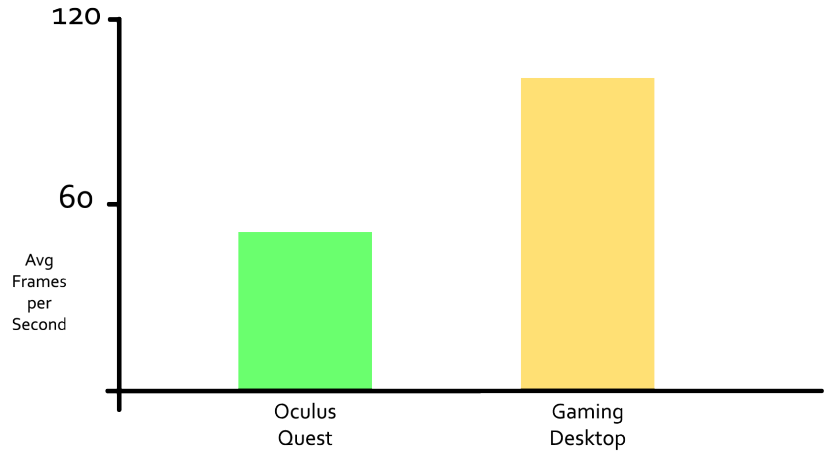


Figure 5.3: Frame Rate Graph.



Security Testing

Security testing involves testing how secure the system is against unauthorized attacks. We implemented security measures regarding the server, deployment and database access. For example on the server, the database credentials were kept securely in an environments file and read in. This environment file is not published to our GitHub repository so there would be no unauthorized attempts to access the data saved. The same attitude was taken with the server and deployment for the AIML requests as the methods are purely GET/PUT requests which just return information and don't modify data.

Usability Testing

Usability testing involves measuring the ease of use from and end user's perspective. This was vital to our game as it's a training experience, so we spent a lot of time working on improving usability of all controls, menus and game play. Throughout the development life cycle we tested the application with a number of friends and family along with our supervisor. From these tests we would collect their thoughts, ideas and improvements which helped us shape our final build. This testing was very important to us and really helped with the user experience design, the easy flow of conversation and simple interaction design between the user and the passengers. Examples of this include using speech to talk to the passengers, using a simple red, blue and green structured ring to display information about the passenger and displaying menu information on the players watch in virtual reality.

Compatibility Testing

Compatibility testing is a measure of how well a system works in different environments (Browsers, devices etc.). This was essential as we were developing the application to work on Android, Windows, and Oculus Quest VR. We initially developed for Windows for testing purposes but quickly started to develop for Android and the Oculus Quest as they both use the same Android build. Based on this it was essential that what we developed would be compatible with all platforms.

5.3 Technological Limitations

5.3.1 Text To Speech Voices

In an ideal world we would have real human voices replying to user but obviously this is impossible therefore being one of our technical limitations. However we found that Azure's AI voices were good enough to use when converting the AIML bot's text back to speech. We found a couple good voices that are included in Azure's API and decided to use those. There was a few other voices on the API that we tested and felt like they were not to the standard that we were looking

for. Even though we were happy with the final product ideally we would have loved to have more dynamic and realistic voices that we could have added, but the current state of Azure's Speech API does not support this currently.

5.3.2 Network Limitation

As seen in Figure 5.2 a high network speed is desired for the best user experience. The quicker the response time the more fluid the conversation appears to be. In an ideal world everyone would have high speed internet sadly this is not the case. If the user has slow internet e.g 10Mbps you could be waiting a second or two for a response which is not ideal. In our opinion this is a technical limitation that we can not control because we have optimised the transferring of data as best to our abilities so if a conversation with one of the NPCs is slow, it is all down to the user's internet connection.

5.3.3 Models & Animations

Ideally we would have ultra realistic models with 4K textures for all the NPCs and scene objects but we are not trained in modeling, texturing and animation so we made use of what we could find on the internet and the Unity asset store. If we did manage to have highly detailed models and texture we would find ourselves with another problem, that being performance on the Oculus Quest. If we had a VR headset that was not mobile we could have added better models, if we had them. Even though this is a technical limitation we were still extremely happy with the final state and aesthetic of the project.

5.4 Completed Objectives

Based on our goals/objectives from the introduction we believe we completed them all. Below is a list of our completed objectives.

- We developed a project based on requirements that the client was happy with.
- We researched all the latest technologies so that the application would not be out-dated.
- We implemented a robust system based on our research.
- We created a project that we were happy with as well as the client.

Chapter 6

Conclusion

This chapter provides a final outlook on the project, a summary of how we met our initial goals, the insights and knowledge we gained and some thoughts on possible future developments.

6.1 Project goals

From the initial project scope laid out in the Introduction chapter we have met these goals within the required time frame, and improved upon them in multiple ways.

6.2 How we achieved our goals

- Project scope and goals were achieved and exceeded. We have built a fully fledged training application that allows a user to move around a virtual environment in VR and interact with NPC's to check their ticket. All interactions are scored and data is stored securely on a database (MongoDB). A lot of work went into improving the user experience, the look and feel of the application and usefulness of training provided to the user.
- The use of a Mixed Research Methodology allowed us to gain knowledge throughout the whole project. Also using Extreme Programming (XP) as our development methodology ensure code quality was upheld always, as a working build was always available for each meeting.
- We put a big emphasis on testing to utilise the Extreme Programming methodology to its full potential. With this we used both functional and non functional testing methods which helped us develop a robust system. More information on these testing efforts can be found in the System Evaluation chapter.

- We designed and developed the application to be fully modular so that new technologies, new scenarios, new languages and much more could be implemented with further development.

6.3 Findings and Insights gained

Throughout the project there were a number of trial and error issues that came with working with new technologies and hardware, but they all worked out due to our development methodologies. One example of such an error was when building to the Oculus Quest, the Azure Speech Services would stop working once the application was closed and opened again. This was a major problem as the application could only be used once and required a lot of effort to fix. More information on this error and how it was fixed can be found in the Systems evaluation section. With this fixed we felt we could help someone that had a similar error due it's difficulty to fix, so we commented in the original forums as these people could have had the same issue but never known about it. Also, we made an issue on the official GitHub repository for Azure Speech Services so hopefully it could be useful to anyone using our same setup of technologies. Another issue we found during development was with the hosting service Heroku. Heroku had an issue with AIML's session saving feature which we needed to use to save the state of every NPC. The only solution we found was to switch hosting services to PythonAnywhere.

Other insights included learning about new technologies. Working with VR was new to us both and was a great experience, the same can be said about the hosting platforms and back-end technologies we used (Heroku and Flask). We also learnt more about technologies we had previous experience with such as MongoDB and the Unity 3D. The amount of new knowledge we learnt throughout development for these technologies is vast. Ultimately, we believe these insights gained will be valuable in our careers.

6.4 Future development

Throughout development we had multiple different ideas from either brainstorming sessions or through our supervisor meetings that weren't core to our initial project goals. However, they would be features we'd like to look at again if we were to develop the project further. Some of these ideas are as follows.

- Implement language support for the training application. Currently the AIML chatbot can only interpret English, and response to English queries. As the training centre is based in Ireland this is what the client required. However, due to the fact Azure STT and TTS can detect and output multiple languages without intervention it would be possible to implement this feature without changing the project structure or adding many new technologies. A possible way this could be achieved is the server would

convert any user input to English which would be compared with the predefined AIML diction we have developed. This conversion could be done using Google Translate API which is freely available for development.

- More scenarios could be implemented. For our final build we have developed a scene and tutorial focused on checking passengers tickets as outlined by our client. However, as we built the project in a modular way this could easily be expanded to included a new conflict resolution scenario for training security guards or other kinds of workers that deal with the general public.

- A web application could be developed to view the results of each training session, and even compare results on a graph. This would be more suited if being used by multiple training centres and if the project was developed and sold as a product that customers could purchase. Each individual centre could create an account and display their own set of testing data, that the application provides. However for this project as we were working with a single client the application is set up to work with their business only, using a predefined ID for security purposes. However the database could easily be set up to handle multiple business ID's.

6.5 Final thoughts

6.5.1 Matthew

This project was extremely enjoyable to work on and will serve me well in my future career as a machine learning engineer with Arm. I got to work with various new technologies such as speech services, VR technology, cloud hosting providers (Heroku) and machine learning chatbots (Keras). I also improved my current knowledge using technologies such as Unity, Python, C#, Flask and much more. I learned a lot in regards to working in a team, which suits my work flow very well. Working each week with our supervisor and with my project partner using Extreme Programming was useful especially for meeting our deadlines as the project was in a working state after each build. The research that was undertaken throughout the whole project was extremely interesting and it was great to get an insight into some of the exciting technologies out there today. Lastly, I am very happy with how the project turned and how we met our initial goals and exceeded them.

6.5.2 Aaron

I will conclude this project and dissertation with my thoughts. From start to finish, this project has been a complete learning experience. From developing my skills in technologies I was already familiar with like using the Unity engine, programming in C#, Python and more, to learning new technologies like AIML, the keras machine learning library and flask. The main skill I believe I have

obtained from this project is the ability to make various programming language communicate with each other via the use of networking. Trying to make the Unity client communicate correctly and efficiently with the python flask server was the most enjoyable part of the project in my opinion and I believe it will be an invaluable skill in my future career at Ericsson. I thoroughly enjoyed every minute developing this project hence the reason I believe we met all or requirements/ goals and even exeeded them.

Chapter 7

Appendices

Here we will briefly describe the structure of the Github repository and how to install the project:

7.1 Github Repo

Github link: https://github.com/MatthewSloyan/final-year-applied-project-and-minor-dissertation

- Chatbot: This folder contains the python flask server along with the AIML files. There is also a test Keras chatbot that was used during research.
- Dissertation: This folder contains all disseration material.
- Presentation: This folder contains the slide material along with a video presentation.
- Research: This folder contains all the research that was performed prier to development.
- UnityEngine: This folder contains the entire Unity side of the project. This includes all assets scripts etc.

7.2 How to Run

Here we will explain how to download and install the project for the Oculus Quest:

1. Download or clone the GitHub repo
2. Add Unity 2019.2.6f1 to your Unity Hub.
3. Navigate to the "UnityEngine" and add the FinalYearProject folder to your Unity projects in Unity Hub.
4. Launch the project.
5. Once the project is open connect your Oculus Quest to your PC.
6. Navigate to "File" then "Build Settings".
7. Highlight Android under platforms and click "Switch Platform".
8. With your Oculus Quest connected, click "Build and Run".
9. The application will then be build and run on the Oculus.

As the server is hosted on PythonAnywhere there is no need to make anymore change to connect to the server and it should connect by default.

Bibliography

[1] Renata Tesch. *Qualitative research: Analysis types and software*. Routledge, 2013.

[2] Suphat Sukamolson. Fundamentals of quantitative research. *Language Institute Chulalongkorn University*, 1:2–3, 2007.

[3] Thomas J Cheatham and John H Crenshaw. Object-oriented vs. waterfall software development. In *Proceedings of the 19th annual conference on Computer Science*, pages 595–599, 1991.

[4] David Cohen and Mikael Lindvall. Agile software development. 2003.

[5] Lowell Lindstrom and Ron Jeffries. Extreme programming and agile software development methodologies. *Information Systems Management*, 21(3):41 – 52, 2004.

[6] K. M. Mustafa, R. E. Al-Qutaish, and M. I. Muhairat. Classification of software testing tools based on the software testing methods. In *2009 Second International Conference on Computer and Electrical Engineering*, volume 1, pages 229–233, 2009.

[7] H. Freeman. Software testing. *IEEE Instrumentation Measurement Magazine*, 5(3):48–50, 2002.

[8] GitHub. Offical github homepage. https://github.com/.

[9] Unity Technologies. Offical unity documentation. https://docs.unity3d.com/Manual/index.html.

[10] Microsoft. Offical c# documentation. https://docs.microsoft.com/en-us/dotnet/csharp/.

[11] Unity Technologies. Offical unity documentation - web request. https://docs.unity3d.com/ScriptReference/Networking.UnityWebRequest.html.

[12] Eric Rescorla et al. Http over tls, 2000.

[13] Hong W Chen. Javascript object notation schema definition language, March 6 2014. US Patent App. 13/596,694.

[14] Jerry Isdale. What is virtual reality. *Virtual Reality Information Resources http://www. isx. com/˜ jisdale/WhatIsVr. html*, 4, 1998.

[15] Seung Hwan (Mark) Lee, Ksenia Sergueeva, Mathew Catangui, and Maria Kandaurova. Assessing google cardboard virtual reality as a content delivery system in business classrooms. *Journal of Education for Business*, 92(4):153 – 160, 2017.

[16] WILL GEORGIADIS. Oculus quest. *PC Pro*, (298):51, 2019.

[17] Garron Hillaire, Francisco Iniesto, and Bart Rienties. Humanising text-to-speech through emotional expression in online courses. *Journal of Interactive Media in Education*, 2019(1), 2019.

[18] A review of the literature on computerized speech-to-text accommodations. nceo report 414. *National Center on Educational Outcomes*, 2019.

[19] Unity Technologies. Offical unity documentation - speech. https://docs.unity3d.com/ScriptReference/Windows.Speech.PhraseRecognitionSystem.html.

[20] Google. Offical google documentation - speech to text. https://cloud.google.com/speech-to-text/docs.

[21] IBM. Offical ibm documentation - speech services pricing. https://www.ibm.com/cloud/watson-text-to-speech/pricing.

[22] Microsoft. Offical microsoft azure documentation - speech services. https://docs.microsoft.com/en-us/azure/cognitive-services/speech-service/overview.

[23] Microsoft. Offical microsoft azure documentation - speech services pricing. https://azure.microsoft.com/en-us/pricing/details/cognitive-services/speech-services/.

[24] François Chollet et al. Keras: The python deep learning library. *Astrophysics Source Code Library*, 2018.

[25] Tech with Tim. Python chat bot tutorial - chatbot with deep learning (part 1) - youtube. `https://www.youtube.com/watch?v=wypVcNIH6D4`.

[26] User. Ai chat bot in python with aiml — devdungeon. `https://www.devdungeon.com/content/ai-chat-bot-python-aiml`.

[27] Tutorials Point. aiml_tutorial.pdf. `https://www.tutorialspoint.com/aiml/aiml_tutorial.pdf`.

[28] Richard Wallace. The elements of aiml style. *Alice AI Foundation*, 139, 2003.

[29] Ryan Dahl. Offical node.js documentation. https://nodejs.org/en/docs/.

[30] Armin Ronacher. Offical flask documentation. https://flask.palletsprojects.com/en/1.1.x/.

[31] James Gardner. The web server gateway interface (wsgi). *The Definitive Guide to Pylons*, pages 369–388, 2009.

[32] Anubhav Hanjura. *Heroku Cloud Application Development*. Packt Publishing Ltd, 2014.

[33] PythonAnywhere LLP. Offical pythonanywhere documentation. https://help.pythonanywhere.com/pages/.

[34] Kristina Chodorow. *MongoDB : the definitive guide*. O'Reilly, 2013.

[35] MongoDB Inc. mlab overview. https://mlab.com/.

[36] Mixamo. `https://www.mixamo.com/#/`.

[37] Bill Bai. Azure text to speech issue. https://stackoverflow.com/questions/57897464/unity-freezes-for-2-seconds-while-microsoft-azure-text-to-speech-processes-input.