UNIVERSITAT DE BARCELONA

Treball final de grau

GRAU D'ENGINYERIA INFORMÀTICA

Facultat de Matemàtiques i Informàtica
Universitat de Barcelona

A conversational agent that evaluates user experience in a Virtual Reality game

Autor: Changhao Wang

Directora: Dra. Inmaculada Rodríguez Santiago
Realitzat a: Departament de Matemàtiques i Informàtica

Barcelona, June 13, 2022

Abstract

Background. In the last few years, both chatbots and Virtual Reality games have become a hot topic of discussion. On the one hand, chatbots have been used in recent years, especially for customer service, and on the other hand, VR is being promoted thanks to the commitment of large companies such as Facebook and also thanks to lower equipment costs. However, there has been little discussion on the use of the conversational agent in Virtual Reality games to gather user opinions. The main objective of this project focuses on the creation of a conversational agent, capable of interacting with the user through voice and text, and then integrate it into a Virtual Reality game to evaluate user experience in that immersive environment. Specifically, to pave the way of using conversational interaction in a Virtual Reality environment to improve the efficacy of the evaluation process. We design and create a survey agent using Rasa open source platform. In order to evaluate UX, we use the Game Experience Questionnaire. Finally, we integrate the Rasa agent into a game scene in Unity that works in Virtual Reality environment.

Resumen

En los últimos años, tanto los chatbots como los juegos de Realidad Virtual se han convertido en un tema candente de discusión. Por un lado los, los chatbots se están utilizando en los ultimos años sobre todo para atencion al cliente, y por otro se está impulsando la VR gracias a la apuesta de grandes empresas como facebook y gracias también al abaratamiento de costes de los equipos. Sin embargo, poco se ha hablado del uso del agente conversacional en juegos de Realidad Virtual para recolectar opiniones del usuario. El objetivo principal se focaliza en la creación de un agente conversacional, capaz de interactuar con el usuario mediante voz y texto, e integrarlo en un juego en Realidad Virtual para evaluar la experiencia de usuario en un entorno inmersivo. Concretamente, allanar el camino para utilizar la interacción conversacional en un entorno de Realidad Virtual para mejorar la eficacia del proceso de evaluación. Diseñamos y creamos un agente conversacional utilizando la plataforma de código abierto Rasa. Para evaluar la UX, utilizamos el Game Experience Questionnaire. Finalmente, integramos el agente Rasa en una escena de juego en Unity que funciona en un entorno de Realidad Virtual.

Resum

Context. En els últims anys, tant els chatbots com els jocs de Realitat Virtual s'han convertit en un tema candent de discussió. D'una banda, els chatbots s'estan utilitzant en els últims anys sobretot per a atenció al client, i per un altre s'està impulsant la VR gràcies a l'aposta de grans empreses com facebook i gràcies també a l'abaratiment de costos dels equips. Tot i això, poc s'ha parlat de l'ús de l'agent conversacional en jocs de Realitat Virtual per a recol·lectar opinions de l'usuari. L'objectiu principal es enfocalitza en la creació d'un agent conversacional, capaç d'interactuar amb l'usuari mitjançant veu i text, i integrar-lo en un joc a Realitat Virtual per avaluar l'experiència d'usuari en un entorn immersiu. Concretament, aplanar el camí per a utilitzar la interacció conversacional en un entorn de Realitat Virtual per a millorar l'eficàcia del procés d'avaluació. Dissenyem i creem un agent d'enquesta utilitzant la plataforma de codi obert Rasa. Per avaluar l'UX, fem servir el Game Experience Questionnaire. Finalment, integrem l'agent Rasa a una escena de joc a Unity que funciona en un entorn de Realitat Virtual.

Contents

Index of figures

1 Introduction

Nowadays, it feels like every week another ground-breaking invention or idea is revealed. The world is growing fast and so is technology. For a few years, the video game sector has been experiencing its golden age. It is not only a hobby, but it has become an industry in continuous growth that adds more followers every day and demands qualified professionals in different areas. Due to the highly competitive nature of the sector, game companies have been continuously improving and adopting new technologies and enhancements in their business. For the game companies, the goal is, on the one hand, to seek methods to improve their game performance and quality and, on the other, aspire to gather opinions about players' experience to improve the design of the games. In the past, most companies would incline to gather opinions in offline festival events, such as San Diego comic-con or similar game con. Despite that, nowadays there are a lot of companies that have extended the channel to online too, through forum, email, live-streaming, twitter votes, etc.

Although, most of the games in the market are mainly from platforms like PCs, video game consoles or mobiles, while virtual reality games only take a small portion of the big game market. Since now is just the beginning for virtual reality games, it's really necessary to study how to do UX evaluation in this kind of games. The advantage of VR games is that they are played using VR headset which makes the immersion so different compared to the traditional game platform like PC, PlayStation, mobile, etc. During the game session, users can actually feel like playing in a real world, that's the potential and capability of virtual reality games.

There are already works that discuss this subject, particularly the one that pushes this project forward [16]. The article proposes the use of questionnaires when users are playing the game in order to gather users' fresh opinions and to avoid the possibility of forgetting some significant contents and the feelings on that particular moment. All this gave birth to this project that aims to study how to evaluate user experience using conversational agents in virtual reality games. In order to achieve the goal, there must be a VR game and there must be a conversational agent.

Regarding the VR game, this project extends from two previous projects carried out by UB students [12][11] that created and enhanced a virtual reality game that aim to teach children logic thinking with programming. This was thought as the final piece of the puzzle, since we need to implement the bot and integrate it in the game. For the conversational agent, after some studies and trials, the RASA open source platform is chosen because it's totally free and covers all the necessity of this project, and not to mention the big friendly community that helps each other out.

So, the motivation is all about designing and implementing a chatbot, then integrating it into a virtual reality game to evaluate the user experience through in-world questionnaires.

2 Goals

2.1 General goal

The main goal of this project is to study in-world evaluation of user experience using conversational agents in virtual reality games. The main goal can be subdivided into two separate goals: the first one is to design a survey chatbot and the second one is to integrate it into a VR game.

To reach these goals, it will be necessary to acquire the skills and knowledge to work with a self designed conversational agent, in addition to acquiring the basic knowledge to work with new frameworks and game engines such as those used in this project.

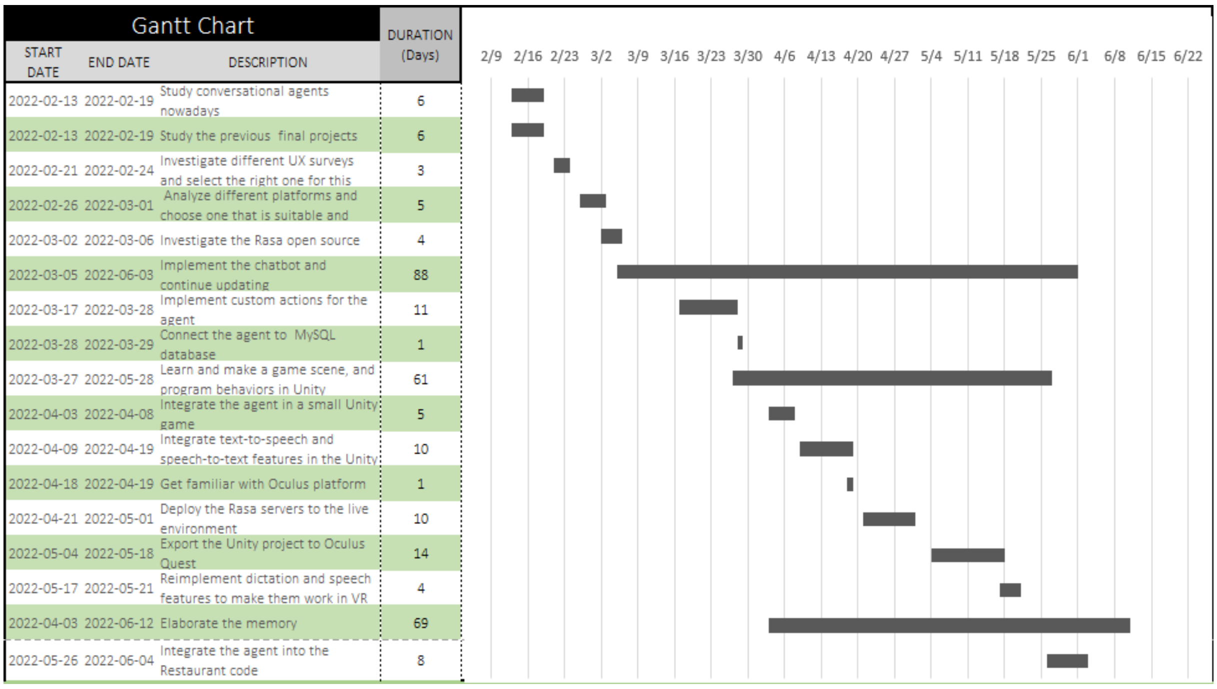
2.2 Specific goals

The main goal is broken down in subgoals that are defined in the following:

- Study conversational agents in this day and age.
- Investigate different UX surveys and select the right one for this project.
- Analyze different conversational platforms and choose one that is suitable and compatible for the implementation of the conversational agent.
- Study the chosen conversational platform and implement the chatbot.
- Learn how to work with MongoDB (database).
- Learn how to create game scenes and to program behaviors in the Unity game engine.
- Investigate how to integrate a conversational agent in a Unity scene and make it work on VR.
- Study how to integrate text-to-speech and speech-to-text features in Unity.
- Get familiar with Oculus platform.
- Investigate how to export a Unity project to Oculus Quest.
- Investigate how to deploy the conversational agent to the live environment so that we can access to it in the VR game or anywhere we want.
- Study the previous final projects that developed a VR game and integrate a conversational agent that engages in a conversation with the user during the game.
- Learn how to integrate the agent into the Restaurant Code VR game using Unity.

2.3 Scheduling

This section shows a Gantt chart that represents the way in which the tasks were distributed along the semester.



Gantt Chart

START DATE	END DATE	DESCRIPTION	DURATION (Days)
2022-02-13	2022-02-19	Study conversational agents nowadays	6
2022-02-13	2022-02-19	Study the previous final projects	6
2022-02-21	2022-02-24	Investigate different UX surveys and select the right one for this	3
2022-02-26	2022-03-01	Analyze different platforms and choose one that is suitable and	5
2022-03-02	2022-03-06	Investigate the Rasa open source	4
2022-03-05	2022-06-03	Implement the chatbot and continue updating	88
2022-03-17	2022-03-28	Implement custom actions for the agent	11
2022-03-28	2022-03-29	Connect the agent to MySQL database	1
2022-03-27	2022-05-28	Learn and make a game scene, and program behaviors in Unity	61
2022-04-03	2022-04-08	Integrate the agent in a small Unity game	5
2022-04-09	2022-04-19	Integrate text-to-speech and speech-to-text features in the Unity	10
2022-04-18	2022-04-19	Get familiar with Oculus platform	1
2022-04-21	2022-05-01	Deploy the Rasa servers to the live environment	10
2022-05-04	2022-05-18	Export the Unity project to Oculus Quest	14
2022-05-17	2022-05-21	Reimplement dictation and speech features to make them work in VR	4
2022-04-03	2022-06-12	Elaborate the memory	69
2022-05-26	2022-06-04	Integrate the agent into the Restaurant code	8

Figure 1: Gantt chart of this project. Source: Own

3 Background

3.1 Virtual Reality games

The initial game project was created in a bachelor final project by Iván Gómez. The aim of the project was to teach young children aged 12 to 16 to learn and understand the logic behind programming. So for that, the Cooking Code game was created to achieve this goal. The game is set in a future world, where robots and humans live together and even the language they speak is mixed. The player will have to cook for everyone as the chef of a restaurant, and the clients will be the robots. The game consists in players needing to understand all the orders for hamburgers that are written in pseudocode and then assemble the hamburger correctly before the time ends.

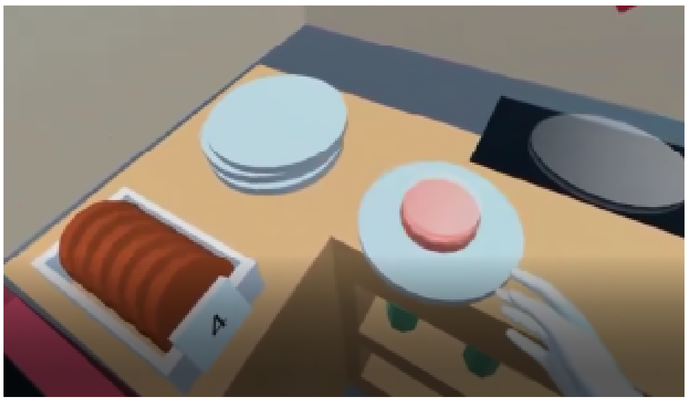
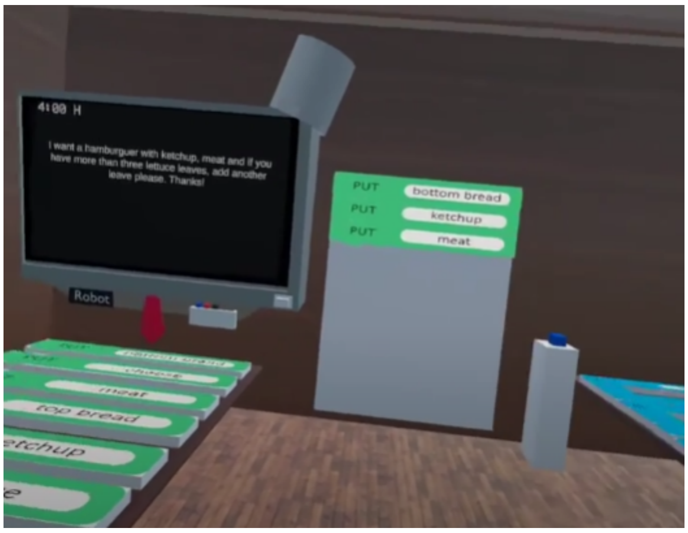


Figure 2: The final project of Iván Gómez. Source: [12]

Based on the initial game, the second one was elaborated also in a bachelor final project by Alex Fuentes. The aim of the project was to extend the original game and add functions that allow users to write pseudocode too and take one more step to understand better the logic of programming. During the game, players will have to act as waiters and go around the restaurant to pick up orders and then translate the natural language of received orders to pseudocode.


4:00 H

I want a hamburguer with ketchup, meat and if you have more than three lettuce leaves, add another leave please. Thanks!

Figure 3: The final project of Alex Fuentes. Source: [11]

3.2 In-game UX questionnaires

This section shows two UX questionnaires that are used to evaluate user's game experience and explains which one is used in this project.

3.2.1 Game Experience Questionnaire

The Gaming Experience Questionnaire (GEQ) [17] captures the gaming experience based on a number of elements. It aims to examine the gaming experience with others (i.e the social experience), as well as the post-game and in-game experience. It has a modular structure composed of: a core questionnaire, a *social presence* module, an *in-game* model and a *post-game* module. These modules are intended to be tested immediately after a game session has ended, except for in-game version, in the order listed above. In order to make the measure to be more robust, each module uses certain items per component to compute the scores.

The core questionnaire evaluates gaming experiences as scores bases on seven components: sensory and imaginative immersion, tension or annoyance, competence, positive affect, negative affect, challenge and flow. It uses five items per component, so that the component scores are computed as the average of its items.

The social presence module evaluates psychological involvement with others co-players [1]. It is based on three components: psychological involvement- empathy, psychological involvement - negative feelings and Behavioural involvement. It uses between five or six items per component depending on its type, so that the scores are computed as the average of its items.

[1]Co-players can be in-game characters or others online player

The post-game module evaluates how players feels after the game finished bases on four components: positive experience, negative experience, tiredness and returning to reality. It uses between two and six items per component depends on the type, so that the component scores are computed as the average of its items.

The in-game version evaluates how players feels when playing the game at multiple intervals[2]. It is based on seven components: sensory and imaginative immersion, tension or annoyance, competence, positive and negative affect, challenge and flow. It uses two items per component, so that the component scores are computed as the average of its items (see Figure 4).

[2]Intervals can be in specific stages, levels, times, etc.

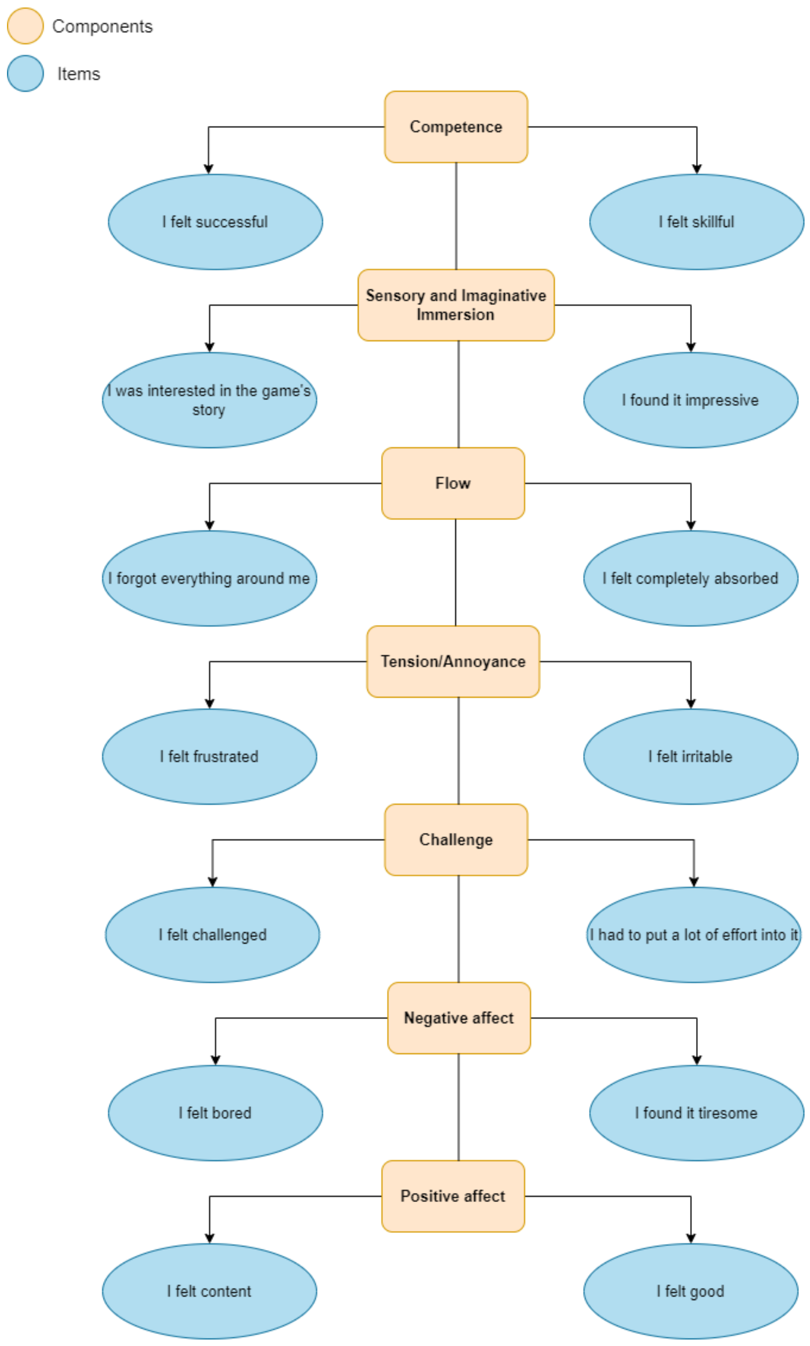


Figure 4: GEQ in-game version module. Source: [17]

To summarize, the first and second parts of GEQ are probes of the player's feelings and thoughts while playing the game; the last part, the post-game module, evaluates how the players feel after they have finished playing; and for the in-game version, it's a succinct version of the core questionnaire that is evaluates how the players feel during a game session. Considering the type of the game and the timing of the survey, the in-game version is chosen for this project because the final goal is to evaluate the UX during the game session.

3.2.2 Game User Experience Satisfaction Scale Questionnaire (GUESS)

The game user experience satisfaction scale (GUESS) captures video game user experience by responding to 55 items that accesses nine construct subscales: usability, narratives, play engrossment, enjoyment, creative freedom, audio aesthetics, personal gratification, social connectivity, and visual aesthetics. [28].

The GUESS is recommended to be used to compare different games of the same genre, and the one with the highest score should be considered as the most satisfying game. It also provides the possibility of removing the narratives and social components when evaluate, if the game doesn't have them, but the validity of the scale in these circumstances needs to be done in further research [28, pg.1238-1239].

Additionally, there is a second version of the game user experience satisfaction scale [19] where aim to develop a shorter version of questionnaire for use in iterative video games, testing and research by truncating the original 55-items to 18-items that access nine constructs as defined in the initial version. Compared to the initial one, the GUESS-18 is recommended when games are iterative and require quick responses [19, pg.9][29].

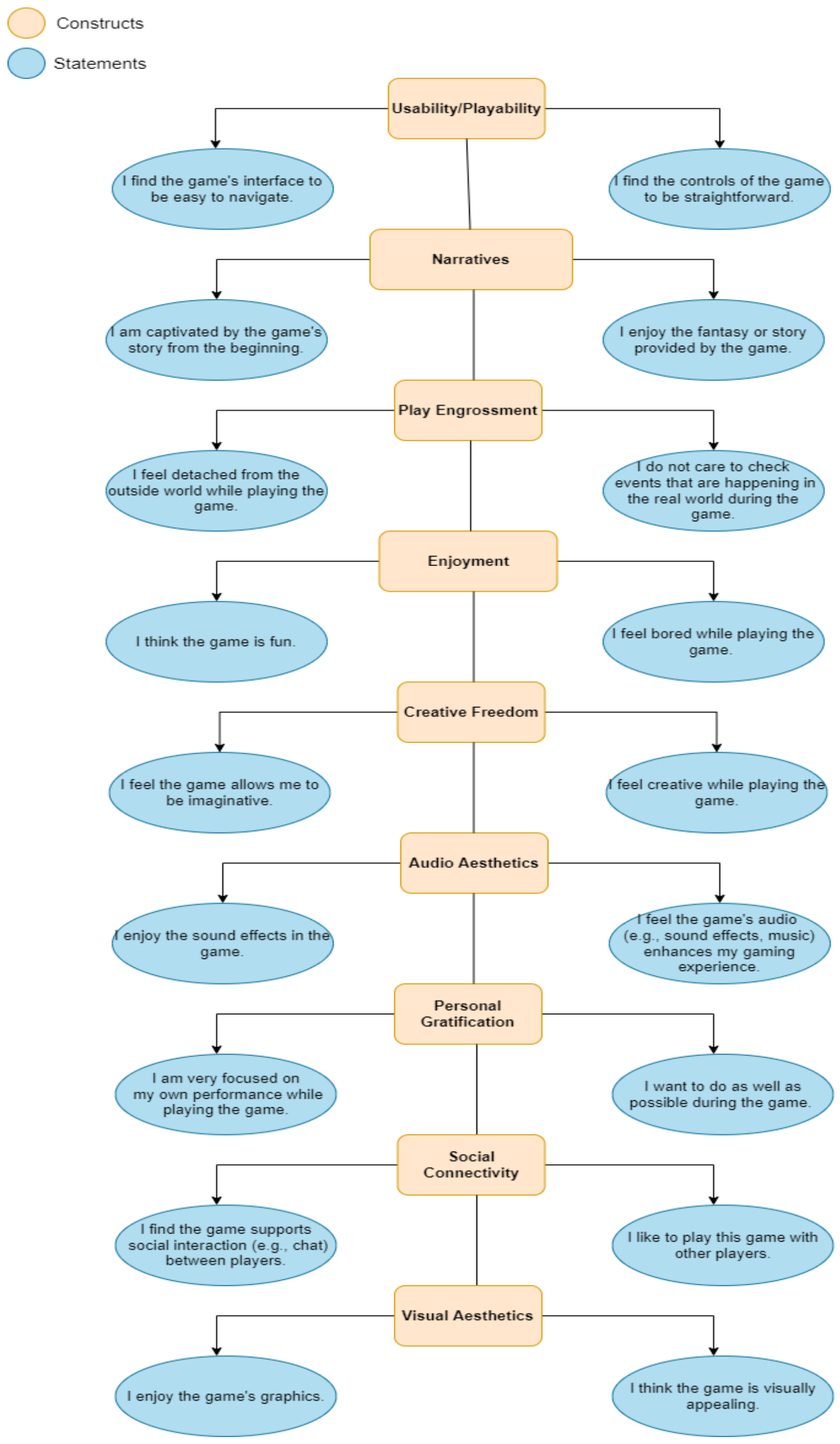

Constructs
Statements
Usability/Playability
I find the game's interface to be easy to navigate.
I find the controls of the game to be straightforward.
Narratives
I am captivated by the game's story from the beginning.
I enjoy the fantasy or story provided by the game.
Play Engrossment
I feel detached from the outside world while playing the game.
I do not care to check events that are happening in the real world during the game.
Enjoyment
I think the game is fun.
I feel bored while playing the game.
Creative Freedom
I feel the game allows me to be imaginative.
I feel creative while playing the game.
Audio Aesthetics
I enjoy the sound effects in the game.
I feel the game's audio (e.g., sound effects, music) enhances my gaming experience.
Personal Gratification
I am very focused on my own performance while playing the game.
I want to do as well as possible during the game.
Social Connectivity
I find the game supports social interaction (e.g., chat) between players.
I like to play this game with other players.
Visual Aesthetics
I enjoy the game's graphics.
I think the game is visually appealing.

Figure 5: GUESS short version. Source: [19]

3.2.3 Conclusion

Both of the two versions of GUESS are rated using a seven-point likert-scale [22] (1 = Strongly disagree, 7 = Strongly agree). An overall score is calculated by summing the scores that are obtained by averaging the ratings within the subscales. The full GUESS normally takes around 5 to 10 minutes to complete and the shorter version, takes around 3-5 minutes approximately instead. As for the GEQ in-game version, the components scores are calculated as the average value of its items.

After comparing these two different type of questionnaires that validate player's game experience, the game experience questionnaire has been chosen because of its simplicity, its short duration and, the most important factor, its in-game version questionnaire, since after all our goal is to evaluate player's feeling during a virtual process, in where users are immersed in the experience, which is exactly what the GEQ offers [4].

4 Related concepts

This section explains the definition of the related concepts that will be referred along this document, although some of them are explained in more detail on section 7.

UX

User Experience, also known as UX, is to create a meaningful and intuitive experience for users. It adopts user's perspective and aims to provide a memorable, complete, and satisfactory experience. In this project, the UX in virtual reality is evaluated using a chatbot.

Chatbot

A chatbot, also known as conversational agent, is an artificial intelligence program that simulates human-like conversation in real time via text message or voice. Chatbots always work the same routine, they interpret what users want, process their requests, and give responses. In this project, a chatbot is used for evaluating UX through the GEQ.

Intent

In the context of conversational interactions, an intent is the goal or purpose of the user's input. Normally, developers need to add examples, also known as synonyms, for each intent that chabots need to address. These examples are then used by the chatbot to figure out different ways in which people might express the same intent. With that being said, we can think of intents as labels for a group of examples that express a common goal or purpose. To take an example, the following expressions might be defined as examples of the "greeting" intent: "Hi", "Hello", "Hey", etc. In this project, intents represent user's answers to the questions of the survey.

Entity

Entities are the pieces of information that chatbots can extract from user inputs and later on use those details in a specific context when responding back to the user or running specific actions. An entity can be any kind of piece of information that is relevant and important to the agent. To take an example for a simple calculator agent, if the user asks "What is the result of 5{operand} add{operator} 5{operand}", where operand and operator are the entities and their values are *5* and *add*, the agent could use the extracted information to perform an addition operation, then answers "The result of 5 add 5 is 10". In this project, entities haven't been considered necessary at the design phases because we use fixed responses (intents) from the GEQ in which we don't need any other value to be extracted, so eventually it's not used at the implementation phase.

Action

An action is what the bot is expected to perform to answer to the user's message. One of the most common actions are responses, which are simple messages that our bot can send back to the users. To take an example, if the user sends to our bot an intent "Hey" then the bot will reply with "Hello, how are you?" (this is a response). In this project, all of the chatbot's responses are actions such as greetings action, questions action, say goodbye action, etc.

Slots

In simple words, slots are the chatbot's memory that enables our conversational agents to collect and store important pieces of information and use them in a specific context. To take an example, if the agent asks the user what his name is and the user answers Alicia, then if at some point way later in the conversation the user asks "Do you remember my name?", the agent would be able to remember what she said and answers Alicia. In this project, slots are not used as part of implementation since it was considered as unnecessary at the design phase because we want to store all the answers in a remote database and we don't need slots to store anything else.

NLP

NLP, stands for Natural Language Processing which includes NLU and NLG, enables computers to understand human language in both written and verbal forms using deep learning techniques to complete tasks. Typical examples for that are things like conducting a conversation in a chat bot, language translation, etc. It works through the identification of entities and identification of word patterns using methods like stemming, lemmatization and tokenization.

NLU

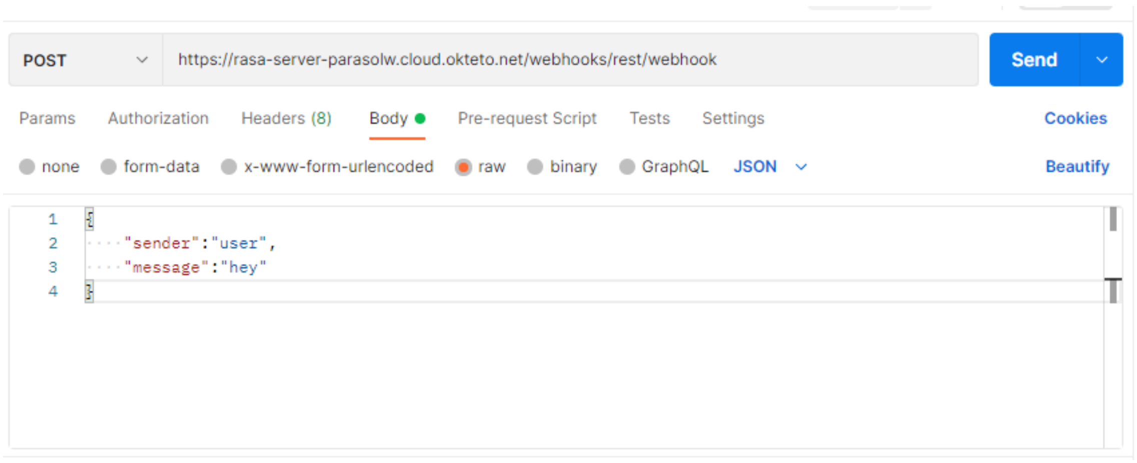
NLU, stands for Natural Language Understanding, uses semantic and syntactic analysis of speech and text to determine the meaning of a sentence. In other words, NLU is all about a subset of NLP processes that deal specifically with converting data from input into something the computers can understand. In Rasa conversational platform, for example intents and their examples are used as training data for the agent's Natural Language Understanding model [33].

NLG

NLG, stands for Natural Language generation, it's the process of producing a human language text response based on some data input, basically it focuses on enabling computers to write or speak in human language. Rasa provides the possibility of using an external HTTP NLG server to generate responses and sends it back to the user. In this project, we don't use NLG, for simplicity, we have predefined responses.

Rasa server

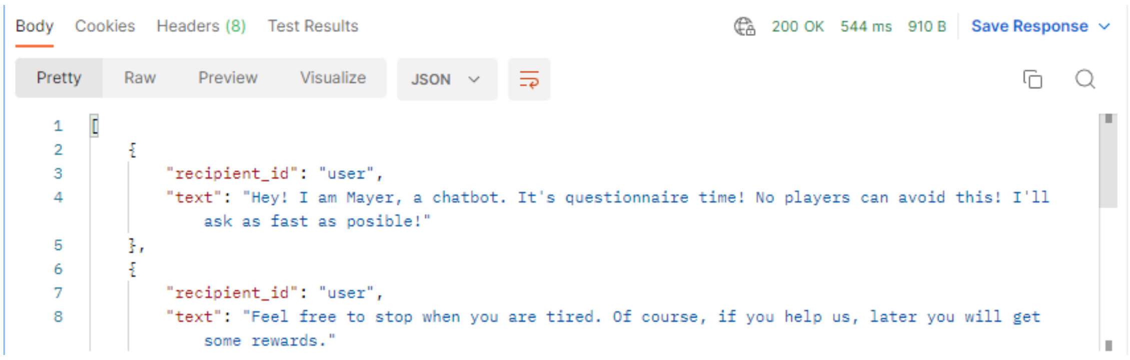
The Rasa server is where the developers communicate with the chatbot using REST API. A REST API, stands for Representational State Transfer, is the interface that allows the exchange of information between two independent software components [13]. The URL to which send the message is, by default, *"http://localhost:5005/"* where *localhost* is the host and *5005* is the port. In order to send a message to the Rasa server, we use POST requests where we specify the URL and the body that contains the sender name and the message to deliver (see Figure 6).


POST https://rasa-server-parasolw.cloud.okteto.net/webhooks/rest/webhook

```
{
    "sender":"user",
    "message":"hey"
}
```

Figure 6: Example of a POST request. Source: Own

If successful, we receive a response as showed in Figure 7.


200 OK 544 ms 910 B

```
[
    {
        "recipient_id": "user",
        "text": "Hey! I am Mayer, a chatbot. It's questionnaire time! No players can avoid this! I'll
            ask as fast as posible!"
    },
    {
        "recipient_id": "user",
        "text": "Feel free to stop when you are tired. Of course, if you help us, later you will get
            some rewards."
```

Figure 7: Example of a bot responses (two consecutive actions) to an intent. Source: Own

Rasa action server

The Rasa action server runs along with the Rasa server, and it is used to run the custom actions (see more details in section 7) for Rasa conversational agents [32]. In the case that the custom actions are implemented and triggered by an intent, the Rasa server sends a POST request to the action server, and it will return a JSON payload of responses. Afterwards, the Rasa server returns the responses to the user. In this project, several custom actions are implemented for performing database queries. These queries are in charge of storing data related to user information and conversations with the agent.

Game engagement

A good game design isn't about if players are having fun because fun is fleeting, instead a good design is about engaging players psychologically which allows for fun to emerge from the experience and keep the players wanting more [14]. The game engagement can be interpreted as players' commitment to the gaming activities. For the game developers, it's never an easy task to decide the right moment for players to answer their in-game survey because depending on the high-, medium- and low-engagement-stages, gamers respond differently to motivations which might influence the accuracy of the questionnaire [15]. In this project, we plan to create conversations between the agent and the player in several game-stages and preferably during the medium- or low-engagement-stage to obtain the relevant information avoiding interruption during high-engagement-stage to ensure the reliability and validity of the questionnaire [30].

Conversational turn

In the world of conversational agents, there are two types of conversational turn: the single/one-turn interaction and multi-turn interaction (also know as follow-up). The single-turn interaction is just one back-and-forth conversation, meaning the user talks and bot responses, or vice-versa. Although in reality, some questions cannot be answered in a single turn. When designing a conversational agent, the user can ask a question that needs to be filtered or refined to determine the correct answer. In this project, one-turn interaction is mainly used in order to make the bot pausable and resumable. We don't need to prepare various multi-turn conversation to handle different situations, instead just break it into several smaller single-turn interactions.

5 Conversational platforms

In this section, we analyze and compare technologies and solutions that can be used to solve our problem. There are many frameworks in the sector, and new platforms continuously appearing more and more often as the big companies see business opportunities in the world of conversational assistant.

In this section, we compare the features and possibilities of various frameworks whose functionality is to create conversational agents, in other words chatbots. So, the discussion will focus on three platforms that have made their name in this market and compare them to the one we chose, Rasa open source: IBM Watson Assistant, DialogFlow and Microsoft Azure.

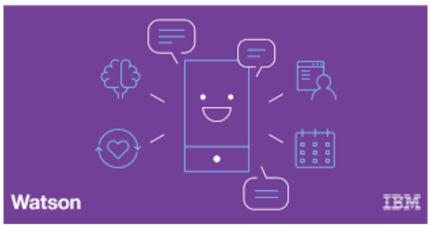


Figure 8: IBM Watson Assistant. Source: http://tossa.com.mx/que-es-ibm-watson-assistant/

Watson Assistant is a conversational agent creating service hosted on IBM Cloud and developed in 2018 [18][3]. The main advantage that IBM Watson assistant offers is that the owner of the data, unlike other platforms, is not IBM but the company that created the agent. In other words, if the University of Barcelona uses this assistant, the data collected by the assistant will not belong to the IBM database, but to UB's, something that wouldn't happen if we used Google Assistant or others. Users can interact with assistants through text or voice channels. Also offers the possibility of customizing the interface making users unaware of the existence of the Watson Assistant. As well as the compatibility it provides with Unity, where the Cooking Code and Restaurant Code were built. The reason why it was excluded in this project is because the expensive paid plan that cost 132,61€/ month and it didn't allow users to use the free plan as described. After several trials with different accounts, identities and credit cards, this option was finally discarded.

5.1 Dialogflow

Dialogflow

Figure 9: Dialogflow. Source: https://dialogflow.cloud.google.com/

In 2016, Google announced the acquisition of api.ai, the platform from which the project known as Dialogflow was born [21]. Later on, it became one of the most popular frameworks in the market because of the easy use of it and the company behind it. DialogFlow stands out among other technologies as it offers a wide range of conversational interfaces, from Google Home to phones or wearables. In addition, it supports more than 14 languages and offers a usable free plan unlike IBM Watson Assistant. The reason why it was discarded from this project is because the DialogFlow version 2 is no longer compatible with Unity, which causes the impossibility to integrate the agent into a Unity game project [2].

5.2 Microsoft Azure

In recent times, the big technology companies are working very hard and investing a lot of resources in a new revolution based on Artificial Intelligence, Machine Learning, Big Data or Virtual Reality. Microsoft, always at the forefront when it comes to innovation, has been one of the companies that has made the strongest commitment to these technologies, offering numerous services supported by Azure, such as the chatbot technology. With Microsoft Azure Bot Service users can create, connect and manage bots with the help of the Microsoft Bot Framework and the BotBuilder SDKs, providing artificial intelligence to any website, Office 365 tool, or use it in other applications such as Slack or Facebook Messenger in order to improve the user experience for the customers. The reason why it was not used in this project is because Microsoft Azure doesn't help users manage their cloud data center, which means users need to know how to do server monitoring and patching that might require a lot of time to learn them.

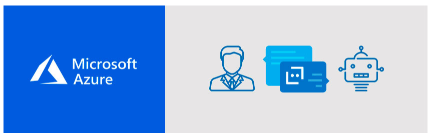


Figure 10: Microsoft Azure Bot Service. Source: https://www.inforges.es/post/microsoft-azure-bot-service

5.3 Rasa Open Source

RASA is an open-source conversational AI framework based on Python and NLU (Natural Language Understanding) which provides the possibility to create custom chatbots. Being open-source provides many advantages since the users data used by the agent does not go through a third party framework and even the software is no longer maintained, the chatbots can work without any problems. In addition to that, it also offers the user the possibility to train models and add custom actions to achieve the specific requirements. Rasa has two main components:

- RASA NLU (Natural Language Understanding): Its mission is to classify intents to understand what the user's sentence means, and extract entities that should take note of.
- RASA Core: This part is in charge of managing the dialog, it takes structured input from the NLU and predicts what action to take at each moment using a machine learning model.

Figure 11: Rasa Open Source. Source: https://github.com/RasaHQ/rasa

In other words, Rasa NLU's job is to interpret the input provided by the user and Rasa Core's job is to decide the next set of actions (responses) performed by the chatbot. Rasa NLU and Rasa Core are independent of each other and can be used separately.

6 Analysis

This section details the functional and non functional requirements and use cases of the project, describing all the possible interactions between the user and the game in order to achieve the objectives. This use case specification makes it easy to understand the system and expresses the intent with which the user will interact with the application, and lays a firm base for beginning the design phase.

6.1 Functional and non functional requirements

In this section, the purpose is to clearly and concisely detail each of the goals presented in this project, as well as all its features. This information has been specified in different meetings established with the project tutor, who in this case takes the user role. As we will see below, some initial requirements have been modified or directly eliminated, since throughout all the meetings with the user it was determined that some functionalities could not be carried out, or had to be carried out in another way, in mostly for the purpose of facing obstacles as well as improving the final result. Therefore, the development of the Rasa conversational agent will cover and have the following main functions:

- The chatbot shall be able to interact with the user in some specific stages of the game.
- The user shall be able to interact with the chatbot by texting in the specific input field.
- The user shall be able to speak with microphone and dictate their answers.
- The user shall be able to skip questions that result hard to answer, and consequently the agent will not show them again.
- The user shall be able to pause the conversation anytime he/she want and the unanswered question will still remain in the questions pool. A question pool is a set of questions that it is used to store all the questions used in the GEQ in-game version.
- The user shall be able to receive a reward related to the game after answering the entire questionnaire.

  For the non functional requirements, the interaction of the chatbot and the user has to be interactive and in real time, that is, the connection between our game and the Rasa server has to be fast. The network issues could caused problems as well, since the features of speech to text and vice-versa require a stable connection. The chatbot design as well as its ability to communicate with the user must be realistic to convey trust to the user.

6.2 Use cases

Once we are cleared about the requirements and needs that the system must be able to cover, we design the use cases. A use case is a description of the steps or activities that must be performed to carry out a process. The characters or entities that participate in a use case are called actors. In the context of software engineering, a use case is a sequence of interactions that will develop between a system and its actors in response to an event initiated by a main actor on the system itself.
Use case diagrams serve to specify the communication and behavior of a system through its interaction

with users and/or other systems. Or what is the same, a diagram that shows the relationship between the actors and the use cases in a system. A relation is a connection between the elements of the model, for example specialization and generalization are relations. Use case diagrams are used to illustrate the requirements of the system by showing how it reacts to events that occur in its scope or itself (see Figure 12).

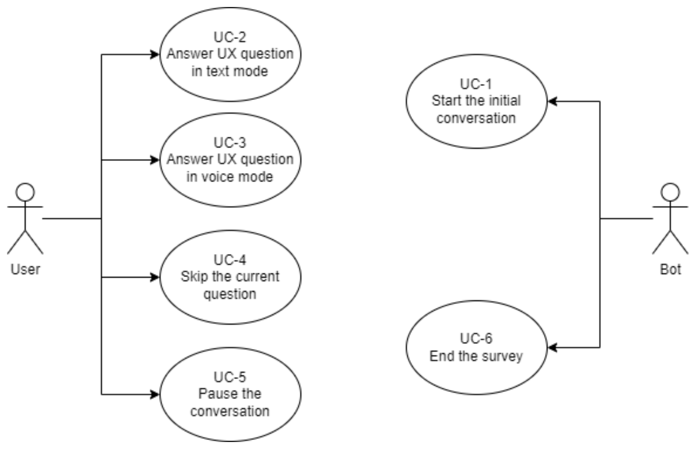

UC-2
Answer UX question in text mode
UC-3
Answer UX question in voice mode
UC-4
Skip the current question
UC-5
Pause the conversation
User
UC-1
Start the initial conversation
UC-6
End the survey
Bot

Figure 12: Use cases diagram. Source: Own

In addition, each use case showed in the diagram will be complemented with a textual description of it, thus achieving a more detailed and formal explanation of each one. The template used for the textual description of the use cases and the nomenclature is the following:

- Name: The name of the use case.
- Actor: The actor that interacts with the current use case. In our this case, we have two actors: user and bot.
- Description: A brief description of the objective.
- Preconditions: Conditions that must be fulfilled beforehand to carry out the operation.
- Normal flow: Description by phases for the fulfillment of the use case.
- Alternative flow: Describes exceptions or deviations from the normal flow.
- Postconditions: State of the system after the execution of the operation.

Name	UC-1 Start the initial conversation
Actor	Bot
Description	The chatbot is triggered and starts the conversation by introducing itself and asking the first question.
Preconditions	None
Normal flow	1. The user moves to the next game-stage where Unity sends a message to the Rasa agent indicating the start of the survey. 2. The Rasa server receives the message from Unity. 3. The Rasa server analyzes the message and makes predictions about which action should be taken as response. 4. The Rasa server predicts that the message is an internal intent and correlates with the introduction action. 5. The Rasa server sends an introduction and the first question as responses to the user. 6. The Rasa server moves to the wait state. 7. The system plays the response using text-to-speech technique.
Alternative Flow	1a. If the user loses internet connection, the message won't reach out to the Rasa chatbot, and an error will pop up saying "Network error! Please check your internet connection".
Postconditions	The chatbot has successfully established the initial conversation with the user.

Name	UC-2 Answer UX question in text mode
Actor	User
Description	User wants to interact with the chatbot in text mode (one-turn interaction).
Preconditions	UC-1 Start the initial conversation
Normal flow	1. The user receives a question from the chatbot. 2. The user writes his answer in the input field. 3. The user sends the answer to the chatbot by clicking on the send button. 4. The Rasa server (chatbot) receives the answer. 5. The Rasa server sends the answer to the action server. 6. The Rasa action server stores the answer into the database. 7. The Rasa server sends the next question as response message to the user. 8. The Rasa server moves to the wait state. 9. The system plays the response using text-to-speech technique.
Alternative Flow	2a. If the user is inactivated for a period of time, the agent will interprets the inactivity as a sign to pause the conversation. 3a. If the user loses internet connection, the message won't reach out to the Rasa chatbot, and an error will pop up saying "Network error! Please check your internet connection".
Postconditions	The user has successfully interacted with the chatbot by texting.

Name	UC-3 Answer UX question in voice mode
Actor	User
Description	User wants to interact with the chatbot in voice mode (One-turn interaction).
Preconditions	UC-1 Start the initial conversation
Normal flow	1. The user receives a question from the chatbot. 2. The user dictates his answer by clicking on the record button. 3. The speech-to-text system converts user's speech to text. 4. The user sends his message to the chatbot by clicking on the send button. 5. The Rasa server (chatbot) receives the answer. 6. The Rasa server sends the answer to the action server. 7. The Rasa action server stores the answer into the database. 8. The Rasa server sends the next question as response message to the user. 9. The Rasa server moves to the wait state. 10. The message is spoken by the text-to-speech system.
Alternative Flow	2a. If the user is inactivated for a period of time, the agent will interprets the inactivity as a sign to pause the conversation. 2b. If the user loses internet connection, the system will show a message to indicate that the speech-to-text module is not working properly. 4a. If the user loses internet connection, the message won't reach out to the Rasa chatbot, and an error will pop up saying "Network error! Please check your internet connection".
Postconditions	The user has successfully interacted with the chatbot using microphone.

Name	UC-4 Skip the current question
Actor	User
Description	User finds hard to answer the question and decide to skip this one.
Preconditions	UC-1 Start the initial conversation
Normal flow	1. The user receives a question from the chatbot. 2. The user intends to skip this question because he has no idea what to answer. 3. The user writes ”skip”, or similar message, in the input field. 4. The user sends the message to the chatbot by clicking on the send button. 5. The Rasa server receives the message. 6. The Rasa server interprets the user intention and marks the question as skipped so it won't be asked again. 7. The Rasa server sends the next question as response message to the user. 8. The Rasa server moves to the wait state. 9. The system plays the response using text-to-speech technique.
Alternative Flow	3a. If the user is inactivated for a period of time, the agent will interprets the inactivity as a sign to pause the conversation. 4a. If the user loses internet connection, the message won't reach out to the Rasa chatbot, and an error will pop up saying ”Network error! Please check your internet connection”.
Postconditions	The user has successfully skip the current question.

Name	UC-5 Pause the conversation
Actor	User
Description	User wants to take a break, so he wants to pause the agent
Preconditions	UC-1 Start the initial conversation
Normal flow	1. The user receives a question from the chatbot. 2. The user feels tired, so he decide to take a break. 3. The user writes ”pause”, or similar message, in the input field. 4. The user sends the message to the chatbot by clicking on the send button 5. The Rasa server receives the message. 6. The Rasa server interprets the user intention. 7. The Rasa server sends the farewell message as response to the user. 8. The Rasa server moves to the wait state. 9. The system plays the response using text-to-speech technique.
Alternative Flow	3a. If the user is inactivated for a period of time, the agent will interprets the inactivity as a sign to pause the conversation. 4a. If the user loses internet connection, the message won’t reach out to the Rasa chatbot, and an error will pop up saying ”Network error! Please check your internet connection”.
Postconditions	The user has successfully paused the interaction with the chatbot.

Name	UC-6 End the survey
Actor	Bot
Description	The user answered all the questions so that the bot wants to end the conversation
Preconditions	
Normal flow	1. The Rasa server receives the user's answer to the last question of the survey. 2. The Rasa server interprets that all the questions are answered. 3. The chatbot send the appreciation and say goodbye messages to the user. 4. The messages are spoken by the text-to-speech. system.
Alternative Flow	3a. If the user loses internet connection, the message won't reach out to the user, and an error will pop up saying "Network error! Please check your internet connection".
Postconditions	The user has successfully paused the interaction with the chatbot.

7 Design

This section describes both the survey agent design and the software design. On the one hand, it details the essential components of the agent designed using Rasa Open Source and their functionalities. On the other hand, it details the scripts used in Unity in order to control the logic of the system, perform features like speech to text and text to speech, and change the game-stages.

7.1 Architecture diagram

The conversational system of the project showed in Figure 13 consists of three modules:

- Chatbot: This module is responsible for generating responses to messages sent by users and communicating with the database.
- Unity game: It acts as the interface between the user and the agent. It's also the brain of the conversational system, since it controls the whole dialogue flow using its agent behaviour. It's also responsible for generating REST requests to send to the Rasa server and extracting the received JSON payload responses to show them to the user.
- Database: This module is responsible for storing the received answers from the chatbot.

Next, we explain the first two models, the chatbot and the unity game, and in section 8.3.2 we detail the database.

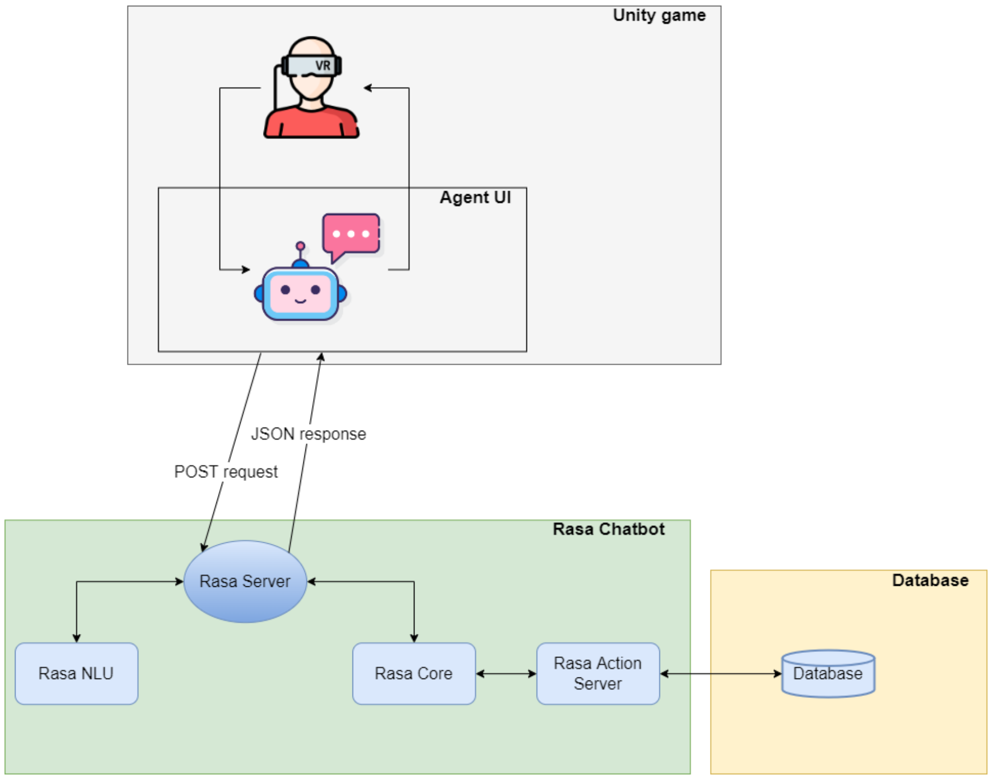


Figure 13: The Architecture Diagram. Source: Own

7.2 Survey chatbot designed in Rasa

Regarding to the agent design, Figure 14 presents an detailed architecture diagram that shows a physical view of the Rasa open source chatbot. We can see the different modules of our agent and within each module the components that make it up and their relationships. This diagram describes a complete single-turn interaction with an agent, from the chatbot receiving the message to the agent sending the response.

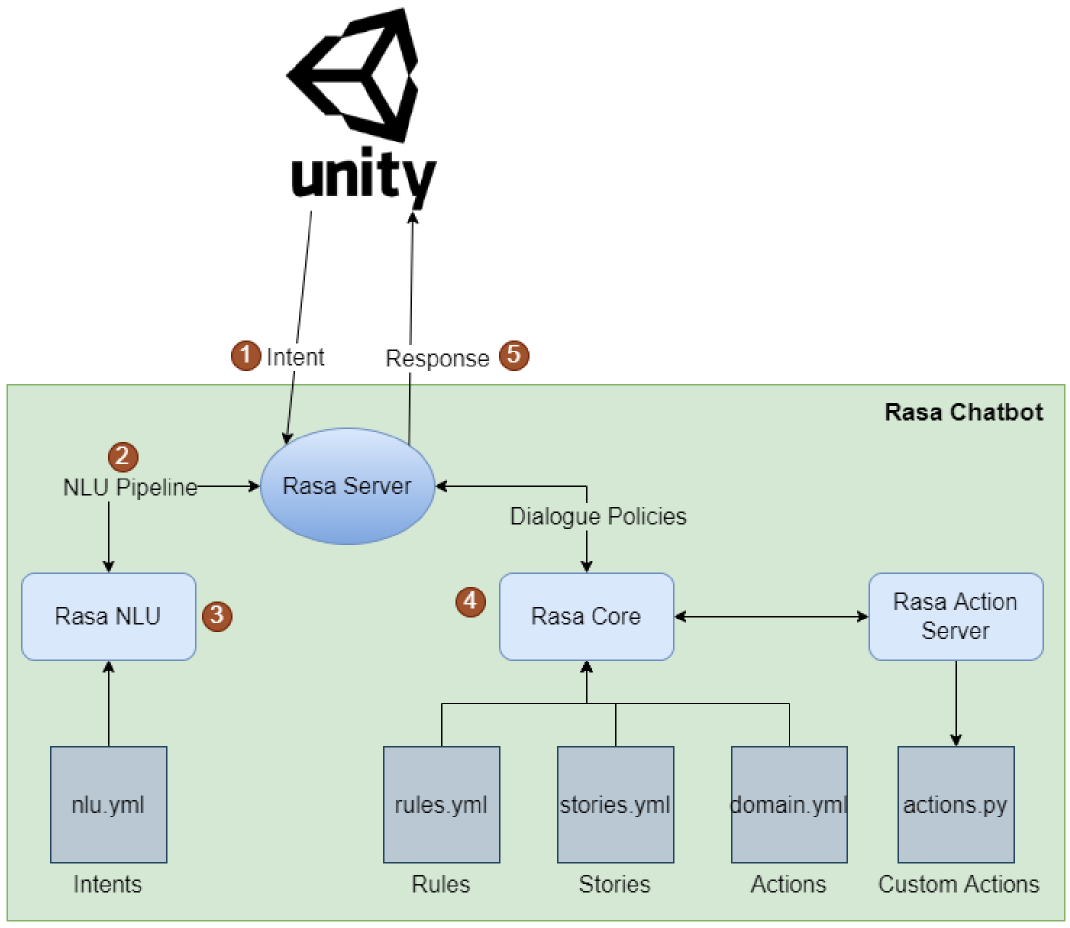


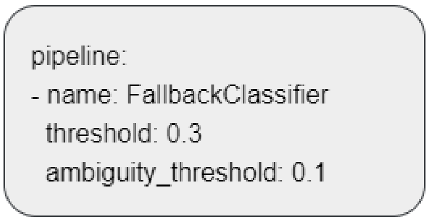
Figure 14: The Arch. diagram and functioning. Source: Own

Now, let's see the working principle of Rasa conversational agent:

1. First, the user's message (intent) is sent to the Rasa server (see top part of Figure 14).

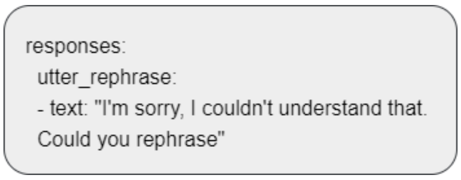
2. Then, the Rasa server redirects the message to the NLU pipeline.

3. The NLU pipeline analyzes the user's input, classifies the intent and extracts the important features of the message, that is, the associated entities.
   When the NLU pipeline try to identify the intent, it could failed if the percentage of understanding is too low. The % of understanding, also called confidence degree of the prediction or fallbacks that if it's less than 30%, a message will be displayed that the agent has not understood the user [31]. The percentage of understanding is defined in `config.yml` file and can be modified (see Figure 15). The message, which the chatbot should send when an input is classified with low confidence, is defined in the utter_default template of the `domain.yml` file (see Figure 16).



```
pipeline:
- name: FallbackClassifier
  threshold: 0.3
  ambiguity_threshold: 0.1
```

Figure 15: FallbackClassifier defined in `config.yml`. Source: Own



```
responses:
  utter_rephrase:
  - text: "I'm sorry, I couldn't understand that.
  Could you rephrase"
```

Figure 16: Low confidence response defined in `domain.yml`. Source: Own

4. On the other hand, the Rasa Core component takes the user's intent and analyzes what it should do after receiving it, based on the stories or rules that have been defined in its training. To be specific, the Rasa Core makes predictions from the training data that the chatbot designer provides, and it will generate percentage for each possible actions, then select the one with the highest match percentage. In this component is where the action is selected and executed. Note that usually the Rasa chatbot's action is an utterance but it can also be, for example, the storing of data in a database.

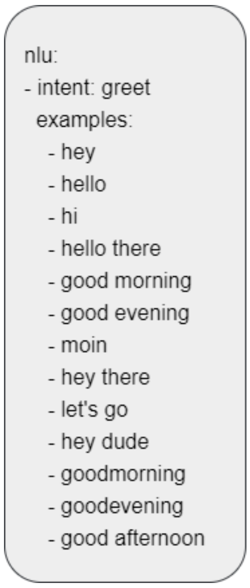
5. Finally, the assistant returns the response to the user through the same channel as the input's.

Now, the following sections detail relevant aspects of the described process, items, actions, domain, stories and rules.

7.2.1 NLU training data: Intents

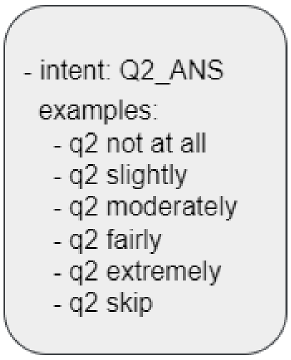
Through the NLU component, the agent must understand what user is saying, so from each message that the bot receives, the associated intent is extracted. An intent is nothing but the intention (goal) that the user has when interacting with our agent, and for each intention, a user could express it in other words, called synonyms, which should be assigned to that intent. To take an example, if the chatbot asks the user if he is minor or not, the synonyms to express the affirmation intent can be performed in many ways, like "Yes, I am", "Sure", "Indeed", "Of course, I am", among others.

In Rasa, intents are defined inside a yml (YAML) format file, called `nlu.yml`, in the data directory. YAML is a data serialization language and it's designed to be both human readable and computationally powerful (more information in section 8.1.5).This is where we defined all the intents and their synonyms. Below, we present two defined intents, and the items under the example section are the synonyms of the intent. In Figure 17, we can see the greet intent and the set of examples that represent how the user would greet the bot. These examples are provided for the bot training by the bot designer. In Figure 18, we can see an internal intent Q2_ANS. Note that each example of an internal intent is composed by the identifier of the question (q2) and the degree of agreement of the user to the UX question asked by the bot.



```
nlu:
- intent: greet
  examples:
    - hey
    - hello
    - hi
    - hello there
    - good morning
    - good evening
    - moin
    - hey there
    - let's go
    - hey dude
    - goodmorning
    - goodevening
    - good afternoon
```

Figure 17: Illustration of intent in `nlu.yml`: greet . Source: Own



```
- intent: Q2_ANS
  examples:
    - q2 not at all
    - q2 slightly
    - q2 moderately
    - q2 fairly
    - q2 extremely
    - q2 skip
```

Figure 18: Illustration of internal intent in `nlu.yml`: Q2_ANS. Source: Own

In this project, the Rasa agent expects those responses (user's intents) defined by the GEQ such as *not at all*, *fairly*, *moderately*, etc. The intents that we defined for the agent (see Figure 20) can be grouped into two blocks, those related to the internal commands to control the dialogue flow and those related to the bot-user interactions:

- Internal intents: Normally, the chatbot is built on back-and-forth conversations and the user takes control of the dialogue flow. In the survey agent case, we want the agent to take the initiative because it is the one who is asking the questions and, depending on the situation, we might want the chatbot to perform some specific actions. With that purpose, we defined several internal intents so that the conversational agent could control the conversation flow using Unity scripts. It is important to note that all the internal intents are created and named with upper-case letters. All of them remain invisible to the user. Next, we can see some of them below:

  1. Qx_ANS (x corresponds to the question number): It's a combination of internal command and user's answer: "qx "+ "user's input" (see Figure 19). The first part of the combination, "qx", is used for multiple purposes such as to identify in which column of the database (column's names are same as the internal commands) should the answer be stored. The second part, "user's

answer", is the pure input of the user, so this value is stored in the column of the database that is identified by the first part of the message, "qx". The result of this combination is used to identify which intent it belongs to and to perform the corresponding rule.
Figure 19 shows an example that helps understanding these intents. First, the user writes his answer, "fairly", to the question 2 asked by the agent. Then, the controller takes the user's input and concatenates it with a internal command, "q2". Afterwards, Unity puts the result of the combination in a JSON request and sends it to the Rasa server. The Rasa server analyzes the received intent and classified it with the intent "Q2_ANS". Therefore, the Rasa server execute the rule that matches with the intent "Q2_ANS".

2. RESTORE_: As designed, the agent appears at certain points in the game to ask the user some UX questions. The agent chooses certain moments (game-stages) so as not to disturb the player, for example, at the end of a challenge, end of a level, etc. This internal intent is sent by the controller in Unity when the agent needs to appear again in a specific game-stage.
3. PAUSE_: As deigned, the agent can appear at a certain moment to ask a question that the user could not to answer to the agent, that is, ignore the agent. In that case, the agent disappears, thus pausing the conversation. This internal intent is sent by the controller in Unity to pause the conversation when the user is inactive within a predefined period of time.

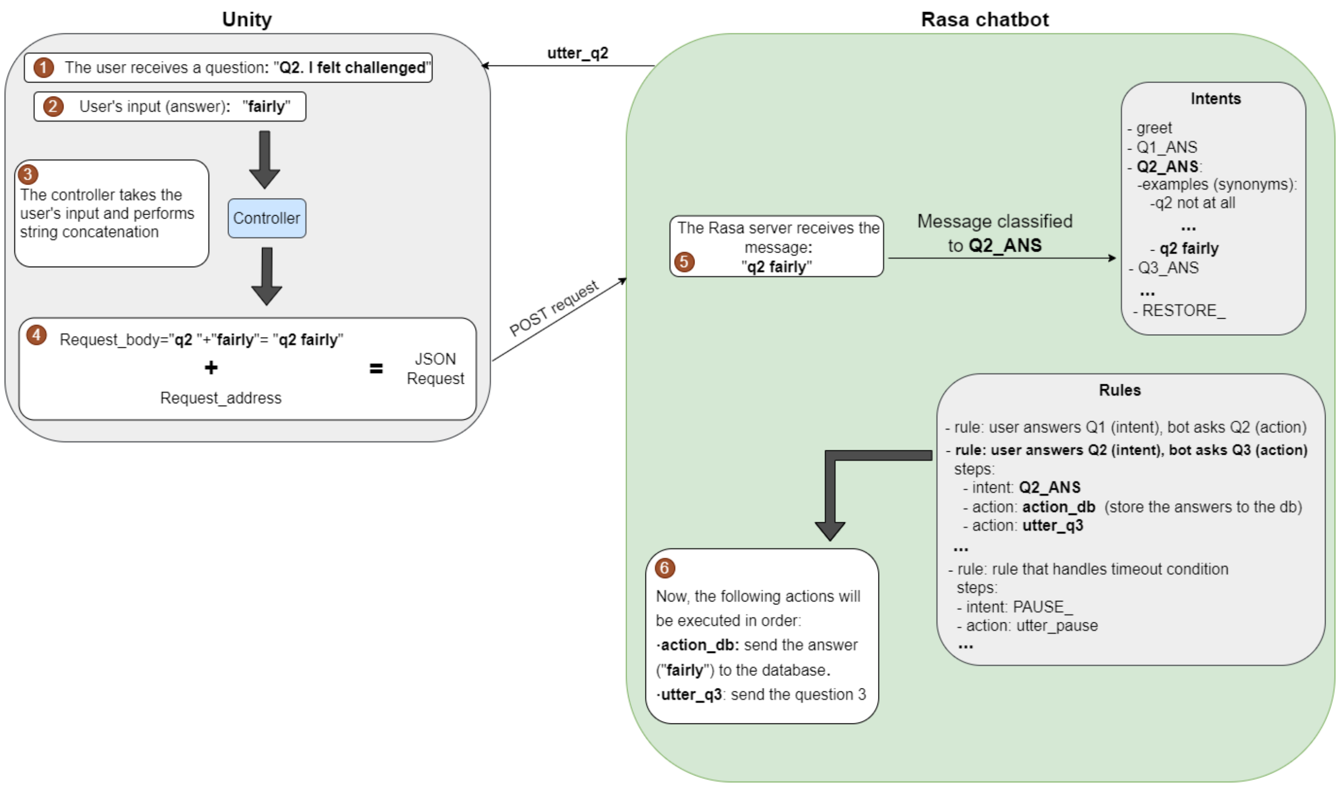
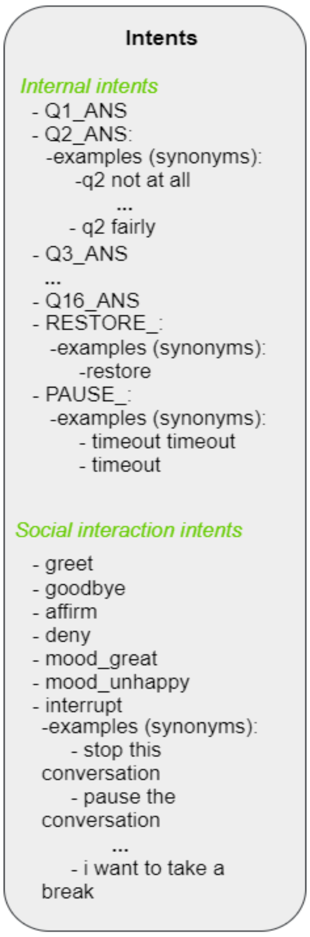

Unity
Rasa chatbot
1 The user receives a question: "Q2. I felt challenged"
2 User's input (answer): "fairly"
3 The controller takes the user's input and performs string concatenation
Controller
4 Request_body="q2 "+"fairly"= "q2 fairly"
+
Request_address
=
JSON Request
utter_q2
POST request
The Rasa server receives the message: "q2 fairly"
5
Message classified to Q2_ANS
Intents
- greet
- Q1_ANS
- Q2_ANS:
-examples (synonyms):
-q2 not at all
...
- q2 fairly
- Q3_ANS
...
- RESTORE_
Rules
- rule: user answers Q1 (intent), bot asks Q2 (action)
- rule: user answers Q2 (intent), bot asks Q3 (action)
steps:
- intent: Q2_ANS
- action: action_db (store the answers to the db)
- action: utter_q3
...
- rule: rule that handles timeout condition
steps:
- intent: PAUSE_
- action: utter_pause
...
6
Now, the following actions will be executed in order:
·action_db: send the answer ("fairly") to the database.
·utter_q3: send the question 3

Figure 19: Illustration that helps understanding the internal intents. Source: Own

• Social interaction intents: These intents are the basic knowledge that the agent has to perform an interaction with the user. Some of them are predefined (e.g. affirm, deny, mood_great, mood_unhappy) in Rasa and they are only used during the implementation phase to test out programs. Next, we show those intents we used in the final model.

1. greet: This intent is only used to activate the survey chatbot at the first game-stage by the controller in Unity. After the first game-stage, we use RESTORE_ intent to wake up the Rasa conversational agent.
2. interrupt: This intent is used when the user intends to take a break and unwilling to continue the conversation.



Intents

Internal intents
- Q1_ANS
- Q2_ANS:
  - examples (synonyms):
    - q2 not at all
    - ...
    - q2 fairly
- Q3_ANS
- ...
- Q16_ANS
- RESTORE_:
  - examples (synonyms):
    - restore
- PAUSE_:
  - examples (synonyms):
    - timeout timeout
    - timeout

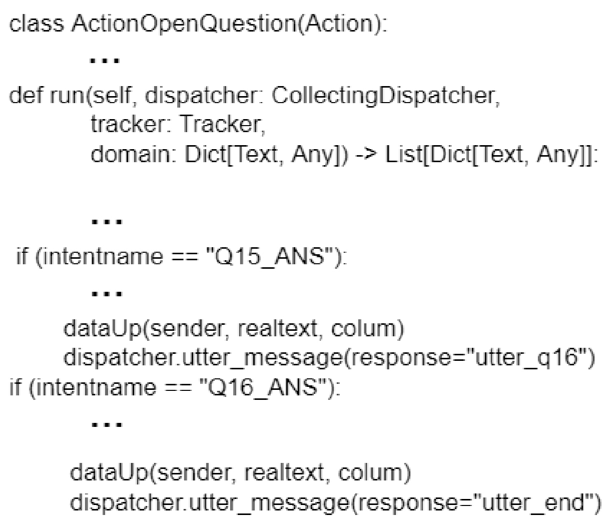
Social interaction intents
- greet
- goodbye
- affirm
- deny
- mood_great
- mood_unhappy
- interrupt
  - examples (synonyms):
    - stop this conversation
    - pause the conversation
    - ...
    - i want to take a break

Figure 20: Illustration that shows all the defined intents. Source: Own

7.2.2 NLU training data: Custom actions

Custom actions are functions written in Python in the `actions.py` that are used to make the agent capable of having custom behavior. This is a very useful file as it contains the action definitions that the bot will perform on specific intents, either to perform some calculations as for example time and date (see Figure 18), call APIs, among others.

In this project, several custom actions are created for storing valuable conversations into the database. The aim of the custom actions is to collect all the user's answers to the questions that are asked by the survey agent. To better understand the custom actions and show an example, in Figure 21 we can see an extract (part) of the custom action to store data in the database.



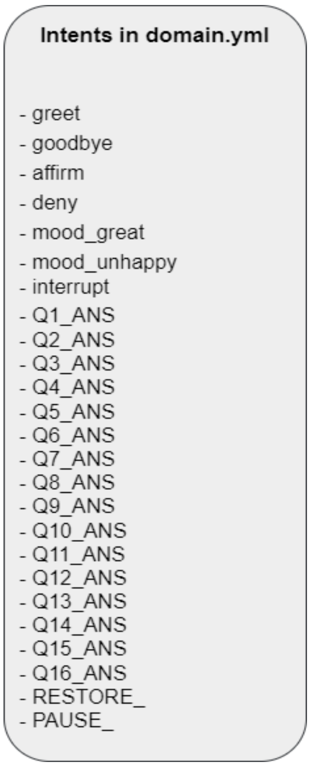
```
class ActionOpenQuestion(Action):
        ...
def run(self, dispatcher: CollectingDispatcher,
        tracker: Tracker,
        domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:

        ...
 if (intentname == "Q15_ANS"):
        ...
      dataUp(sender, realtext, colum)
      dispatcher.utter_message(response="utter_q16")
if (intentname == "Q16_ANS"):
        ...

       dataUp(sender, realtext, colum)
       dispatcher.utter_message(response="utter_end")
```

Figure 21: Illustration of an extract (part) of the custom action. Source: Own

7.2.3 NLU training data: Domain

Domain is the most important file in Rasa (see an extract of the full domain file in appendix 12). It's the universe in which our agent operates. As we mentioned in section 7.2, the NLU module processes what the user asked to the bot, and the Core processes what the bot will respond to. So, when the bot is trained for both NLU and Core side, the training takes reference from this file and creates models. Since, the `domain.yml` file specifies the intents, actions, slots, entities that the chatbot contains and also response templates for the answers of the bot.

At the beginning of the domain file, we defined the intents that must be matched to those from `nlu.yml` file, as showed in Figure 22.

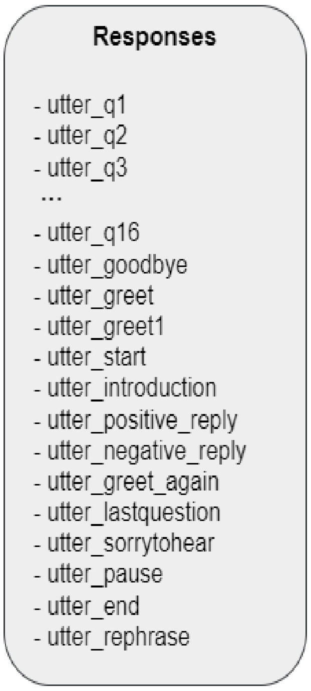


Intents in domain.yml

```
- greet
- goodbye
- affirm
- deny
- mood_great
- mood_unhappy
- interrupt
- Q1_ANS
- Q2_ANS
- Q3_ANS
- Q4_ANS
- Q5_ANS
- Q6_ANS
- Q7_ANS
- Q8_ANS
- Q9_ANS
- Q10_ANS
- Q11_ANS
- Q12_ANS
- Q13_ANS
- Q14_ANS
- Q15_ANS
- Q16_ANS
- RESTORE_
- PAUSE_
```

Figure 22: Illustration of declared intents in `domain.yml`. Source: Own

In `domain.yml`, we also declared the responses (see Figure 23) correspond to the user intents shown in the figure above.


Responses

- utter_q1
- utter_q2
- utter_q3

...
- utter_q16
- utter_goodbye
- utter_greet
- utter_greet1
- utter_start
- utter_introduction
- utter_positive_reply
- utter_negative_reply
- utter_greet_again
- utter_lastquestion
- utter_sorrytohear
- utter_pause
- utter_end
- utter_rephrase

Figure 23: Illustration of declared responses in `domain.yml`. Source: Own

To better understand the defined responses, we show them in the table below with a brief description:

Responses	**Description and text of the response, literally** (except for the questions of the GEQ)
From utter_q1 to utter_q16	From utter_q1 to utter_q14 are the questions from the Game Experience Questionnaire. The utter_q15 and utter_q16 are open questions.
utter_goodbye	When the user intends to say goodbye, our agent replies with this response. Text: *"Thank you for listening, see you next time"*.
utter_greet	This one is used to introduce our agent to the user. Text: *"Hey! I am Mayer, a chatbot. Can you answer few questions to help us improve our game? "*.

utter_greet1	Our agent sends this one along with utter_greet. They were together once, but after encountering some memory leak problem they were splitted up into two responses. Text: *"Feel free to take a break if you are tired by saying 'pause'. Of course, if you help us, later you will get some rewards."*.
utter_start	Our agent send this one right after utter_greet1. Normally, the user can only see it when the agent appears the first time. Text: *"Let's start with the first one:"*.
utter_introduction	Our agent sends this response before asking questions. This one tells to the user the evaluate scale. Text: *"Tell me how you feel about the following expression using: 'not at all', 'slightly', 'moderately', 'fairly' , 'extremely' (from 0 to 4) or 'skip' if don't know what to say."*.
utter_positive_reply	When the user answers positively, our agent replies with this response to express happiness. We want to add some random feature, so we defined three different responses so that the agent can reply using one of them randomly (this is how Rasa works by default). The three texts are: *"I are so happy to hear that!"*, *"That's wonderful!"* and *"Glad you feel that way."*.
utter_negative_reply	When the user answers negatively, our agent replies with this response to express consolation. Same as the utter_positive_reply, we defined two texts: *"I'm sorry that you feel that way..."* and *"Seems like I still have much to improve."*.
utter_greet_again	When our agent is triggered again, it sends this response instead of utter_greet. Text: *"Hey again! May I continue with the game experience questionnaire, it means a lot to us."*.
utter_lastquestion	The agent sends this response to appreciate the user when all the question from GEQ are answered. Text: *"I appreciate you answer all of them. I have last two open questions, and this time please answer with your own word!"*.
utter_sorrytohear	Our agent sends this response when the user intends to pause the conversation. Same as the utter_positive_reply, we defined three texts: *"Oh..I are so sorry to hear that. Thanks anyways, I'll see you later! Enjoy the game!"*, *"It's okay, I wish you the best luck! See you later!"* and *"Thanks a lot for helping us to keep improving the game! See you later!"*.

utter_pause	Our agent sends this response if the user is inactive for a period of time. Text: *"There was no activity for a while so I guess you might want to take a break, I'll catch up with you later, bye. "*.
utter_end	Our agent sends this one when all questions are answered. Text: *"Thank you so much for answering all the questions!"*.
utter_rephrase	Our agent sends this one when there is a fallback situation. Text: *"I'm sorry, I couldn't understand that. Could you rephrase?"*.

Finally, all the custom actions are also declared in this file as we can see in Figure 24.

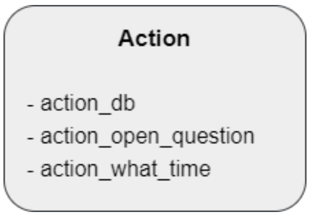
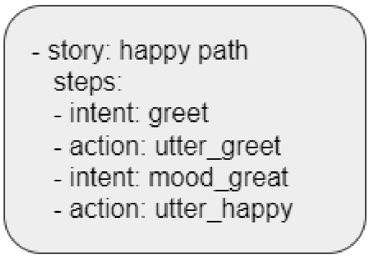


Figure 24: Illustration of the defined custom actions. Source: Own

7.2.4 NLU training data: Stories and rules

Rasa stories are part of the training data that are used to train Rasa's dialogue management models (Rasa Core). A story is a representation of a conversation between a user and the agent, converted to a specific format where user input is expressed as the corresponding intents while responses are expressed as corresponding action. In Rasa, it's recommended to use stories for multiple-turn interactions. Both intents and actions are defined with a "-" in front of the name and below the "steps" section as we can see in Figure 25. It is always the user who initiates the flow (see Figure 26).



```
- story: happy path
  steps:
  - intent: greet
  - action: utter_greet
  - intent: mood_great
  - action: utter_happy
```

Figure 25: Illustration of a simple story. Source: Own

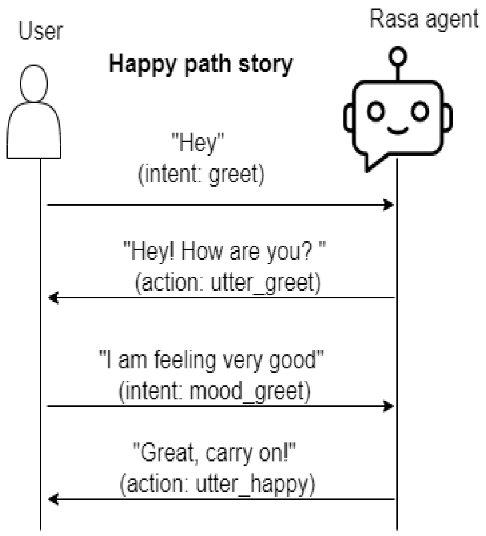


Figure 26: Conversation diagram of the happy path story example. Source: Own

On the other hand, rules are a way to describe short pieces of conversations that always go the same way. It's used to handle small chunks of conversation where it's always going to happen the same way, as we can see in Figure 27. Both intents and actions are defined with a "-" in front of the name and below the "steps" section, as we can see in Figure 28.

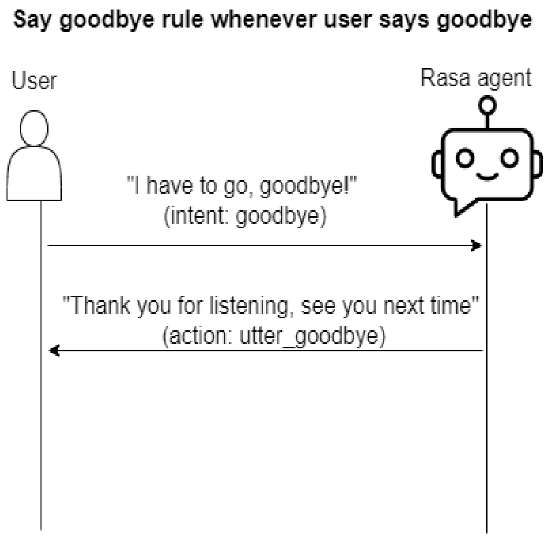
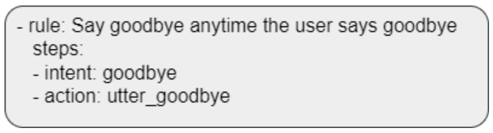

Say goodbye rule whenever user says goodbye
User
Rasa agent
"I have to go, goodbye!"
(intent: goodbye)
"Thank you for listening, see you next time"
(action: utter_goodbye)

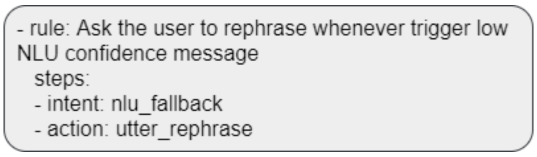
Figure 27: Conversation diagram of the say goodbye rule example. Source: Own



```
- rule: Say goodbye anytime the user says goodbye
  steps:
  - intent: goodbye
  - action: utter_goodbye
```

Figure 28: Illustration of the say goodbye rule. Source: Own

The NLU fallback response mentioned in section 7.2 is also used in rules (see Figure 29).



```
- rule: Ask the user to rephrase whenever trigger low
NLU confidence message
  steps:
  - intent: nlu_fallback
  - action: utter_rephrase
```

Figure 29: Illustration of the fallback rule. Source: Own

Obviously, rules and stories are quite similar concepts and in most occasions work the same way. The main difference between rules and stories is that rules are recommended for one-turn interaction and stories are recommended to use for multiple-turn interaction. In this project, rules are mainly used since the project consists in a pausable and resumable questionnaire agent. Pausable means users have the right to stop the conversation anytime they want, but it doesn't imply that the session will finish once for all, it's only a pause and will be continued at a later time at that same point in its duration. Resumable means the next time our chatbot appears, the conversation flow will start from the last unanswered question in order to guarantee that all the questions from the game experience questionnaire are completed. In RASA, stories and rules are defined inside the data folder (see Figure 30), where we find files in yml format called `stories.yml` and `rules.yml`, also the `nlu.yml` file described in section 7.2.1.

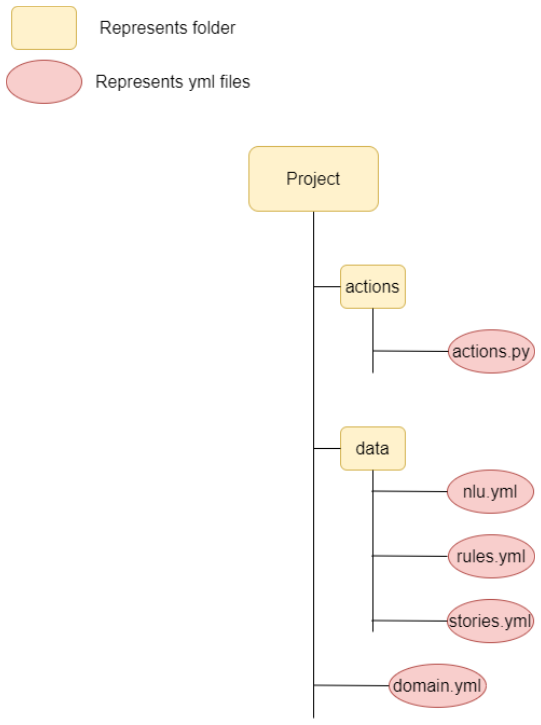


Figure 30: Illustration of the data folder. Source: Own

7.3 Agent designed in Unity

This section explains the software designed to give to our Rasa chatbot appearance and behaviour in Unity, starting with the main components of the Unity game. Then, we present two sequence diagrams associated with the three use cases defined in section 6.2. Afterwards, we show the different behaviours of the conversational agent in Unity. Finally, we show two dialogues between the user and the conversational agent.

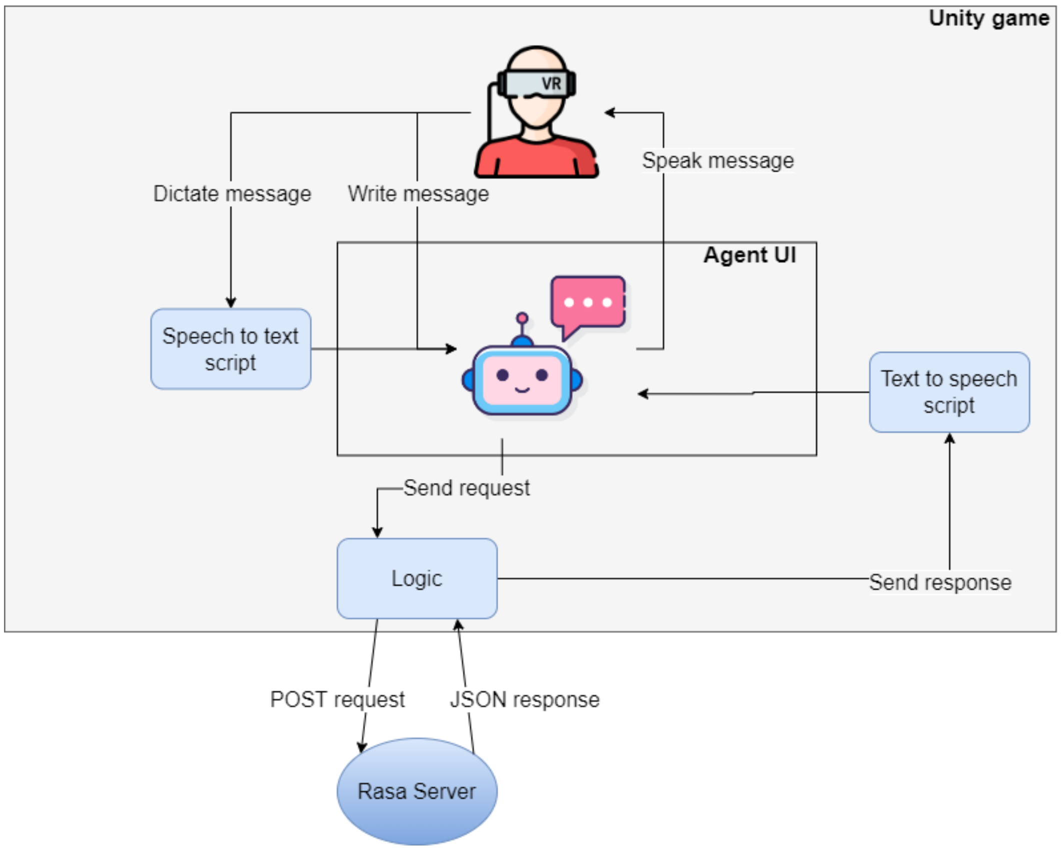


Figure 31: The Architecture Diagram of the Unity game. Source: Own

The diagram in Figure 31 shows a complete single-turn interaction with the agent, from the user sending the message by texting or dictating, to the user receiving the response. All components are explained below.

- **Logic**:
  This component represents a Unity script called `Controller.cs` that controls the logic of the whole project, and the REST API that communicates with the Rasa server. This script is designed to bring the logic of the agent into the game and is responsible for sending the user request to the Rasa server and showing the response that returns from the server to the user. On the one hand, the

agent observes events that happen on every frame using the Update() function that is provided by the MonoBehaviour class and triggers the bot if the player arrives to the correspondent game-stages by sending a message to the Rasa server. On the other hand, the script handles the flow of HTTP communication with the Rasa server, such as building POST requests in JSON format and using UnityWebRequest class (provided by UnityEngine.Networking) to retrieve the responses from the server.

- **Text to Speech**:
  This component represents a Unity script called `TxtToSpeech.cs` that uses the Microsoft Cognitive Services Speech SDK to make calls to the Microsoft Azure Speech service to get it to perform text-to-speech function without having to tediously code them manually. This script is responsible for converting the agent's responses into humanlike synthesized speech. Every time we receive a message from the Rasa server, this service will read it for the user. Since the goal is to communicate with the chatbot in a VR game, so having the text to speech feature accommodate a wide array of users while creating a more immersive and realistic experience.
- **Speech to Text**:
  This component represents a Unity script called `SpeechToTxt.cs`, which also uses the Microsoft Cognitive Services Speech SDK to make calls to the Microsoft Azure Speech service to get it to perform speech-to-text function without having to tediously code them manually. This script is responsible for transcribing user's audio into text. Since it's not accommodating the usage of virtual keyboards in the VR game, so having the speech to text component users can dictate their answers using microphone and have them converted into text.
- **Other components (not showed in the diagram)**:
  Since the chatbot needs to be triggered in some specific moments, we need a game component to simulate game stages. In order to perform that, we have created a text that represents the game score and a button to increase this score. Then, each time we click on the button, the score increases by 1. This increased value indicates that the simulated game has reached a new stage (a challenge has finished, a level has finished), and the it is time for the bot to appear. This part is controlled by a Unity script called `SimulateGameStep.cs`.

7.3.1 Sequence diagrams

This section shows several diagrams that represent the communication path between a user and the conversational agent. The sequence diagram shows the sequence of messages between Unity and Rasa during a given use case.
Figure 32 shows the path that the agent follows when initial a conversation that correspond with use case 1 (UC-1 Start the initial conversation). In the context of the game score is equal to 2, then our agent appears in the scene.
Figure 33 shows the path that the agent follows when a user answers the questions via text and voice (correspond to UC-2, Answer UX question in text mode, and UC-3, Answer UX question in voice mode). In the context of the user wants to write or dictate her answer and send it to the agent.

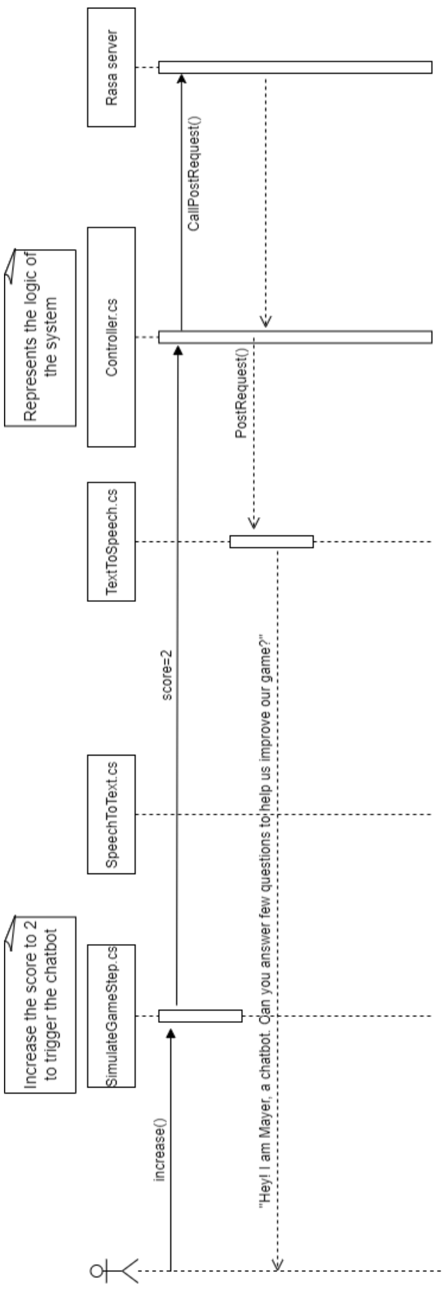

Rasa server
Controller.cs
Represents the logic of the system
TextToSpeech.cs
SpeechToText.cs
SimulateGameStep.cs
Increase the score to 2 to trigger the chatbot
increase()
score=2
CallPostRequest()
PostRequest()
"Hey! I am Mayer, a chatbot. Can you answer few questions to help us improve our game?"

Figure 32: Sequence diagram of UC-1: Start the initial conversation. Source: Own

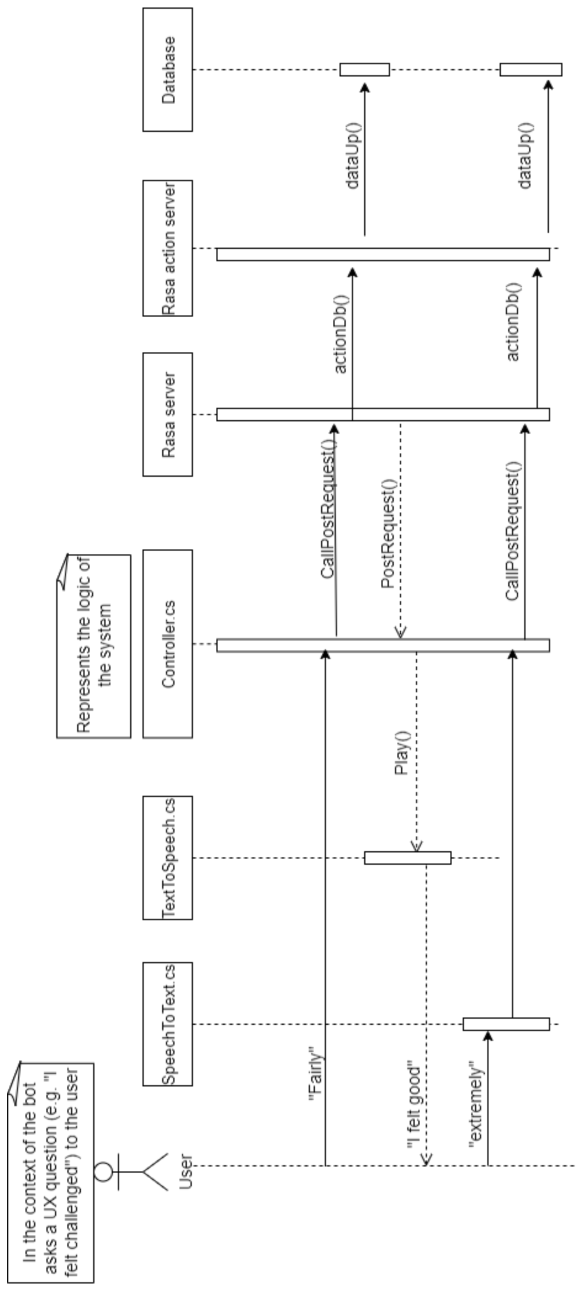


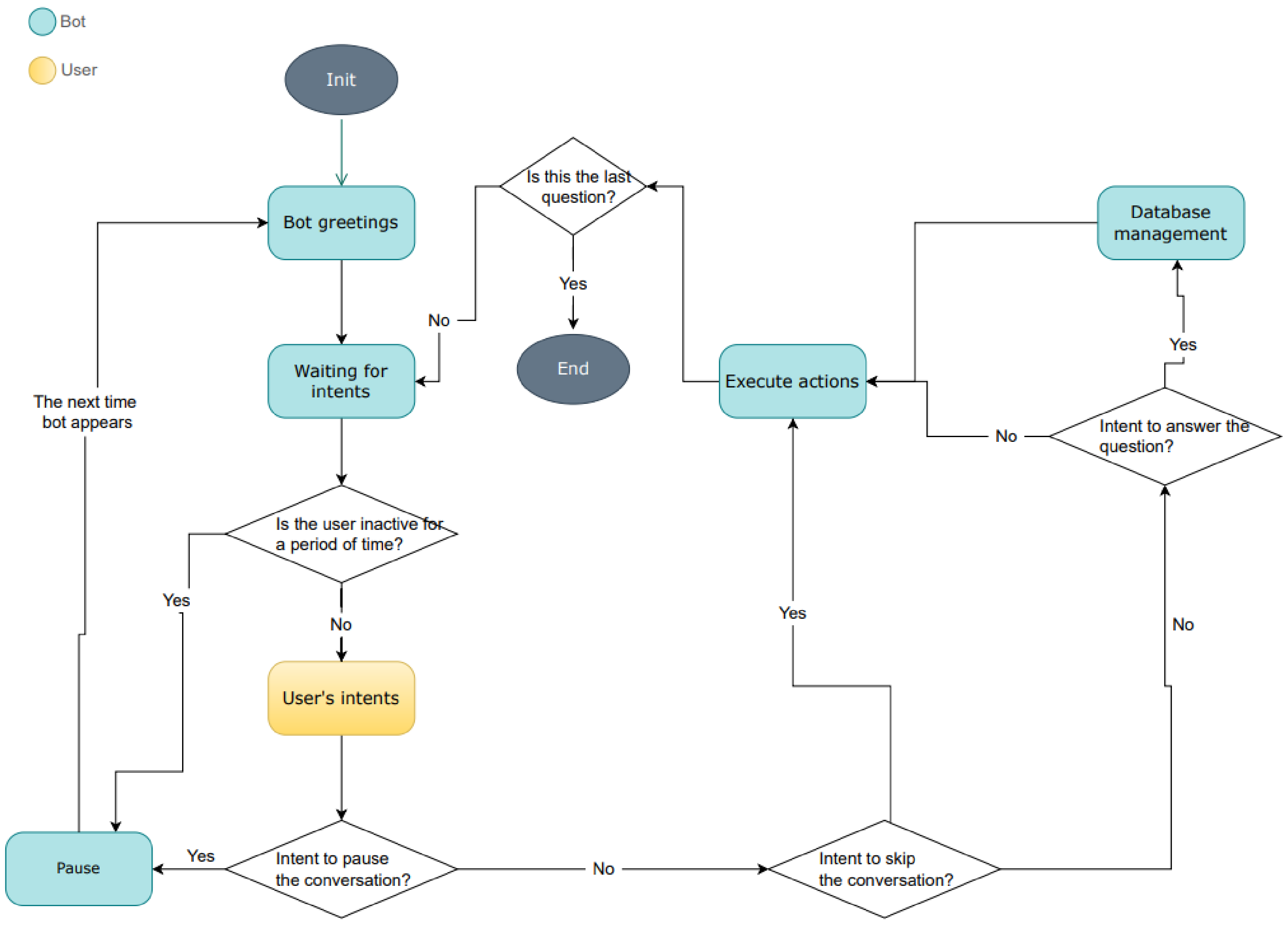
Figure 33: Sequence diagram of UC-3 and UC-4: Answer UX question in text mode and voice mode. Source: Own

7.3.2 Agent behavior

This section we present the different behaviours of the conversational agent in Unity. The diagram in Figure 34 depicts the steps through which the user-agent interaction passes. In blue, we find the stages that correspond to the conversational agent and in yellow, those of the user. First of all, the agent greets and asks the question, and moves to the waiting stage. Then, the user sends an intent to the agent. Our chatbot analyzes the intent and depending on several conditions the agent will move to the next stage that could be either the pause behaviour, where the conversation will be paused until the next time our agent shows up, or the database management behaviour, where a query will be executed to store the data (i.e answer of the user to the UX questionnaire) into the database. Figure 34 shows the logic of the survey bot, as we can see there are several behaviours:

1. The agent greetings behaviour. In this behaviour, the agent starts to talk with the user either at the beginning of the conversation or when it appears again.
2. Waiting behaviour. In this behaviour, the agent waits for any intent that is sent to it, in this case the intent could be the user's answer to a specific question or pause the conversation, among others.
3. User's intents behaviour. In this behaviour, the user could send any intent (e.g. pause, answer, skip) to the agent.
4. The pause behaviour. In this behaviour, the agent "hides" for a while (until the next stage) that any intent related to take a break or timeout condition could caused it.
5. The action execution behaviour. In this behaviour, the agent could execute any action depending on the intent. To be specific, it could execute an action that give a new question to the user or just an simple introduction to the questionnaire.
6. The database management behaviour. In this behaviour, the agent connects to the database and stores the valid answer into it.

.

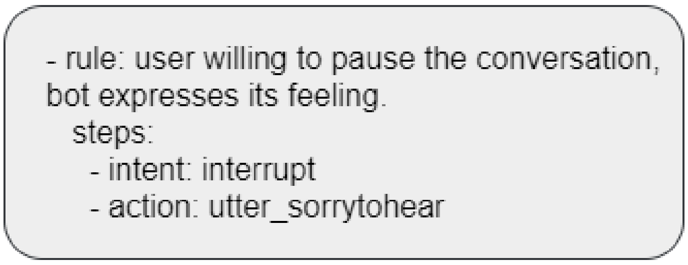


Bot
User
Init
Bot greetings
Waiting for intents
Is this the last question?
Yes
No
End
Execute actions
Database management
Yes
Intent to answer the question?
No
The next time bot appears
Is the user inactive for a period of time?
Yes
No
User's intents
Yes
No
Pause
Yes
Intent to pause the conversation?
No
Intent to skip the conversation?

Figure 34: Illustration that shows the agent's behaviours. Source: Own

7.3.3 Pause and resume behaviours of the Rasa agent

The goal is to make our chatbot pausable and resumable (more details in section 28). To make the Rasa agent pausable, we create a rule that makes the bot stop asking questions whenever the user has intention to take a break, as we can see in Figure 35. When the user is inactive for a period of time, the agent interprets there is a timeout situation, then it forces to pause the conversation and disappear.



```
- rule: user willing to pause the conversation,
bot expresses its feeling.
  steps:
    - intent: interrupt
    - action: utter_sorrytohear
```

Figure 35: The rule that handles interruption intent. Source: Own

On the other hand, we want to make the chatbot resumable so that when the player moves to the next game-stage, the survey agent could appear again and asks to the user the question that was left unanswered in the previous game-stage. In order to achieve this, we add a new feature to the `Controller.cs`, the file that controls the logic of the system, to make it stores the last sent user's intent in a variable. Therefore, the next time the Rasa agent needs to appear, we will not only send it the *RESTORE_* intent as mentioned in section 7.2.1, but also the last sent intent. Figure 36 shows an example that helps understanding how all this work. The example simulates a scenario where the user made a pause after answered the question 2 but left the question 3 unanswered. Now, the player moves to the next game-stage, where the Rasa agent needs to appears again. First, the controller sends the *RESTORE_* intent to the Rasa server and receives the response *utter_greet_again*. Then, the controller packages the last sent intent, which was "q2 fairly", to JSON format and sends the POST request to the Rasa server. The Rasa server analyzes the received request's body and classified it with the intent "Q2_ANS". Therefore, the Rasa server execute the rule that matches with the intent "Q2_ANS".

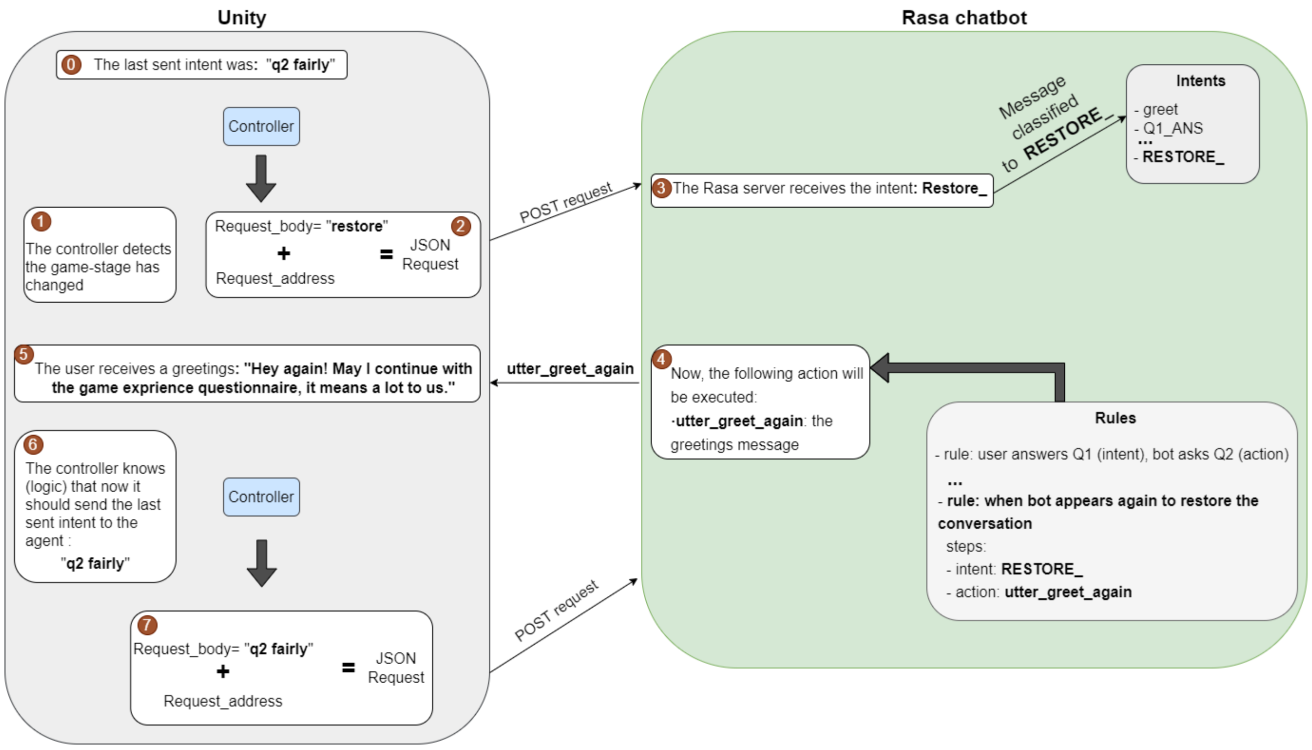
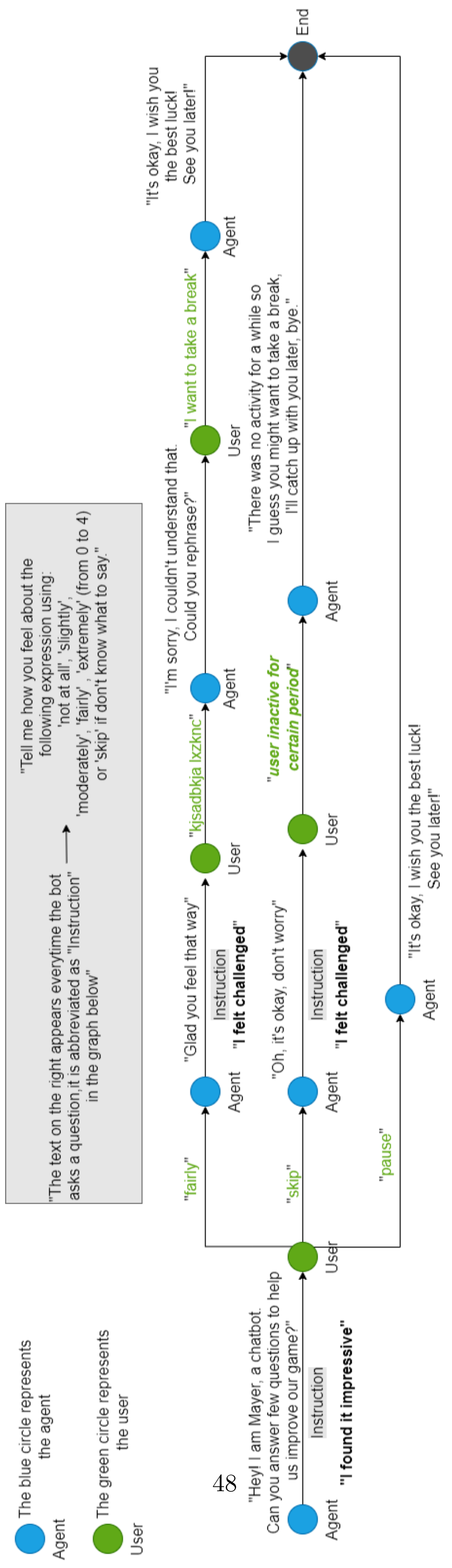

Unity
Rasa chatbot
0 The last sent intent was: "q2 fairly"
Controller
1 The controller detects the game-stage has changed
Request_body= "restore" + Request_address = JSON Request 2
POST request
3 The Rasa server receives the intent: Restore_
Message classified to RESTORE_
Intents
- greet
- Q1_ANS
...
- RESTORE_
5 The user receives a greetings: "Hey again! May I continue with the game exprience questionnaire, it means a lot to us."
utter_greet_again
4 Now, the following action will be executed: ·utter_greet_again: the greetings message
Rules
- rule: user answers Q1 (intent), bot asks Q2 (action)
...
- rule: when bot appears again to restore the conversation
steps:
- intent: RESTORE_
- action: utter_greet_again
6 The controller knows (logic) that now it should send the last sent intent to the agent : "q2 fairly"
Controller
7 Request_body= "q2 fairly" + Request_address = JSON Request
POST request

Figure 36: Illustration that helps understanding the internal intent RESTORE_. Source: Own

7.3.4 Dialogue overview

This section shows two dialogues between the user and the agent. The green circle represents the user and the blue one represents the agent. In Figure 37, we can see several situations that could happen during the survey such as when it's a normal conversation flow, when the user is inactive for a while, when the user says something that the agent cannot comprehend and when the user asks to the agent to take a break (in the dialogue, the user says ”pause”). Figure 37 shows a conversation between the agent and the user when answering the last question of the survey and the agent's appreciation to him.



Agent
The blue circle represents the agent
User
The green circle represents the user
"The text on the right appears everytime the bot asks a question,it is abbreviated as "Instruction" in the graph below"
"Tell me how you feel about the following expression using: 'not at all', 'slightly', 'moderately', 'fairly' , 'extremely' (from 0 to 4) or 'skip' if don't know what to say."
Agent
"Hey! I am Mayer, a chatbot. Can you answer few questions to help us improve our game?"
Instruction
"I found it impressive"
User
"fairly"
"skip"
"pause"
Agent
"Glad you feel that way"
Instruction
"I felt challenged"
User
"kjsadbkja lxzknc"
Agent
"I'm sorry, I couldn't understand that. Could you rephrase?"
User
"I want to take a break"
Agent
"It's okay, I wish you the best luck! See you later!"
Agent
"Oh, it's okay, don't worry"
Instruction
"I felt challenged"
User
"user inactive for certain period"
Agent
"There was no activity for a while so I guess you might want to take a break, I'll catch up with you later, bye."
Agent
"It's okay, I wish you the best luck! See you later!"
End

Figure 37: User-Agent dialogue: question-answer. Source: Own

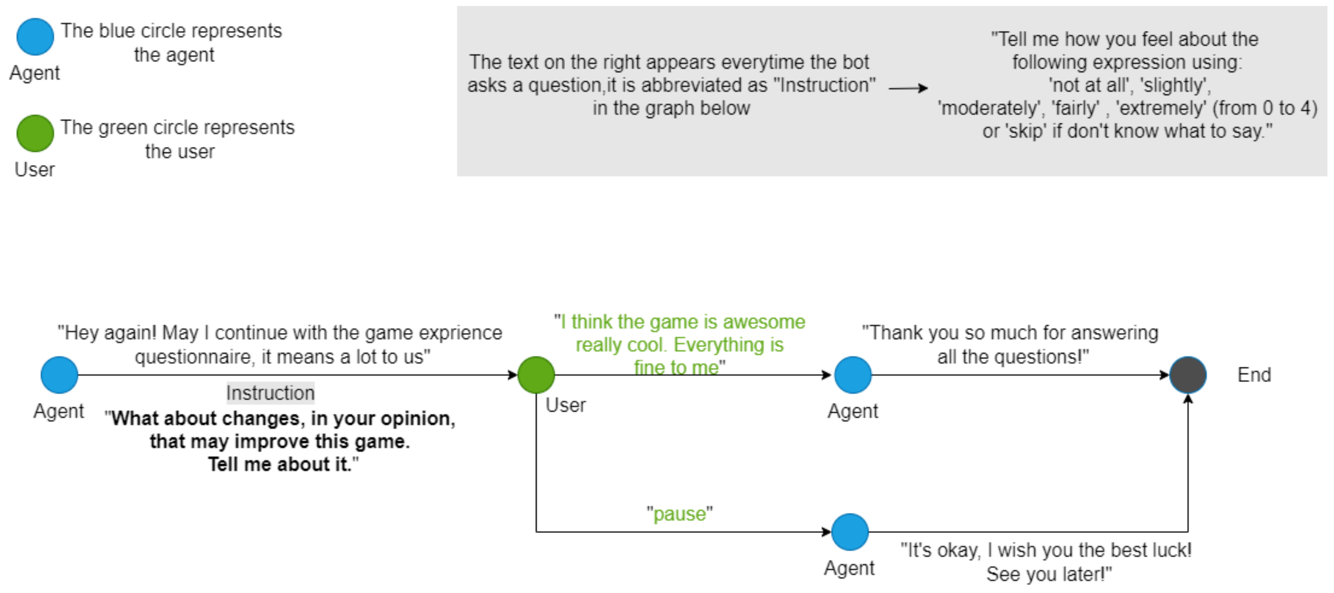

The blue circle represents the agent
Agent
The green circle represents the user
User
The text on the right appears everytime the bot asks a question,it is abbreviated as "Instruction" in the graph below
"Tell me how you feel about the following expression using: 'not at all', 'slightly', 'moderately', 'fairly' , 'extremely' (from 0 to 4) or 'skip' if don't know what to say."
"Hey again! May I continue with the game exprience questionnaire, it means a lot to us"
Agent
Instruction
"What about changes, in your opinion, that may improve this game. Tell me about it."
"I think the game is awesome really cool. Everything is fine to me"
User
"pause"
Agent
"Thank you so much for answering all the questions!"
Agent
"It's okay, I wish you the best luck! See you later!"
End

Figure 38: User-Agent dialogue: last question. Source: Own

7.3.5 Class diagram

This section presents a class diagram (see Figure 39) that shows the main classes that are used in this project. In blue, we can see the classes involved in text-to-speech and speech-to-text features. In green, the controller of the game. In yellow, the class that allows interaction with the Canvas's components in virtual reality environment. Finally, in orange, the class that controls the game-stage, in which the survey agent appears to ask the user a UX question.

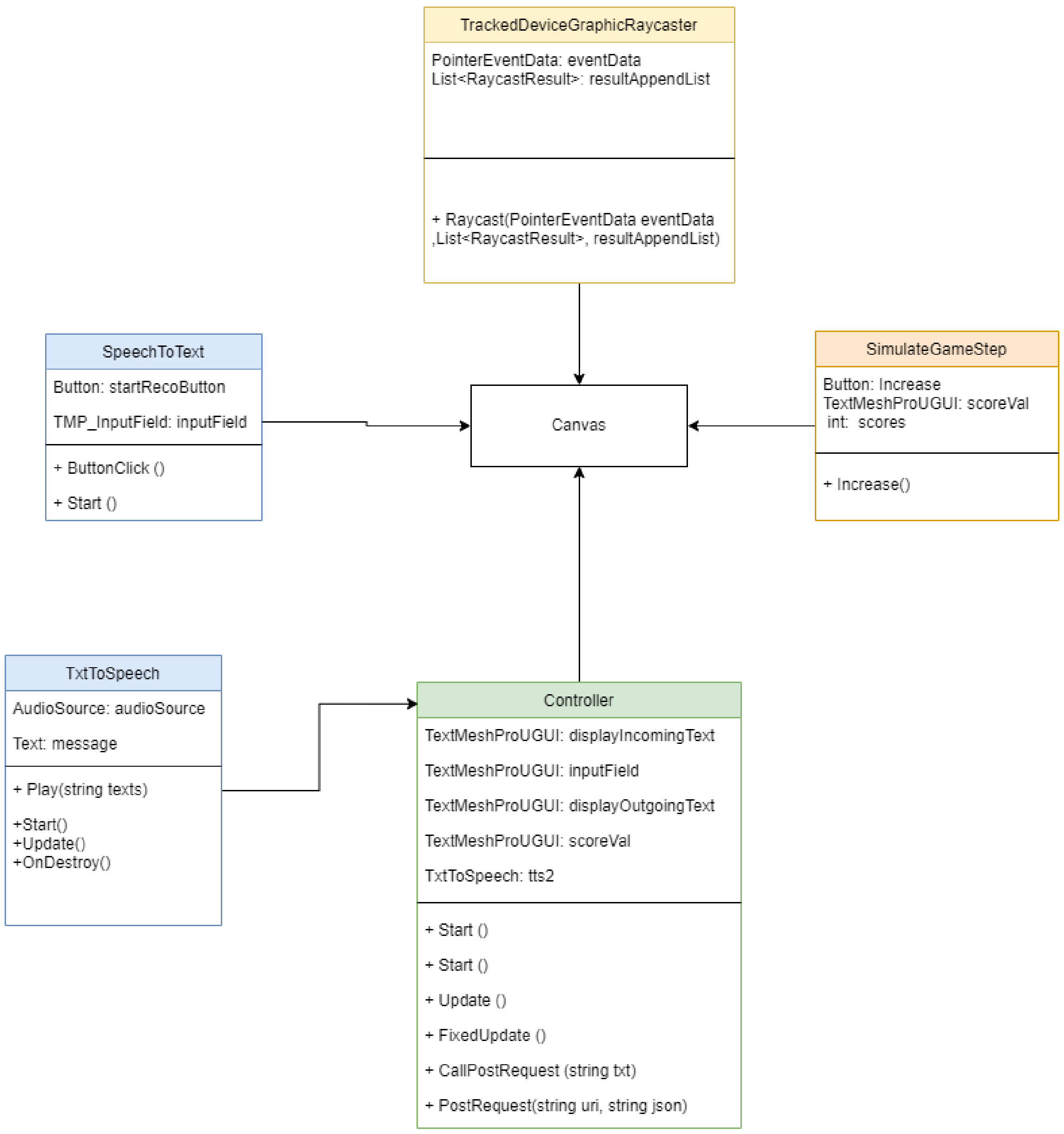

TrackedDeviceGraphicRaycaster
PointerEventData: eventData
List<RaycastResult>: resultAppendList
+ Raycast(PointerEventData eventData ,List<RaycastResult>, resultAppendList)
SpeechToText
Button: startRecoButton
TMP_InputField: inputField
+ ButtonClick ()
+ Start ()
Canvas
SimulateGameStep
Button: Increase
TextMeshProUGUI: scoreVal
int: scores
+ Increase()
TxtToSpeech
AudioSource: audioSource
Text: message
+ Play(string texts)
+Start()
+Update()
+OnDestroy()
Controller
TextMeshProUGUI: displayIncomingText
TextMeshProUGUI: inputField
TextMeshProUGUI: displayOutgoingText
TextMeshProUGUI: scoreVal
TxtToSpeech: tts2
+ Start ()
+ Start ()
+ Update ()
+ FixedUpdate ()
+ CallPostRequest (string txt)
+ PostRequest(string uri, string json)

Figure 39: Class diagram. Source: Own

8 Implementation

This section describes the implementation of the survey agent. First, it shows the most important aspects of the development phase, presenting the technologies used for the implementation of the project. Then, discuss some dismissed technologies that are no longer in use in this project. Finally, explains the integration of the agent into the Unity project.

8.1 Technologies used

This section list the different technologies, tools and programming languages that have been used in order to develop the chatbot and integrate it in the game.

8.1.1 Unity

Unity is a professional-level video game engine where allows users to build game on using its built-in functionalities and features. It can be used to build VR Games, mobile games, PC games and also console games [1]. The first thing a Unity game developer must do is decide which version of Unity should be used to create the project. The way that Unity does its versioning can be quite confusing. Normally, forwards compatibility is not supported in Unity, so that older versions of Unity can not open projects opened or created by newer versions. If, by accident, an old project is opened in the new version of Unity, its files could possibly been overwritten and impossible to go back unless having a backup.

The Restaurant Code mentioned in the previous section was also created using Unity. And that is why, Unity is still the main game engine and IDE for this project in order to integrate our chatbot.

8.1.2 C#

C-sharp (C#) is a simple, object-oriented, general-purpose language. According to Unity's official documentation [9], it's the only language that Unity supports locally for scripting. Scripting in Unity defines game objects behaviors and how they interact with each other. Each script creates its connection to the internal Unity processes by implementing a class that derives from the built-in class, called MonoBehaviour. The MonoBehaviour is the base class from which all Unity script derives [8]. And Mono is nothing more than a cross-platform implementation of .NET framework[3]. So this is one of the reason why C# is the primary language of Unity, and all the Unity libraries are built in C#.

8.1.3 Oculus Quest

Oculus Quest is the VR hardware that has been used to develop the Restaurant Code game and still our main tool to test the chatbot integration. The Oculus Quest are one of the Oculus virtual reality glasses models, being well known for their versatility and the good user experience they present. They have been classified using the term "ALL-IN-ONE-VR", since it has its own built-in screen processor, ram, storage and battery, among others. In addition, both the Oculus Quest 1 and the Oculus Quest 2 have another great advantage, they also do not need to use external sensors since they come equipped with all the

[3].NET framework is used for building and running applications on Windows [23]

necessary sensors in the glasses themselves. This is extremely useful as it makes its use much easier, as it does not require the prior installation of the sensors in the room, and allows the glasses to be used anywhere. In our case, the Oculus Quest 1 is used because it was the one available.

Figure 40: Oculus Quest logo. (Source: https://www.instavr.co/solutions/vr-outputs/oculus-quest)

8.1.4 Python

Python is a multi-purpose, high level programming language with a simple, clean and friendly syntax and dynamic semantic. Among the advantages of using Python compared to other languages are:

- The large number of libraries, which favors productivity when programming.
- The ease of use of data structures, Python also offers the option of high-level dynamic data typing that reduces code length.
- Python is open source that developed under an OSI[4]-approved open source license, which makes it free to use and distribute, even for commercial purposes.
- Python is extensible, Python can be extended to other languages.
- Python is easy to understand and code, because it's not such a verbose language, reading Python is a lot like reading English, which also means less coding is required than other languages.

In this project, Python is mostly used to write custom actions of Rasa conversational platform that allows API calls, database queries, etc.

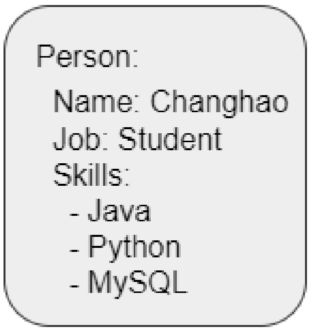
8.1.5 YAML

YAML, stands for YAML Ain't Markup Language, it's a lightweight data serialization language[5] inspired by XML. Its purpose is to format serialized data, and in Rasa it is used as a unified and extendable way

[4]The Open Source Initiative, or OSI, is an organization that tries to promote open source development without any profit.

[5]A serialization language allows applications written with different technologies, languages, which have different data structures can transfer data to each other using a common and standard format.

to manage NLU data, stories and rules [34]. It's frequently used in multiple applications because it is a very readable language for humans, more than JSON[6] and XML[7] format.



```
Person:
  Name: Changhao
  Job: Student
  Skills:
    - Java
    - Python
    - MySQL
```

Figure 41: An example of a YAML file. Source: Own

As we can see from Figure 41, YAML's syntax is very simple, intuitive and straightforward. In this example, we created an object called "Person". Under the object, we declared two key-value attributes: Job and name. And finally, we created a list called "Skills" with several values using dashes (""). A dash is used to place each item in the list at the same level.

8.1.6 Okteto: a deployment platform

Before get into Okteto, it's necessary to bring up Docker to help understanding the tool we used in this project. Docker is a software development platform that allows users to develop and deploy applications inside virtual containerized environments, containers [5]. In other words, with Docker apps run consistently regardless of the machine or operating system that it's running.

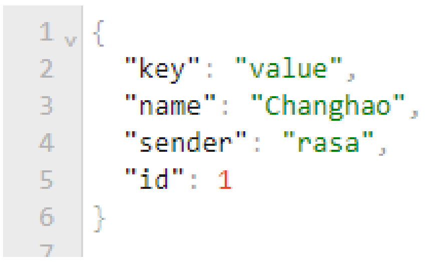
Okteto is a tool that allows the developer to launch development environments in Kubernetes, as well as being able to debug the application in it, and to deploy their applications directly on the cloud, among other things [27]. Kubernetes is an open-source system for managing containers across multiple hosts [10]. Basically, Okteto allow users to work inside any docker image container, which inherits the same volumes or any other configuration value of the original Kubernetes deployment. In this project, Okteto is used to build and deploy the *Rasa server* along with the *Rasa action server* on the Okteto cloud because of it's free and its ease of use (more information in section 8.3.3). We use Docker Compose (which is why we mentioned Docker) to define a Docker container image to avoid dealing with the complexities of Kubernetes manifests. We also use the Okteto cloud to access to Kubernetes namespace (Okteto provides them) where we can deploy the Docker containers for free. Namespaces are Kubernetes objects which partition a single Kubernetes cluster into multiple virtual clusters. A cluster is a collection of linked node machines on which the applications run.

[6]JSON is a file format that stores structured information and is primarily used to transfer data between servers and clients.

[7]XML, or Extensible Markup Language, is a markup language used to store and exchange structured data, whether it is documents, configurations, transactions or just data.

8.1.7 MongoDB

This project uses a NoSQL database because it's more flexible, and it allows adapting to the needs of this work in a much easier way than relational databases. NoSQL refers to non-relational or non-SQL database. NoSQL databases are document-oriented and allow users to store and retrieve data in formats other than tables [26].
MongoDB is an open source document-oriented NoSQL database [25]. Documents in MongoDB are similar to JSON (JavaScript Object Notation) objects, but it is based on another type called BSON or binary JSON that accommodates more data types. These documents are the replacement of what was previously known as rows in SQL systems. The structure of these documents is a key-value data model, that is:



```
{
  "key": "value",
  "name": "Changhao",
  "sender": "rasa",
  "id": 1
}
```

Figure 42: Example of a MongoDB document. Source: Own

Along with all of this, MongoDB assigns to each document a unique identification number (_id) within its collection. A collection is what we know as a set of documents. Compared to SQL systems, a collection would be the equivalent of tables. The great advantages that collections offer, among others, are, for example, their automatic creation if they do not exist, and their great flexibility, since the documents they contain do not have to have the same format. Another feature of these collections is that they can be grouped into sub-collections. In this project, MongoDB is chosen because of its flexibility and ease of deployment of the database along with the Rasa servers.

8.2 Dismissed technologies

During the implementation phase, we used various technologies to accomplish our goals, although some of them were discarded later because either we found better solutions or not compatible with some environment. This section details some technologies used during the implementation phase, but discarded in the end.

8.2.1 Text-to-speech

At the beginning of the implementation phase, we made an implementation of Rust text to speech to use in Unity. Rust is a system programming language designed for safety and performance [20]. Even though, learning how to program with Rust programming language was not part of the objective, but still we used the tts library [6] of Rust to implement a Rust program that contains the text to speech feature. In order to export the Rust program to Unity, we needed to compile the code to a DLL file to make use of it in the C# script. Since DLL, stands for Dynamic Link Library, is the file that consist of executable code that can be used by another module such as application or DLL [24]. The problem that we encountered was that the script didn't work in VR environment, as it only works in Windows operating system.

8.2.2 Speech-to-txt

At the beginning of the implementation phase, we used the Windows native tool and wrote a script in Unity called `DictationRecognizer.cs` to achieve the speech to text feature. To be able to use it, we must enable the dictation feature in the user's Speech privacy policy, otherwise the Dictation Recognizer will fail to initialize. The script worked incredibly well, even though some modification needed to be made to adapt to our logic. The problem occurred when we tried to interact with the conversational agent in the VR game because the script is currently functional only on Windows 10. The goal was to make this feature work in a virtual reality environment, so we needed to find another solution to solve the problem.

8.2.3 Dismissed database platforms

SQL stands for Structured Query Language, and it is the most common standardized language for accessing databases. And MySQL is one of the most popular open source SQL database management systems. It supports cross-platform compatibility with the most used main operating systems, among them the most outstanding in the market: Linux, Mac and Windows. It's a very flexible and easy-to-use system, while providing us with a very high level of security.

In this project, MySQL was used as the database management system for the Rasa chatbot at the beginning of the implementation phase. The problem occurred when deploying the database that continuously failed and after several trials we decided to use another database platform, MongoDB.

8.3 Unity 3D scene and Rasa Agent

The prototype developed in this project, previously to integrate the conversational agent in the Restaurant Code game, takes place in a Unity 3D scene where we simulate that we have reached the end of the game-stage, by pressing a button that indicates the stage pass, and the agent appears. In this case, a game-stage is the stage of the game in which the survey agent appears in the game to ask the user a UX question. This section details the implementation and the steps we took to make the agent working in virtual reality environment.

8.3.1 Architecture diagram of the game scene

This section presents an architecture diagram of the game scene, that shows different game objects that we created for the Rasa agents to work in Unity and for the project to run correctly in a virtual reality (VR) environment. In Figure 43, there are three types of gameObjects: the green type represents the gameObjects we created for the Rasa conversational agent to work properly in Unity, the orange type represents the gameObjects we used for the project to work in VR environment, as well as to interact with the elements in it. Finally, the pink type represents other necessary gameObject such as the lights and the cameras.

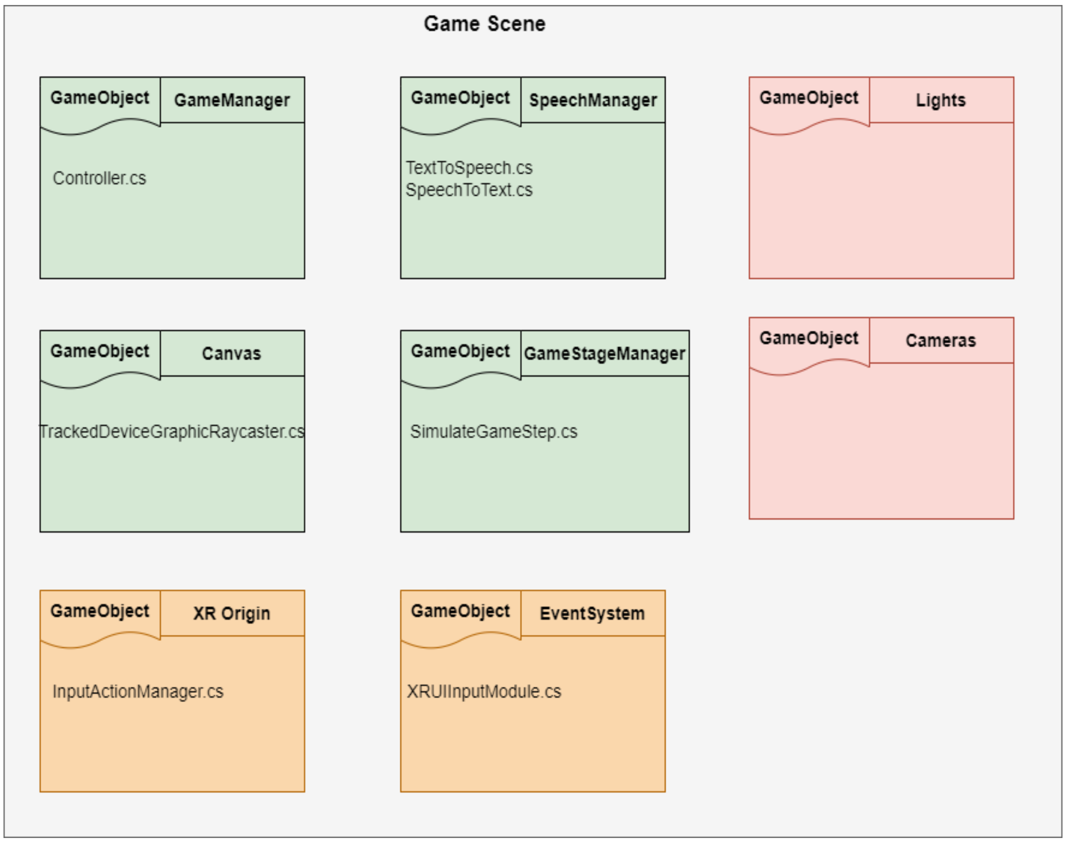
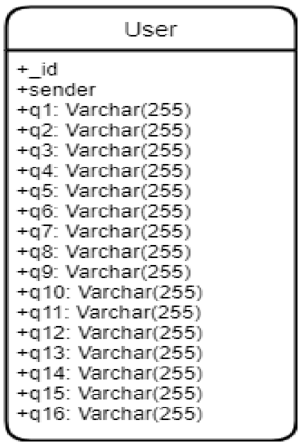


Figure 43: The architecture diagram of the game scene. Source: Own

8.3.2 Databases

Having the Rasa conversational agent work without a database is meaningless because, after all, we aim to evaluate user's opinions and use them either for statistical analysis or the development of new features. For that purpose, we need to store all the user's answers into a remote database and we use MongoDB to achieve this goal. In order to establish the connection between the Rasa server and the MongoDB we need to go through the Rasa action server. We have created several custom (see section 7.2.3) actions to establish the connection between the Rasa server and the Rasa action server, to perform MongoDB queries and to reply positively and negatively depending on whether the user's feedback is positive or negative (e.g. If the user's feedback is positive, the agent may replies with "That's wonderful!"). For this project, we create a database called Rasa and a collection called User.



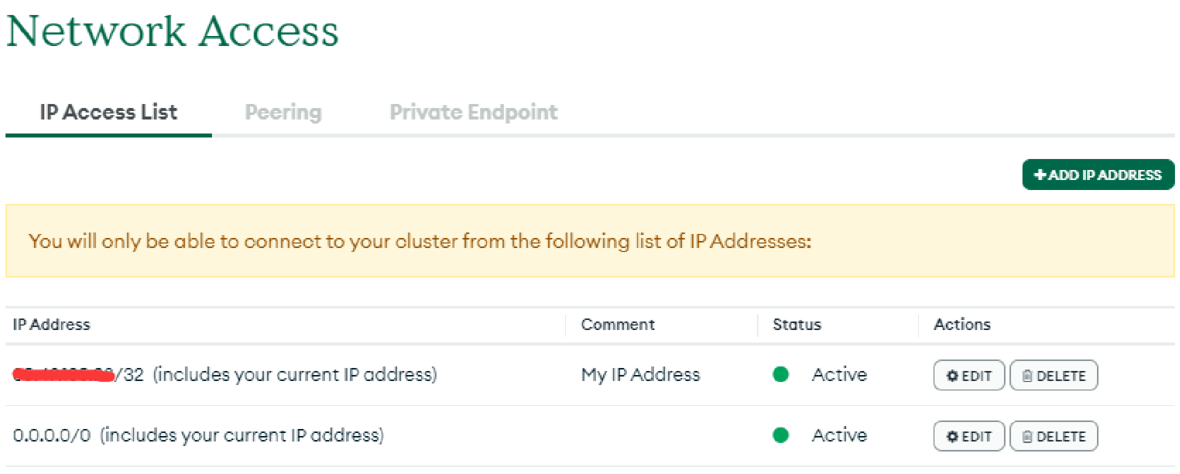
+_id
+sender
+q1: Varchar(255)
+q2: Varchar(255)
+q3: Varchar(255)
+q4: Varchar(255)
+q5: Varchar(255)
+q6: Varchar(255)
+q7: Varchar(255)
+q8: Varchar(255)
+q9: Varchar(255)
+q10: Varchar(255)
+q11: Varchar(255)
+q12: Varchar(255)
+q13: Varchar(255)
+q14: Varchar(255)
+q15: Varchar(255)
+q16: Varchar(255)

Figure 44: Collection of the user database. Source: Own

In Figure 44, we can see the collection of the database where we store the information of each user that is asked by the agent, and all their answers. Now, we explain the collection fields:

- _id: The primary key that generated automatically when adding a new user (creating a new document) in the collection.
- sender: This sender ID, also known as conversation ID, is used to identify the user who is having the conversation. This ID is set by the user when creating the POST request to send to the agent. In this project, we assign this ID to the user. The initial idea was to use the player's id of the Restaurant code game.
- q1-q16: From q1 field to q16 field we store all the answers given by the user in the survey session. These fields are designed in such a way to export the data easily to any kind of platforms or files such as Excel, CSV file, etc.

In MongoDB, users can only connect to the database from a trusted IP address. Therefore, we need to create a list of trusted IP addresses that can be used to connect to the database and access the data. At the beginning of the implementation phase, we add our IP address to the trusted list so that we are allowed to connect (send REST requests) to the database (more detailed in appendix) and test the Rasa agent. Nevertheless, we have to add another IP address "0.0.0.0/0" to the whitelist (see Figure 45) to bind to all IPv4 and IPv6 addresses in order to avoid further connection issues when other people want to explore and run the project.


Network Access

IP Access List | Peering | Private Endpoint

+ADD IP ADDRESS

You will only be able to connect to your cluster from the following list of IP Addresses:

IP Address	Comment	Status	Actions
[redacted]/32 (includes your current IP address)	My IP Address	● Active	EDIT DELETE
0.0.0.0/0 (includes your current IP address)		● Active	EDIT DELETE

Figure 45: IP access list of the database. Source: Own

8.3.3 Deployment

Having the conversational agent working locally is not enough because if we integrate it in the VR environment, there's no way to reach out the endpoints of the Rasa server and Rasa action server and therefore the communication between the agent and the user will not be initialized. To solve this problem, we need to deploy the Rasa server and the Rasa action server to the live environment so that we can communicate with the agent anywhere and anytime we want. This process is accomplished using a Kubernetes development platform called Okteto (more details in appendix 13.3).

8.3.4 Integration in Unity

Now we want to integrate our survey conversational agent in a Unity 3D project and make it works in VR environment. For that purpose, we created a simple game scene with several UI components inside a Canvas object in order to interact with the Rasa conversational agent and see the responses (see Figure 46). The Canvas is the area that all UI components should be inside[7]. In the top left corner of the screen, there are the score component that is used to indicate the current game-stage and a button to increase the score value by 1 to change the game-stage. In the middle of the screen, there is a chatbot character and a bubble text to show the responses from the conversational agent. At the bottom of the screen, there is a record button to transcribe audio into text. Next to the record button, there is an input component for users to write their message by texting. On the right side of the input component, there is a blue button to send the message in the input field.

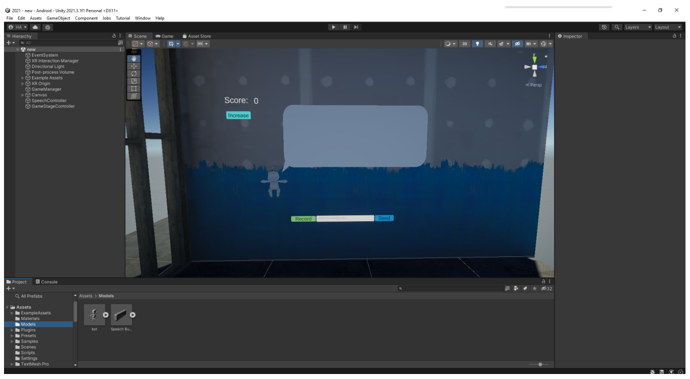
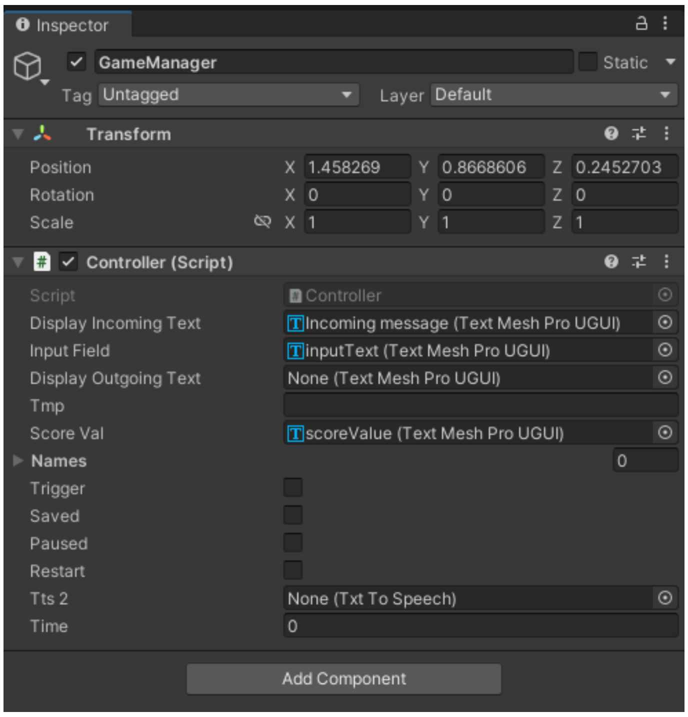


Figure 46: Illustration of the game scene. Source: Own

Behind the game scene, we have to attach all the scripts to the gameObjects showed in section 8.3.1. Therefore, we need to associate the `GameManager`, `SpeechManager`, and `GameStageController` to each required parameters to the correspondent UI components created in the Canvas. For the logic defined in `Controller.cs` in the `GameManager` (see Figure 47), we associate the incoming message to the bubble text in the center of the screen to show the responses from the agent, the input field to the input component at the bottom of the screen and the score value to the text component in the top-left corner of the screen to control the game-stage logic.


Inspector

GameManager Static

Tag Untagged Layer Default

Transform

Position	X 1.458269	Y 0.8668606	Z 0.2452703
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

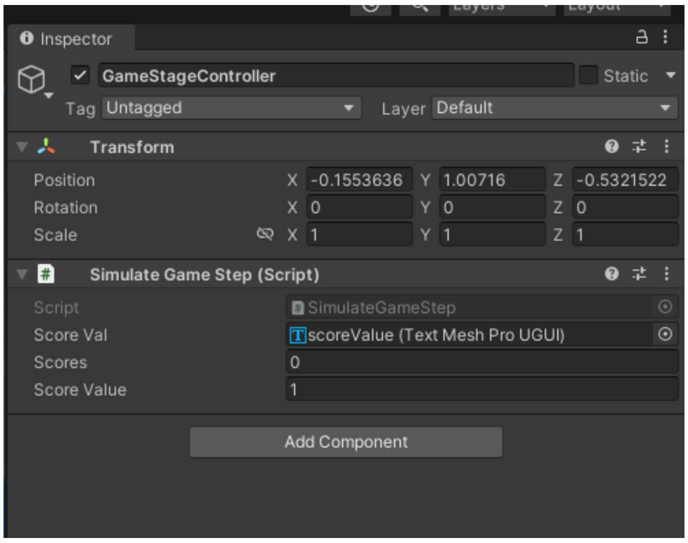
Controller (Script)

Script	Controller
Display Incoming Text	Incoming message (Text Mesh Pro UGUI)
Input Field	inputText (Text Mesh Pro UGUI)
Display Outgoing Text	None (Text Mesh Pro UGUI)
Tmp	
Score Val	scoreValue (Text Mesh Pro UGUI)
Names	0
Trigger	☐
Saved	☐
Paused	☐
Restart	☐
Tts 2	None (Txt To Speech)
Time	0

Add Component

Figure 47: Illustration of the `GameManager`. Source: Own

For the `SimulateGameStep.cs` defined in the `GameStageController` (see Figure 48), we associate the score value to the text component in the top-left corner of the screen to increase its value by clicking on the increase button.



Inspector

GameStageController Static

Tag Untagged Layer Default

Transform

Position X -0.1553636 Y 1.00716 Z -0.5321522

Rotation X 0 Y 0 Z 0

Scale X 1 Y 1 Z 1

Simulate Game Step (Script)

Script SimulateGameStep

Score Val scoreValue (Text Mesh Pro UGUI)

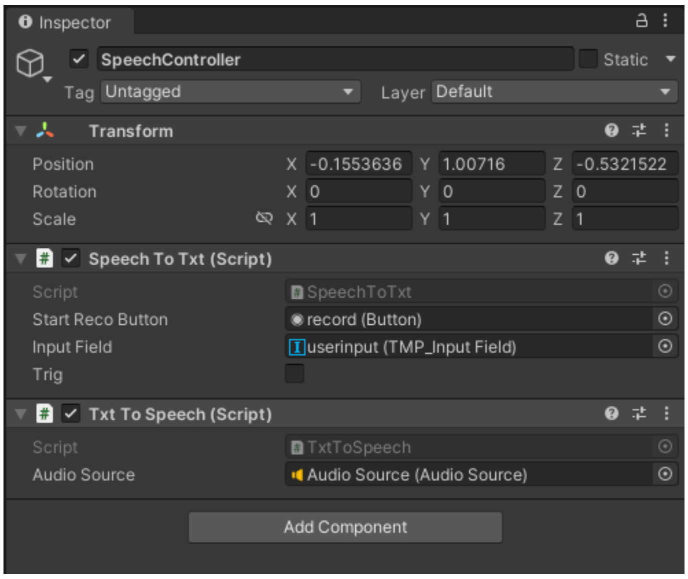
Scores 0

Score Value 1

Add Component

Figure 48: Illustration of the `GameStageController`. Source: Own

For the text to speech script defined in the `SpeechManager` (see Figure 50), we specify the audio source object we created in the scene to read the message automatically each time we receive a message from the chatbot. For the speech to text script, we associate the input field to input box at the bottom of the screen to transcribe user's audio into text by clicking on the record button.



Inspector

SpeechController Static

Tag Untagged Layer Default

Transform

Position X -0.1553636 Y 1.00716 Z -0.5321522

Rotation X 0 Y 0 Z 0

Scale X 1 Y 1 Z 1

Speech To Txt (Script)

Script SpeechToTxt

Start Reco Button record (Button)

Input Field userinput (TMP_Input Field)

Trig

Txt To Speech (Script)

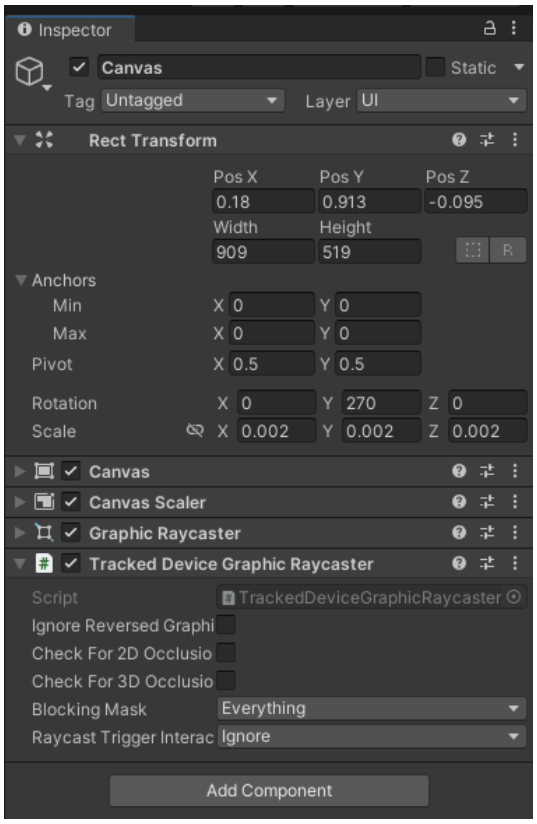
Script TxtToSpeech

Audio Source Audio Source (Audio Source)

Add Component

Figure 49: Illustration of the `SpeechManager`. Source: Own

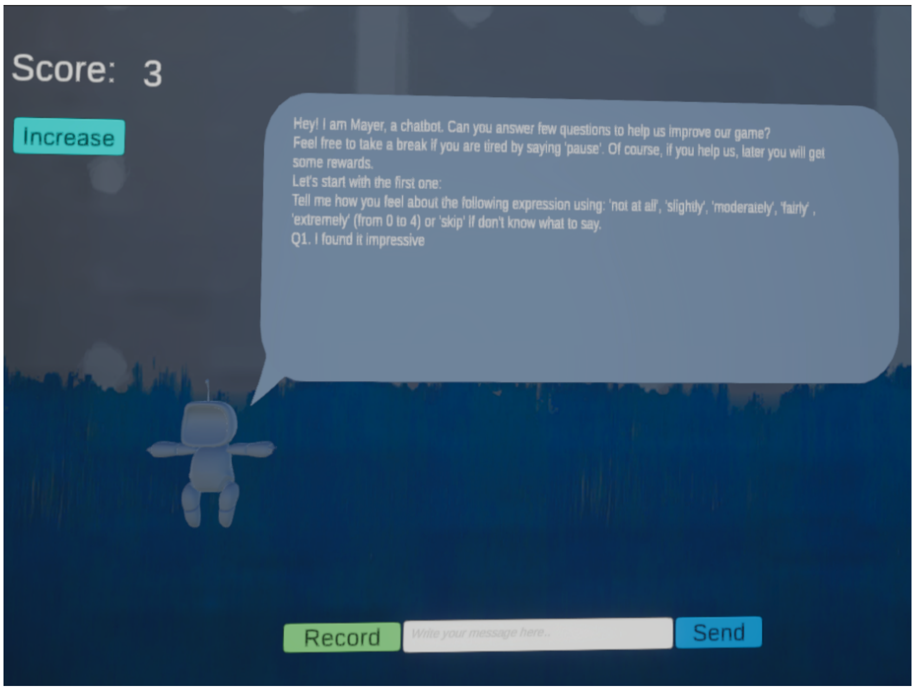
Finally, to be able to interact with all these UI components of the Canvas in virtual reality environment, we need to attach the `TrackedDeviceGraphicRayCaster.cs` script to the Canvas. The `TrackedDeviceGraphicRayCaster.` script will allow the raycast from the left hand and the right hand controllers to interact directly with UI elements of the Canvas in VR. In Figure 51 we can see the Rasa agent in VR environment [8].



```
Inspector
✓ Canvas                                Static
Tag Untagged              Layer UI
Rect Transform
                     Pos X      Pos Y      Pos Z
                     0.18       0.913      -0.095
                     Width      Height
                     909        519                R
Anchors
  Min                X 0        Y 0
  Max                X 0        Y 0
Pivot                X 0.5      Y 0.5
Rotation             X 0        Y 270      Z 0
Scale                X 0.002    Y 0.002    Z 0.002
✓ Canvas
✓ Canvas Scaler
✓ Graphic Raycaster
✓ Tracked Device Graphic Raycaster
Script                     TrackedDeviceGraphicRaycaster
Ignore Reversed Graphi
Check For 2D Occlusio
Check For 3D Occlusio
Blocking Mask              Everything
Raycast Trigger Interac    Ignore
Add Component
```

Figure 50: Illustration of the Canvas. Source: Own

[8]We upload a video sample of the game scene in VR, where we show some the main features of this project, on this link: `https://youtu.be/287qdp9DwRs`.



Hey! I am Mayer, a chatbot. Can you answer few questions to help us improve our game?
Feel free to take a break if you are tired by saying 'pause'. Of course, if you help us, later you will get some rewards.
Let's start with the first one:
Tell me how you feel about the following expression using: 'not at all', 'slightly', 'moderately', 'fairly' , 'extremely' (from 0 to 4) or 'skip' if don't know what to say.
Q1. I found it impressive

Figure 51: Illustration of a game scene in VR. Source: Own

9 Conclusions and future work

The aim of this project was to analyze, design and implement a conversational survey agent which would be used to evaluate UX in a virtual reality game. In addition, the aim was to be able to integrate the agent in the Restaurant Code project created by another student. Now, we review the established goals and their level of achievement:

- It has been investigated and analyzed different chatbots platforms and chose the Rasa open source as the main framework to implement the conversational agent.
- It has been successfully implemented a survey agent using Rasa open source platform.
- It has been designed a database using MongoDB and implemented several custom actions in Rasa to connect with it and perform queries.
- It has been created a simple game scene in Unity and made it works in virtual reality environment.
- It has been integrated the conversational agent in a virtual reality game scene.
- It has been written and modified several Unity scripts in C# from zero to add more features to the project.
- It has been studied how to program text to speech and speech to text features using C# and successfully made them work in the virtual reality game.
- It has become familiar with the Oculus Quest platform and learned how to export the Unity project to it.
- It has been deployed all the servers and the database to the live environment.
- It hasn't been accomplished how to integrate the agent into the Restaurant Code game, because of technical issues and time restrictions.

In summary, the analysis, design and implementation of the final project, with which it is considered that most of the objectives have been achieved. At the end of this project, we have created a chatbot using Rasa open source and successfully integrated it in a virtual reality game. However, we have not yet achieved the last goal, that is to integrate the agent into the Restaurant Code game.

Although we has been accomplished most of the goals established at the beginning of the project, still there is a final goal that hasn't been achieved yet and there are some possible extension and future work to be done:

- Integrate the agent into the Restaurant Code.
- Redesign the question pool to make the agent ask certain questions at certain moments in the game. Assign each GEQ's component to a specific game-stage to explore the maximum advantage of it. Also, after each game-stage we plan to ask various open questions related to the difficulty or playability of the challenge (game-stage).

- Train the agent so that those expressions similar to the responses defined by the GEQ could also be understood by the agent.
- Make the agent able to recognize emotions through dialogue, and to react differently depending on the emotion.
- Test with users to see how would them evaluate the UX in a virtual reality game with our agent integrated. Then, based on the feedback make a statistical analysis to study the accuracy of the Game Experience Questionnaire.

10 References

[1] Axon, S. Unity at 10: For better or worse game development has never been easier. https://arstechnica.com/gaming/2016/09/unity-at-10-for-better-or-worse-game-development-has-never-been-easier/, 09 2016. Accessed: 2022-05-03.

[2] Carroll, M. Dialogflow-unity-sdk. https://github.com/dialogflow/dialogflow-unity-client, 2019.

[3] Cloud, I. Ibm cloud docs. https://cloud.ibm.com/docs/watson-assistant. Online; accessed 09 May 2022.

[4] de Oliveira, R. P., de Oliveira, D. C. P., and Tavares, T. F. Measurement methods for phenomena associated with immersion , engagement , flow , and presence in digital games. pp. 127–135.

[5] Docker. Docker documentation. https://docs.docker.com/get-started/. Accessed: 2022-05-17.

[6] Docs.rs. tts - rust. https://docs.rs/tts/0.13.1/tts/. Accessed: 2022-05-01.

[7] Documentation, U. Canvas. https://docs.unity3d.com/Packages/com.unity.ugui@1.0/manual/UICanvas.html. Accessed: 2022-06-01.

[8] Documentation, U. Monobehaviour. https://docs.unity3d.com/ScriptReference/MonoBehaviour.html. Accessed: 2022-05-03.

[9] Documentation, U. Unity- manual: Creating and using scripts. https://docs.unity3d.com/Manual/CreatingAndUsingScripts.html. Accessed: 2022-05-03.

[10] Foundation, C. N. C. Kubernetes documentation. https://kubernetes.io/docs/home/. Accessed: 2022-05-27.

[11] Fuentes, A. Restaurant code: juego en realidad virtual para aprender a programar. http://hdl.handle.net/2445/182271. Accessed: 2021-06-20.

[12] Gómez, I. Cooking code: juego para aprender a programar con realidad virtual. http://hdl.handle.net/2445/172160. Accessed: 2022-04-14.

[13] Hat, R. What is a rest api. https://www.redhat.com/en/topics/api/what-is-a-rest-api. Accessed: 2022-05-11.

[14] Henrik, S.-F. The player engagement process - an exploration of continuation desire in digital games. In *DiGRA &#3911 - Proceedings of the 2011 DiGRA International Conference: Think Design Play* (January 2011), DiGRA/Utrecht School of the Arts.

[15] Huang, Y., Jasin, S., and Manchanda, P. "level up": Leveraging skill and engagement to maximize player game-play in online video games. *Information Systems Research 30*, 3 (2019), 927–947.

[16] I. Rodríguez, A. P. Open the microphone, please! conversational ux evaluation in virtual reality. 1–4.

[17] IJsselsteijn, W., de Kort, Y., and Poels, K. *The Game Experience Questionnaire*. Technische Universiteit Eindhoven, 2013.

[18] Kareem Yusuf, P. How watson assistant helps you treat customers like vips. `https://www.ibm.com/blogs/think/2018/03/watson-assistant/`, 2018. Online; accessed 09 May 2022.

[19] Keebler, J. R., Shelstad, W. J., Smith, D. C., Chaparro, B. S., and Phan, M. H. Validation of the guess-18: A short version of the game user experience satisfaction scale (guess). *J. Usability Studies 16*, 1 (nov 2020), 49–62.

[20] Klabnik, S., and Carol Nichols, w. c. f. t. R. C. Introduction. `https://doc.rust-lang.org/book/ch00-00-introduction.html`. Accessed: 2022-05-21.

[21] Lee, H. *Voice User Interface Projects: Build voice-enabled applications using Dialogflow for Google Home and Alexa Skills Kit for Amazon Echo*. Packt Publishing Ltd, 2018.

[22] Likert, R. *A technique for the measurement of attitudes*. Archives of psychology. New York: The Science Press, 1931.

[23] Microsoft. .net and .net framework. `https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet-framework`. Accessed: 2022-05-03.

[24] Microsoft. What is a dll. `https://docs.microsoft.com/en-us/troubleshoot/windows-client/deployment/dynamic-link-library`. Accessed: 2022-05-01.

[25] MongoDB. Mongodb documentation. `https://www.mongodb.com/docs/`. Accessed: 2022-05-26.

[26] MongoDB. What is nosql? `https://www.mongodb.com/en/nosql-explained`. Accessed: 2022-05-26.

[27] Okteto. Okteto documentation. `https://www.okteto.com/docs/welcome/overview/`. Accessed: 2022-05-26.

[28] Phan, M., Keebler, J., and Chaparro, B. The development and validation of the game user experience satisfaction scale (guess). *Human Factors: The Journal of the Human Factors and Ergonomics Society 58* (09 2016), 1217–1247.

[29] Shelstad, W., Chaparro, B., and Keebler, J. Assessing the user experience of video games: Relationships between three scales. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting 63* (11 2019), 1488–1492.

[30] Taherdoost, H. Validity and reliability of the research instrument; how to test the validation of a questionnaire/survey in a research. *International Journal of Academic Research in Management 5* (01 2016), 28–36.

[31] Technologies, R. Fallback and human handoff. `https://rasa.com/docs/rasa/fallback-handoff/`. Accessed: 2022-05-15.

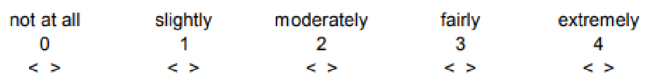
[32] Technologies, R. Introduction to rasa action servers. `https://rasa.com/docs/action-server/`. Accessed: 2022-05-11.

[33] Technologies, R. Rasa playground doc. `https://www.rasa.com/docs/rasa/playground`. Accessed: 2022-04-11.

[34] Technologies, R. Training data format. `https://rasa.com/docs/rasa/training-data-format`. Accessed: 2022-04-17.

11 Appendix I - GEQ In-game version

- The scale used in GEQ (see Figure 52).



not at all	slightly	moderately	fairly	extremely
0	1	2	3	4
< >	< >	< >	< >	< >

Figure 52: The measurement scale in GEQ. Source: [17]

- The items used in the in-game version.
  1. I was interested in the game's story.
  2. I felt successful.
  3. I felt bored.
  4. I found it impressive-
  5. I forgot everything around me.
  6. I felt frustrated.
  7. I found it tiresome.
  8. I felt irritable.
  9. I felt skillful.
  10. I felt completely absorbed.
  11. I felt content.
  12. I felt challenged.
  13. I had to put a lot of effort into it.
  14. I felt good.

12 Appendix II - Domain.yml

In this appendix, we show an extract of the complete `domain.yml` file (see Figure 53) that contains all the declared intents, responses and action.

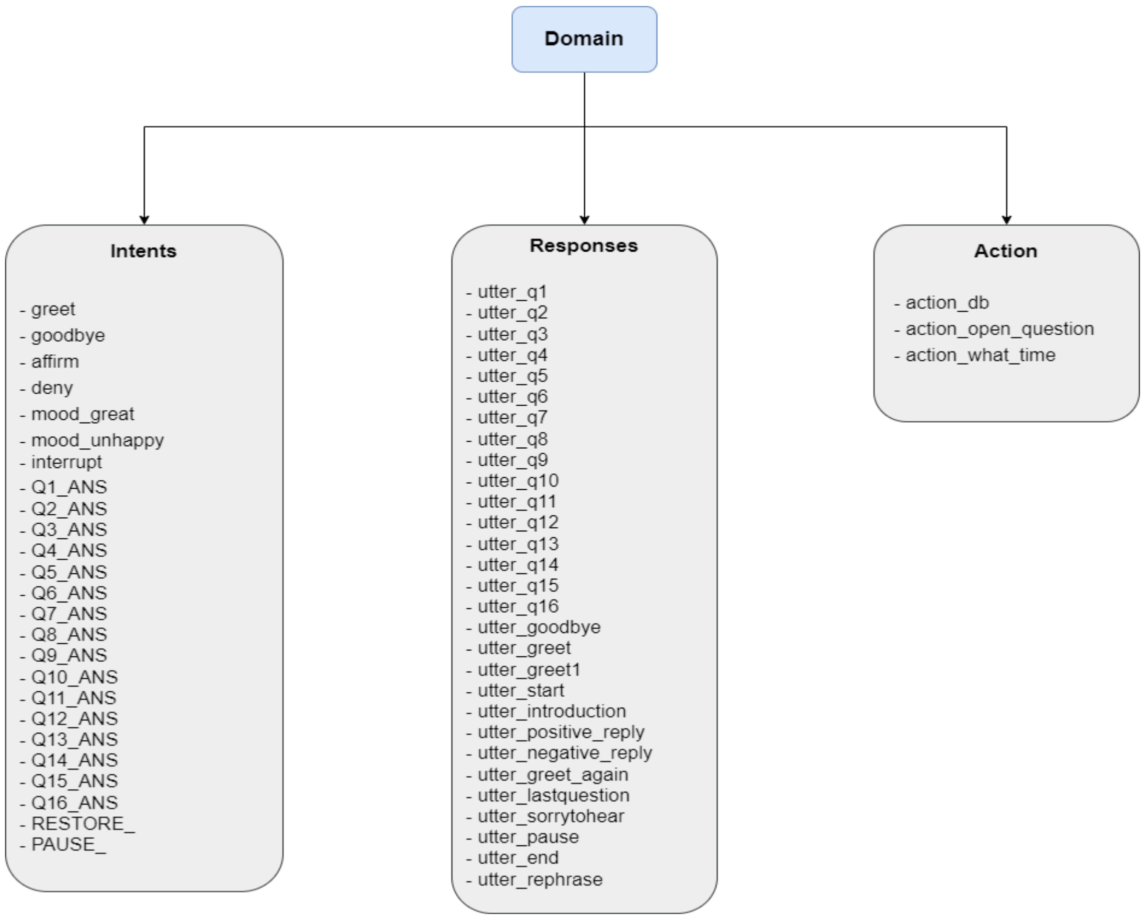


Figure 53: Illustration of the complete `domain.yml`. Source: Own

13 Appendix III - Technical guide

This appendix cover everything needed to be know for a developer who want to explore this project or extend it.

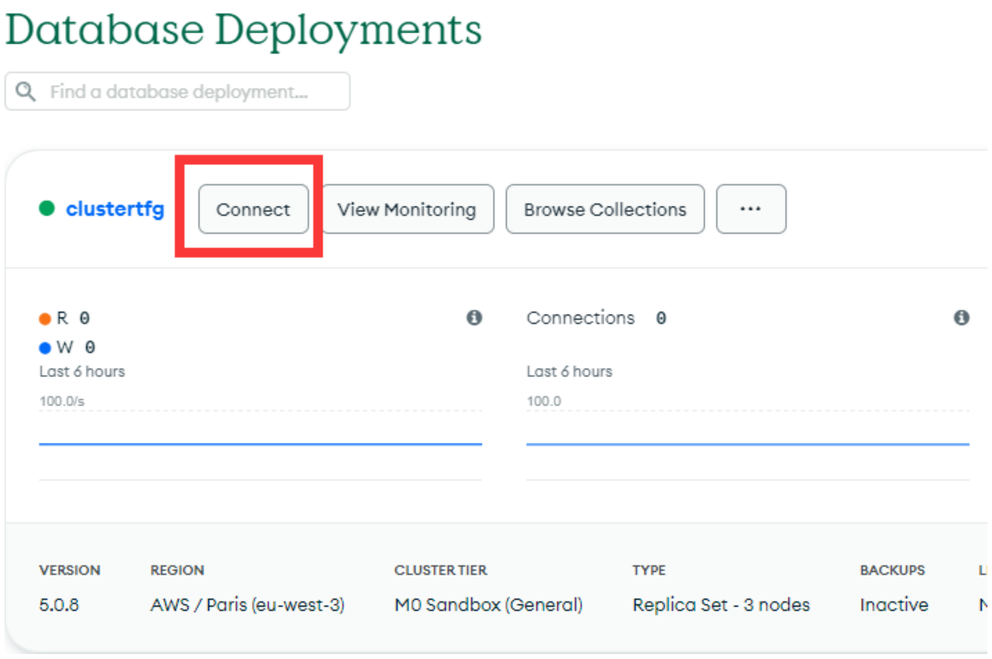
13.1 Rasa

To be able to run the Rasa agent, it's necessary to have the Python 3.7 or 3.8 installed. Then, follow this guide to install Rasa Open Source step-by-step: `https://rasa.com/docs/rasa/installation/`. Once everything is installed, train the model using command *rasa train*. When finished the training part, there are two ways to interact with the agent:

1. Use *rasa shell* command to load the trained model and talk to the agent on the command line.
2. Use *rasa run* command to start a server with the trained model. In order to interact with the agent it is needed to send a POST request to the endpoint specified in `endpoints.yml` file (see Figure 5 in section 4). If there are any custom actions, it's also needed to run the Rasa action server using this command *rasa run actions*.

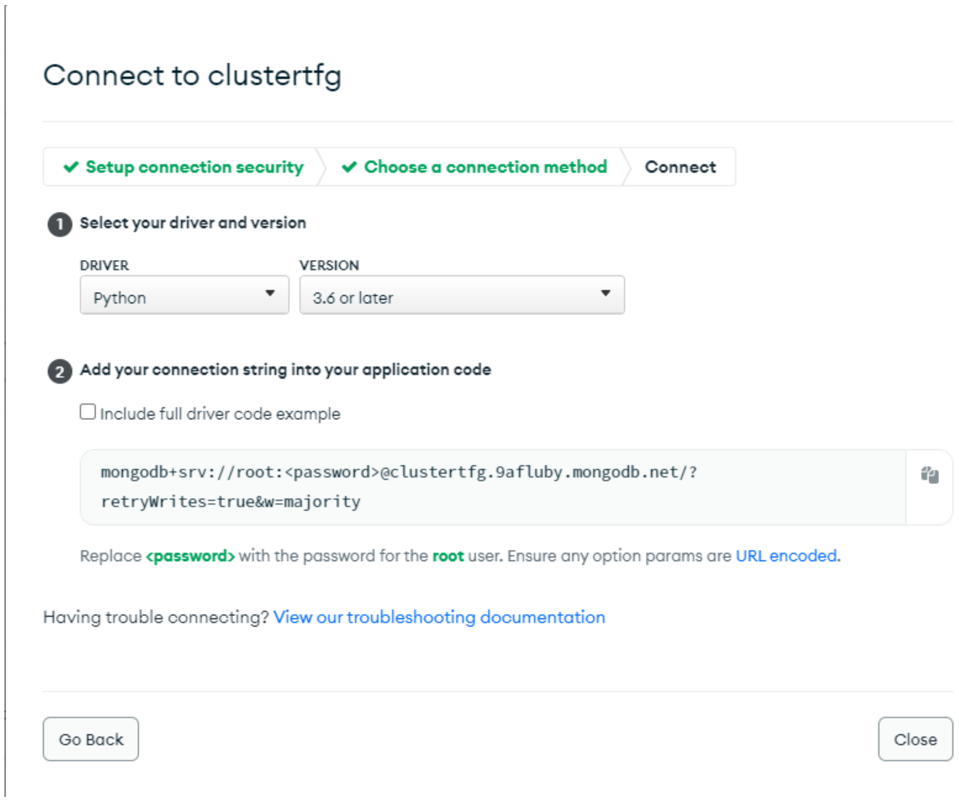
13.2 Database

This project uses MongoDB as the main database to store the information. To be able to use MongoDB, there is no need to install MongoDB, instead just use the web version following this guide: `https://www.MongoDB.com/docs/atlas/getting-started/`. Once the database is created, we click on the connect button (see Figure 54), then it will generate an URL (see Figure 55).



VERSION	REGION	CLUSTER TIER	TYPE	BACKUPS
5.0.8	AWS / Paris (eu-west-3)	M0 Sandbox (General)	Replica Set - 3 nodes	Inactive

Figure 54: Illustration of how to connect to the MongoDB database. Source: Own



Connect to clustertfg

✔ Setup connection security ✔ Choose a connection method Connect

1 Select your driver and version

DRIVER: Python

VERSION: 3.6 or later

2 Add your connection string into your application code

☐ Include full driver code example

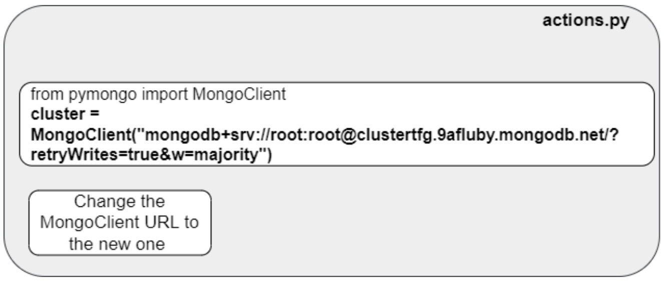
```
mongodb+srv://root:<password>@clustertfg.9afluby.mongodb.net/?
retryWrites=true&w=majority
```

Replace **<password>** with the password for the **root** user. Ensure any option params are URL encoded.

Having trouble connecting? View our troubleshooting documentation

Figure 55: Illustration of how to generate the connection URL. Source: Own

Now, we have to copy the generated URL and go to the Rasa project and find `project/actions/actions.py` file, then change the remote database's url `cluster=MongoClient("mongoDB+srv://root:root@clustertfg.9afluby.MongoDB.net/?retryWrites=true&w=majority")` to the new one (see Figure 56).



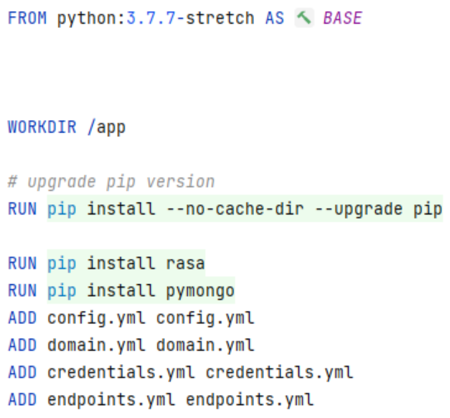
actions.py

```
from pymongo import MongoClient
cluster =
MongoClient("mongodb+srv://root:root@clustertfg.9afluby.mongodb.net/?
retryWrites=true&w=majority")
```

Figure 56: Illustration of how to change the connection URL in `actions.py`. Source: Own

13.3 Deployment

This project uses Okteto to deploy the project, in order to deploy it with other platforms just change the URL specified in `Controller.cs` script to the new one. If we want to use Okteto, then follow the following guide to create Dockerfile, `docker-compose.yml` file and the `okteto.yml` file. Everything can be found in the Okteto Documentation: `https://www.okteto.com/docs/welcome/overview/`. First, we need to create a Docker file which contains the build instructions for the Rasa server and the Rasa action server image.



```
FROM python:3.7.7-stretch AS BASE


WORKDIR /app

# upgrade pip version
RUN pip install --no-cache-dir --upgrade pip

RUN pip install rasa
RUN pip install pymongo
ADD config.yml config.yml
ADD domain.yml domain.yml
ADD credentials.yml credentials.yml
ADD endpoints.yml endpoints.yml
```

Figure 57: Docker file of the two Rasa servers. Source: Own

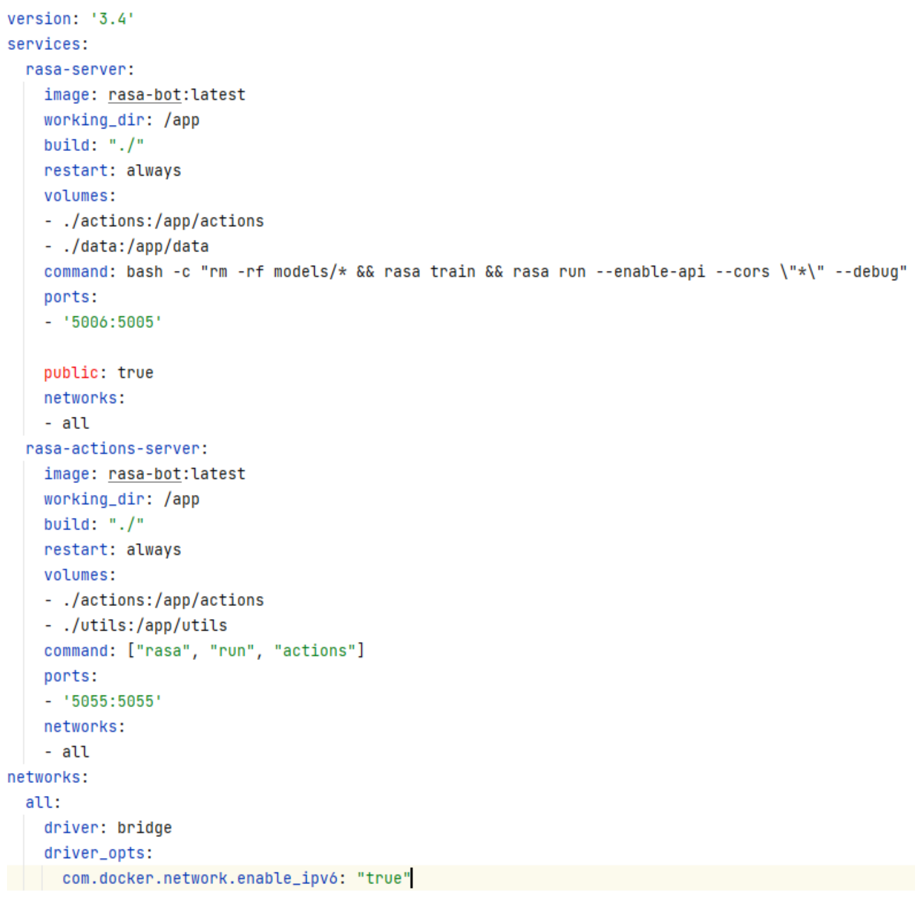
As we can see in Figure 57, first we indicate the version of Python as a base image, next we add a working directory where action (folder), data (folder), and other files which are needed for abort, will be stored. Then, we install the dependencies which are compulsory needed for our agent to work properly. Finally, we use *ADD* command to copy files from the local storage into the Docker image.

Now, we need to write the docker-compose manifest to build each service listed in it during the deployment process. In the docker-compose file (see Figure 58), we define two services to be built and deployed. The first one, *rasa-server*, corresponds to the Rasa server, and *rasa-actions-server* corresponds to the Rasa Action Server. Both services have the same working directory as specified in the Docker file, and with the same restart option set to *always*, indicating that in case the container fails, it will restart until its removal. For the *rasa-server*, we add all the data that is present in actions directory and data directory. For the *rasa-actions-server*, we only add the data that is present in actions directory. Next, we specify the command which will be run for the *rasa-server* service:

- *bash -c "rm -rf models/**: This command removes any modules that is present inside the models folder.
- *rasa train*: With this command, we train the model.
- *rasa run –enable-api –cors \"*\" –debug*: With this command, we run the Rasa server

Finally, we specify the command which will be run for the *rasa-actions-server* service:

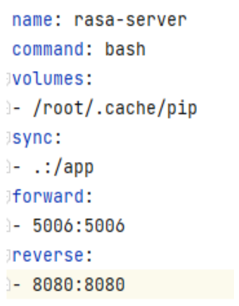
- *rasa run actions*: With this command, we run the Rasa action server.



```yaml
version: '3.4'
services:
  rasa-server:
    image: rasa-bot:latest
    working_dir: /app
    build: "./"
    restart: always
    volumes:
    - ./actions:/app/actions
    - ./data:/app/data
    command: bash -c "rm -rf models/* && rasa train && rasa run --enable-api --cors \"*\" --debug"
    ports:
    - '5006:5005'

    public: true
    networks:
    - all
  rasa-actions-server:
    image: rasa-bot:latest
    working_dir: /app
    build: "./"
    restart: always
    volumes:
    - ./actions:/app/actions
    - ./utils:/app/utils
    command: ["rasa", "run", "actions"]
    ports:
    - '5055:5055'
    networks:
    - all
networks:
  all:
    driver: bridge
    driver_opts:
      com.docker.network.enable_ipv6: "true"
```

Figure 58: Docker-compose file of the servers. Source: Own

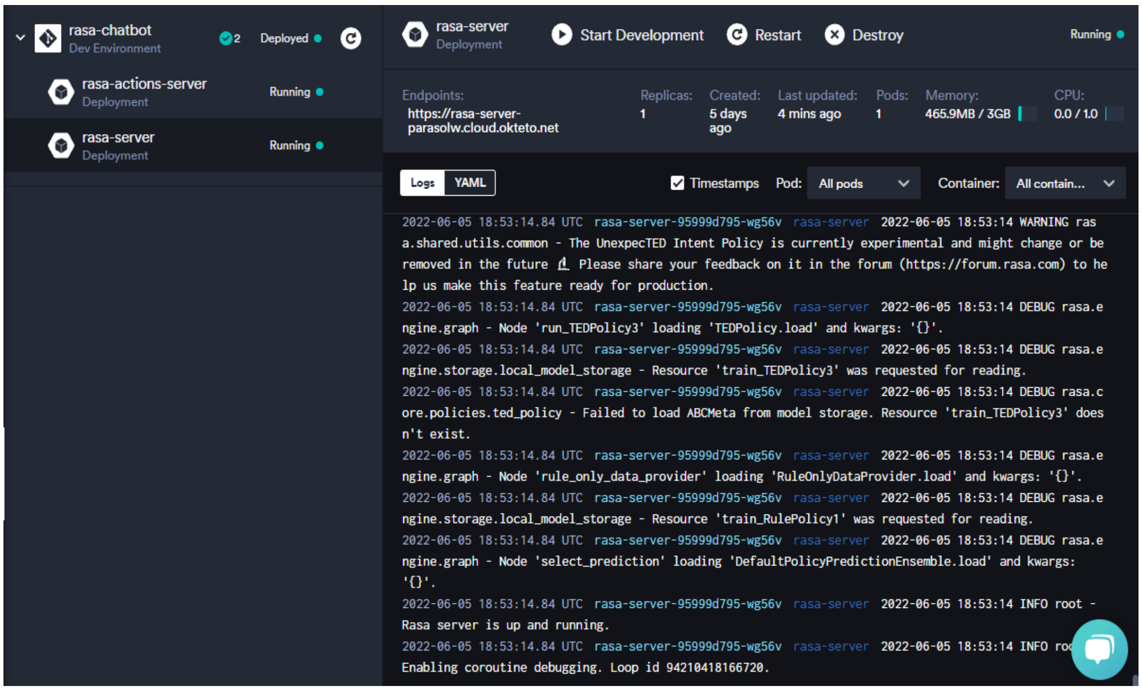
Since we have two services running as specified in the docker-compose file, we need to write an `okteto.yml` file to indicate of which service we want to keep an external connection. For this project, we keep an external connection for the Rasa server (see Figure 59).



```yaml
name: rasa-server
command: bash
volumes:
- /root/.cache/pip
sync:
- .:/app
forward:
- 5006:5006
reverse:
- 8080:8080
```

Figure 59: `okteto.yml` file that configure the external connection. Source: Own

Finally, we use the Okteto Cloud to deploy the Rasa server along with the Rasa action server (see Figure 60).



Endpoints: https://rasa-server-parasolw.cloud.okteto.net
Replicas: 1
Created: 5 days ago
Last updated: 4 mins ago
Pods: 1
Memory: 465.9MB / 3GB
CPU: 0.0 / 1.0

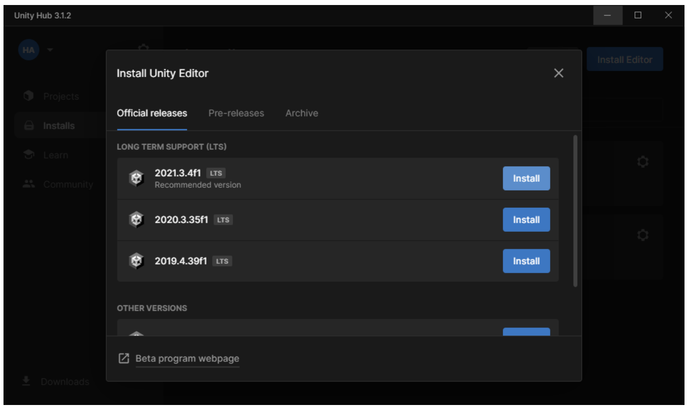
```
2022-06-05 18:53:14.84 UTC rasa-server-95999d795-wg56v rasa-server 2022-06-05 18:53:14 WARNING ras
a.shared.utils.common - The UnexpecTED Intent Policy is currently experimental and might change or be
removed in the future 🔬 Please share your feedback on it in the forum (https://forum.rasa.com) to he
lp us make this feature ready for production.
2022-06-05 18:53:14.84 UTC rasa-server-95999d795-wg56v rasa-server 2022-06-05 18:53:14 DEBUG rasa.e
ngine.graph - Node 'run_TEDPolicy3' loading 'TEDPolicy.load' and kwargs: '{}'.
2022-06-05 18:53:14.84 UTC rasa-server-95999d795-wg56v rasa-server 2022-06-05 18:53:14 DEBUG rasa.e
ngine.storage.local_model_storage - Resource 'train_TEDPolicy3' was requested for reading.
2022-06-05 18:53:14.84 UTC rasa-server-95999d795-wg56v rasa-server 2022-06-05 18:53:14 DEBUG rasa.c
ore.policies.ted_policy - Failed to load ABCMeta from model storage. Resource 'train_TEDPolicy3' does
n't exist.
2022-06-05 18:53:14.84 UTC rasa-server-95999d795-wg56v rasa-server 2022-06-05 18:53:14 DEBUG rasa.e
ngine.graph - Node 'rule_only_data_provider' loading 'RuleOnlyDataProvider.load' and kwargs: '{}'.
2022-06-05 18:53:14.84 UTC rasa-server-95999d795-wg56v rasa-server 2022-06-05 18:53:14 DEBUG rasa.e
ngine.storage.local_model_storage - Resource 'train_RulePolicy1' was requested for reading.
2022-06-05 18:53:14.84 UTC rasa-server-95999d795-wg56v rasa-server 2022-06-05 18:53:14 DEBUG rasa.e
ngine.graph - Node 'select_prediction' loading 'DefaultPolicyPredictionEnsemble.load' and kwargs:
'{}'.
2022-06-05 18:53:14.84 UTC rasa-server-95999d795-wg56v rasa-server 2022-06-05 18:53:14 INFO root -
Rasa server is up and running.
2022-06-05 18:53:14.84 UTC rasa-server-95999d795-wg56v rasa-server 2022-06-05 18:53:14 INFO roo
Enabling coroutine debugging. Loop id 94210418166720.
```

Figure 60: Illustration of Rasa servers running on the Okteto Cloud. Source: Own

13.4 Unity

- Download and follow the instruction to install the Unity Hub.
  - Open the Unity Hub and install the Unity Editor of 2021.3.1f1 version. Make sure that the

Android Build Support option is also installed (see Figure 61).


Unity Hub 3.1.2

Install Unity Editor

Official releases | Pre-releases | Archive

LONG TERM SUPPORT (LTS)

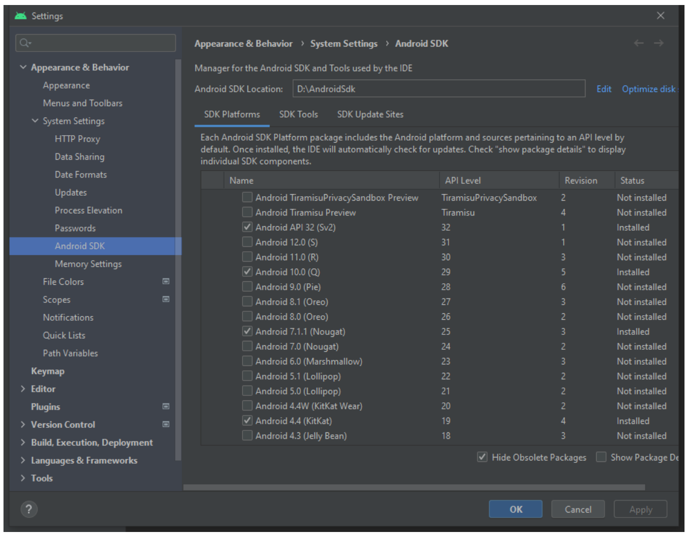
2021.3.4f1 LTS Recommended version	Install
2020.3.35f1 LTS	Install
2019.4.39f1 LTS	Install

OTHER VERSIONS

Beta program webpage

Figure 61: Installation of Android Build Support Module. Source: Own

- Download and follow the instruction to install Android Studio.
  - Open Android Studio and go to File - Settings - Appearance & Behavior - System Settings - Android SDK - SDK Platforms (see Figure 62).
  - Download and install:
    - ∗ Android 10.0(Q).
    - ∗ Android 7.1.1 (Nougat)
    - ∗ Android 4.4 (KitKat)


Settings

Appearance & Behavior › System Settings › Android SDK

Manager for the Android SDK and Tools used by the IDE

Android SDK Location: D:\AndroidSdk

SDK Platforms | SDK Tools | SDK Update Sites

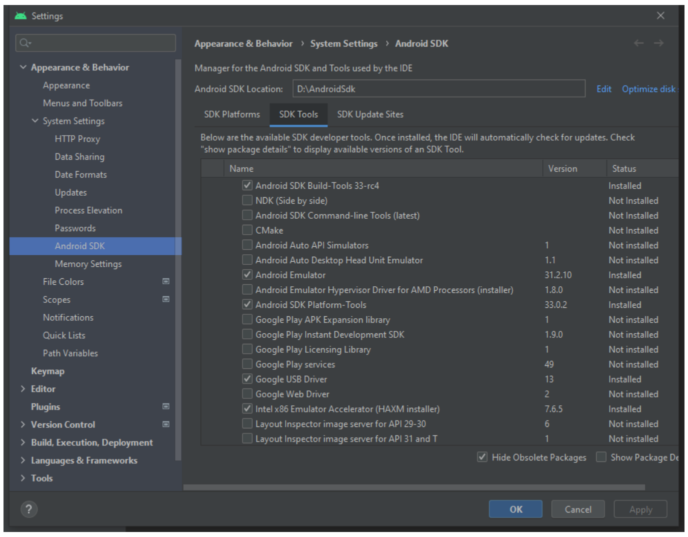
Each Android SDK Platform package includes the Android platform and sources pertaining to an API level by default. Once installed, the IDE will automatically check for updates. Check "show package details" to display individual SDK components.

	Name	API Level	Revision	Status
☐	Android TiramisuPrivacySandbox Preview	TiramisuPrivacySandbox	2	Not installed
☐	Android Tiramisu Preview	Tiramisu	4	Not installed
✓	Android API 32 (Sv2)	32	1	Installed
☐	Android 12.0 (S)	31	1	Not installed
☐	Android 11.0 (R)	30	3	Not installed
✓	Android 10.0 (Q)	29	5	Installed
☐	Android 9.0 (Pie)	28	6	Not installed
☐	Android 8.1 (Oreo)	27	3	Not installed
☐	Android 8.0 (Oreo)	26	2	Not installed
✓	Android 7.1.1 (Nougat)	25	3	Installed
☐	Android 7.0 (Nougat)	24	2	Not installed
☐	Android 6.0 (Marshmallow)	23	3	Not installed
☐	Android 5.1 (Lollipop)	22	2	Not installed
☐	Android 5.0 (Lollipop)	21	2	Not installed
☐	Android 4.4W (KitKat Wear)	20	2	Not installed
✓	Android 4.4 (KitKat)	19	4	Installed
☐	Android 4.3 (Jelly Bean)	18	3	Not installed

✓ Hide Obsolete Packages ☐ Show Package De

Figure 62: Installation of Android SDK Platform. Source: Own

  - Switch from SDK Platforms to SDK Tools and install (see Figure 63):
    - Android SDK Build-Tools 33-rc4.
    - Android SDK Platform-Tools.



Appearance & Behavior › System Settings › Android SDK

Manager for the Android SDK and Tools used by the IDE

Android SDK Location: D:\AndroidSdk

SDK Platforms | SDK Tools | SDK Update Sites

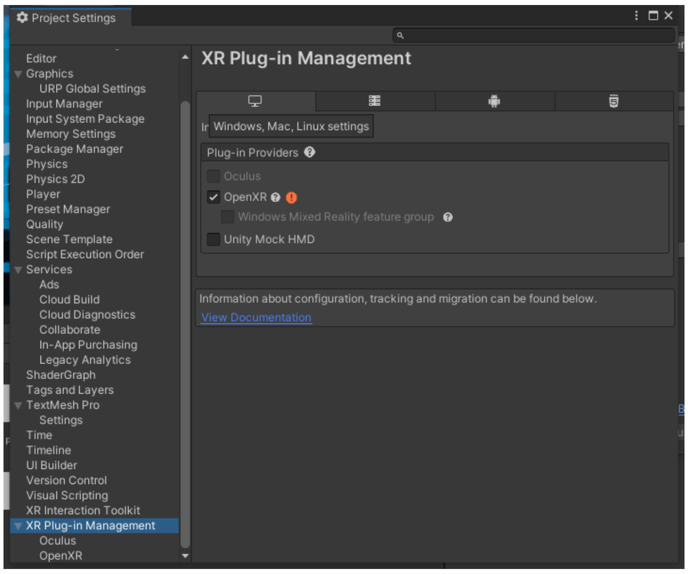
Below are the available SDK developer tools. Once installed, the IDE will automatically check for updates. Check "show package details" to display available versions of an SDK Tool.

	Name	Version	Status
✓	Android SDK Build-Tools 33-rc4		Installed
	NDK (Side by side)		Not Installed
	Android SDK Command-line Tools (latest)		Not Installed
	CMake		Not Installed
	Android Auto API Simulators	1	Not installed
	Android Auto Desktop Head Unit Emulator	1.1	Not installed
✓	Android Emulator	31.2.10	Installed
	Android Emulator Hypervisor Driver for AMD Processors (installer)	1.8.0	Not installed
✓	Android SDK Platform-Tools	33.0.2	Installed
	Google Play APK Expansion library	1	Not installed
	Google Play Instant Development SDK	1.9.0	Not installed
	Google Play Licensing Library	1	Not installed
	Google Play services	49	Not installed
✓	Google USB Driver	13	Installed
	Google Web Driver	2	Not installed
✓	Intel x86 Emulator Accelerator (HAXM installer)	7.6.5	Installed
	Layout Inspector image server for API 29-30	6	Not installed
	Layout Inspector image server for API 31 and T	1	Not installed

✓ Hide Obsolete Packages

Figure 63: Installation of Android SDK Tools. Source: Own

- Download and follow the instructions to install Oculus. Then, follow this guide to setup the device and enable the developer mode: `https://developer.oculus.com/documentation/native/android/mobile-device-setup/`
- In Unity, download all the packages that is needed.
  - Go to Edit -> Project Setting.
    - Find XR Plug-in Management at the bottom, then install it.
    - Inside the XR Plug-in Management, select the OpenXR option for Windows, Mac, Linux settings.



XR Plug-in Management

Windows, Mac, Linux settings

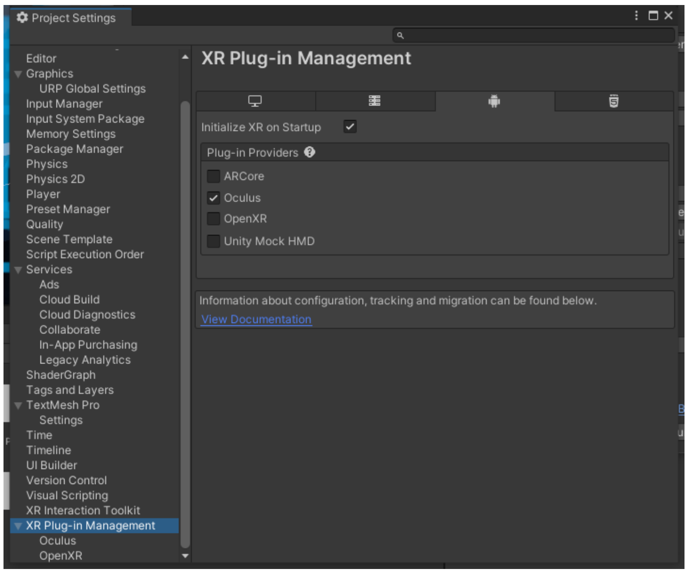
Plug-in Providers

- [ ] Oculus
- [x] OpenXR
  - [ ] Windows Mixed Reality feature group
- [ ] Unity Mock HMD

Information about configuration, tracking and migration can be found below.
View Documentation

Figure 64: XR Plug-in Management Windows, Mac, Linux settings. Source: Own

* Then, select the Oculus option for Android settings.



XR Plug-in Management

Initialize XR on Startup ✓

Plug-in Providers

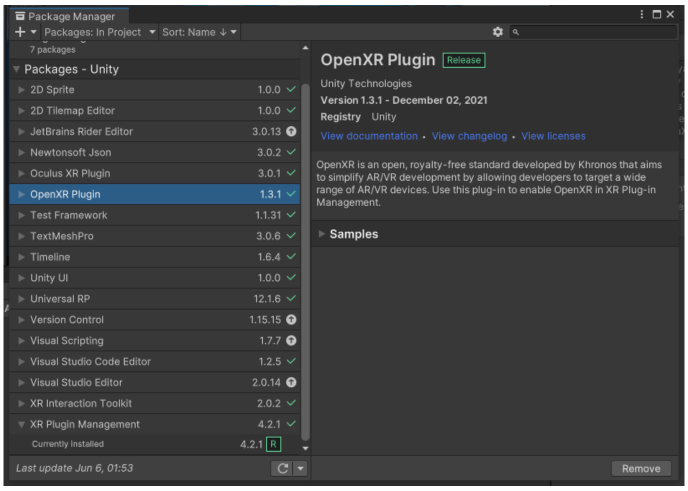
- [ ] ARCore
- [x] Oculus
- [ ] OpenXR
- [ ] Unity Mock HMD

Information about configuration, tracking and migration can be found below.
View Documentation

Figure 65: XR Plug-in Management Android settings. Source: Own

- Go to Window -> Package Manager, then download and install:
  * TextMeshPro.
  * XR Interaction Toolkit.

      * OpenXR Plugin.



7 packages

Packages - Unity

2D Sprite	1.0.0 ✓
2D Tilemap Editor	1.0.0 ✓
JetBrains Rider Editor	3.0.13
Newtonsoft Json	3.0.2 ✓
Oculus XR Plugin	3.0.1 ✓
OpenXR Plugin	1.3.1 ✓
Test Framework	1.1.31 ✓
TextMeshPro	3.0.6 ✓
Timeline	1.6.4 ✓
Unity UI	1.0.0 ✓
Universal RP	12.1.6 ✓
Version Control	1.15.15
Visual Scripting	1.7.7
Visual Studio Code Editor	1.2.5 ✓
Visual Studio Editor	2.0.14
XR Interaction Toolkit	2.0.2 ✓
XR Plugin Management	4.2.1 ✓
Currently installed	4.2.1 R

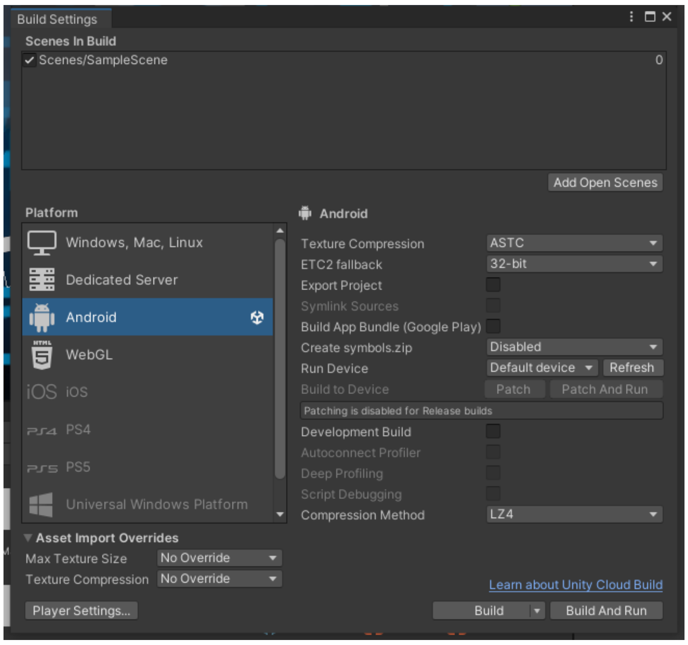
Last update Jun 6, 01:53

OpenXR Plugin Release
Unity Technologies
Version 1.3.1 - December 02, 2021
Registry Unity
View documentation • View changelog • View licenses

OpenXR is an open, royalty-free standard developed by Khronos that aims to simplify AR/VR development by allowing developers to target a wide range of AR/VR devices. Use this plug-in to enable OpenXR in XR Plug-in Management.

Figure 66: Package Manager. Source: Own

- In Unity, configure the build settings ().
  - Go to Edit -> Build Setting.
  - Switch the platform from Windows to Android.
  - Change the texture compression to ASTC.



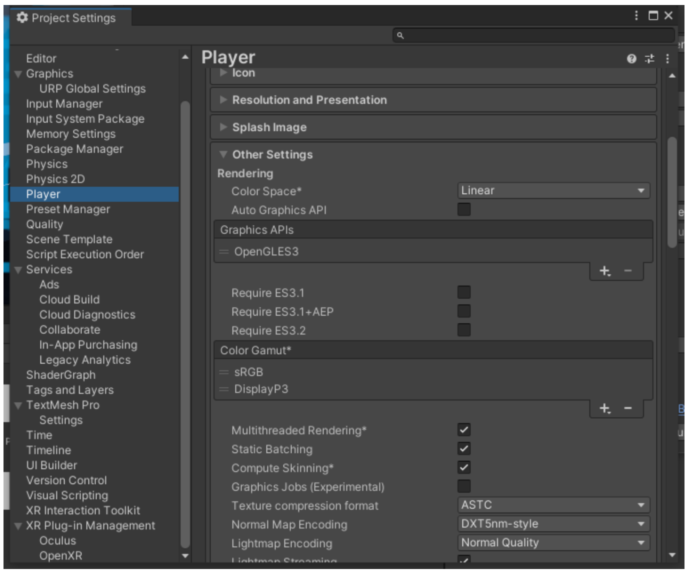
✓ Scenes/SampleScene 0

Texture Compression: ASTC
ETC2 fallback: 32-bit
Export Project
Symlink Sources
Build App Bundle (Google Play)
Create symbols.zip: Disabled
Run Device: Default device
Build to Device
Patching is disabled for Release builds
Development Build
Autoconnect Profiler
Deep Profiling
Script Debugging
Compression Method: LZ4

Asset Import Overrides
Max Texture Size: No Override
Texture Compression: No Override

Figure 67: Build Settings. Source: Own

– Go to Player Settings.
– Change the company name and the product name.
– In Player settings, go to Android settings -> Other Settings.
– Unchecked Auto Graphics API, and delete Vulcan.



Project Settings

Editor
Graphics
URP Global Settings
Input Manager
Input System Package
Memory Settings
Package Manager
Physics
Physics 2D
Player
Preset Manager
Quality
Scene Template
Script Execution Order
Services
Ads
Cloud Build
Cloud Diagnostics
Collaborate
In-App Purchasing
Legacy Analytics
ShaderGraph
Tags and Layers
TextMesh Pro
Settings
Time
Timeline
UI Builder
Version Control
Visual Scripting
XR Interaction Toolkit
XR Plug-in Management
Oculus
OpenXR

Player

Icon
Resolution and Presentation
Splash Image
Other Settings
Rendering
Color Space* — Linear
Auto Graphics API — ☐
Graphics APIs
OpenGLES3
Require ES3.1 — ☐
Require ES3.1+AEP — ☐
Require ES3.2 — ☐
Color Gamut*
sRGB
DisplayP3
Multithreaded Rendering* — ✓
Static Batching — ✓
Compute Skinning* — ✓
Graphics Jobs (Experimental) — ☐
Texture compression format — ASTC
Normal Map Encoding — DXT5nm-style
Lightmap Encoding — Normal Quality

Figure 68: Player Settings. Source: Own

- Now, the project is ready to build and run in VR environment.
  - Turn the Oculus Quest on.
  - Connect the Oculus Quest to the PC via USB.
  - In the Oculus app, select Devices and add the headset (Skip if already done).
  - Follow the instruction to setup everything (Skip if already done).
  - In Unity, go to File -> Build Setting.
  - Click on Build and Run.
  - Store the APK file in a local directory.
  - Now, the game should be run in Oculus Quest.

14 Appendix IV - User guide

In general, the game scene is quite easy to understand. We have three buttons: the increase button to change the game-stage, the record button to perform speech to text feature and the send button to send the message to the Rasa server. The only thing that we have to keep in mind is that the Rasa agent will show up at the following game-stages:

Game-stage	**Agent behaviour**
Score == 2	The Rasa agent shows up and introduce himself.
Score == 6	The Rasa agent shows up to restore the conversation and ask the question that we didn't answer before.
Score == 10	The Rasa agent shows up to restore the conversation and ask the question that we didn't answer before.